

2019Android Framework 高频面试题总结

1. AMS 、 PMS

1.AMS 概述

AMS 是系统的引导服务，应用进程的启动、切换和调度、四大组件的启动和管理都需要 AMS 的支持。从这里可以看出 AMS 的功能会十分的繁多，当然它并不是一个类承担这个重责，它有一些关联类，这在文章后面会讲到。AMS 的涉及的知识点非常多，这篇文章主要会讲解 AMS 的以下几个知识点：

- AMS 的启动流程。
- AMS 与进程启动。
- AMS 家族。

2.AMS 的启动流程

AMS 的启动是在 SyetemServer 进程中启动的，在 [Android 系统启动流程（三）解析 SyetemServer 进程启动过程](#) 这篇文章中提及过，这里从 SyetemServer 的 main 方法开始讲起：

frameworks/base/services/java/com/android/server/SystemServer.java

```
public static void main(String[] args) {  
    new SystemServer().run();  
}
```

main 方法中只调用了 SystemServer 的 run 方法，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
private void run() {  
    ...  
    System.loadLibrary("android_servers");//1  
    ...  
    mSystemServiceManager = new  
SystemServiceManager(mSystemContext);//2  
    LocalServices.addService(SystemServiceManager.class,  
mSystemServiceManager);  
    ...  
    try {  
        Trace.traceBegin(Trace.TRACE_TAG_SYSTEM_SERVER,  
"StartServices");
```

```

        startBootstrapServices();//3
        startCoreServices();//4
        startOtherServices();//5
    } catch (Throwable ex) {
        Slog.e("System",
"*****");
        Slog.e("System", "***** Failure starting system
services", ex);
        throw ex;
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
    }
    ...
}

```

在注释 1 处加载了动态库 `libandroid_servers.so`。接下来在注释 2 处创建 `SystemServiceManager`，它会对系统的服务进行创建、启动和生命周期管理。在注释 3 中的 `startBootstrapServices` 方法中用 `SystemServiceManager` 启动了 `ActivityManagerService`、`PowerManagerService`、`PackageManagerService` 等服务。在注释 4 处的 `startCoreServices` 方法中则启动了 `BatteryService`、`UsageStatsService` 和 `WebViewUpdateService`。注释 5 处的 `startOtherServices` 方法中启动了 `CameraService`、`AlarmManagerService`、`VrManagerService` 等服务。这些服务的父类均为 `SystemService`。从注释 3、4、5 的方法可以看出，官方把系统服务分为了三种类型，分别是引导服务、核心服务和其他服务，其中其他服务是一些非紧要和一些不需要立即启动的服务。系统服务总共大约有 80 多个，我们主要来查看引导服务 AMS 是如何启动的，注释 3 处的 `startBootstrapServices` 方法如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```

private void startBootstrapServices() {
    Installer installer =
mSystemServiceManager.startService(Installer.class);
    // Activity manager runs the show.
    mActivityManagerService = mSystemServiceManager.startService(
ActivityManagerService.Lifecycle.class).getService();//1

mActivityManagerService.setSystemServiceManager(mSystemServiceManager)
;
    mActivityManagerService.setInstaller(installer);
    ...
}

```

在注释 1 处调用了 `SystemServiceManager` 的 `startService` 方法，方法的参数是 `ActivityManagerService.Lifecycle.class`：

frameworks/base/services/core/java/com/android/server/SystemServiceManager.java

```
@SuppressWarnings("unchecked")
public <T extends SystemService> T startService(Class<T> serviceClass)
{
    try {
        ...
        final T service;
        try {
            Constructor<T> constructor =
serviceClass.getConstructor(Context.class); //1
            service = constructor.newInstance(mContext); //2
        } catch (InstantiationException ex) {
            ...
        }
        // Register it.
        mServices.add(service); //3
        // Start it.
        try {
            service.onStart(); //4
        } catch (RuntimeException ex) {
            throw new RuntimeException("Failed to start service " + name
                + ": onStart threw an exception", ex);
        }
        return service;
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
    }
}
```

`startService` 方法传入的参数是 `Lifecycle.class`，`Lifecycle` 继承自 `SystemService`。首先，通过反射来创建 `Lifecycle` 实例，注释 1 处得到传进来的 `Lifecycle` 的构造器 `constructor`，在注释 2 处调用 `constructor` 的 `newInstance` 方法来创建 `Lifecycle` 类型的 `service` 对象。接着在注释 3 处将刚创建的 `service` 添加到 `ArrayList` 类型的 `mServices` 对象中来完成注册。最后在注释 4 处调用 `service` 的 `onStart` 方法来启动 `service`，并返回该 `service`。`Lifecycle` 是 AMS 的内部类，代码如下所示。

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
public static final class Lifecycle extends SystemService {
    private final ActivityManagerService mService;
    public Lifecycle(Context context) {
        super(context);
        mService = new ActivityManagerService(context); //1
    }
}
```

```

    }
    @Override
    public void onStart() {
        mService.start();//2
    }
    public ActivityManagerService getService() {
        return mService;//3
    }
}

```

上面的代码结合 `SystemServiceManager` 的 `startService` 方法来分析，当通过反射来创建 `Lifecycle` 实例时，会调用注释 1 处的方法创建 AMS 实例，当调用 `Lifecycle` 类型的 `service` 的 `onStart` 方法时，实际上是调用了注释 2 处 AMS 的 `start` 方法。在 `SystemService` 的 `startBootstrapServices` 方法的注释 1 处，调用了如下代码：

```

mActivityManagerService = mSystemServiceManager.startService(
    ActivityManagerService.Lifecycle.class).getService();

```

我们知道 `SystemServiceManager` 的 `startService` 方法最终会返回 `Lifecycle` 类型的对象，紧接着又调用了 `Lifecycle` 的 `getService` 方法，这个方法会返回 AMS 类型的 `mService` 对象，见注释 3 处，这样 AMS 实例就会被创建并且返回。

3.AMS 与进程启动

在 [Android 系统启动流程（二）解析 Zygote 进程启动过程](#) 这篇文章中，我提到了 `Zygote` 的 Java 框架层中，会创建一个 Server 端的 `Socket`，这个 `Socket` 用来等待 AMS 来请求 `Zygote` 来创建新的应用程序进程。要启动一个应用程序，首先要保证这个应用程序所需要的应用程序进程已经被启动。AMS 在启动应用程序时会检查这个应用程序需要的应用程序进程是否存在，不存在就会请求 `Zygote` 进程将需要的应用程序进程启动。`Service` 的启动过程中会调用 `ActiveServices` 的 `bringUpServiceLocked` 方法，如下所示。

frameworks/base/services/core/java/com/android/server/am/ActiveServices.java

```

private String bringUpServiceLocked(ServiceRecord r, int intentFlags,
boolean execInFg,
    boolean whileRestarting, boolean permissionsReviewRequired)
    throws TransactionTooLargeException {
    ...
    final String procName = r.processName;//1
    ProcessRecord app;
    if (!isolated) {
        app = mAm.getProcessRecordLocked(procName, r.appInfo.uid,
false);//2
    }
}

```

```

        if (DEBUG_MU) Slog.v(TAG_MU, "bringUpServiceLocked:
appInfo.uid=" + r.appInfo.uid
        + " app=" + app);
        if (app != null && app.thread != null) { //3
            try {
                app.addPackage(r.appInfo.packageName,
r.appInfo.versionCode,
                mAm.mProcessStats);
                realStartServiceLocked(r, app, execInFg); //4
                return null;
            } catch (TransactionTooLargeException e) {
                ...
            }
        } else {
            app = r.isolatedProc;
        }
        if (app == null && !permissionsReviewRequired) { //5
            if ((app=mAm.startProcessLocked(procName, r.appInfo, true,
intentFlags,
                "service", r.name, false, isolated, false)) == null)
                { //6
                    ...
                }
            if (isolated) {
                r.isolatedProc = app;
            }
        }
        ...
    }
}

```

在注释 1 处得到 **ServiceRecord** 的 **processName** 的值赋值给 **procName**，其中 **ServiceRecord** 用来描述 **Service** 的 **android:process** 属性。注释 2 处将 **procName** 和 **Service** 的 **uid** 传入到 AMS 的 **getProcessRecordLocked** 方法中，来查询是否存在一个与 **Service** 对应的 **ProcessRecord** 类型的对象 **app**，**ProcessRecord** 主要用来记录运行的应用程序进程的信息。注释 5 处判断 **Service** 对应的 **app** 为 **null** 则说明用来运行 **Service** 的应用程序进程不存在，则调用注释 6 处的 AMS 的 **startProcessLocked** 方法来创建对应的应用程序进程，具体的过程请查看 [Android 应用程序进程启动过程（前篇）](#)。

4.AMS 家族

ActivityManager 是一个和 AMS 相关联的类，它主要对运行中的 **Activity** 进行管理，这些管理工作并不是由 **ActivityManager** 来处理的，而是交由 AMS 来处理，**ActivityManager** 中的方法会通过 **ActivityManagerNative**（以后简称 **AMN**）的 **getDefault** 方法来得到 **ActivityManagerProxy**（以后简称 **AMP**），通过 **AMP** 就可以和

AMN 进行通信，而 AMN 是一个抽象类，它会将功能交由它的子类 AMS 来处理，因此，AMP 就是 AMS 的代理类。AMS 作为系统核心服务，很多 API 是不会暴露给 ActivityManager 的，因此 ActivityManager 并不算是 AMS 家族一份子。为了讲解 AMS 家族，这里拿 Activity 的启动过程举例，Activity 的启动过程中会调用 Instrumentation 的 execStartActivity 方法，如下所示。

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) {
    ...
    try {
        intent.migrateExtraStreamToClipData();
        intent.prepareToLeaveProcess(who);
        int result = ActivityManagerNative.getDefault()
            .startActivity(whoThread, who.getBasePackageName(),
intent,

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

execStartActivity 方法中会调用 AMN 的 getDefault 来获取 AMS 的代理类 AMP。接着调用了 AMP 的 startActivity 方法，先来查看 AMN 的 getDefault 方法做了什么，如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
static public IActivityManager getDefault() {
    return gDefault.get();
}

private static final Singleton<IActivityManager> gDefault = new
Singleton<IActivityManager>() {
    protected IActivityManager create() {
        IBinder b = ServiceManager.getService("activity");//1
        if (false) {
            Log.v("ActivityManager", "default service binder = " + b);
        }
        IActivityManager am = asInterface(b);//2
    }
}
```

```

        if (false) {
            Log.v("ActivityManager", "default service = " + am);
        }
        return am;
    }+
};}

```

getDefault 方法调用了 gDefault 的 get 方法，我们接着往下看，gDefault 是一个 Singleton 类。注释 1 处得到名为“activity”的 Service 引用，也就是 IBinder 类型的 AMS 的引用。接着在注释 2 处将它封装成 AMP 类型对象，并将它保存到 gDefault 中，此后调用 AMN 的 getDefault 方法就会直接获得 AMS 的代理对象 AMP。注释 2 处的 asInterface 方法如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```

static public IActivityManager asInterface(IBinder obj) {
    if (obj == null) {
        return null;
    }
    IActivityManager in =
        (IActivityManager)obj.queryLocalInterface(descriptor);
    if (in != null) {
        return in;
    }
    return new ActivityManagerProxy(obj);}

```

asInterface 方法的主要作用就是将 IBinder 类型的 AMS 引用封装成 AMP，AMP 的构造方法如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```

class ActivityManagerProxy implements IActivityManager{
    public ActivityManagerProxy(IBinder remote)
    {
        mRemote = remote;
    }...
}

```

AMP 的构造方法中将 AMS 的引用赋值给变量 mRemote，这样在 AMP 中就可以使用 AMS 了。

其中 IActivityManager 是一个接口，AMN 和 AMP 都实现了这个接口，用于实现代理模式和 Binder 通信。

再回到 Instrumentation 的 execStartActivity 方法，来查看 AMP 的 startActivity 方法，AMP 是 AMN 的内部类，代码如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```

public int startActivity(IApplicationThread caller, String
callingPackage, Intent intent,
        String resolvedType, IBinder resultTo, String resultWho, int
requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle options)
throws RemoteException {
    ...
    data.writeInt(requestCode);
    data.writeInt(startFlags);
    ...
    mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0); //1
    reply.readException();+
    int result = reply.readInt();
    reply.recycle();
    data.recycle();
    return result;
}

```

首先会将传入的参数写入到 Parcel 类型的 data 中。在注释 1 处，通过 IBinder 类型对象 mRemote（AMS 的引用）向服务端的 AMS 发送一个 START_ACTIVITY_TRANSACTION 类型的进程间通信请求。那么服务端 AMS 就会从 Binder 线程池中读取我们客户端发来的数据，最终会调用 AMN 的 onTransact 方法，如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```

@Override
public boolean onTransact(int code, Parcel data, Parcel reply, int
flags)
    throws RemoteException {
    switch (code) {
        case START_ACTIVITY_TRANSACTION:
        {
            ...
            int result = startActivity(app, callingPackage, intent,
resolvedType,
                resultTo, resultWho, requestCode, startFlags,
profilerInfo, options);
            reply.writeNoException();
            reply.writeInt(result);
            return true;
        }
    }
}

```


onTransact 中会调用 AMS 的 startActivity 方法，如下所示。

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
@Override
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());
}
```

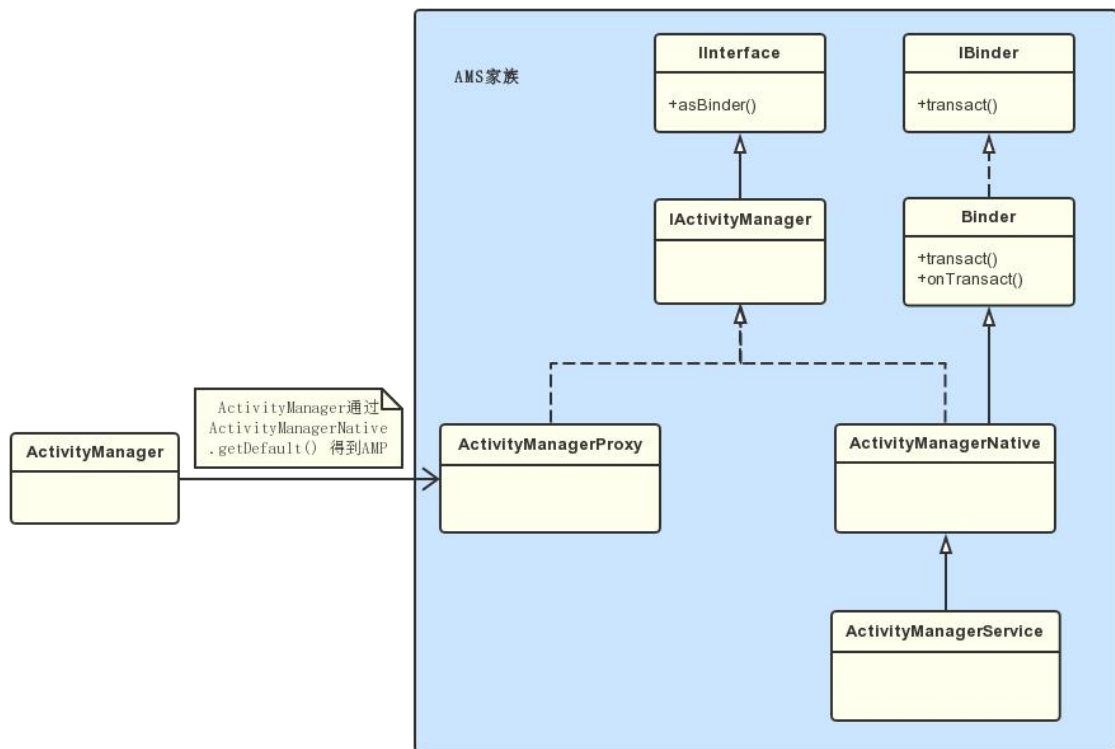
startActivity 方法会最后 return startActivityAsUser 方法，如下所示。

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
@Override
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) {
    enforceNotIsolatedCaller("startActivity");
    userId = mUserController.handleIncomingUser(Binder.getCallingPid(),
Binder.getCallingUid(),
        userId, false, ALLOW_FULL_ONLY, "startActivity", null);
    return mActivityStarter.startActivityMayWait(caller, -1,
callingPackage, intent,
        resolvedType, null, null, resultTo, resultWho, requestCode,
startFlags,
        profilerInfo, null, null, bOptions, false, userId, null,
null);
}
```

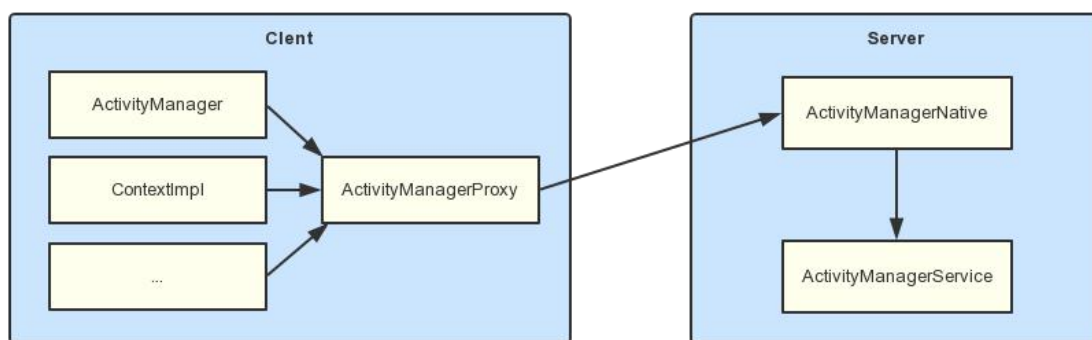
startActivityAsUser 方法最后会 return ActivityStarter 的 startActivityMayWait 方法，这一调用过程已经脱离了本节要讲的 AMS 家族，因此这里不做介绍了，具体的调用过程可以查看 [Android 深入四大组件（一）应用程序启动过程（后篇）](#) 这篇文章。

在 Activity 的启动过程中提到了 AMP、AMN 和 AMS，它们共同组成了 AMS 家族的主要部分，如下图所示。



AMP 是 AMN 的内部类，它们都实现了 **IActivityManager** 接口，这样它们就可以实现代理模式，具体来讲是远程代理：AMP 和 AMN 是运行在两个进程的，AMP 是 Client 端，AMN 则是 Server 端，而 Server 端中具体的功能都是由 AMN 的子类 AMS 来实现的，因此，AMP 就是 AMS 在 Client 端的代理类。AMN 又实现了 **Binder** 类，这样 AMP 可以和 AMS 就可以通过 **Binder** 来进行进程间通信。

ActivityManager 通过 AMN 的 `getDefault` 方法得到 AMP，通过 AMP 就可以和 AMN 进行通信，也就是间接的与 AMS 进行通信。除了 **ActivityManager**，其他想要与 AMS 进行通信的类都需要通过 AMP，如下图所示。



PMS 之前言

PMS 的创建过程分为两个部分进行讲解，分别是 SyetemServer 处理部分和 PMS 构造方法。其中 SyetemServer 处理部分和 AMS 和 WMS 的创建过程是类似的，可以将它们进行对比，这样可以更好的理解和记忆这一知识点。

1. PMS 之 SyetemServer 处理部分

PMS 是在 SyetemServer 进程中被创建的，SyetemServer 进程用来创建系统服务，不了解它的可以查看 [Android 系统启动流程（三）解析 SyetemServer 进程启动过程](#) 这篇文章。

从 SyetemServer 的入口方法 main 方法开始讲起，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
public static void main(String[] args) {
    new SystemServer().run();
}
```

main 方法中只调用了 SystemServer 的 run 方法，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
private void run() {
    try {
        ...
        //创建消息 Looper
        Looper.prepareMainLooper();
        //加载了动态库 libandroid_servers.so
        System.loadLibrary("android_servers");//1
        performPendingShutdown();
        // 创建系统的 Context
        createSystemContext();
    }
}
```

```

        // 创建 SystemServiceManager
        mSystemServiceManager = new
SystemServiceManager(mSystemContext); //2
        mSystemServiceManager.setRuntimeRestarted(mRuntimeRestart);
        LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);
        SystemServerInitThreadPool.get();
    } finally {
        traceEnd();
    }
    try {
        traceBeginAndSlog("StartServices");
        //启动引导服务
        startBootstrapServices(); //3
        //启动核心服务
        startCoreServices(); //4
        //启动其他服务
        startOtherServices(); //5
        SystemServerInitThreadPool.shutdown();
    } catch (Throwable ex) {
        Slog.e("System", "*****");
        Slog.e("System", "***** Failure starting system services",
ex);
        throw ex;
    } finally {
        traceEnd();
    }
    ...}

```

在注释 1 处加载了动态库 libandroid_servers.so。接下来在注释 2 处创建 **SystemServiceManager**，它会对系统的服务进行创建、启动和生命周期管理。在注释 3 中的 **startBootstrapServices** 方法中用 **SystemServiceManager** 启动了 **ActivityManagerService**、**PowerManagerService**、**PackageManagerService** 等服务。在注释 4 处的 **startCoreServices** 方法中则启动了 **DropBoxManagerService**、**BatteryService**、**UsageStatsService** 和 **WebViewUpdateService**。注释 5 处的 **startOtherServices** 方法中启动了 **CameraService**、**AlarmManagerService**、**VrManagerService** 等服务。这些服务的父类均为 **SystemService**。从注释 3、4、5 的方法可以看出，官方把系统服务分为了三种类型，分别是引导服务、核心服务和其他服务，其中其他服务是一些非紧要和一些不需要立即启动的服务。这些系统服务总共有 100 多个，我们熟知的 **AMS** 属于引导服务，**WMS** 属于其他服务，本文要讲的 **PMS** 属于引导服务，因此这里列出引导服务以及它们的作用，见下表。

引导服务

作用

引导服务	作用
Installer	系统安装 apk 时的一个服务类，启动完成 Installer 服务之后才能启动其他的系统服务
ActivityManagerService	负责四大组件的启动、切换、调度。
PowerManagerService	计算系统中和 Power 相关的计算，然后决策系统应该如何反应
LightsService	管理和显示背光 LED
DisplayManagerService	用来管理所有显示设备
UserManagerService	多用户模式管理
SensorService	为系统提供各种感应器服务
PackageManagerService	用来对 apk 进行安装、解析、删除、卸载等等操作

查看启动引导服务的注释 3 处的 startBootstrapServices 方法。

frameworks/base/services/java/com/android/server/SystemServer.java

```
private void startBootstrapServices() {
    ...
    traceBeginAndSlog("StartPackageManagerService");
    mPackageManagerService = PackageManagerService.main(mSystemContext,
    installer,
        mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF,
    mOnlyCore); //1
    mFirstBoot = mPackageManagerService.isFirstBoot(); //2
    mPackageManager = mSystemContext.getPackageManager();
    traceEnd();
    ...}

```

注释 1 处的 PMS 的 main 方法主要用来创建 PMS，其中最后一个参数 mOnlyCore 代表是否只扫描系统的目录，它在本篇文章中会出现多次，一般情况下它的值为 false。注释 2 处获取 boolean 类型的变量 mFirstBoot，它用于表示 PMS 是否首次被启动。mFirstBoot 是后续 WMS 创建时所需要的参数，从这里就可以看出系统服务之间是有依赖关系的，它们的启动顺序不能随意被更改。

2. PMS 构造方法

PMS 的 main 方法如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
public static PackageManagerService main(Context context, Installer
installer,
    boolean factoryTest, boolean onlyCore) {

```

```

        PackageManagerServiceCompilerMapping.checkProperties();
        PackageManagerService m = new PackageManagerService(context,
installer,
            factoryTest, onlyCore);
        m.enableSystemUserPackages();
        ServiceManager.addService("package", m);
        return m;
    }

```

main 方法主要做了两件事，一个是创建 PMS 对象，另一个是将 PMS 注册到 ServiceManager 中。

PMS 的构造方法大概有 600 多行，分为 5 个阶段，每个阶段会打印出相应的 EventLog，EventLog 用于打印 Android 系统的事件日志。

1. BOOT_PROGRESS_PMS_START（开始阶段）
2. BOOT_PROGRESS_PMS_SYSTEM_SCAN_START（扫描系统阶段）
3. BOOT_PROGRESS_PMS_DATA_SCAN_START（扫描 Data 分区阶段）
4. BOOT_PROGRESS_PMS_SCAN_END（扫描结束阶段）
5. BOOT_PROGRESS_PMS_READY（准备阶段）

2.1 开始阶段

PMS 的构造方法中会获取一些包管理需要属性，如下所示。

frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```

public PackageManagerService(Context context, Installer installer,
    boolean factoryTest, boolean onlyCore) {
    LockGuard.installLock(mPackages, LockGuard.INDEX_PACKAGES);
    Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "create package
manager");
    //打印开始阶段日志
    EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_START,
        SystemClock.uptimeMillis())
    ...
    //用于存储屏幕的相关信息
    mMetrics = new DisplayMetrics();
    //Settings 用于保存所有包的动态设置
    mSettings = new Settings(mPackages);
    //在 Settings 中添加多个默认的 sharedUserId
    mSettings.addSharedUserLPw("android.uid.system",
Process.SYSTEM_UID,
        ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);//1
    mSettings.addSharedUserLPw("android.uid.phone", RADIO_UID,

```

```

        ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.log", LOG_UID,
        ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);

        ...
        mInstaller = installer;
        //创建 Dex 优化工具类
        mPackageDexOptimizer = new PackageDexOptimizer(installer,
mInstallLock, context,
        "*dexopt*");
        mDexManager = new DexManager(this, mPackageDexOptimizer,
installer, mInstallLock);
        mMoveCallbacks = new MoveCallbacks(FgThread.get().getLooper());
        mOnPermissionChangeListeners = new OnPermissionChangeListeners(
        FgThread.get().getLooper());
        getDefaultDisplayMetrics(context, mMetrics);
        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "get system
config");
        //得到全局系统配置信息。
        SystemConfig systemConfig = SystemConfig.getInstance();
        //获取全局的 groupId
        mGlobalGids = systemConfig.getGlobalGids();
        //获取系统权限
        mSystemPermissions = systemConfig.getSystemPermissions();
        mAvailableFeatures = systemConfig.getAvailableFeatures();
        Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
        mProtectedPackages = new ProtectedPackages(mContext);
        //安装 APK 时需要的锁，保护所有对 installd 的访问。
        synchronized (mInstallLock) { //1
        //更新 APK 时需要的锁，保护内存中已经解析的包信息等内容
        synchronized (mPackages) { //2
                //创建后台线程 ServiceThread
                mHandlerThread = new ServiceThread(TAG,
                        Process.THREAD_PRIORITY_BACKGROUND, true
/*allowIo*/);
                mHandlerThread.start();
                //创建 PackageHandler 绑定到 ServiceThread 的消息队列
                mHandler = new
PackageHandler(mHandlerThread.getLooper()); //3
                mProcessLoggingHandler = new ProcessLoggingHandler();
                //将 PackageHandler 添加到 Watchdog 的检测集中
                Watchdog.getInstance().addThread(mHandler,
WATCHDOG_TIMEOUT); //4

```

```

        mDefaultPermissionPolicy = new
DefaultPermissionGrantPolicy(this);
        mInstantAppRegistry = new InstantAppRegistry(this);
        //在 Data 分区创建一些目录
        File dataDir = Environment.getDataDirectory();//5
        mAppInstallDir = new File(dataDir, "app");
        mAppLib32InstallDir = new File(dataDir, "app-lib");
        mAsecInternalPath = new File(dataDir, "app-asec").getPath();
        mDrmAppPrivateInstallDir = new File(dataDir, "app-private");
        //创建多用户管理服务
        sUserManager = new UserManagerService(context, this,
            new UserDataPreparer(mInstaller, mInstallLock,
mContext, mOnlyCore), mPackages);
        ...
        mFirstBoot
= !mSettings.readLPw(sUserManager.getUsers(false))//6
        ...    }

```

在开始阶段中创建了很多 PMS 中的关键对象并赋值给 PMS 中的成员变量，下面简单介绍这些成员变量。

- **mSettings**：用于保存所有包的动态设置。注释 1 处将系统进程的 `sharedUserId` 添加到 Settings 中，`sharedUserId` 用于进程间共享数据，比如两个 App 之间的数据是不共享的，如果它们有了共同的 `sharedUserId`，就可以运行在同一个进程中共享数据。
- **mInstaller**：`Installer` 继承自 `SystemService`，和 PMS、AMS 一样是系统的服务（虽然名称不像是服务），PMS 很多的操作都是由 `Installer` 来完成的，比如 APK 的安装和卸载。在 `Installer` 内部，通过 `IInstalld` 和 `installd` 进行 Binder 通信，由位于 `native` 层的 `installd` 来完成具体的操作。
- **systemConfig**：用于得到全局系统配置信息。比如系统的权限就可以通过 `SystemConfig` 来获取。
- **mPackageDexOptimizer**：`Dex` 优化的工具类。
- **mHandler**（`PackageHandler` 类型）：`PackageHandler` 继承自 `Handler`，在注释 3 处它绑定了后台线程 `ServiceThread` 的消息队列。PMS 通过 `PackageHandler` 驱动 APK 的复制和安装工作，具体的请看看 [Android 包管理机制（三）PMS 处理 APK 的安装](#) 这篇文章。

`PackageHandler` 处理的消息队列如果过于繁忙，有可能导致系统卡住，因此在注释 4 处将它添加到 `Watchdog` 的监测集中。

`Watchdog` 主要有两个用途，一个是定时检测系统关键服务（AMS 和 WMS 等）是否可能发生死锁，还有一个是定时检测线程的消息队列是否长时间处于工作状态（可能阻塞等待了很长时间）。如果出现上述问题，`Watchdog` 会将日志保存起来，必要时还会杀掉自己所在的进程，也就是 `SystemServer` 进程。

- **sUserManager**（`UserManagerService` 类型）：多用户管理服务。

除了创建这些关键对象，在开始阶段还有一些关键代码需要去讲解：

- 注释 1 处和注释 2 处加了两个锁，其中 `mInstallLock` 是安装 APK 时需要的锁，保护所有对 `installId` 的访问；`mPackages` 是更新 APK 时需要的锁，保护内存中已经解析的包信息等内容。
- 注释 5 处后的代码创建了一些 Data 分区中的子目录，比如 `/data/app`。
- 注释 6 处会解析 `packages.xml` 等文件的信息，保存到 `Settings` 的对应字段中。`packages.xml` 中记录系统中所有安装的应用信息，包括基本信息、签名和权限。如果 `packages.xml` 有安装的应用信息，那么注释 6 处 `Settings` 的 `readLPw` 方法会返回 `true`，`mFirstBoot` 的值为 `false`，说明 PMS 不是首次被启动。

2.2 扫描系统阶段

```
...public PackageManagerService(Context context, Installer installer,
    boolean factoryTest, boolean onlyCore) {...
    //打印扫描系统阶段日志

    EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SYSTEM_SCAN_START,
        startTime);

    ...
    //在/system中创建 framework 目录
    File frameworkDir = new File(Environment.getRootDirectory(),
"framework");

    ...
    //扫描/vendor/overlay 目录下的文件
    scanDirTracedLI(new File(VENDOR_OVERLAY_DIR), mDefParseFlags
        | PackageParser.PARSE_IS_SYSTEM
        | PackageParser.PARSE_IS_SYSTEM_DIR
        | PackageParser.PARSE_TRUSTED_OVERLAY, scanFlags |
SCAN_TRUSTED_OVERLAY, 0);
    mParallelPackageParserCallback.findStaticOverlayPackages();
    //扫描/system/framework 目录下的文件
    scanDirTracedLI(frameworkDir, mDefParseFlags
        | PackageParser.PARSE_IS_SYSTEM
        | PackageParser.PARSE_IS_SYSTEM_DIR
        | PackageParser.PARSE_IS_PRIVILEGED,
        scanFlags | SCAN_NO_DEX, 0);
    final File privilegedAppDir = new
File(Environment.getRootDirectory(), "priv-app");
    //扫描 /system/priv-app 目录下的文件
    scanDirTracedLI(privilegedAppDir, mDefParseFlags
        | PackageParser.PARSE_IS_SYSTEM
        | PackageParser.PARSE_IS_SYSTEM_DIR
        | PackageParser.PARSE_IS_PRIVILEGED, scanFlags, 0);
    final File systemAppDir = new
File(Environment.getRootDirectory(), "app");
```

```

//扫描/system/app 目录下的文件
scanDirTracedLI(systemAppDir, mDefParseFlags
    | PackageParser.PARSE_IS_SYSTEM
    | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);
File vendorAppDir = new File("/vendor/app");
try {
    vendorAppDir = vendorAppDir.getCanonicalFile();
} catch (IOException e) {
    // failed to look up canonical path, continue with original
one
}
//扫描 /vendor/app 目录下的文件
scanDirTracedLI(vendorAppDir, mDefParseFlags
    | PackageParser.PARSE_IS_SYSTEM
    | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

//扫描/oem/app 目录下的文件
final File oemAppDir = new File(Environment.getOemDirectory(),
"app");
scanDirTracedLI(oemAppDir, mDefParseFlags
    | PackageParser.PARSE_IS_SYSTEM
    | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

//这个列表代表有可能有升级包的系统 App
final List<String> possiblyDeletedUpdatedSystemApps = new
ArrayList<String>(); //1
if (!mOnlyCore) {
    Iterator<PackageSetting> psit =
mSettings.mPackages.values().iterator();
    while (psit.hasNext()) {
        PackageSetting ps = psit.next();
        if ((ps.pkgFlags & ApplicationInfo.FLAG_SYSTEM) == 0)
{
            continue;
        }
        //这里的 mPackages 的是 PMS 的成员变量，代表
scanDirTracedLI 方法扫描上面那些目录得到的
        final PackageParser.Package scannedPkg =
mPackages.get(ps.name);
        if (scannedPkg != null) {
            if (mSettings.isDisabledSystemPackageLPr(ps.name))
{ //2
                ...

```

```

//将这个系统 App 的 PackageSetting 从 PMS 的
mPackages 中移除
        removePackageLI(scannedPkg, true);
        //将升级包的路径添加到 mExpectingBetter 列表中
        mExpectingBetter.put(ps.name, ps.codePath);
    }
    continue;
}

if (!mSettings.isDisabledSystemPackageLPr(ps.name)) {
    ...
} else {
    final PackageSetting disabledPs =
mSettings.getDisabledSystemPkgLPr(ps.name);
    //这个系统App升级包信息在 mDisabledSysPackages 中,
    但是没有发现这个升级包存在
    if (disabledPs.codePath == null
|| !disabledPs.codePath.exists()) { //5

possiblyDeletedUpdatedSystemApps.add(ps.name); //
    }
    }
}
}
...
}

```

/system 可以称之为 **System** 分区，里面主要存储谷歌和其他厂商提供的 **Android** 系统相关文件和框架。**Android** 系统架构分为应用层、应用框架层、系统运行库层（**Native** 层）、硬件抽象层（**HAL** 层）和 **Linux** 内核层，除了 **Linux** 内核层在 **Boot** 分区，其他层的代码都在 **System** 分区。下面列出 **System** 分区的部分子目录。

目录	含义
app	存放系统App，包括了谷歌内置的App也有厂商或者运营商提供的App
framework	存放应用框架层的jar包
priv-app	存放特权App
lib	存放so文件
fonts	存放系统字体文件
media	存放系统的各种声音，比如铃声、提示音，以及系统启动播放的动画

上面的代码还涉及到/vendor 目录，它用来存储厂商对 Android 系统的定制部分。

系统扫描阶段的主要工作有以下 3 点：

1. 创建/system 的子目录，比如/system/framework、/system/priv-app 和/system/app 等等
2. 扫描系统文件，比如/vendor/overlay、/system/framework、/system/app 等等目录下的文件。
3. 对扫描到的系统文件做后续处理。

主要来说第 3 点，一次 OTA 升级对于一个系统 App 会有三种情况：

- 这个系统 APP 无更新。
- 这个系统 APP 有更新。
- 新的 OTA 版本中，这个系统 APP 已经被删除。

当系统 App 升级，PMS 会将该系统 App 的升级包设置数据（PackageSetting）存储到 Settings 的 mDisabledSysPackages 列表中（具体见 PMS 的 replaceSystemPackageLIF 方法），mDisabledSysPackages 的类型为 `ArrayMap<String, PackageSetting>`。mDisabledSysPackages 中的信息会被 PMS 保存到 packages.xml 中的 <updated-package> 标签下（具体见 Settings 的 writeDisabledSysPackageLPr 方法）。

注释 2 处说明这个系统 App 有升级包，那么就将该系统 App 的 PackageSetting 从 mDisabledSysPackages 列表中移除，并将系统 App 的升级包的路径添加到 mExpectingBetter 列表中，mExpectingBetter 的类型为 `ArrayMap<String, File>` 等待后续处理。

注释 5 处如果这个系统 App 的升级包信息存储在 mDisabledSysPackages 列表中，但是没有发现这个升级包存在，则将它加入到 possiblyDeletedUpdatedSystemApps 列表中，意为“系统 App 的升级包可能被删除”，之所以是“可能”，是因为系统还没有扫描 Data 分区，只能暂放到 possiblyDeletedUpdatedSystemApps 列表中，等到扫描完 Data 分区后再做处理。

2.3 扫描 Data 分区阶段

```
public PackageManagerService(Context context, Installer installer,
    boolean factoryTest, boolean onlyCore) {
    ...
    mSettings.pruneSharedUsersLPw();
    //如果不是只扫描系统的目录，那么就开始扫描 Data 分区。
    if (!mOnlyCore) {
        //打印扫描 Data 分区阶段日志

        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_DATA_SCAN_START,
            SystemClock.uptimeMillis());
        //扫描/data/app 目录下的文件
        scanDirTracedLI(mAppInstallDir, 0, scanFlags |
SCAN_REQUIRE_KNOWN, 0);
        //扫描/data/app-private 目录下的文件
        scanDirTracedLI(mDrmAppPrivateInstallDir, mDefParseFlags
            | PackageParser.PARSE_FORWARD_LOCK,
            scanFlags | SCAN_REQUIRE_KNOWN, 0);
        //扫描完 Data 分区后，处理 possiblyDeletedUpdatedSystemApps 列表
        for (String deletedAppName : possiblyDeletedUpdatedSystemApps) {
            PackageParser.Package deletedPkg =
mPackages.get(deletedAppName);
            // 从 mSettings.mDisabledSysPackages 变量中移除去此应用
            mSettings.removeDisabledSystemPackageLPw(deletedAppName);
            String msg;
            //1: 如果这个系统 App 的包信息不在 PMS 的变量 mPackages 中，说明
            是残留的 App 信息，后续会删除它的数据。
            if (deletedPkg == null) {
                msg = "Updated system package " + deletedAppName
                    + " no longer exists; it's data will be wiped";
                // Actual deletion of code and data will be handled by later
                // reconciliation step
            } else {
                //2: 如果这个系统 App 在 mPackages 中，说明是存在于 Data 分区，
                不属于系统 App，那么移除其系统权限。
                msg = "Updated system app + " + deletedAppName
                    + " no longer present; removing system privileges for
"
                    + deletedAppName;
                deletedPkg.applicationInfo.flags &=
~ApplicationInfo.FLAG_SYSTEM;
                PackageSetting deletedPs =
mSettings.mPackages.get(deletedAppName);
```

```

        deletedPs.pkgFlags &= ~ApplicationInfo.FLAG_SYSTEM;
    }
    logCriticalInfo(Log.WARN, msg);
}
//遍历 mExpectingBetter 列表
for (int i = 0; i < mExpectingBetter.size(); i++) {
    final String packageName = mExpectingBetter.keyAt(i);
    if (!mPackages.containsKey(packageName)) {
        //得到系统 App 的升级包路径
        final File scanFile = mExpectingBetter.valueAt(i);
        logCriticalInfo(Log.WARN, "Expected better " + packageName
            + " but never showed up; reverting to system");
        int reparseFlags = mDefParseFlags;
        //3: 根据系统 App 所在的目录设置扫描的解析参数
        if (FileUtils.contains(privilegedAppDir, scanFile)) {
            reparseFlags = PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR
                | PackageParser.PARSE_IS_PRIVILEGED;
        }
        ...
        //将 packageName 对应的包设置数据 (PackageSetting) 添加到
mSettings 的 mPackages 中
        mSettings.enableSystemPackageLPw(packageName);//4
        try {
            //扫描系统 App 的升级包
            scanPackageTracedLI(scanFile, reparseFlags, scanFlags,
0, null);//5
        } catch (PackageManagerException e) {
            Slog.e(TAG, "Failed to parse original system package:
"
                + e.getMessage());
        }
    }
}
//清除 mExpectingBetter 列表
mExpectingBetter.clear();...}

```

/data 可以称为 **Data** 分区，它用来存储所有用户的个人数据和配置文件。下面列出 **Data** 分区部分子目录：

目录	含义
app	存储用户自己安装的App
data	存储所有已安装的App数据的目录，每个App都有自己单独的子目录
app-private	App的私有存储空间
app-lib	存储所有App的jni库
system	存放系统配置文件
anr	用于存储ANR发生时系统生成的traces.txt文件

扫描 Data 分区阶段主要做了以下几件事：

1. 扫描/data/app 和/data/app-private 目录下的文件。
2. 遍历 possiblyDeletedUpdatedSystemApps 列表，注释 1 处如果这个系统 App 的包信息不在 PMS 的变量 mPackages 中，说明是残留的 App 信息，后续会删除它的数据。注释 2 处如果这个系统 App 的包信息在 mPackages 中，说明是存在于 Data 分区，不属于系统 App，那么移除其系统权限。
3. 遍历 mExpectingBetter 列表，注释 3 处根据系统 App 所在的目录设置扫描的解析参数，注释 4 处的方法内部会将 packageName 对应的包设置数据（PackageSetting）添加到 mSettings 的 mPackages 中。注释 5 处扫描系统 App 的升级包，最后清除 mExpectingBetter 列表。

2.4 扫描结束阶段

```
//打印扫描结束阶段日志
EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SCAN_END,
    SystemClock.uptimeMillis());
Slog.i(TAG, "Time to scan packages: "
    + ((SystemClock.uptimeMillis()-startTime)/1000f)
    + " seconds");
int updateFlags = UPDATE_PERMISSIONS_ALL;
// 如果当前平台 SDK 版本和上次启动时的 SDK 版本不同，重新更新
APK 的授权
if (ver.sdkVersion != mSdkVersion) {
    Slog.i(TAG, "Platform changed from " + ver.sdkVersion + "
to "
        + mSdkVersion + "; regranting permissions for
internal storage");
    updateFlags |= UPDATE_PERMISSIONS_REPLACE_PKG |
UPDATE_PERMISSIONS_REPLACE_ALL;
}
```

```

        updatePermissionsLPw(null, null,
StorageManager.UUID_PRIVATE_INTERNAL, updateFlags);
        ver.sdkVersion = mSdkVersion;
        //如果是第一次启动或者是 Android M 升级后的第一次启动，需要初
        始化所有用户定义的默认首选 App
        if (!onlyCore && (mPromoteSystemApps || mFirstBoot)) {
            for (UserInfo user : sUserManager.getUsers(true)) {
                mSettings.applyDefaultPreferredAppsLPw(this,
user.id);

                applyFactoryDefaultBrowserLPw(user.id);
                primeDomainVerificationsLPw(user.id);
            }
        }
        ...
        //OTA 后的第一次启动，会清除代码缓存目录。
        if (mIsUpgrade && !onlyCore) {
            Slog.i(TAG, "Build fingerprint changed; clearing code
        caches");
            for (int i = 0; i < mSettings.mPackages.size(); i++) {
                final PackageSetting ps =
mSettings.mPackages.valueAt(i);
                if
        (Objects.equals(StorageManager.UUID_PRIVATE_INTERNAL, ps.volumeUuid))
        {
                    clearAppDataLIF(ps.pkg, UserHandle.USER_ALL,
                        StorageManager.FLAG_STORAGE_DE |
StorageManager.FLAG_STORAGE_CE
                        |
        Installer.FLAG_CLEAR_CODE_CACHE_ONLY);
                }
            }
            ver.fingerprint = Build.FINGERPRINT;
        }
        ...
        // 把 Settings 的内容保存到 packages.xml 中
        mSettings.writeLPw();
        Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);

```

扫描结束结束阶段主要做了以下几件事：

1. 如果当前平台 SDK 版本和上次启动时的 SDK 版本不同，重新更新 APK 的授权。
2. 如果是第一次启动或者是 Android M 升级后的第一次启动，需要初始化所有用户定义的默认首选 App。
3. OTA 升级后的第一次启动，会清除代码缓存目录。

4. 把 Settings 的内容保存到 packages.xml 中，这样此后 PMS 再次创建时会读到此前保存的 Settings 的内容。

2.5 准备阶段

```
EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_READY,
                    SystemClock.uptimeMillis());

...
mInstallerService = new PackageInstallerService(context, this); //1
...
Runtime.getRuntime().gc(); //2
Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "loadFallbacks");
FallbackCategoryProvider.loadFallbacks();
Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
mInstaller.setWarnIfHeld(mPackages);
LocalServices.addService(PackageManagerInternal.class, new
PackageManagerInternalImpl()); //3
Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER); }
```

注释 1 处创建 PackageInstallerService，PackageInstallerService 是用于管理安装会话的服务，它会为每次安装过程分配一个 SessionId，在 [Android 包管理机制（二）PackageInstaller 安装 APK](#) 这篇文章中提到过 PackageInstallerService。

注释 2 处进行一次垃圾收集。注释 3 处将 PackageManagerInternalImpl（PackageManager 的本地服务）添加到 LocalServices 中，LocalServices 用于存储运行在当前的进程中的本地服务。

2. Activity 启动流程，App 启动流程

Activity 的启动模式

- 1.standard:默认标准模式，每启动一个都会创建一个实例，
- 2.singleTop: 栈顶复用，如果在栈顶就调用 onNewIntent 复用，从 onResume()开始
- 3.singleTask: 栈内复用，本栈内只要用该类型 Activity 就会将其顶部的 activity 出栈
- 4.singleInstance: 单例模式，除了 3 中特性，系统会单独给该 Activity 创建一个栈，

1.什么是 Zygote 进程

1.1 简单介绍

- Zygote 进程是所有的 android 进程的父进程, 包括 SystemServer 和各种应用进程都是通过 Zygote 进程 fork 出来的。Zygote (孵化) 进程相当于是 android 系统的根进程, 后面所有的进程都是通过这个进程 fork 出来的
- 虽然 Zygote 进程相当于 Android 系统的根进程, 但是事实上它也是由 Linux 系统的 init 进程启动的。

1.2 各个进程的先后顺序

- init 进程 --> Zygote 进程 --> SystemServer 进程 --> 各种应用进程

1.3 进程作用说明

- init 进程: linux 的根进程, android 系统是基于 linux 系统的, 因此可以算是整个 android 操作系统的第一个进程;
- Zygote 进程: android 系统的根进程, 主要作用: 可以作用 Zygote 进程 fork 出 SystemServer 进程和各种应用进程;
- SystemService 进程: 主要是在这个进程中启动系统的各项服务, 比如 ActivityManagerService, PackageManagerService, WindowManagerService 服务等等;
- 各种应用进程: 启动自己编写的客户端应用时, 一般都是重新启动一个应用进程, 有自己的虚拟机与运行环境;

2.Zygote 进程的启动流程

2.1 源码位置

- 位置: frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
- Zygote 进程 main 方法主要执行逻辑:
 - 初始化 DDMS;
 - 注册 Zygote 进程的 socket 通讯;
 - 初始化 Zygote 中的各种类, 资源文件, OpenGL, 类库, Text 资源等等;
 - 初始化完成之后 fork 出 SystemServer 进程;
 - fork 出 SystemServer 进程之后, 关闭 socket 连接;

2.2 ZygoteInit 类的 main 方法

- init 进程在启动 Zygote 进程时一般都会调用 ZygoteInit 类的 main 方法, 因此这里看一下该方法的具体实现(基于 android23 源码);
 - 调用 enableDdms(), 设置 DDMS 可用, 可以发现 DDMS 启动的时机还是比较早的, 在整个 Zygote 进程刚开始要启动时候就设置可用。
 - 之后初始化各种参数
 - 通过调用 registerZygoteSocket 方法, 注册为 Zygote 进程注册 Socket
 - 然后调用 preload 方法实现预加载各种资源
 - 然后通过调用 startSystemServer 开启 SystemServer 服务, 这个是重点

```
public static void main(String argv[]) {  
    try {  
  
        //设置 ddms 可以用  
  
        RuntimeInit.enableDdms();  
        SamplingProfilerIntegration.start();  
    }  
}
```

```

        boolean startSystemServer = false;
        String socketName = "zygote";
        String abiList = null;
        for (int i = 1; i < argv.length; i++) {
            if ("start-system-server".equals(argv[i])) {
                startSystemServer = true;
            } else if (argv[i].startsWith(ABI_LIST_ARG)) {
                abiList =
argv[i].substring(ABI_LIST_ARG.length());
            } else if (argv[i].startsWith(SOCKET_NAME_ARG))
{
                socketName =
argv[i].substring(SOCKET_NAME_ARG.length());
            } else {
                throw new RuntimeException("Unknown
command line argument: " + argv[i]);
            }
        }

        if (abiList == null) {
            throw new RuntimeException("No ABI list
supplied.");
        }

        registerZygoteSocket(socketName);

        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
            SystemClock.uptimeMillis());
        preload();

        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
            SystemClock.uptimeMillis());

        SamplingProfilerIntegration.writeZygoteSnapshot();

        gcAndFinalize();
        Trace.setTracingEnabled(false);

        if (startSystemServer) {
            startSystemServer(abiList, socketName);
        }

        Log.i(TAG, "Accepting command socket
connections");

```

```

        runSelectLoop(abiList);

        closeServerSocket();
    } catch (MethodAndArgsCaller caller) {
        caller.run();
    } catch (RuntimeException ex) {
        Log.e(TAG, "Zygote died with exception", ex);
        closeServerSocket();
        throw ex;
    }
}

```

2.3 registerZygoteSocket(socketName)分析

- 调用 registerZygoteSocket (String socketName) 为 Zygote 进程注册 socket
- ```

private static void registerZygoteSocket(String
socketName) {
 if (sServerSocket == null) {
 int fileDesc;
 final String fullSocketName =
 ANDROID_SOCKET_PREFIX + socketName;
 try {
 String env = System.getenv(fullSocketName);
 fileDesc = Integer.parseInt(env);
 } catch (RuntimeException ex) {
 throw new RuntimeException(fullSocketName + "
unset or invalid", ex);
 }

 try {
 FileDescriptor fd = new FileDescriptor();
 fd.setInt$(fileDesc);
 sServerSocket = new LocalServerSocket(fd);
 } catch (IOException ex) {
 throw new RuntimeException(
 "Error binding to local socket '" +
fileDesc + "'", ex);
 }
 }
}

```
- 

### 2.4 preLoad()方法分析

- 源码如下所示
- ```

static void preload() {

```

```

    Log.d(TAG, "begin preload");
    preloadClasses();
    preloadResources();
    preloadOpenGL();
    preloadSharedLibraries();
    preloadTextResources();
    // Ask the WebViewFactory to do any initialization that
    must run in the zygote process,
    // for memory sharing purposes.
    WebViewFactory.prepareWebViewInZygote();
    Log.d(TAG, "end preload");
}

```

- 大概操作是这样的：
 - preloadClasses()用于初始化 Zygote 中需要的 class 类；
 - preloadResources()用于初始化系统资源；
 - preloadOpenGL()用于初始化 OpenGL；
 - preloadSharedLibraries()用于初始化系统 libraries；
 - preloadTextResources()用于初始化文字资源；
 - prepareWebViewInZygote()用于初始化 webview；

2.5 startSystemServer()启动进程

- 这段逻辑的执行逻辑就是通过 Zygote fork 出 SystemServer 进程

```

private static boolean startSystemServer(String abiList,
String socketName)
    throws MethodAndArgsCaller, RuntimeException {
    long capabilities = posixCapabilitiesAsBits(
        OsConstants.CAP_BLOCK_SUSPEND,
        OsConstants.CAP_KILL,
        OsConstants.CAP_NET_ADMIN,
        OsConstants.CAP_NET_BIND_SERVICE,
        OsConstants.CAP_NET_BROADCAST,
        OsConstants.CAP_NET_RAW,
        OsConstants.CAP_SYS_MODULE,
        OsConstants.CAP_SYS_NICE,
        OsConstants.CAP_SYS_RESOURCE,
        OsConstants.CAP_SYS_TIME,
        OsConstants.CAP_SYS_TTY_CONFIG
    );
    /* Hardcoded command line to start the system server
    */
    String args[] = {
        "--setuid=1000",
        "--setgid=1000",

```

```

"--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1
009,1010,1018,1021,1032,3001,3002,3003,3006,3007",
    "--capabilities=" + capabilities + "," +
capabilities,
    "--nice-name=system_server",
    "--runtime-args",
    "com.android.server.SystemServer",
};
ZygoteConnection.Arguments parsedArgs = null;

int pid;

try {
    parsedArgs = new
ZygoteConnection.Arguments(args);

ZygoteConnection.applyDebuggerSystemProperty(parsedArg
s);

ZygoteConnection.applyInvokeWithSystemProperty(parsedA
rgs);

    /* Request to fork the system server process */
    pid = Zygote.forkSystemServer(
        parsedArgs.uid, parsedArgs.gid,
        parsedArgs.gids,
        parsedArgs.debugFlags,
        null,
        parsedArgs.permittedCapabilities,
        parsedArgs.effectiveCapabilities);
} catch (IllegalArgumentException ex) {
    throw new RuntimeException(ex);
}

/* For child process */
if (pid == 0) {
    if (hasSecondZygote(abiList)) {
        waitForSecondaryZygote(socketName);
    }

    handleSystemServerProcess(parsedArgs);
}

```

```

        return true;
    }
    •

```

3.SystemServer 进程启动流程

3.1 SystemServer 进程简介

- SystemServer 进程主要的作用是在这个进程中启动各种系统服务，比如 ActivityManagerService, PackageManagerService, WindowManagerService 服务，以及各种系统性的服务其实都是在 SystemServer 进程中启动的，而我们的应用需要使用各种系统服务的时候其实也是通过与 SystemServer 进程通讯获取各种服务对象的句柄的。

3.2 SystemServer 的 main 方法

- 如下所示，比较简单，只是 new 出一个 SystemServer 对象并执行其 run 方法，查看 SystemServer 类的定义我们知道其实 final 类型的，所以我们一般不能重写或者继承。

```

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

public SystemServer() {
    // Check for factory test mode.
    mFactoryTestMode = FactoryTest.getMode();
    // Remember if it's runtime restart (when sys.
    mRuntimeRestart = "1".equals(SystemProperties
}

```

main方法入口

更多可以查看

3.3 查看 run 方法

- 代码如下所示
 - 首先判断系统当前时间，若当前时间小于 1970 年 1 月 1 日，则一些初始化操作可能会处所，所以当系统的当前时间小于 1970 年 1 月 1 日的时候，设置系统当前时间为该时间点。
 - 然后是设置系统的语言环境等
 - 接着设置虚拟机运行内存，加载运行库，设置 SystemServer 的异步消息

```

private void run() {
    if (System.currentTimeMillis() <
        EARLIEST_SUPPORTED_TIME) {

```

```

        Slog.w(TAG, "System clock is before 1970; setting
to 1970.");

        SystemClock.setCurrentTimeMillis(EARLIEST_SUPPORTED_TIME);
    }

    if
(!SystemProperties.get("persist.sys.language").isEmpty
()) {
        final String languageTag =
Locale.getDefault().toLanguageTag();

        SystemProperties.set("persist.sys.locale",
languageTag);
        SystemProperties.set("persist.sys.language",
"");
        SystemProperties.set("persist.sys.country", "");
        SystemProperties.set("persist.sys.localevar",
"");
    }

    Slog.i(TAG, "Entered the Android system server!");

    EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_SYSTEM_
RUN, SystemClock.uptimeMillis());

    SystemProperties.set("persist.sys.dalvik.vm.lib.2",
VMRuntime.getRuntime().vmLibrary());

    if (SamplingProfilerIntegration.isEnabled()) {
        SamplingProfilerIntegration.start();
        mProfilerSnapshotTimer = new Timer();
        mProfilerSnapshotTimer.schedule(new TimerTask() {
            @Override
            public void run() {

SamplingProfilerIntegration.writeSnapshot("system_serv
er", null);
            }
        }, SNAPSHOT_INTERVAL, SNAPSHOT_INTERVAL);
    }

    // Mmmmmm... more memory!

```



```

    VMRuntime.getRuntime().clearGrowthLimit();

    // The system server has to run all of the time, so it
    needs to be
    // as efficient as possible with its memory usage.

VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);

    // Some devices rely on runtime fingerprint generation,
    so make sure
    // we've defined it before booting further.
    Build.ensureFingerprintProperty();

    // Within the system server, it is an error to access
    Environment paths without
    // explicitly specifying a user.
    Environment.setUserRequired(true);

    // Ensure binder calls into the system always run at
    foreground priority.
    BinderInternal.disableBackgroundScheduling(true);

    // Prepare the main looper thread (this thread).
    android.os.Process.setThreadPriority(

android.os.Process.THREAD_PRIORITY_FOREGROUND);
    android.os.Process.setCanSelfBackground(false);
    Looper.prepareMainLooper();

    // Initialize native services.
    System.loadLibrary("android_servers");

    // Check whether we failed to shut down last time we
    tried.
    // This call may not return.
    performPendingShutdown();

    // Initialize the system context.
    createSystemContext();

    // Create the system service manager.
    mSystemServiceManager = new
SystemServiceManager(mSystemContext);

```

```

        LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);

        // Start services.
        try {
            startBootstrapServices();
            startCoreServices();
            startOtherServices();
        } catch (Throwable ex) {
            Slog.e("System",
"*****");
            Slog.e("System", "***** Failure starting
system services", ex);
            throw ex;
        }

        // For debug builds, log event loop stalls to dropbox
for analysis.
        if (StrictMode.conditionallyEnableDebugLogging()) {
            Slog.i(TAG, "Enabled StrictMode for system server
main thread.");
        }

        // Loop forever.
        Looper.loop();
        throw new RuntimeException("Main thread loop
unexpectedly exited");
    }

```

• 然后下面的代码是:

```

// Initialize the system context.
createSystemContext();
// Create the system service manager.
mSystemServiceManager = new
SystemServiceManager(mSystemContext);
LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);
// Start services.
try {
    startBootstrapServices();
    startCoreServices();
    startOtherServices();
} catch (Throwable ex) {
    Slog.e("System",
"*****");
}

```

```

        Slog.e("System", "***** Failure starting
system services", ex);
        throw ex;
    }
    •

```

3.4 run 方法中 createSystemContext()解析

- 调用 createSystemContext()方法:
 - 可以看到在 SystemServer 进程中也存在着 Context 对象，并且是通过 ActivityThread.systemMain 方法创建 context 的,这一部分的逻辑以后会通过介绍 Activity 的启动流程来介绍，这里就不在扩展，只知道在 SystemServer 进程中也需

```

private void createSystemContext() {
    ActivityThread activityThread = ActivityThread.systemMain();
    mSystemContext = activityThread.getSystemContext();

    mSystemContext.setTheme(android.R.style.Theme_DeviceDe
fault_Light_DarkActionBar);
}

```

3.5 mSystemServiceManager 的创建

- 看 run 方法中，通过 SystemServiceManager 的构造方法创建了一个新的 SystemServiceManager 对象，我们知道 SystemServer 进程主要是用来构建系统各种 service 服务的，而 SystemServiceManager 就是这些服务的管理对象。
- 然后调用:
 - 将 SystemServiceManager 对象保存 SystemServer 进程中的一个数据结构中。

```

LocalServices.addService(SystemServiceManager.class,
mSystemServiceManager);

```

```

    • 最后开始执行:
    // Start services.try {
        startBootstrapServices();
        startCoreServices();
        startOtherServices();
    } catch (Throwable ex) {
        Slog.e("System",
"***** Failure starting
system services", ex);
        throw ex;
    }
    •
    •

```

- 里面主要涉及了三个方法:
 - startBootstrapServices() 主要用于启动系统 Boot 级服务

- `startCoreServices()` 主要用于启动系统核心的服务
- `startOtherServices()` 主要用于启动一些非紧要或者是非需要及时启动的服务

4.启动服务

4.1 启动哪些服务

- 在开始执行启动服务之前总是会先尝试通过 `socket` 方式连接 `Zygote` 进程，在成功连接之后才会开始启动其他服务。

```
// Start services.
try {
    traceBeginAndSlog( name: "StartServices");
    startBootstrapServices();
    startCoreServices();
    startOtherServices();
    SystemServerInitThreadPool.shutdown();
} catch (Throwable ex) {
    Slog.e("System", "*****");
    Slog.e("System", "***** Failure start");
    throw ex;
} finally {
    traceEnd();
}
```

这里面是启动服务，接下来

更多可

4.2 启动服务流程源码分析

- 首先看一下 `startBootstrapServices` 方法:

```
private void startBootstrapServices() {
    Installer installer =
mSystemServiceManager.startService(Installer.class);

    mActivityManagerService =
mSystemServiceManager.startService(
ActivityManagerService.Lifecycle.class).getService();

mActivityManagerService.setSystemServiceManager(mSystem
mServiceManager);
    mActivityManagerService.setInstaller(installer);
}
```

```

        mPowerManagerService =
mSystemServiceManager.startService(PowerManagerService.
class);

        mActivityManagerService.initPowerManagement();

        // Manages LEDs and display backlight so we need it to
bring up the display.

mSystemServiceManager.startService(LightsService.class)
;

        // Display manager is needed to provide display metrics
before package manager
        // starts up.
        mDisplayManagerService =
mSystemServiceManager.startService(DisplayManagerServi
ce.class);

        // We need the default display before we can initialize
the package manager.

mSystemServiceManager.startBootPhase(SystemService.PHA
SE_WAIT_FOR_DEFAULT_DISPLAY);

        // Only run "core" apps if we're encrypting the device.
        String cryptState =
SystemProperties.get("vold.decrypt");
        if (ENCRYPTING_STATE.equals(cryptState)) {
            Slog.w(TAG, "Detected encryption in progress - only
parsing core apps");
            mOnlyCore = true;
        } else if (ENCRYPTED_STATE.equals(cryptState)) {
            Slog.w(TAG, "Device encrypted - only parsing core
apps");
            mOnlyCore = true;
        }

        // Start the package manager.
        Slog.i(TAG, "Package Manager");
        mPackageManagerService =
PackageManagerService.main(mSystemContext, installer,
mFactoryTestMode !=
FactoryTest.FACTORY_TEST_OFF, mOnlyCore);

```

```

        mFirstBoot = mPackageManagerService.isFirstBoot();
        mPackageManager =
mSystemContext.getPackageManager();

        Slog.i(TAG, "User Service");
        ServiceManager.addService(Context.USER_SERVICE,
UserManagerService.getInstance());

        // Initialize attribute cache used to cache resources
from packages.
        AttributeCache.init(mSystemContext);

        // Set up the Application instance for the system
process and get started.
        mActivityManagerService.setSystemProcess();

        // The sensor service needs access to package manager
service, app ops
        // service, and permissions service, therefore we start
it after them.
        startSensorService();
    }

```

- 先执行:
- `Installer installer = mSystemServiceManager.startService(Installer.class);`
- `mSystemServiceManager` 是系统服务管理对象，在 `main` 方法中已经创建完成，这里我们看一下其 `startService` 方法的具体实现：
 - 可以看到通过反射器构造方法创建出服务类，然后添加到 `SystemServiceManager` 的服务列表数据中，最后调用了 `service.onStart()` 方法，因为传递的是 `Installer.class`

```

public <T extends SystemService> T startService(Class<T>
serviceClass) {
    final String name = serviceClass.getName();
    Slog.i(TAG, "Starting " + name);

    // Create the service.
    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);
        service = constructor.newInstance(mContext);
    } catch (InstantiationException ex) {
        throw new RuntimeException("Failed to create
service " + name
+ ": service could not be instantiated", ex);
    } catch (IllegalAccessException ex) {
        throw new RuntimeException("Failed to create
service " + name
+ ": service must have a public constructor
with a Context argument", ex);
    } catch (NoSuchMethodException ex) {
        throw new RuntimeException("Failed to create
service " + name
+ ": service must have a public constructor
with a Context argument", ex);
    } catch (InvocationTargetException ex) {
        throw new RuntimeException("Failed to create
service " + name
+ ": service constructor threw an exception",
ex);
    }

    // Register it.
    mServices.add(service);

    // Start it.
    try {
        service.onStart();
    } catch (RuntimeException ex) {
        throw new RuntimeException("Failed to start
service " + name
+ ": onStart threw an exception", ex);
    }
    return service;
}

```

- 看一下 `Installer` 的 `onStart` 方法:
 - 很简单就是执行了 `mInstaller` 的 `waitForConnection` 方法, 这里简单介绍一下 `Installer` 类, 该类是系统安装 apk 时的一个服务类, 继承 `SystemService` (系统

服务的一个抽象接口），需要在启动完成 **Installer** 服务之后才能启动其他的系统服务。

```
@Override public void onStart() {
    Slog.i(TAG, "Waiting for installd to be ready.");
    mInstaller.waitForConnection();
}
```

- 然后查看 `waitForConnection()` 方法：
 - 通过追踪代码可以发现，其在不断的通过 `ping` 命令连接 `Zygote` 进程（`SystemService` 和 `Zygote` 进程通过 `socket` 方式通讯，其他进程通过 `Binder` 方式通讯）

```
public void waitForConnection() {
    for (;;) {
        if (execute("ping") >= 0) {
            return;
        }
        Slog.w(TAG, "installd not ready");
        SystemClock.sleep(1000);
    }
}
```

- 继续看 `startBootstrapServices` 方法：
 - 这段代码主要是用于启动 `ActivityManagerService` 服务，并为其设置 `SystemServiceManager` 和 `Installer`。`ActivityManagerService` 是系统中一个非常重要的服务，`Activity`，`service`，`Broadcast`，`contentProvider` 都需要通过其余系统交互。

```
// Activity manager runs the show.mActivityManagerService
= mSystemServiceManager.startService(
```

```
ActivityManagerService.Lifecycle.class).getService();
mActivityManagerService.setSystemServiceManager(mSystem
mServiceManager);
mActivityManagerService.setInstaller(installer);
```

- 首先看一下 `Lifecycle` 类的定义：
 - 可以看到其实 `ActivityManagerService` 的一个静态内部类，在其构造方法中会创建一个 `ActivityManagerService`，通过刚刚对 `Installer` 服务的分析我们知道，`SystemServiceManager` 的 `startService` 方法会调用服务的 `onStart()` 方法，而在 `Lifecycle` 类的定义中我们看到其 `onStart()` 方法直接调用了 `mService.start()` 方法，`mService` 是 `Lifecycle` 类中对 `ActivityManagerService` 的引用

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

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



```

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

public ActivityManagerService getService() {
    return mService;
}
}

```

4.3 启动部分服务

- 启动 **PowerManagerService** 服务：
 - 启动方式跟上面的 **ActivityManagerService** 服务相似都会调用其构造方法和 **onStart** 方法，**PowerManagerService** 主要用于计算系统中和 **Power** 相关的计算，然后决策系统应该如何反应。同时协调 **Power** 如何与系统其它模块的交互，比如没有用户活动时，屏幕变暗等等。

```

mPowerManagerService =
mSystemServiceManager.startService(PowerManagerService.
class);

```

- 然后是启动 **LightsService** 服务
 - 主要是手机中关于闪光灯，LED 等相关的服务；也是会调用 **LightsService** 的构造方法和 **onStart** 方法；
- 然后是启动 **DisplayManagerService** 服务
 - 主要是手机显示方面的服务
- 然后是启动 **PackageManagerService**，该服务也是 **android** 系统中一个比较重要的服务
 - 包括多 **apk** 文件的安装，解析，删除，卸载等等操作。
 - 可以看到 **PackageManagerService** 服务的启动方式与其他服务的启动方式有一些区别，直接调用了 **PackageManagerService** 的静态 **main** 方法

```

Slog.i(TAG, "Package Manager");
mPackageManagerService =
PackageManagerService.main(mSystemContext, installer,
    mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF,
mOnlyCore);
mFirstBoot = mPackageManagerService.isFirstBoot();
mPackageManager = mSystemContext.getPackageManager();

```

- 看一下其 **main** 方法的具体实现：

- 可以看到也是直接使用 `new` 的方式创建了一个 `PackageManagerService` 对象，并在其构造方法中初始化相关变量，最后调用了 `ServiceManager.addService` 方法，主要是通过 `Binder` 机制与 `JNI` 层交互

```
public static PackageManagerService main(Context context,
    Installer installer,
    boolean factoryTest, boolean onlyCore) {
    PackageManagerService m = new
    PackageManagerService(context, installer,
        factoryTest, onlyCore);
    ServiceManager.addService("package", m);
    return m;
}
```

- 然后查看 `startCoreServices` 方法：

- 可以看到这里启动了 `BatteryService`（电池相关服务），`UsageStatsService`，`WebViewUpdateService` 服务等。

```
private void startCoreServices() {
    // Tracks the battery level. Requires LightService.

    mSystemServiceManager.startService(BatteryService.class);

    // Tracks application usage stats.

    mSystemServiceManager.startService(UsageStatsService.class);
    mActivityManagerService.setUsageStatsManager(
        LocalServices.getService(UsageStatsManagerInternal.class));
    // Update after UsageStatsService is available, needed
    before performBootDexOpt.

    mPackageManagerService.getUsageStatsIfNoPackageUsageInfo();

    // Tracks whether the updatable WebView is in a ready
    state and watches for update installs.

    mSystemServiceManager.startService(WebViewUpdateService.class);
}
```

总结：

- SystemServer 进程是 android 中一个很重要的进程由 Zygote 进程启动；
- SystemServer 进程主要用于启动系统中的服务；
- SystemServer 进程启动服务的启动函数为 main 函数；
- SystemServer 在执行过程中首先会初始化一些系统变量，加载类库，创建 Context 对象，创建 SystemServiceManager 对象等之后才开始启动系统服务；
- SystemServer 进程将系统服务分为三类：boot 服务，core 服务和 other 服务，并逐步启动
- SertemServer 进程在尝试启动服务之前会首先尝试与 Zygote 建立 socket 通讯，只有通讯成功之后才会开始尝试启动服务；
- 创建的系统服务过程中主要通过 SystemServiceManager 对象来管理，通过调用服务对象的方法构造和 onStart 方法初始化服务的相关变量；
- 服务对象都有自己的异步消息对象，并运行在单独的线程中；

3. Binder 机制（IPC、AIDL 的使用）

1、什么是 AIDL 以及如何使用（★★★★）

①aidl 是 Android interface definition Language 的英文缩写，意思 Android 接口定义语言。

②使用 aidl 可以帮助我们发布以及调用远程服务，实现跨进程通信。

③将服务的 aidl 放到对应的 src 目录，工程的 gen 目录会生成相应的接口类

我们通过 bindService(Intent, ServiceConnect, int) 方法绑定远程服务，在 bindService 中有一个 ServiceConnec 接口，我们需要覆写该类的 onServiceConnected(ComponentName,IBinder)方法，这个方法的第二个参数 IBinder 对象其实就是已经在 aidl 中定义的接口，因此我们可以将 IBinder 对象强制转换为 aidl 中的接口类。

我们通过 IBinder 获取到的对象（也就是 aidl 文件生成的接口）其实是系统产生的代理对象，该代理对象既可以跟我们的进程通信，又可以跟远程进程通信，作为一个中间的角色实现了进程间通信。

2、AIDL 的全称是什么?如何工作?能处理哪些类型的数据? (★★★★)

AIDL 全称 Android Interface Definition Language (Android 接口描述语言) 是一种接口描述语言; 编译器可以通过 `aidl` 文件生成一段代码, 通过预先定义的接口达到两个进程内部通信进程跨界对象访问的目的。需要完成2件事情:

1. 引入 AIDL 的相关类; 2. 调用 `aidl` 产生的 `class`。理论上, 参数可以传递基本数据类型和 `String`, 还有就是 `Bundle` 的派生类, 不过在 Eclipse 中, 目前的 ADT 不支持 `Bundle` 做为参数。

3.android 的 IPC 通信方式 , 线程 (进程间) 通信机制有哪些

- 1.ipc 通信方式: `binder`、`contentprovider`、`socket`
- 2.操作系统进程通讯方式: 共享内存、`socket`、管道

4. 为什么使用 Parcelable, 好处是什么?

实现机制

```
* Interface for classes whose instances can be written to
* and restored from a {@link Parcel}. Classes implementing the Parcelable
* interface must also have a static field called CREATOR, which
* is an object implementing the {@link Parcelable.Creator Parcelable.Creator}
* interface.
```

如上，摘自 **Parcelable** 注释：如果想要写入 **Parcel** 或者从中恢复，则必须 implements **Parcelable** 并且必须有一个 **static field** 而且名字必须是 **CREATOR**....

好吧，感觉好复杂。有如下疑问：

- 1、**Parcelable** 是干啥的？为什么需要它？
- 2、**Parcel** 又是干啥的？
- 3、如果是写入 **Parcel** 中、从 **Parcelable** 中恢复，那要 **Parcelable** 岂不是“多此一举”？

下面逐个回答上述问题：

- 1、**Parcelable** 是干啥的？从源码看：

```
public interface Parcelable {  
    ...  
    public void writeToParcel(Parcel dest, int flags);  
    ...  
}
```

简单来说，**Parcelable** 是一个 **interface**，有一个方法 **writeToParcel(Parcel dest, int flags)**，该方法接收两个参数，其中第一个参数类型是 **Parcel**。看起来 **Parcelable** 好像是对 **Parcelable** 的一种包装，从实际开发中，会在方法 **writeToParcel** 中调用 **Parcel** 的某些方法，完成将对象写入 **Parcelable** 的过程。

为什么往 **Parcel** 写入或恢复数据，需要继承 **Parcelable** 呢？我们看 **Intent** 的 **putExtra** 系列方法：

- putExtra(String, boolean) : Intent
- putExtra(String, byte) : Intent
- putExtra(String, char) : Intent
- putExtra(String, short) : Intent
- putExtra(String, int) : Intent
- putExtra(String, long) : Intent
- putExtra(String, float) : Intent
- putExtra(String, double) : Intent
- putExtra(String, String) : Intent
- putExtra(String, CharSequence) : Intent
- putExtra(String, Parcelable) : Intent
- putExtra(String, Parcelable[]) : Intent
- putParcelableArrayListExtra(String, ArrayList<? extends Parcelable>) : Intent
- putIntegerArrayListExtra(String, ArrayList<Integer>) : Intent
- putStringArrayListExtra(String, ArrayList<String>) : Intent
- putCharSequenceArrayListExtra(String, ArrayList<CharSequence>) : Intent
- putExtra(String, Serializable) : Intent
- putExtra(String, boolean[]) : Intent
- putExtra(String, byte[]) : Intent
- putExtra(String, short[]) : Intent
- putExtra(String, char[]) : Intent
- putExtra(String, int[]) : Intent
- putExtra(String, long[]) : Intent
- putExtra(String, float[]) : Intent
- putExtra(String, double[]) : Intent
- putExtra(String, String[]) : Intent
- putExtra(String, CharSequence[]) : Intent
- putExtra(String, Bundle) : Intent
- putExtras(Intent) : Intent
- putExtras(Bundle) : Intent

往 Intent 中添加数据，无法就是添加以上各种类型，简单的数据类型有对应的方法，如 putExtra(String, String)，复杂一点的有 putExtra(String, Bundle)、putExtra(String, Serializable)、putExtra(String, Bundle)、putExtra(String, Parcelable)、putExtra(String, Parcelable[])。现在想想，如果往 Intent 里添加一个我们自定义的类型对象 person（Person 类的实例），咋整？总不能用 putExtra(String, person)吧？为啥，类型不符合啊！如果 Person 没有基础任何类，那它不可以用 putExtra 的任何一个方法，比较不存在 putExtra(String, Object) 这样一个方法。

那为了可以用 putExtra 方法，Person 就需要继承一个可以用 putExtra 方法的类（接口），可以继承 Parcelable——继承其他类（接口）也没有问题。

现在捋一捋：为了使用 putExtra 方法，需要继承 Parcelable 类——事实上，还有更深的含义，且看后面。

2、Parcel 又是干啥的？前面说过，继承了 **Parcelable** 接口的类，如果不是抽象类，必须实现方法 **writeToParcel**，该方法有一个 **Parcel** 类型的参数，**Parcel** 源码：



```
public final class Parcel {  
    ...  
    public static Parcel obtain() {  
        final Parcel[] pool = sOwnedPool;  
        synchronized (pool) {  
            Parcel p;  
            for (int i=0; i<POOL_SIZE; i++) {  
                p = pool[i];  
                if (p != null) {  
                    pool[i] = null;  
                    if (DEBUG_RECYCLE) {  
                        p.mStack = new RuntimeException();  
                    }  
                    return p;  
                }  
            }  
            return new Parcel(0);  
        }  
        ...public final native void writeInt(int val);  
        public final native void writeLong(long val);  
        ...  
    }
```



Parcel 是一个 **final** 不可继承类，其代码很多，其中重要的一些部分是它有许多 **native** 的函数，在 **writeToParcel** 中调用的这些方法都直接或间接的调用 **native** 函数完成。

现在有一个问题，在 **public void writeToParcel(Parcel dest, int flags)**中调用 **dest** 的函数，这个 **dest** 是传入进来的，是形参，那实参在哪里？没有看到有什么地方生成了一个 **Parcel** 的实例，然后调用 **writeToParcel** 啊？？那它又不可能凭空出来。现在回到 **Intent** 这边来，看看它的内部做了什么事：

```
Intent i = new Intent();
```

```

    Person person = new Person();


    i.putExtra("person", person);

    i.setClass(this, SecondeActivity.class);

    startActivity(i);

```

为了简单说明情况，我写了如上的代码，就不解释了。看看 `putExtra` 做了什么事情，看源码：



```

    public Intent putExtra(String name, Parcelable value) {

        if (mExtras == null) {

            mExtras = new Bundle();


        }

        mExtras.putParcelable(name, value);

        return this;

    }

```



这里调用的 `putExtra` 的第二个参数是 `Parcelable` 类型的，也印证了前面必须要类型符合（这里多说一句，面向对象的六大原则里有一个非常非常重要的“里氏替换”原则，子类出现的地方可以用父类代替，这样所有继承了 `Parcelable` 的类都可以传入这个 `putExtra` 中）。原来这里用到了 `Bundle` 类，看源码：

```

    public void putParcelable(String key, Parcelable value) {

        unparcel();


        mMap.put(key, value);

        mFdsKnown = false;

    }

```

`mMap` 是一个 `Map`。看到这里，原来我们传入的 `person` 被写入了 `Map` 里面了。这个 `Bundle` 也是继承自 `Parcelable` 的。其他 `putExtra` 系列的方法都是调用这个 `mMap` 的 `put`。为什么要用 `Bundle` 的类里的 `Map`？统一管理啊！所有传到 `Intent` 的 `extra` 我都不管，交给 `Bundle` 类来管理了，这样 `Intent` 类就不会太笨重（面向对象六大原则之迪米特原则——我不管你怎么整，整对了就行）。看 `Bundle` 源码：



```

public final class Bundle implements Parcelable, Cloneable {

    private static final String LOG_TAG = "Bundle";

    public static final Bundle EMPTY;

    //Bundle 类一加载就生成了一个 Bundle 实例

```



```

static {
    EMPTY = new Bundle();

    EMPTY.mMap = Collections.unmodifiableMap(new HashMap<String, Object>());
} /* package */ Map<String, Object> mMap = null;

/* package */ Parcel mParcelledData = null;

```



看到这里，还是没有发现 **Parcel** 实例在什么地方生成，继续往下看，看 **startActivity** 这个方法，找到最后会发现最终启动 **Activity** 的是一个 **ActivityManagerNative** 类，查看对应的方法：



```

public int startActivity(IApplicationThread caller, Intent intent,
    String resolvedType, IBinder resultTo, String resultWho, int
requestCode,
    int startFlags, String profileFile,
    ParcelFileDescriptor profileFd, Bundle options) throws
RemoteException {
    Parcel data = Parcel.obtain(); //在这里生成了 Parcel 实例
    Parcel reply = Parcel.obtain(); //又生成了一个 Parcel 实例
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller != null ? caller.asBinder() : null);
    intent.writeToParcel(data, 0);
    data.writeString(resolvedType);
    data.writeStrongBinder(resultTo);
    data.writeString(resultWho);
    data.writeInt(requestCode);
    data.writeInt(startFlags);
    data.writeString(profileFile);
    if (profileFd != null) {
        data.writeInt(1);
        profileFd.writeToParcel(data,
Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
    } else {
        data.writeInt(0);
    }
    if (options != null) {

```

```

        data.writeInt(1);

        options.writeToParcel(data, 0);

    } else {

        data.writeInt(0);

    }

    mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0);

    reply.readException();

    int result = reply.readInt();

    reply.recycle();

    data.recycle();

    return result;

}

```



千呼万唤的 **Parcel** 对象终于出现了，这里生成了俩 **Parcel** 对象：**data** 和 **reply**，主要的是 **data** 这个实例。**obtain** 是一个 **static** 方法，用于从 **Parcel** 池（**pool**）中找出一个可用的 **Parcel**，如果都不可用，则生成一个新的。每一个 **Parcel**(Java)都与一个 C++的 **Parcel** 对应。



```

    public static Parcel obtain() {

        final Parcel[] pool = sOwnedPool;

        synchronized (pool) {

            Parcel p;

            for (int i=0; i<POOL_SIZE; i++) {

                p = pool[i];

                if (p != null) {

                    pool[i] = null;

                    if (DEBUG_RECYCLE) {

                        p.mStack = new RuntimeException();

                    }

                    return p;

                }

            }

        }

        return new Parcel(0);
    }

```

```
}
```



在 `intent.writeToParcel(data, 0)` 里，查看源码：



```
public void writeToParcel(Parcel out, int flags) {  
    out.writeString(mAction);  
    Uri.writeToParcel(out, mData);  
    out.writeString(mType);  
    out.writeInt(mFlags);  
    out.writeString(mPackage);  
    ComponentName.writeToParcel(mComponent, out);  
  
    if (mSourceBounds != null) {  
        out.writeInt(1);  
        mSourceBounds.writeToParcel(out, flags);  
    } else {  
        out.writeInt(0);  
    }  
  
    if (mCategories != null) {  
        out.writeInt(mCategories.size());  
        for (String category : mCategories) {  
            out.writeString(category);  
        }  
    } else {  
        out.writeInt(0);  
    }  
  
    if (mSelector != null) {  
        out.writeInt(1);  
        mSelector.writeToParcel(out, flags);  
    } else {  
        out.writeInt(0);  
    }  
}
```

```

        if (mClipData != null) {
            out.writeInt(1);
            mClipData.writeToParcel(out, flags);
        } else {
            out.writeInt(0);
        }

        out.writeBundle(mExtras); //终于把我们自定义的 person 实例送走了
    }

```



看到这里 Parcel 实例终于生成了，但是我们重写的从 Parcelable 接口而来的 writeToParcel 这个方法在什么地方被调用呢？从上面的 Intent 中的 out.writeBundle(mExtras)-->writeBundle(Bundle val)-->writeToParcel(Parcel parcel, int flags)-->writeMapInternal(Map<String,Object> val)-->writeValue(Object v)-->writeParcelable(Parcelable p, int parcelableFlags)（除了 out.writeBundle(mExtras)这个方法，其他的方法都是在 Bundle 和 Parcel 里面调来调去的，真心累！）：



```

    public final void writeParcelable(Parcelable p, int parcelableFlags) {
        if (p == null) {
            writeString(null);
            return;
        }

        String name = p.getClass().getName();
        writeString(name);

        p.writeToParcel(this, parcelableFlags); //调用自己实现的方法
    }

```



OK，终于出来了。。。到这里，写入的过程已经出来了。

那如何恢复呢？这里用到的是我们自己写的 createFromParcel 这个方法，从一个 Intent 中恢复 person：

```

    Intent i= getIntent();

    Person p = i.getParcelableExtra("person");

```

调啊调，调到这个：



```
public final <T extends Parcelable> T readParcelable(ClassLoader loader) {
    String name = readString();

    if (name == null) {
        return null;
    }

    Parcelable.Creator<T> creator;

    synchronized (mCreators) {
        HashMap<String, Parcelable.Creator> map = mCreators.get(loader);

        if (map == null) {
            map = new HashMap<String, Parcelable.Creator>();
            mCreators.put(loader, map);
        }

        creator = map.get(name);

        if (creator == null) {
            try {
                Class c = loader == null ?
                    Class.forName(name) : Class.forName(name, true, loader);
                Field f = c.getField("CREATOR");
                creator = (Parcelable.Creator) f.get(null);
            }

            catch (IllegalAccessException e) {
                Log.e(TAG, "Class not found when unmarshalling: "
                    + name + ", e: " + e);

                throw new BadParcelableException(
                    "IllegalAccessException when unmarshalling: " + name);
            }

            catch (ClassNotFoundException e) {
                Log.e(TAG, "Class not found when unmarshalling: "
                    + name + ", e: " + e);

                throw new BadParcelableException(
                    "ClassNotFoundException when unmarshalling: " + name);
            }

            catch (ClassCastException e) {
```

```

        throw new BadParcelableException("Parcelable protocol requires
a "
        + "Parcelable.Creator object called "
        + " CREATOR on class " + name);
    }
    catch (NoSuchFieldException e) {
        throw new BadParcelableException("Parcelable protocol requires
a "
        + "Parcelable.Creator object called "
        + " CREATOR on class " + name);
    }
    if (creator == null) {
        throw new BadParcelableException("Parcelable protocol requires
a "
        + "Parcelable.Creator object called "
        + " CREATOR on class " + name);
    }

    map.put(name, creator);
}
}

if (creator instanceof Parcelable.ClassLoaderCreator<?>) {
    return
    ((Parcelable.ClassLoaderCreator<T>) creator).createFromParcel(this, loader);
}
return creator.createFromParcel(this); //调用我们自定义的那个方法
}

```



最后终于从我们自定义的方法中恢复那个保存的实例。

5. Android 图像显示相关流程，Vsync 信号等

Android Vsync 原理浅析

Preface

Android 中，Client 测量和计算布局，SurfaceFlinger（server）用来渲染绘制界面，client 和 server 的是通过匿名共享内存（SharedClient）通信。

每个应用和 SurfaceFlinger 之间都会创建一个 SharedClient，一个 SharedClient 最多可以创建 31 个 SharedBufferStack，每个 surface 对应一个 SharedBufferStack，也就是一个 Window。也就意味着，每个应用最多可以创建 31 个窗口。

Android 4.1 之后，AndroidOS 团队对 Android Display 进行了不断地进化和改变。引入了三个核心元素：Vsync，Triple Buffer，Choreographer。

首先来理解一下，图形界面的绘制，大概是有 CPU 准备数据，然后通过驱动层把数据交给 GPU 来进行绘制。图形 API 不允许 CPU 和 GPU 直接通信，所以就有了图形驱动（Graphics Driver）来进行联系。Graphics Driver 维护了一个序列（Display List），CPU 不断把需要显示的数据放进去，GPU 不断取出来进行显示。

其中 Choreographer 起调度的作用。统一绘制图像到 Vsync 的某个时间点。

Choreographer 在收到 Vsync 信号时，调用用户设置的回调函数。函数的先后顺序如下：

CALLBACK_INPUT：与输入事件有关
CALLBACK_ANIMATION：与动画有关
CALLBACK_TRAVERSAL：与 UI 绘制有关

Vsync 是什么呢？首先来说一下什么是 FPS，FPS 就是 Frame Per Second（每秒的帧数）的缩写，我们知道，FPS \geq 60 时，我们就不会觉得动画卡顿。当 FPS=60 时是个什么概念呢？

$1000/60 \approx 16.6$ ，也就是说在大概 16ms 中，我们要进行一次屏幕的刷新绘制。Vsync 是垂直同步的缩写。这里我们可以简单的理解成，这就是一个时间中断。例如，每 16ms 会有一个 Vsync 信号，那么系统在每次拿到 Vsync 信号时刷新屏幕，我们就不会觉得卡顿了。

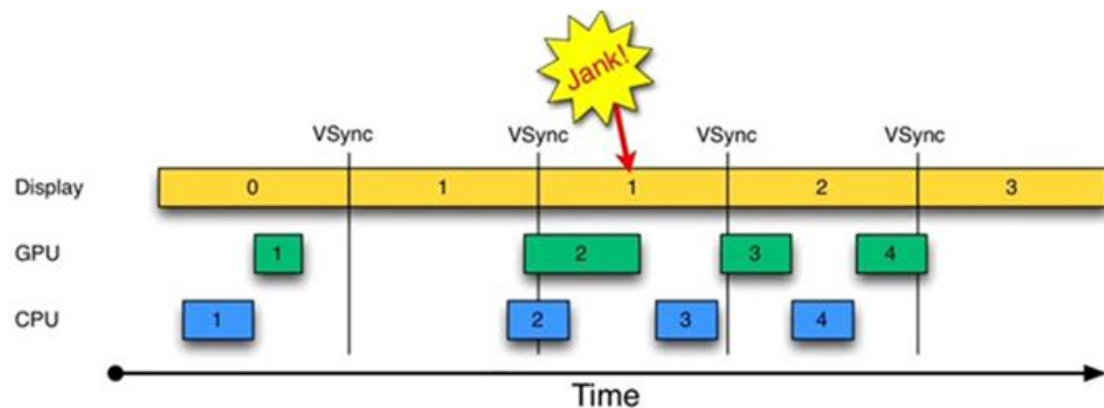
但实现起来还是有点困难的。

多重缓冲是什么技术呢？我们先来说双重缓冲。在 Linux 上，通常使用 FrameBuffer 来做显示输出。双重缓冲会创建一个 FrontBuffer 和一个 BackBuffer，顾名思义，FrontBuffer 是当前显示的页面，BackBuffer 是下一个要显示的画面。然后滚动电梯式显示数据。为什么呢？这样好在哪里呢？首先

他并不是不卡了，他还是会卡。但是如果是单重缓冲，页面可能会有这种情况：A 面数据需要显示，然后是 B 面数据显示，B 面数据显示需要耗费一定时间，但是这个时间里，C 面数据也请求了展示，我们可能会看到，在展示 C 面数据的时候，还有 B 面数据的残影...

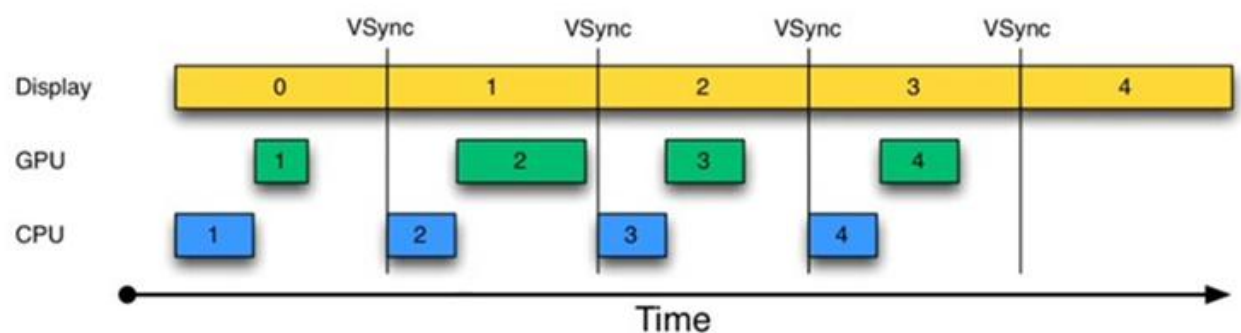
下面分情况来具体说明一下（Vsync 每 16 秒一次）。

1.没有使用 Vsync 的情况



可以看出，在第一个 16ms 之内，一切正常。然而在第二个 16ms 之内，几乎是在时间段最后 CPU 才计算出了数据，交给了 Graphics Driver,导致 GPU 也是在第二段的末尾时间才进行了绘制，整个动作延后到了第三段内。从而影响了下一个画面的绘制。这时会出现 Jank（闪烁，可以理解为卡顿或者停顿）。那么在第二个 16ms 前半段的时间 CPU 和 GPU 干什么了？哦，他们可能忙别的事情了。这就是卡顿出现的原因和情况。CPU 和 GPU 很随意，爱什么时候刷新什么时候刷新，很随意。

2.有 Vsync 的情况

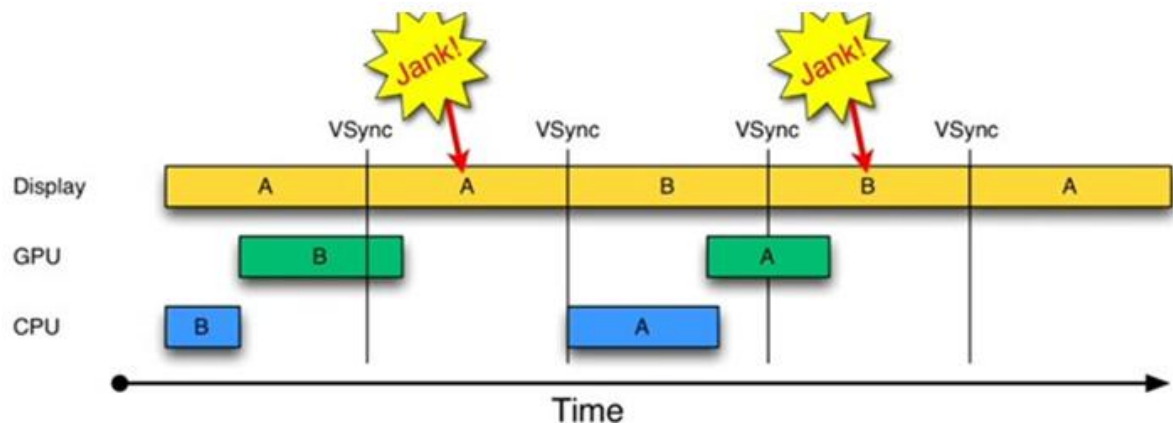


如果，按照之前的前提来说，Vsync 每 16ms 一次，那么在每次发出 Vsync 命令时，CPU 都会进行刷新的操作。也就是在每个 16ms 的第一时间，CPU 就会想赢 Vsync 的命令，来进行数据刷新的动作。CPU 和 GPU 的刷新时间，和 Display 的 FPS 是一致的。因为只有到发出 Vsync 命令的时候，

CPU 和 GPU 才会进行刷新或显示的动作。图中是正常情况。那么不正常情况是怎么个情况？我们先来说一下双重缓冲，然后再说。

3.双重缓冲

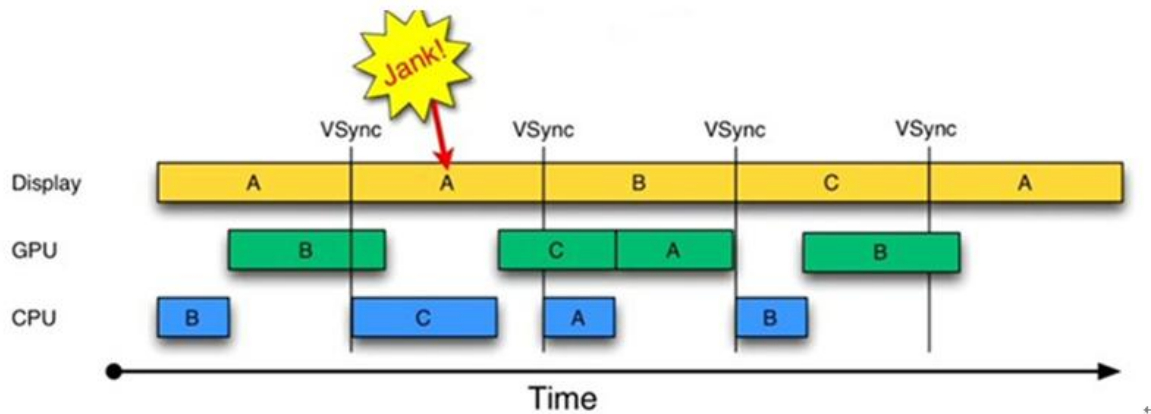
逻辑就是和之前一样。多重缓冲页面在 Back Buffer，然后根据需求来显示不同数据。但是会有什么问题呢（这就是 2 中提到的问题）？



首先我们看 Display 行，A 页面需要了两个时间单位，为什么？因为 B Buffer 在处理的时候太耗时了。然后导致了，在第一个 Vsync 发出的时候，还在 GPU 还在绘制 B Buffer。那么，刚好，第一个 Vsync 发出之后很短的时间，A 页面展示完了，B Buffer 的也在一开始的时候就不进行计算了。那么接下来的时间呢？屏幕还是展示着 B Buffer，这时候就会造成 Jank 现象。他不会动，因为他在等下一个 Vsync 过来的时候，才会显示下一个数据。

那么，如图，在 A Buffer 过来的时候，展示 B 页面的数据。这个时候！重复了上一个情况，也是太耗时了，然后又覆盖了下一个 Vsync 发出的时间，再次造成卡顿！依次类推，会造成多次卡顿。这个时候就有了三重缓冲的概念。

4.三重缓冲



首先看图。我们看到，B Buffer 依旧很耗时，同样覆盖了第一个 Vsync 发出的时间点。但是，在第一个 Vsync 发出的时候，C Buffer 站了出来，说，我来展示这个页面，你去缓冲 A 后面需要缓冲的页面吧！然后会发生什么？然后就是出现了一个 Jank...，但是这个 Jank 只在这一个时间单位出现，是可以忽略不计的。因为之后的逻辑都是顺畅的了。依次类推，除了 A 和 B 两个图层在交替显示，还有个“第三者”在不断帮他们两个可能需要展示的数据进行缓冲。但是注意了：

只有在需要时，才会进行三重缓冲。正常情况下，只使用二级缓冲！

另外，缓冲区不是越多越好。上图，C 页面在第四个时间段才展示出来，就是因为中间多了一个 Buffer（C Buffer）来进行缓冲。

但是，虽然谷歌给了你这么牛逼的前提逻辑，实际开发中你写的 APP 还是会卡，为什么呢？原因大概有两点：

1. 界面太复杂。
2. 主线程(UI 线程)太忙。他可能还在处理用户交互或者其他事情。