ActivityManagerService知多少

概述

ActivityManagerService（以后简称AMS），是Android中最核心的服务，主要负责系统中四大组件的启动、切换、调度及应用进程的管理和调度等动作，其职责与操作系统中的进程管理和调度模块相类似，因此它在Android中非常重要。

本文从以下的角度阐述所理解的AMS：

1. 准备知识；
2. AMS的启动分析
3. startActivity的流程；
4. AMS的小知识；

准备知识

Binder知识

1. Binder简介

Android系统中，每个应用程序是由Android的`Activity`，`Service`，`Broadcast`，`ContentProvider`这四大组件的中一个或多个组合而成，这四大组件所涉及的多进程间的通信底层都是依赖于Binder IPC机制。例如当进程A中的Activity要向进程B中的Service通信，这便需要依赖于Binder IPC。不仅于此，整个Android系统架构中，大量采用了Binder机制作为IPC（进程间通信）方案，当然也存在部分其他的IPC方式，比如Zygote通信便是采用socket。

2. Binder的优势

Binder是Android系统IPC（进程间通信）方式之一。Linux已经拥有管道、System V IPC（消息队列/共享内存/信号量）、socket等IPC手段，却还要倚赖Binder来实现进程间通信，说明Binder具有无可比拟的优势。

一方面是传输性能。socket作为一款通用接口，其传输效率低，开销大，主要用在跨网络的进程间通信和本机上进程间的低速通信。消息队列和管道采用存储-转发方式，即数据先从发送方缓存区拷贝到内核开辟的缓存区中，然后再从内核缓存区拷贝到接收方缓存区，至少有两次拷贝过程。共享内存虽然无需拷贝，但控制复杂，难以使用。

IPC方式	数据拷贝次数
共享内存	0
Binder	1
Socket/管道/消息队列	2

还有一点是出于安全性考虑。Android作为一个开放式，拥有众多开发者的的平台，应用程序的来源广泛，确保智能终端的安全是非常重要的。终端用户不希望从网上下载的程序在不知情的情况下偷窥隐私数据，连接无线网络，长期操作底层设备导致电池很快耗尽等等。传统IPC没有任何安全措施，完全依赖上层协议来确保。

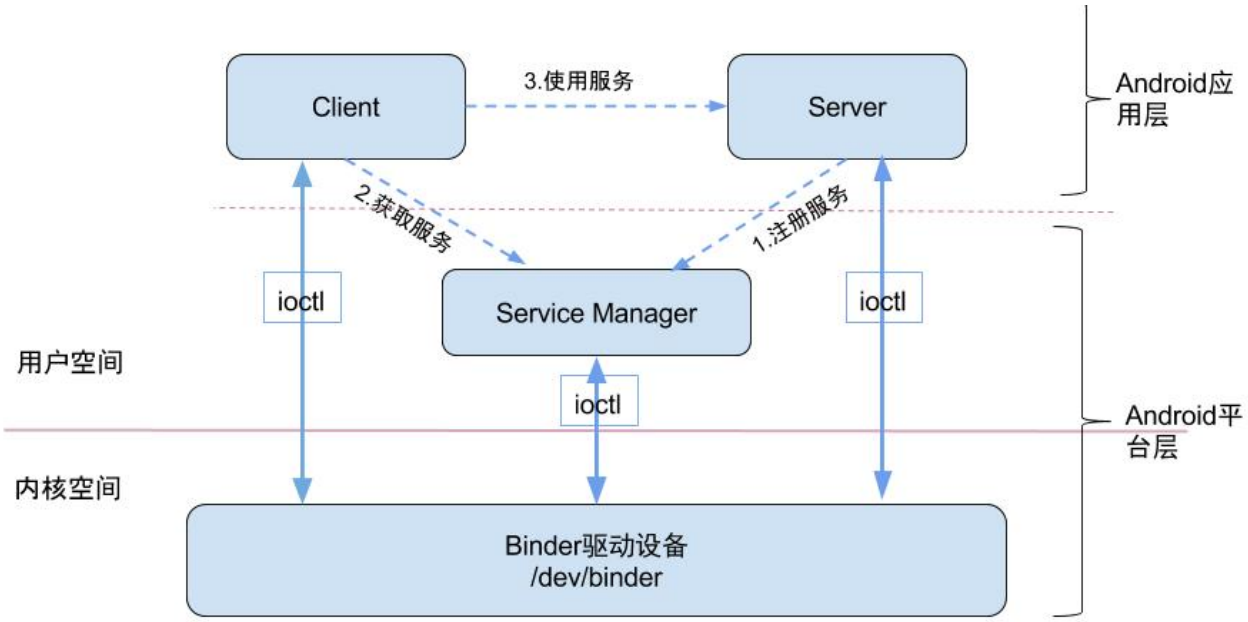
- 首先传统IPC的接收方无法获得对方进程可靠的UID/PID（用户ID/进程ID），从而无法鉴别对方身份。Android为每个安装好的应用程序分配了自己的UID，故进程的UID是鉴别进程身份的重要标志。使用传统IPC只能由用户在数据包里填入UID/PID，但这样不可靠，容易被恶意程序利用。可靠的身份标记只有由IPC机制本身在内核中添加。
- 其次传统IPC访问接入点是开放的，无法建立私有通道。比如命名管道的名称，system V的键值，socket的ip地址或文件名都是开放的，只要知道这些接入点的程序都可以和对端建立连接，不管怎样都无法阻止恶意程序通过猜测接收方地址获得连接。

  基于以上原因，Android需要建立一套新的IPC机制来满足系统对通信方式，传输性能和安全性的要求，这就是Binder。Binder基于Client-Server通信模式，传输过程只需一次拷贝，为发送发添加UID/PID身份，既支持实名Binder也支持匿名Binder，安全性高。

  另外，Binder对象是一个可以跨进程引用的对象，它的实体位于一个进程中，而它的引用却遍布于系统的各个进程之中。最诱人的是，这个引用和Java里引用一样既可以是强类型，也可以是弱类型，而且可以从一个进程传给其它进程，让大家都能访问同一Server，就象将一个对象或引用赋值给另一个引用一样。Binder模糊了进程边界，淡化了进程间通信过程，整个系统仿佛运行于同一个面向对象的程序之中。形形色色的Binder对象以及星罗棋布的引用仿佛粘接各个应用程序的胶水，这也是Binder在英文里的原意。

3. Binder的架构

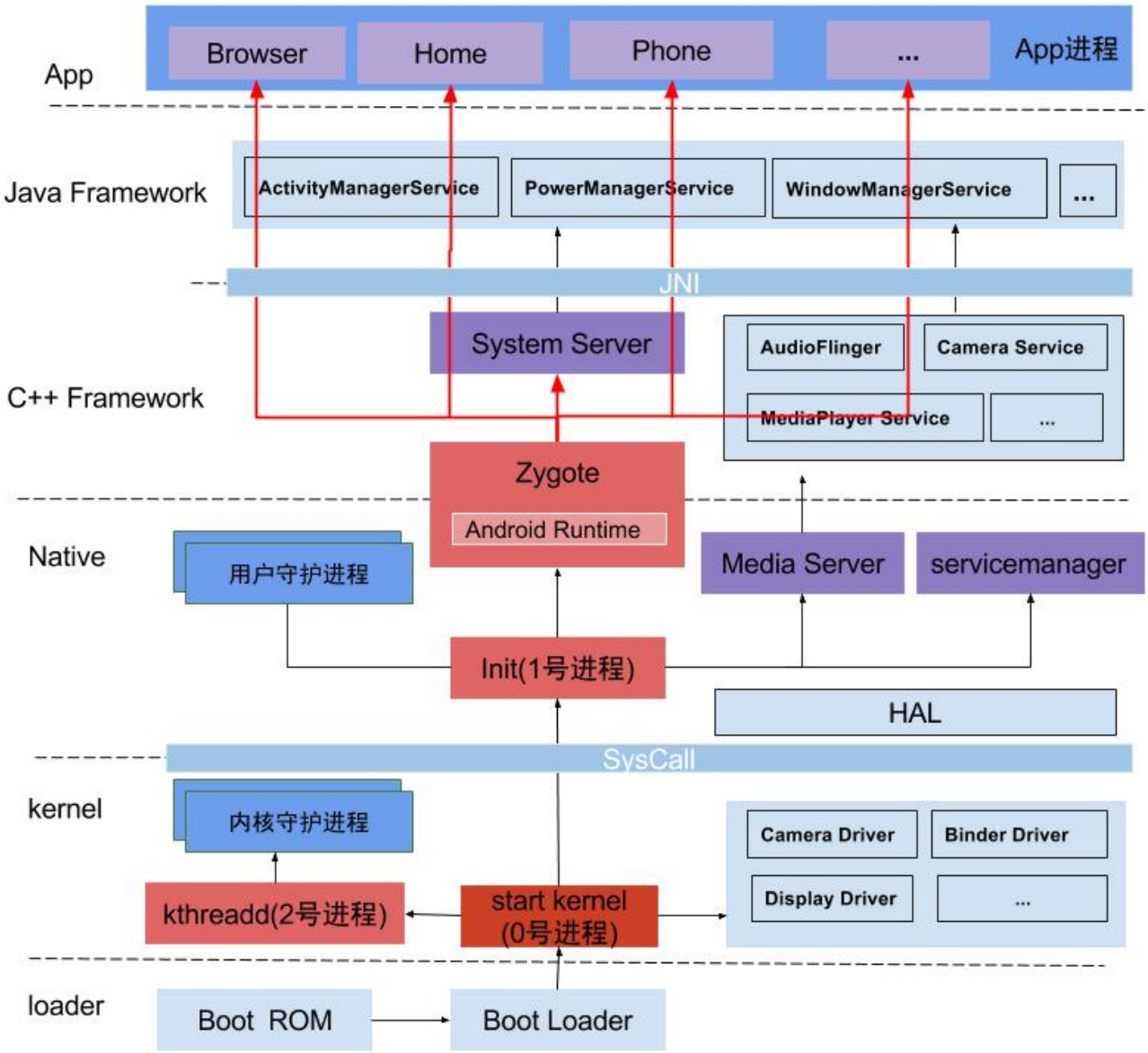
Binder通信采用C/S架构，从组件视角来说，包含Client、Server、ServiceManager以及binder驱动，其中ServiceManager用于管理系统中的各种服务。架构图如下所示：



ServiceManager是整个Binder通信机制的大管家，是Android进程间通信机制Binder的守护进程。当Service Manager启动之后，Client端和Server端通信时都需要先获取Service Manager接口，才能开始通信服务。

1. 注册服务(addService)：Server进程要先注册Service到ServiceManager。该过程：Server是客户端，ServiceManager是服务端。
2. 获取服务(getService)：Client进程使用某个Service前，须先向ServiceManager中获取相应的Service。该过程：Client是客户端，ServiceManager是服务端。
3. 使用服务：Client根据得到的Service信息建立与Service所在的Server进程通信的通路，然后就可以直接与Service交互。该过程：client是客户端，server是服务端。

Android的系统启动



从Android系统启动的角度分析

图解： Android系统启动过程由上图从下往上的一个过程：`Loader` -> `Kernel` -> `Native` -> `Framework` -> `App`，接来下简要说说每个过程：

1. Loader层

- Boot ROM: 当手机处于关机状态时，长按Power键开机，引导芯片开始从固化在`ROM`里的预设出代码开始执行，然后加载引导程序到`RAM`；
- Boot Loader：这是启动Android系统之前的引导程序，主要是检查RAM，初始化硬件参数等功能。

2. Kernel层

Kernel层是指Android内核层，到这里才刚刚开始进入Android系统。

- 启动Kernel的0号进程：初始化进程管理、内存管理，加载Display、Camera Driver、Binder Driver等相关工作；
- 启动kthreadd进程（pid=2）：是Linux系统的内核进程，会创建内核工作线程kworkder，软中断线程ksoftirqd，thermal等内核守护进程。`kthreadd进程是所有内核进程的鼻祖`。

3. Native层

这里的Native层主要包括init孵化来的用户空间的守护进程、HAL层以及开机动画等。启动`init`程（pid=1），是Linux系统的用户进程，`init进程是所有用户进程的鼻祖`。

- `init`进程会孵化出ueventd、logd、healthd、installd、adbd、lmkd等用户守护进程；
- `init`进程还启动`servicemanager`(binder服务管家)、`bootanim`(开机动画)等重要服务；
- `init`进程孵化出Zygote进程，Zygote进程是Android系统的第一个Java进程，`Zygote是所有Java进程的父进程`，Zygote进程本身是由init进程孵化而来的。

4. Framework层

- Zygote进程，是由init进程通过解析init.rc文件后fork生成的，Zygote进程主要包含：
  - 加载ZygoteInit类，注册Zygote Socket服务端套接字；
  - 加载虚拟机；
  - preloadClasses；
  - preloadResouces。
- System Server进程，是由Zygote进程fork而来，`System Server是Zygote孵化的第一个进程`，System Server负责启动和管理整个Java framework，包含ActivityManager，PowerManager等服务。
- Media Server进程，是由init进程fork而来，负责启动和管理整个C++ framework，包含AudioFlinger，Camera Service，等服务。

5. App层

- Zygote进程孵化出的第一个App进程是Launcher，这是用户看到的桌面App；
- Zygote进程还会创建Browser，Phone，Email等App进程，每个App至少运行在一个进程上。
- 所有的App进程都是由Zygote进程fork生成的。

从进程/线程的视角来分析

1.父进程

在所有进程中，以父进程的姿态存在的进程(即图中的浅红色项)，如下：

- kthreadd进程: 是所有内核进程的父进程
- init进程 ： 是所有用户进程的父进程(或者父父进程)
- zygote进程 ： 是所有上层Java进程的父进程，另外`zygote`的父进程是`init`进程。

2. 重量级进程

在Android进程中，有3个非常重要的进程(即图中的深紫色项)，如下：

- system_server：是由zygote孵化而来的，是zygote的首席大弟子，托起整个Java framework的所有service，比如ActivityManagerService, PowerManagerService等等。
- mediaserver：是由init孵化而来的，托起整个C++ framework的所有service，比如AudioFlinger, MediaPlayerService等等。
- servicemanager：是由init孵化而来的，是整个Binder架构(IPC)的大管家，所有大大小小的service都需要先请示servicemanager。

Zygote知识

zygote意为“受精卵“。Android是基于Linux系统的，而在Linux中，所有的进程都是由init进程直接或者是间接fork出来的，zygote进程也不例外。

在Android系统里面，zygote是一个进程的名字。Android是基于Linux System的，当你的手机开机的时候，Linux的内核加载完成之后就会启动一个叫`init`的进程。在Linux System里面，所有的进程都是由init进程fork出来的，我们的zygote进程也不例外。

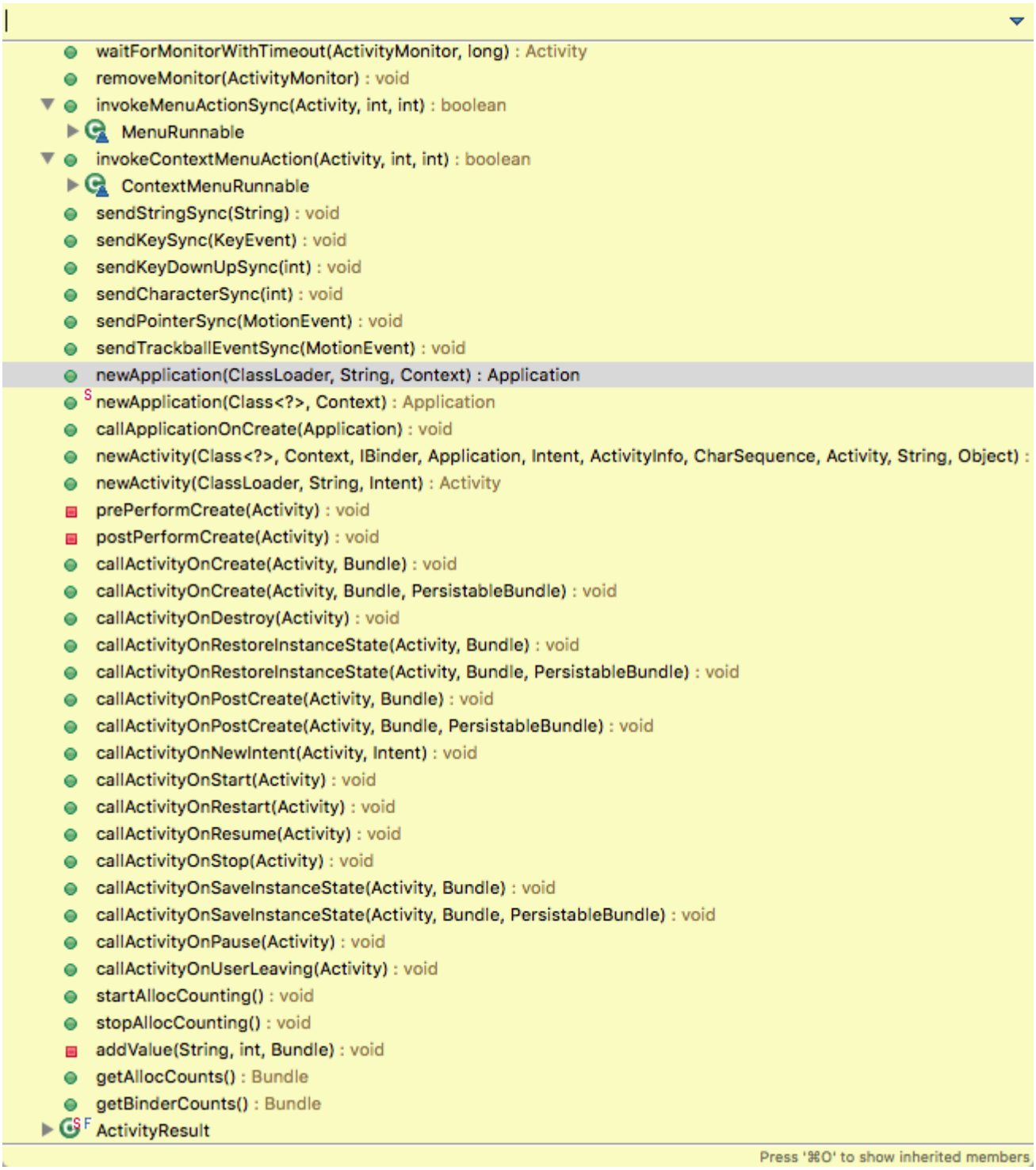
我们都知道，每一个App其实都是：

- 一个单独的dalvik虚拟机；
- 一个单独的进程；

所以当系统里面的第一个zygote进程运行之后，在这之后再开启App，就相当于开启一个新的进程。而为了实现资源共用和更快的启动速度，Android系统开启新进程的方式，是通过fork第一个zygote进程实现的。所以说，除了第一个zygote进程，其他应用所在的进程都是zygote的子进程，这下你明白为什么这个进程叫“受精卵”了吧？因为就像是一个受精卵一样，它能快速的分裂，并且产生遗传物质一样的细胞！

主要对象功能介绍

- ActivityManagerService，简称AMS，服务端对象，负责系统中所有Activity的生命周期。
- ActivityThread，App的真正入口。当开启App之后，会调用main()开始运行，开启消息循环队列，这就是传说中的UI线程或者叫主线程。与ActivityManagerServices配合，一起完成Activity的管理工作。
- ApplicationThread，实现ActivityManagerService与ActivityThread之间的交互。在ActivityManagerService需要管理相关Application中的Activity的生命周期时，通过ApplicationThread的代理对象与ActivityThread通讯。
- ApplicationThreadProxy，是ApplicationThread在服务器端的代理，负责和客户端的ApplicationThread通讯。AMS就是通过该代理与ActivityThread进行通信的。
- Instrumentation，每一个应用程序只有一个Instrumentation对象，每个Activity内都有一个对该对象的引用。Instrumentation可以理解为应用进程的管家，ActivityThread要创建或暂停某个Activity时，都需要通过Instrumentation来进行具体的操作。



- ActivityStask，Activity在AMS的栈管理，用来记录已经启动的Activity的先后关系，状态信息等。通过ActivityStack决定是否需要启动新的进程。
- ActivityRecord，ActivityStack的管理对象，每个Activity在AMS对应一个ActivityRecord，来记录Activity的状态以及其他的管理信息。其实就是服务器端的Activity对象的映像。
- TaskRecord，AMS抽象出来的一个“任务”的概念，是记录ActivityRecord的栈，一个“Task”包含若干个ActivityRecord。AMS用TaskRecord确保Activity启动和退出的顺序。如果你清楚Activity的4种launchMode，那么对这个概念应该不陌生。

AMS的启动分析

从上面的知识我们得知，SystemServer是由zygote进程fork出来的一个进程，系统里面重要的服务都是在这个进程里面开启的，比如ActivityManagerService、PackageManagerService、WindowManagerService等等。

所以ActivityManagerService进行初始化的时机很明确，就是在SystemServer进程开启的时候，就会初始化ActivityManagerService。从下面的代码中可以看到：

```
public final class SystemServer {
    /**
    * The main entry point from zygote.
    */
    public static void main(String[] args) {
        new SystemServer().run();
    }

    private void run() {
        // 加载android_servers.so库，该库包含的源码在frameworks/base/services/目录下。
        System.loadLibrary("android_servers");

        // 初始化系统上下文
        createSystemContext();

        // 创建系统服务管理，下面的系统服务开启都需要调用
SystemServiceManager.startService(Class<T>)，这个方法通过反射来启动对应的服务。
        mSystemServiceManager = new SystemServiceManager(mSystemContext);

        //启动各种系统服务
        try {
            startBootstrapServices();  // 启动引导服务
            startCoreServices();       // 启动核心服务
            startOtherServices();      // 启动其他服务
        } catch (Throwable ex) {
            Slog.e("System", "******************************************");
            Slog.e("System", "************ Failure starting system services", ex);
            throw ex;
        }
    }

    // 初始化系统上下文对象mSystemContext，并设置默认的主题，mSystemContext实际上是一个ContextImpl对
象。调用ActivityThread.systemMain()的时候，会调用ActivityThread.attach(true)，而在attach()里面，则创
建了Application对象，并调用了Application.onCreate()。
    private void createSystemContext() {
        ActivityThread activityThread = ActivityThread.systemMain();
        mSystemContext = activityThread.getSystemContext();
        mSystemContext.setTheme(android.R.style.Theme_DeviceDefault_Light_DarkActionBar);
    }

}
```

本节讲述system_server进程中AMS服务的启动过程，以startBootstrapServices()方法为起点，紧跟着startCoreServices()，startOtherServices()共3个方法。

startBootStrapServices

```
   // 这里开启了几个核心的服务，因为这些服务之间相互依赖，所以都放到这个方法里面。
private void startBootstrapServices() {
    // 启动AMS服务
    mActivityManagerService = mSystemServiceManager.startService(
            ActivityManagerService.Lifecycle.class).getService();
    // 设置AMS的系统服务器
    mActivityManagerService.setSystemServiceManager(mSystemServiceManager);

    // 初始化AMS的APP安装器
    mActivityManagerService.setInstaller(installer);

    // 初始化AMS相关的PMS
    mActivityManagerService.initPowerManagement();

    // 设置SystemServer
    mActivityManagerService.setSystemProcess();
}
```

```
public class SystemServiceManager {

    public SystemService startService(String className) {
        final Class<SystemService> serviceClass;
        try {
            serviceClass = (Class<SystemService>)Class.forName(className);
        } catch (ClassNotFoundException ex) {
          ...
        }
        return startService(serviceClass); // 启动服务
    }

    public <T extends SystemService> T startService(Class<T> serviceClass) {
        final String name = serviceClass.getName();

        if (!SystemService.class.isAssignableFrom(serviceClass)) {
            throw new RuntimeException("Failed to create " + name
                    + ": service must extend " + SystemService.class.getName());
        }
        final T service;
        try {
            Constructor<T> constructor = serviceClass.getConstructor(Context.class);
            // 创建ActivityManagerService.Lifecycle对象
            service = constructor.newInstance(mContext);
        } catch (Exception ex) {
            ...
        }
        // 注册Lifecycle服务，并添加到成员变量mServices
        mServices.add(service);

        try {
            // 启动ActivityManagerService.Lifecycle的onStart()
            service.onStart();
        } catch (RuntimeException ex) {
             ...
        }
        return service;
}
```

mSystemServiceManager.startService(xxx.class) 功能主要：

1. 创建xxx类的对象；
2. 将刚创建的对象添加到mSystemServiceManager的成员变量mServices；
3. 调用刚创建对象的onStart()方法。

```
public static final class Lifecycle extends SystemService {
    private final ActivityManagerService mService;

    public Lifecycle(Context context) {
        super(context);
        mService = new ActivityManagerService(context); // 创建ActivityManagerService
    }

    @Override
    public void onStart() {
        mService.start();
    }

    public ActivityManagerService getService() {
        return mService;
```

该过程：

1. 创建AMS内部类的Lifecycle对象，以及创建AMS对象；
2. 将Lifecycle对象添加到mSystemServiceManager的成员变量mServices；
3. 调用AMS.start();

```
public ActivityManagerService(Context systemContext) {
    mContext = systemContext;
    mFactoryTest = FactoryTest.getMode(); // 默认为FACTORY_TEST_OFF
    mSystemThread = ActivityThread.currentActivityThread();

    // 创建名为TAG（值为“ActivityManager”）的线程，并获取mHandler
    mHandlerThread = new ServiceThread(TAG,
            android.os.Process.THREAD_PRIORITY_FOREGROUND, false /*allowIo*/);
    mHandlerThread.start();
    mHandler = new MainHandler(mHandlerThread.getLooper());

    // 通过UiThread类，创建名为“android.ui”的线程
    mUiHandler = new UiHandler();

    // 前台广播队列，运行超时时间为10s
    mFgBroadcastQueue = new BroadcastQueue(this, mHandler,
            "foreground", BROADCAST_FG_TIMEOUT, false);
    // 后台广播队列，运行超时时间为60s
    mBgBroadcastQueue = new BroadcastQueue(this, mHandler,
            "background", BROADCAST_BG_TIMEOUT, true);
    mBroadcastQueues[0] = mFgBroadcastQueue;
    mBroadcastQueues[1] = mBgBroadcastQueue;

    // 创建ActiveService，它是service的服务管理类
    mServices = new ActiveServices(this);
    mProviderMap = new ProviderMap(this);

    // 新建目录/data/system
    File dataDir = Environment.getDataDirectory();
    File systemDir = new File(dataDir, "system");
    systemDir.mkdirs();

    // 创建服务BatteryStatsService
    mBatteryStatsService = new BatteryStatsService(systemDir, mHandler);
    mBatteryStatsService.getActiveStatistics().readLocked();
    mBatteryStatsService.scheduleWriteToDisk();
    mOnBattery = DEBUG_POWER ? true
            : mBatteryStatsService.getActiveStatistics().getIsOnBattery();
    mBatteryStatsService.getActiveStatistics().setCallback(this);

    // 创建进行统计服务，信息保存在目录/data/system/procstats
    mProcessStats = new ProcessStatsService(this, new File(systemDir, "procstats"));

    // 权限管理服务
    mAppOpsService = new AppOpsService(new File(systemDir, "appops.xml"), mHandler);

    mGrantFile = new AtomicFile(new File(systemDir, "urigrants.xml"));

    // User 0 是第一个，也是为唯一能在开机过程中运行的用户
    mStartedUsers.put(UserHandle.USER_OWNER, new UserState(UserHandle.OWNER, true));
    mUserLru.add(UserHandle.USER_OWNER);
    updateStartedUserArrayLocked();
```

```
    GL_ES_VERSION = SystemProperties.getInt("ro.opengles.version",
        ConfigurationInfo.GL_ES_VERSION_UNDEFINED);

    mTrackingAssociations = "1".equals(SystemProperties.get("debug.track-associations"));

    mConfiguration.setToDefaults();
    mConfiguration.setLocale(Locale.getDefault());
    mConfigurationSeq = mConfiguration.seq = 1;

    // 进程CPU使用情况的追踪器初始化
    mProcessCpuTracker.init();

    mCompatModePackages = new CompatModePackages(this, systemDir, mHandler);
    mIntentFirewall = new IntentFirewall(new IntentFirewallInterface(), mHandler);
    mRecentTasks = new RecentTasks(this);

    // 创建Activity的栈对象
    mStackSupervisor = new ActivityStackSupervisor(this, mRecentTasks);
    mTaskPersister = new TaskPersister(systemDir, mStackSupervisor, mRecentTasks);

    // 创建名为“CpuTracker”的线程
    mProcessCpuThread = new Thread("CpuTracker") {
        @Override
        public void run() {
            while (true) {
                try {
                    try {
                        synchronized(this) {
                            final long now = SystemClock.uptimeMillis();
                            long nextCpuDelay = (mLastCpuTime.get()+MONITOR_CPU_MAX_TIME)-now;
                            long nextWriteDelay = (mLastWriteTime+BATTERY_STATS_TIME)-now;
                            if (nextWriteDelay < nextCpuDelay) {
                                nextCpuDelay = nextWriteDelay;
                            }
                            if (nextCpuDelay > 0) {
                                mProcessCpuMutexFree.set(true);
                                this.wait(nextCpuDelay);
                            }
                        }
                    } catch (InterruptedException e) {
                    }
                    updateCpuStatsNow(); // 更新CPU状态
                } catch (Exception e) {
                    Slog.e(TAG, "Unexpected exception collecting process stats", e);
                }
            }
        }
    };

    // Watchdog添加对AMS的监控
    Watchdog.getInstance().addMonitor(this); // 死锁检查
    Watchdog.getInstance().addThread(mHandler); // 线程执行超时检查

}
```

该过程共创建了3个线程：分别为”ActivityManager”，”android.ui”，”CpuTracker”。

```
private void start() {
    Process.removeAllProcessGroups(); // 移除所有的进程组
    mProcessCpuThread.start(); // 启动CpuTracker线程

    mBatteryStatsService.publish(mContext); // 启动电池统计服务
    mAppOpsService.publish(mContext);
    // 创建LocalService，并添加到LocalServices
    LocalServices.addService(ActivityManagerInternal.class, new LocalService());
}
```

Tip: LocalServices与ServiceManager的区别？

```
public void setSystemProcess() {
    try {
        ServiceManager.addService(Context.ACTIVITY_SERVICE, this, true);
        ServiceManager.addService(ProcessStats.SERVICE_NAME, mProcessStats);
        ServiceManager.addService("meminfo", new MemBinder(this));
        ServiceManager.addService("gfxinfo", new GraphicsBinder(this));
        ServiceManager.addService("dbinfo", new DbBinder(this));
        if (MONITOR_CPU_USAGE) {
            ServiceManager.addService("cpuinfo", new CpuBinder(this));
        }
        ServiceManager.addService("permission", new PermissionController(this));
        ServiceManager.addService("processinfo", new ProcessInfoService(this));

        ApplicationInfo info = mContext.getPackageManager().getApplicationInfo(
                "android", STOCK_PM_FLAGS);
        // 调用ActivityThread的installSystemApplicationInfo()方法
        mSystemThread.installSystemApplicationInfo(info, getClass().getClassLoader());

        synchronized (this) {
            // 创建ProcessRecord对象
            ProcessRecord app = newProcessRecordLocked(info, info.processName, false, 0);
            app.persistent = true;
            app.pid = MY_PID;
            app.maxAdj = ProcessList.SYSTEM_ADJ;
            app.makeActive(mSystemThread.getApplicationThread(), mProcessStats);
            synchronized (mPidsSelfLocked) {
                mPidsSelfLocked.put(app.pid, app);
            }
            updateLruProcessLocked(app, false, null);
            updateOomAdjLocked();
        }
    } catch (PackageManager.NameNotFoundException e) {
        throw new RuntimeException(
                "Unable to find android system package", e);
    }
}
```

该方法主要工作，注册下面的服务：

服务名	类名	功能
activity	ActivityManagerService	AMS
procstats	ProcessStatsService	进程统计
meminfo	MemBinder	内存
gfxinfo	GraphicsBinder	graphics
dbinfo	DbBinder	数据库
cpuinfo	CpuBinder	CPU
permission	PermissionController	权限
processinfo	ProcessInfoService	进程服务

想要查看这些服务的信息，可通过`dumpsys <服务名>`命令。比如查看CPU信息命令`dumpsys cpuinfo`，查看graphics信息命令`dumpsys gfxinfo`。

```
public void installSystemApplicationInfo(ApplicationInfo info, ClassLoader classLoader) {
    synchronized (this) {
        // 调用ContextImpl的nstallSystemApplicationInfo()方法，
        // 最终调用LoadedApk的installSystemApplicationInfo，加载名为“android”的package。
        getSystemContext().installSystemApplicationInfo(info, classLoader);

        // give ourselves a default profiler
        mProfiler = new Profiler();
    }
}
```

startCoreServices

```
private void startCoreServices() {
    // 设置AMS的App使用情况统计服务
    mActivityManagerService.setUsageStatsManager(
            LocalServices.getService(UsageStatsManagerInternal.class));
}
```

可通过`dumpys usagestats`查看用户的每个应用的使用情况

startOtherServices

```
private void startOtherServices() {
    final Context context = mSystemContext;

    // 启动SettingsProvider
    mActivityManagerService.installSystemProviders();

    // 初始Watchdog，利用AMS保存异常日志到dropbox，并接收reboot广播
    final Watchdog watchdog = Watchdog.getInstance();
    watchdog.init(context, mActivityManagerService);

    // 初始化WMS，并设置到AMS中
    wm = WindowManagerService.main(context, inputManager,
            mFactoryTestMode != FactoryTest.FACTORY_TEST_LOW_LEVEL,
            !mFirstBoot, mOnlyCore);
    mActivityManagerService.setWindowManager(wm);

    mActivityManagerService.systemReady(new Runnable() {
        @Override
        public void run() {
            mSystemServiceManager.startBootPhase(
                    SystemService.PHASE_ACTIVITY_MANAGER_READY);

            mActivityManagerService.startObservingNativeCrashes();

            // 启动SystemUI
            startSystemUi(context);

            // 省略了一系列服务的执行systemReady方法

            Watchdog.getInstance().start();

            mSystemServiceManager.startBootPhase(
                    SystemService.PHASE_THIRD_PARTY_APPS_CAN_START);
        }
    });
}

static final void startSystemUi(Context context) {
    Intent intent = new Intent();
    intent.setComponent(new ComponentName("com.android.systemui",
                "com.android.systemui.SystemUIService"));
    context.startServiceAsUser(intent, UserHandle.OWNER);
}
```

总结

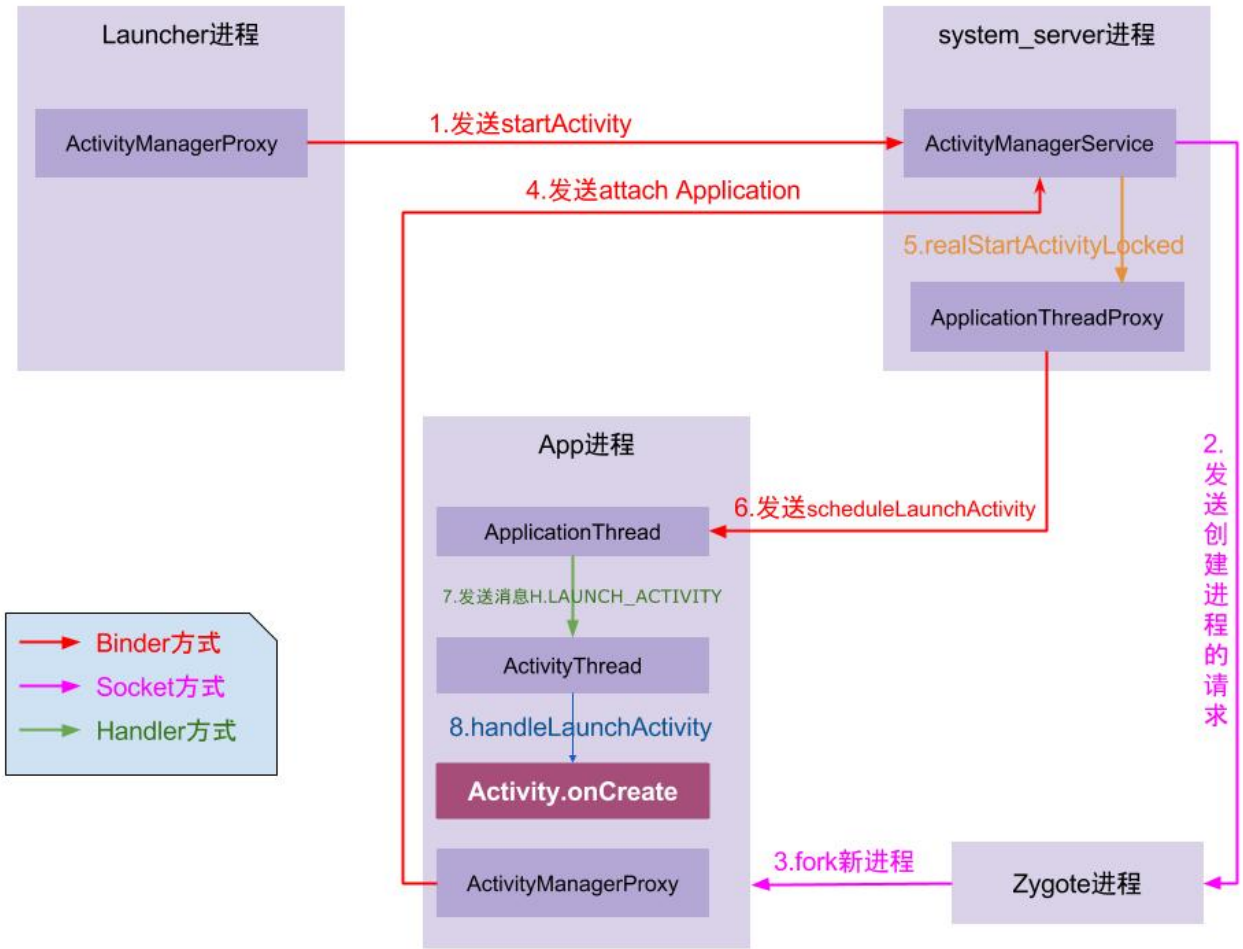
1. 创建AMS实例对象，创建Andoid Runtime，ActivityThread和Context对象；
2. setSystemProcess：注册AMS、meminfo、cpuinfo等服务到ServiceManager；再创建ProcessRecord对象；
3. installSystemProviderss，加载SettingsProvider；
4. 启动SystemUIService，再调用一系列服务的systemReady()方法；

当这些都创建完毕后，便启动HomeActivity界面。

startActivity的流程

启动过程

Activity的启动过程的源码相当复杂，涉及Instrumentation、ActivityThread和AMS。简单理解，启动Activity的请求会由Instrument来处理，然后它通过Binder向AMS发送请求，AMS内部维护者一个ActivityStack并负责栈内的Activity的状态同步，AMS通过ActivityThread去同步Activity的状态从而完成生命周期方法的调用。可用如下图来概括：



启动流程：

1. 点击桌面App图标，Launcher进程采用Binder IPC向system_server进程发起startActivity请求；
2. system_server进程接收到请求后，向zygote进程发送创建进程的请求；
3. Zygote进程fork出新的子进程，即App进程；
4. App进程，通过Binder IPC向sytem_server进程发起attachApplication请求；
5. system_server进程在收到请求后，进行一系列准备工作后，再通过binder IPC向App进程发送scheduleLaunchActivity请求；
6. App进程的binder线程（ApplicationThread）在收到请求后，通过handler向主线程发送LAUNCH_ACTIVITY消息；
7. 主线程在收到Message后，通过发射机制创建目标Activity，并回调Activity.onCreate()等方法。

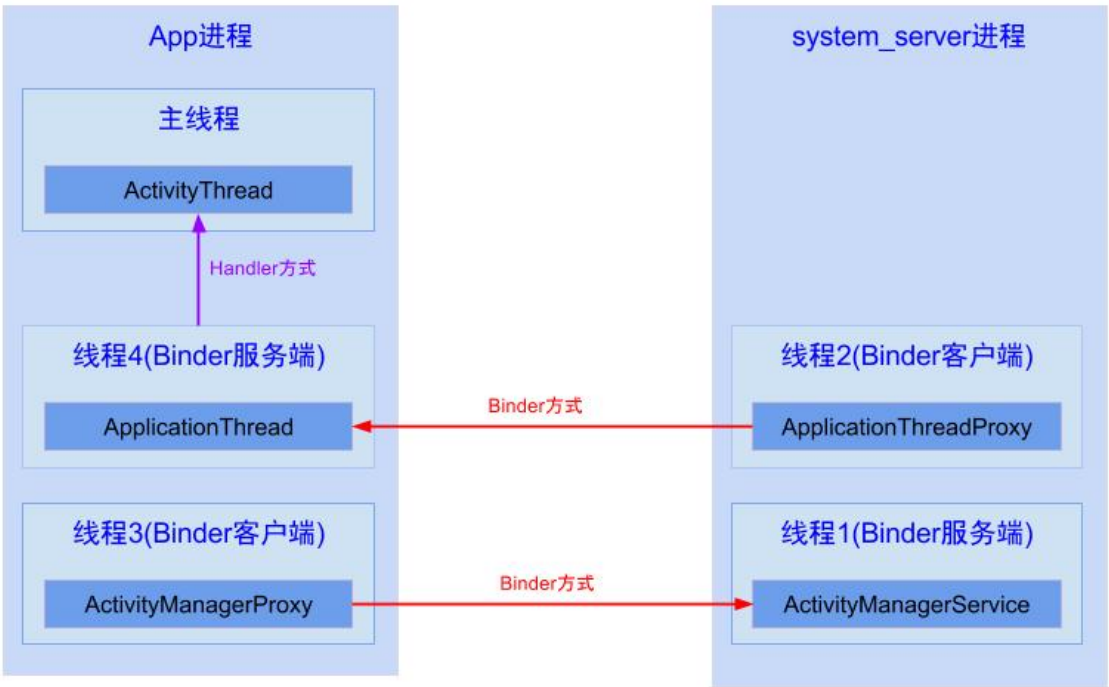
到此，App便正式启动，开始进入Activity生命周期，执行完onCreate/onStart/onResume方法，UI渲染结束后便可以看到App的主界面。

生命周期

对于App来说，其Activity的生命周期执行是由系统进程中的`ActivityManagerService`服务触发的，接下来从进程和线程的角度来分析Activity的生命周期，这里涉及到系统进程和应用进程：

system_server进程是系统进程，java framework框架的核心载体，里面运行了大量的系统服务，比如这里提供ApplicationThreadProxy（简称ATP），ActivityManagerService（简称AMS），这个两个服务都运行在system_server进程的不同线程中，由于ATP和AMS都是基于IBinder接口，都是binder线程，binder线程的创建与销毁都是由binder驱动来决定的。

App进程是应用程序所在进程，主线程主要负责Activity/Service等组件的生命周期以及UI相关操作都运行在这个线程； 另外，每个App进程中至少会有两个binder线程 ApplicationThread(简称AT)和ActivityManagerProxy（简称AMP），除了下图中所示的线程，其实还有很多线程，比如signal catcher线程等。



`Binder`用于不同进程之间通信，由一个进程的Binder客户端向另一个进程的服务端发送事件，比如图中线程2向线程4发送事务；而`handler`用于同一个进程中不同线程的通信，比如图中线程4向主线程发送消息。

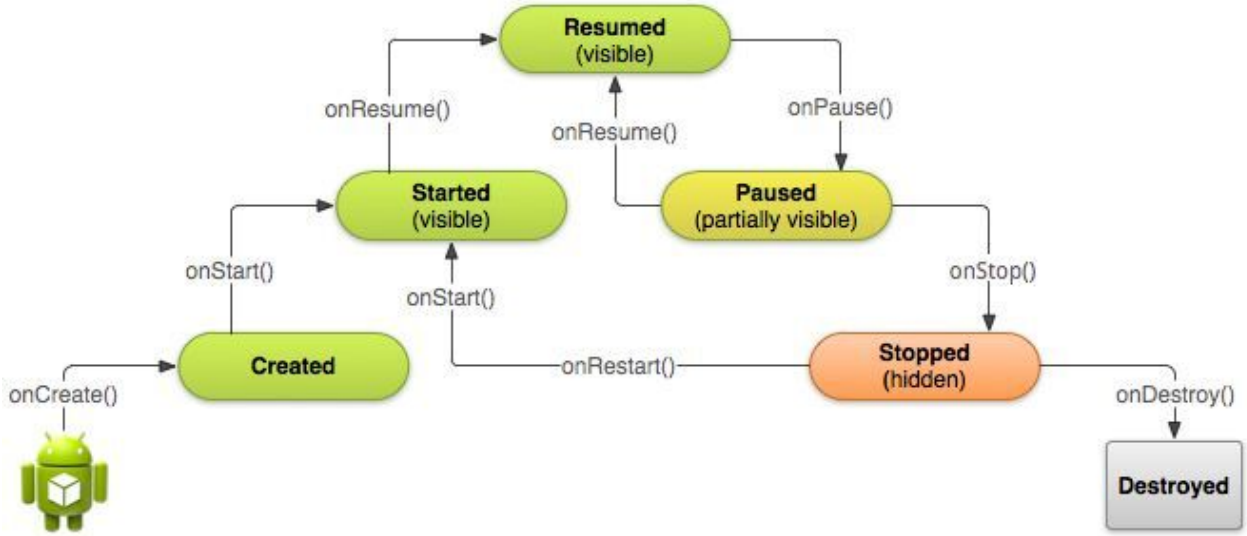
结合图说说Activity生命周期，比如暂停Activity的流程如下：

- `线程1`的AMS中调用`线程2`的ATP来发送事件；（由于同一个进程的线程间资源共享，可以相互直接调用，但需要注意多线程并发问题）
- `线程2`通过binder将暂停Activity的事件传输到App进程的`线程4`；
- `线程4`通过handler消息机制，将暂停Activity的消息发送给`主线程`；
- `主线程`在looper.loop()中循环遍历消息，当收到暂停Activity的消息(`PAUSE_ACTIVITY`)时，便将消息分发给ActivityThread.H.handleMessage()方法，再经过方法的层层调用，最后便会调用到Activity.onPause()方法。

这便是由AMS完成了onPause()控制，那么同理Activity的其他生命周期也是这么个流程来进行控制的。

AMS的小知识

1. Activity状态



- Resumed（运行状态）：Activity处于前台，且用户可以与其交互。
- Paused（暂停状态）: Activity被在前台中处于半透明状态或者未覆盖全屏的其他Activity部分遮挡。 暂停的Activity不会接收用户输入，也无法执行任何代码。
- Stopped（停止状态）：Activity被完全隐藏，且对用户不可见；被视为后台Activity。 停止的Activity实例及其诸如成员变量等所有状态信息将保留，但它无法执行任何代码。

除此之外，其他状态都是过渡状态(或称为暂时状态)，比如onCreate()，onStart()后很快就会调用onResume()方法。

2. onPause与onResume

> 假设当前Activity为A，如果这时用户打开一个新的Activity B，那么B的onResume和A的onPause哪个先执行？

这个问题可以从Android的源码中得到解释。在ActivityStack中的resumeTopActivityInnerLocked方法中，有这么一段代码:

```
private boolean resumeTopActivityInnerLocked(ActivityRecord prev, Bundle options) {
    ...
    // We need to start pausing the current activity so the top one
    // can be resumed...
    boolean dontWaitForPause = (next.info.flags&ActivityInfo.FLAG_RESUME_WHILE_PAUSING) != 0;
    boolean pausing = mStackSupervisor.pauseBackStacks(userLeaving, true, dontWaitForPause);
    if (mResumedActivity != null) {
        if (DEBUG_STATES) Slog.d(TAG_STATES,
                "resumeTopActivityLocked: Pausing " + mResumedActivity);
        pausing |= startPausingLocked(userLeaving, false, true, dontWaitForPause);
    }
    ...
}
```

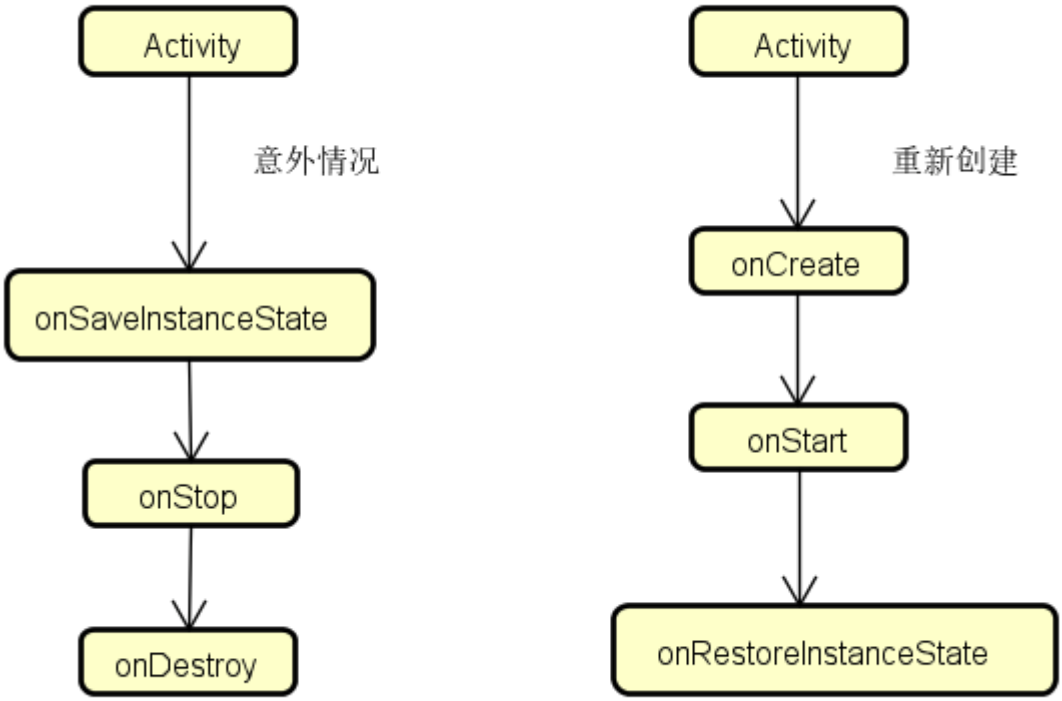
从上述代码可以看出，在新的Activity启动之前，栈顶的Activity需要先onPause后，新Activity才能启动。

3. OnSaveInstanceState时机

> Android calls onSaveInstanceState() before the activity becomes vulnerable to being destroyed by the system, but does not bother calling it when the instance is actually being destroyed by a user action (such as pressing the BACK key)

当某个activity是在异常情况下终止的，其onPause、onStop、OnDestory均会被调用，系统会调用onSaveInstanceState来保存当前Activity的状态。这个方法的调用时机是在onStop之前，它和onPause没有既定的时序关系，它既可能在onPause之前调用，也可能在onPause之后调用。需要注意的一点是，这个方法只会出现在Activity被异常终止的情况下，正常情况下系统不会回调这个方法。

当Activity被重新创建后，系统会调用onRestoreInstance，并且把Activity销毁onSaveInstanceState方法所保存的Bundle对象作为参数同时传递给onRestoreInstanceState和OnCreate。因此，我们可以通过onRestoreInstanceState和onCreate方法来判断Activity是否被重建了，如果被重建了，那么我们就可以取出之前保存的数据并恢复，从时序上来说，OnRestoreInstanceState的调用时机在OnStart之后。



onSaveInstanceState方法会在什么时候被执行，有这么几种情况：

- 当用户按下HOME键时；
- 长按Home键，选择运行其它的程序时；
- 按下电源按键（关闭屏幕显示）时；
- 从Activity A中启动一个新的Activity时；
- 屏幕方向切换时，例如从竖屏切换为横屏时。

总而言之，onSaveInstanceState的调用遵循一个重要原则，即当系统“未经你许可”时销毁了你的activity，则onSaveInstanceState会被系统调用，这是系统的责任，因为它必须要提供一个机会让你保存你的数据。

4. 查看应用启动速度

APP的启动速度大致可以分为三类：

1. 应用首次安装之后首次启动的时间；
2. 应用非首次启动的时间；
3. 应用存活在后台，重新进入应用的时间。

你想比较的是哪一个，同时也应该尽量保证测试环境的一致，比如说同时没有其他的应用程序运行在后台等等，减少误差。

测试方法有以下五种：

1. 使用秒表来计算

这是最简单、最容易操作、同时误差也最大的方法。

1. 使用高速相机拍摄后用视频播放器读毫秒

这个方法误差不大，但是需要特定的仪器辅助，并且耗时耗力。

1. 通过屏幕录制命令来读毫秒

使用adb shell screenrecord –bugreport /sdcard/xx.mp4，这样抓出来的视频包含帧数和毫秒数。这个方法不需要高速相机，但是通过视频播放器读毫秒耗费时间。

1. 使用adb命令来测试启动速度

- 清除掉所有后台运行的应用程序。
- 打开命令行，执行 `adb shell am start -W -n 包名/类名` 。

```
PS D:\> adb shell am start -W -n csumissu.sfc.android/.MainActivity
Starting: Intent { cmp=csumissu.sfc.android/.MainActivity }
Status: ok
Activity: csumissu.sfc.android/.MainActivity
ThisTime: 304
TotalTime: 304
WaitTime: 335
Complete
```

WaitTime就是总的耗时，包括前一个应用Activity pause的时间和新应用启动的时间；ThisTime表示一连串启动Activity的最后一个Activity的启动耗时；TotalTime表示新应用启动的耗时，包括新进程的启动和Activity的启动，但不包括前一个应用Activity pause的耗时。

总结一下，如果只关心某个应用自身启动耗时，参考TotalTime；如果关心系统启动应用耗时，参考WaitTime；如果关心应用界面Activity启动耗时，参考ThisTime。

1. 通过AMS的日志查看启动速度

```
ActivityRecord.java ×
855
856     private void reportLaunchTimeLocked(final long curTime) {
857         final ActivityStack stack = task.stack;
858         final long thisTime = curTime - displayStartTime;
859         final long totalTime = stack.mLaunchStartTime != 0
860                 ? (curTime - stack.mLaunchStartTime) : thisTime;
861         if (ActivityManagerService.SHOW_ACTIVITY_START_TIME) {
862             Trace.asyncTraceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER, "launching", 0);
863             EventLog.writeEvent(EventLogTags.AM_ACTIVITY_LAUNCH_TIME,
864                     userId, System.identityHashCode(this), shortComponentName,
865                     thisTime, totalTime);
866             StringBuilder sb = service.mStringBuilder;
867             sb.setLength(0);
868             sb.append("Displayed ");
869             sb.append(shortComponentName);
870             sb.append(": ");
871             TimeUtils.formatDuration(thisTime, sb);
872             if (thisTime != totalTime) {
873                 sb.append(" (total ");
874                 TimeUtils.formatDuration(totalTime, sb);
875                 sb.append(")");
876             }
877             Log.i(ActivityManagerService.TAG, sb.toString());
878         }
879         mStackSupervisor.reportActivityLaunchedLocked(false, this, thisTime, totalTime);
```

这里的thisTime、totalTime与上面通过am start命令得到的启动速度的表述是同一个意思。

5. 多进程模式

正常情况下，在Android中多进程是指一个应用中存在多个进程的情况，因此这里不讨论两个应用之间的多进程情况。首先，在Android中使用多进程只有一种方法，那就是给四大组件（Activity、Service、Receiver、ContentProvider）在AndroidManifest中指定android:process属性，除此之外没有其他办法，也就是说我们无法给一个线程或者给一个实体类指定其运行时所在的进程。其实还有另一种非常规的多进程的方法，那就是通过JNI在native层去fork一个新的进程，但是这种方法属于特殊情况，也不是常用的创建多进程的方式，因为我们暂时不考虑这种方式。

下面是一个示例：

```
<!-- 进程为 csumissu -->
<activity android:name=".AActivity"/>
<!-- 进程为 csumissu:remote -->
<activity android:name=".BActivity" android:process=":remote"/>
<!-- 进程为 csumissu.remote -->
<activity android:name=".CActivity" android:process="csumissu.remote"/>
```

如果没有为四大组件指定process属性，那么它运行在默认进程中，默认进程的进程名就是包名。另外注意下B Activity与C Activity的android:process属性分别为`csumissu:remote`和`csumissu.remote`，那么这两种方式有区别吗？其实是有区别的，区别有两方面：

- 首先，“:”的含义是指要在当前的进程名前面附加上当前的包名，这是一种简写的方法，通过ps命令可以看到B Activity的完整进程名为`csumissu:remote`，而对于C Activity中的声明方式，它是一种完整的命名方式，不会附加包名信息。

- 其次，进程名以“:”开头的进程属于当前应用的私有进程，其他应用的组件不可以和它跑在同一个进程中，而进程名不以“:”开头的进程属于全局进程，其他应用通过ShareUID方式可以和它跑在同一个进程中。

我们知道Android系统会为每个应用分配一个唯一的UID，具有相同UID的应用才能共享数据。这里要说明的是，两个应用通过ShareUID跑在同一个进程中是有要求的，需要这两个应用有相同的ShareUID并且签名相同才可以。在这种情况下，它们可以互相访问对方的私有数据，比如data目录、组件信息等，不管它们是否跑在同一个进程中。当然如果它们跑在同一个进程中，那么除了能共享data目录、组件信息，还可以共享内存数据，或者说它们看起来就像是一个应用的两个部分。

> 假设应用中有个类叫DataManager，其静态变量sUserId默认值为1，在A Activity中将sUserId重新赋值为2，打印出这个静态变量后启动B Activity，此时在B Activity再打印一下sUserId的值，会是多少？

我们知道，Android为每一个应用分配了一个独立的虚拟机，或者说为每个进程都分配一个独立的虚拟机，不同虚拟机在内存分配上有不同的地址空间，这就是导致在不同的虚拟机中访问同一个类的对象会产生多分副本。拿我们的例子来说，在进程`csumissu`和进程`csumissu:remote`中都存在一个DataManager类，并且这两个类是互不干扰的，在一个进程中修改sUserId的值只会影响当前进程，对其他进程不会造成任何影响，这样就可以理解为什么在A Activity中修改了sUserId的值，但是在B Activity中sUserId的值却没有发生改变这个现象。

所有运行在不同进程中的四大组件，只要它们之间需要通过内存来共享数据，都会共享失败，这也是多进程所带来的主要影响。

一般来说，使用多进程会造成如下几方面的问题：

1. 静态成员和单例模式完全失效；
2. 线程同步机制完全失效；
3. SharedPreferences的可靠性下降；
4. Application会多次创建。

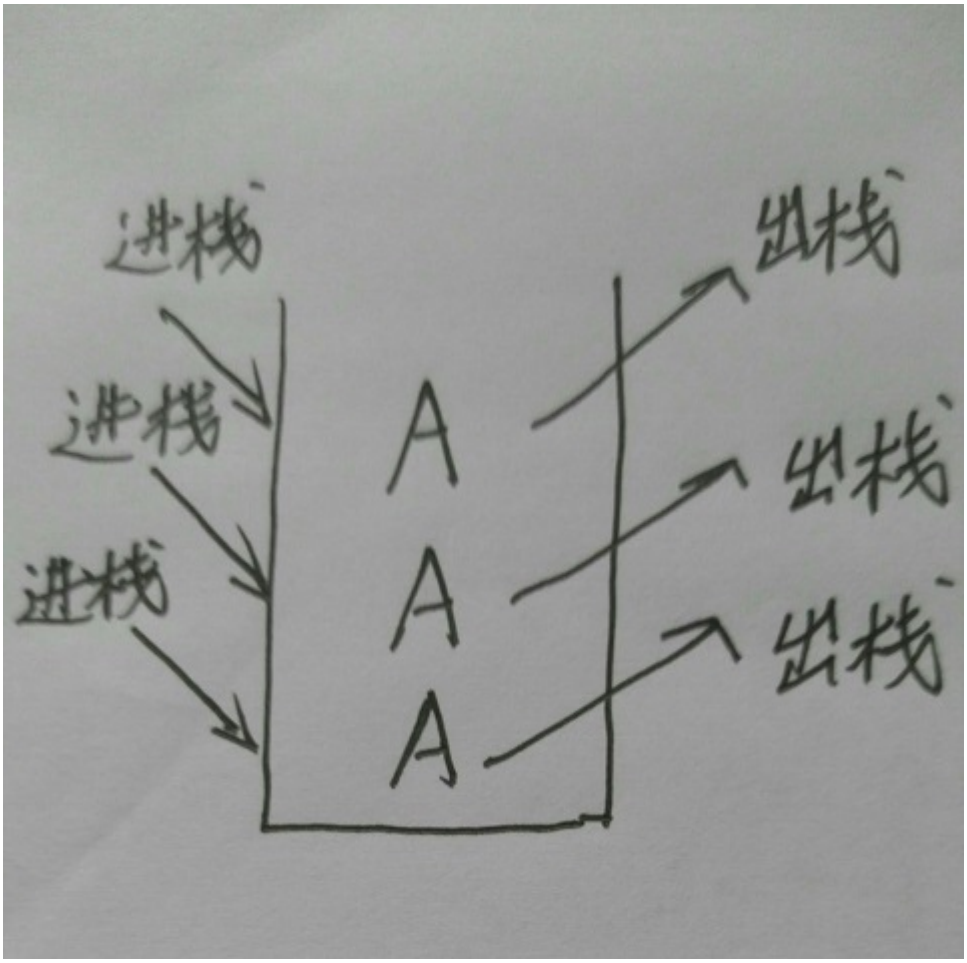
第二个是由于SharedPreferences不支持两个进程同时去执行写操作，否则会导致一定几率的数据丢失，这是因为SharedPreferences底层是通过读/写XML文件来实现的，并发写显示是可能出问题的。

6. Activity的启动模式（LaunchMode）

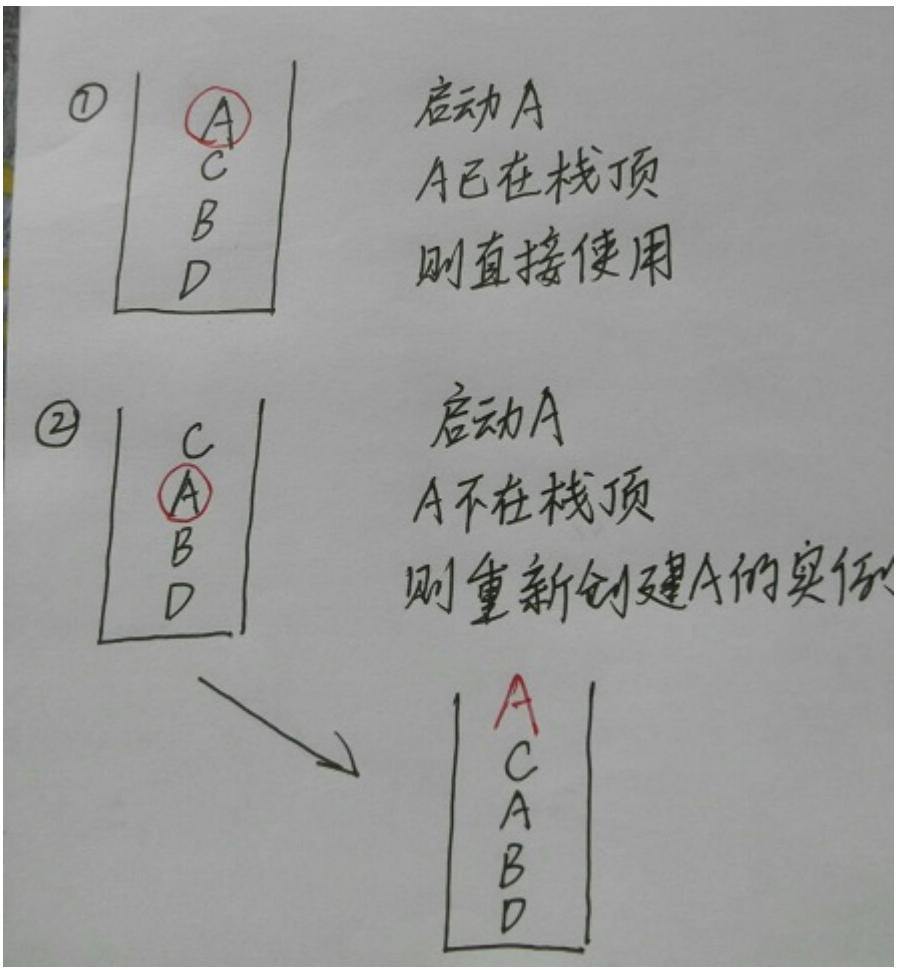
首先说一下Activity为什么需要启动模式。我们知道，在默认情况下，我们多次启动同一个Activity的时候，系统会创建多个实例并把它们一一放入任务栈中，当我们单击back键，会发现这些Activity会一一回退。任务栈是一种“后进先出”的栈结构，这个比较好理解，每按一下back键就会有一个Activity出栈，直到栈空为止，当栈中无任何Activity的时候，系统就会回收这个任务栈。

知道了Activity的默认启动以后，我们可能就会发现一个问题：多次启动同一个Activity，系统重复创建多个实例，这样不是很傻吗？这样的确有点傻，Android在设计的时候不可能不考虑到这个问题，所以它提供了启动模式来修改系统的默认行为。目前有四种启动模式：standard、singleTop、singleTask和singleInstance，下面介绍各种启动模式的含义：

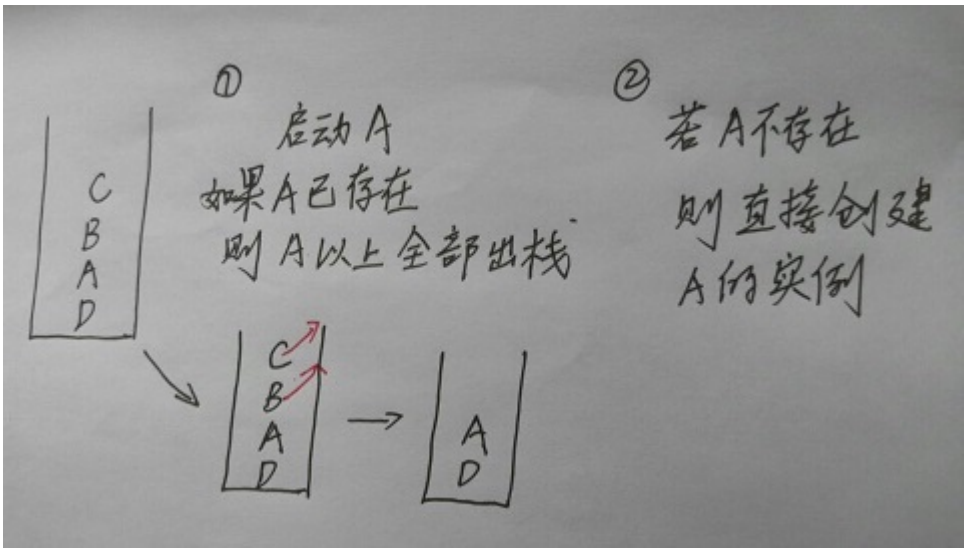
1. standard：标准模式，这也是系统的默认模式。每次启动一个Activity都会重新创建一个新的实例，不管这个实例是否已经存在。一个任务栈中可以有多个实例，每个实例也可以属于不同的任务栈。在这种模式下，谁启动了这个Activity，那么这个Activity就运行在启动它的那个Activity所在的栈中。



1. singleTop：栈顶复用模式。在这种模式下，如果新Activity已经位于任务栈的栈顶，那么此Activity不会被重复创建，同时它的onNewIntent方法会被回调，通过此方法的参数我们可以取出当前请求的信息。需要注意的是，这个Activity的onCreate、onStart不会被系统调用，因为它并没有发生改变。如果新Activity的实例已经存在但不是位于栈顶，那么新Activity仍然会重新重建。



1. singleTask：栈内复用模式。这是一种单实例模式，在这种模式下，只要Activity在一个栈中存在，那么多次启动此Activity都不会重新创建实例，和singleTop一样，系统也会回调其onNewIntent。具体一点，当一个具有singleTask模式的Activity请求启动后，比如Activity A，系统首先会寻找是否存在A想要的任务栈，如果不存在，就重新创建一个任务栈，然后创建A的实例后把A放到栈中。如果存在A所需的任务栈，这时要看A是否在栈中有实例存在，如果有实例存在，那么系统就会把A调到栈顶并调用它的onNewIntent方法，如果实例不存在，就创建A的实例并把A压入栈中。



1. singleInstance：单实例模式。这是一种加强的singleTask模式，它除了具有singleTask模式的所有特性外，还加强了一点，那就是具有此种模式的Activity只能单独地位于一个任务栈中，换句话说，比如Activity A是

singleInstance模式，当A启动后，系统会为它创建一个新的任务栈，然后A独自在这个新的任务栈中，由于栈内复用的特性，后续的请求均不会创建新的Activity，除非这个独特的任务栈被系统销毁了。