第一章：Activity深度解析

（一）Activity启动流程详解

前言

Activity作为Android四大组件之一，他的启动绝对没有那么简单。这里涉及到了系统服务进程，启动过程细节很多，这里我只展示主体流程。activity的启动流程随着版本的更替，代码细节一直在进行更改，每次都会有很大的修改，如android5.0 android8.0。我这里的版本是基于android api28，也是目前我可以查得到的最新源码了。事实上大题的流程是相同的，掌握了一个版本，其他的版本通过源码也可以很快地掌握。

因为涉及到不同的进程之间的通信：系统服务进程和本地进程，在最新版本的android使用的是AIDL来跨进程通信。所以需要对AIDL有一定的了解，会帮助理解整个启动流程。

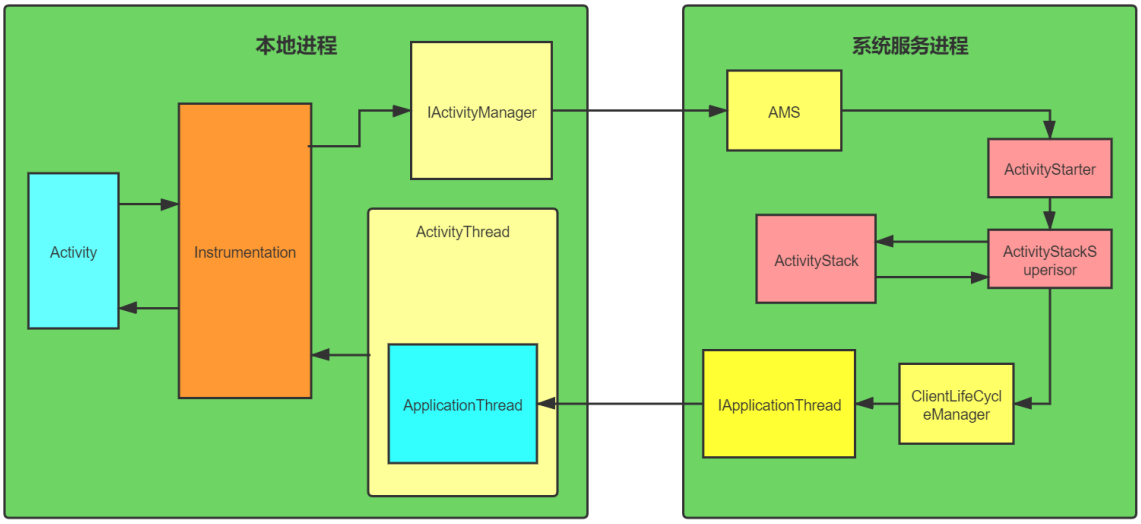
> 源码部分的讲解涉及到很多的代码讲解，可能会有一点不适，但还是建议看完源码。源码的关键代码处我都会加上注释，方便理解。
>
> 代码不会过分关注细节，只注重整体流程。想知道具体细节可以去查看源码。每份代码所在的路径我都会在代码前面标注出来，各位可以去查看相对应的源码。
>
> 每部分源码前我都会放流程图，一定要配合流程图食用，不然可能会乱。

整体流程概述

这一部分侧重于对整个启动流程的概述，在心中有大体的概念，这样可以帮助对下面具体细节流程的理解。

普通Activity的创建

普通Activity创建也就是平常我们在代码中采用`startActivity(Intent intent)`方法来创建Activity的方式。总体流程如下图：



启动过程设计到两个进程：本地进程和系统服务进程。本地进程也就是我们的应用所在进程，系统服务进程为所有应用共用的服务进程。整体思路是：

1. activity向Instrumentation请求创建
2. Instrumentation通过AMS在本地进程的IBinder接口，访问AMS，这里采用的跨进程技术是AIDL。

3. 然后AMS进程一系列的工作，如判断该activity是否存在，启动模式是什么，有没有进行注册等等。
4. 通过ClientLifeCycleManager，利用本地进程在系统服务进程的IBinder接口直接访问本地ActivityThread。

   > ApplicationThread是ActivityThread的内部类，IApplicationThread是在远程服务端的Binder接口

5. ApplicationThread接收到服务端的事务后，把事务直接转交给ActivityThread处理。
6. ActivityThread通过Instrumentation利用类加载器进行创建实例，同时利用Instrumentation回调activity的生命中周期

这里涉及到了两个进程，本地进程主要负责创建activity以及回调生命周期，服务进程主要判断该activity是否合法，是否需要创建activity栈等等。进程之间就涉及到了进程通信：AIDL。（如果不熟悉可以先去了解一下，但可以简单理解为接口回调即可）

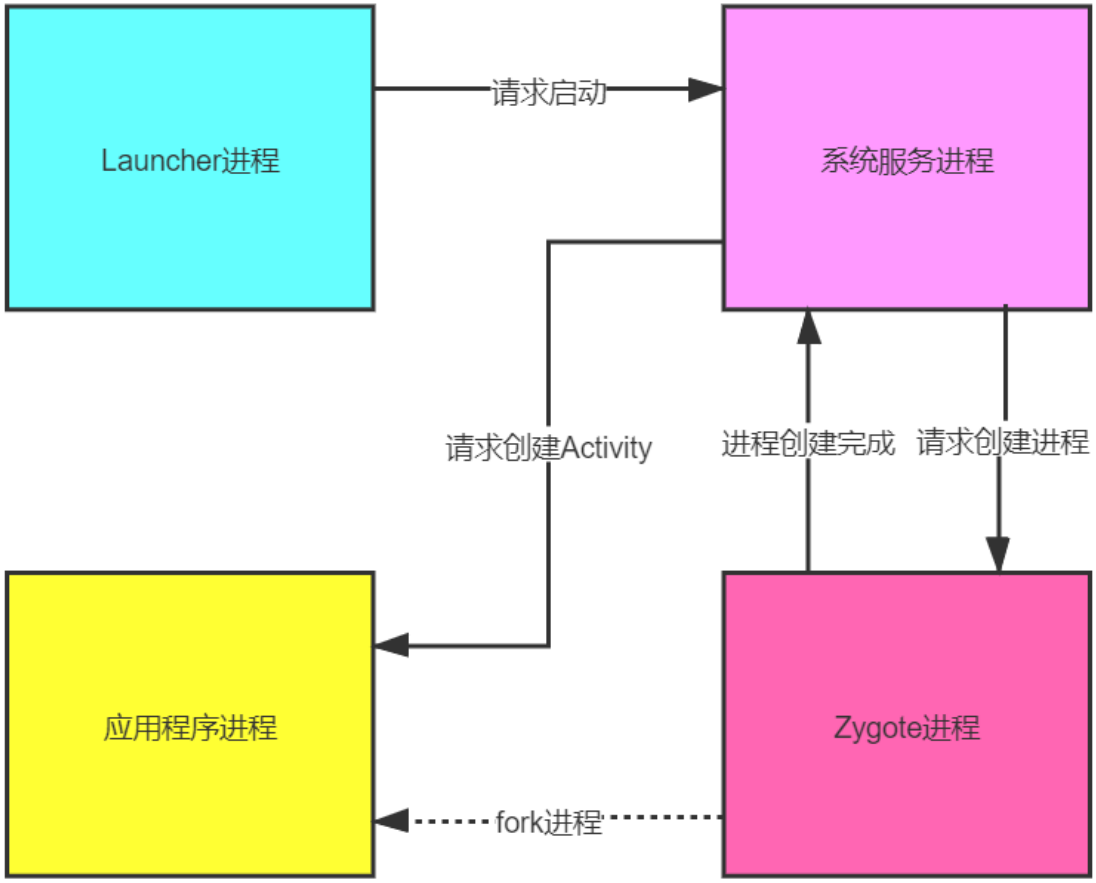
下面介绍几个关键类：

- Instrumentation是activity与外界联系的类（不是activity本身的统称外界，相对activity而言），activity通过Instrumentation来请求创建，ActivityThread通过Instrumentation来创建activity和调用activity的生命周期。
- ActivityThread，每个应用程序唯一一个实例，负责对Activity创建的管理，而ApplicationThread只是应用程序和服务端进程通信的类而已，只负责通信，把AMS的任务交给ActivityThread。
- AMS，全称ActivityManagerService，负责统筹服务端对activity创建的流程。

  其他的类，后面的源码解析会详解。

根Activity的创建

根Activity也就是我们点击桌面图标的时候，应用程序第一个activity启动的流程。这里我侧重讲解多个进程之间的关系，下面的源码也不会讲细节，只讲解普通activity的创建流程。这里也相当于一个补充。先看整体流程图：



主要涉及四个进程：

- Launcher进程，也就是桌面进程
- 系统服务进程，AMS所在进程
- Zygote进程，负责创建进程
- 应用程序进程，也就是即将要启动的进程

主要流程：

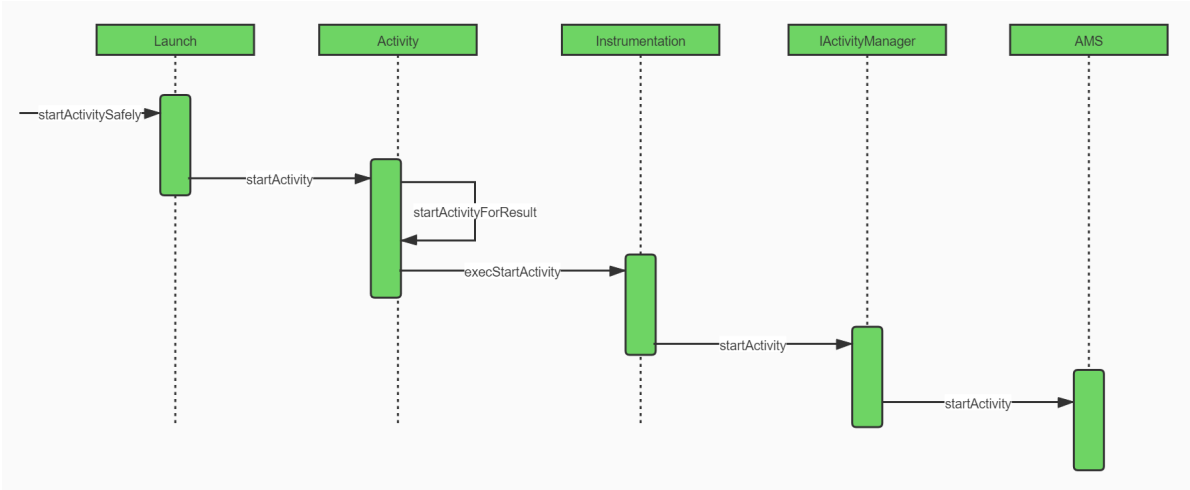
1. Launcher进程请求AMS创建activity
2. AMS请求Zygote创建进程。
3. Zygote通过fork自己来创建进程。并通知AMS创建完成。
4. AMS通知应用进程创建根Activity。

和普通Activity的创建很像，主要多了创建进程这一步。

源码讲解

Activity请求AMS的过程

流程图



源码

1. 系统通过调用Launcher的startActivitySafely方法来启动应用程序。Launcher是一个类，负责启动根Activity。

   > 这一步是根Activity启动才有的流程，普通启动是没有的，放在这里是作为一点补充而已

   复制代码

```
packages/apps/Launcher3/src/com/android/launcher3/Launcher.java/;
public boolean startActivitySafely(View v, Intent intent, ItemInfo item) {
        //这里调用了父类的方法，继续查看父类的方法实现
        boolean success = super.startActivitySafely(v, intent, item);
        ...
        return success;
    }
```

   复制代码

```
packages/apps/Launcher3/src/com/android/launcher3/BaseDraggingActivity.java/
;
public boolean startActivitySafely(View v, Intent intent, ItemInfo item) {
        ...
        // Prepare intent
        //设置标志singleTask，意味着在新的栈打开
        intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        if (v != null) {
            intent.setSourceBounds(getViewBounds(v));
        }
        try {
            boolean isShortcut = Utilities.ATLEAST_MARSHMALLOW
                    && (item instanceof ShortcutInfo)
                    && (item.itemType == Favorites.ITEM_TYPE_SHORTCUT
                    || item.itemType == Favorites.ITEM_TYPE_DEEP_SHORTCUT)
                    && !((ShortcutInfo) item).isPromise();
            //下面注释1和注释2都是直接采用startActivity进行启动。注释1会做一些设置
            //BaseDraggingActivity是继承自BaseActivity，而BaseActivity是继承自
Activity
            //所以直接就跳转到了Activity的startActivity逻辑。
            if (isShortcut) {
                // Shortcuts need some special checks due to legacy reasons.
                startShortcutIntentSafely(intent, optsBundle, item);//1
```

```
            } else if (user == null || user.equals(Process.myUserHandle()))
{
                // Could be launching some bookkeeping activity
                startActivity(intent, optsBundle);//2
            } else {

 LauncherAppsCompat.getInstance(this).startActivityForProfile(
                        intent.getComponent(), user,
intent.getSourceBounds(), optsBundle);
            }

           ...
        }
        ...
        return false;
    }
```

2. Activity通过Instrumentation来启动Activity

   复制代码

```
/frameworks/base/core/java/android/app/Activity.java/;
public void startActivity(Intent intent, @Nullable Bundle options) {
        //最终都会跳转到startActivityForResult这个方法
        if (options != null) {
            startActivityForResult(intent, -1, options);
        } else {
            // Note we want to go through this call for compatibility with
            // applications that may have overridden the method.
            startActivityForResult(intent, -1);
        }
    }

public void startActivityForResult(@RequiresPermission Intent intent, int
requestCode,
            @Nullable Bundle options) {
        //mParent是指activityGroup，现在已经采用Fragment代替，这里会一直是null
        //下一步会通过mInstrumentation.execStartActivity进行启动
        if (mParent == null) {
            options = transferSpringboardActivityOptions(options);
            Instrumentation.ActivityResult ar =
                mInstrumentation.execStartActivity(
                    this, mMainThread.getApplicationThread(), mToken, this,
                    intent, requestCode, options);//1
            if (ar != null) {
                mMainThread.sendActivityResult(
                    mToken, mEmbeddedID, requestCode, ar.getResultCode(),
                    ar.getResultData());
            }
            ...
        }
    ...
}
```

3. Instrumentation请求AMS进行启动。该类的作用是监控应用程序和系统的交互。到此为止，任务就交给了AMS了，AMS进行一系列处理后，会通过本地的接口IActivityManager来进行回调启动activity。

复制代码

```
/frameworks/base/core/java/android/app/Instrumentation.java/;
public ActivityResult execStartActivity(
            Context who, IBinder contextThread, IBinder token, Activity
target,
            Intent intent, int requestCode, Bundle options) {
    ...
    //这个地方比较复杂，先说结论。下面再进行解释
    //ActivityManager.getService()获取到的对象是ActivityManagerService，简称AMS
    //通过AMS来启动activity。AMS是全局唯一的，所有的活动启动都要经过他的验证，运行在独立
的进程中
    //所以这里是采用AIDL的方式进行跨进程通信，获取到的对象其实是一个IBinder接口

    //注释2是进行检查启动结果，如果异常则抛出，如没有注册。
    try {
            intent.migrateExtraStreamToClipData();
            intent.prepareToLeaveProcess(who);
            int result = ActivityManager.getService()
                .startActivity(whoThread, who.getBasePackageName(), intent,

 intent.resolveTypeIfNeeded(who.getContentResolver()),
                        token, target != null ? target.mEmbeddedID : null,
                        requestCode, 0, null, options);//1
            checkStartActivityResult(result, intent);//2
        } catch (RemoteException e) {
            throw new RuntimeException("Failure from system", e);
        }
        return null;

}
```

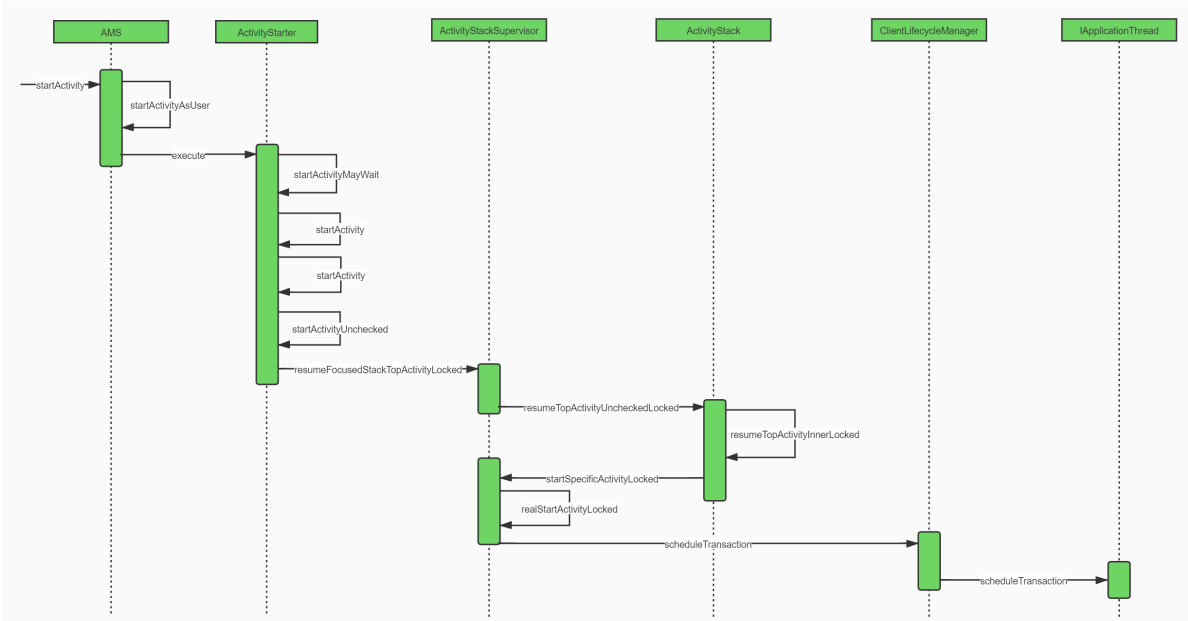
这一步是通过AIDL技术进行跨进行通信。拿到AMS的代理对象，把启动任务交给了AMS。

复制代码

```
/frameworks/base/core/java/android/app/ActivityManager.java/;
//单例类
public static IActivityManager getService() {
return IActivityManagerSingleton.get();
}

private static final Singleton<IActivityManager>
IActivityManagerSingleton =
new Singleton<IActivityManager>() {
@Override
protected IActivityManager create() {
//得到AMS的IBinder接口
final IBinder b = ServiceManager.getService(Context.ACTIVITY_SERVICE);
//转化成IActivityManager对象。远程服务实现了这个接口，所以可以直接调用这个
//AMS代理对象的接口方法来请求AMS。这里采用的技术是AIDL
final IActivityManager am = IActivityManager.Stub.asInterface(b);
return am;
}
};
```

AMS处理请求的过程

流程图



源码

1. 接下来看AMS的实现逻辑。AMS这部分的源码是通过ActivityStartController来创建一个ActivityStarter，然后把逻辑都交给ActivityStarter去执行。ActivityStarter是android 7.0加入的类。

   复制代码

```
 /frameworks/base/services/core/java/com/android/server/am/ActivityManagerSe
rvice.java/;
//跳转到startActivityAsUser
//注意最后多了一个参数UserHandle.getCallingUserId()，表示调用者权限
public final int startActivity(IApplicationThread caller, String
callingPackage,
        Intent intent, String resolvedType, IBinder resultTo, String
resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle bOptions) {
    return startActivityAsUser(caller, callingPackage, intent, resolvedType,
resultTo,
            resultWho, requestCode, startFlags, profilerInfo, bOptions,
            UserHandle.getCallingUserId());
}

public final int startActivityAsUser(IApplicationThread caller, String
callingPackage,
            Intent intent, String resolvedType, IBinder resultTo, String
resultWho, int requestCode,
            int startFlags, ProfilerInfo profilerInfo, Bundle bOptions, int
userId,
            boolean validateIncomingUser) {
        enforceNotIsolatedCaller("startActivity");

        userId = mActivityStartController.checkTargetUser(userId,
validateIncomingUser,
                Binder.getCallingPid(), Binder.getCallingUid(),
"startActivityAsUser");
```

```
        // TODO: Switch to user app stacks here.
        //这里通过ActivityStartController获取到ActivityStarter，通过
ActivityStarter来
        //执行启动任务。这里就把任务逻辑给到了AcitivityStarter
        return mActivityStartController.obtainStarter(intent,
"startActivityAsUser")
                .setCaller(caller)
                .setCallingPackage(callingPackage)
                .setResolvedType(resolvedType)
                .setResultTo(resultTo)
                .setResultWho(resultWho)
                .setRequestCode(requestCode)
                .setStartFlags(startFlags)
                .setProfilerInfo(profilerInfo)
                .setActivityOptions(bOptions)
                .setMayWait(userId)
                .execute();

    }
```

> ActivityStartController获取ActivityStarter
>
> 复制代码
>
> ```
> /frameworks/base/services/core/java/com/android/server/am/ActivityStart
> Controller.java/;
> //获取到ActivityStarter对象。这个对象仅使用一次，当他的execute被执行后，该对象作
> 废
> ActivityStarter obtainStarter(Intent intent, String reason) {
> return mFactory.obtain().setIntent(intent).setReason(reason);
> }
> ```

2. 这部分主要是ActivityStarter的源码内容，涉及到的源码非常多。AMS把整个启动逻辑都丢给ActivityStarter去处理了。这里主要做启动前处理，创建进程等等。

   复制代码

```
/frameworks/base/services/core/java/com/android/server/am/ActivityStarter.ja
va/;
//这里需要做启动预处理，执行startActivityMayWait方法
int execute() {
        try {
            ...
            if (mRequest.mayWait) {
                return startActivityMayWait(mRequest.caller,
mRequest.callingUid,
                        mRequest.callingPackage, mRequest.intent,
mRequest.resolvedType,
                        mRequest.voiceSession, mRequest.voiceInteractor,
mRequest.resultTo,
                        mRequest.resultWho, mRequest.requestCode,
mRequest.startFlags,
                        mRequest.profilerInfo, mRequest.waitResult,
mRequest.globalConfig,
                        mRequest.activityOptions,
mRequest.ignoreTargetSecurity, mRequest.userId,
```

```
                        mRequest.inTask, mRequest.reason,
                        mRequest.allowPendingRemoteAnimationRegistryLookup);
            }
            ...
        }
        ...
    }

//启动预处理
private int startActivityMayWait(IApplicationThread caller, int callingUid,
            String callingPackage, Intent intent, String resolvedType,
            IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
            IBinder resultTo, String resultWho, int requestCode, int
startFlags,
            ProfilerInfo profilerInfo, WaitResult outResult,
            Configuration globalConfig, SafeActivityOptions options, boolean
ignoreTargetSecurity,
            int userId, TaskRecord inTask, String reason,
            boolean allowPendingRemoteAnimationRegistryLookup) {
        ...
        //跳转startActivity
         final ActivityRecord[] outRecord = new ActivityRecord[1];
        int res = startActivity(caller, intent, ephemeralIntent,
resolvedType, aInfo, rInfo,
                voiceSession, voiceInteractor, resultTo, resultWho,
requestCode, callingPid,
                callingUid, callingPackage, realCallingPid, realCallingUid,
startFlags, options,
                ignoreTargetSecurity, componentSpecified, outRecord, inTask,
reason,
                allowPendingRemoteAnimationRegistryLookup);
}

//记录启动进程和activity的信息
private int startActivity(IApplicationThread caller, Intent intent, Intent
ephemeralIntent,
        String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
        IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
        IBinder resultTo, String resultWho, int requestCode, int callingPid,
int callingUid,
        String callingPackage, int realCallingPid, int realCallingUid, int
startFlags,
        SafeActivityOptions options,
        boolean ignoreTargetSecurity, boolean componentSpecified,
ActivityRecord[] outActivity,
        TaskRecord inTask, boolean
allowPendingRemoteAnimationRegistryLookup) {
    ...
    //得到Launcher进程
    ProcessRecord callerApp = null;
    if (caller != null) {
        callerApp = mService.getRecordForAppLocked(caller);
        ...
    }
    ...
    //记录得到的activity信息
```

```
    ActivityRecord r = new ActivityRecord(mService, callerApp, callingPid,
callingUid,
            callingPackage, intent, resolvedType, aInfo,
mService.getGlobalConfiguration(),
            resultRecord, resultWho, requestCode, componentSpecified,
voiceSession != null,
            mSupervisor, checkedOptions, sourceRecord);
    if (outActivity != null) {
        outActivity[0] = r;
    }
    ...
    mController.doPendingActivityLaunches(false);
    //继续跳转
    return startActivity(r, sourceRecord, voiceSession, voiceInteractor,
startFlags,
            true /* doResume */, checkedOptions, inTask, outActivity);
}

//跳转startActivityUnchecked
private int startActivity(final ActivityRecord r, ActivityRecord
sourceRecord,
            IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
            int startFlags, boolean doResume, ActivityOptions options,
TaskRecord inTask,
            ActivityRecord[] outActivity) {
    int result = START_CANCELED;
    try {
        mService.mWindowManager.deferSurfaceLayout();
        //跳转
        result = startActivityUnchecked(r, sourceRecord, voiceSession,
voiceInteractor,
                startFlags, doResume, options, inTask, outActivity);
    }
    ...
    return result;
}

//主要做与栈相关的逻辑处理，并跳转到ActivityStackSupervisor进行处理
private int startActivityUnchecked(final ActivityRecord r, ActivityRecord
sourceRecord,
        IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
        int startFlags, boolean doResume, ActivityOptions options,
TaskRecord inTask,
        ActivityRecord[] outActivity) {
    ...
    int result = START_SUCCESS;
    //这里和我们最初在Launcher设置的标志FLAG_ACTIVITY_NEW_TASK相关，会创建一个新栈
    if (mStartActivity.resultTo == null && mInTask == null && !mAddingToTask
            && (mLaunchFlags & FLAG_ACTIVITY_NEW_TASK) != 0) {
        newTask = true;
        result = setTaskFromReuseOrCreateNewTask(taskToAffiliate, topStack);
    }
    ...
    if (mDoResume) {
        final ActivityRecord topTaskActivity =
            mStartActivity.getTask().topRunningActivityLocked();
```

```
        if (!mTargetStack.isFocusable()
            || (topTaskActivity != null && topTaskActivity.mTaskOverlay
                && mStartActivity != topTaskActivity)) {
            ...
        } else {
            if (mTargetStack.isFocusable() &&
!mSupervisor.isFocusedStack(mTargetStack)) {
                mTargetStack.moveToFront("startActivityUnchecked");
            }
            //跳转到ActivityStackSupervisor进行处理
            mSupervisor.resumeFocusedStackTopActivityLocked(mTargetStack,
mStartActivity,
                                                            mOptions);
        }
    }
}
```

3. ActivityStackSupervisor主要负责做activity栈的相关工作，会结合ActivityStack来进行工作。主要判断activity的状态，是否处于栈顶或处于停止状态等

   复制代码

```
/frameworks/base/services/core/java/com/android/server/am/ActivityStackSuper
visor.java/;

boolean resumeFocusedStackTopActivityLocked(
        ActivityStack targetStack, ActivityRecord target, ActivityOptions
targetOptions) {
    ...
    //判断要启动的activity是不是出于停止状态或者Resume状态
    final ActivityRecord r = mFocusedStack.topRunningActivityLocked();
    if (r == null || !r.isState(RESUMED)) {
        mFocusedStack.resumeTopActivityUncheckedLocked(null, null);
    } else if (r.isState(RESUMED)) {
        // Kick off any lingering app transitions form the MoveTaskToFront
operation.
        mFocusedStack.executeAppTransition(targetOptions);
    }
    return false;
}
```

4. ActivityStack主要处理activity在栈中的状态

   复制代码

```
/frameworks/base/services/core/java/com/android/server/am/ActivityStack.java
/;
//跳转resumeTopActivityInnerLocked
boolean resumeTopActivityUncheckedLocked(ActivityRecord prev,
ActivityOptions options) {
    if (mStackSupervisor.inResumeTopActivity) {
        // Don't even start recursing.
        return false;
    }
    boolean result = false;
    try {
        // Protect against recursion.
```

```
        mStackSupervisor.inResumeTopActivity = true;
        //跳转resumeTopActivityInnerLocked
        result = resumeTopActivityInnerLocked(prev, options);
    ...
    } finally {
        mStackSupervisor.inResumeTopActivity = false;
    }
    return result;
}

//跳转到StackSupervisor.startSpecificActivityLocked，注释1
@GuardedBy("mService")
private boolean resumeTopActivityInnerLocked(ActivityRecord prev,
ActivityOptions options) {
    ...
    if (next.app != null && next.app.thread != null) {
        ...
    } else {
        ...
        mStackSupervisor.startSpecificActivityLocked(next, true, true);//1
    }
    if (DEBUG_STACK) mStackSupervisor.validateTopActivitiesLocked();
    return true;
}
```

5. 这里又回到了ActivityStackSupervisor，判断进程是否已经创建，未创建抛出异常。然后创建事务交回给本地执行。这里的事务很关键，Activity执行的工作就是这个事务，事务的内容是里面的item，所以要注意下面的两个item。

   复制代码

```
/frameworks/base/services/core/java/com/android/server/am/ActivityStackSuper
visor.java/;

void startSpecificActivityLocked(ActivityRecord r,
        boolean andResume, boolean checkConfig) {
    //得到即将启动的activity所在的进程
    ProcessRecord app = mService.getProcessRecordLocked(r.processName,
            r.info.applicationInfo.uid, true);
    getLaunchTimeTracker().setLaunchTime(r);

    //判断该进程是否已经启动,跳转realStartActivityLocked，真正启动活动
    if (app != null && app.thread != null) {
        try {
            if ((r.info.flags&ActivityInfo.FLAG_MULTIPROCESS) == 0
                    || !"android".equals(r.info.packageName)) {
                app.addPackage(r.info.packageName,
r.info.applicationInfo.longVersionCode,
                        mService.mProcessStats);
            }
            realStartActivityLocked(r, app, andResume, checkConfig);//1
            return;
        }
        ...
    }
    mService.startProcessLocked(r.processName, r.info.applicationInfo, true,
0,
```

```
            "activity", r.intent.getComponent(), false, false, true);
}


//主要创建事务交给本地执行
final boolean realStartActivityLocked(ActivityRecord r, ProcessRecord app,
    boolean andResume, boolean checkConfig) throws RemoteException {
    ...
    //创建启动activity的事务ClientTransaction对象
    // Create activity launch transaction.
    final ClientTransaction clientTransaction =
ClientTransaction.obtain(app.thread,
            r.appToken);
    // 添加LaunchActivityItem，该item的内容是创建activity
    clientTransaction.addCallback(LaunchActivityItem.obtain(new
Intent(r.intent),
            System.identityHashCode(r), r.info,
            // TODO: Have this take the merged configuration instead of
separate global
            // and override configs.
            mergedConfiguration.getGlobalConfiguration(),
            mergedConfiguration.getOverrideConfiguration(), r.compat,
            r.launchedFromPackage, task.voiceInteractor, app.repProcState,
r.icicle,
            r.persistentState, results, newIntents,
mService.isNextTransitionForward(),
            profilerInfo));

    // Set desired final state.
    //添加执行Resume事务ResumeActivityItem,后续会在本地被执行
    final ActivityLifecycleItem lifecycleItem;
    if (andResume) {
        lifecycleItem =
ResumeActivityItem.obtain(mService.isNextTransitionForward());
    } else {
        lifecycleItem = PauseActivityItem.obtain();
    }
    clientTransaction.setLifecycleStateRequest(lifecycleItem);

    // 通过ClientLifecycleManager来启动事务
    // 这里的mService就是AMS
    // 记住上面两个item：LaunchActivityItem和ResumeActivityItem，这是事务的执行单
位
    // Schedule transaction.
    mService.getLifecycleManager().scheduleTransaction(clientTransaction);
}
```

> 通过AMS获取ClientLifecycleManager
>
> 复制代码

```
/frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java/;
//通过AMS获取ClientLifecycleManager
ClientLifecycleManager getLifecycleManager() {
return mLifecycleManager;
}
```

6. ClientLifecycleManager是事务管理类，负责执行事务

   复制代码

```
/frameworks/base/services/core/java/com/android/server/am/ClientLifecycleManager.java
void scheduleTransaction(ClientTransaction transaction) throws
RemoteException {
    final IApplicationThread client = transaction.getClient();
    //执行事务
    transaction.schedule();
    if (!(client instanceof Binder)) {
        transaction.recycle();
    }
}
```

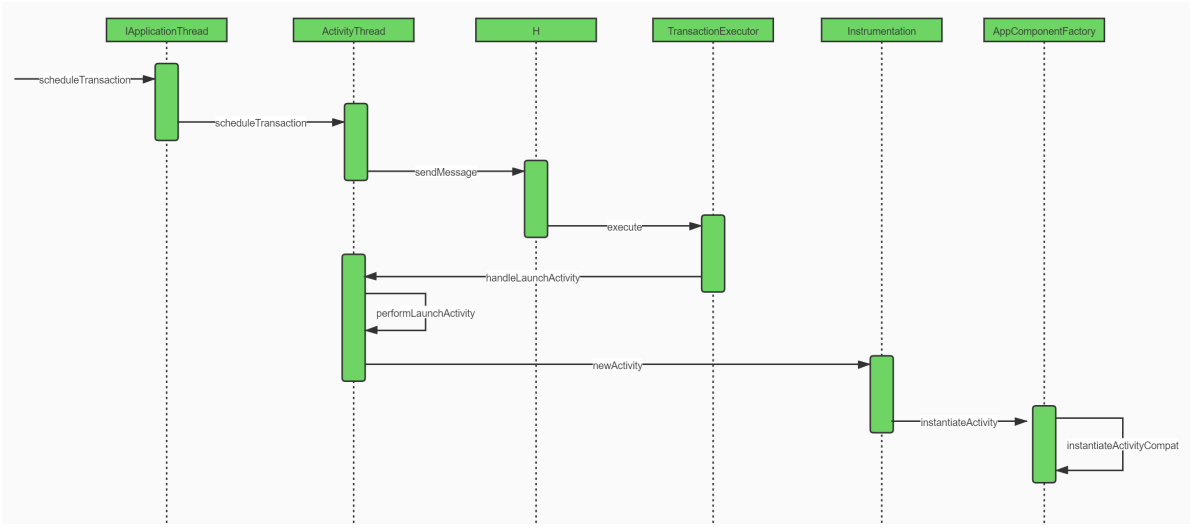
7. 把事务交给本地ActivityThread执行。这里通过本地ApplicationThread在服务端的接口IApplicationThread来进行跨进程通信。后面的逻辑就回到了应用程序进程了。

   复制代码

```
/frameworks/base/core/java/android/app/servertransaction/ClientTransaction.java/;

//这里的IApplicationThread是要启动进程的IBinder接口
//ApplicationThread是ActivityThread的内部类，IApplicationThread是IBinder代理接口
//这里将逻辑转到本地来执行
private IApplicationThread mClient;
public void schedule() throws RemoteException {
    mClient.scheduleTransaction(this);
}
```

ActivityThread创建Activity的过程

流程图



源码

1. IApplicationThread接口的本地实现类ActivityThread的内部类ApplicationThread

   复制代码

```
/frameworks/base/core/java/android/app/ActivityThread.java/ApplicationThread
.class/;
//跳转到ActivityThread的方法实现
public void scheduleTransaction(ClientTransaction transaction) throws
RemoteException {
    ActivityThread.this.scheduleTransaction(transaction);
}
```

2. ActivityThread执行事务。ActivityThread是继承ClientTransactionHandler，scheduleTransaction的具体实现是在ClientTransactionHandler实现的。这里的主要内容是把事务发送给ActivityThread的内部类H去执行。H是一个Handle，通过这个Handle来切到主线程执行逻辑。

   复制代码

```
/frameworks/base/core/java/android/app/ClientTransactionHandler.java
void scheduleTransaction(ClientTransaction transaction) {
    //事务预处理
    transaction.preExecute(this);
    //这里很明显可以利用Handle机制切换线程，下面看看这个方法的实现
    //该方法的具体实现是在ActivityThread，是ClientTransactionHandler的抽象方法
    sendMessage(ActivityThread.H.EXECUTE_TRANSACTION, transaction);
}

/frameworks/base/core/java/android/app/ActivityThread.java/;
final H mH = new H();
private void sendMessage(int what, Object obj, int arg1, int arg2, boolean
async) {
    if (DEBUG_MESSAGES) Slog.v(
        TAG, "SCHEDULE " + what + " " + mH.codeToString(what)
        + ": " + arg1 + " / " + obj);
    Message msg = Message.obtain();
    msg.what = what;
    msg.obj = obj;
    msg.arg1 = arg1;
    msg.arg2 = arg2;
    if (async) {
        msg.setAsynchronous(true);
    }
    //利用Handle进行切换。mH是H这个类的实例
    mH.sendMessage(msg);
}
```

3. H对事务进行处理。调用事务池来处理事务

   复制代码

```
/frameworks/base/core/java/android/app/ActivityThread.java/H.class
//调用事务池对事务进行处理
public void handleMessage(Message msg) {
```

```
    if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " +
codeToString(msg.what));
    switch (msg.what) {
        ...
        case EXECUTE_TRANSACTION:
            final ClientTransaction transaction = (ClientTransaction)
msg.obj;
            //调用事务池对事务进行处理
            mTransactionExecutor.execute(transaction);
            if (isSystem()) {
                transaction.recycle();
            }
            // TODO(lifecycler): Recycle locally scheduled transactions.
            break;
            ...
    }
    ...
}
```

4. 事务池对事务进行处理。事务池会把事务中的两个item拿出来分别执行。这两个事务就是上面我讲的两个Item。对应不同的初始化工作。

   复制代码

```
/frameworks/base/core/java/android/app/servertransaction/TransactionExecutor
.java

public void execute(ClientTransaction transaction) {
    final IBinder token = transaction.getActivityToken();
    log("Start resolving transaction for client: " + mTransactionHandler +
", token: " + token);

    //执行事务
    //这两个事务就是当时在ActivityStackSupervisor中添加的两个事件（第8步）
    //注释1执行activity的创建，注释2执行activity的窗口等等并调用onStart和onResume方
法
    //后面主要深入注释1的流程
    executeCallbacks(transaction);//1
    executeLifecycleState(transaction);//2
    mPendingActions.clear();
    log("End resolving transaction");
}

public void executeCallbacks(ClientTransaction transaction) {
    ...
        //执行事务
        //这里的item就是当初添加的Item，还记得是哪个吗？
        // 对了就是LaunchActivityItem
        item.execute(mTransactionHandler, token, mPendingActions);
        item.postExecute(mTransactionHandler, token, mPendingActions);
    ...
}

private void executeLifecycleState(ClientTransaction transaction) {
    ...
    // 和上面的一样，执行事务中的item，item类型是ResumeActivityItem
    lifecycleItem.execute(mTransactionHandler, token, mPendingActions);
```

```
    lifecycleItem.postExecute(mTransactionHandler, token, mPendingActions);
}
```

5. LaunchActivityItem调用ActivityThread执行创建逻辑。

   复制代码

```
/frameworks/base/core/java/android/app/servertransaction/LaunchActivityItem.
java/;

public void execute(ClientTransactionHandler client, IBinder token,
        PendingTransactionActions pendingActions) {
    Trace.traceBegin(TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
    ActivityClientRecord r = new ActivityClientRecord(token, mIntent,
mIdent, mInfo,
            mOverrideConfig, mCompatInfo, mReferrer, mVoiceInteractor,
mState, mPersistentState,
            mPendingResults, mPendingNewIntents, mIsForward,
            mProfilerInfo, client);
    // ClientTransactionHandler是ActivityThread实现的接口，具体逻辑回到
ActivityThread
    client.handleLaunchActivity(r, pendingActions, null /* customIntent */);
    Trace.traceEnd(TRACE_TAG_ACTIVITY_MANAGER);
}
```

6. ActivityThread执行Activity的创建。主要利用Instrumentation来创建activity和回调activity的生命周期，并创建activity的上下文和app上下文（如果还没创建的话）。

   复制代码

```
/frameworks/base/core/java/android/app/ActivityThread.java/;

public Activity handleLaunchActivity(ActivityClientRecord r,
        PendingTransactionActions pendingActions, Intent customIntent) {
    ...
        // 跳转到performLaunchActivity
        final Activity a = performLaunchActivity(r, customIntent);
    ...
}

//使用Instrumentation去创建activity回调生命周期
private Activity performLaunchActivity(ActivityClientRecord r, Intent
customIntent) {
        //获取ActivityInfo，用户存储代码、AndroidManifes信息。
        ActivityInfo aInfo = r.activityInfo;
        if (r.packageInfo == null) {
            //获取apk描述类
            r.packageInfo = getPackageInfo(aInfo.applicationInfo,
r.compatInfo,
                    Context.CONTEXT_INCLUDE_CODE);
        }

        // 获取activity的包名类型信息
        ComponentName component = r.intent.getComponent();
        if (component == null) {
            component = r.intent.resolveActivity(
                mInitialApplication.getPackageManager());
```

```
            r.intent.setComponent(component);
        }
    ...
        // 创建context上下文
        ContextImpl appContext = createBaseContextForActivity(r);
        // 创建activity
        Activity activity = null;
        try {
            java.lang.ClassLoader cl = appContext.getClassLoader();
            // 通过Instrumentation来创建活动
            activity = mInstrumentation.newActivity(
                    cl, component.getClassName(), r.intent);
            StrictMode.incrementExpectedActivityCount(activity.getClass());
            r.intent.setExtrasClassLoader(cl);
            r.intent.prepareToEnterProcess();
            if (r.state != null) {
                r.state.setClassLoader(cl);
            }
        }
    ...
        try {
            // 根据包名创建Application，如果已经创建则不会重复创建
            Application app = r.packageInfo.makeApplication(false,
mInstrumentation);
            ...
            // 为Activity添加window
            Window window = null;
            if (r.mPendingRemoveWindow != null && r.mPreserveWindow) {
                window = r.mPendingRemoveWindow;
                r.mPendingRemoveWindow = null;
                r.mPendingRemoveWindowManager = null;
            }
            appContext.setOuterContext(activity);
            activity.attach(appContext, this, getInstrumentation(), r.token,
                    r.ident, app, r.intent, r.activityInfo, title, r.parent,
                    r.embeddedID, r.lastNonConfigurationInstances, config,
                    r.referrer, r.voiceInteractor, window,
r.configCallback);
            }
    ...
        // 通过Instrumentation回调Activity的onCreate方法
        ctivity.mCalled = false;
        if (r.isPersistable()) {
            mInstrumentation.callActivityOnCreate(activity, r.state,
r.persistentState);
        } else {
            mInstrumentation.callActivityOnCreate(activity, r.state);
        }
}
```

> 这里深入来看一下onCreate什么时候被调用
>
> 复制代码

```
/frameworks/base/core/java/android/app/Instrumentation.java/;
public void callActivityOnCreate(Activity activity, Bundle icicle,
PersistableBundle persistentState) {
```

```
prePerformCreate(activity);
// 调用了activity的performCreate方法
activity.performCreate(icicle, persistentState);
postPerformCreate(activity);
}

/frameworks/base/core/java/android/app/Activity.java/;
final void performCreate(Bundle icicle, PersistableBundle
persistentState) {
mCanEnterPictureInPicture = true;
restoreHasCurrentPermissionRequest(icicle);
// 这里就回调了onCreate方法了
if (persistentState != null) {
onCreate(icicle, persistentState);
} else {
onCreate(icicle);
}
...
}
```

7. Instrumentation通过类加载器来创建activity实例

   复制代码

```
/frameworks/base/core/java/android/app/Instrumentation.java/;

public Activity newActivity(ClassLoader cl, String className,
        Intent intent)
        throws InstantiationException, IllegalAccessException,
        ClassNotFoundException {
    String pkg = intent != null && intent.getComponent() != null
            ? intent.getComponent().getPackageName() : null;
    // 利用AppComponentFactory进行实例化
    return getFactory(pkg).instantiateActivity(cl, className, intent);
}
```

8. 最后一步，通过AppComponentFactory工厂来创建实例。

   复制代码

```
/frameworks/support/compat/src/main/java/androidx/core/app/AppComponentFacto
ry.java
//其实就相当于直接返回instantiateActivityCompat
public final Activity instantiateActivity(ClassLoader cl, String className,
Intent intent)
    throws InstantiationException, IllegalAccessException,
ClassNotFoundException {
    return checkCompatWrapper(instantiateActivityCompat(cl, className,
intent));
}

//泛型方法
static <T> T checkCompatWrapper(T obj) {
        if (obj instanceof CompatWrapped) {
            T wrapper = (T) ((CompatWrapped) obj).getWrapper();
            if (wrapper != null) {
                return wrapper;
```

```
            }
        }
        return obj;
    }
//终于到了尽头了。利用类加载器来进行实例化。到此activity的启动就告一段落了。
public @NonNull Activity instantiateActivityCompat(@NonNull ClassLoader cl,
        @NonNull String className, @Nullable Intent intent)
        throws InstantiationException, IllegalAccessException,
ClassNotFoundException {
    try {
        return (Activity)
cl.loadClass(className).getDeclaredConstructor().newInstance();
    } catch (InvocationTargetException | NoSuchMethodException e) {
        throw new RuntimeException("Couldn't call constructor", e);
    }
}
```

小结

上文通过整体流程和代码详解解析了一个activity启动时的整体流程。不知道读者们会不会有个疑问：了解这些有什么用呢？日常又用不到。当走完这整个流程的时候，你会发现你对于android又深入了解了很多了，面对开发的时候，内心也会更加有自信心，出现的一些bug，可能别人要解决好久，而你，很快就可以解决。另外，这一部分内容在插件化也有很大的使用，也是学习插件化必学的知识。

(二) Activity生命周期源码分析

生命状态概述

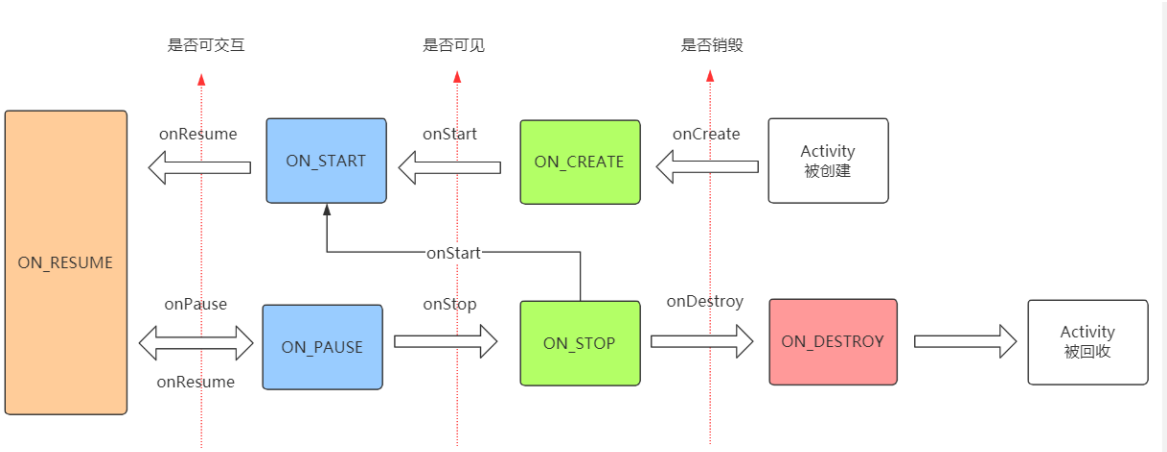
Activity是一个很重要、很复杂的组件，他的启动不像我们平时直接new一个对象就完事了，他需要经历一系列的初始化。例如"刚创建状态"，“后台状态”，“可见状态”等等。当我们在界面之间进行切换的时候，activity也会在多种状态之间进行切换，例如可见或者不可见状态、前台或者后台状态。**当Activity在不同的状态之间切换时，会回调不同的生命周期方法。我们可以重写这一些方法，当进入不同的状态的时候，执行对应的逻辑。**

在ActivityLifecycleItem`抽象类中定义了9种状态。这个抽象类有很多的子类，是AMS管理Activity生命周期的事务类。（其实就像一个圣旨，AMS丢给应用程序，那么应用程序就必须执行这个圣旨）Activity主要使用其中6个（这里的6个是笔者在源码中明确看到调用setState来设置状态，其他的三种并未看到调用setState方法来设置状态，所以这里主要讲这6种），如下：

复制代码

```
// Activity刚被创建时
public static final int ON_CREATE = 1;
// 执行完转到前台的最后准备工作
public static final int ON_START = 2;
// 执行完即将与用户交互的最后准备工作
// 此时该activity位于前台
public static final int ON_RESUME = 3;
// 用户离开，activity进入后台
public static final int ON_PAUSE = 4;
// activity不可见
public static final int ON_STOP = 5;
// 执行完被销毁前最后的准备工作
public static final int ON_DESTROY = 6;
```

状态之间的跳转不是随意的，例如不能从`ON_CREATE`直接跳转到`ON_PAUSE`状态，状态之间的跳转收到AMS的管理。当Activity在这些状态之间切换的时候，就会回调对应的生命周期。这里的状态看着很不好理解，笔者画了个图帮助理解一下：



这里按照「可交互」「可见」「可存在」三个维度来区分Activity的生命状态。可交互则为是否可以与用户操作；可见则为是否显示在屏幕上；可存在，则为该activity是否被系统杀死或者调用了finish方法。箭头的上方为进入对应状态会调用的方法。这里就先不展开讲每个状态之间的切换，主要是让读者可以更好地理解activity的状态与状态切换。

> 注意，这里使用的三个维度并不是非常严谨的，是结合总体的显示规则来进行区分的。
>
> 在谷歌的官方文档中对于`onStart`方法是这样描述的：`onStart()` 调用使 Activity 对用户可见，因为应用会为 Activity 进入前台并支持互动做准备。这也符合我们上面的维度的区分。而当activity进入ON_PAUSE状态的时候，Activity是可能依旧可见的，但是不可交互。如操作另一个应用的悬浮窗口的时候，当前应用的activity会进入ON_PAUSE状态。
>
> 但是！在activity启动的流程中，直到onResume方法被调用，界面依旧是不可见的。这点在后面的源码分析再详细解释。所以这里的状态区分维度，仅仅只是总体上的一种区分，可以这么认为，但细节上并不是非常严谨的。需要读者注意一下。

生命周期的一个重要作用就是让activity在不同状态之间切换的时候，可以执行对应的逻辑。举个栗子。我们在界面A使用了相机资源，当我们切换到下个界面B的时候，那么界面A就必须释放相机资源，这样才不会导致界面B无法使用相机；而当我们切回界面A的时候，又希望界面A继续保持拥有相机资源的状态；那么我们就需要在界面不可见的时候释放相机资源，而在界面恢复的时候再次获取相机资源。每个Activity一般情况下可以认为是一个界面或者说，一个屏幕。当我们在界面之间进行导航切换的时候，其实就是在切换Activity。当界面在不同状态之间进行切换的时候，也就是Activity状态的切换，就会回调activity相关的方法。例如当界面不可见的时候会回调`onStop`方法，恢复的时候会回调`onReStart`方法等。

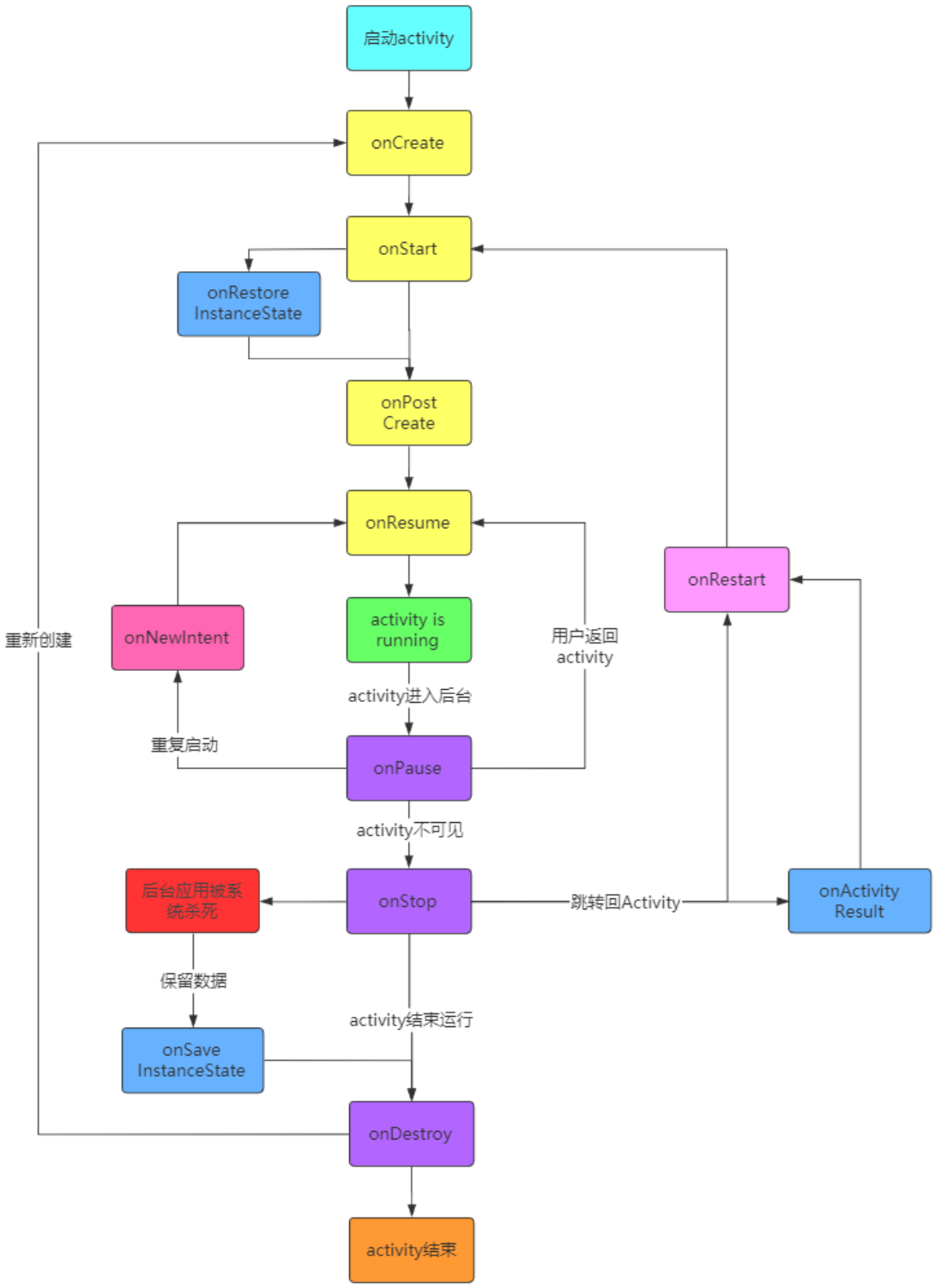
在合适的生命周期做合适的工作会让app变得更加有鲁棒性。避免当用户跳转别的app的时候发生崩溃、内存泄露、当用户切回来的时候失去进度、当用户旋转屏幕的时候失去进度或者崩溃等等。这些都需要我们对生命周期有一定的认知，才能在具体的场景下做出最正确的选择。

这一部分概述并没有展开讲生命周期，而是需要重点理解**状态与状态之间的切换，生命周期的回调就发生在不同的状态之间的切换。**我们学习生命周期的一个重要目的就是**能够在对应的业务场景下做合适的工作**，例如资源的申请、释放、存储、恢复，让app更加具有鲁棒性。

重要生命周期解析

关于Activity重要的生命周期回调方法谷歌官方有了一张非常重要的流程图，可以说是人人皆知。我在这张图上加上了一些常用的回调方法。这些方法严格上并不算Activity的生命周期，因为并没有涉及到状态的切换，但却在Activity的整个生命历程中发挥了非常大的作用，也是很重要的回调方法。方法很多，我们先看图，再一个个地解释。看具体方法解释的时候一定要结合下面这张图以及上一部分概述的图一起

理解。



主要生命周期

首先我们先看到最重要的七个生命周期，这七个生命周期是严格意义上的生命周期，他符合**状态切换**这个关键定义。这部分内容建议结合概述部分的图一起理解。（onRestart并不涉及状态变换，但因为执行完他之后会马上执行onStart，所以也放在一起讲）

- onCreate：当Activity创建实例完成，并调用attach方法赋值PhoneWindow、ContextImpl等属性之后，调用此方法。该方法在整个Activity生命周期内只会调用一次。调用该方法后Activity进入`ON_CREATE`状态。

> 该方法是我们使用最频繁的一个回调方法。

> 我们需要在这个方法中初始化基础组件和视图。如viewmodel，textview。同时必须在该方法中调用setContentView来给activity设置布局。
>
> 这个方法接收一个参数，该参数保留之前状态的数据。如果是第一次启动，则该参数为空。该参数来自onSaveInstanceState存储的数据。只有当activity暂时销毁并且预期一定会被重新创建的时候才会被回调，如屏幕旋转、后台应用被销毁等

- onStart：当Activity准备进入前台时会调用此方法。调用后Activity会进入`ON_START`状态。

> 要注意理解这里的前台的意思。虽然谷歌文档中表示调用该方法之后activity可见，如下图：
>
> 但是我们前面讲到，**前台并不意味着Activity可见，只是表示activity处于活跃状态。**这也是谷歌文档里让我比较迷惑的地方之一。（事实上谷歌文档有挺多地方写的缺乏严谨，可能考虑到易懂性，就牺牲了一点严谨性吧）。
>
> 前台activity一般只有一个，所以这也意味着**其他的activity都进入后台了**。这里的前后台需要结合activity返回栈来理解，后续笔者再写一篇关于返回栈的。
>
> **这个方法一般用于从别的activity切回来本activity的时候调用。**

- onResume：当Activity准备与用户交互的时候调用。调用之后Activity进入`ON_RESUME`状态。

> 注意，这个方法一直被认为是activity一定可见，且准备好与用户交互的状态。但事实并不**一直**是这样。如果在`onReume`方法中弹出popupWindow你会收获一个异常：`token is null`，表示界面尚没有被添加到屏幕上。
>
> 但是，这种情况只出现在第一次启动activity的时候。当activity启动后decorview就已经拥有token了，再次在`onReume`方法中弹出popupWindow就不会出现问题了。
>
> 因此，**在onResume调用的时候activity是否可见要区分是否是第一次创建activity。**
>
> onStart方法是后台与前台的区分，而这个方法是是否可交互的区分。使用场景最多是在当弹出别的activity的窗口时，原activity就会进入`ON_PAUSE`状态，但是仍然可见；当再次回到原activity的时候，就会回调onResume方法了。

- onPause：当前activity窗口失去焦点的时候，会调用此方法。调用后activity进入`ON_PAUSE`状态，并进入后台。

> 这个方法一般在另一个activity要进入前台前被调用。只有当前activity进入后台，其他的activity才能进入前台。所以，**该方法不能做重量级的操作，不然则会引用界面切换卡顿。**
>
> 一般的使用场景为界面进入后台时的轻量级资源释放。
>
> 最好理解这个状态就是弹出另一个activity的窗口的时候。因为前台activity只能有一个，所以当前可交互的activity变成另一个activity后，原activity就必须调用onPause方法进入`ON_PAUSE`状态；但是！！仍然是可见的，只是无法进行交互。这里也可以更好地体会前台可交互与可见性的区别。

- onStop：当activity不可见的时候进行调用。调用后activity进入`ON_STOP`状态。

> 这里的不可见是严谨意义上的不可见。
>
> 当activity不可交互时会回调onPause方法并进入`ON_PAUSE`状态。但如果进入的是另一个全屏的activity而不是小窗口，那么当新的activity界面显示出来的时候，原Activity才会进入`ON_STOP`状态，并回调onStop方法。同时，activity第一创建的时候，界面是在onResume方法之后才显示出来，所以onStop方法会在新activity的onResume方法回调之后再被回调。
>
> 注意，被启动的activity并不会等待onStop执行完毕之后再显示。因而如果onStop方法里做一些比较耗时的操作也不会导致被启动的activity启动延迟。

> onStop方法的目的就是做**资源释放操作**。因为是在另一个activity显示之后再被回调，所以这里可以做一些相对重量级的资源释放操作，如中断网络请求、断开数据库连接、释放相机资源等。
>
> 如果一个应用的全部activity都处于`ON_STOP`状态，那么这个应用是很有可能被系统杀死的。而如果一个`ON_STOP`状态的activity被系统回收的话，系统会保留该activity中view的相关信息到bundle中，下一次恢复的时候，可以在onCreate或者onRestoreInstanceState中进行恢复。

- onRestart：当从另一个activity切回到该activity的时候会调用。调用该方法后会立即调用onStart方法，之后activity进入`ON_START`状态。

> 这个方法一般在activity从`ON_STOP`状态被重新启动的时候会调用。执行该方法后会立即执行onStart方法，然后Activity进入`ON_START`状态，进入前台。

- onDestroy：当activity被系统杀死或者调用finish方法之后，会回调该方法。调用该方法之后activity进入`ON_DESTROY`状态。

> 这个方法是activity在被销毁前回调的最后一个方法。我们需要在这个方法中释放所有的资源，防止造成内存泄漏问题。
>
> 回调该方法后的activity就等待被系统回收了。如果再次打开该activity需要从onCreate开始执行，重新创建activity。

那到这里七个最关键的生命周期方法就讲完了。需要读者注意的是，在概述一图中，我们使用了三个维度来进行区分不同类型的状态，但是很明显，同一类型的状态并不是等价的。如`ON_START`状态表示activity进入前台，而`ON_PAUSE`状态却表示activity进入后台。这可能也是为什么谷歌要区分出on_start和on_pause两个状态，他们代表并不是一致的状态。

这七个生命周期回调方法是最重要的七个生命周期回调方法，需要读者好好理解每个回调方法设计到的activity的状态转换。而理解了状态转换后，也就可以写出更加强壮的代码了。

其他生命周期回调方法

- onActivityResult

这个方法也很常见，他需要结合`startActivityForResult`一起使用。

使用的场景是：启动一个activity，并期望在该activity结束的时候返回数据。

当启动的activity结束的时候，返回原activity，原activity就会回调onActivityResult方法了。**该方法执行在其他所有的生命周期方法前。**关于onActivityResult如何使用这里就不展开了，我们主要介绍生命周期。

- onSaveInstanceState/onRestoreInstanceState

这两个方法，主要用于在Activity被意外杀死的情况下进行界面数据存储与恢复。什么叫意外杀死呢?

如果你主动点击返回键、调用finish方法、从多任务列表清除后台应用等等，这些操作表示用户想要完整得退出activity，那么就没有必要保留界面数据了，所以也不会调用这两个方法。而当应用被系统意外杀死，或者系统配置更改导致的activity销毁，这个时候当用户返回activity时，期望界面的数据还在，则会通过回调onSaveInstanceState方法来保存界面数据，而在activity重新创建并运行的时候调用onRestoreInstanceState方法来恢复数据。事实上，onRestoreInstanceState方法的参数和onCreate方法的参数是一致的，只是他们两个方法回调的时机不同。因此，判断是否执行的关键因素就是**用户是否期望返回该activity时界面数据仍然存在。**

这里需要注意几个点：

1. 不同android版本下，onSaveInstanceState方法的调用时机是不同的。目前笔者的源码是api30，在官方注释中可以看到这么一句话：

复制代码

```
/*If called, this method will occur after {@link #onStop} for applications
 * targeting platforms starting with {@link
android.os.Build.VERSION_CODES#P}.
 * For applications targeting earlier platform versions this method will
occur
 * before {@link #onStop} and there are no guarantees about whether it will
 * occur before or after {@link #onPause}.
 */
```

翻译过来意思就是，在api28及以上版本onSaveInstanceState是在onStop之后调用的，但是在低版本中，他是在onStop之前被调用的，且与onPause之间的顺序是不确定的。

2. 当activity进入后台的时候，onSaveInstanceState方法则会被调用，而不是异常情况下才会调用onSaveInstanceState方法，因为并不确定在后台时，activity是否会被系统杀死，所以以最保险的方法，先保存数据。当确实是因为异常情况被杀死时，返回activity用户期望界面需要恢复数据，才会调用onRestoreInstanceState来恢复数据。但是，activity直接按返回键或者调用finish方法直接结束Activity的时候，是不会回调onSaveInstanceState方法，因为非常明确下一次返回该activity用户期望的是一个干净界面的新activity。
3. onSaveInstanceState不能做重量级的数据存储。onSaveInstanceState存储数据的原理是把数据序列化到磁盘中，如果存储的数据过于庞大，会导致界面卡顿，掉帧等情况出现。
4. 正常情况下，每个view都会重写这两个方法，当activity的这两个方法被调用的时候，会向上委托window去调用顶层viewGroup的这两个方法；而viewGroup会递归调用子view的onSaveInstanceState/onRestoreInstanceState方法，这样所有的view状态就都被恢复了。

关于界面数据恢复这里也不展开细讲了，有兴趣的读者可以自行深入研究。

- onPostCreate

这个方法其实和onPostResume是一样的，同样的还有onContextChange方法。这三个方法都是不常用的，这里也点出其中一个来统一讲一下。

onPostCreate方法发生在onRestoreInstanceState之后，onResume之前，他代表着界面数据已经完全恢复，就差显示出来与用户交互了。在onStart方法被调用时这些操作尚未完成。

onPostResume是在Resume方法被完全执行之后的回调。

onContentChange是在setContentView之后的回调。

这些方法都不常用，仅做了解。如果真的遇到了具体的业务需求，也可以拿出来用一下。

- onNewIntent

这个方法涉及到的场景也是重复启动，但是与onRestart方法被调用的场景是不同的。

我们知道activity是有多种启动模式的，其中singleInstance、singleTop、singleTask都保证了在一定情况下的单例状态。如singleTop，如果我们启动一个正处于栈顶且启动模式为singleTop的activity，那么他并不会在创建一个activity实例，而是会回调该activity的onNewIntent方法。该方法接收一个intent参数，该参数就是新的启动Intent实例。

其他的情况如singleTask、singleInstance，当遇到这种强制单例情况时，都会回调onNewIntent方法。关于启动模式这里也不展开，后续笔者可能会再出一期文章讲启动模式。

场景生命周期流程

这一部分主要是讲解在一些场景下，生命周期方法的回调顺序。对于当个Activity而言，上述流程图已经展示了各种情况下的生命周期回调顺序了。但是，当启动另一个activity的时候，到底是onStop先执行，还是被启动的onStart先执行呢？这些就变得比较难以确定。

验证生命周期回调顺序最好的方法就是写demo，通过日志打印，可以很明显地观察到生命周期的回调顺序。当然，查看源码也是一个不错的方法，但是需要对系统源码有一定的认识，我们还是选择简单粗暴的方法。

正常启动与结束

> onCreate -> onStart -> onResume -> onPause -> onStop -> onDestroy

这种情况的生命周期比较好理解，就是常规的启动与结束，也不会涉及到第二个activity。最后看日志打印：

```
D/修仙的第一个Activity: : onCreate:
D/修仙的第一个Activity: : onStart:
D/修仙的第一个Activity: : onPostCreate:
D/修仙的第一个Activity: : onResume:
D/修仙的第一个Activity: : onPostResume:
D/修仙的第一个Activity: : onPause:
D/修仙的第一个Activity: : onStop:
D/修仙的第一个Activity: : onDestroy:
```

Activity切换

> Activity1:onPause
> Activity2:onCreate -> onStart -> onResume
> Activity1:onStop

当切换到另一个activity的时候，本activity会先调用onPause方法，进入后台；被启动的activity依次调用三个回调方法后准备与用户交互；这时原activity再调用onStop方法变得不可见，最后被启动的activity才会显示出来。

理解这个生命周期顺序只需要记住两个点：前后台、是否可见。onPause调用之后，activity会进入后台。而前台交互的activity只能有一个，所以原activity必须先进入后台后，目标activity才能启动并进入前台。onStop调用之后activity变得不可见，因而只有在目标activity即将要与用户交互的时候，需要进行显示了，原Activity才会调用onStop方法进入不可见状态。

当从Activity2回退到Activity1的时候，流程也是类似的，只是Activity1会在其他生命周期之前执行一次onRestart，跟前面的流程是类似的。读者可以看一下下面的日志打印，这里就不再赘述了。

下面看一下切换到另一个activity的生命周期日志打印：

```
D/修仙的第一个Activity: : onCreate:
D/修仙的第一个Activity: : onStart:
D/修仙的第一个Activity: : onPostCreate:
D/修仙的第一个Activity: : onResume:
D/修仙的第一个Activity: : onPostResume:
D/修仙的第一个Activity: : onPause:
D/修仙的第二个Activity: onCreate:
D/修仙的第二个Activity: onStart:
D/修仙的第二个Activity: onResume:
D/修仙的第一个Activity: : onStop:
D/修仙的第一个Activity: : onSaveInstanceState:
```

这里我们看到最后回调了onSaveInstanceState方法，前面我们讲到了，当activity进入后台的时候，会调用该方法来保存数据。因为并不知道在后台时activity是否会被系统杀死。下面再看一下从activity2返回的时候，生命周期的日志打印：

```
D/修仙的第二个Activity: onPause:
D/修仙的第一个Activity: : onRestart:
D/修仙的第一个Activity: : onStart:
D/修仙的第一个Activity: : onResume:
D/修仙的第一个Activity: : onPostResume:
D/修仙的第二个Activity: onStop:
D/修仙的第二个Activity: onDestroy:
```

屏幕旋转

> running -> onPause -> onStop -> onSaveInstanceState -> onDestroy
>
> onCreate -> onStart -> onRestoreInstanceState -> onResume

当因资源配置改变时，activity会销毁重建，最常见的就是屏幕旋转。这个时候属于异常情况的Activity生命结束。因而，在销毁的时候，会调用onSaveInstanceState来保存数据，在重新创建新的activity的时候，会调用onRestoreInstanceState来恢复数据。

看一下日志打印：

```
D/修仙的第一个Activity: : onPause:
D/修仙的第一个Activity: : onStop:
D/修仙的第一个Activity: : onSaveInstanceState:
D/修仙的第一个Activity: : onDestroy:
D/修仙的第一个Activity: : onCreate:
D/修仙的第一个Activity: : onStart:
D/修仙的第一个Activity: : onRestoreInstanceState:
D/修仙的第一个Activity: : onPostCreate:
D/修仙的第一个Activity: : onResume:
D/修仙的第一个Activity: : onPostResume:
```

后台应用被系统杀死

> onDestroy
>
> onCreate -> onStart -> onRestoreInstanceState -> onResume

这个流程跟上面的资源配置更改是很像的，只是每个activity不可见的时候，会回调onSaveInstanceState提前保存数据，那么在被后台杀死的时候，就不需要再次保存数据了。

具有返回值的启动

> onActivityResult -> onRestart -> onResume

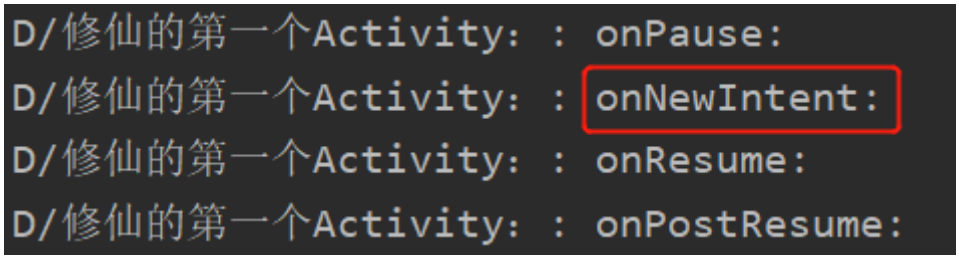
这里主要针对使用startActivityForResult方法启动另一个activity，当该activity销毁并返回时，原activity的onActivityResult方法的执行时机。大部分流程和activity切换是一样的。但在返回原Activity时，onActivityResult方法会在其他所有的生命周期方法执行前被执行。看一下日志打印：

```
D/修仙的第二个Activity: onPause:
D/修仙的第一个Activity: : onActivityResult:
D/修仙的第一个Activity: : onRestart:
D/修仙的第一个Activity: : onStart:
D/修仙的第一个Activity: : onResume:
D/修仙的第一个Activity: : onPostResume:
D/修仙的第二个Activity: onStop:
D/修仙的第二个Activity: onDestroy:
```

重复启动

> onPause -> onNewIntent -> onResume

这个流程是比较容易在学习生命周期的时候被忽略的。前面已经有讲到了关于onNewIntent的相关介绍，这里就不再赘述。主要是记得如果当前activity正处于栈顶，那么会先回调onPause之后再回调onNewIntent。关于启动模式以及返回栈的设计这里就不展开讲了，记住生命周期就可以了。看一下日志打印：



从源码看生命周期

到这里关于生命周期的一些应用知识就已经讲得差不多了，这一部分是深入源码，去探究生命周期在源码中是如何实现的。这样对生命周期会有更加深刻的理解，同时可以更加了解android系统的源码设计。

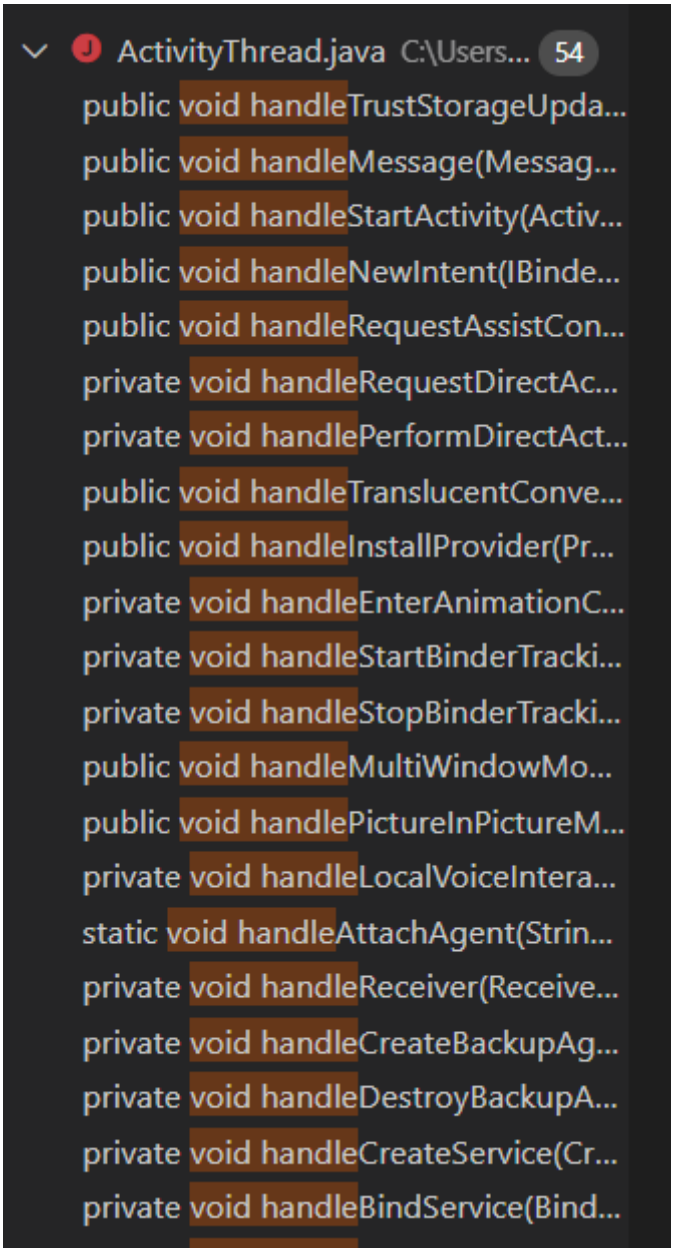
由于生命周期方法很多，笔者不可能一一讲解，这样篇幅就太大了且没有意义。这一部分的内容一共分为两个部分：第一部分是概述一下ActivityThread中关于每个生命周期的调用方法，这样大家就懂得如何去寻找对应的源码来研究；第二部分是拿onResume这个方法来举例讲解，同时解释为什么在第一次启动时，当onResume被调用时界面依然不可见。

从ActivityThread看生命周期

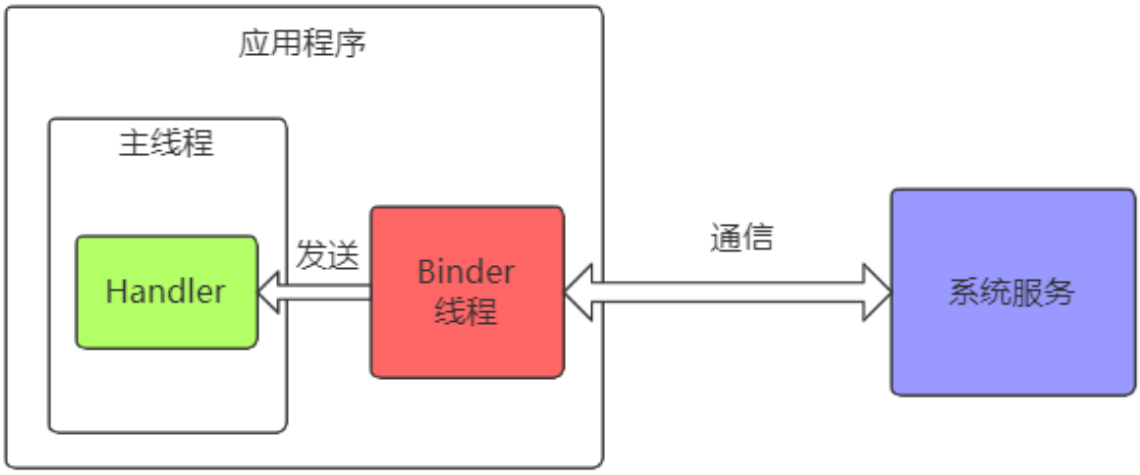
我们都知道，Activity的启动是受AMS调配的，那具体的调配方式是如何的呢?

通过Handler机制一文我们知道，android的程序执行是使用handler机制来实现消息驱动型的。AMS想要控制Activity的生命周期，就必须不断地向主线程发送message；而程序想要执行AMS的命令，就必须handle这些message执行逻辑，两端配合，才能达到这种效率。

打个比方，领导要吩咐下属去工作，他肯定不会把工作的具体流程都给下属，而只是会发个命令，如：给明天的演讲做个ppt，给我预约个下星期的飞机等等。那么下属，就必须根据这些命令来执行指定的逻辑。所以，在android程序，肯定有一系列的逻辑，来分别执行来自AMS的“命令”。这就是ActivityThread中的一系列handlexxx方法。给个我在vs code中的搜索图感受一下：



当然，应用程序不止收到AMS的管理，同样的还有WMS、PMS等等系统服务。系统服务是运行在系统服务进程的，当系统服务需要控制应用程序的时候，会通过Binder跨进程通信把消息发送给应用程序。应用程序的Binder线程会通过handler把消息发送给主线程去执行。因而，从这里也可以看出，当应用程序刚被创建的时候，必须初始化的有主线程、binder线程、主线程handler、以及提前编写了命令的执行逻辑的类ActivityThread。光说不画假解释，画个图感受一下：



回到我们的生命周期主题。关于生命周期命令的执行方法主要有：

复制代码

```
handleLaunchActivity;
handleStartActivity;
handleResumeActivity;
handlePauseActivity;
handleStopActivity;
handleDestroyActivity;
```

具体的方法当然不止这么多，只是列出一些比较常用的。这些方法都在ActivityThread中。ActivityThread每个应用程序有且只有一个，他是系统服务"命令"的执行者。

了解了AMS如何调配之后，那么他们的执行顺序如何确定呢？AMS是先发送handleStartActivity命令呢，还是先发送handleResumeActivity？这里就需要我们对Activity的启动流程有一定的认知，感兴趣读者可以点击Activity启动流程前往学习，这里就不展开了。

最后再延伸一下，那，ActivityThread可不可以自己决定执行逻辑，而不理会AMS的命令呢？答案肯定是no。你想啊，如果在公司里，你没有老板的同意，你能动用公司的资源吗？回到Android系统也是一样的，没有AMS的授权，应用程序是无法得到系统资源的，所以AMS就保证了每一个程序都必须符合一定的规范。关于这方面的设计，读者感兴趣可以阅读context机制这篇文章了解一下,重新认识一下context。

好了，扯得有点远，我们回到主题。下面呢就不展开去跟踪整个流程了，而是定位到具体的handle方法去看看具体执行了什么逻辑，对源码流程感性去的读者可以自行研究，限于篇幅这里就不展开了。下面主要介绍`handleResumeActivity`方法。

解析onRusume源码

根据我们前面的学习，`handleResumeActivity`肯定是在`handleLaunchActivity`和`handleStartActivity`之后被执行的。我们直接来看源码：

复制代码

```
public void handleResumeActivity(IBinder token, boolean finalStateRequest,
boolean isForward,
        String reason) {
    ...
    final ActivityClientRecord r = performResumeActivity(token,
finalStateRequest, reason);
    ...;
    if (!r.activity.mFinished && willBeVisible && r.activity.mDecor != null &&
!r.hideForNow) {
        ...
        if (r.activity.mVisibleFromClient) {
            r.activity.makeVisible();
        }
    }
    ...
}
```

代码我截取了两个非常重要的部分。`performResumeActivity`最终会执行`onResume`方法；`activity.makeVisible();`是真正让界面显示在屏幕个上的方法，我们看一下`makeVisible()`：

复制代码

```
void makeVisible() {
    if (!mWindowAdded) {
        ViewManager wm = getWindowManager();
        wm.addView(mDecor, getWindow().getAttributes());
        mWindowAdded = true;
    }
    mDecor.setVisibility(View.VISIBLE);
}
```

如果尚未添加到屏幕上，那么会调用windowManager的addView方法来添加，之后，activity界面才真正显示在屏幕上。回应之前的问题：为什么在onResume弹出popupWindow会抛异常而弹出dialog却不会？原因就是这个时候activity的界面尚未添加到屏幕上，而popupWindow需要依附于父界面，这个时候弹出就会抛出`token is null`异常了。而Dialog属于应用层级窗口，不需要依附于任何窗口，所以dialog在onCreate方法中弹出都是没有问题的。为了验证我们的判断，我在生命周期中打印decorView的windowToken，当decorView被添加到屏幕上后，就会被赋值token了，看日志打印：

```
D/修仙的第一个Activity: : onCreate: null
D/修仙的第一个Activity: : onStart: null
D/修仙的第一个Activity: : onPostCreate:
D/修仙的第一个Activity: : onResume: null
D/修仙的第一个Activity: : onPostResume: null
D/修仙的第一个Activity: : onWindowFocusChanged: android.view.ViewRootImpl$W@75e86c8
D/修仙的第一个Activity: : onPause: android.view.ViewRootImpl$W@75e86c8
D/修仙的第一个Activity: : onWindowFocusChanged: android.view.ViewRootImpl$W@75e86c8
D/修仙的第一个Activity: : onStop:android.view.ViewRootImpl$W@75e86c8
D/修仙的第一个Activity: : onDestroy: android.view.ViewRootImpl$W@75e86c8
```

可以看到，直到onPostResume方法执行，界面依旧没有显示在屏幕上。而直到onWindowFocusChange被执行时，界面才是真正显示在屏幕上了。

好了，让我们再回到一开始的源码，深入performResumeActivity方法中看看，在哪里执行了onResume方法：

复制代码

```
public ActivityClientRecord performResumeActivity(IBinder token, boolean
finalStateRequest,
        String reason) {
    ...
    try {
        ...
        if (r.pendingIntents != null) {
            // 判断是否需要执行newIntent方法
            deliverNewIntents(r, r.pendingIntents);
            r.pendingIntents = null;
        }
        if (r.pendingResults != null) {
            // 判断是否需要执行onActivityResult方法
            deliverResults(r, r.pendingResults, reason);
            r.pendingResults = null;
        }
        // 回调onResume方法
        r.activity.performResume(r.startsNotResumed, reason);

        r.state = null;
        r.persistentState = null;
        // 设置状态
```

```
            r.setState(ON_RESUME);

            reportTopResumedActivityChanged(r, r.isTopResumedActivity,
"topWhenResuming");
        }
        ...
}
```

这个方法的重点就是，先判断是否是需要执行onNewIntent或者onActivityResult的场景，如果没有则执行调用`performResume`方法，我们深入`performResume`看一下：

复制代码

```
final void performResume(boolean followedByPause, String reason) {
    dispatchActivityPreResumed();
    performRestart(true /* start */, reason);
    ...
    mInstrumentation.callActivityOnResume(this);
    ...
    onPostResume();
    ...
}

public void callActivityOnResume(Activity activity) {
    activity.mResumed = true;
    activity.onResume();
    ...
}
```

同样只看重点。首先会调用performRestart方法，这个方法内部会判断是否需要执行onRestart方法和onStart方法，如果是从别的activity返回这里是肯定要执行的。然后使用Instrumentation来回调Activity的onResume方法。当onResume回调完成后，会再调用onPostResume()方法。

那么到这里关于handleResumeActivity的方法就讲完了，为什么在onResume甚至onPostResume方法被回调的时候界面尚未显示，也有了更加深刻的认识。具体的代码逻辑非常多，而关于生命周期的代码我只挑了重点来讲，其他的源码，感兴趣的读者可以自行去查阅源码。笔者这里更多的是充当一个抛砖引玉的效果。要从源码中学习到知识，就必须自己手动去阅读源码，跟着文章看完事实上收获是不大的。

从系统设计看Activity与其生命周期

在笔者认为，每一个知识，都是在具体的场景下为了解决具体的问题，通过权衡各种条件设计出来的。在学习了每一个知识之后，笔者总是喜欢反过来，思考一下这一块知识的底层设计思想是什么，他是需要解决什么问题，权衡了什么条件。通过不断思考来从一个更高的角度来看待每一个知识点。

要理解生命周期的设计，首先需要理解Activity本身。想一下，如果没有Activity，那么我们该如何编写程序？有没有忽然反应到，没有了activity，我们的程序竟无处下手？因为这涉及到Activity的一个最大的作用：`Activity` **类是 Android 应用的关键组件，而 Activity 的启动和组合方式则是该平台应用模型的基本组成部分。**

相信很多读者都写过c语言、java或者其他语言的课程设计，我们的程序入口就是`main`函数。从main函数开始，根据用户的输入来进入不同的功能模块，如更改信息模块、查阅模块等等。**以功能模块为基本组成部分的应用模型**是我们最初的程序设计模型。而android程序，我们会说这个程序有几个**界面**。我们更关注的是界面之间的跳转，而不是功能模块之间的跳转。我们在设计程序的时候，我们会说这个界面有什么功能，那个界面有什么功能，多个界面之间如何协调。对于用户来说，他们感知的也是一个个独立的界面。当我们通过通讯软件调用邮箱app的发送邮件界面时，我们喜欢看到的只是发送邮件的界

面，而不需要看到收件箱、登录注册等界面。以功能模块为应用模型的设计从一个主功能模块入口，然后通过用户的输入去调用不同的功能模块。与其类似，android程序也有一个主界面，通过这个主界面，接受用户的操作去调用其他的界面。组成android程序的，是一个个的界面，而每一个界面，对应一个Activity。因此，**Activity是android平台应用模型的基本组成成分。**

功能模块的应用模型从main方法进入主功能模块，而android程序从ActivityThread的main方法开始，接收AMS的调度启动“LaunchActivity”，也就是我们在AndroidManifest中配置为main的activity，当应用启动的时候，就会首先打开这个activity。那么第一个界面被打开，其他的界面就根据用户的操作来依次跳转了。

那如何做到每个界面之间彼此解耦、各自的显示不发生混乱、界面之间的跳转有条不紊等等？这些工作，官方都帮我们做好了。Activity就是在这个设计思想下开发出来的。当我们在Activity上开发的时候，就已经沿用了他的这种设计思想。当我们开发一个app的时候，最开始要考虑的，是界面如何设计。设计好界面之后，就是考虑如何开发每个界面了。那我们如何自定义好每一个界面？如何根据我们的需求去设计每个界面的功能？Activity并没有main方法，我们的代码该写在哪里被执行？答案就是：**生命周期回调方法。**

到这里，你应该可以理解为什么启动activity并不是一句new就可以解决的吧？Activity承担的责任非常多，需要初始化的逻辑也非常多。当Activity被启动，他会根据自身的启动情况，来回调不同的生命周期方法。其中，**承担初始化整个界面已经各个功能组件的初始化任务**的就是onCreate方法。他有点类似于我们功能模块的入口函数，在这里我们通过setContentView来设计我们界面的布局，通过setOnClickListenner来给每个view设置监听等等。在MVVM设计模式中，还需要初始化viewModel、绑定数据等等。这是生命周期的第一个非常重要的意义所在。

而当界面的显示、退出，我们需要为之申请或者释放资源。如我上文举的相机例子，我在微信的扫一扫申请了相机权限，如果进入后台的时候没有释放资源，那么打开系统相机就无法使用了，资源被占领了。因此，生命周期的另一个重要的作用就是：**做好资源的申请与释放，避免内存泄露。**

其他生命周期的作用，如界面数据恢复、界面切换逻辑处理等等就不再赘述了，前面已经都有涉及到。

这一部分的重点就是理解**android应用程序是以Activity为基本组成部分的应用模型**这个点。当界面的启动以及不同界面之间进行切换的时候，也就可以更加感知生命周期的作用了。

最后

关于Activity生命周期的内容，在一篇文章讲完整是不可能的。当对他研究地越深，涉及到内容就会越多。每个知识点就是像是瓜藤架上的一个瓜，如果单纯摘瓜，那就是一个瓜；如果抓着藤蔓往外拔，那么整个架子都会被扯出来。这篇文章也当是抛砖引玉，在讲生命周期相关的知识讲完之后，提供给读者一个思考的思路。

第二章： Fragment深度解析

(一) Fragment事务管理机制详解

概述

在Fragment使用中，有时候需要对Fragment进行`add`、`remove`、`show`、`hide`、`replace`等操作来进行Fragment的显示隐藏等管理，这些管理是通过`FragmentTransaction`进行事务管理的。事务管理是对于一系列操作进行管理，一个事务包含一个或多个操作命令，是逻辑管理的工作单元。一个事务开始于第一次执行操作语句，结束于Commit。通俗地将，就是把多个操作缓存起来，等调用commit的时候，统一批处理。下面会对Fragmeng的事务管理做一个代码分析

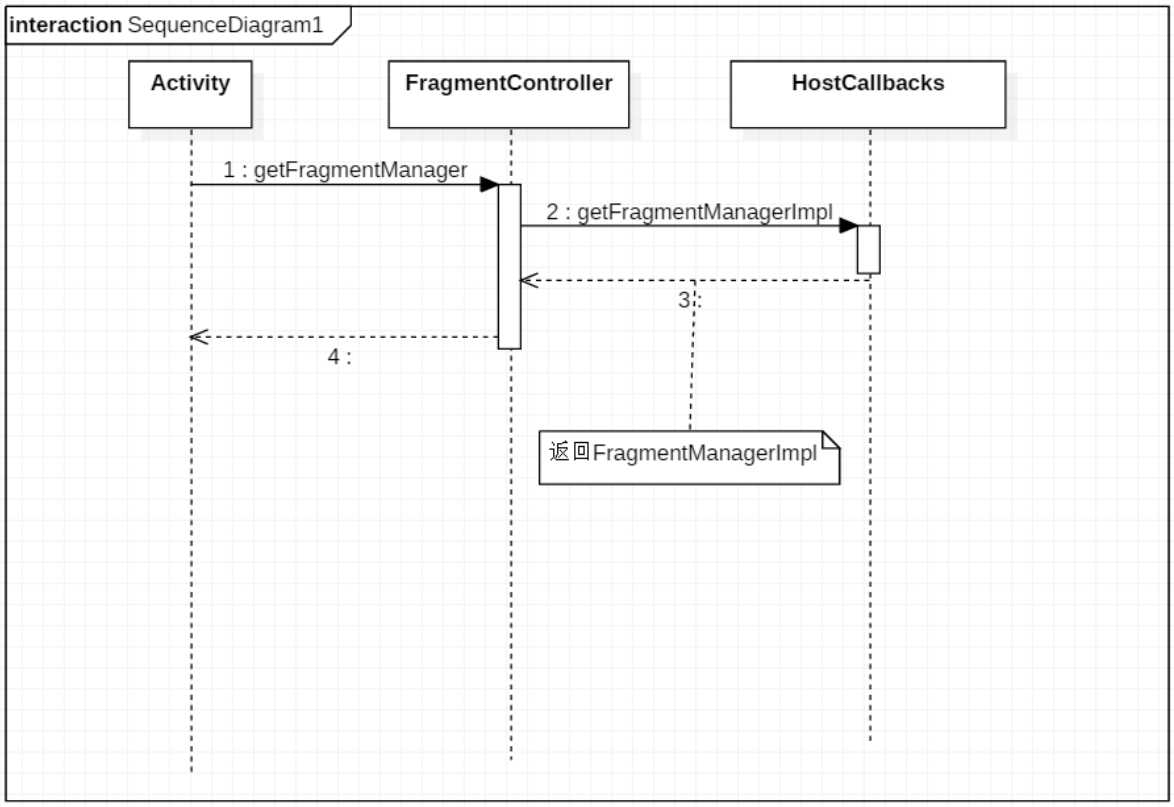
分析入口

```
    /**
     * 显示Fragment，如果Fragment已添加过，则直接show，否则构造一个Fragment
     *
     * @param containerViewId 容器控件id
     * @param clz             Fragment类
     */
    protected void showFragment(@IdRes int containerViewId, Class<? extends
Fragment> clz) {
        FragmentManager fm = getFragmentManager();
        FragmentTransaction ft = fm.beginTransaction();//开始事务管理
        Fragment f;
        if ((f = fm.findFragmentByTag(clz.getName())) == null) {
            try {
                f = clz.newInstance();
                ft.add(containerViewId, f, clz.getName());//添加操作
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else {
            ft.show(f);//添加操作
        }
        ft.commit();//提交事务
    }
```

上面是一个简单的显示Fragment的栗子，简单判断一下Fragment是否已添加过，添加过就直接show，否则构造一个Fragment，最后提交事务。

代码分析

FragmentManager



上图是获取FragmentManager的大体过程

要管理Fragment事务，首先是需要拿到FragmentManager，在Activity中可以通过`getFragmentManager()`方法获取(使用兼容包的话，通过`FragmentActivity###getSupportFragmentManager()`)，在这里我们就不对兼容包进行分析了

```
final FragmentController mFragments = FragmentController.createController(new
HostCallbacks());

    /**
     * Return the FragmentManager for interacting with fragments associated
     * with this activity.
     */
    public FragmentManager getFragmentManager() {
        return mFragments.getFragmentManager();
    }
```

FragmentManager是一个抽象类，它是通过mFragments.getFragmentManager()来获取的，mFragments是FragmentController对象，它通过`FragmentController.createController(new HostCallbacks())`生成，这是一个静态工厂方法：

```
    public static final FragmentController
createController(FragmentHostCallback<?> callbacks) {
        return new FragmentController(callbacks);
    }
```

在这里面直接new了一个FragmentController对象，注意FragmentController的构造方法需要传入一个`FragmentHostCallback`

FragmentController构造方法

```
    private final FragmentHostCallback<?> mHost;
    private FragmentController(FragmentHostCallback<?> callbacks) {
        mHost = callbacks;
    }
```

构造方法很简单，传入了一个FragmentHostCallback实例

FragmentController###getFragmentManager

```
    public FragmentManager getFragmentManager() {
        return mHost.getFragmentManagerImpl();
    }
```

这里又调用了mHost的getFragmentManagerImpl方法，希望童鞋们没有被绕晕，mHost是一个FragmentHostCallback实例，那我们回过头来看看它传进来的地方

FragmentHostCallback

这个FragmentHostCallback是一个抽象类，我们可以看到，在Activity中是传入了`Activity###HostCallbacks`内部类，这个就是FragmentHostCallback的实现类

FragmentHostCallback###getFragmentManagerImpl

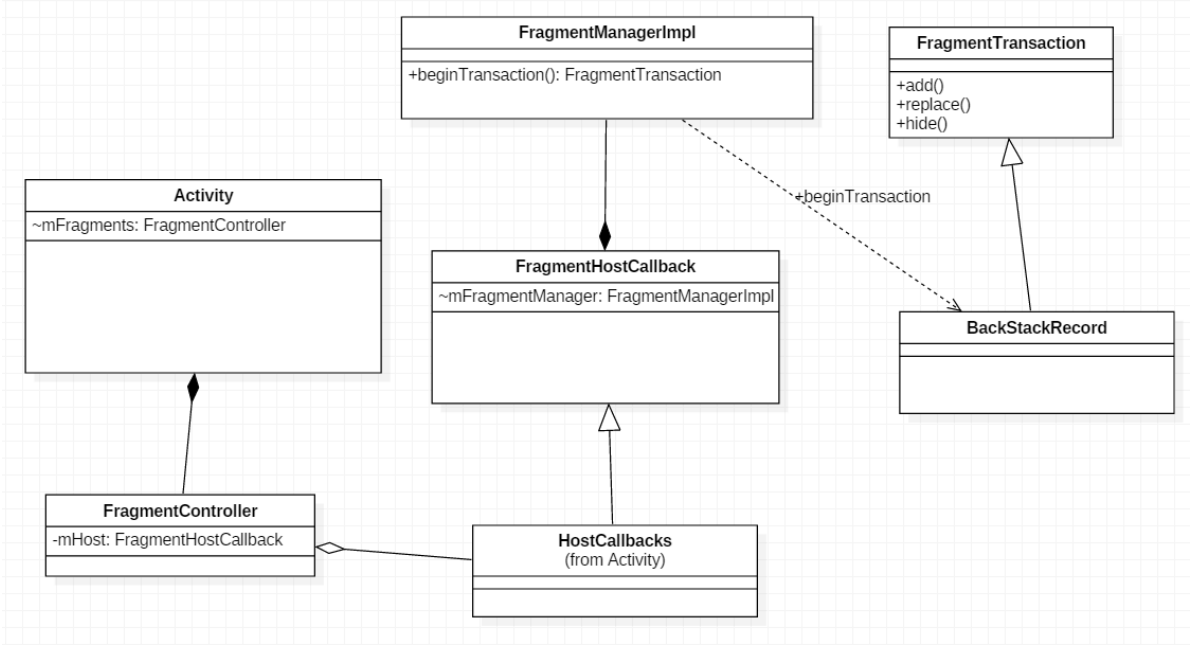
```
    final FragmentManagerImpl mFragmentManager = new FragmentManagerImpl();
    FragmentManagerImpl getFragmentManagerImpl() {
        return mFragmentManager;
    }
```

终于找到FragmentManager的真身`FragmentManagerImpl`了

FragmentManagerImpl###beginTransaction

```
    @Override
    public FragmentTransaction beginTransaction() {
        return new BackStackRecord(this);
    }
```

可以看到，所谓的FragmentTransaction其实就是一个BackStackRecord。到现在，FragmentManager和FragmentTransaction我们都找到了。下图就是各个类之间的关系：



下面开始真正的事务管理分析，我们先选择一个事务add来进行分析

FragmentTransaction###add

```
 public FragmentTransaction add(int containerViewId, Fragment fragment, String
tag) {
        doAddOp(containerViewId, fragment, tag, OP_ADD);
        return this;
    }

    private void doAddOp(int containerViewId, Fragment fragment, String tag, int
opcmd) {

        //设置fragment的FragmentManagerImpl，mManager其实就是
Activity###HostCallbacks中的成员变量
        fragment.mFragmentManager = mManager;

        //设置fragment的tag
        if (tag != null) {
            if (fragment.mTag != null && !tag.equals(fragment.mTag)) {
```

```
                throw new IllegalStateException("...");
            }
            fragment.mTag = tag;
        }

        if (containerViewId != 0) {
            if (containerViewId == View.NO_ID) {
                throw new IllegalArgumentException("...");
            }
            if (fragment.mFragmentId != 0 && fragment.mFragmentId !=
containerViewId) {
                throw new IllegalStateException("");
            }
            //设置fragment的mContainerId以及mFragmentId
            fragment.mContainerId = fragment.mFragmentId = containerViewId;
        }

        //新增一个操作
        Op op = new Op();
        op.cmd = opcmd;
        op.fragment = fragment;
        //添加操作
        addOp(op);
    }

    //插入到链表的最后
    void addOp(Op op) {
        if (mHead == null) {
            mHead = mTail = op;
        } else {
            op.prev = mTail;
            mTail.next = op;
            mTail = op;
        }
        op.enterAnim = mEnterAnim;
        op.exitAnim = mExitAnim;
        op.popEnterAnim = mPopEnterAnim;
        op.popExitAnim = mPopExitAnim;
        mNumOp++;
    }
```

add的操作步骤为：

1. 设置fragment的FragmentManagerImpl
2. 设置fragment的tag
3. 设置fragment的mContainerId以及mFragmentId
4. 插入一个类型为OP_ADD的操作到链表最后

这里用到了一个类：

```
    static final class Op {        Op next;//下一操作节点        Op prev;//上一操作
节点        int cmd;//操作类型，可选有:
OP_NULL|OP_ADD|OP_REPLACE|OP_REMOVE|OP_HIDE|OP_SHOW|OP_DETACH|OP_ATTACH
Fragment fragment;//操作的Fragment对象        int enterAnim;//入场动画        int
exitAnim;//出场动画        int popEnterAnim;//弹入动画        int popExitAnim;//弹出
动画        ArrayList<Fragment> removed;    }
```

这是一个操作链表节点。所有add、remove、hide等事物最终会形成一个操作链

FragmentTransaction###commit

等所有操作都插入后，最后我们需要调用FragmentTransaction的commit方法，操作才会真正地执行。

```
public int commit() {        return commitInternal(false);    }    int
commitInternal(boolean allowStateLoss) {        //防止重复commit        if
(mCommitted) {            throw new IllegalStateException("commit already
called");        }        //DEBUG代码统统不管        if (FragmentManagerImpl.DEBUG)
{            Log.v(TAG, "Commit: " + this);            LogWriter logw = new
LogWriter(Log.VERBOSE, TAG);            PrintWriter pw = new
FastPrintWriter(logw, false, 1024);            dump("  ", null, pw, null);
     pw.flush();        }        mCommitted = true;        //只有调用了
addToBackStack方法之后，这个标记才会为true        if (mAddToBackStack) {
mIndex = mManager.allocBackStackIndex(this);        } else {            mIndex =
-1;        }        //插入事物队列        mManager.enqueueAction(this,
allowStateLoss);        return mIndex;    }
```

FragmentManagerImpl###enqueueAction

```
    /**     * Adds an action to the queue of pending actions.     *     * @param
action the action to add     * @param allowStateLoss whether to allow loss of
state information     * @throws IllegalStateException if the activity has been
destroyed     */    public void enqueueAction(Runnable action, boolean
allowStateLoss) {        if (!allowStateLoss) {            checkStateLoss();
   }        synchronized (this) {            if (mDestroyed || mHost == null) {
            throw new IllegalStateException("Activity has been destroyed");
       }            if (mPendingActions == null) {                mPendingActions
= new ArrayList<Runnable>();            }            mPendingActions.add(action);
           if (mPendingActions.size() == 1) {
mHost.getHandler().removeCallbacks(mExecCommit);
mHost.getHandler().post(mExecCommit);            }        }    }
```

这里把操作添加到`mPendingActions`列表里去。并通过mHost.getHandler()获取Handler发送执行请求。从上面的分析知道，mHost就是Activity的HostCallbacks，构造方法中把Activity的mHandler传进去了，这里执行的`mHost.getHandler()`获取到的也就是Activity中的mHandler，这样做是因为需要在主线程中执行

```
final Handler mHandler = new Handler();
```

再看看mExecCommit中做了什么操作：

```
 Runnable mExecCommit = new Runnable() {        @Override        public void
run() {            execPendingActions();        }    };    /**     * Only call
from main thread!     */    public boolean execPendingActions() {        if
(mExecutingActions) {            throw new IllegalStateException("Recursive entry
to executePendingTransactions");        }        //再次检测是否主线程        if
(Looper.myLooper() != mHost.getHandler().getLooper()) {            throw new
IllegalStateException("Must be called from main thread of process");        }
   boolean didSomething = false;        while (true) {            int numActions;
          synchronized (this) {                //参数检测                if
(mPendingActions == null || mPendingActions.size() == 0) {
break;                }                numActions = mPendingActions.size();
         if (mTmpActions == null || mTmpActions.length < numActions) {
         mTmpActions = new Runnable[numActions];               }
mPendingActions.toArray(mTmpActions);               mPendingActions.clear();
          mHost.getHandler().removeCallbacks(mExecCommit);           }
   mExecutingActions = true;            //遍历执行待处理的事务操作            for
(int i=0; i<numActions; i++) {                mTmpActions[i].run();
mTmpActions[i] = null;            }            mExecutingActions = false;
   didSomething = true;        }        doPendingDeferredStart();        return
didSomething;    }
```

插入了事物之后，就是在主线程中把需要处理的事务统一处理，处理事务是通过执行`mTmpActions[i].run()`进行的，这个mTmpActions[i]就是前面我们通过enqueueAction方法插入的BackStackRecord，童鞋们可能没注意到，它可是一个Runnable，我们来看看它的定义

```
final class BackStackRecord extends FragmentTransaction implements
FragmentManager.BackStackEntry, Runnable {    static final String TAG =
FragmentManagerImpl.TAG; ... ...}
```

兜兜转转，我们又回到了BackStackRecord

BackStackRecord###run

```
 public void run() {        ......        if (mManager.mCurState >=
Fragment.CREATED) {            SparseArray<Fragment> firstOutFragments = new
SparseArray<Fragment>();            SparseArray<Fragment> lastInFragments = new
SparseArray<Fragment>();            calculateFragments(firstOutFragments,
lastInFragments);            beginTransition(firstOutFragments, lastInFragments,
false);        }        //遍历链表，根据cmd事务类型依次处理事务        Op op = mHead;
       while (op != null) {            switch (op.cmd) {                case
OP_ADD: {                    Fragment f = op.fragment;
f.mNextAnim = op.enterAnim;                    mManager.addFragment(f, false);
             }                break;                case OP_REPLACE: {
         Fragment f = op.fragment;                    int containerId =
f.mContainerId;                    if (mManager.mAdded != null) {
       for (int i = mManager.mAdded.size() - 1; i >= 0; i--) {
         Fragment old = mManager.mAdded.get(i);                            if
(old.mContainerId == containerId) {                                if (old == f)
{                                    op.fragment = f = null;
           } else {                                    if (op.removed == null) {
                                    op.removed = new ArrayList<Fragment>();
                                }
op.removed.add(old);                                    old.mNextAnim =
op.exitAnim;                                    if (mAddToBackStack) {
                            old.mBackStackNesting += 1;
           }                                    mManager.removeFragment(old,
mTransition, mTransitionStyle);                                    }
           }                        }                    }                    if
(f != null) {                        f.mNextAnim = op.enterAnim;
       mManager.addFragment(f, false);                    }                }
           break;                case OP_REMOVE: {                    Fragment f
= op.fragment;                    f.mNextAnim = op.exitAnim;
mManager.removeFragment(f, mTransition, mTransitionStyle);                }
         break;                case OP_HIDE: {                    Fragment f =
op.fragment;                    f.mNextAnim = op.exitAnim;
mManager.hideFragment(f, mTransition, mTransitionStyle);                }
       break;                case OP_SHOW: {                    Fragment f =
op.fragment;                    f.mNextAnim = op.enterAnim;
mManager.showFragment(f, mTransition, mTransitionStyle);                }
       break;                case OP_DETACH: {                    Fragment f =
op.fragment;                    f.mNextAnim = op.exitAnim;
mManager.detachFragment(f, mTransition, mTransitionStyle);                }
         break;                case OP_ATTACH: {                    Fragment f =
op.fragment;                    f.mNextAnim = op.enterAnim;
mManager.attachFragment(f, mTransition, mTransitionStyle);                }
         break;                default: {                    throw new
IllegalArgumentException("Unknown cmd: " + op.cmd);                }            }
           op = op.next;        }        mManager.moveToState(mManager.mCurState,
mTransition,                mTransitionStyle, true);        if (mAddToBackStack)
{            mManager.addBackStackState(this);        }    }
```

到这一步，提交的事务就被真正执行了，我们知道，即使commit了事务之后，也不是同步执行的，是通过Handler发送到主线程执行的。

所有事务的处理都是在run方法里面执行，但是我们留意到，想要搞清楚add、remove等事务背后真正做了什么，还需要深入了解FragmentManagerImpl。

(二) Fragment生命周期深入详解

本章为视频讲解，有需要的小伙伴加VX:cnn1050 备注“优化”即可获取！

(三) 性能优化相关方案-Fragment 懒加载的新实现

老的懒加载处理方案

> 如果你熟悉老一套的 Fragment 懒加载机制，你可以直接查看 Androix 懒加载相关章节

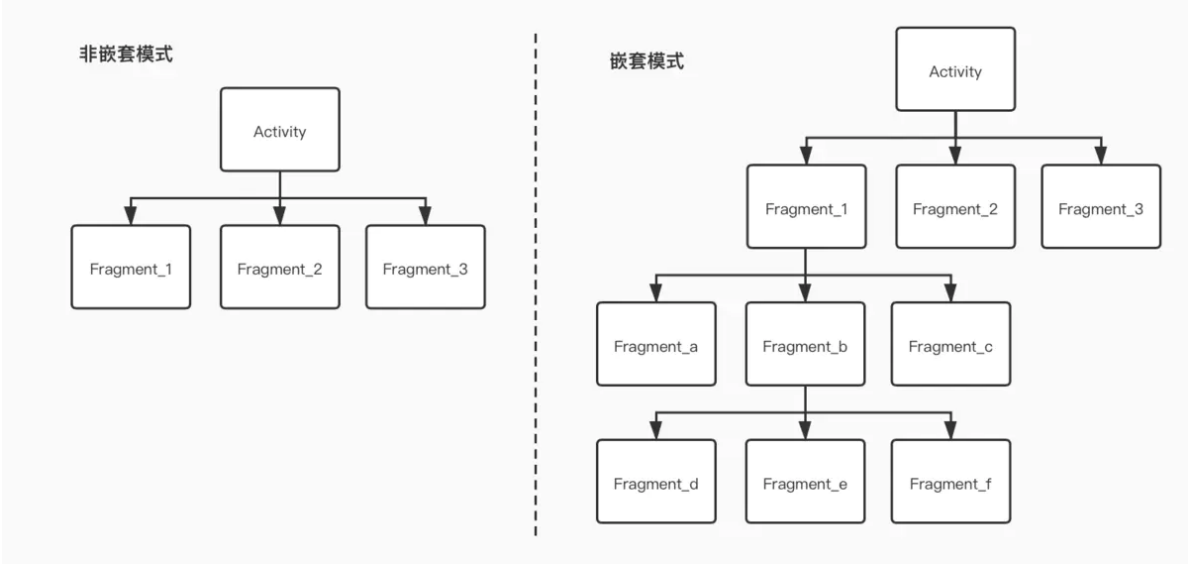
add+show+hide 模式下的老方案

在没有添加懒加载之前，只要使用 `add+show+hide` 的方式控制并显示 Fragment, 那么不管 Fragment 是否嵌套，在初始化后，如果 只调用了`add+show`，同级下的 Fragment 的相关生命周期函数都会被调用。且调用的生命周期函数如下所示：

```
onAttach -> onCreate -> onCreatedView -> onActivityCreated -> onStart ->
onResume
```

> Fragment 完整生命周期：onAttach -> onCreate -> onCreatedView -> onActivityCreated -> onStart -> onResume -> onPause -> onStop -> onDestroyView -> onDestroy -> onDetach

什么是同级 Frament 呢？看下图



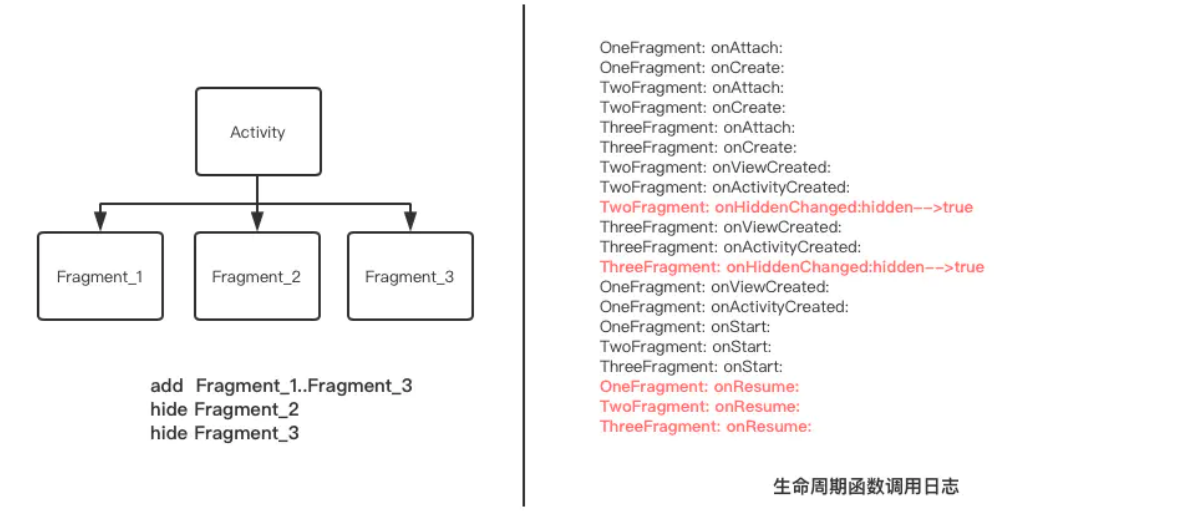
同级Fragment.jpg

> 上图中，都是使用 `add+show+hide` 的方式控制 Fragment,

在上图两种模式中:

- Fragment_1、Fragment_2、Fragment_3 属于同级 Fragment
- Fragment_a、Fragment_b、Fragment_c 属于同级 Fragment
- Fragment_d、Fragment_e、Fragment_f 属于同级 Fragment

那这种方式会带来什么问题呢？结合下图我们来分别分析。



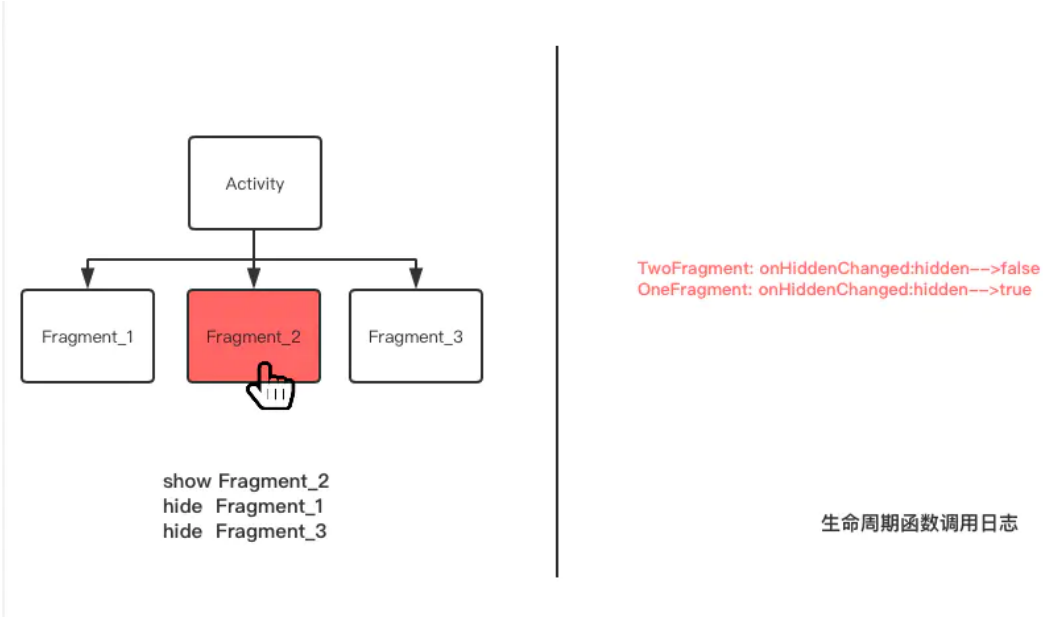
show1.png

观察上图我们可以发现，同级的Fragment_1、Fragment_2、Fragment_3 都调用了 `onAttach...onResume` 系列方法，也就是说，如果我们没有对 Fragment 进行懒加载处理，那么我们就会无缘无故的加载一些并`不可见`的 Fragment , 也就会造成用户流量的无故消耗（我们会在 Fragment 相关生命周期函数中，请求网络或其他数据操作）。

> 这里`"不可见的Fragment"`是指，实际不可见但是相关可见生命周期函数(如 `onResume` 方法）被调用的 Fragment

如果使用嵌套 Fragment ， 这种浪费流量的行为就更明显了。以本节的图一为例，当 Fragment_1 加载时，如果你在 Fragment_1 生命周期函数中使用 `show+add+hide` 的方式添加 `Fragment_a、Fragment_b、Fragment_c` , 那么 Fragment_b 又会在其生命周期函数中继续加载 `Fragment_d、Fragment_e、Fragment_f` 。

那如何解决这种问题呢？我们继续接着上面的例子走，当我们 `show Fragment_2` ，并 hide其他 Fragment 时，对应 Fragment 的生命周期调用如下：



show2.png

从上图中，我们可以看出 Fragment_2 与 Fragment_3 都调用了 `onHiddenChanged` 函数，该函数的官方 API 声明如下：

```
    /**
     * Called when the hidden state (as returned by {@link #isHidden()} of
     * the fragment has changed.  Fragments start out not hidden; this will
     * be called whenever the fragment changes state from that.
     * @param hidden True if the fragment is now hidden, false otherwise.
     */
    public void onHiddenChanged(boolean hidden) {
    }
```

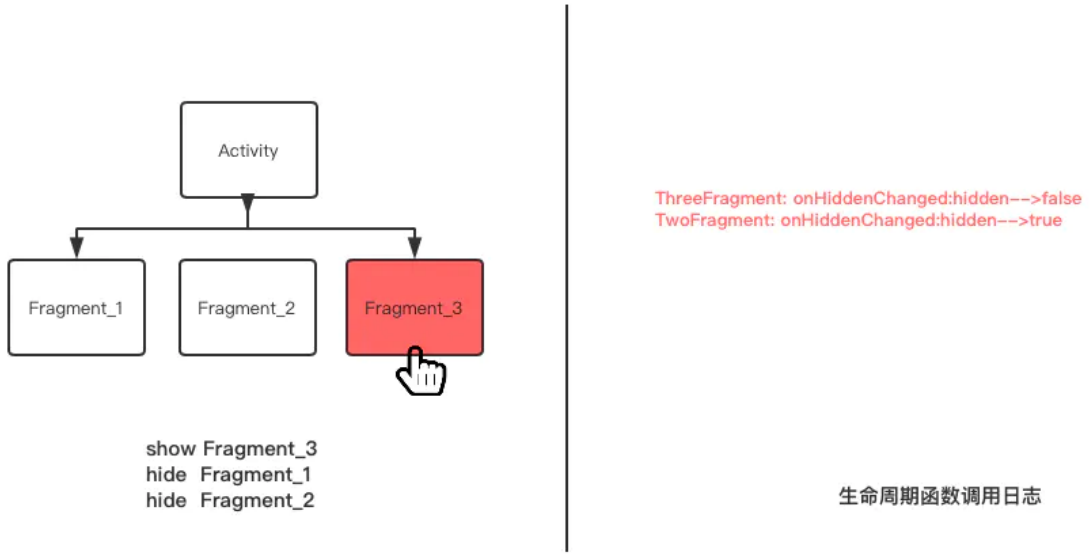
根据官方 API 的注释，我们大概能知道，当 Fragment 隐藏的状态发生改变时，该函数将会被调用，如果当前 Fragment 隐藏， `hidden` 的值为 true, 反之为 false。最为重要的是 `hidden` 的值，可以通过调用 `isHidden()` 函数获取。

那么结合上述知识点，我们能推导出：

- 因为 Fragment_1 的 `隐藏状态` 从 `可见转为了不可见` ，所以其 `onHiddenChanged` 函数被调用，同时 `hidden` 的值为 true。
- 同理对于 Fragment_2 ， 因为其 `隐藏状态` 从 `不可见转为了可见` ， 所以其 hidden 值为 false。
- 对于 Fragment_3 ,因为其隐藏状态从始至终都没有发生变化，所以其 onHiddenChanged 函数并不会调用。

嗯，好像有点眉目了。不急，我们继续看下面的例子。

show Fragment_3 并 hide 其他 Fragment ， 对应生命周期函数调用如下所示：



show3.png

从图中，我们可以看出，确实只有`隐藏状态`发生了改变的 Fragment 其`onHiddenChanged`函数才会调用，那么结合以上知识点，我们能得出如下重要结论：

只要通过`show+hide`方式控制 Fragment 的显隐，那么在第一次初始化后，Fragment 任何的生命周期方法都不会调用，只有`onHiddenChanged`方法会被调用。

那么，假如我们要在`add+show+hide`模式下控制 Fragment 的懒加载，我们只需要做这两步：

- 我们需要在`onResume()`函数中调用`isHidden()`函数，来处理默认显示的 Fragment
- 在`onHiddenChanged`函数中控制其他不可见的Fragment，

也就是这样处理：

```
abstract class LazyFragment:Fragment(){

    private var isLoaded = false //控制是否执行懒加载

    override fun onResume() {
        super.onResume()
        judgeLazyInit()

    }
    override fun onHiddenChanged(hidden: Boolean) {
        super.onHiddenChanged(hidden)
        isVisibleToUser = !hidden
        judgeLazyInit()
    }

    private fun judgeLazyInit() {
        if (!isLoaded && !isHidden) {
            lazyInit()
            isLoaded = true
        }
    }

     override fun onDestroyView() {
        super.onDestroyView()
```

```
            isLoaded = false
        }

        //懒加载方法
        abstract fun lazyInit()
    }
```

> 该懒加载的实现，是在 `onResume` 方法中操作，当然你可以在其他生命周期函数中控制。但是建议在该方法中执行懒加载。

ViewPager+Fragment 模式下的老方案

使用传统方式处理 ViewPager 中 Fragment 的懒加载，我们需要控制 `setUserVisibleHint(boolean isVisibleToUser)` 函数，该函数的声明如下所示：

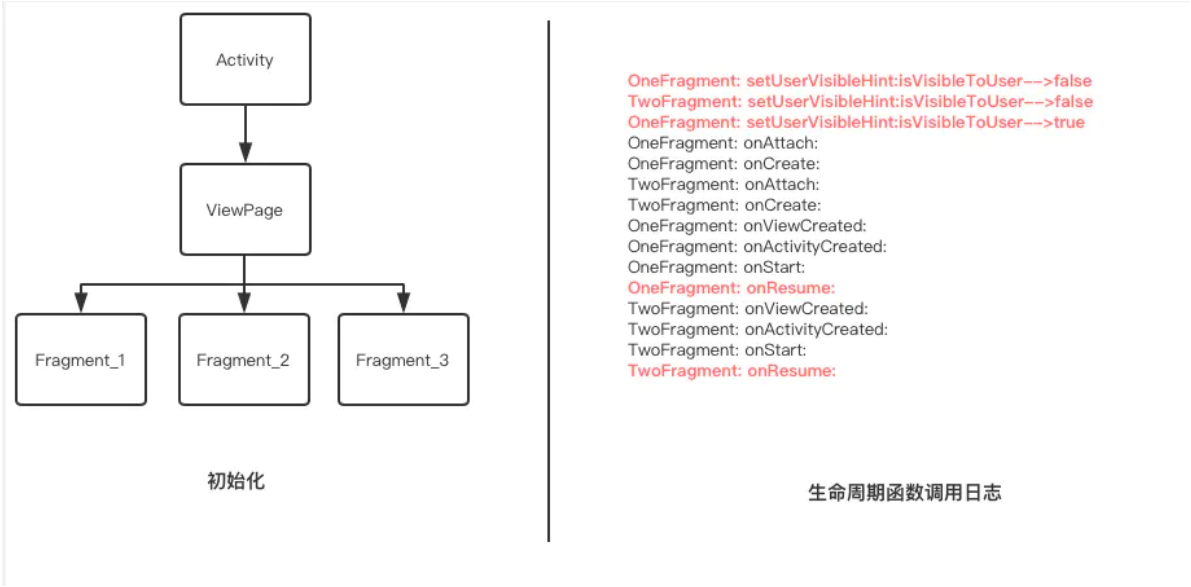
```
    public void setUserVisibleHint(boolean isVisibleToUser) {}
```

该函数与之前我们介绍的 `onHiddenChanged()` 作用非常相似，都是通过传入的参数值来判断当前 Fragment 是否对用户可见，只是 `onHiddenChanged()` 是在 `add+show+hide` 模式下使用，而 `setUserVisibleHint` 是在 ViewPager+Fragment 模式下使用。

在本节中，我们用 `FragmentPagerAdapter + ViewPager` 为例，向大家讲解如何实现 Fragment 的懒加载。

> 注意：在本例中没有调用 `setOffscreenPageLimit` 方法去设置 ViewPager 预缓存的 Fragment 个数。默认情况下 ViewPager 预缓存 Fragment 的个数为 `1` 。

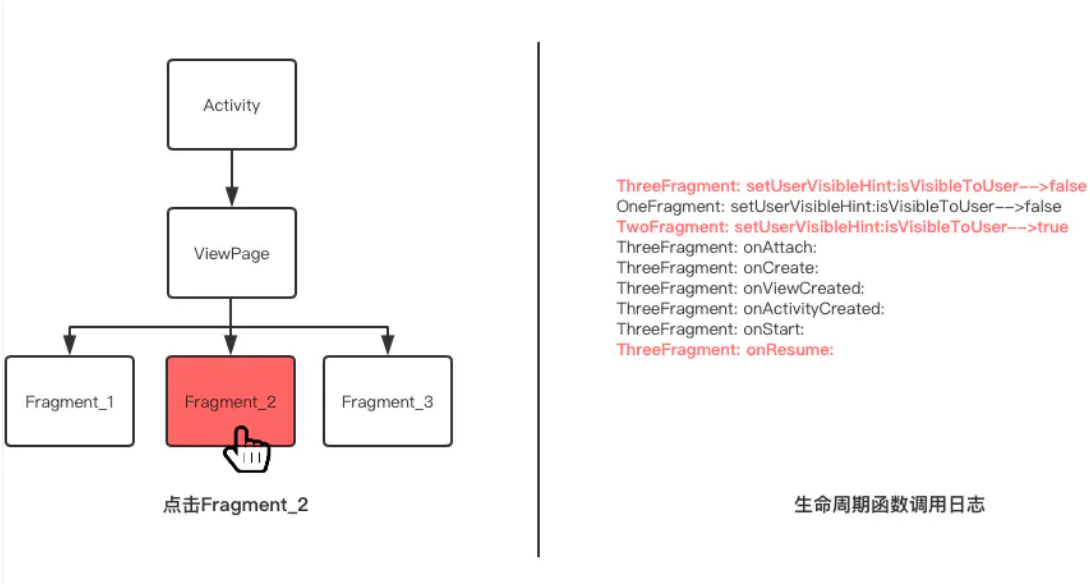
初始化 ViewPager 查看内部 Fragment 生命周期函数调用情况：



viewpager1.png

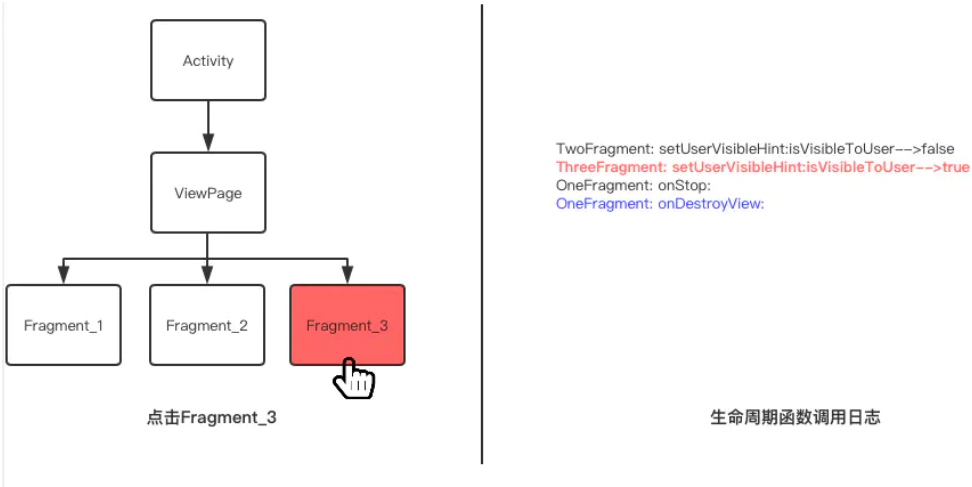
观察上图，我们能发现 ViePager 初始化时，默认会调用其内部 Fragment 的 setUserVisibleHint 方法，因为其预缓存 Fragment 个数为 `1` 的原因，所以只有 Fragment_1 与 Fragment_2 的生命周期函数被调用。

我们继续切换到 Fragment_2，查看各个Fragment的生命周期函数的调用变化。如下图所示：



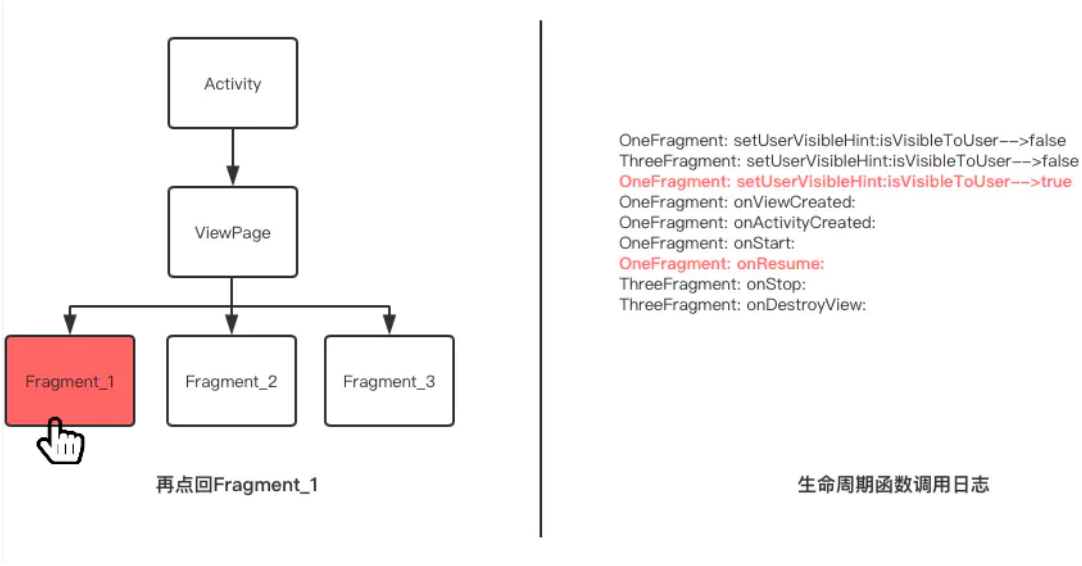
viewpage2.png

观察上图，我们同样发现 Fragment 的 setUserVisibleHint 方法被调用了，并且 Fragment_3 的一系列生命周期函数被调用了。继续切换到 Fragment_3:



viewpager_3.png

观察上图可以发现，Fragment_3 调用了 setUserVisibleHint 方法，继续又切换到 Fragment_1，查看调用函数的变化：



viewpager4.png

> 因为之前在切换到 Fragment_3 时，Frafgment_1 已经走了 onDestoryView(图二，蓝色标记处)方法，所以 Fragment_1 需要重新走一次生命周期。

那么结合本节的三幅图，我们能得出以下结论：

- 使用 ViewPager，切换回上一个 Fragment 页面时（已经初始化完毕），不会回调任何生命周期方法以及onHiddenChanged()，只有 setUserVisibleHint(boolean isVisibleToUser) 会被回调。
- setUserVisibleHint(boolean isVisibleToUser) 方法总是会优先于 Fragment 生命周期函数的调用。

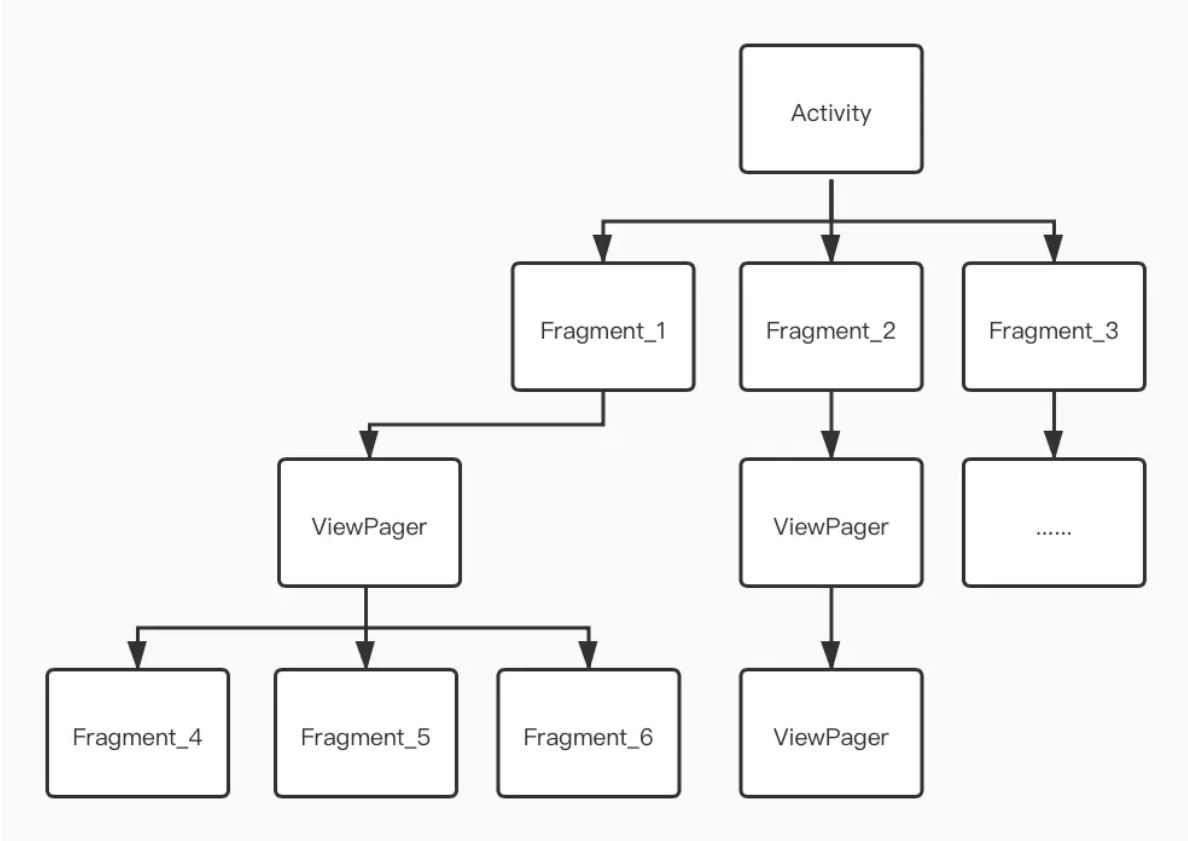
所以如果我们想对 ViewPager 中的 Fragment 懒加载，我们需要这样处理：

```
abstract class LazyFragment : Fragment() {

    /**
     * 是否执行懒加载
     */
    private var isLoaded = false

    /**
     * 当前Fragment是否对用户可见
     */
    private var isVisibleToUser = false

    /**
     * 当使用ViewPager+Fragment形式会调用该方法时，setUserVisibleHint会优先Fragment生命周期函数调用，
     * 所以这个时候就,会导致在setUserVisibleHint方法执行时就执行了懒加载，
     * 而不是在onResume方法实际调用的时候执行懒加载。所以需要这个变量
     */
    private var isCallResume = false

    override fun onResume() {
        super.onResume()
        isCallResume = true
        judgeLazyInit()
    }


    private fun judgeLazyInit() {
        if (!isLoaded && isVisibleToUser && isCallResume) {
            lazyInit()
            Log.d(TAG, "lazyInit:!!!!!!!")
            isLoaded = true
        }
    }

    override fun onHiddenChanged(hidden: Boolean) {
        super.onHiddenChanged(hidden)
        isVisibleToUser = !hidden
        judgeLazyInit()
    }

    //在Fragment销毁View的时候，重置状态
    override fun onDestroyView() {
        super.onDestroyView()
```

```
        isLoaded = false
        isVisibleToUser = false
        isCallResume = false
    }

    override fun setUserVisibleHint(isVisibleToUser: Boolean) {
        super.setUserVisibleHint(isVisibleToUser)
        this.isVisibleToUser = isVisibleToUser
        judgeLazyInit()
    }

    abstract fun lazyInit()
}
```

复杂 Fragment 嵌套的情况

当然，在实际项目中，我们可能会遇到更为复杂的 Fragment 嵌套组合。比如 Fragment+Fragment、Fragment+ViewPager、ViewPager+ViewPager....等等。
如下图所示：



复杂嵌套Fragment.jpg

对于以上场景，我们就需要重写我们的懒加载，以支持不同嵌套组合模式下 Fragment 正确懒加载。我们需要将 LazyFragment 修改成如下这样：

```
abstract class LazyFragment : Fragment() {

    /**
     * 是否执行懒加载
     */
    private var isLoaded = false
```

```
    /**
     * 当前Fragment是否对用户可见
     */
    private var isVisibleToUser = false

    /**
     * 当使用ViewPager+Fragment形式会调用该方法时，setUserVisibleHint会优先Fragment生命周期函数调用，
     * 所以这个时候就,会导致在setUserVisibleHint方法执行时就执行了懒加载，
     * 而不是在onResume方法实际调用的时候执行懒加载。所以需要这个变量
     */
    private var isCallResume = false

    /**
     * 是否调用了setUserVisibleHint方法。处理show+add+hide模式下，默认可见 Fragment 不调用
     * onHiddenChanged 方法，进而不执行懒加载方法的问题。
     */
    private var isCallUserVisibleHint = false

    override fun onResume() {
        super.onResume()
        isCallResume = true
        if (!isCallUserVisibleHint) isVisibleToUser = !isHidden
        judgeLazyInit()
    }


    private fun judgeLazyInit() {
        if (!isLoaded && isVisibleToUser && isCallResume) {
            lazyInit()
            Log.d(TAG, "lazyInit:!!!!!!!")
            isLoaded = true
        }
    }

    override fun onHiddenChanged(hidden: Boolean) {
        super.onHiddenChanged(hidden)
        isVisibleToUser = !hidden
        judgeLazyInit()
    }

    override fun onDestroyView() {
        super.onDestroyView()
        isLoaded = false
        isVisibleToUser = false
        isCallUserVisibleHint = false
        isCallResume = false
    }

    override fun setUserVisibleHint(isVisibleToUser: Boolean) {
        super.setUserVisibleHint(isVisibleToUser)
        this.isVisibleToUser = isVisibleToUser
        isCallUserVisibleHint = true
        judgeLazyInit()
    }

    abstract fun lazyInit()
```

```
}
```

Androidx 下的懒加载

虽然之前的方案就能解决轻松的解决 Fragment 的懒加载，但这套方案有一个最大的弊端，`就是不可见的 Fragment 执行了 onResume() 方法`。onResume 方法设计的初衷，难道不是当前 Fragment 可以和用户进行交互吗？你他妈既不可见，又不能和用户进行交互，你执行 onResume 方法干嘛？

基于此问题，Google 在 Androidx 在 `FragmentTransaction` 中增加了 `setMaxLifecycle` 方法来控制 Fragment 所能调用的最大的生命周期函数。如下所示：

```
   /**
     * Set a ceiling for the state of an active fragment in this FragmentManager.
If fragment is
     * already above the received state, it will be forced down to the correct
state.
     *
     * <p>The fragment provided must currently be added to the FragmentManager to
have it's
     * Lifecycle state capped, or previously added as part of this transaction.
The
     * {@link Lifecycle.State} passed in must at least be {@link
Lifecycle.State#CREATED}, otherwise
     * an {@link IllegalArgumentException} will be thrown.</p>
     *
     * @param fragment the fragment to have it's state capped.
     * @param state the ceiling state for the fragment.
     * @return the same FragmentTransaction instance
     */
    @NonNull
    public FragmentTransaction setMaxLifecycle(@NonNull Fragment fragment,
            @NonNull Lifecycle.State state) {
        addOp(new Op(OP_SET_MAX_LIFECYCLE, fragment, state));
        return this;
    }
```

根据官方的注释，我们能知道，该方法可以设置活跃状态下 Fragment 最大的状态，如果该 Fragment 超过了设置的最大状态，那么会强制将 Fragment 降级到正确的状态。

那如何使用该方法呢？我们先看该方法在 Androidx 模式下 ViewPager+Fragment 模式下的使用例子。

ViewPager+Fragment 模式下的方案

在 FragmentPagerAdapter 与 FragmentStatePagerAdapter 新增了含有 `behavior` 字段的构造函数，如下所示：

```
  public FragmentPagerAdapter(@NonNull FragmentManager fm,
          @Behavior int behavior) {
      mFragmentManager = fm;
      mBehavior = behavior;
  }

  public FragmentStatePagerAdapter(@NonNull FragmentManager fm,
          @Behavior int behavior) {
      mFragmentManager = fm;
      mBehavior = behavior;
  }
```

其中 Behavior 的声明如下：

```
   @Retention(RetentionPolicy.SOURCE)
    @IntDef({BEHAVIOR_SET_USER_VISIBLE_HINT,
BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT})
    private @interface Behavior { }

     /**
     * Indicates that {@link Fragment#setUserVisibleHint(boolean)} will be called
when the current
     * fragment changes.
     *
     * @deprecated This behavior relies on the deprecated
     * {@link Fragment#setUserVisibleHint(boolean)} API. Use
     * {@link #BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT} to switch to its
replacement,
     * {@link FragmentTransaction#setMaxLifecycle}.
     * @see #FragmentPagerAdapter(FragmentManager, int)
     */
    @Deprecated
    public static final int BEHAVIOR_SET_USER_VISIBLE_HINT = 0;

    /**
     * Indicates that only the current fragment will be in the {@link
Lifecycle.State#RESUMED}
     * state. All other Fragments are capped at {@link Lifecycle.State#STARTED}.
     *
     * @see #FragmentPagerAdapter(FragmentManager, int)
     */
    public static final int BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT = 1;
```

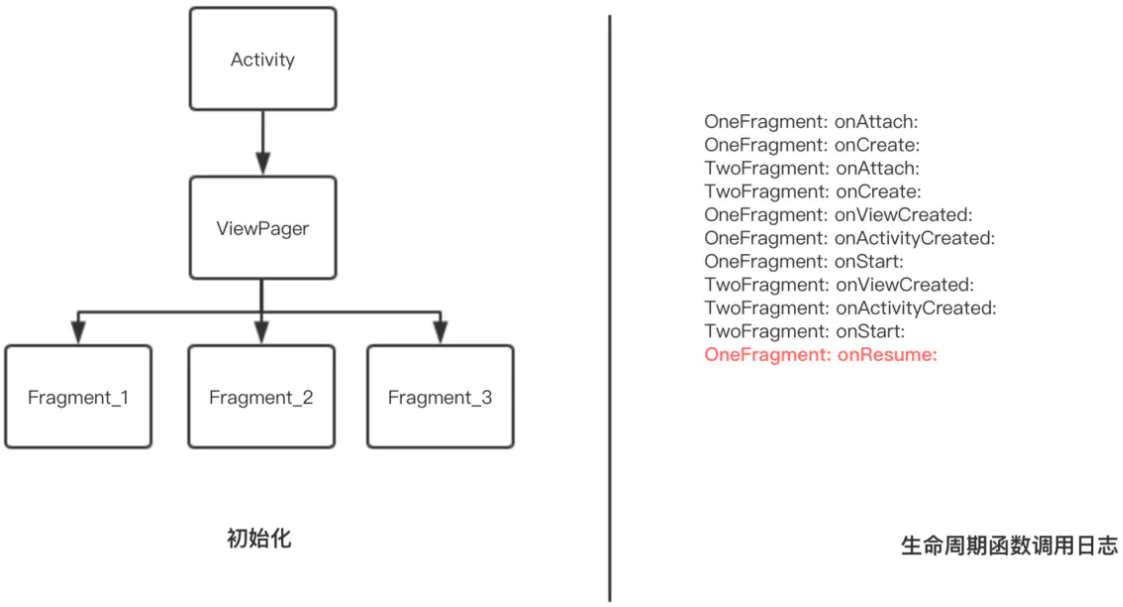
从官方的注释声明中，我们能得到如下两条结论：

- 如果 behavior 的值为 `BEHAVIOR_SET_USER_VISIBLE_HINT`，那么当 Fragment 对用户的可见状态发生改变时，`setUserVisibleHint` 方法会被调用。
- 如果 behavior 的值为 `BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT`，那么当前选中的 Fragment 在 `Lifecycle.State#RESUMED` 状态，其他不可见的 Fragment 会被限制在 `Lifecycle.State#STARTED` 状态。

那 BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT 这个值到底有什么作用呢？我们看下面的例子：

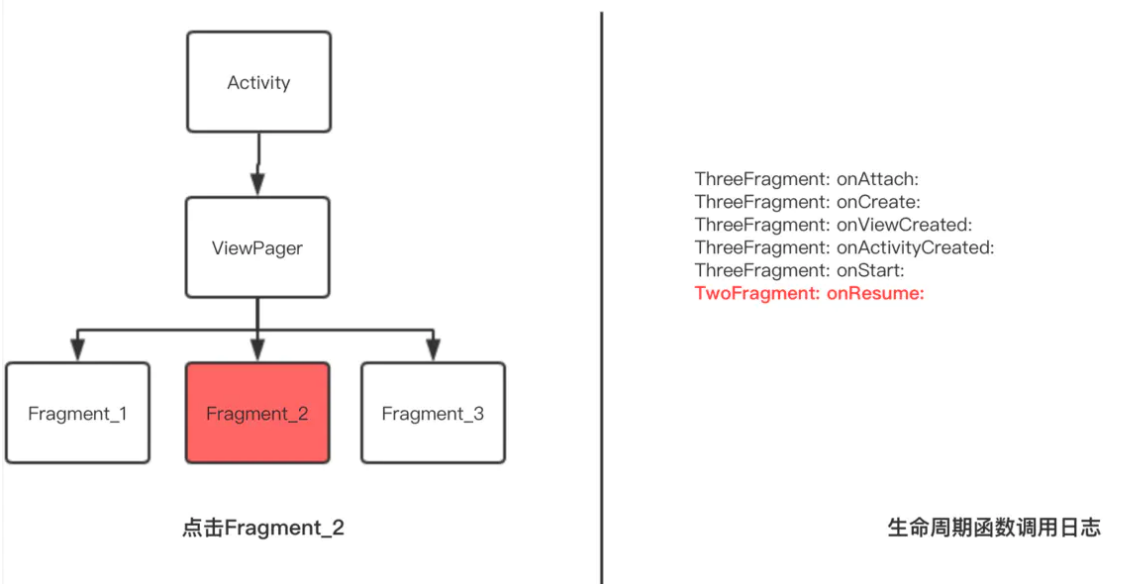
> 在该例子中设置了 ViewPager 的适配器为 FragmentPagerAdapter 且 behavior 值为 `BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT`。

默认初始化ViewPager，Fragment 生命周期如下所示：



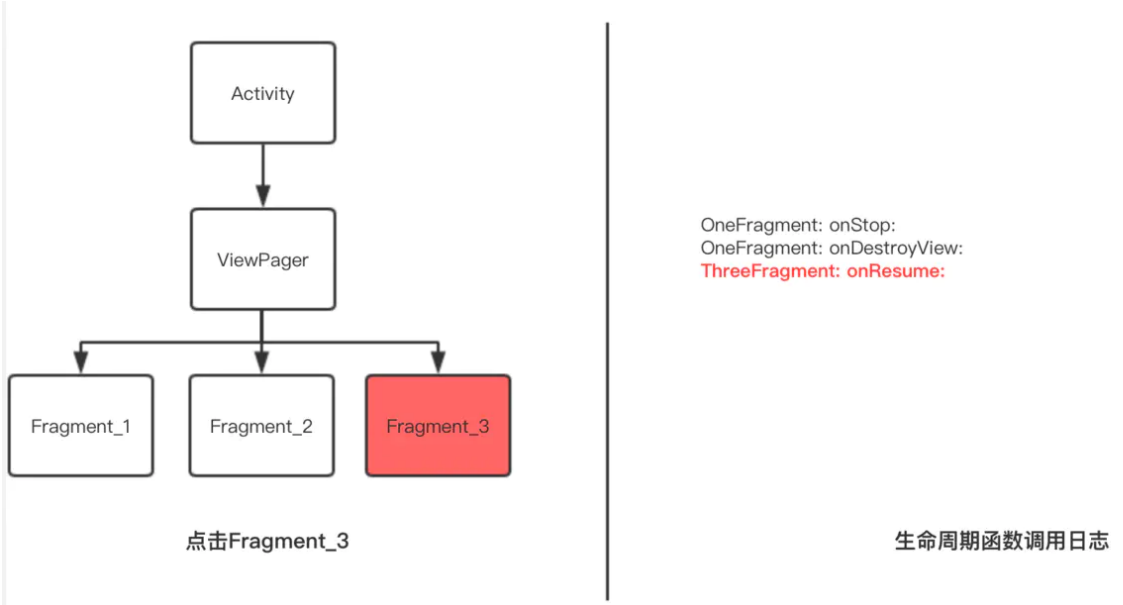
androix1.png

切换到 Fragment_2 时，日志情况如下所示：



androix2.png

切换到 Fragment_3 时，日志情况如下所示:



androidx3.png

> 因为篇幅的原因，本文没有在讲解 FragmentStatePagerAdapter 设置 behavior 下的使用情况，但是原理以及生命周期函数调用情况一样，感兴趣的小伙伴，可以根据 AndroidxLazyLoad 项目自行测试。

观察上述例子，我们可以发现，使用了 `BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT` 后，确实只有当前可见的 Fragment 调用了 onResume 方法。而导致产生这种改变的原因，是因为 FragmentPagerAdapter 在其 `setPrimaryItem` 方法中调用了 `setMaxLifecycle` 方法，如下所示：

```
    public void setPrimaryItem(@NonNull ViewGroup container, int position,
@NonNull Object object) {
        Fragment fragment = (Fragment)object;
        //如果当前的fragment不是当前选中并可见的Fragment,那么就会调用
        // setMaxLifecycle 设置其最大生命周期为 Lifecycle.State.STARTED
        if (fragment != mCurrentPrimaryItem) {
            if (mCurrentPrimaryItem != null) {
                mCurrentPrimaryItem.setMenuVisibility(false);
                if (mBehavior == BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT) {
                    if (mCurTransaction == null) {
                        mCurTransaction = mFragmentManager.beginTransaction();
                    }
                    mCurTransaction.setMaxLifecycle(mCurrentPrimaryItem,
Lifecycle.State.STARTED);
                } else {
                    mCurrentPrimaryItem.setUserVisibleHint(false);
                }
            }
        //对于其他非可见的Fragment,则设置其最大生命周期为
        //Lifecycle.State.RESUMED
            fragment.setMenuVisibility(true);
            if (mBehavior == BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT) {
                if (mCurTransaction == null) {
                    mCurTransaction = mFragmentManager.beginTransaction();
                }
                mCurTransaction.setMaxLifecycle(fragment,
Lifecycle.State.RESUMED);
            } else {
```

```
                fragment.setUserVisibleHint(true);
            }

            mCurrentPrimaryItem = fragment;
        }
    }
```

既然在上述条件下，只有实际可见的 Fragment 会调用 onResume 方法， 那是不是为我们提供了 ViewPager 下实现懒加载的新思路呢？也就是我们可以这样实现 Fragment 的懒加载：

```
abstract class LazyFragment : Fragment() {

    private var isLoaded = false

    override fun onResume() {
        super.onResume()
        if (!isLoaded) {
            lazyInit()
            Log.d(TAG, "lazyInit:!!!!!!!")
            isLoaded = true
        }
    }

    override fun onDestroyView() {
        super.onDestroyView()
        isLoaded = false
    }

    abstract fun lazyInit()
}
```

add+show+hide 模式下的新方案

虽然我们实现了Androidx 包下 ViewPager下的懒加载，但是我们仍然要考虑 add+show+hide 模式下的 Fragment 懒加载的情况，基于 ViewPager 在 `setPrimaryItem` 方法中的思路，我们可以在调用 add+show+hide 时，这样处理：

> 完整的代码请点击--->ShowHideExt

```
/**
 * 使用add+show+hide模式加载fragment
 *
 * 默认显示位置[showPosition]的Fragment，最大Lifecycle为Lifecycle.State.RESUMED
 * 其他隐藏的Fragment，最大Lifecycle为Lifecycle.State.STARTED
 *
 *@param containerViewId 容器id
 *@param showPosition  fragments
 *@param fragmentManager FragmentManager
 *@param fragments  控制显示的Fragments
 */
private fun loadFragmentsTransaction(
    @IdRes containerViewId: Int,
    showPosition: Int,
```

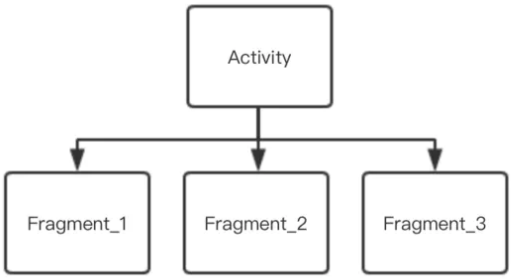
```
    fragmentManager: FragmentManager,
    vararg fragments: Fragment
) {
    if (fragments.isNotEmpty()) {
        fragmentManager.beginTransaction().apply {
            for (index in fragments.indices) {
                val fragment = fragments[index]
                add(containerViewId, fragment, fragment.javaClass.name)
                if (showPosition == index) {
                    setMaxLifecycle(fragment, Lifecycle.State.RESUMED)
                } else {
                    hide(fragment)
                    setMaxLifecycle(fragment, Lifecycle.State.STARTED)
                }
            }

        }.commit()
    } else {
        throw IllegalStateException(
            "fragments must not empty”
        )
    }
}

/** 显示需要显示的Fragment[showFragment]，并设置其最大Lifecycle为
Lifecycle.State.RESUMED。
 *  同时隐藏其他Fragment,并设置最大Lifecycle为Lifecycle.State.STARTED
 * @param fragmentManager
 * @param showFragment
 */
private fun showHideFragmentTransaction(fragmentManager: FragmentManager,
showFragment: Fragment) {
    fragmentManager.beginTransaction().apply {
        show(showFragment)
        setMaxLifecycle(showFragment, Lifecycle.State.RESUMED)

        //获取其中所有的fragment,其他的fragment进行隐藏
        val fragments = fragmentManager.fragments
        for (fragment in fragments) {
            if (fragment != showFragment) {
                hide(fragment)
                setMaxLifecycle(fragment, Lifecycle.State.STARTED)
            }
        }
    }.commit()
}
```

上述代码的实现也非常简单：

- 将需要显示的 Fragment ，在调用 add 或 show 方法后，`setMaxLifecycle(showFragment, Lifecycle.State.RESUMED)`
- 将需要隐藏的 Fragment ，在调用 hide 方法后，`setMaxLifecycle(fragment, Lifecycle.State.STARTED)`

结合上述操作模式，查看使用 setMaxLifecycle 后，Fragment 生命周期函数调用的情况。

add Fragment_1、Fragment_2、Fragment_3，并 hide Fragment_2,Fragment_3
：



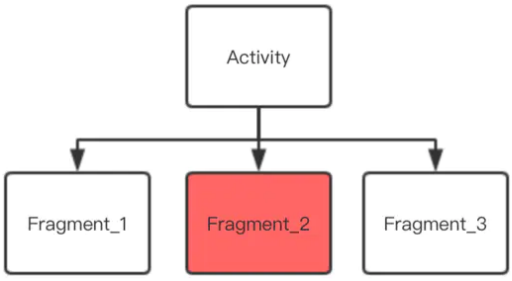
add Fragment_1...Fragment3
hide Fragment_2
hide Fragment_3

OneFragment: onAttach:
OneFragment: onCreate:
TwoFragment: onAttach:
TwoFragment: onCreate:
ThreeFragment: onAttach:
ThreeFragment: onCreate:
OneFragment: onViewCreated:
OneFragment: onActivityCreated:
TwoFragment: onViewCreated:
TwoFragment: onActivityCreated:
TwoFragment: onHiddenChanged:hidden-->true
ThreeFragment: onViewCreated:
ThreeFragment: onActivityCreated:
ThreeFragment: onHiddenChanged:hidden-->true
OneFragment: onStart:
TwoFragment: onStart:
ThreeFragment: onStart:
OneFragment: onResume:

生命周期函数调用日志

show_new1.png

show Fragment_2，hide 其他 Fragment:



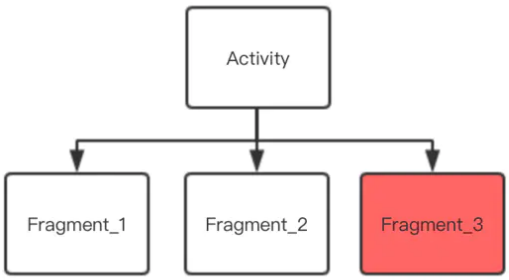
show Fragment_2
hide Fragment_1
hide Fragment_3

TwoFragment: onHiddenChanged:hidden-->false
TwoFragment: onResume:
OneFragment: onHiddenChanged:hidden-->true

生命周期函数调用日志

show_new2.png

show Fragment_3 hide 其他 Fragment:



show Fragment_3
hide Fragment_1
hide Fragment_2

ThreeFragment: onHiddenChanged:hidden-->false
ThreeFragment: onResume:
TwoFragment: onHiddenChanged:hidden-->true

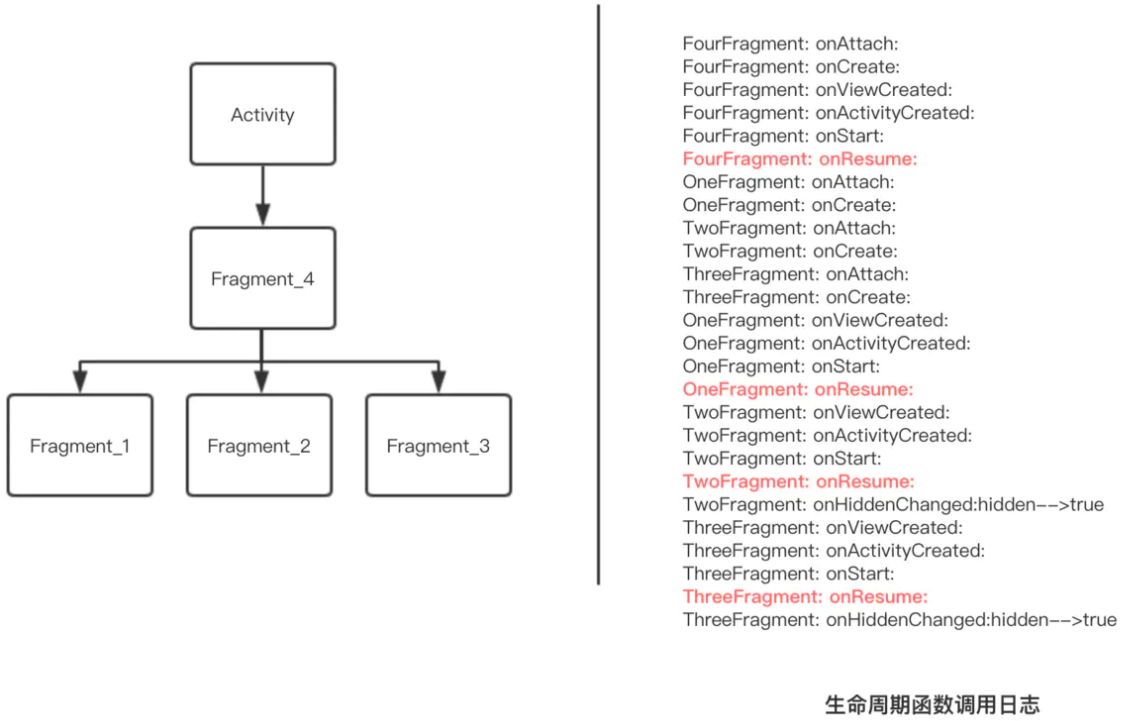
生命周期函数调用日志

show_new3.png

参考上图，好像真的也能处理懒加载！！！！！美滋滋

并不完美的 setMaxLifecycle

当我第一次使用 setMaxLifycycle 方法时，我也和大家一样觉得万事大吉。但这套方案仍然有点点瑕疵，当 Fragment 的嵌套时，即使使用了 setMaxLifycycle 方法，第一次初始化时，同级不可见的 Fragment，仍然 TMD 要调用可见生命周期方法。看下面的例子：



生命周期函数调用日志

瑕疵.png

不知道是否是谷歌大大没有考虑到 Fragment 嵌套的情况，所以这里我们要对之前的方案就行修改，也就是如下所示：

```
abstract class LazyFragment : Fragment() {

    private var isLoaded = false

    override fun onResume() {
        super.onResume()
        //增加了Fragment是否可见的判断
        if (!isLoaded && !isHidden) {
            lazyInit()
            Log.d(TAG, "lazyInit:!!!!!!!")
            isLoaded = true
        }
    }

    override fun onDestroyView() {
        super.onDestroyView()
        isLoaded = false
    }

    abstract fun lazyInit()
```

```
}
```

在上述代码中，因为同级的 Fragment 在嵌套模式下，仍然要调用 onResume 方法，所以我们增加了 Fragment 可见性的判断，这样就能保证嵌套模式下，新方案也能完美的支持 Fragment 的懒加载。

ViewPager2 的处理方案

ViewPager2 本身就支持对实际可见的 Fragment 才调用 onResume 方法。关于 ViewPager2 的内部机制。感兴趣的小伙伴可以自行查看源码。

关于 ViewPager2 的懒加载测试，已上传至 AndroidxLazyLoad，大家可以结合项目查看Log日志。

两种方式的对比与总结

老一套的懒加载

- 优点：不用去控制 FragmentManager的 add+show+hide 方法，所有的懒加载都是在Fragment 内部控制，也就是控制 `setUserVisibleHint + onHiddenChanged` 这两个函数。
- 缺点：实际不可见的 Fragment，其 `onResume` 方法任然会被调用，这种反常规的逻辑，无法容忍。

新一套的懒加载（Androidx下setMaxLifecycle）

- 优点：`在非特殊的情况下(缺点1)`，只有实际的可见 Fragment，其 `onResume` 方法才会被调用，这样才符合方法设计的初衷。
- 缺点：
    1. 对于 Fragment 的嵌套，及时使用了 `setMaxLifecycle` 方法。同级不可见的Fragment， 仍然要调用 `onResume` 方法。
    2. 需要在原有的 add+show+hide 方法中，继续调用 setMaxLifecycle 方法来控制Fragment 的最大生命状态。

最后

这两种方案的优缺点已经非常明显了，到底该选择何种懒加载模式，还是要基于大家的意愿，作者我更倾向于使用新的方案。

第三章: Service深度解析

一、前言

Service对于广大安卓开发者来说算是耳熟能详的，作为安卓四大组件之一，应用非常广泛，本文将全面总结Service定义、分类及使用，同时解析一些常见问题，如与Thread的区别，如何保证不被系统杀死等。

常见问题：

1、Service的定义及作用?
2、谈一谈Service的生命周期?
3、Service的两种启动方式有什么区别?
4、Service有哪些分类及其应用场景?
5、IntentService的作用，与Service的区别?
6、为什么使用Servie，而不是使用Thread?
7、如何保证Servie不被杀死，有哪些思路?
8、Service能否开启耗时操作？要怎么做?

9、用过哪些系统的Service?
10、定义过哪些Service？作用？能否用Thread实现?
11、使用Service要注意哪些问题，有什么适配问题?

二、Service的定义及作用

定义：一个一种可以在后台执行长时间运行操作而没有用户界面的应用组件。服务可由其他应用组件启动（如Activity），服务一旦被启动将在后台一直运行，即使启动服务的组件（Activity）已销毁也不受影响（startService启动方式）。 此外，组件可以绑定到服务，以与之进行交互，甚至是执行进程间通信 (IPC)。
作用：处理复杂计算、下载文件、网络事务、播放音乐，执行文件 I/O 或与内容提供程序交互。
特点：无用户界面、在后台运行、生命周期长。

> Service运行在主线程，所以不能执行耗时操作，如果需要执行耗时处理，可以启动一个工作线程处理。

三、启动方式及生命周期

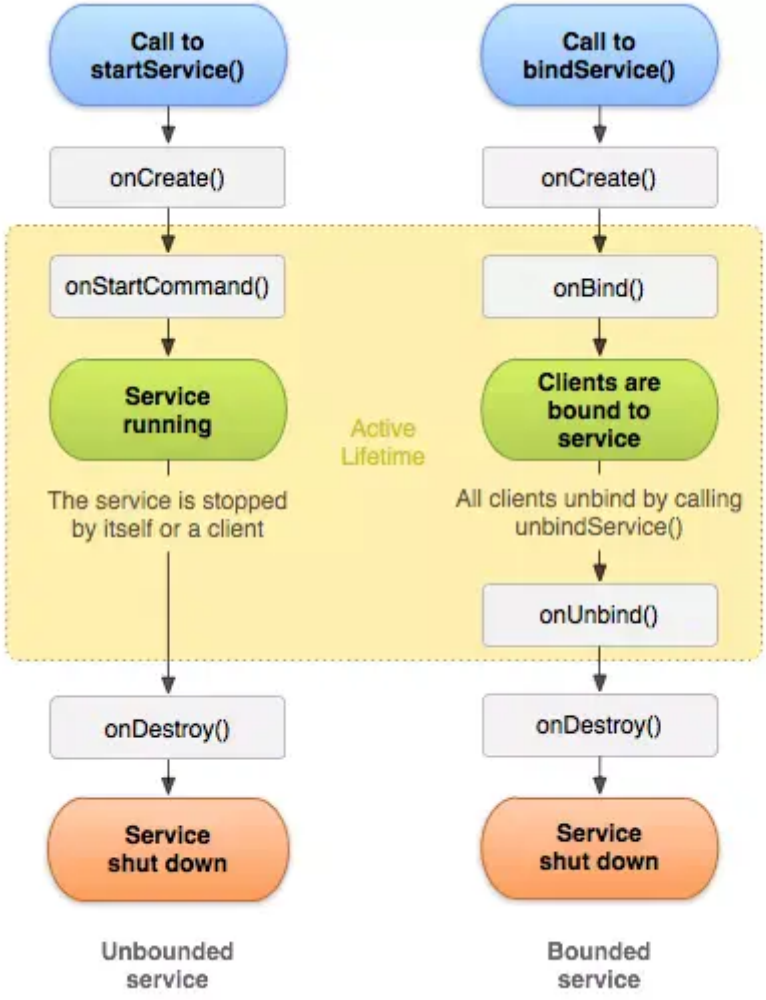
3.1、Service的启动方式

- **startService**：通过startService启动Service后，Service即处于“启动”状态，且在后台无限期运行，即使启动的组件（如Activity）已销毁也不受影响，除非手动调用停止服务，其生命周期与启动组件的生命周期无关。
- **bindService**：通过bindService启动Service后，Service即处于“绑定”状态，绑定的组件与Service可以进行交互、发送和获取数据、甚至可以跨进程（IPC）通讯，绑定服务的生命周期与绑定的组件有关：

> 多个组件可以绑定到同一个服务上，如果只有一个组件绑定服务，当绑定的组件被销毁时，服务也就会停止了。如果是多个组件绑定到一个服务上，当绑定到该服务的所有组件都被销毁时，服务才会停止。

3.2、生命周期

官方图：



生命周期

3.3、startService的生命周期分析

- onCreate：首次启动服务，再次启动不再执行，做一些初始化的操作，比如要执行耗时的操作，可以在这里创建线程，要播放音乐，可以在这里初始化音乐播放器
- onStartCommand：通过startService启动服务，onCreate之后执行， 再次启动直接执行，可以通过startService，设置指定action，通知Service执行特定任务
- onDestroy：服务不再使用，调用stopService或者unbindService之后触发，清理资源，退出线程、注销广播接收器等

3.4、bindService的生命周期分析

- onCreate：首次绑定服务。做一些初始化的操作，比如要执行耗时的操作，可以在这里创建线程，要播放音乐，可以在这里初始化音乐播放器
- onBind：onCreate之后执行，其他组件再次绑定直接执行onBind。必须实现并返回IBinder接口实例，供绑定的组件与服务通讯
- onUnbind：当所有与该服务绑定的组件解除绑定触发。可以执行解绑后的一些事务
- onDestroy：服务不再使用，调用stopService或者unbindService之后触发。清理资源，退出线程、注销广播接收器等

ServiceConnection

绑定服务需要调用下面的方法：

```
    @Override
    public boolean bindService(Intent service, ServiceConnection conn,
            int flags) {
        return mBase.bindService(service, conn, flags);
    }
```

解绑服务调用的方法：

```
    @Override
    public void unbindService(ServiceConnection conn) {
        mBase.unbindService(conn);
    }
```

绑定和解绑都必须参数ServiceConnection，是一个接口类，需要实现该接口，在接口回调中，能获取Binder对象，与服务通讯。

```
private CustomService.MyBinder mMyBinder ;

  // 绑定/解除绑定 Service 回调接口
    private ServiceConnection mConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            // 绑定成功后回调
            //1 ,获取Binder接口对象
            mMyBinder = (CustomService.MyBinder) service;
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {
           // 解除绑定后回调
            mMyBinder = null;
        }
    };
```

如果ServiceConnection传入空，会抛异常：

```
E/AndroidRuntime: FATAL EXCEPTION: main
Process: xxx, PID: 30960
java.lang.IllegalArgumentException: connection is null
    at android.app.ContextImpl.bindServiceCommon(ContextImpl.java:1782)
    at android.app.ContextImpl.bindService(ContextImpl.java:1737)
    at android.content.ContextWrapper.bindService(ContextWrapper.java:678)
```

3.5、onStartCommand 返回值

返回值已经定义在了 Service 基类中了，常用的有：

- START_NOT_STICKY：如果系统在 onStartCommand 返回后终止服务，则除非有挂起 Intent 要传递，否则系统不会重建服务。这是最安全的选项，可以避免在不必要时以及应用能够轻松重启所有

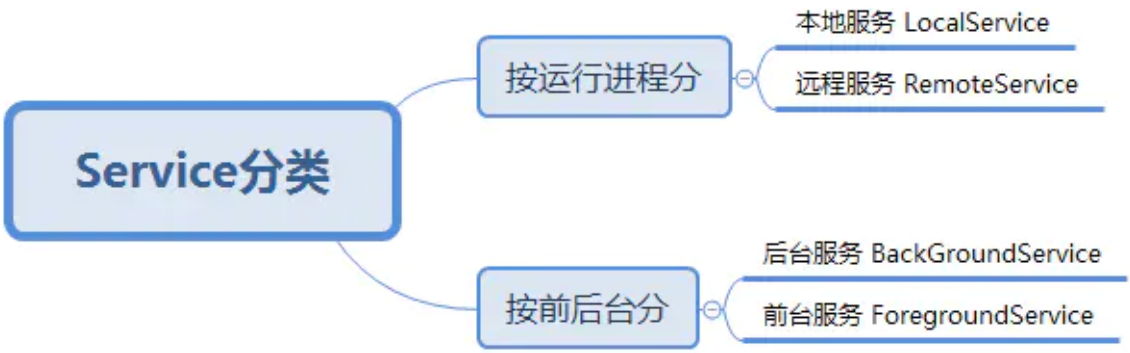
未完成的作业时运行服务。

- START_STICKT：如果系统在 onStartCommand 返回后终止服务，则会重建服务并调用 onStartCommand，但绝对不会重新传递最后一个 Intent。相反，除非有挂起 Intent 要启动服务，否则系统会通过空 Intent 调用 onStartCommand。这适用于不执行命令、但无限期运行并等待作业的媒体播放器等。
- START_REDELIVER_INTENT：如果系统在 onStartCommand 返回后终止服务，则会重建服务，并通过传递给服务等最后一个 Intent 调用 onStartCommand。任何挂起 Intent 均依次传递。这适用于主动执行应该立即恢复的作业的服务，例如下载文件。

四、Service的分类

4.1、Service分类及应用场景

Service可按照运行地点、运行类型 & 功能进行分类，具体如下：



Service分类

- 本地服务：运行在主进程中，主进程退出后，服务也终止，依附主进程的场景，不需要IPC，如音乐播放、下载等
- 远程服务：运行在独立进程中，不受调用进程的影响，需要IPC通讯，系统级别服务
- 前台服务：会在通知栏显示，客户能看到，进程优先级高，不会因内存不足被杀死，音乐播放
- 后台服务：在后台运行，客户看不到，进程优先级低，容易因内存不足被杀死，下载、I/O操作、复杂计算等

4.2、远程服务例子

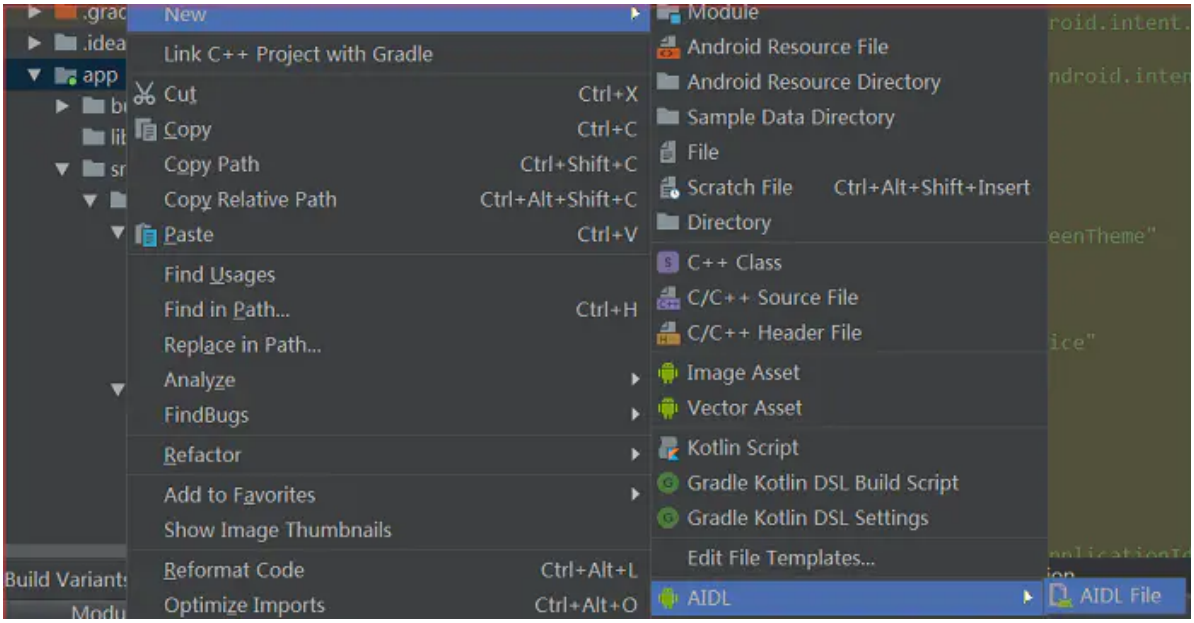
远程服务是运行在独立进程里的，与调用者不在同一进程，另外，多个应用程序（客户端）可以共同调用远程服务（服务端），为了实现调用进程和服务进程之间的通讯（IPC），需要用到AIDL。

> *IPC：Inter-Process Communication，即跨进程通信
> *AIDL：Android Interface Definition Language，即Android接口定义语言；用于让某个Service与多个应用程序组件之间进行跨进程通信，从而可以实现多个应用程序共享同一个Service的功能。

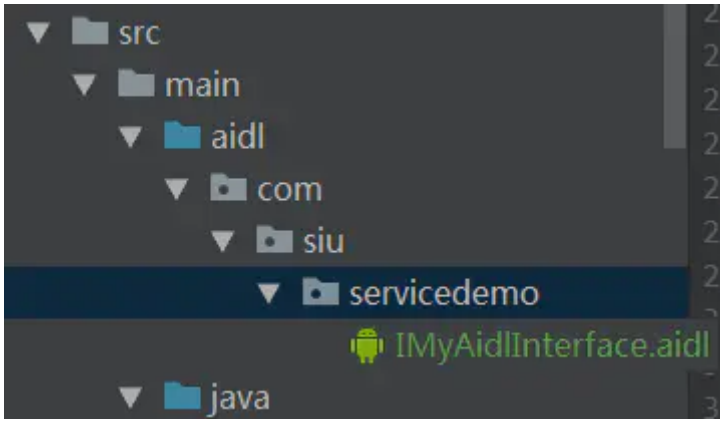
下面介绍远程服务的定义流程，比较简单。

4.2.1、创建服务端（Service）

1、创建AIDL文件
在需要创建远程服务的工程右键，创建AIDL文件，新建后会自动创建包目录和文件。



创建AIDL文件



AIDL文件目录

2、定义客户端-服务端之间的通信方法，

```
// IMyAidlInterface.aidl
package com.siu.servicedemo;
// Declare any non-default types here with import statements
interface IMyAidlInterface {
    int cal(int a,int b);
}
```

定义好后，点击AS的Make Project.
3、在Service中实现AIDL定义的方法

```
public class RemoteService extends Service {

    IMyAidlInterface.Stub mStub = new IMyAidlInterface.Stub() {
        @Override
        public int cal(int a, int b) throws RemoteException {
```

```
            return a + b;
        }
    };

    @Override
    public IBinder onBind(Intent intent) {
        return mStub;
    }
}
```

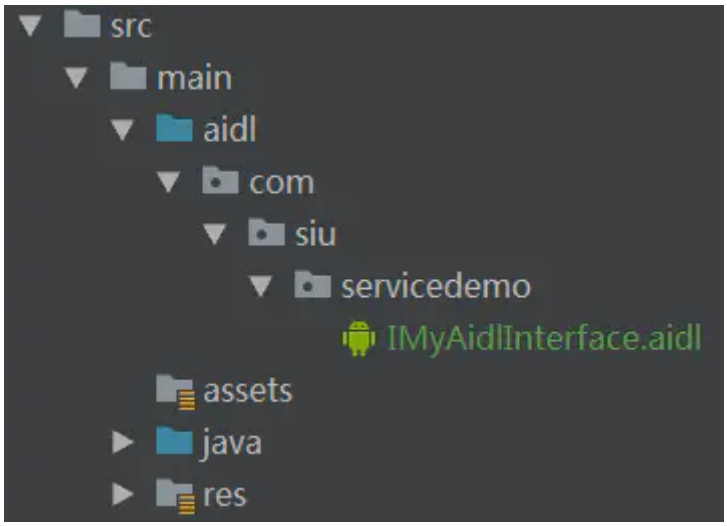
4、注册Service到AndroidMainfest.xml

```
        <service
            android:name=".RemoteService"
            android:exported="true" //外部可调用
            android:process=":remote">//声明为独立进程
            <intent-filter>
                <action android:name="${applicationId}.CalService"/>//自定义action
            </intent-filter>
        </service>
```

到此，Service的工作已完成。

4.2.2、创建客户端（Client）

1、拷贝整个aidl目录到Client工程



aidl目录

2、定义ServiceConnection，获得binder对象，调用AIDL定义的方法

```
    private IMyAidlInterface mBinder;

     private ServiceConnection mConnection = new ServiceConnection() {
         @Override
```

```
        public void onServiceConnected(ComponentName name, IBinder service) {
            mBinder = IMyAidlInterface.Stub.asInterface(service);
            try {
                int c = mBinder.cal(1, 2);
                Log.e("remoteService", "remoteService cal result=" + c);
            } catch (Exception e) {
                Log.e("remoteService", e.getMessage());
            }
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {
            mBinder = null;
        }
    };
```

3、绑定服务

```
        findViewById(R.id.btBindService).setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent it = new Intent("com.siu.servicedemo.CalService");
                it.setPackage("com.siu.servicedemo");
                bindService(it, mConnection, Context.BIND_AUTO_CREATE);
            }
        });
```

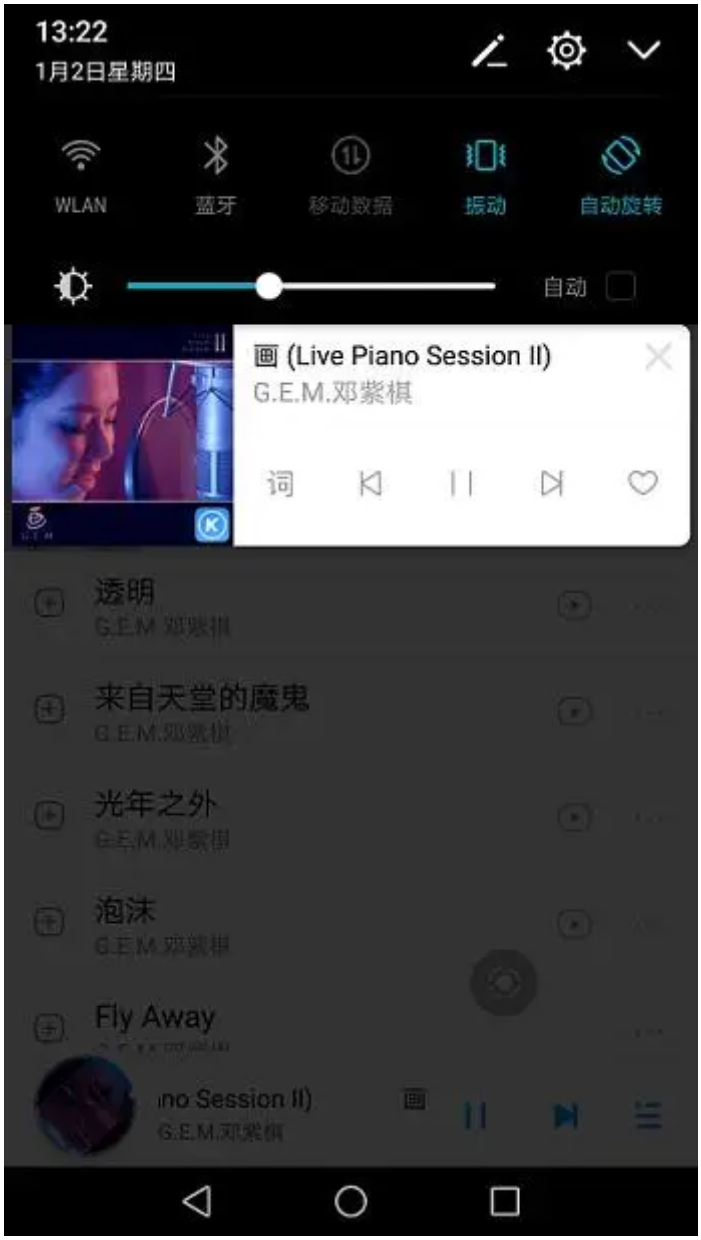
4、验证效果

```
remoteService: remoteService cal result=3
```

从Service端返回了计算结果，验证通过。

4.3、前台服务例子

前台服务最大的特点是会在通知栏显示通知，而后台服务没有，如下图音乐播放正是后台服务。除非服务停止或从前台删除，否则不能清除改通知。



前台服务

官方给出了启动和停止前台Service的方法：

- **startForeground(int id, Notification notification)**
  该方法的作用是把当前服务设置为前台服务，其中id参数代表唯一标识通知的整型数，需要注意的是提供给 startForeground() 的整型 ID 不得为 0，而notification是一个状态栏的通知。
- **stopForeground(boolean removeNotification)**
  该方法是用来从前台删除服务，此方法传入一个布尔值，指示是否也删除状态栏通知，true为删除。 注意该方法并不会停止服务。 但是，如果在服务正在前台运行时将其停止，则通知也会被删除。
  代码也比较简单：

```
@Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Notification notification;
        NotificationManager mNotificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        int importance = NotificationManager.IMPORTANCE_HIGH;
        AudioManager audioManager = (AudioManager)
getSystemService(Context.AUDIO_SERVICE);
```

```
        String title = "前台服务通知标题";
        String content = "前台服务通知内容";
        if (Build.VERSION.SDK_INT >= 26) {
            String id = "channel_id";  // 通知渠道的id
            CharSequence name = "channelName";   // 用户可以看到的通知渠道的名字.
            String description = "channelDesc";// 用户可以看到的通知渠道的描述
            NotificationChannel mChannel = new NotificationChannel(id, name,
importance);
            mChannel.setDescription(description);  // 配置通知渠道的属性
            mChannel.enableLights(true); // 设置通知出现时的闪灯（如果 android 设备支
持的话）
            if (audioManager.getRingerMode() == AudioManager.RINGER_MODE_VIBRATE
                    || audioManager.getRingerMode() == AudioManager.MODE_NORMAL)
{
                mChannel.enableVibration(true);  // 设置通知出现时的震动（如果
android 设备支持的话）
            }
            mChannel.setVibrationPattern(new long[]{100, 200, 300, 400, 500,
400, 300, 200, 400});
            mNotificationManager.createNotificationChannel(mChannel);

            notification = new Notification.Builder(this, id)
                    .setContentTitle(title).setContentText(content)
                    .setLargeIcon(BitmapFactory.decodeResource(getResources(),
R.drawable.icon))
                    .setSmallIcon(R.drawable.ic_launcher)
                    .build();
        } else {
            int defaults = Notification.DEFAULT_LIGHTS |
Notification.DEFAULT_SOUND;
            if (audioManager.getRingerMode() == AudioManager.RINGER_MODE_VIBRATE
                 || audioManager.getRingerMode() == AudioManager.MODE_NORMAL) {
                     defaults |= Notification.DEFAULT_VIBRATE;//设置
Notification.DEFAULT_VIBRATE的flag后可能会在任何
                     情况下都震动，部分系统的bug，所以要判断是否开启振动
            }
            notification = new Notification.Builder(this)
                    .setLargeIcon(BitmapFactory.decodeResource(getResources(),
R.drawable.icon))
                    .setContentTitle(title)
                    .setSmallIcon(R.drawable.ic_launcher)
                    .setContentText(content)
                    .setAutoCancel(true)
                    .setDefaults(defaults).getNotification();
        }

        startForeground(1, notification);
        return super.onStartCommand(intent, flags, startId);
    }
```

测试效果：

前台服务demo

五、IntentService

5.1、定义和特点

IntentService是Service的子类，是系统封装的用于处理异步任务的类，内部默认启动一个工作线程，逐一处理任务，当执行完毕，服务自动退出，具有以下特点：
*特殊的Service，继承于Service，本身是一个抽象类；
*默认启动一个工作线程HandlerThread，逐一处理耗时异步任务，处理完后自动停止，不适于处理并行任务；

*拥有较高的优先级（Service），不容易被系统杀死，适用于执行较高优先级的异步任务；
*使用简单，只需要实现构造方法和onHandleIntent，onHandleIntent为异步方法，可执行耗时任务。

5.2、简单例子

```
public class MyIntenteService extends IntentService {
    public MyIntenteService() {
        super("MyIntenteService");
    }

  @Override
  protected void onHandleIntent(@Nullable Intent intent) {
        //执行耗时任务
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
        }
        Log.e("MyIntenteService", "handle task name=" +
intent.getStringExtra("taskName"));
    }

    @Override
    public void onDestroy() {
        Log.e("MyIntenteService", "onDestroy");
        super.onDestroy();
    }
}
```

通过上面代码定义了简单的IntentService例子，只需要实现构造方法和onHandleIntent。

```
 findViewById(R.id.btTest).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                   for (int i = 0; i < 10; i++) {
                          Intent it = new Intent(this, MyIntenteService.class);
                          it.putExtra("taskName", "taskName" + i);
                          startService(it);//循环启动任务
                    }
             }
  }
```

在调用的地方循环多次启动Service，执行多个任务，输入的日志如下：

```
E/MyIntenteService: onCreate
E/MyIntenteService: handle task name=taskName0
E/MyIntenteService: handle task name=taskName1
E/MyIntenteService: handle task name=taskName2
E/MyIntenteService: handle task name=taskName3
E/MyIntenteService: handle task name=taskName4
E/MyIntenteService: handle task name=taskName5
E/MyIntenteService: handle task name=taskName6
E/MyIntenteService: handle task name=taskName7
E/MyIntenteService: handle task name=taskName8
E/MyIntenteService: handle task name=taskName9
E/MyIntenteService: onDestroy
```

通过日志发现，即使启动了多次MyIntenteService，但实例只有1个，任务都添加到消息队列，执行完所有任务后自动销毁。

5.3、源码分析

```
public abstract class IntentService extends Service {
    private volatile Looper mServiceLooper;

    private volatile ServiceHandler mServiceHandler;
    private String mName;
    private boolean mRedelivery;

    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }

        @Override
        public void handleMessage(Message msg) {
            onHandleIntent((Intent) msg.obj);
            stopSelf(msg.arg1);
        }
    }

    public IntentService(String name) {
        super();
        mName = name;
    }

    public void setIntentRedelivery(boolean enabled) {
        mRedelivery = enabled;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        HandlerThread thread = new HandlerThread("IntentService[" + mName +
"]");
        thread.start();
        mServiceLooper = thread.getLooper();
        mServiceHandler = new ServiceHandler(mServiceLooper);
```

```
    }

    @Override
    public void onStart(Intent intent, int startId) {
        Message msg = mServiceHandler.obtainMessage();
        msg.arg1 = startId;
        msg.obj = intent;
        mServiceHandler.sendMessage(msg);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        onStart(intent, startId);
        return mRedelivery ? START_REDELIVER_INTENT : START_NOT_STICKY;
    }

    @Override
    public void onDestroy() {
        mServiceLooper.quit();
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @WorkerThread
    protected abstract void onHandleIntent(Intent intent);
}
```

源码比较容易理解，分析下主要的方法：

1、onCreate

第一次启动服务，会调用onCreate方法，内部会开启一个异步线程HandlerThread，HandlerThread带有消息循环对象Looper，然后创建ServiceHandler（继承Handler），传入HandlerThread持有的Looper对象，这样ServiceHandler就能与异步线程的Looper进行绑定，就能处理异步任务了。

2、ServiceHandler

继承于Handler，需要传入Looper对象，在handleMessage中处理消息队列中的任务，然后回调抽象方法onHandleIntent，这就是创建IntentService需要实现的方法，在方法内执行异步任务。执行完onHandleIntent后，调用stopSelf(msg.arg1)，不是stopSelf()，因为stopSelf()会立即停止服务，而stopSelf（int startId）会等所有任务执行完才停止。

3、onStartCommand、onStart

每次启动Service会执行onStartCommand，在方法内再执行onStart方法，而onStart则通过mServiceHandler对象发送一个消息，该消息将在HandlerThread中执行，会触发2中的handleMessage方法，执行异步任务。

六、Service和Thread的区别

不同	Thread	Service
概念不同	程序执行的最小单元，分配CPU的基本单位	Android的一种机制，没有UI的后台组件，与其他组件的关系类似CS，通过IPC通信
执行任务	可执行耗时任务	运作在主线程，不可执行耗时任务，如果需要，在内部启动Thread
使用场景	执行可能会影响UI主线程的耗时任务	不需要UI的长时间后台任务，如播放音乐、下载

七、如何保证 Service 不被杀死

7.1、开启不死服务的途径

在一些特定的场景，Service需要保持运行不能被系统杀死，有以下途径：

- **1、onStartCommand返回START_STICKY或START_REDELIVER_INTENT**
  针对内存资源不足系统可能会杀非前台Service的情况，可以将onStartCommand的返回值设为START_STICKY或START_REDELIVER_INTENT，Service被杀死后，当资源足够时会自动恢复该服务。
- **2、提升Service优先级**
  将Service设置为前台服务，拥有前台进程的优先级，这样系统需要释放资源时不会优先杀死该进程。
- **3、Service互拉**
  同时启动A、B两个服务，互相监听对方的广播，在各自的onDestroy方法中，发送广播，收到广播后重新启动对方，如A被杀死触发onDestroy方法，A发出广播，B监听到后马上重新启动A，方法有效的前提是进程仍在。

> 对于内存不足系统回收，方法1和3都有效，而在内存不足系统回收、应用管理中杀死正在运行服务，都会触发onDestroy，方法3都有效，但是如果是用户强行杀死进程，所有方法都无效。

- **4、监听系统静态广播**
  监听系统广播，如网络变化、解锁屏，然后重启服务，不过当进程被杀，方法也无效。
- **5、第三方APP唤醒**
  极光、个推等第三方推送平台sdk的做法，为了保证推送服务常驻，如果应用A、B、C都用了该sdk，当用户打开应用A时，同时拉起B、C的推送服务。6.0以后系统为了省电，提高续航能力，引入了Doze模式，该方法开始不理想。
- **6、守护进程**
  一般采用Native进程，5.0以下拉活没问题，5.0以上由于改为杀死进程的同时干掉了进程组，所以父子型守护进程将无法保证重启。
- **7、双Native独立进程**
  双Native独立进程，互相守护，对上面守护进程的补充，非父子关系，这种方式在github上看到过开源项目MarsDaemon ，但仅兼容到Android6.0。
- **8、使用AlarmManager**
  利用AlarmManager定时唤起（killBackgroundProcess可以唤起但force-stop后无法唤起）。
- **9、使用JobSheduler机制**
  Android5.x、6.x有效，但AndroidN失效，Android N可以了解下scheduleAsPackage。

7.2、到底要不要开启永不退出的服务

Android service 不被杀死"永不退出的服务"（双进程，服务，多进程，微信）在文中引入了一篇文献，总结得比较好。总的原则来说就是：

> android开发者在可能的情况下尽量避免开发长时间运行的服务，只有服务真的在做事情的时候才开启服务，不要让服务坐着干等。是否要停止服务，应该遵循用户的意愿和系统的调度。如果服务消耗资源比较多，又强制持续运行，将会严重占用系统资源和加速电量消耗，影响用户的使用，还可能被用户认为是流氓应用，随时被卸载掉。

八、Service的适配

8.1、Android 5.0 以上不支持隐式启动服务

官方说明：绑定到服务

> Context.bindService()方法现在需要显式Intent，如果提供隐式 intent，将引发异常。为确保应用的安全性，请使用显式 intent 启动或绑定Service，且不要为服务声明 intent 过滤器。
> 例如下面代码：

```
Intent it = new Intent();
it.setAction("test_demo_service");
bindService(it, new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder
iBinder) {
        }
        @Override
        public void onServiceDisconnected(ComponentName componentName) {
         }
}, Service.BIND_AUTO_CREATE);
```

无论是startService还是bindService，启动后报错如下：

```
  E/AndroidRuntime: FATAL EXCEPTION: main
  Process: xxx, PID: 30060
  java.lang.IllegalArgumentException: Service Intent must be explicit: Intent {
act=test_demo_service }
        at android.app.ContextImpl.validateServiceIntent(ContextImpl.java:1644)
        at android.app.ContextImpl.startServiceCommon(ContextImpl.java:1685)
        at android.app.ContextImpl.startService(ContextImpl.java:1657)
        at android.content.ContextWrapper.startService(ContextWrapper.java:644)
```

解决方案：

- 改为显式启动
- 指定包名

8.2 、Android 8.0 以上禁止后台启动服务

官方说明：后台 Service 限制
处于前台时，应用可以自由创建和运行前台与后台 Service。 进入后台时，在一个持续数分钟的时间窗内，应用仍可以创建和使用 Service。 在该时间窗结束后，应用将被视为处于空闲状态。 此时，系统将停止应用的后台 Service，就像应用已经调用 Service 的Service.stopSelf()方法一样。
在这些情况下，后台应用将被置于一个临时白名单中并持续数分钟。 位于白名单中时，应用可以无限制地启动 Service，并且其后台 Service 也可以运行。 处理对用户可见的任务时，应用将被置于白名单中，例如：

- 处理一条高优先级Firebase 云消息传递 (FCM)消息。
- 接收广播，例如短信/彩信消息。
- 从通知执行PendingIntent
- 在 VPN 应用将自己提升为前台进程前开启VpnService

> 请注意： IntentService 是一项 Service，因此其遵守针对后台 Service 的新限制。 因此，许多依赖 IntentService 的应用在适配 Android 8.0 或更高版本时无法正常工作。 出于这一原因，Android 支持库 26.0.0 引入了一个新的JobIntentService类，该类提供与 IntentService 相同的功能，但在 Android 8.0 或更高版本上运行时使用作业而非 Service。

在 Android 8.0 之前，创建前台 Service 的方式通常是先创建一个后台 Service，然后将该 Service 推到前台。 Android 8.0 有一项复杂功能：系统不允许后台应用创建后台 Service。 因此，Android 8.0 引入了一种全新的方法，即 **startForegroundService()**，以在前台启动新 Service。 在系统创建 Service 后，应用有五秒的时间来调用该 Service 的 startForeground() 方法以显示新 Service 的用户可见通知。 如果应用在此时间限制内未调用 **startForeground()**，则系统将停止此 Service 并声明此应用为 ANR。

8.3、源码分析（api 26）

调用Activity的startService实际是调用ContextWrapper.startService。

```
public class ContextWrapper extends Context {    Context mBase;    ...
 @Override    public ComponentName startService(Intent service) {        return
mBase.startService(service);    }    ...}
```

上面方法中的mBase的对象类型是Context，其实现类是ContextImpl，接着看ContextImpl.startService

```
@Override    public ComponentName startService(Intent service) {
 warnIfCallingFromSystemProcess();        return startServiceCommon(service,
mUser);    }private ComponentName startServiceCommon(Intent service, UserHandle
user) {        try {            validateServiceIntent(service);
 service.prepareToLeaveProcess(this);            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 e.rethrowFromSystemServer();
}    }
```

其中，有个方法验证Intent的方法，针对非显式启动且没指定包名的，5.0以上直接抛异常，低版本则输出警告日志。

```
private void validateServiceIntent(Intent service) {    if
(service.getComponent() == null && service.getPackage() == null) {        if
(getApplicationInfo().targetSdkVersion >= Build.VERSION_CODES.LOLLIPOP) {
     IllegalArgumentException ex = new IllegalArgumentException(
   "Service Intent must be explicit: " + service);            throw ex;        }
else {            Log.w(TAG, "Implicit intents with startService are not safe: "
+ service                    + " " + Debug.getCallers(2, 3));        }    }}
```

这里不展开分析更多的源码，感兴趣的可以阅读startService方法源码.

8.4、Service应用中遇到的坑

项目中有不少场景用了Service（其实有些地方根据Google推荐可以改为JobScheduler），其中有个Service是在进入首页的onPostCreate启动，是显式启动，然后收到了不少崩溃日志，主要分为两种：

崩溃一：java.lang.SecurityException: Unable to start service Intent

```
Caused by: java.lang.SecurityException: Unable to start service Intent {
act=start_for_mainpage cmp=xxx/.xxxService (has extras) }: Unable to launch app
xxx/10098 for service Intent { act=start_for_mainpage cmp=xxx/.xxxService }:
user 0 is restricted\n\tat
android.app.ContextImpl.startServiceCommon(ContextImpl.java:1729)\n\tatandroid.a
pp.ContextImpl.startService(ContextImpl.java:1702)\n\tatandroid.content.ContextW
rapper.startService(ContextWrapper.java:494)\n\tat
```

崩溃二：java.lang.IllegalStateException: Not allowed to start service Intent

```
java.lang.IllegalStateException: Not allowed to start service Intent {
act=start_for_mainpage cmp=xxx/.xxxService (has extras) }: app is in background
uid UidRecord{70cfbb u0a159 TPSL idle procs:1 seq(0,0,0)}
```

第一个类型崩溃率比第二个高，接近百万分一，都是OPPO手机（系统版本为4.4.4和5.1.1，机型有A31c，A31，A33，A53m，A33m），第二个崩溃都是8.0系统，品牌有小米和华为，但是两个类型崩溃我都重现不了，模拟在后台启动也重现不了。

解决方案:

后来修改为启动服务的时机，在首页的onResume启动，并且加了try catch，如果抛异常，再重试。后面也收到这样的崩溃日志了，可能是加了try catch的原因，也没收到功能上有问题的反馈。

另外，对于第一个异常，也有开发者碰到，主要是OPPO手机针对电量优化，息屏后一段时间禁止自启动导致。

第四章： 手写Android组件通信框架EventBus

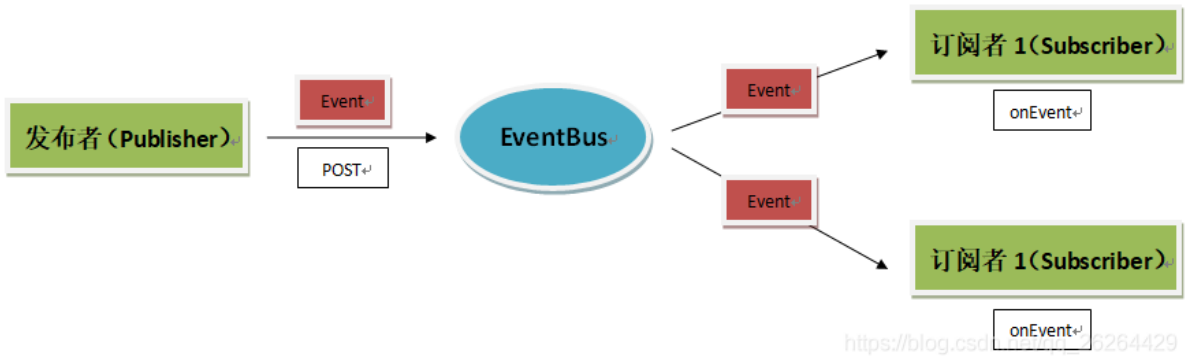
EventBus原理详解

前言

如今Android项目越来越庞大，参与人数越来越多，采用组件化开发已经是很有必要的了。组件化开发的优缺点就不多做介绍了，而EventBus大家或多或少也使用过。EventBus是一种用于Android的发布/订阅事件总线。它有很多优点：简化应用组件间的通信；解耦事件的发布者和接收者；避免复杂和容易出错的依赖和生命周期的问题等等。我们所看重的就是高度解耦和使用方便。

基础知识

EventBus采用了发布者/订阅者模式



订阅者：通过EventBus来订阅事件，即发布者发布事件的时候，如果该订阅者的订阅事件处理方法就会被调用

发布者：通过EventBus把事件POST出去。

使用方法

1. **添加依赖**
2. **基础用法**
3. **粘性事件**
4. **线程模式**
5. **事件优先级**
6. **订阅者索引**

1.添加依赖

在模块的**build.gradle**添加依赖EventBus依赖

```
dependencies {
    ...
    implementation 'org.greenrobot:eventbus:3.1.1'
}
```

如果开启了混淆的话，请添加EventBus的混淆规则。ProGuard工具混淆了方法名，并可能会移除未被调用的方法。而订阅者的事件处理方法是没有被直接调用的，如果开启的话，就必须保留这些方法，则在模块的混淆配置里面**proguard-rules.pro**混淆规则文件添加如下规则：

```
# EventBus
-keepattributes *Annotation*
-keepclassmembers class ** {
    @org.greenrobot.eventbus.Subscribe <methods>;
}
-keep enum org.greenrobot.eventbus.ThreadMode { *; }
```

2.基础用法

主要有三个步骤：

1. **定义事件**
   事件可以是任意的java对象。例如：

```
public class EvenBusMessageVo {
    private String mMessage;
    public EvenBusMessageVo(String mMessage) {
        this.mMessage = mMessage;
    }
    public String getmMessage() {
        return mMessage;
    }
    public void setmMessage(String mMessage) {
        this.mMessage = mMessage;
    }
}
```

1. **定义事件处理方法（订阅者方法）**
   订阅者需要定义事件处理方法（订阅者方法），在发布对应类型数据（即你post的对象类型），该方法会被调用。EventBus 3使用 **@Subscribe**注解来表明该方法就是订阅者方法。方法名可以是任意，参数类型为订阅事件的类型。例如：

```
@Subscribe(threadMode = ThreadMode.MAIN)
    public void sub(EvenBusMessageVo message) {
        Log.d(TAG, "发送过来数据是：" + message.getmMessage());
    }
```

订阅者还需要在总线上进行注册（EventBus），在不需要的时候需要注销。只有注册了的订阅者，才会收到事件。一般我们是在 **Activity** 或 **Fragment**的生命周期的 **onCreate()** 和 **onDestroy()** 方法来进行注册和注销。例如：

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity===>>";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 注册订阅者
        EventBus.getDefault().register(this);
    }
    @Subscribe(threadMode = ThreadMode.MAIN)
    public void sub(EvenBusMessageVo message) {
        Log.d(TAG, "发送过来数据是：" + message.getmMessage());
    }
    public void toTwoActivity(View view) {
        Intent intent = new Intent(this, TwoActivity.class);
        startActivity(intent);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // 注销订阅者
        EventBus.getDefault().unregister(this);
    }
}
```

1. **发送事件**
   在需要的时候发布事件，所有**订阅了该类型事件的订阅者** 都会收到该事件。例如：

```
public class TwoActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);
    }
    /**
     * 点击发布事件
     * @param view
     */
    public void sendMain(View view) {
        EventBus.getDefault().post(new EvenBusMessageVo("发布一个事件"));
    }
}
```

当点击了发布事件的按钮时候，TwoActivity会发布一个EvenBusMessageVo事件。此时可以看到一条log打印：

```
2019-06-12 15:23:10.646 5436-5436/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: 发送过来数据是：发布一个事件
```

3.粘性事件

如果先发布事件，然后才有订阅者订阅了该事件，除非再次发布，否则订阅者永远收不到该事件。此时可以使用粘性事件，发布一个粘性事件后，EventBus将在内存中缓存该粘性事件，当有订阅者订阅订阅该类型的粘性事件，订阅会收到该事件。
例如：

订阅粘性事件

```
public class TwoActivity extends AppCompatActivity {
    private static final String TAG = "TwoActivity===>>";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);
        // 注册订阅者
        EventBus.getDefault().register(this);
    }
    @Subscribe(threadMode = ThreadMode.MAIN ,sticky = true)
    public void sub(EvenBusMessageVo message) {
        Log.d(TAG, "发送过来数据是：" + message.getmMessage());
        // 如果不清除该粘性事件，则在当次启动应用过程中，每次打开TwoActivity都会收到该事件
        EventBus.getDefault().removeStickyEvent(message);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // 注销订阅者
        EventBus.getDefault().unregister(this);
    }
}
```

发布粘性事件

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity===>>";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    /**
     * 发送粘性事件
     * @param view
     */
    public void send(View view){
        EventBus.getDefault().postSticky(new EvenBusMessageVo("发布一个粘性事件"));
    }
    /**
     * 跳转到TwoActivity
     * @param view
     */
    public void toTwoActivity(View view) {
        Intent intent = new Intent(this, TwoActivity.class);
        startActivity(intent);
    }
}
```

运行，先点击发布按钮，然后再点击跳转按钮，跳转到TwoActivity后，确实收到了该事件。
日志输出结果：

```
2019-06-12 16:02:17.429 5663-5663/com.kbs.client.dn_20190605_eventbus
D/TwoActivity===>>: 发送过来数据是：发布一个粘性事件
```

4.线程模式

EventBus支持订阅者方法在不同的线程中被调用。可以使用指定线程模式来指定订阅者方法运行在哪个线程，EventBus中支持五种线程模式：

- **ThreadMode.POSTING**订阅者方法将在发布者所在的线程中被调用，即发布者是主线程，则订阅者方法就是在主线程中被调用。这个是 **默认的线程模式**。优势：事件的传递是同步的，一旦发布事件，那么所有的该模式下的订阅者都将被调用。这种线程模式避免了线程的切换，有最少的性能开销，因此不要求是必须是主线程并且耗时很多的简单任务推荐使用该模式。使用该模式的订阅者方法不能做耗时操作，因为很有可能是在主线程，会导致主线程的阻塞，出现ANR错误。
- **ThreadMode.MAIN**订阅者方法将强制在主线程（UI线程）中被调用，因此可以在该模式下的订阅者方法可以直接更新UI。如果发布者是主线程，那么该模式的订阅方法将会被直接调用，反之会切换到主线程后，再调用该模式的订阅者方法。使用该模式的订阅者方法不能做耗时操作，会导致主线程的阻塞，出现ANR错误。
- **ThreadMode.MAIN_ORDERED**订阅者方法将强制在主线程（UI线程）中被调用。因此可以在该模式下的订阅者方法可以直接更新UI。如果发布者是主线程，否则会先进行线程的切换，然后该模式的订阅方法将先进入队列后才发送给订阅者。使得事件的处理保持严格的串行顺序。使用该模式的订阅者方法不能做耗时操作，会导致主线程的阻塞，出现ANR错误。
- **ThreadMode.BACKGROUND**订阅者方法将强制在后台线程（子线程）中被调用。如果发布者不是主线程，则该模式下订阅者方法会被直接调用，否则会进行线程的切换，再调用订阅者方法。由于该模式下订阅者方法是运行在后台线程（子线程）中，所以不能直接做UI更新操作。
- **ThreadMode.ASYNC**订阅者方法是在独立的线程中运行，既不在发布事件的线程中，也不在主线程中。 一般可以使用这来进行网络和一些耗时操作。EventBus有专门的线程池对子线程进行管理，但仍然要避免同一时间开启太多的ASYNC模式线程。

订阅者方法：

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity===>>";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 注册订阅者
        EventBus.getDefault().register(this);
    }
    @Subscribe(threadMode = ThreadMode.POSTING)
    public void sub(EvenBusMessageVo message) {
        Log.d(TAG, "ThreadMode.POSTING，发送过来数据是：" + message.getmMessage() +
"，线程名：" + Thread.currentThread().getName());
    }
    @Subscribe(threadMode = ThreadMode.MAIN)
    public void sub1(EvenBusMessageVo message) {
        Log.d(TAG, "ThreadMode.MAIN，发送过来数据是：" + message.getmMessage() + ",
线程名：" + Thread.currentThread().getName());
    }
    @Subscribe(threadMode = ThreadMode.MAIN_ORDERED)
    public void sub2(EvenBusMessageVo message) {
        Log.d(TAG, "ThreadMode.MAIN_ORDERED，发送过来数据是：" +
message.getmMessage() + "，线程名：" + Thread.currentThread().getName());
    }
    @Subscribe(threadMode = ThreadMode.BACKGROUND)
```

```
    public void sub4(EvenBusMessageVo message) {
        Log.d(TAG, "ThreadMode.BACKGROUND，发送过来数据是：" + message.getmMessage()
+ "，线程名：" + Thread.currentThread().getName());
    }
    @Subscribe(threadMode = ThreadMode.ASYNC)
    public void sub5(EvenBusMessageVo message) {
        Log.d(TAG, "ThreadMode.ASYNC，发送过来数据是：" + message.getmMessage() +
"，线程名：" + Thread.currentThread().getName());
    }
    public void toTwoActivity(View view) {
        Intent intent = new Intent(this, TwoActivity.class);
        startActivity(intent);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // 注销订阅者
        EventBus.getDefault().unregister(this);
    }
}
```

发布者方法:

```
public class TwoActivity extends AppCompatActivity {
    private static final String TAG = "TwoActivity===>>";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);
    }
    /**
     * 主线程发布事件
     *
     * @param view
     */
    public void sendMain(View view) {
        EventBus.getDefault().post(new EvenBusMessageVo("主线程发布一个事件"));
        Log.d(TAG, "发布的线程：" + Thread.currentThread().getName());
    }
    /**
     * 子线程发布事件
     *
     * @param view
     */
    public void sendMain2(View view) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                EventBus.getDefault().post(new EvenBusMessageVo("子线程发布一个事
件"));
                Log.d(TAG, "发布的线程：" + Thread.currentThread().getName());
            }
        }).start();
    }
}
12345678910111213141516171819202122232425262728293031
```

点击主线程发布事件，结果如下：

```
2019-06-12 17:32:09.451 8753-8753/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.POSTING，发送过来数据是：主线程发布一个事件，线程名：
main
2019-06-12 17:32:09.451 8753-8753/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.MAIN，发送过来数据是：主线程发布一个事件，线程名：main
2019-06-12 17:32:09.451 8753-8753/com.kbs.client.dn_20190605_eventbus
D/TwoActivity===>>: 发布的线程：main
2019-06-12 17:32:09.460 8753-8952/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.BACKGROUND，发送过来数据是：主线程发布一个事件，线程名：
pool-1-thread-4
2019-06-12 17:32:09.461 8753-8953/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.ASYNC，发送过来数据是：主线程发布一个事件，线程名：pool-
1-thread-5
2019-06-12 17:32:09.467 8753-8753/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.MAIN_ORDERED，发送过来数据是：主线程发布一个事件，线程
名：main
```

点击子线程发布事件，结果如下：

```
2019-06-12 17:31:09.116 8753-8943/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.POSTING，发送过来数据是：子线程发布一个事件，线程名：
Thread-4
2019-06-12 17:31:09.116 8753-8943/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.BACKGROUND，发送过来数据是：子线程发布一个事件，线程名：
Thread-4
2019-06-12 17:31:09.116 8753-8943/com.kbs.client.dn_20190605_eventbus
D/TwoActivity===>>: 发布的线程：Thread-4
2019-06-12 17:31:09.120 8753-8944/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.ASYNC，发送过来数据是：子线程发布一个事件，线程名：pool-
1-thread-3
2019-06-12 17:31:09.130 8753-8753/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.MAIN，发送过来数据是：子线程发布一个事件，线程名：main
2019-06-12 17:31:09.130 8753-8753/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.MAIN_ORDERED，发送过来数据是：子线程发布一个事件，线程
名：main
```

由此可以推断以上的结论。

5.事件优先级

EventBus支持在定义订阅者方法的时候指定事件传递的优先级。通常情况下，订阅者方法的事件传递优先级为0，数值越大，则表示优先级越高，在相同的线程模式下，优先级越高的更快接收到该事件。
PS:事件优先级只有在相同的线程模式下才有效。

那么如何在订阅者方法下进行指定优先级。例如：

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity===>>";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 注册订阅者
        EventBus.getDefault().register(this);
```

```
    }
    @Subscribe(threadMode = ThreadMode.MAIN, priority = 1)
    public void sub(EvenBusMessageVo message) {
        Log.d(TAG, "priority = 1，发送过来数据是：" + message.getmMessage());
    }
    @Subscribe(threadMode = ThreadMode.MAIN, priority = 2)
    public void sub2(EvenBusMessageVo message) {
        Log.d(TAG, "priority = 2，发送过来数据是：" + message.getmMessage());
    }
    @Subscribe(threadMode = ThreadMode.MAIN, priority = 3)
    public void sub3(EvenBusMessageVo message) {
        Log.d(TAG, "priority = 3，发送过来数据是：" + message.getmMessage());
    }
    public void toTwoActivity(View view) {
        Intent intent = new Intent(this, TwoActivity.class);
        startActivity(intent);

    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // 注销订阅者
        EventBus.getDefault().unregister(this);
    }
}
```

在TwoActivity中点击多次发布事件按钮，日志打印如下：

```
2019-06-12 16:19:25.932 5790-5790/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: priority = 3，发送过来数据是：发布一个事件
2019-06-12 16:19:25.933 5790-5790/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: priority = 2，发送过来数据是：发布一个事件
2019-06-12 16:19:25.933 5790-5790/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: priority = 1，发送过来数据是：发布一个事件
```

通过日志可以知道，优先级越高的越早接收到事件。

6.订阅者索引

默认情况下，EventBus在查找订阅方法的时采用的是反射。订阅者索引是3.0后才有的新特性。它可以加速订阅者的注册，是一个可优化的选项。原理是：使用EventBus的注解处理器在应用构建期间创建订阅者索引类，此类包含了订阅者和订阅者方法的相关信息，官方推荐在Android中使用订阅者索引类以获得最佳的性能。以前没怎么用过，此了解来源于网络。

要开启订阅者索引的生成，你需要在构建脚本中使用 **annotationProcessor** 属性将EventBus的注解处理器添加到应用的构建中，还要设置一个 **eventBusIndex** 参数来指定要生成的订阅者索引的完全限定类名。

第一步：修改下模块下面的 **build.gradle** 构建脚本

```
 defaultConfig {
        ...
        javaCompileOptions {
            annotationProcessorOptions {
                arguments = [eventBusIndex:
'com.kbs.client.dn_20190605_eventbus.MyEventBusIndex']
            }
        }
    }
```

第二步：重新rebuild一下，EventBus注解处理器会生产一个订阅者索引类，里面保存了所有订阅者和订阅者方法信息，如下所示：

```
/** This class is generated by EventBus, do not edit. */
public class MyEventBusIndex implements SubscriberInfoIndex {
    private static final Map<Class<?>, SubscriberInfo> SUBSCRIBER_INDEX;

    static {
        SUBSCRIBER_INDEX = new HashMap<Class<?>, SubscriberInfo>();

        putIndex(new SimpleSubscriberInfo(MainActivity.class, true, new
SubscriberMethodInfo[] {
            new SubscriberMethodInfo("sub", EvenBusMessageVo.class),
            new SubscriberMethodInfo("sub1", EvenBusMessageVo.class,
ThreadMode.MAIN),
            new SubscriberMethodInfo("sub2", EvenBusMessageVo.class,
ThreadMode.MAIN_ORDERED),
            new SubscriberMethodInfo("sub4", EvenBusMessageVo.class,
ThreadMode.BACKGROUND),
            new SubscriberMethodInfo("sub5", EvenBusMessageVo.class,
ThreadMode.ASYNC),
        }));

    }

    private static void putIndex(SubscriberInfo info) {
        SUBSCRIBER_INDEX.put(info.getSubscriberClass(), info);
    }

    @Override
    public SubscriberInfo getSubscriberInfo(Class<?> subscriberClass) {
        SubscriberInfo info = SUBSCRIBER_INDEX.get(subscriberClass);
        if (info != null) {
            return info;
        } else {
            return null;
        }
    }
```

第三步：
使用订阅者索引时，需要先进行初始化操作，在应用启动的时候进行初始化，例如：

```
public class App extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        // 配置EventBus
        EventBus.builder().addIndex(new
MyEventBusIndex()).installDefaultEventBus();
    }
}
```

在`AndroidManifest.xm`文件中的`application`标签里面添加**android:name =".App"**，就能进行初始化操作了。

运行结果：

```
2019-06-12 18:33:04.107 8753-8753/com.kbs.client.dn_20190605_eventbus
D/MainActivity===>>: ThreadMode.MAIN，发送过来数据是：主线程发布一个事件，线程名：main
```

手写简易版EventBus

我们需要手写这样的一个框架的话，就必须要清楚，该框架的原理，这里我不会单去将原理，我们直接通过步骤代码来明白原理。其实原理其实很简单，就是获取订阅者和订阅者方法，通过辨别订阅者方法参数类型来确定调用的什么方法，然后通过反射来调用订阅者方法，那样就实现了，具体里面的一些粘性事件，优先级，订阅者索引，只是一些额外的功能和优化点而已，在这我就没有写了，跨进程也暂时没有实现，那些需要慢慢的去添加，去优化框架，这我只是通过最简单的实现来剖析原理。主要通过以下部分来手写实现的，有什么不足的希望大家多多指点。

1. **定义注解**
2. **对订阅者方法的封装**
3. **注册订阅者**
4. **发布事件**
5. **执行订阅者方法**
6. **线程模式处理**

1.定义注解

在你的工程项目中，先新建一个lib，它是属于java Library的。命名为`buslibrary`
新建注解，我们也取名为`Subscribe`

```
/**
 * author   : android 老鲜肉
 * document : 声明注解
 */
@Target(ElementType.METHOD) // 注解的作用域(表示放在哪个上面的)
@Retention(RetentionPolicy.RUNTIME) // 声明这个注解的生命周期  SOURCE:源码期 RUNTIME:
运行期 CLASS:编译期
public @interface Subscribe {
    ThreadMode threadMode() default ThreadMode.POSTING; // 默认线程为POSION
}
```

注解里面就一个方法，`threadMode()`，来获取线程模式，在这我们还需要一个线程模式的枚举类，那我们就是创建这个枚举类`ThreadMode`,里面的值和我上面介绍的功能一样，当然你也可以自己任意命名的

```
/** * author    : android 老鲜肉 * document : 声明线程模式枚举 */public enum ThreadMode {    POSTING, // 发布者在什么线程，订阅者方法就运行在什么线程    MAIN, // 强制切换到主线程    BACKGROUND, // 强制切换到子线程}
```

2.对订阅者方法的封装

首先我们要分析下，订阅者方法包含哪些内容。

```
    @Subscribe(threadMode = ThreadMode.MAIN, priority = 3)    public void sub3(EvenBusMessageVo message) {        Log.d(TAG, "priority = 3，发送过来数据是：" + message.getmMessage());    }
```

我截取了之前我们实现过的订阅者方法，优先级咱们就先不加了，当然这个可以自行添加其他的标识属性。可以发现订阅者方法包含了`ThreadMode`、`EvenBusMessageVo（订阅者方法参数对象类型）`、`方法本身`。那我们就把订阅者方法封装成一个对象。

```
/** * author    : android 老鲜肉 * document : 声明线程模式枚举 */public class MethodManager {    // 方法接受的参数对象的类型    private Class<?> type;    // 线程模式    private ThreadMode threadMode;    // 方法本身    private Method method;    public MethodManager(Class<?> type, ThreadMode threadMode, Method method) {        this.type = type;        this.threadMode = threadMode;        this.method = method;    }    public Class<?> getType() {        return type;    }    public ThreadMode getThreadMode() {        return threadMode;    }    public Method getMethod() {        return method;    }}
```

3.注册订阅者

EventBus中注册订阅者方式：`EventBus.getDefault().register(this);`，那我们也新建一个名字也为`EventBus`的类，可以看的出来它是单例的，代码如下：

```
/** * author    : android 老鲜肉 * document : 声明线程模式枚举 */ public class EventBus {    private static EventBus instance;    // 存储所有订阅者的map    // map的键就是注册的时候传进去的对象    private Map<Object, List<MethodManager>> map;    /**     * 初始化单例EventBus     *     * @return EventBus.this     */    public static EventBus getDefault() {        if (instance == null)            synchronized (EventBus.class) {                if (instance == null) instance = new EventBus();            }        return instance;    }    /**     * 私有化构造方法     */    private EventBus() {        map = new HashMap<>();    } }
```

注册订阅者方法定义：

```
    /**     * 传进来的组件 找到这个组件中的所有订阅方法     *     * @param object
*/    public void register(Object object) {        // 获取这个组件所有的订阅者方法集合
       List<MethodManager> methodManagers = map.get(object);        if
(methodManagers == null) {            // 通过组件对象找到这个组件所有标识了Subscribe注解
的方法即为订阅方法             methodManagers = findAnnotationMethod(object);
   map.put(object, methodManagers);        }    }        /**     * 通过传进来的组件
来找它所有符合条件的订阅方法     *     * @param object 组件对象     * @return 这个组件所
有的订阅者方法集合     */    private List<MethodManager> findAnnotationMethod(Object
object) {        List<MethodManager> methodList = new ArrayList<>();        // 获
取到组件的类对象         Class<?> aClass = object.getClass();        // 通过类对象获取
到所有方法的集合         Method[] declaredMethods = aClass.getDeclaredMethods();
   // 循环方法集合进行判别那个是订阅者方法         for (Method declaredMethod :
declaredMethods) {            // 返回该方法的注解在此方法的指定注解类型
Subscribe annotation = declaredMethod.getAnnotation(Subscribe.class);
if (annotation == null) { // 如果注解为空就表示不是订阅者方法                continue;
           }            // 添加严格的校验             // 获取方法返回值的类型
Type genericReturnType = declaredMethod.getGenericReturnType();            if
(!genericReturnType.toString().equals("void")) {                throw new
RuntimeException("方法返回值不是void");            }            // 获取到方法所有接收
参数类型数组             Class<?>[] parameterTypes =
declaredMethod.getParameterTypes();            if (parameterTypes.length != 1) {
              throw new RuntimeException("方法必须有且只有一个参数");            }
         // 生成订阅者方法封装对象             MethodManager methodManager = new
MethodManager(parameterTypes[0], annotation.threadMode(), declaredMethod);
     methodList.add(methodManager);        }        return methodList;    }
```

4.发布事件

发布事件我们也在`EventBus`类中，定义了一个`post`方法。

```
    /**     * 发布事件     * @param setter 事件发布的对象类型     */    public void
post(final Object setter) {        Set<Object> objects = map.keySet();        //
循环存储所有订阅者的map         for (final Object object : objects) {            //
获取到订阅者所有的订阅者方法的集合             List<MethodManager> methodManagers =
map.get(object);            if (methodManagers != null && methodManagers.size() >
0) {                for (final MethodManager methodManager : methodManagers) {
                // 判断发布者发布事件的发布的对象类型和订阅者方法的定义的类型是否一致
            if (methodManager.getType().isAssignableFrom(setter.getClass())) {
                  // 通过反射来执行订阅者方法
invoke(methodManager, object, setter);                    }                }
       }        }    }
```

5.执行订阅者方法

执行订阅者方法是通过反射来执行的。

```
 /**     * 通过反射执行订阅者方法     * @param methodManager 订阅者方法封装对象     *
@param object 订阅者对象（组价）     * @param setter 订阅者方法定义的参数对象     */
 private void invoke(MethodManager methodManager, Object object, Object setter)
{        Method method = methodManager.getMethod();        try {            // 通
过反射执行方法             method.invoke(object, setter);        } catch
(IllegalAccessException e) {            e.printStackTrace();        } catch
(InvocationTargetException e) {            e.printStackTrace();        }
}123456789101112131415161718
```

至此就已经可以实现组件通信功能了，但是无法做到订阅者方法在指定的线程中执行。

6.线程模式处理

在`MethodManager`对象中是有线程模式这个属性的，也就是说在`invoke()`之前进行线程判断和切换就可以了。
在`post`方法中对应位置添加

```
 if (methodManager.getType().isAssignableFrom(setter.getClass())) {
 // 线程切换操作             switch (methodManager.getThreadMode()) { // 判断线程模式
               case POSTING: // 默认 直接invoke,不进行线程切换
invoke(methodManager, object, setter);                    break;
case BACKGROUND:                    // 判断当前线程是否是主线程                    if
(Looper.getMainLooper() == Looper.myLooper()) {                        // 切换到子
线程操作 executorService 是线程池对象
executorService.execute(new Runnable() {                            @Override
                        public void run() {
invoke(methodManager, object, setter);                            }
         });                    } else { // 当前是子线程
invoke(methodManager, object, setter);                    }
break;                case MAIN:                    // 判断当前线程是否是主线程
             if (Looper.getMainLooper() == Looper.myLooper()) {
     invoke(methodManager, object, setter);                    } else { // 当前是
子线程 handler 是用来切换到主线程的                        handler.post(new
Runnable() {                            @Override
public void run() {                                invoke(methodManager, object,
setter);                            }                        });
   }                    break;            }
```

在EventBus的构造方法里面生成线程池对象`executorService`

```
    /**     * 私有化构造方法     */    private EventBus() {        map = new
HashMap<>();        executorService = Executors.newCachedThreadPool();    }
```

构建成员变量 Handler ，来进行切换到主线程的操作

```
private Handler handler = new Handler();
```

最后还需要进行注销订阅者

```
  /**     * 注销订阅方法     * @param object     */    public void
unregister(Object object) {        if (object != null && map.get(object) != null
&& map.get(object).size() > 0) {            map.remove(object);        }    }
```

这样简易版EventBus就已经写完了，比较简单，接下来运行一下
订阅者代码：

```
import com.kbs.client.buslibrary.EventBus;import
com.kbs.client.buslibrary.Subscribe;import
com.kbs.client.buslibrary.ThreadMode;public class MainActivity extends
AppCompatActivity {    private static final String TAG = "MainActivity===>>";
 @Override    protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);        // 注册订阅者
EventBus.getDefault().register(this);    }    @Subscribe(threadMode =
ThreadMode.MAIN)    public void sub(EvenBusMessageVo message) {        Log.d(TAG,
"ThreadMode.MAIN，发送过来数据是：" + message.getmMessage() + "，线程名：" +
Thread.currentThread().getName());    }    @Subscribe(threadMode =
ThreadMode.POSTING)    public void sub2(EvenBusMessageVo message) {
Log.d(TAG, "ThreadMode.POSTING，发送过来数据是：" + message.getmMessage() + "，线程
名：" + Thread.currentThread().getName());    }    @Subscribe(threadMode =
ThreadMode.BACKGROUND)    public void sub3(EvenBusMessageVo message) {
Log.d(TAG, "ThreadMode.BACKGROUND，发送过来数据是：" + message.getmMessage() + "，线
程名：" + Thread.currentThread().getName());    }    public void
toTwoActivity(View view) {        Intent intent = new Intent(this,
TwoActivity.class);        startActivity(intent);    }    @Override    protected
void onDestroy() {        super.onDestroy();        // 注销订阅者
EventBus.getDefault().unregister(this);    }}
```

发布事件代码：

```
import com.kbs.client.buslibrary.EventBus;public class TwoActivity extends
AppCompatActivity {    private static final String TAG = "TwoActivity===>>";
 @Override    protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main2);    }    /**     * 主线程发布事件     *
@param view     */    public void sendMain(View view) {    Log.d(TAG, "发布的线
程：" + Thread.currentThread().getName());        EventBus.getDefault().post(new
EvenBusMessageVo("主线程发布一个事件"));    }    /**     * 子线程发布事件     *     *
@param view     */    public void sendMain2(View view) {        new Thread(new
Runnable() {            @Override            public void run() {
Log.d(TAG, "发布的线程：" + Thread.currentThread().getName());
 EventBus.getDefault().post(new EvenBusMessageVo("子线程发布一个事件"));
}        }).start();    }}
```

运行结果：

```
2019-06-13 13:33:59.162 2210-2210/com.kbs.client.dn_20190605_eventbus
D/TwoActivity===>>: 发布的线程：main2019-06-13 13:33:59.163 2210-
2210/com.kbs.client.dn_20190605_eventbus D/MainActivity===>>: ThreadMode.MAIN，发
送过来数据是：主线程发布一个事件，线程名：main2019-06-13 13:33:59.163 2210-
2210/com.kbs.client.dn_20190605_eventbus D/MainActivity===>>:
ThreadMode.POSTING，发送过来数据是：主线程发布一个事件，线程名：main2019-06-13
13:33:59.167 2210-2817/com.kbs.client.dn_20190605_eventbus D/MainActivity===>>:
ThreadMode.BACKGROUND，发送过来数据是：主线程发布一个事件，线程名：pool-1-thread-12019-
06-13 13:34:06.978 2210-2818/com.kbs.client.dn_20190605_eventbus
D/TwoActivity===>>: 发布的线程：Thread-42019-06-13 13:34:06.985 2210-
2818/com.kbs.client.dn_20190605_eventbus D/MainActivity===>>:
ThreadMode.POSTING，发送过来数据是：子线程发布一个事件，线程名：Thread-42019-06-13
13:34:06.988 2210-2818/com.kbs.client.dn_20190605_eventbus D/MainActivity===>>:
ThreadMode.BACKGROUND，发送过来数据是：子线程发布一个事件，线程名：Thread-42019-06-13
13:34:07.001 2210-2210/com.kbs.client.dn_20190605_eventbus D/MainActivity===>>:
ThreadMode.MAIN，发送过来数据是：子线程发布一个事件，线程名：main
```

运行结果可见，我们的功能确实是实现了。