第10章 Framework内核解析面试题汇总

摘要：本章内容，享学课堂在Framework定制安卓系统中有系统化-全面完整的直播讲解，详情加微信：xxgfwx03

10.1 Android中多进程通信的方式有哪些?

详细讲解

享学课堂移动开发课程：Framework专题 Binder分析部分

这道题想考察什么?

对计算机中进程的了解，为了完成跨进程通信需要解决的问题以及现有的跨进程通信方式大致实现原理与区别

考察的知识点

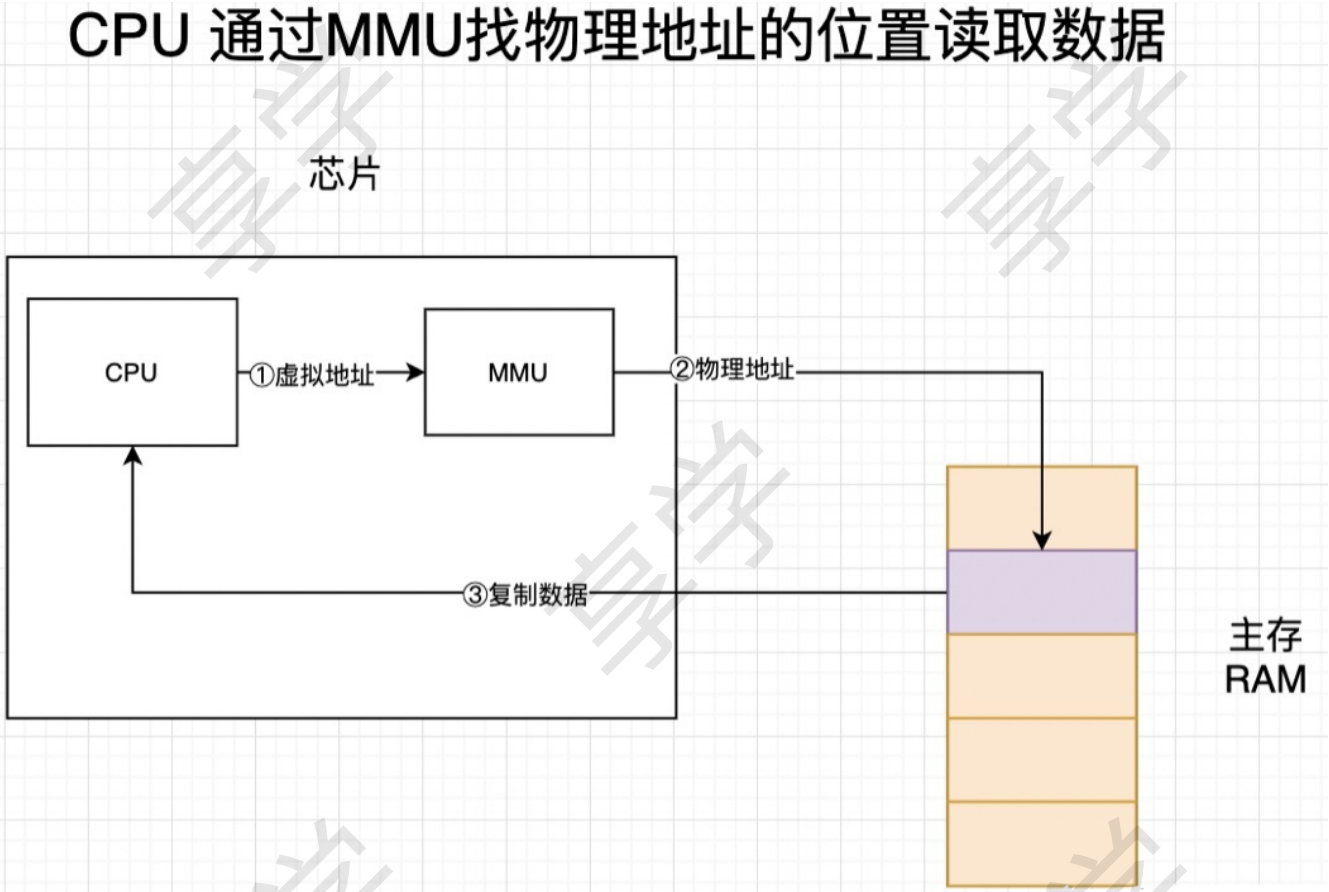
操作系统内存、进程与通信

考生应该如何回答

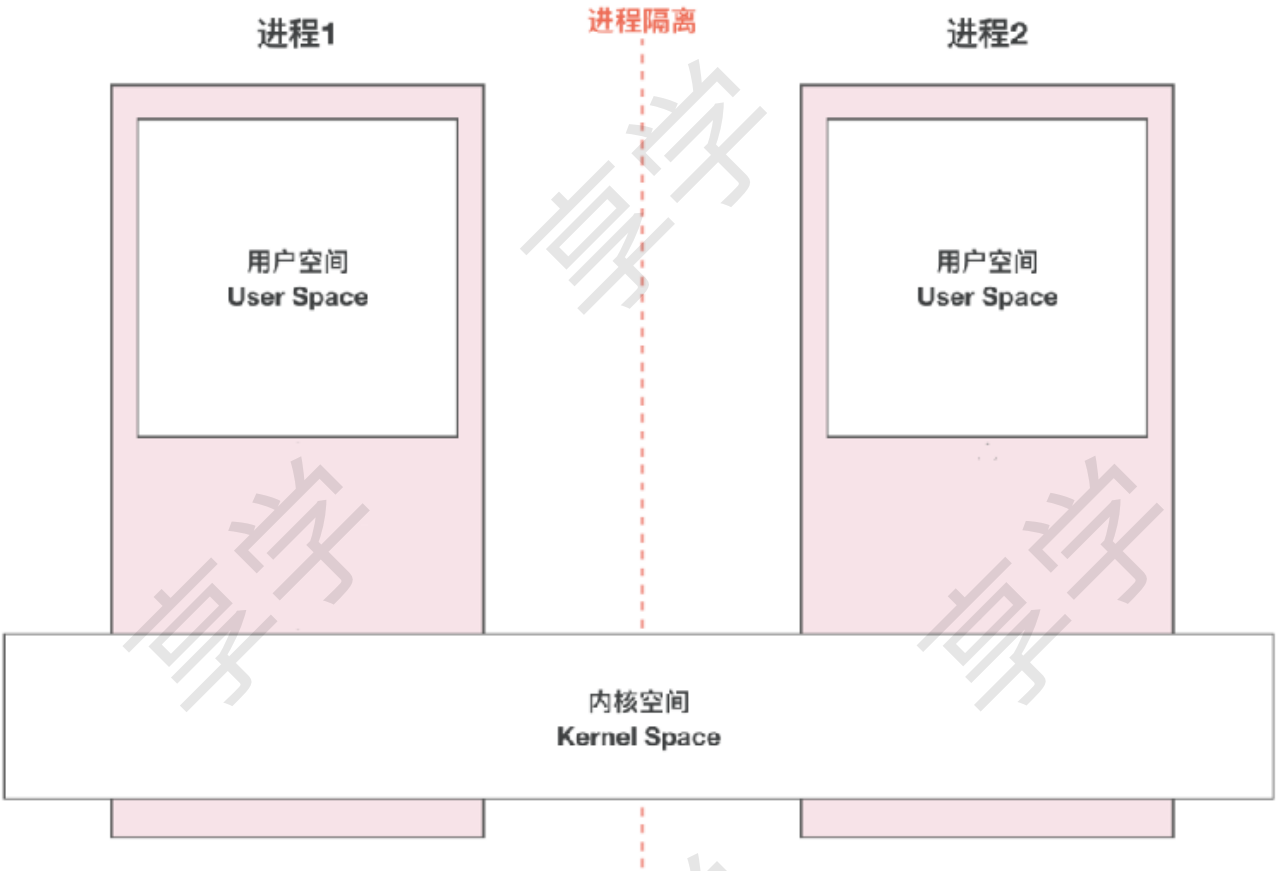
总的来说，进程间通信方案有很多他们分别是：管道，信号，信号量，内存共享，socket，binder，消息队列，但是使用最多的还是binder，尤其是用户空间的跨进程通信，基本大多采用的是binder。

进程隔离

操作系统有虚拟内存与物理内存的概念。物理内存指通过物理内存条而获得的内存空间，而虚拟内存则是计算机系统内存管理的一种技术，虚拟内存并非真正的内存，而是通过虚拟映射的手段让每个应用进程认为它拥有连续的可用的内存。在使用了虚拟存储器的情况下， 通过MMU（负责处理CPU的内存管理的计算单元）完成虚拟地址到物理地址的转换。



程序使用的虚拟内存被操作系统划分成两块：用户空间和内核空间。用户空间是用户程序代码运行的地方，内核空间是内核代码运行的地方，内核空间由所有进程共享。为了安全，内核空间与用户空间是隔离的，这样即使用户的程序崩溃了，内核也不受影响。同样为了安全，不同进程的各自的用户空间也是隔离的，这样就避免了进程间相互操作数据的现象发生，从而引起各自的安全问题。不同进程基于各自的虚拟地址不同，从逻辑上来实现彼此间的隔离。

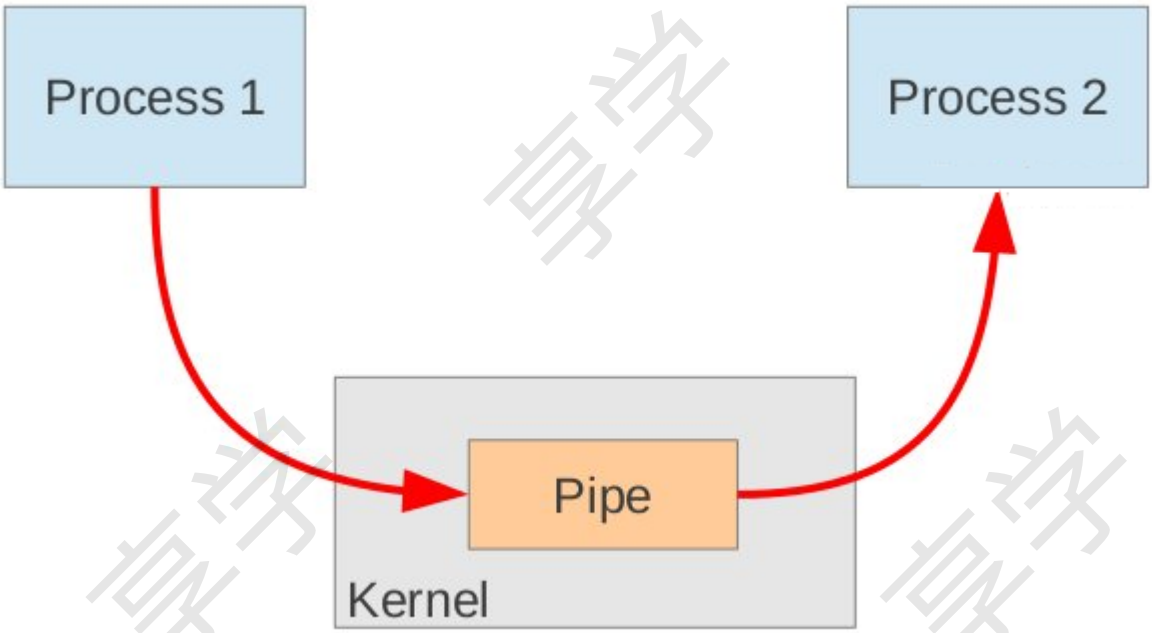


IPC通信

为了能使不同的进程互相访问资源并进行协调工作，需要在不同进程之间完成通信。而通过进程隔离可知不同进程之间无法直接完成通信的工作。此时就需要特殊的方式来实现：IPC（Inter-Process Communication ）即进程间通信。不同进程存在进程隔离，但是内核空间被所有进程共享，因此绝大多数的IPC机制就利用这个特点来实现通信的需求。因为Android是在Linux内核基础之上运行，因此Linux中存在的IPC机制在Android中基本都能使用。

管道

管道是UNIX中最古老的进程间通信形式，它实际上是由内核管理的一个固定大小的缓冲区。管道的一端连接一个进程的输出，这个进程会向管道中放入信息。管道的另一端连接一个进程的输入，这个进程取出被放入管道的信息。



可以通过pipe创建一个管道:

```
//匿名管道（PIPE）
#include <unistd.h>
int pipe (int fd[2]); //创建pipe
ssize_t write(int fd, const void *buf, size_t count);   //写数据
ssize_t read(int fd, void *buf, size_t count);     //读数据
```

pipe创建的是匿名管道，它存在以下限制：

1、大小限制（一般为4k）

2、半双工（同一个时刻只数据只能向一个方向流动，需要双方通信时，需要建立两个管道）

3、只支持父子和兄弟进程之间的通信

另外还有FIFO实名管道，支持双向的数据通信，建立命名管道时给它指定一个名字，任何进程都可以通过该名字打开管道的另一端，但是要同时和多个进程通信它就力不从心了。需要了解更多关于管道机制的内容可以在腾讯课堂搜索享学课堂。

信号

信号主要用于用于通知接收进程某个事件的发生。信号的原理是在软件层次上对中断机制的一种模拟，一个进程收到一个信号与处理器收到一个中断请求是一样的。

信号是一种异步通信机制，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。也就是说信号接收函数不需要一直阻塞等待信号的到达。进程可以通过 `sigaction` 注册接收的信号，就会执行响应函数，如果没有地方注册这个信号，该信号就会被忽略。

```
int sigaction(int signum,const struct sigaction *act ,struct sigaction *oldact);
//根据参数signum指定的信号编号来设置该信号的处理函数。
//参数signum可以指定SIGKILL和SIGSTOP以外的所有信号。
```

在Android中，如果程序出现ANR问题会发出：**SIGNALQUIT** 信号，应用程序可注册此信号的响应实现监听ANR，爱奇艺xCrash，友盟+ U-APM、腾讯Matrix都实现了该方式。可以在腾讯课堂搜索享学课堂了解更多ANR监控相关内容。

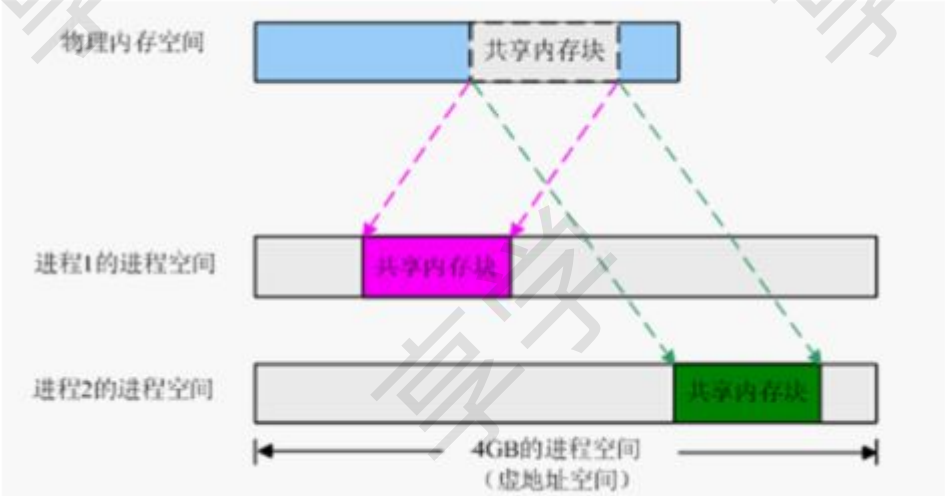
信号量

信号量实际上可以看成是一个计数器，用来控制多个进程对共享资源的访问。它不以传送数据为主要目的，主要作为进程间以及同一进程内不同线程之间的同步手段。

信号量会有初值（>0），每当有进程申请使用信号量，通过一个P操作来对信号量进行-1操作，当计数器减到0的时候就说明没有资源了，其他进程要想访问就必须等待，当该进程执行完这段工作（我们称之为临界区）之后，就会执行V操作来对信号量进行+1操作。

共享内存

通过进程隔离可知，不同进程基于各自的虚拟地址不同，从逻辑上来实现彼此间的隔离。 而共享内存则是让不同进程可以将同一段物理内存连接到他们自己的地址空间中，完成连接的进程都可以访问这块共享内存中的数据。



由于多个进程共享同一块内存区域，所以通常需要用其他的机制来同步对共享内存的访问，如信号量 。而在Android中提供了独特的匿名共享内存Ashmem（Anonymous Shared Memory）。Android的匿名共享内存基于 Linux的共享内存，都是在临时文件系统上创建虚拟文件，再映射到不同的进程。 它可以让多个进程操作同一块内存区域，并且除了物理内存限制，没有其他大小限制。相对于 Linux 的共享内存，Ashmem 对内存的管理更加精细化，并且添加了互斥锁。

在开发中，可以借助Java中的 MemoryFile 使用匿名共享内存，它封装了 native 代码。 Java 层使用匿名共享内存的步骤一般为：

1. 通过 MemoryFile 开辟内存空间，获得ParcelFileDescriptor；

```
MemoryFile memoryFile = new MemoryFile("test", 1024);
Method method = MemoryFile.class.getDeclaredMethod("getFileDescriptor");
FileDescriptor des = (FileDescriptor) method.invoke(memoryFile);
ParcelFileDescriptor pfd = ParcelFileDescriptor.dup(des);
```

2. 将 ParcelFileDescriptor 传递给其他进程；
3. A进程往共享内存写入数据；

```
memoryFile.getOutputStream().write(new byte[]{1, 2, 3, 4, 5});
```

4. B进程从共享内存读取数据。

```
ParcelFileDescriptor parcelFileDescriptor;
FileDescriptor descriptor = parcelFileDescriptor.getFileDescriptor();
FileInputStream fileInputStream = new FileInputStream(descriptor);
fileInputStream.read(content);
```

> 在第二步中一般的利用Binder机制进行FD的传输，传输完成后，就可以直接借助FD完成跨进程数据通信，而且没有内存大小的限制。因此当需要跨进程进行大数据的传递时，可以借助匿名共享内存完成！

在Android中视图数据与SurfaceFlinger的通信、腾讯MMKV、Facebook Fresco等等技术都有利用到匿名共享内存。

消息队列

消息队列是一个消息的链表,存放在内核中并由消息队列标识符标识。它克服了Linux早期IPC机制的很多缺点，比如消息队列具有异步能力，又克服了具有同样能力的信号承载信息量少的问题；具有数据传输能力，又克服了管道只能承载无格式字节流以及缓冲区大小受限的问题。

但是缺点是比信号和管道都要更加重量，在内核中会使用更多内存，并且消息队列能传输的数据也有限制，一般上限为两页 16kb。

```
int msgget(key_t, key, int msgflg); //创建和访问消息队列
int msgsend(int msgid, const void *msg_ptr, size_t msg_sz, int msgflg);  //发送消息
int msgrcv(int msgid, void *msg_ptr, size_t msg_st, long int msgtype, int msgflg); //获取消息
```

受限于性能，数据量等问题的限制，Android系统没有直接使用Linux消息队列来进行IPC的场景，但是有大量的场景都利用了消息队列的特性来设计通信方案，比如进行线程间通信的Handler，就是一个消息队列。

socket

socket 原本是为网络通讯设计的，但后来在 socket 的框架上发展出一种 IPC 机制，就是 UNIX domain socket。虽然网络 socket 也可用于同一台主机的进程间通讯(通过 loopback 地址 127.0.0.1)，但是 UNIX domain socket 用于 IPC 更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。 在Android系统中，Zygote进程就是通过LocalSocket（UNIX domain socket）接收启动应用进程的通知。

```
socket(AF_INET, SOCK_STREAM, 0); // 对应java  Socket
socket(AF_UNIX, SOCK_STREAM, 0); // 对应java  LocalSocket
```

Binder

Android Binder源于Palm的OpenBinder，在Android中Binder更多用在system_server进程与上层App层的IPC交互。 在Android中Intent、ContentProvider、Messager、Broadcast、AIDL等等都是基于Binder机制完成的跨进程通信。

总结

Android是在Linux内核基础之上运行，因此Linux中存在的IPC机制在Android中基本都能使用，如：

1. **管道：**在创建时分配一个page大小的内存，缓存区大小比较有限；
2. **信号**: 不适用于信息交换，更适用于进程中断控制，比如非法内存访问，杀死某个进程等；
3. **信号量**：常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
4. **共享内存**：无须复制，共享缓冲区直接付附加到进程虚拟地址空间，速度快；但进程间的同步问题操作系统无法实现，必须各进程利用同步工具解决；
5. **消息队列**：信息复制两次，额外的CPU消耗；不合适频繁或信息量大的通信；
6. **套接字**：作为更通用的接口，传输效率低；

为什么Android选择Binder作为应用程序中主要的IPC机制?

Binder基于C/S架构，进行跨进程通信数据拷贝只需要一次，而管道、消息队列、Socket都需要2次，共享内存方式一次内存拷贝都不需要；从性能角度看，Binder性能虽然比管道等方式好，但是不如共享内存。

但是传统Linux IPC的接收方无法获得对方进程可靠的UID/PID，只能由使用者在传递的数据包里填入UID/PID，伪造身份非常简单；而Binder不同，可靠的身份标记只有由IPC机制本身在内核中添加，binder就是这么做的，不由用户应用程序控制，直接在内核向数据中添加了进程身份标记。

因此综合考虑，Binder更加适合system_server进程与上层App层的IPC交互。

10.2 描述下Binder机制原理？（东方头条）

详细讲解

享学课堂移动开发课程：Framework专题 Binder分析部分

详细视频讲解：https://wx59a7e2633f445ef7.wx.finezb.com/share/recording/de17eb75126929454782d38164558533cb8980a2dfbeb3bca83b6ea6f005ca53

这道题想考察什么?

在Android中广泛运用的Binder机制是如何实现跨进程通信的

考察的知识点

Binder原理、内存映射

考生应该如何回答

Binder概述

Binder是Android提供的一套进程间相互通信框架，它是一种效率更高、更安全的基于C/S架构的IPC通信机制，其本质也是调用系统底层的内存共享实现。它基于开源的 OpenBinder 实现，从字面上来解释 Binder 有胶水、粘合剂的意思，顾名思义就是粘和不同的进程，使之实现通信。

Binder实现机制

进程隔离

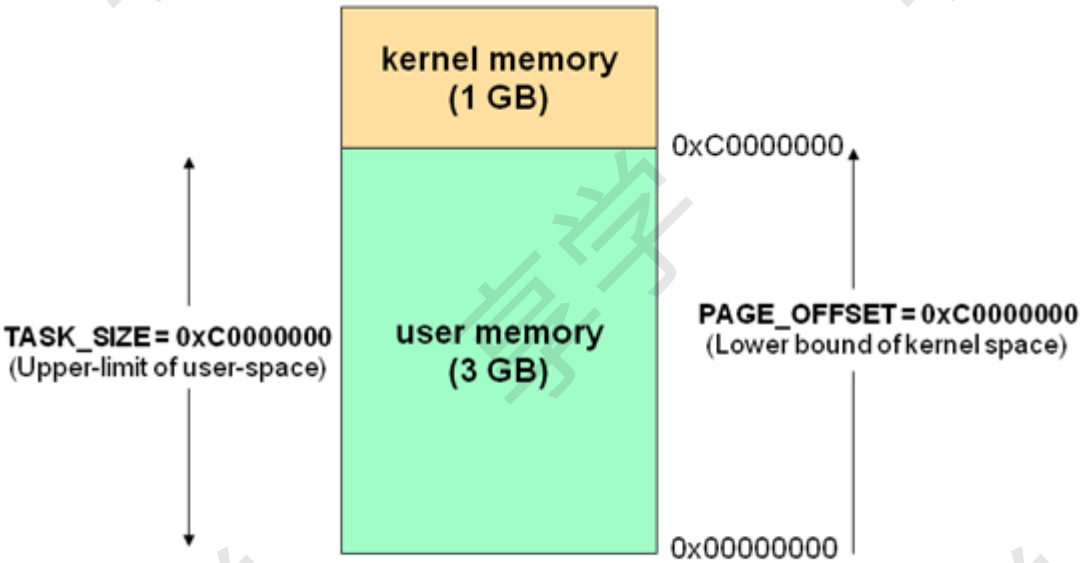
进程隔离不懂的同学可以先阅读8.1章节的进程隔离介绍。

通过进程隔离可知，不同进程无法直接通信，但是内核空间是被不同进程共享。那么是不是可以将数据从数据发送方进程用户空间交给内核，再由内核传递至数据接收方用户空间从而完成数据的交互，实现通信呢？ 没错，Linux下传统的IPC机制正是借助了这点从而实现了跨进程通信。由于进程用户空间与内核空间同样存在隔离，无法完成数据在用户空间与内核空间**直接传递**，所以现在解决**跨进程通信**就变成了解决**用户**空间与**内核**空间数据**传递**的问题。

进程空间划分：用户空间(User Space)/内核空间(Kernel Space)

现在操作系统都是采用的虚拟存储器，对于 32 位系统而言，它的寻址空间（虚拟存储空间）就是 2 的 32 次方，也就是 4GB。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也可以访问底层硬件设备的权限。为了保护用户进程不能直接操作内核，保证内核的安全，操作系统从逻辑上将虚拟空间划分为用户空间（User Space）和内核空间（Kernel Space）。针对 Linux 操作系统而言，将最高的 1GB 字节供内核使用，称为内核空间；较低的 3GB 字节供各进程使用，称为用户空间。

> 简单的说就是，内核空间（Kernel）是系统内核运行的空间，用户空间（User Space）是用户程序运行的空间。为了保证安全性，它们之间是隔离的。



系统调用：用户态与内核态

虽然从逻辑上进行了用户空间和内核空间的划分，但不可避免的用户空间需要访问内核资源，比如文件操作、访问网络等等。为了突破隔离限制，就需要借助**系统调用**来实现。系统调用是用户空间访问内核空间的唯一方式，保证了所有的资源访问都是在内核的控制下进行的，避免了用户程序对系统资源的越权访问，提升了系统安全性和稳定性。

Linux 使用两级保护机制：0 级供系统内核使用，3 级供用户程序使用。

当一个任务（进程）执行系统调用而陷入内核代码中执行时，称进程处于**内核运行态（内核态）**。此时处理器处于特权级最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。

当进程在执行用户自己的代码的时候，我们称其处于**用户运行态（用户态）**。此时处理器在特权级最低的（3级）用户代码中运行。

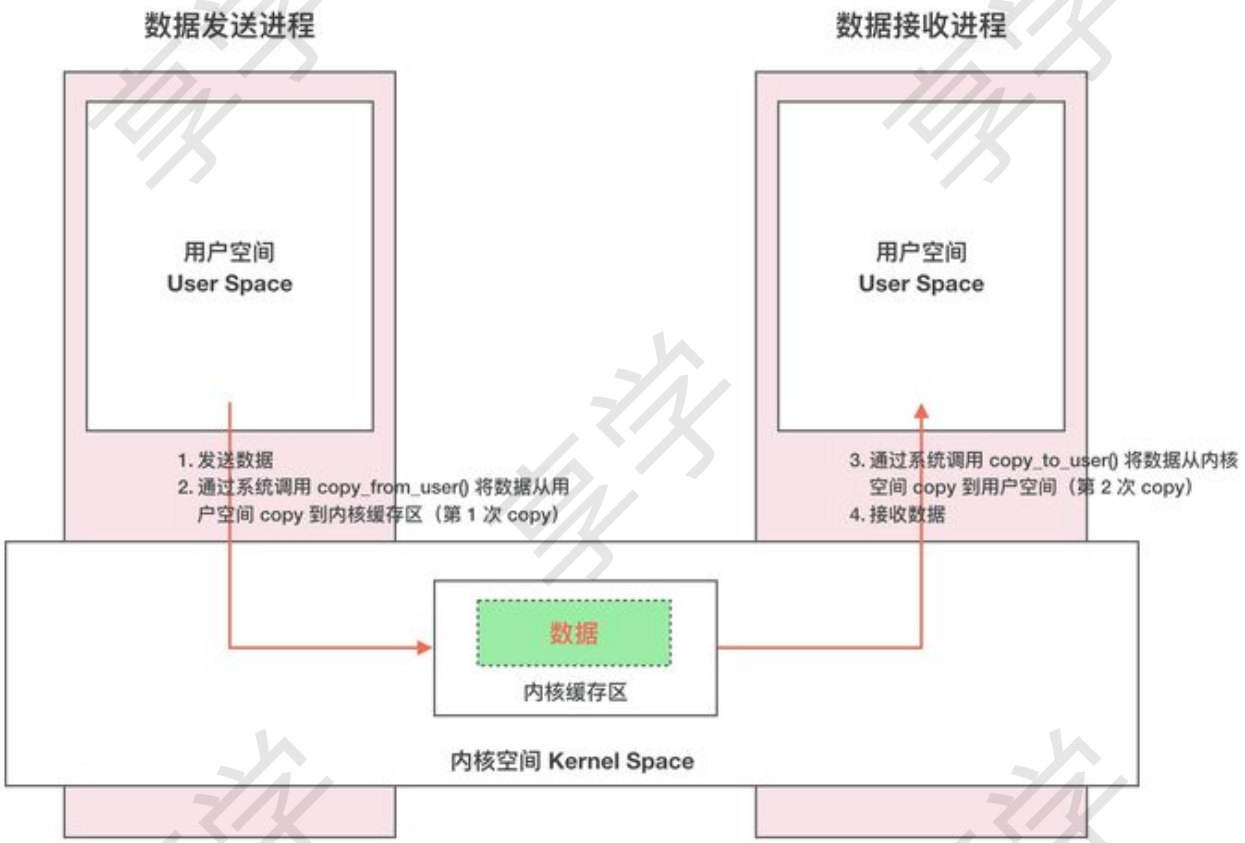
系统调用主要通过如下两个函数来实现：

```
copy_from_user() //将数据从用户空间拷贝到内核空间
copy_to_user() //将数据从内核空间拷贝到用户空间
```

Linux 下的传统 IPC 通信原理

理解了上面的几个概念，我们再来看看传统的 IPC 方式中，进程之间是如何实现通信的。

通常的做法是消息发送方将要发送的数据存放在内存缓存区中，通过系统调用进入内核态。然后内核程序在内核空间分配内存，开辟一块内核缓存区，调用 copy*from*user() 函数将数据从用户空间的内存缓存区拷贝到内核空间的内核缓存区中。同样的，接收方进程在接收数据时在自己的用户空间开辟一块内存缓存区，然后内核程序调用 copy*to*user() 函数将数据从内核缓存区拷贝到接收进程的内存缓存区。这样数据发送方进程和数据接收方进程就完成了一次数据传输，我们称完成了一次进程间通信。如下图：



这种传统的 IPC 通信方式有两个问题：

1. 性能低下，一次数据传递需要经历：内存缓存区 --> 内核缓存区 --> 内存缓存区，需要 2 次数据拷贝；
2. 接收数据的缓存区由数据接收进程提供，但是接收进程并不知道需要多大的空间来存放将要传递过来的数据，因此只能开辟尽可能大的内存空间或者先调用 API 接收消息头来获取消息体的大小，这两种做法不是浪费空间就是浪费时间。

Binder 跨进程通信原理

理解了 Linux IPC 相关概念和通信原理，接下来我们正式介绍下 Binder IPC 的原理。

动态内核可加载模块 && 内存映射

正如前面所说，跨进程通信是需要内核空间做支持的。传统的 IPC 机制如管道、Socket 都是内核的一部分，因此通过内核支持来实现进程间通信自然是没问题的。但是 Binder 并不是 Linux 系统内核的一部分，那怎么办呢？这就得益于 Linux 的**动态内核可加载模块**（Loadable Kernel Module，LKM）的机制；模块是具有独立功能的程序，它可以被单独编译，但是不能独立运行。它在运行时被链接到内核作为内核的一部分运行。这样，Android 系统就可以通过动态添加一个内核模块运行在内核空间，用户进程之间通过这个内核模块作为桥梁来实现通信。

> 在 Android 系统中，这个运行在内核空间，负责各个用户进程通过 Binder 实现通信的内核模块就叫 **Binder 驱动**（Binder Dirver）。

那么在 Android 系统中用户进程之间是如何通过这个内核模块（Binder 驱动）来实现通信的呢？难道是和前面说的传统 IPC 机制一样，先将数据从发送方进程拷贝到内核缓存区，然后再将数据从内核缓存区拷贝到接收方进程，通过两次拷贝来实现吗？显然不是，否则也不会有开篇所说的 Binder 在性能方面的优势了。

这就不得不通道 Linux 下的另一个概念：**内存映射**。

Binder IPC 机制中涉及到的内存映射通过 mmap() 来实现，mmap() 是操作系统中一种内存映射的方法。内存映射简单的讲就是将用户空间的一块内存区域映射到内核空间。映射关系建立后，用户对这块内存区域的修改可以直接反应到内核空间；反之内核空间对这段区域的修改也能直接反应到用户空间。

内存映射能减少数据拷贝次数，实现用户空间和内核空间的高效互动。两个空间各自的修改能直接反映在映射的内存区域，从而被对方空间及时感知。也正因为如此，内存映射能够提供对进程间通信的支持。

Binder IPC 实现原理

Binder IPC 正是基于内存映射（mmap）来实现的，但是 mmap() 通常是用在有物理介质的文件系统上的。

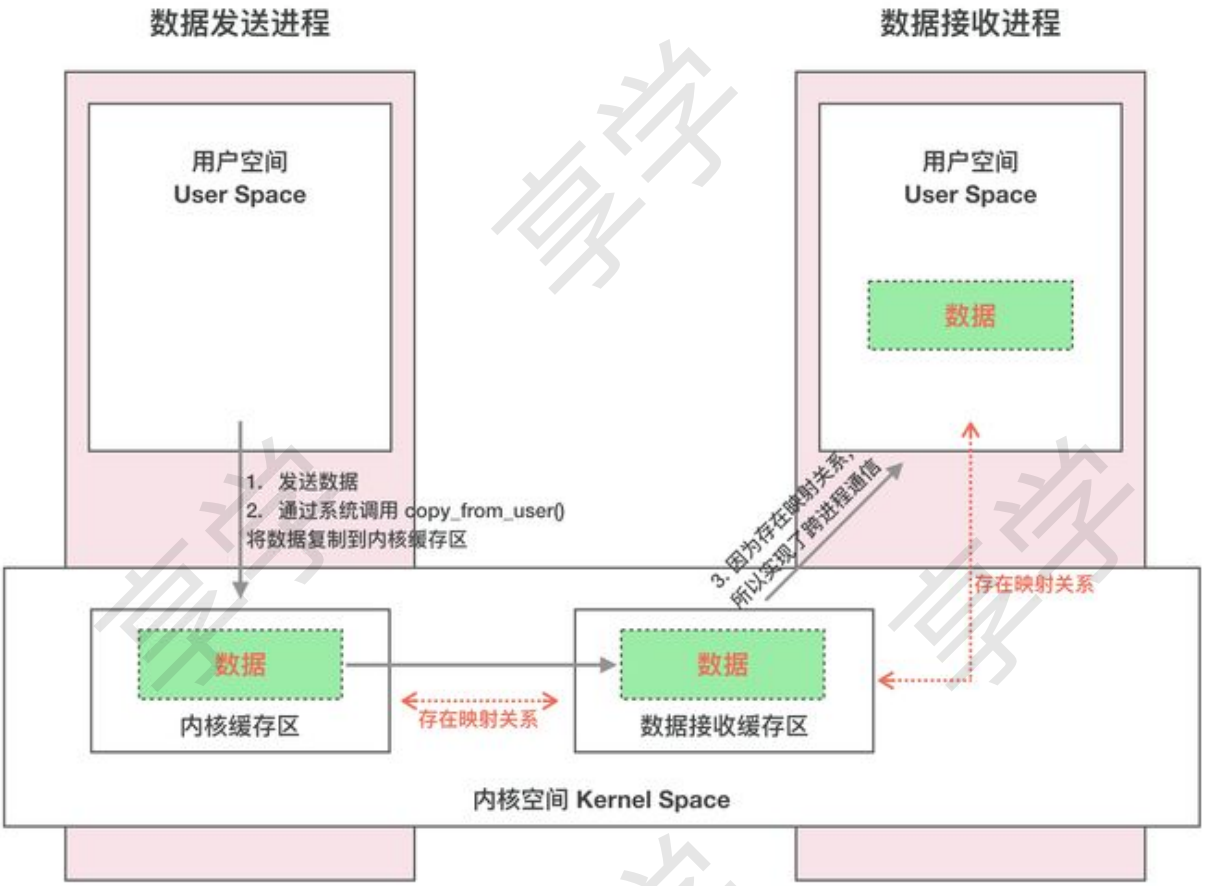
比如进程中的用户区域是不能直接和物理设备打交道的，如果想要把磁盘上的数据读取到进程的用户区域，需要两次拷贝（磁盘-->内核空间-->用户空间）；通常在这种场景下 mmap() 就能发挥作用，通过在物理介质和用户空间之间建立映射，减少数据的拷贝次数，用内存读写取代I/O读写，提高文件读取效率。

而 Binder 并不存在物理介质，因此 Binder 驱动使用 mmap() 并不是为了在物理介质和用户空间之间建立映射，而是用来在内核空间创建数据接收的缓存空间。

一次完整的 Binder IPC 通信过程通常是这样：

1. 首先 Binder 驱动在内核空间创建一个数据接收缓存区；
2. 接着在内核空间开辟一块内核缓存区，建立**内核缓存区**和**内核中数据接收缓存区**之间的映射关系，以及**内核中数据接收缓存区**和**接收进程用户空间地址**的映射关系；
3. 发送方进程通过系统调用 copy*from*user() 将数据 copy 到内核中的**内核缓存区**，由于内核缓存区和接收进程的用户空间存在内存映射，因此也就相当于把数据发送到了接收进程的用户空间，这样便完成了一次进程间的通信。

如下图：



Binder的优势

Android 系统是基于 Linux 内核的，Linux 已经提供了管道、消息队列、共享内存和 Socket 等 IPC 机制。那为什么 Android 还要提供 Binder 来实现 IPC 呢？主要是基于**性能**、**稳定性**和**安全性**几方面的原因。

性能

首先说说性能上的优势。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。Binder 只需要一次数据拷贝，性能上仅次于共享内存。

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

稳定性

再说说稳定性，Binder 基于 C/S 架构，客户端（Client）有什么需求就丢给服务端（Server）去完成，架构清晰、职责明确又相互独立，自然稳定性更好。共享内存虽然无需拷贝，但是控制负责，难以使用。从稳定性的角度讲，Binder 机制是优于内存共享的。

安全性

另一方面就是安全性。Android 作为一个开放性的平台，市场上有各类海量的应用供用户选择安装，因此安全性对于 Android 平台而言极其重要。作为用户当然不希望我们下载的 APP 偷偷读取我的通信录，上传我的隐私数据，后台偷跑流量、消耗手机电量。传统的 IPC 没有任何安全措施，完全依赖上层协议来确保。首先传统的 IPC 接收方无法获得对方可靠的进程用户ID/进程ID（UID/PID），从而无法鉴别对方身份。Android 为每个安装好的 APP 分配了自己的 UID，故而进程的 UID 是鉴别进程身份的重要标志。传统的 IPC 只能由用户在数据包中填入 UID/PID，但这样不可靠，容易被恶意程序利用。可靠的身份标识只有由 IPC 机制在内核中添加。其次传统的 IPC 访问接入点是开放的，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。同时 Binder 既支持实名 Binder，又支持匿名 Binder，安全性高。

基于上述原因，Android 需要建立一套新的 IPC 机制来满足系统对稳定性、传输性能和安全性方面的要求，这就是 Binder。

最后用一张表格来总结下 Binder 的优势：

优势	描述
性能	只需要一次数据拷贝，性能上仅次于共享内存
稳定性	基于 C/S 架构，职责明确、架构清晰，因此稳定性好
安全性	为每个 APP 分配 UID，进程的 UID 是鉴别进程身份的重要标志

10.3 为什么 Android 要采用 Binder 作为 IPC 机制?

详细讲解

享学课堂移动开发课程：Framework专题 Binder分析部分

详细视频讲解：https://wx59a7e2633f445ef7.wx.finezb.com/share/recording/de17eb75126929454782d38164558533cb8980a2dfbeb3bca83b6ea6f005ca53

这道题想考察什么?

Binder作为IPC机制的优势。

考生应该如何回答

简单来说，Binder 是android系统工程师为android 定制的一个跨进程通信方法，当然它也不是android 系统原创的，是参考了OpenBinder的实现而引进到Google的。Binder是综合了android系统的特点，从性能，设计架构，安全性等几个方面的综合平衡而设计的，具体的关于Binder的实现细节，朋友们可以参考 上面的题目 《描述下Binder机制原理》进行系统学习。

应该从几个方面与传统IPC机制做对比。

1. 性能方面
    - 拷贝数据需要花时间,Binder只需拷贝一次，共享内存无需拷贝，其他的需要拷贝两次。
    - 从速度上来说，Binder仅次于共享内存，优于Socket，消息队列，管道，信号，信号量等。
2. 特点方面
    - Binder：基于C/S 架构，易用性高。
    - 共享内存：
        - 多个进程共享同一块内存区域，必然需要某种同步机制。
        - 使用麻烦，容易出现数据不同步，死锁等问题。
    - Socket：
        - 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。
3. 安全性方面
    - Binder：(安全性高)
        - 为每个APP分配不同UID，通过UID鉴别进程身份。
        - 即支持实名Binder，又支持匿名Binder。
    - 传统IPC：(不安全)
        - 完全依赖上层协议，只能由用户在数据包中填入UID/PID。
        - 访问接入点是开放的，任何程序都可以与其建立连接。

通过上面几个比较，特别是安全性这块，所以最终Android选择使用Binder机制进行通信。

10.4 Binder线程池的工作过程是什么样？（东方头条）

详细讲解

享学课堂移动开发课程：Framework专题 Binder分析部分

这道题想考察什么？

这道题想考察同学对于Binder线程池的理解。

考生应该如何回答

总的来说，有以下几点情况：1）binder线程池并非一个传统意义上的线程池结构，它在client进程中只有一个继承自Thread的PoolThread类。而线程的启动以及管理都是由binderDriver来控制的。2）binder线程有主线程和非主线程之分，主线程是启动的时候才会有的，每个binder线程池只有一个。其他情况下申请的都是非主线程。 3）binder线程池启动的时候，实际上只是启动了client中的binder主线程。4）binder线程(非主线程)有两种情况启动：client进程向binderDriver发送IPC请求，以及 client进程向binderDriver回复IPC请求结果。5）binder线程池的默认大小是16，1个主线程和15个非主线程。更多的细节我们就需要从Binder线程创建开始讲解了，大家可以参考下面的内容：

1.Binder线程创建

Binder线程创建与其所在进程的创建中产生，Java层进程的创建都是通过Process.start()方法，向Zygote进程发出创建进程的socket消息，Zygote收到消息后会调用Zygote.forkAndSpecialize()来fork出新进程，在新进程中会调用到RuntimeInit.nativeZygoteInit方法，该方法经过jni映射，最终会调用到app_main.cpp中的onZygoteInit，那么接下来从这个方法说起。

2.onZygoteInit

```
// app_main.cpp
virtual void onZygoteInit() {
    //获取ProcessState对象
    sp<ProcessState> proc = ProcessState::self();
    //启动新binder线程
    proc->startThreadPool();
}
```

ProcessState::self()是单例模式，主要工作是调用open()打开/dev/binder驱动设备，再利用mmap()映射内核的地址空间，将Binder驱动的fd赋值ProcessState对象中的变量mDriverFD，用于交互操作。startThreadPool()是创建一个新的binder线程，不断进行talkWithDriver()。

3.PS.startThreadPool

```
// ProcessState.cpp
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);    //多线程同步
    if (!mThreadPoolStarted) {
        mThreadPoolStarted = true;
        spawnPooledThread(true);
    }
}
```

启动Binder线程池后, 则设置mThreadPoolStarted=true. 通过变量mThreadPoolStarted来保证每个应用进程只允许启动一个binder线程池, 且本次创建的是binder主线程(isMain=true). 其余binder线程池中的线程都是由Binder驱动来控制创建的。

4.PS.spawnPooledThread

```
// ProcessState.cpp
void ProcessState::spawnPooledThread(bool isMain)
{
    if (mThreadPoolStarted) {
        //获取Binder线程名
        String8 name = makeBinderThreadName();
        //此处isMain=true
        sp<Thread> t = new PoolThread(isMain);
        t->run(name.string());
    }
}
```

4-1.makeBinderThreadName

```
// ProcessState.cpp
String8 ProcessState::makeBinderThreadName() {
    int32_t s = android_atomic_add(1, &mThreadPoolSeq);
    String8 name;
    name.appendFormat("Binder_%X", s);
    return name;
}
```

获取Binder线程名，格式为Binder_x, 其中x为整数。每个进程中的binder编码是从1开始，依次递增; 只有通过spawnPooledThread方法来创建的线程才符合这个格式，对于直接将当前线程通过joinThreadPool加入线程池的线程名则不符合这个命名规则。 另外,目前Android N中Binder命令已改为Binder:_x格式, 则对于分析问题很有帮忙,通过binder名称的pid字段可以快速定位该binder线程所属的进程p.

4-2.PoolThread.run

```
// ProcessState.cpp
class PoolThread : public Thread
{
public:
    PoolThread(bool isMain)
        : mIsMain(isMain)
    {
    }

protected:
    virtual bool threadLoop() {
        IPCThreadState::self()->joinThreadPool(mIsMain);
        return false;
    }
    const bool mIsMain;
};
```

从函数名看起来是创建线程池，其实就只是创建一个线程，该PoolThread继承Thread类。t->run()方法最终调用PoolThread的threadLoop()方法。

5. IPC.joinThreadPool

```
// IPCThreadState.cpp
void IPCThreadState::joinThreadPool(bool isMain)
{
    //创建Binder线程
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
    set_sched_policy(mMyThreadId, SP_FOREGROUND); //设置前台调度策略

    status_t result;
    do {
        processPendingDerefs(); //清除队列的引用
        result = getAndExecuteCommand(); //处理下一条指令

        if (result < NO_ERROR && result != TIMED_OUT
                && result != -ECONNREFUSED && result != -EBADF) {
            abort();
        }

        if(result == TIMED_OUT && !isMain) {
            break; ////非主线程出现timeout则线程退出
        }
    } while (result != -ECONNREFUSED && result != -EBADF);

    mOut.writeInt32(BC_EXIT_LOOPER);  // 线程退出循环
    talkWithDriver(false); //false代表bwr数据的read_buffer为空
}
```

- 对于isMain=true的情况下， command为BC_ENTER_LOOPER，代表的是Binder主线程，不会退出的线程；
- 对于isMain=false的情况下，command为BC_REGISTER_LOOPER，表示是由binder驱动创建的线程。

6. processPendingDerefs

```
// IPCThreadState.cpp
void IPCThreadState::processPendingDerefs()
{
    if (mIn.dataPosition() >= mIn.dataSize()) {
        size_t numPending = mPendingWeakDerefs.size();
        if (numPending > 0) {
            for (size_t i = 0; i < numPending; i++) {
                RefBase::weakref_type* refs = mPendingWeakDerefs[i];
                refs->decWeak(mProcess.get()); //弱引用减一
            }
            mPendingWeakDerefs.clear();
        }

        numPending = mPendingStrongDerefs.size();
        if (numPending > 0) {
            for (size_t i = 0; i < numPending; i++) {
                BBinder* obj = mPendingStrongDerefs[i];
                obj->decStrong(mProcess.get()); //强引用减一
            }
            mPendingStrongDerefs.clear();
        }
```

```
    }
}
```

7. getAndExecuteCommand

```
// IPCThreadState.cpp
status_t IPCThreadState::getAndExecuteCommand()
{
    status_t result;
    int32_t cmd;

    result = talkWithDriver(); //与binder进行交互
    if (result >= NO_ERROR) {
        size_t IN = mIn.dataAvail();
        if (IN < sizeof(int32_t)) return result;
        cmd = mIn.readInt32();

        pthread_mutex_lock(&mProcess->mThreadCountLock);
        mProcess->mExecutingThreadsCount++;
        pthread_mutex_unlock(&mProcess->mThreadCountLock);

        result = executeCommand(cmd); //执行Binder响应码

        pthread_mutex_lock(&mProcess->mThreadCountLock);
        mProcess->mExecutingThreadsCount--;
        pthread_cond_broadcast(&mProcess->mThreadCountDecrement);
        pthread_mutex_unlock(&mProcess->mThreadCountLock);

        set_sched_policy(mMyThreadId, SP_FOREGROUND);
    }
    return result;
}
```

8. talkWithDriver

```
//mOut有数据，mIn还没有数据。doReceive默认值为true
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    binder_write_read bwr;
    ...
    // 当同时没有输入和输出数据则直接返回
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;
    ...

    do {
        //ioctl执行binder读写操作，经过syscall，进入Binder驱动。调用Binder_ioctl
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        ...
    } while (err == -EINTR);
    ...
```

```
    return err;
}
```

在这里调用的isMain=true，也就是向mOut例如写入的便是BC_ENTER_LOOPER. 经过talkWithDriver(), 接下来程序往哪进行呢？在文章彻底理解Android Binder通信架构详细讲解了Binder通信过程，那么从binder_thread_write()往下说BC_ENTER_LOOPER的处理过程。

8-1.binder_thread_write

```
// binder.c
static int binder_thread_write(struct binder_proc *proc,
            struct binder_thread *thread,
            binder_uintptr_t binder_buffer, size_t size,
            binder_size_t *consumed)
{
    uint32_t cmd;
    void __user *buffer = (void __user *)(uintptr_t)binder_buffer;
    void __user *ptr = buffer + *consumed;
    void __user *end = buffer + size;
    while (ptr < end && thread->return_error == BR_OK) {
        //拷贝用户空间的cmd命令，此时为BC_ENTER_LOOPER
        if (get_user(cmd, (uint32_t __user *)ptr)) -EFAULT;
        ptr += sizeof(uint32_t);
        switch (cmd) {
          case BC_REGISTER_LOOPER:
              if (thread->looper & BINDER_LOOPER_STATE_ENTERED) {
                //出错原因：线程调用完BC_ENTER_LOOPER，不能执行该分支
                thread->looper |= BINDER_LOOPER_STATE_INVALID;

              } else if (proc->requested_threads == 0) {
                //出错原因：没有请求就创建线程
                thread->looper |= BINDER_LOOPER_STATE_INVALID;

              } else {
                proc->requested_threads--;
                proc->requested_threads_started++;
              }
              thread->looper |= BINDER_LOOPER_STATE_REGISTERED;
              break;

          case BC_ENTER_LOOPER:
              if (thread->looper & BINDER_LOOPER_STATE_REGISTERED) {
                //出错原因：线程调用完BC_REGISTER_LOOPER，不能立刻执行该分支
                thread->looper |= BINDER_LOOPER_STATE_INVALID;
              }
              //创建Binder主线程
              thread->looper |= BINDER_LOOPER_STATE_ENTERED;
              break;

          case BC_EXIT_LOOPER:
              thread->looper |= BINDER_LOOPER_STATE_EXITED;
              break;
        }
```

```
        ...
    }
    *consumed = ptr - buffer;
  }
  return 0;
}
```

处理完BC_ENTER_LOOPER命令后，一般情况下成功设置thread->looper |= BINDER_LOOPER_STATE_ENTERED。那么binder线程的创建是在什么时候呢？ 那就当该线程有事务需要处理的时候，进入binder_thread_read()过程。

8-2.binder_thread_read

```
binder_thread_read () {
  ...
retry:
    //当前线程todo队列为空且transaction栈为空，则代表该线程是空闲的
    wait_for_proc_work = thread->transaction_stack == NULL &&
        list_empty(&thread->todo);

    if (thread->return_error != BR_OK && ptr < end) {
        ...
        put_user(thread->return_error, (uint32_t __user *)ptr);
        ptr += sizeof(uint32_t);
        goto done; //发生error，则直接进入done
    }

    thread->looper |= BINDER_LOOPER_STATE_WAITING;
    if (wait_for_proc_work)
        proc->ready_threads++; //可用线程个数+1
    binder_unlock(__func__);

    if (wait_for_proc_work) {
        if (non_block) {
            ...
        } else
            //当进程todo队列没有数据,则进入休眠等待状态
            ret = wait_event_freezable_exclusive(proc->wait, binder_has_proc_work(proc,
thread));
    } else {
        if (non_block) {
            ...
        } else
            //当线程todo队列没有数据，则进入休眠等待状态
            ret = wait_event_freezable(thread->wait, binder_has_thread_work(thread));
    }

    binder_lock(__func__);
    if (wait_for_proc_work)
        proc->ready_threads--; //可用线程个数-1
    thread->looper &= ~BINDER_LOOPER_STATE_WAITING;

    if (ret)
        return ret; //对于非阻塞的调用，直接返回
```

```
    while (1) {
        uint32_t cmd;
        struct binder_transaction_data tr;
        struct binder_work *w;
        struct binder_transaction *t = NULL;

        //先考虑从线程todo队列获取事务数据
        if (!list_empty(&thread->todo)) {
            w = list_first_entry(&thread->todo, struct binder_work, entry);
        //线程todo队列没有数据，则从进程todo对获取事务数据
        } else if (!list_empty(&proc->todo) && wait_for_proc_work) {
            w = list_first_entry(&proc->todo, struct binder_work, entry);
        } else {
            ... //没有数据,则返回retry
        }

        switch (w->type) {
            case BINDER_WORK_TRANSACTION: ...  break;
            case BINDER_WORK_TRANSACTION_COMPLETE:...  break;
            case BINDER_WORK_NODE: ...    break;
            case BINDER_WORK_DEAD_BINDER:
            case BINDER_WORK_DEAD_BINDER_AND_CLEAR:
            case BINDER_WORK_CLEAR_DEATH_NOTIFICATION:
                struct binder_ref_death *death;
                uint32_t cmd;

                death = container_of(w, struct binder_ref_death, work);
                if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION)
                  cmd = BR_CLEAR_DEATH_NOTIFICATION_DONE;
                else
                  cmd = BR_DEAD_BINDER;
                put_user(cmd, (uint32_t __user *)ptr;
                ptr += sizeof(uint32_t);
                put_user(death->cookie, (void * __user *)ptr);
                ptr += sizeof(void *);
                ...
                if (cmd == BR_DEAD_BINDER)
                  goto done; //Binder驱动向client端发送死亡通知，则进入done
                break;
        }

        if (!t)
            continue; //只有BINDER_WORK_TRANSACTION命令才能继续往下执行
        ...
        break;
    }

done:
    *consumed = ptr - buffer;
    //创建线程的条件
    if (proc->requested_threads + proc->ready_threads == 0 &&
        proc->requested_threads_started < proc->max_threads &&
```

```
        (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
         BINDER_LOOPER_STATE_ENTERED))) {
        proc->requested_threads++;
        // 生成BR_SPAWN_LOOPER命令，用于创建新的线程
        put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer);
    }
    return 0;
}
```

当发生以下3种情况之一，便会进入done：

- 当前线程的return_error发生error的情况；
- 当Binder驱动向client端发送死亡通知的情况；
- 当类型为BINDER_WORK_TRANSACTION(即收到命令是BC_TRANSACTION或BC_REPLY)的情况；

任何一个Binder线程当同时满足以下条件，则会生成用于创建新线程的BR_SPAWN_LOOPER命令：

1. 当前进程中没有请求创建binder线程，即requested_threads = 0；
2. 当前进程没有空闲可用的binder线程，即ready_threads = 0； （线程进入休眠状态的个数就是空闲线程数）
3. 当前进程已启动线程个数小于最大上限(默认15)；
4. 当前线程已接收到BC_ENTER_LOOPER或者BC_REGISTER_LOOPER命令，即当前处于BINDER_LOOPER_STATE_REGISTERED或者BINDER_LOOPER_STATE_ENTERED状态。前面已设置状态为BINDER_LOOPER_STATE_ENTERED，显然这条件是满足的。

从system_server的binder线程一直的执行流: IPC.joinThreadPool –> IPC.getAndExecuteCommand() -> IPC.talkWithDriver() ,但talkWithDriver收到事务之后, 便进入IPC.executeCommand(), 接下来,从executeCommand说起.

** 9.IPC.executeCommand**

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    status_t result = NO_ERROR;
    switch ((uint32_t)cmd) {
      ...
      case BR_SPAWN_LOOPER:
          //创建新的binder线程
          mProcess->spawnPooledThread(false);
          break;
      ...
    }
    return result;
}
```

Binder主线程的创建是在其所在进程创建的过程一起创建的，后面再创建的普通binder线程是由spawnPooledThread(false)方法所创建的。

9. 总结

Binder系统中可分为3类binder线程：

- Binder主线程：进程创建过程会调用startThreadPool()过程中再进入spawnPooledThread(true)，来创建Binder主线程。编号从1开始，也就是意味着binder主线程名为binder_1，并且主线程是不会退出的。

- Binder普通线程：是由Binder Driver来根据是否有空闲的binder线程来决定是否创建binder线程，回调spawnPooledThread(false) ，isMain=false，该线程名格式为binder_x。
- Binder其他线程：其他线程是指并没有调用spawnPooledThread方法，而是直接调用IPC.joinThreadPool()，将当前线程直接加入binder线程队列。例如： mediaserver和servicemanager的主线程都是binder线程，但system_server的主线程并非binder线程。

Binder的transaction有3种类型：

1. call: 发起进程的线程不一定是在Binder线程， 大多数情况下，接收者只指向进程，并不确定会有哪个线程来处理，所以不指定线程；
2. reply: 发起者一定是binder线程，并且接收者线程便是上次call时的发起线程(该线程不一定是binder线程，可以是任意线程)。
3. async: 与call类型差不多，唯一不同的是async是oneway方式不需要回复，发起进程的线程不一定是在Binder线程， 接收者只指向进程，并不确定会有哪个线程来处理，所以不指定线程。

10.5 AIDL 的全称是什么？如何工作？能处理哪些类型的数据?

详细讲解

享学课堂移动开发课程：Framework专题 Binder分析部分

这道题想考察什么?

这道题想考察同学对AIDL的理解。

考生应该如何回答

AIDL全称是什么?

AIDL 全称是 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，生成的代码封装了binder，可以当成是binder的延伸。

AIDL实质

AIDL的使用实质就是对Binder机制的封装，Binder原理详见《8.2 描述下Binder机制原理》，主要就是将Binder封装成一个代理对象proxy，从用户的角度看，就像是客户端直接调用了服务端的代码。

AIDL使用方法

服务端：1.创建aidl接口；2.实现接口，并向客户端开放接口；3.创建服务，返回binder。

客户端：1.绑定服务；2.实现ServiceConnection绑定监听；3.在绑定成功的回调中，将IBinder转换成AIDL的接口代理对象。

客户端和服务端绑定成功后，就可以通过AIDL的接口代理对象，就像直接调用本地方法一样，调用服务端的方法了。需要注意的是，AIDL间传递的对象要实现Parcelable接口。

AIDL绑定过程

客户端先调用bindService方法，发起绑定服务的请求，通过ServiceManager，拿到ActivityManagerService，也就是AMS，然后通过AMS向服务端发起bindService的请求。然后服务端接收到绑定请求，以Handler消息机制的方式，发送一个绑定服务的Message，然后在ActivityThread中处理这个绑定请求，调用onBind函数，并返回对应的IBinder对象。这个返回IBinder对象的操作，基本就和绑定过程的通过ServiceManager、AMS类似了。

AIDL支持的数据类型

1. Java基本数据类型(int,boolean等)
2. 引用数据类型，比如：String与CharSequence，当然也可以使用自定义数据类型。
3. 自定义的数据类型，但是有个条件，那就是数据需要实现了 Parcelable 接口。
4. List, Map和Parcelables类型, 这些类型内所包含的数据成员也只能是基本数据类型、String等其他支持的类型。

10.6 Android中Pid&Uid的区别和联系

这道题想考察什么?

考察Android 系统中Pid 以及Uid的概念和理解

考试应该如何回答

众所周知，Pid是进程ID，Uid是用户ID，只是Android和计算机不一样，计算机每个用户都具有一个Uid，哪个用户start的程序，这个程序的Uid就是那个那个用户，而Android中每个程序都有一个Uid，默认情况下，Android会给每个程序分配一个普通级别互不相同的 Uid，如果用互相调用，只能是Uid相同才行，这就使得共享数据具有了一定安全性，每个软件之间是不能随意获得数据的。而同一个application 只有一个Uid，所以application下的Activity之间不存在访问权限的问题。

PID

Android中的PID全称为Process Identifier，来源于Linux中，在进程启动的时候系统会为进程分配一个独一无二的标识，进程销毁后PID会被系统回收，但是在Android中一般不会重新分配，后面的进程PID会比前面的进程的大。但是对同一个安卓应用可以具有多个PID，添加也很方便，只需要在声明类时指定进程名称即可,代码如下:

```
<activity android:name=".TestActivty" android:process="com.xiaohan.test"/>
```

添加很方便，但是不能随意使用，因为在同一个应用（PID）中，设计到程序之间最多的是线程间的通信，一旦独立出PID则涉及到进程间通信，类似于不同的两个应用，当然也可以通过上面的私有暴露和权限暴露的方式实现数据的通信，但是系统的开销较大。

UID

Android中的UID一般认为是User Identifier,同样来源于Linux中。但是在Android中不太一样，Android最初的设计是单用户，所以UID并不是为了区别用户的，而是为了不同程序间进行数据共享。Android中每个程序都有一个Uid，默认情况下，Android会给每个程序分配一个普通级别互不相同的Uid，**因此如果程序期望互相调用，只有Uid相同才行**，这就使得共享数据具有了一定安全性，每个软件之间是不能随意获得数据的。而同一个application只有一个Uid，所以application下的Activity之间不存在访问权限的问题。

android中uid用于标识一个应用程序，uid在应用安装时被分配，并且在应用存在于手机上期间，都不会改变。一个应用程序只能有一个uid，多个应用可以使用sharedUserId 方式共享同一个uid，前提是这些应用的签名要相同。

UID如何分配

在应用启动的时候，或者手机启动的时候，PackageManagerService都会去解析apk，并且为app分配一个UID，具体的流程如下（基于Android11）：

PackageManagerService.java 中在PMS 构造函数初始化的时候会执行到一个函数scanDirTracedLI(),在scanDirTracedLI()中会执行scanDirLI(),然后会执行scanPackageTracedLI(),然后再执行scanPackageLI(),然后再执行addForInitLI()，然后执行scanPackageNewLI(),当然这个过程中会涉及到很多代码细节，建议大家去学习本章中关于PKMS部分的内容。在这个scanPackageNewLI函数中会执行mSettings.getSharedUserLPw()，去获取共享用户，如果没有就创建新的，大家看下面的代码：

```
SharedUserSetting getSharedUserLPw(String name, int pkgFlags, int pkgPrivateFlags,
        boolean create) throws PackageManagerException {
    SharedUserSetting s = mSharedUsers.get(name);
    if (s == null && create) {
        s = new SharedUserSetting(name, pkgFlags, pkgPrivateFlags);
        s.userId = acquireAndRegisterNewAppIdLPw(s); //核心代码
        if (s.userId < 0) {
            // < 0 means we couldn't assign a userid; throw exception
            throw new PackageManagerException(INSTALL_FAILED_INSUFFICIENT_STORAGE,
                    "Creating shared user " + name + " failed");
        }
        Log.i(PackageManagerService.TAG, "New shared user " + name + ": id=" + s.userId);
        mSharedUsers.put(name, s);
    }
    return s;
}
```

上面的代码显示，如果没有为这个app创建SharedUserSetting 信息且需要创建一个，那么就创建一个，然后再调用acquireAndRegisterNewAppIdLPw函数为其分配UID。

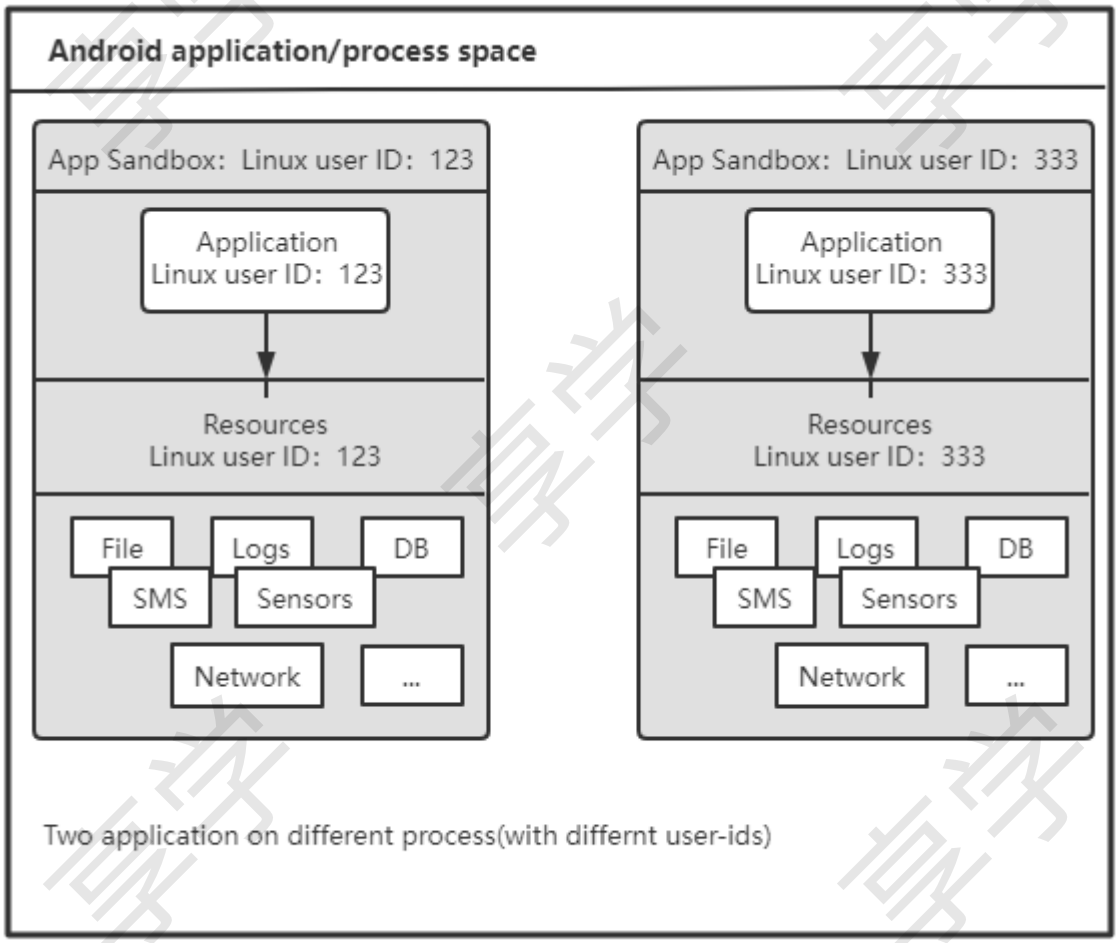
```
private int acquireAndRegisterNewAppIdLPw(SettingBase obj) {
    // Let's be stupidly inefficient for now...
    final int size = mAppIds.size();
    //code1 从0开始，找到第一个未使用的ID，此处对应之前有应用被移除的情况，复用之前的ID
    for (int i = mFirstAvailableUid; i < size; i++) {
        if (mAppIds.get(i) == null) {
            mAppIds.set(i, obj);
            return Process.FIRST_APPLICATION_UID + i;
        }
    }

   //code2 最多只能安装 9999 个应用
    // None left?
    if (size > (Process.LAST_APPLICATION_UID - Process.FIRST_APPLICATION_UID)) {
        return -1;
    }

    mAppIds.add(obj);
```

```
    // code3 可以解释为什么普通应用的UID 都是从 10000开始的
    return Process.FIRST_APPLICATION_UID + size;
}
```

从上面我们可以看到，给APP分配的UID是遵守如下几个原则：1）从code1知道，id是可以复用的，当一个app卸载了，那么它的id会被后面安装的app复用，2）从code2得知，用户id不能超过19999，也就是说可以安装的app数不能超过9999个，否则安装的app没有uid分配；3）每个UID 都是一个大于 Process.FIRST_APPLICATION_UID 的数字，也就是都大于10000；

Android中UID的价值

在Android中一个UID的对应的就是一个可执行的程序，程序在Android系统留存期间，其UID不变。 在Android中采用沙箱的概念来管理程序，不同的程序具有唯一的UID和PID，通过该UID来标识其所具有的“资源”，包括文件目录、数据库的访问、网络、传感器和日志等，和Linux一样，相互之间互不影响。



不同的应用程序一般是运行在不同的进程中，相互之间的“资源”不可以访问，但可以通过进程共享的方式，实现不同程序之间的数据访问主要是针对Activity、Service和ContentProvider，其实现方式按照权限暴露级别分为：完全暴露、权限提示暴露和私有暴露三种方式。

完全暴露

是指通过android:exported=”true”实现，在AndroidMaindfest.xml中申明Activity、Service或ContentProvider时，将该属性设置为true后，就表明该类允许外界的数据访问。如果在申明时添加了intentFilter属性，则默认exported就为true，此时也可强制的设置为fasle。如未做其他设置（exported/intentFilter），则默认exported为fasle，即不对外暴露。如下代码所示：

```
<activity android:name=".TestActivty" android:exported="true"/>
```

权限提示暴露

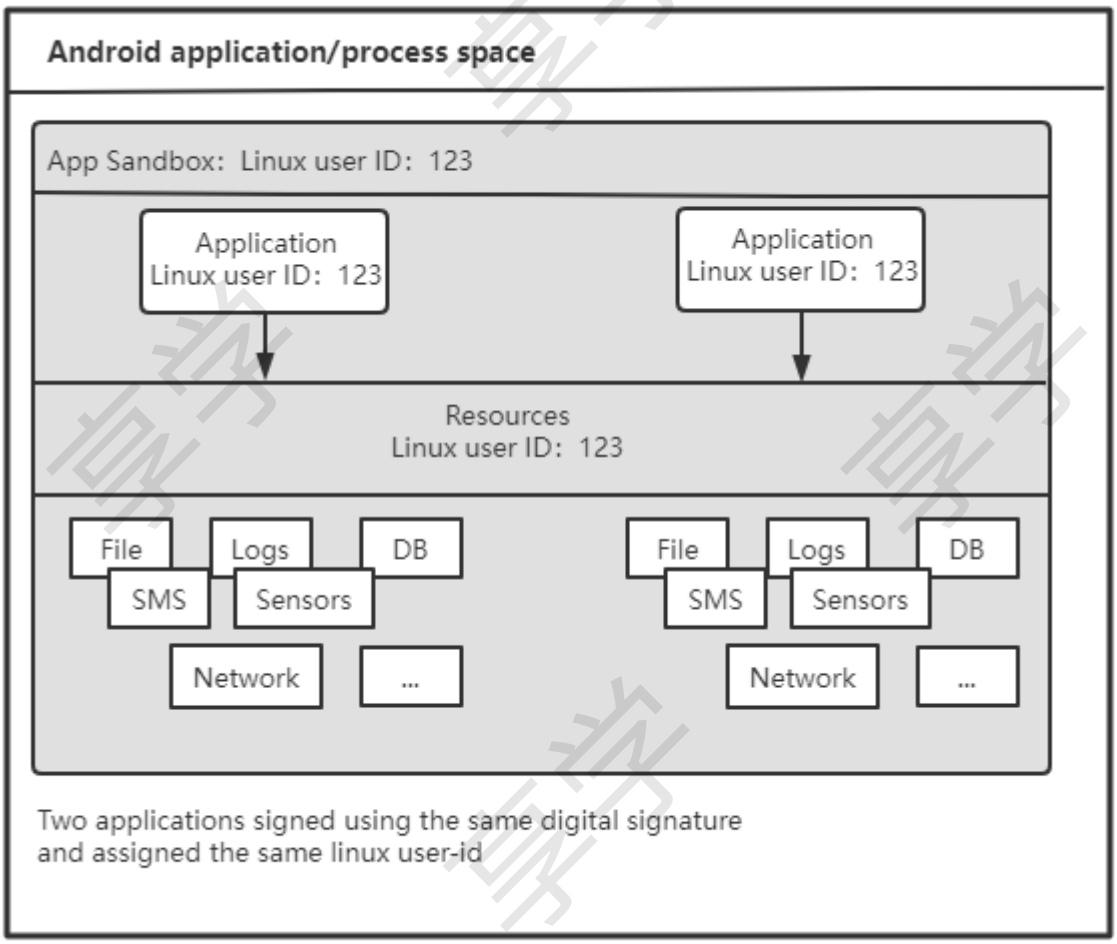
在AndroidMaindfest.xml中申明Activity、Service或ContentProvider时，添加了permission，表明如果其他应用需要访问该类时，需要在该应用添加该类声明时的权限，声明私有的权限，如下所示。

```
<activity android:name=".TestActivty" android:permission="com.xiaohan.permission"/>
//添加访问的权限说明
<uses-permission android:name="com.xiaohan.permission"/>
```

私有暴露

不同于以上两种类的暴露方式，如果想对于不同应用之间可以互相访问任何数据，则需要通过sharedUserId+同一套签名的方式实现，只有这样才能运行在同一个进程中（同一个沙箱中）保证数据的相互访问。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.xiaohan.test"       //应用包名
    android:sharedUserId="com.xiaohan.sharedUID"  //暴露的唯一标识
    android:sharedUserLabel="@string/app_name"  //必须是引入资源文件中的字符串
    android:versionCode="1"
    android:versionName="1.1.0"
    //安装位置，默认在内部目录，还包括auto：自动、  preferExternal：外置SD卡中
    android:installLocation="internalOnly">
```



总结

不同的应用具有唯一的UID，同一个UID可具有不同的PID；

针对不同的PID之间数据的暴露可采用私有暴露和权限暴露，针对不同的UID之间可通过完全暴露的方式；

如果一个应用是系统应用，则不需要其他应用暴露，便可直接访问该应用的数据。

10.7 Handler怎么进行线程通信，原理是什么？（东方头条）

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分；

深入原理讲解：https://wx59a7e2633f445ef7.wx.finezb.com/share/recording/de17eb75126929454782d38164558533378b43ee2676ef1179aa56c53c2dc08a

这道题想考察什么？

这道题想考察同学对Handler的多线程通信原理是否清楚。

考生应该如何回答

Handler整体思想

在多线程的应用场景中，**将工作线程中需更新 `UI` 的操作信息 传递到 `UI` 主线程**，从而实现 工作线程对 `UI` 的更新处理，最终实现异步消息的处理。

Handler工作流程

`Handler` 机制的工作流程主要包括4个步骤：

1. 异步通信准备
2. 消息发送
3. 消息循环
4. 消息处理

具体如下图：

步骤	具体描述	备注
1. 异步通信准备	在主线程中创建： • 处理器 对象（Looper） • 消息队列 对象（Message Queue） • Handler 对象	• Looper、Message Queue均属于主线程 • 创建Message Queue后，Looper则自动进入消息循环 • 此时，Handler自动绑定了主线程的Looper、Message Queue
2. 消息入队	工作线程 通过 Handler 发送消息（Message）到消息队列（Message Queue）中	该消息内容 = 工作线程对UI的操作
3. 消息循环	• 消息出队：Looper循环取出 消息队列（Message Queue） 中的的消息（Message） • 消息分发：Looper将取出的消息（Message） 发送给 创建该消息的处理者(Handler)	在消息循环过程中，若消息队列为空，则线程阻塞
4. 消息处理	• 处理者(Handler）接收 处理器（Looper）发送过来的消息（Message） • 处理者(Handler）根据消息（Message）进行UI操作	

Handler工作流程图

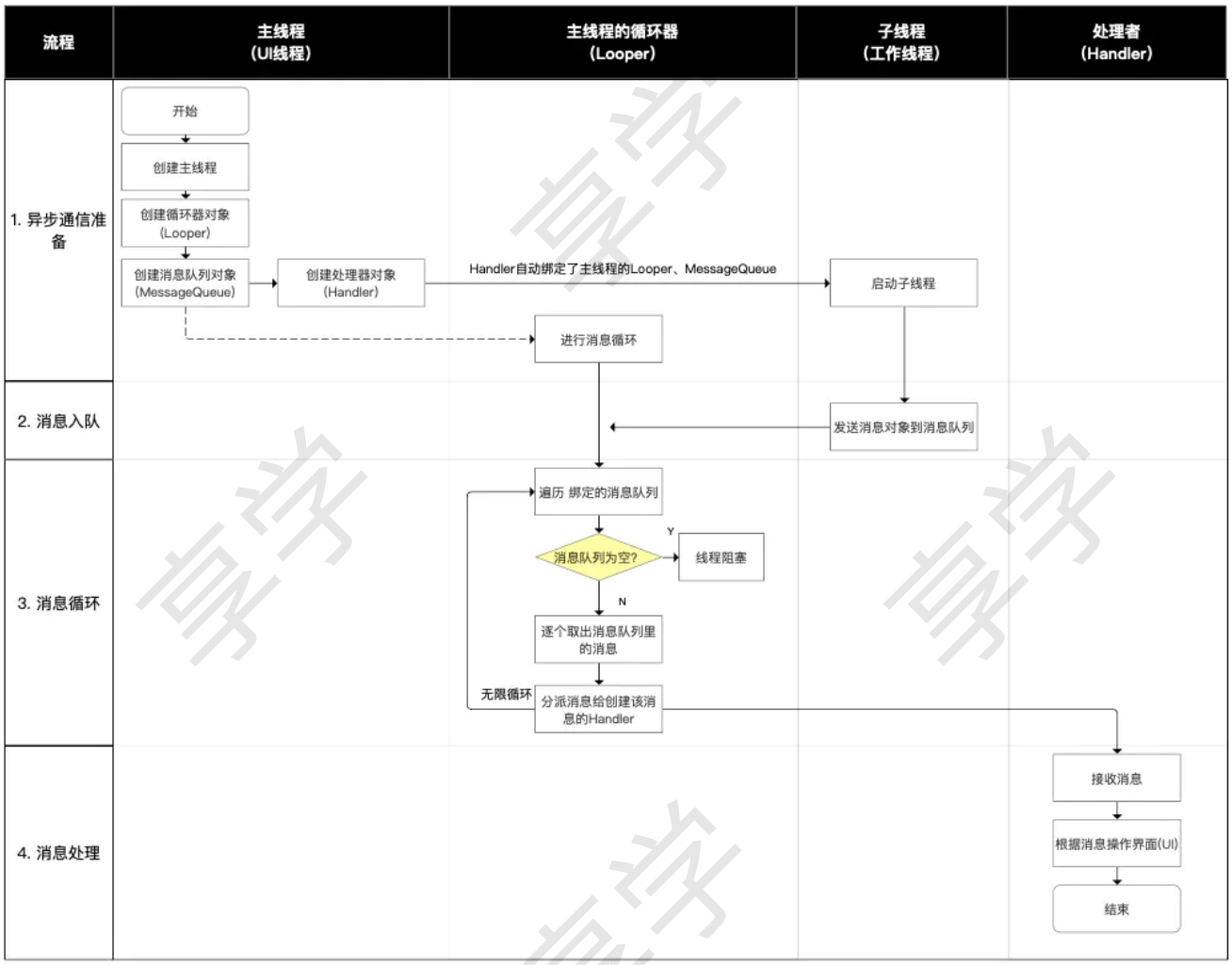
如下图所示，handler的总体流程大致如下：

首先，系统通过调用 Looper.prepare（）为线程准备Looper 和承接Message 的MessageQueue；

然后，系统再调用Looper.loop()函数，这个函数会开启一个死循环，在循环中不断的轮询MessageQueue消息队列，从消息队列中取出可以执行的Message消息，然后进行执行。

再然后，用户通过Handler.sendMessage这一类的函数调用，向MessageQueue里面不断的发送消息。

最后，由于Looper 中的loop是在不断轮询MessageQueue的，一旦发现MessageQueue里面有可执行的消息，那么就会将消息取出来，然后通过消息所携带的handler去执行。



总结

Handler 是一个消息管理机制，android 程序的运行必须依托于Handler机制，那么handler机制进行线程通信的原理思想如下： 1） 在启动handler通信机制的时候，首先会准备Looper，以及必须让Looper调用loop函数进入死循环；2）在Looper启动后，loop死循环就会不断的去轮询MessageQueue的next函数，试图通过这个方式获取到MessageQueue中存储的Message；3）当用户希望往当前创建handler的这个线程里面传递消息的时候，就可以调用这个handler的sendMessage或者postMessage 相关的函数，次数Message将被发送到对应的MessageQueue中；4）在第2）步中，loop里面的循环就会轮询到我们在第3）步中添加的消息，当这个消息被loop轮询到的时候，这个消息就会被处理掉，而且处理的线程就是当前轮询MessageQueue的线程。

总的来说，Handler机制可以理解为 在一个线程中创建了一个Message 对象（内存块），这个Message对象包含了我们要执行的动作；然后我们通过handler将这个Message内存块传给了MessageQueue，也就是说MessageQueue是存储的Message；然后Looper所在的线程通过loop()函数从MessageQueue中取到我们存储的Message，并在当前这个线程中去执行的过程。

10.8 ThreadLocal的原理，以及在Looper是如何应用的？（字节跳动、小米）

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么?

ThreadLocal 是必须要掌握的，这是因为 Looper 的工作原理，就跟 ThreadLocal 有很大的关系，理解 ThreadLocal 的实现方式有助于我们理解 Looper 的工作原理，这篇文章就从 ThreadLocal 的用法讲起，一步一步带大家理解 ThrealLocal。

考察的知识点

1. ThreadLocal的内部运行原理
2. Looper相关知识

考生应该如何回答

ThreadLocal 可以把一个对象保存在指定的线程中，对象保存后，只能在指定线程中获取保存的数据，对于其他线程来说则无法获取到数据。日常开发中 ThreadLocal 使用的地方比较少，但是系统在 Handler 机制中使用了它来保证每一个 Handler 所在的线程中都有一个独立的 Looper 对象，为了更好的理解 Handler 机制。

ThreadLocal 是什么

ThreadLocal 是一个关于创建线程局部变量的类。

其实就是这个变量的作用域是线程，其他线程访问不了。通常我们创建的变量是可以被任何一个线程访问的，而使用 ThreadLocal 创建的变量只能被当前线程访问，其他线程无法访问。

那么ThreadLocal是如何确保只有当前线程可以访问呢？我们先来分析一下ThreadLocal里面最重要的两个函数，get()，set()两个函数。

首先看下get()方法中的源码

```
public T get() {
    Thread t = Thread.currentThread();  //code 1
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}

public void set(T value) {
    Thread t = Thread.currentThread();  //code 2
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}

ThreadLocalMap getMap(Thread t) {
```

```
    return t.threadLocals;
}
    /**
     * Variant of set() to establish initialValue. Used instead
     * of set() in case user has overridden the set() method.
     *
     * @return the initial value
     */
    private T setInitialValue() {
        T value = initialValue();
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
        return value;
    }
```

从code1 和code2 大家应该不难发现，在给ThreadLocal去set值或者get值的时候都会先获取当前线程，然后基于线程去调用getMap(thread),getMap返回的就是线程thread的成员变量threadLocals。所以通过get 和set都执行对于的函数，这样就保证了threadLocal的访问，一定是只能访问或许修改当前线程的值，这就保障了这个变量是线程的局部变量。

那么接下来ThreadLocalMap又是什么呢?

ThreadLocalMap是什么

```
static class ThreadLocalMap {
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
    private Entry[] table;
    //省略部分代码
    }
```

从源码可以看出来ThreadLocalMap是一个数组，数组里面是继承自弱引用的Entry。弱引用的使用也是为了当出现异常情况，比如死循环的时候内存能得到回收。

再回过头来看一下set()的源码

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```

这里面的代码就很清晰，根据当前线程取出ThreadLocalMap，然后进行存储数据的操作，如果Map为空的话就先创建，再赋值。

探秘一下map.set()

```
private void set(ThreadLocal<?> key, Object value) {
    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len-1);

    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        ThreadLocal<?> k = e.get();

        if (k == key) {
            e.value = value;
            return;
        }

        if (k == null) {
            replaceStaleEntry(key, value, i);
            return;
        }
    }

    tab[i] = new Entry(key, value);
    int sz = ++size;
    if (!cleanSomeSlots(i, sz) && sz >= threshold)
        rehash();
}
```

代码中int i = key.threadLocalHashCode & (len-1)与HashMap中key的hash值方法一样，主要是避免hash值的冲突。再往下走是遍历Map有三种情况：1. 找存在的key有效，然后赋值；2. 找存在的key但是无效，替换掉过期的Entry；3. 没找到相同key值，新建一个Entry然后赋值。

那么ThreadLocal在Looper中又是如何应用的呢?

ThreadLocal在Looper中的应用

找到Looper的源码

```
static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();

 private static void prepare(boolean quitAllowed) {
        if (sThreadLocal.get() != null) {//code 1
            throw new RuntimeException("Only one Looper may be create per thread");
        }
        sThreadLocal.set(new Looper(quitAllowed));// code 2
    }
```

Looper与线程的关系是一对一的关系，不同线程之间Looper对象是隔离的，那么Looper是怎么保障这一点的呢？通过上面的代码大家应该不难发现Looper初始化是必须调用prepare函数进行，在调用prepare函数的时候代码会执行到code 1,在code1会先去判断当前线程对于的ThreadLocal中是否存在looper的value，如果存在，那么就抛出异常，这样的执行就保证了一个线程只会设置一次Looper。这个代码的执行流程，就是确保了一个线程只有一个ThreadLocal，一个ThreadLocal就只有一个looper。

总结

ThreadLocal是一个创建线程局部变量的类，它的实现机制决定了这个变量的作用域是线程，其他线程访问不了，利用这个机制，可以保障一个线程只有唯一的一个ThreadLocal变量。然后，在looper中通过prepare函数的设计，确保了一个ThreadLocal 只会和一个Looper进行绑定。通过这两个方式确保了一个线程只有一个ThreadLocal变量，一个ThreadLocal变量只有一个Looper，从而形成了一一对应的关系。

10.9 Handler如果没有消息处理是阻塞的还是非阻塞的？（字节跳动、小米）

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么？

这道题想考察同学对Handler消息处理的理解。

考生应该如何回答

Handler消息机制原理查看《 **Handler怎么进行线程通信，原理是什么？** 》一题。 消息处理这块的代码是 `Looper.loop()`，里面会执行 `MessageQueeu.next()` 方法，代码如下：

```
for (;;) {
    Message msg = queue.next(); // might block
    if (msg == null) {
        // No message indicates that the message queue is quitting.
        return;
    }
```

MessageQueen.next代码如下：

```
Message next() {
    for (;;) {
        nativePollOnce(ptr, nextPollTimeoutMillis);
    }
}
```

当没有消息时，或者消息不满足处理条件，则 `nativePollOnce` 方法会阻塞线程，直到有满足条件的消息到来。

10.10 handler.post(Runnable) runnable是如何执行的？（字节跳动、小米）

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么？

是否了解Handler的运行机制？

考察的知识点

Handler的内部原理

考生应该如何回答

主要分析runnable 是如何被封装成为一个message，以及它如何被添加到messageQueue里面的过程，最后再分析这个message如何运行的过程。

Runnable分发

首先我们看一下handler.post(Runnable)的相关源码

```
public final boolean post(@NonNull Runnable r) {
       return  sendMessageDelayed(getPostMessage(r), 0);
}

private static Message getPostMessage(Runnable r) {
        Message m = Message.obtain();
        m.callback = r;
        return m;
}

public final boolean sendMessage(@NonNull Message msg) {
        return sendMessageDelayed(msg, 0);
}
```

从上面代码我们可以看出，`post(Runnable)`方法使用到的sendMessageDelayed函数，其实也就是调用了`sendMessage(Message msg)`。只是它使用到了我们的`getPostMessage`函数，将我们的Runnable转化为了我们的Message，由此可知handler.post(Runnable) 分发阶段实质上是和普通的Message是一样的。

Runnable执行

接着Message会通过`queue.enqueueMessage(msg, uptimeMillis)`方法被添加到MessageQueen中，由`Looper#loop`完成消息的分发执行。最终`Looper#loop`中会将消息发送至`Handler#dispatchMessage`处理：

```
/**
 * Handle system messages here.
 */
public void dispatchMessage(@NonNull Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}

private static void handleCallback(Message message) {
    message.callback.run();
}
```

在`Handler#dispatchMessage`中Handler.dispatchMessage中如果判断需要分发执行的Message对象中callback（即post的Runnable）不为空，最终调用的是callback.run()方法，完成runnable的执行动作。

10.11 Handler的Callback存在，但返回true，handleMessage是否会执行？（字节跳动、小米）

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么？

这道题想考察同学对Handler的Callback 的理解。

考生应该如何回答

首先，如果Handler的Callback存在，若Callback返回true，handleMessage则不会执行；若Callback返回false，则handleMessage仍然会执行。Handler对于消息的处理都会在`Handler#dispatchMessage`中完成分发，具体细节分析如下代码所示：

```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        // 1. 设置了Message.Callback (Runnable)
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            // 2. 设置了 Handler.Callback (Callback )
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        // 3. 未设置 Handler.Callback 或 返回 false
        handleMessage(msg);
    }
}
public interface Callback {
    public boolean handleMessage(Message msg);
}
```

我们一起来看一下 Callback的源码，Handler.Callback的定义如下：

```
public interface Callback {
    /**
     * @param msg A {@link android.os.Message Message} object
     * @return True if no further handling is desired
     */
    boolean handleMessage(@NonNull Message msg);
}
```

`Handler.Callback#handleMessage`在处理Message后需要返回boolean类型的结果，而此返回值如果为true，那么在`Handler#dispatchMessage`中会直接return:

```
if (mCallback.handleMessage(msg)) {
    return;
}
```

因此`Handler#handleMessage`则不会执行。

10.12 Handler的sendMessage和postDelay的区别？ (字节跳动)

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么?

1. 是否了解Handler的运行机制?

考察的知识点

1. Handler的内部原理

考生应该如何回答

Handler消息机制原理查看**《8.6 Handler怎么进行线程通信，原理是什么？》**。

1. 首先我们看一下Handler的相关源码

```
public final boolean sendMessage(@NonNull Message msg) {
        return sendMessageDelayed(msg, 0);
}

public final boolean postDelayed(@NonNull Runnable r, long delayMillis) {
        return sendMessageDelayed(getPostMessage(r), delayMillis);
}

private static Message getPostMessage(Runnable r) {
        Message m = Message.obtain();
        m.callback = r;
        return m;
}
```

总的来说，从上面代码我们可以看出，postDelayed和sendMessage方法都是使用到了sendMessageDelayed函数，本质上没多大差别。postDelayed使用到了getPostMessage函数，将Runnable转化为Message，然后相较sendMessage多了一个延时操作。

10.13 Looper.loop会不会阻塞主线程?

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么?

这道题想考察同学对Handler的 Looper.loop 机制的理解。

考生应该如何回答

答案是肯定的，主线程的loop会阻塞主线程，不过这并不是问题，那么原因是什么呢?

对Handler消息处理流程不熟悉的同学请先阅读上面的题目《Handler怎么进行线程通信，原理是什么》和《Handler如果没有消息处理是阻塞的还是非阻塞的》。这道题的细节涉及线程，先说说 Android 中每个应用所对应的 进程以及线程。

进程

任意一个app运行时前会先创建一个进程，该进程是由 Zygote(孵化器) fork 出来的，用于装载各种 Activity, Service 等组件。由于Google 有意为之，进程对于上层应用来说是完全透明的，App 程序都是运行在 Android Runtime。通常情况一个 App 就运行在一个进程中，但是也有很多例外，比如：在 AndroidManifest.xml 中申明了 Android:process 属性，或使用 native 代码 fork 进程，这些方式可以让一个app拥有多个进程。

线程

线程对应用来说十分常见，比如每次 new Thread().start 都会创建一个新的线程。该线程与App所在进程之间共享资源，从 Linux 角度来讲进程与线程除了是否资源共享外，并不存在本质的区别，都是一个task_struct 结构体，在 CPU 看来进程或线程都是一段可执行的代码，CPU 采用 CFS 调度算法，保证每个task都公平的享有 CPU 时间片用于执行任务。

ActivityThread

ActivityThread 是App的入口，这里你可以看到写Java程序时十分常见的 main 方法，它是整个Java程序的入口。ActivityThread 的 main 方法主要作用是做消息循环，一旦消息循环停止，那么你的程序也就可以退出了。 Android 是事件驱动的，在Loop.loop()中不断接收、处理事件，而Activity的生命周期都由主线程的Loop.loop()来调度，所以主线程Looper的存活周期和App的生命周期基本是一致的。当目前没有事件需要处理的时候，主线程就会阻塞；当子线程向消息队列发送消息，主线程就被唤醒。 ActivityThread 其实不是一个 Thread，就只是一个 final 类而已。我们常说的主线程就是从这个类的 main 方法开始，main 方法十分简单，我们看到里面有 Looper 了。代码如下所示：

```
public static void main(String[] args) {
   ...
    Looper.prepareMainLooper();

    ...
    //之前SystemServer调用attach传入的是true，这里到应用进程传入false就行
    ActivityThread thread = new ActivityThread();
    thread.attach(false, startSeq);

    ...
    //一直循环，保障进程一直执行，如果退出，说明程序关闭
    Looper.loop();

    throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

上面的核心代码就是准备主线程的Looper，并且运行Looper#loop（）。

大家可以参考题目《Handler如果没有消息处理是阻塞的还是非阻塞的》，既然主线程的looper和其他的线程的looper是一样的，如果没有消息，那么loop里面就会被 messageQueue 阻塞。只不过因为我们写的代码就是通过handler驱动起来的，我们activity的onCreate、onResume、onStop等等这些生命周期方法，包括我们的UI绘制的信号，这些UI绘制的事件都是通过Handler Looper循环内部发起的。这些所有的事务都被封装成为了一个一个的message，然后通过looper来调用handleMessage回调我们的各Activity和 各Fragment，然后执行这些组件里面的各个生命周期方法，所以我们的代码就是在循环里面执行的，也就是主线程一切皆Message。

ActivityThread 的 main 方法主要作用就是做消息循环，一旦退出消息循环，那么你的程序也就可以退出了。 从消息队列中取消息可能会引起阻塞，取到消息会做出相应的处理。如果某个消息执行时间过长，就可能会影响 UI 线程的刷新效率，造成卡顿的现象。

死循环问题

那么死循环会不会导致我们的CPU在我们没消息的时候在那里空转呢？实际上如果你没有什么事情可做，那么MessageQueue里面有一个nativePollOnce的这个方法，这个方法是个native方法，本地方法调用这个方法其实就是通过加你调用我们在Linux里面的一个管道机制epoll。这时候调用完epoll_wait之后，没有消息的时候，它就会等待在那里，实际上调用完wait之后，就会释放我们的CPU，就等于我们的应用在休眠状态，它不会让CPU一直在那里空转。

10.14 Looper无限循环的阻塞为啥没有ANR

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么?

1. 是否了解Looper的运行机制?

考察的知识点

1. Handler的内部原理
2. ANR发生的原因

考生应该如何回答

1. **首先回答ANR是什么？引起ANR的主要原因有哪些?**

- ANR(Application Not Responding)是应用无响应。ndroid系统对于一些事件需要在一定的时间范围内完成，如果超过预定时间能未能得到有效响应或者响应时间过长，都会造成ANR。
- 发生ANR的主要四种情况：

  1）Service Timeout：前台服务在20s内未执行完成； 2）BroadcastQueue Timeout：前台广播在10s内未执行完成 3）ContentProvider Timeout：内容提供者在publish过超时10s; 4）InputDispatching Timeout：输入事件分发超时5s，包括按键和触摸事件。

  对于Service、Broadcast、Provider组件类的ANR而言，如果把发生ANR比作是引爆炸弹，那么整个流程包含三部分组成：

  埋炸弹：中控系统(system_server进程)启动倒计时埋下定时器，在规定时间内如果目标(Servcie、Broadcast、Provider)没有干完所有的活，则中控系统会定向炸毁(杀进程)目标，就相当于埋下一个定时炸弹。 拆炸弹：在规定的时间内干完工地的所有活，并及时向中控系统报告完成，请求解除定时炸弹，则幸免于难。 引爆炸弹：中控系统立即封装现场，抓取快照，搜集目标执行慢的罪证(traces)，便于后续调试分析，最后是炸毁目标。

  对于输入超时，与其他3个组件类ANR是不同的，Input类型的超时机制并非时间到了一定就会爆炸，而是处理后续上报事件的过程才会去检测是否该爆炸，所以更像是扫雷过程。具体的逻辑是这样的：对于输入系统而言，即使某次事件执行时间超过预期的时长，只要用户后续没有再生成输入事件，那么也不需要ANR。而只有当新一轮的输入事件到来，此时正在分发事件的窗口（即App应用本身）迟迟无法释放资源给新的事件去分发，这时InputDispatcher才会根据超时时间，动态的判断是否需要向对应的窗口提示ANR信息。

  那么明白了ANR的原因后，我们再来看一下Looper的阻塞原理。

Looper无限循环如何导致阻塞的

- Looper无限循环是Looper不停取MessageQueen中的Message并执行这个message的一种机制。我们的APP中的事件，如Activity的生命周期切换、点击、长按、滑动、都是依赖这种机制。
- 如果主线程的MessageQueue中没有消息，便会阻塞在Loop的queue.next()中的nativePollOnce方法。这个时候主线程会进入休眠状态并释放CPU资源，如果下一个消息到达或者有事物发生，通过向pipe管道写入数据来进行唤醒主线程工作。

Looper无限循环为啥没有ANR?

- Looper循环的阻塞是在消息队列无消息需要处理时的一种机制，这种机制就是让CPU停下来去做别的事。而且消息队列无消息，那么就是需要需要让cpu停下来，避免cpu空转，这个机制和ANR是没有关系的，完全不是同一个事，所以自然不会导致ANR

10.15 Looper如何在子线程中创建？（字节跳动、小米）

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么?

这道题想考察同学对 Looper 的理解。

考生应该如何回答

创建子线程的looper必须要通过Looper.prepare()初始化looper,然后再通过Looper.loop()方法让Loop运行起来。

那么具体的细节请看下面的说明：

Handler消息处理流程前面已经介绍过了，还不清楚的朋友，一定要先看下前面章节《Handler怎么进行线程通信，原理是什么》。 首先我们要知道Looper相关的几个重要方法：

Looper.prepare()：Looper 初始化, 同时会初始化MessageQueue， Looper消息机制必须要初始化Looper。

Looper.myLooper()： 获取当前调用线程中ThreadLocal缓存的Looper对象。

Looper.loop()：让Loop进入死循环。

getLooper()： Handler中的方法, 获取Handler中缓存的Looper对象。

Looper.quit(): 终止 Looper.looper() 死循环, 执行 quit后Handler机制将失效, 执行时如果MessageQueue中还有Message未执行, 将不会执行未执行Message, 直接退出, 调用quit后将不能发消息给Handler。

Looper.quitSafely(): 作用同 quit(), 但终止Looper之前会先执行完消息队列中所有非延时任务, 调用quit后将不能发消息给Handler。

子线程中使用Looper如下：

```
Handler mHandler;
new Thread(new Runnable() {
    @Override
    public void run() {
        Looper.prepare();//Looper初始化
        //Handler初始化 需要注意，Handler初始化传入Looper对象是子线程中缓存的Looper对象
        mHandler = new Handler(Looper.myLooper());
        Looper.loop();//死循环
        //注意：Looper.loop()之后的位置代码在Looper退出之前不会执行,(并非永远不执行)
    }
}).start();
```

其中主要需要注意的就是Handler的初始化，需要传入当前线程的 Looper 对象。然而，一般情况下我们并不会这样去给子线程创建looper，因为这样的方式只能创建一个handler对象。我们最正确的创建子线程 Looper的方法一般是调用HandlerThread。

10.16 Looper、handler、线程间的关系。例如一个线程可以有几个Looper可以对应几个Handler?

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么?

这道题想考察同学对 线程中Looper，Handler 的理解。

考生应该如何回答

一个线程可以只能创建一个Looper，但是可以创建任意多个handler 对象。

具体的实现原理细节如下：

Handler消息处理流程如果不熟悉的同学请先阅读《Handler怎么进行线程通信，原理是什么？》

Looper相关

Looper的创建是通过在线程中执行Looper.prepare()方法创建，那么这个方法到底做了什么呢？请看下面的代码：

```
public static void prepare() {
    prepare(true);
}

private static void prepare(boolean quitAllowed) {
    if (sThreadLocal.get() != null) { //code1
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    sThreadLocal.set(new Looper(quitAllowed));
}
```

其中关键性的一句，就是sThreadLocal.set(new Looper(quitAllowed))，那我们来看看sThreadLocal。

```
static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();
```

ThreadLocal：代表了一个线程局部的变量，每个线程中的值都是独立存在、互不影响。在这里ThreadLocal是保证了每个线程都有各自的Looper。而且通过code1 我们知道，一旦sThreadLocal有值，那么再次prepare的时候就会报错，这就保障了每个线程只能有一个Looper可以被创建。

接下来看看建立Looper实例的方法new Looper(quitAllowed)：

```
private Looper(boolean quitAllowed) {
    mQueue = new MessageQueue(quitAllowed);
    mThread = Thread.currentThread();
}
```

即一个Looper中有一个 MessageQueue。

Handler相关

当前线程创建了Looper之后，就可以创建Handler用来处理消息，Handler是怎么跟Looper关联上的呢？请看下面的代码：

```
public Handler(@Nullable Callback callback, boolean async) {
    mLooper = Looper.myLooper();
    if (mLooper == null) {
        throw new RuntimeException(
            "Can't create handler inside thread " + Thread.currentThread()
                    + " that has not called Looper.prepare()");
    }
    mQueue = mLooper.mQueue;
    mCallback = callback;
    mAsynchronous = async;
}
```

在Handler中有两个全局变量mLooper(当前Handler关联Looper)和mQueue(消息队列)，并在构造函数中进行了初始化，重要的就是调用了：Looper.myLooper()：

```
public static @Nullable Looper myLooper() {
    return sThreadLocal.get();
}
```

从上面代码可以知道handler通过调用线程局部变量sThreadLocal，获取当前线程的Looper，这里需要注意的是，如果当前线程没有关联的Looper，这个方法会返回null。 注意：Handler在哪个线程创建的，就跟哪个线程的Looper关联，也可以在Handler的构造方法中传入指定的Looper。

总结

一个线程 只能有一个 Looper，一个MessageQueue，可以有无数个 Handler。

10.17 子线程发消息到主线程进行更新 UI， 除了 handler 和 AsyncTask， 还有什么

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么?

这道题想考察同学对 如何更新UI 的理解。

考生应该如何回答

android给我们提供了一些接口用于在异步线程中更新UI，比如 AsyncTask(),runOnUiThread(),View.post()方法等等,但是这些方法的底层都是调用handler来完成。具体的细节如下：

方式一： 在子线程中可以使用Activity的runOnUiThread方法更新

```
new Thread(){
    public void run(){
        runOnUiThread(new Runnable(){
            @Override
            public void run(){
                //更新UI
            }
        });
    }
}
```

其实原理是Handler的post方法，其内部代码如下：

```
public final void runOnUiThread(Runnable action) {
    if (Thread.currentThread() != mUiThread) {
        mHandler.post(action);
    } else {
        action.run();
    }
}
```

先判断当前的线程是不是主线程，如果是，则直接运行，如果不是，调用post方法，所以最终仍旧是使用handler来进行的处理。

方式二： 用View.post()方法更新

```
imageView.post(new Runnable(){
    @Override
    public void run(){
        // 更新UI
    }
});
```

View中的post源码如下：

```
public boolean post(Runnable action) {
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        return attachInfo.mHandler.post(action);
    }

    // Postpone the runnable until we know on which thread it needs to run.
    // Assume that the runnable will be successfully placed after attach.
    getRunQueue().post(action);
    return true;
}
```

```
public class HandlerActionQueue {
    private HandlerAction[] mActions;
    private int mCount;

    public void post(Runnable action) {
        postDelayed(action, 0);
    }

    public void postDelayed(Runnable action, long delayMillis) {
        final HandlerAction handlerAction = new HandlerAction(action, delayMillis);

        synchronized (this) {
            if (mActions == null) {
                mActions = new HandlerAction[4];
            }
            mActions = GrowingArrayUtils.append(mActions, mCount, handlerAction);
            mCount++;
```

```
        }
    }
```

所以View自己内部也是有自己的异步处理机制，从上面代码就可以看出，调用的是HandlerActionQueue 的post方法，而在Handler内部调用post的时候，先执行的是sendMessageDelayed方法，然后执行sendMessageAtTime方法，紧接着执行enqueueMessage，最终执行的是queue.enqueueMessage，最终执行的方式都是一样的。

总结：无论是哪种UI更新方式，其核心内容依旧还是Handler机制。

10.18 IdleHandler是什么？怎么使用，能解决什么问题？

详细讲解

享学课堂移动开发课程：Framework专题 handler分析部分

这道题想考察什么？

这道题想考察同学对Handler的 IdleHandler 的理解。

考生应该如何回答

IdleHandler 是 MessageQueue内定义的一个接口，一般可用于做性能优化。当消息队列内没有需要立即执行的 message时，会主动触发 IdleHandler 的 queueIdle方法。返回值为 false，即只会执行一次；返回值为 true，即每次当消息队列内没有需要立即执行的消息时，都会触发该方法。具体的细节说明如下：

IdleHandler 是 闲时Handler 机制，可以在 Looper 事件循环的过程中，当出现空闲的时候，允许我们执行任务。IdleHandler 被定义在 MessageQueue 中，它是一个接口。

```
// MessageQueue.java
public static interface IdleHandler {
    boolean queueIdle();
}
```

可以看出，定义时需要实现其 queueIdle() 方法。同时返回值为 true 表示重复使用 IdleHandler，返回 false 表示是一个一次性的 。 既然 IdleHandler 被定义在 MessageQueue 中，使用它也需要借用 MessageQueue。在 MessageQueue 中申明了对应的 add 和 remove 方法。

```
// MessageQueue.java
public void addIdleHandler(@NonNull IdleHandler handler) {
    // ...
    synchronized (this) {
        mIdleHandlers.add(handler);
    }
}

public void removeIdleHandler(@NonNull IdleHandler handler) {
    synchronized (this) {
```

```
        mIdleHandlers.remove(handler);
    }
}
```

可以看到 add 、 remove 其实操作的都是 mIdleHandlers，它的类型是一个 ArrayList。 既然 IdleHandler 主要是在 MessageQueue 出现空闲的时候被执行，那么什么时候出现空闲？ MessageQueue 是一个基于消息触发时间的优先级队列，队列出现空闲存在两种情况。

1. 其一MessageQueue 为空，没有消息；
2. 其二MessageQueue 中最近需要处理的消息，是一个延迟消息（when>currentTime），需要滞后执行；

这两个场景，都会尝试调用 IdleHandler。 处理 IdleHandler 的情况，就在 Message.next() 这个获取消息队列下一个待执行消息的方法中，我们看下具体的逻辑。

```
Message next() {
    // ...
    int pendingIdleHandlerCount = -1;
    int nextPollTimeoutMillis = 0;
    for (;;) {
        nativePollOnce(ptr, nextPollTimeoutMillis);

        synchronized (this) {
            // ...
            if (msg != null) {
                if (now < msg.when) {
                    // 得出休眠的时间
                    nextPollTimeoutMillis = (int) Math.min(msg.when - now,
Integer.MAX_VALUE);
                } else {
                    // Other code
                    // 寻找消息处理后返回
                    return msg;
                }
            } else {
                // 没有更多的消息
                nextPollTimeoutMillis = -1;
            }

           if (pendingIdleHandlerCount < 0
               && (mMessages == null || now < mMessages.when)) {
               pendingIdleHandlerCount = mIdleHandlers.size();
            }
            if (pendingIdleHandlerCount <= 0) {
                mBlocked = true;
                continue;
            }

            if (mPendingIdleHandlers == null) {
                mPendingIdleHandlers = new IdleHandler[Math.max(pendingIdleHandlerCount,
4)];
            }
            mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);
```

```
        }

        for (int i = 0; i < pendingIdleHandlerCount; i++) {
            final IdleHandler idler = mPendingIdleHandlers[i];
            mPendingIdleHandlers[i] = null;

            boolean keep = false;
            try {
                keep = idler.queueIdle();
            } catch (Throwable t) {
                Log.wtf(TAG, "IdleHandler threw exception", t);
            }

            if (!keep) {
                synchronized (this) {
                    mIdleHandlers.remove(idler);
                }
            }
        }

        pendingIdleHandlerCount = 0;
        nextPollTimeoutMillis = 0;
    }
}
```

我们先了解一下 next() 中关于 IdleHandler 执行的主要逻辑:

1. 准备调用 IdleHandler 时，说明当前待调用的消息为 null，或者这条消息的执行时间没有到;
2. 当 pendingIdleHandlerCount < 0 时，根据 mIdleHandlers.size() 赋值给 pendingIdleHandlerCount，它是后面循环的基础;
3. 将 mIdleHandlers 中的 IdleHandler 复制到 mPendingIdleHandlers 数组中，这个数组是临时的，之后进入 for 循环;
4. 循环中从mPendingIdleHandlers数组中取出 IdleHandler，并执行其 queueIdle() 记录返回值存到 keep 中;
5. 当 keep 为 false 时，从 mIdleHandler 中移除当前循环的 IdleHandler，反之则保留;

可以看到 IdleHandler 机制中，最重要的就是在 next() 中，如果messageQueue队列空闲(没有消息需要立刻处理)，那么就循环 mIdleHandler 中的 IdleHandler 对象，如果其 queueIdle() 返回为 false 时，就会从 mIdleHander 中移除。 需要特别注意的是，对 mIdleHandler 这个 List 的所有操作，都是通过 synchronized 来保证线程安全的。

一些关于IdleHandler的细节

IdleHandler 可以是可以保证不会进入死循环。 当队列空闲时，会循环调用一遍 mIdleHandlers 数组并执行 IdleHandler.queueIdle() 函数。而如果数组中有一系列 IdleHander 的 queueIdle() 返回了 true，则会保留在 mIdleHanders 数组中，下次依然会再调用一遍。 注意此时代码逻辑还在 MessageQueue.next() 的循环中，在这个情况下 IdleHandler 机制是如何保证不会进入死循环的? 部分文章会说 IdleHandler 不会死循环，是因为下次循环调用了 nativePollOnce() 借助 epoll 机制进入休眠状态，下次有新消息入队的时候会被重新唤醒，但这是显然是不对的。 注意看前面 next() 中的代码，在方法的末尾会重置 pendingIdleHandlerCount 和 nextPollTimeoutMillis。

```
Message next() {
    // ...
    int pendingIdleHandlerCount = -1;
```

```
    int nextPollTimeoutMillis = 0;
    for (;;) {
        nativePollOnce(ptr, nextPollTimeoutMillis);
        // ...
        // 循环执行 mIdleHandlers
        // ...
        pendingIdleHandlerCount = 0;
        nextPollTimeoutMillis = 0;
    }
}
```

nextPollTimeoutMillis 确定下次进入 nativePollOnce() 超时的时间，它传递 0 的时候代表不会进入休眠，所以讲 natievPollOnce() 进入休眠所以不会死循环是不正确的。 这很好理解，毕竟 IdleHandler.queueIdle() 执行在主线程，它执行的时间是没办法控制的，那么 MessageQueue 中的消息情况可能会变化，所以需要重新处理一遍。 实际不会引起死循环的关键是在于 pendingIdleHandlerCount，我们看看下面的代码。

```
Message next() {
    // ...
    // Step 1
    int pendingIdleHandlerCount = -1;
    int nextPollTimeoutMillis = 0;
    for (;;) {
        nativePollOnce(ptr, nextPollTimeoutMillis);

        synchronized (this) {
            // ...
            // Step 2
            if (pendingIdleHandlerCount < 0
                && (mMessages == null || now < mMessages.when)) {
                pendingIdleHandlerCount = mIdleHandlers.size();
            }
            // Step 3
            if (pendingIdleHandlerCount <= 0) {
                mBlocked = true;
                continue;
            }
            // ...
        }
        // Step 4
        pendingIdleHandlerCount = 0;
        nextPollTimeoutMillis = 0;
    }
}
```

我们梳理一下：

- Step 1，循环执行前，pendingIdleHandlerCount 的初始值为 -1；
- Step 2，在 pendingIdleHandlerCount<0 时，才会通过 mIdleHandlers.size() 赋值。也就是说只有第一次循环才会改变 pendingIdleHandlerCount 的值；
- Step 3，如果 pendingIdleHandlerCount<=0 时，则循环 continue；
- Step 4，重置 pendingIdleHandlerCount 为 0；

当第一次循环时，pendingIdleHandlerCount<0，会给pendingIdleHandlerCount赋值，然后执行到Step4，此时Step3不会执行，当第二次循环时，pendingIdleHandlerCount 等于 0，在 Step 2 不会改变它的值，那么在 Step 3 中会直接 continus 继续会下一次循环，此时没有机会修改 nextPollTimeoutMillis。 那么 nextPollTimeoutMillis 有两种可能：-1 或者下次唤醒的等待间隔时间，在执行到 nativePollOnce() 时则会进入休眠，等待下一次被唤醒。 下次唤醒的时候，mMessage 必然会有一个等待执行的 Message，则 MessageQueue.next() 返回到 Looper.loop() 的循环中，分发处理这个 Message，之后又是一轮新的 next() 中去循环。

IdleHandler能解决什么问题?

通过上面的分析我们不难发现，IdleHandler是在主线程空闲的时候被执行。因此基于这个点，我们可以将一些相对耗时或者一些影响整个线程运行的事务放到IdleHandler里面处理，譬如当我们需要收到调用GC的时候，GC一般会带来STW问题，于是我们可以将这个动作在IdleHandler里面执行，而android源码确实也是这样进行的。

10.19 Android 系统启动流程

详细讲解

享学课堂移动开发课程：Framework专题 系统启动流程分析部分

启动源码解析：https://wx59a7e2633f445ef7.wx.finezb.com/share/recording/de17eb75126929454782d381645585330849a546b82e51bc79aa56c53c2dc08a

这道题想考察什么?

考察我们对Android 系统启动流程的理解，尤其是考察大家是否掌握Android系统运行机制。

考生应该如何回答

Bootloader开始到启动launcher app的整个流程的回答。

概要分析

从系统角度看，`Android`的启动过程可分为3个大的阶段：

1）Bootloader引导阶段

2）装载和启动Linux内核阶段

3）启动Android系统阶段，当然这个阶段很复杂，大体可以可分为如下阶段：

3.1）启动Init进程

3.2）启动Zygote

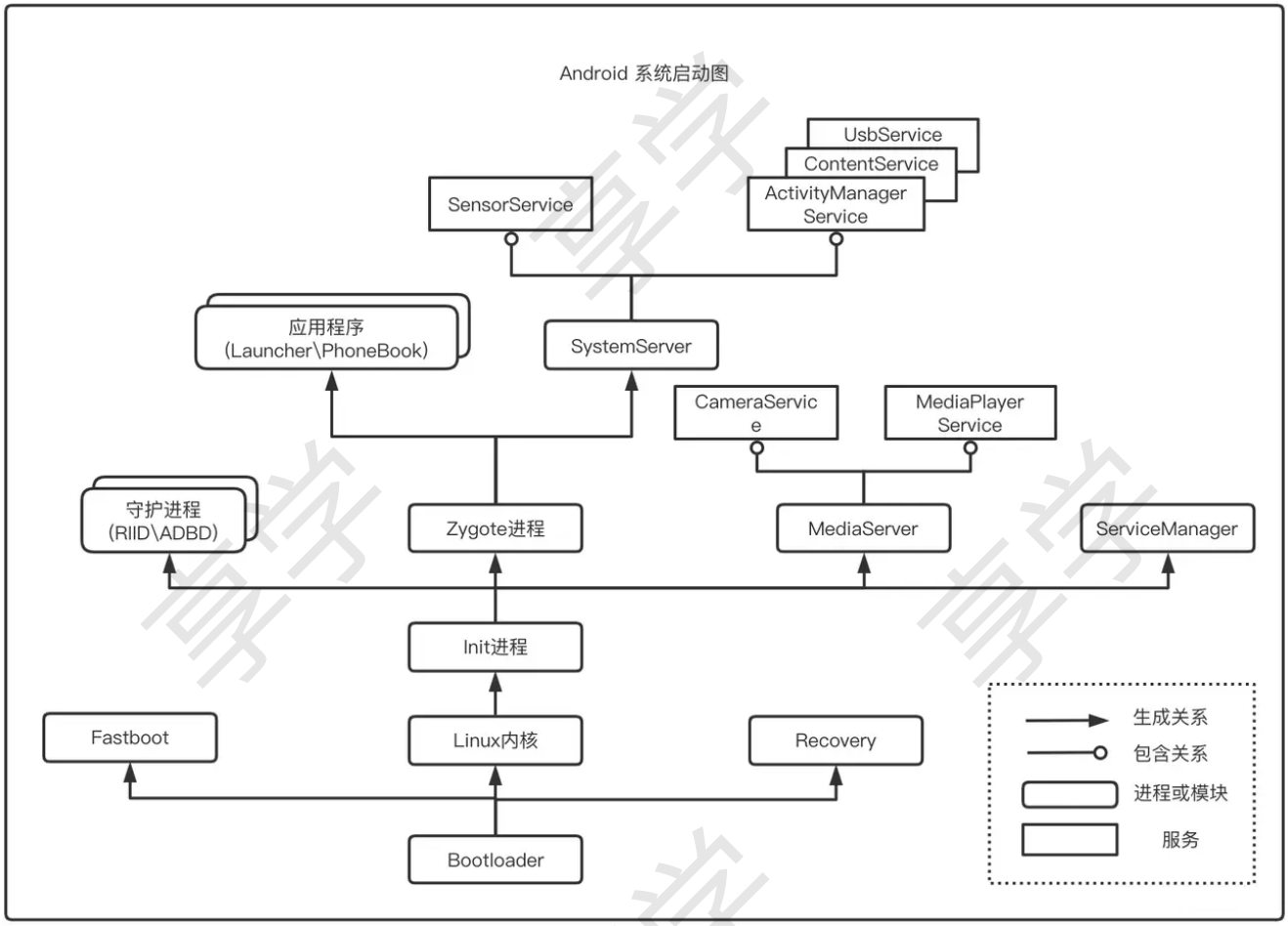
3.3）启动SystemService

3.4）启动SystemServer

3.5）启动Home

等等...

具体的启动，大家可以观察一下下面的流程图，下图展示了手机系统启动的总流程。



下面简单介绍下启动过程：

1. Bootloader引导阶段

当按下电源键开机时，最先运行的就是Bootloader。

Bootloader的主要作用是初始化基本的硬件设备（如 CPU、内存、Flash等）并且建立内存空间映射，为装载Linux内核准备好合适的运行环境。

一旦Linux内核装载完毕，Bootloader将会从内存中清除掉

如果在Bootloader运行期间，按下预定义的的组合键，可以进入系统的更新模块。Android的下载更新可以选择进入Fastboot模式或者Recovery模式:

- Fastboot是Android设计的一套通过USB来更新Android分区映像的协议，方便开发人员快速更新指定分区。
- Recovery是Android特有的升级系统。利用Recovery模式可以进行恢复出厂设置，或者执行OTA、补丁和固件升级。进入Recovery模式实际上是启动了一个文本模式的Linux

2. 装载和启动Linux内核

Android 的 boot.img 存放的就是Linux内核和一个根文件系统

- Bootloader会把boot.img映像装载进内存
- 然后Linux内核会执行整个系统的初始化
- 然后装载根文件系统
- 最后启动Init进程

3.启动Init进程

Linux内核加载完毕后，会首先启动Init进程，Init进程是系统的第一个进程。

Init进程启动过程中，会解析Linux的配置脚本init.rc文件。根据init.rc文件的内容，Init进程会做如下的事物：

- 装载Android的文件系统
- 创建系统目录
- 初始化属性系统
- 启动Android系统重要的守护进程，像USB守护进程、adb守护进程、vold守护进程、rild守护进程等
- 最后，Init进程也会作为守护进程来执行修改属性请求，重启崩溃的进程等操作

4.启动ServiceManager

ServiceManager由Init进程启动。ServiceManager的主要作用是管理Binder服务，负责Binder服务的注册与查找。ServcieManager 是一个守护进程，这个进程会一直存在于后台。当SystemServer启动 AMS，WMS，PMS等服务的时候，会将服务的binder注册到ServiceManager中，由ServcieManager来进行统一的保存。当某个进程如App进程想调用AMS和其他进程通信的时候，就会去ServiceManager中获取AMS的binder，然后通过这个binder来调用AMS的代码进行相关的操作。

5.启动Zygote进程

Init进程初始化结束时，会启动Zygote进程。Zygote进程负责fork出应用进程，是所有应用进程的父进程

- Zygote进程初始化时会创建Android 虚拟机、预装载系统的资源文件和Java类
- 所有从Zygote进程fork出的用户进程都将继承和共享这些预加载的资源，不用浪费时间重新加载，加快的应用程序的启动过程
- 启动结束后，Zygote进程也将变为守护进程，负责响应启动APK的请求。

更多关于zygote启动流程的细节请注意学习后面的zygote相关的章节

6. 启动SystemServer

SystemServer是Zygote进程fork出的第一个进程，也是整个Android系统的核心进程

- SystemServer中运行着Android系统大部分的Binder服务
- SystemServer首先启动本地服务SensorManager
- 接着启动包括ActivityManagerService、WindowsManagerService、PackageManagerService在内的所有Java服务

更多关于SystemServer章节的细节请学习后面的SystemServer相关的章节。

7.启动MediaServer

MediaServer由Init进程启动。它包含了一些多媒体相关的本地Binder服务，包括：CameraService、AudioFlingerService、MediaPlayerService、AudioPolicyService

8.启动Launcher

- SystemServer加载完所有的Java服务后，最后会调用ActivityManagerService的SystemReady()方法
- 在SystemReady()方法中，会发出Intent<android.intent,category.HOME>
- 凡是响应这个Intent的apk都会运行起来，一般Launcher应用才回去响应这个Intent

总结

总的来说，系统启动流程是先通电，然后进入BootLoader引导阶段，然后再启动linux系统，然后再启动Android系统各进程。由底层开始启动，逐步启动上层的过程。

10.20 Zygote进程的启动流程

详细讲解

享学课堂移动开发课程：Framework专题 系统启动流程zygote分析部分

这道题想考察什么?

这道题想考察同学对Zygote进程的了解。

考生应该如何回答

zygote进程是由init进程启动的，init进程通过解读init.rc文件的方式启动了zygote，但是zygote所涉及的内容非常多，我们需要一步一步的分析它的细节。

1.Zygote是什么

Zygote是孵化器，在init进程启动时创建的，Zygote通过fork（复制进程）的方式创建android中几乎所有的应用程序进程和SystemServer进程。另外在Zygote进程启动的时候会创建DVM或者ART虚拟机，因此使用fork而建立的应用程序进程和SystemServer进程可以在内部得到一个DVM或者ART的实例副本，这样就保障了每个app进程在Zygote fork的那一刻就有了虚拟机。

2.Zygote启动脚本

init.rc文件中采用了如下所示的Import类型语句来导入Zygote启动脚本：`import /init.${ro.zygote}.rc` 这里根据属性ro.zygote的内容来导入不同的Zygote启动脚本。从Android 5.0开始，Android开始支持64位程序，Zygote有了32/64位之别，ro.zygote属性的取值有4种：

- init.zygote32.rc
- init.zygote64.rc
- init.zygote64_32.rc

注意：在system/core/rootdir目录中存放上面的Zygote的启动脚本。 上述脚本文件中的部分代码如下：

```
service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server
    class main
```

意思就是，执行 app_process64，该程序的路径是 `/system/bin/app_process64`，类名是 main，进程名为 zygote。

3.Zygote进程启动流程

1.上述init脚本实际执行的是 `/frameworks/base/cmds/app_process/app_main.cpp` 中的 main 方法。 2.启动 zygote 进程在main 方法中会执行。

```
runtime.start("com.android.internal.os.ZygoteInit", args, zygote);
```

3. `runtime.start` 实际执行的是 `/frameworks/base/core/jni/AndroidRuntime.cpp` 的 start 方法

```
void AndroidRuntime::start(const char* className, const Vector<String8>& options, bool
zygote)
```

```
{
    // 1.启动 java 虚拟机
    if (startVm(&mJavaVM, &env, zygote, primary_zygote) != 0) {
        return;
    }
    onVmCreated(env);

    // 2.为Java虚拟机注册JNI方法
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }

    // 3.classNameStr是传入的参数，值为com.android.internall.os.ZygoteInit
    classNameStr = env->NewStringUTF(className);
    // 4.使用toSlashClassName函数将className的 "." 替换为 "/"，得到
com/android/internal/os/ZygoteInit
    char* slashClassName = toSlashClassName(className != NULL ? className : "");
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
    } else {
        // 5.找到 ZygoteInit 中的 main 函数
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
        } else {
            // 6.使用JNI调用ZygoteInit的main函数
            env->CallStaticVoidMethod(startClass, startMeth, strArray);
        }
    }
}
```

通过上面的代码我们不难发现，Zygote在启动过程中分别做了以下几件事：1）启动java虚拟机；2）注册android SDK需要使用的JNI方法；3）执行ZygoteInit的main函数，进而启动zygote的java层代码，最终Zygote就从Native层进入了Java FrameWork层。

特别说明

启动java虚拟机和 注册JNI的过程是zygote进程存在的一个非常关键的原因。正是因为zygote做了这样的事，因此它被用来fork其他进程。

到目前为止，并没有任何代码进入Java FrameWork层面，因此可以认为，Zygote开创了Java FrameWork层。进入java层代码后，main函数的实现如下所示：

```
@UnsupportedAppUsage
    public static void main(String argv[]) {
        // Server socket class for zygote processes.
        ZygoteServer zygoteServer = null; //用来管理和子进程通信的socket服务端

        // Mark zygote start. This ensures that thread creation will throw
        // an error.
```

```
        ZygoteHooks.startZygoteNoThreadCreation();  //这里其实只是设置一个标志位，为创建Java线程时做判断处理，如果是zygote进程，则不需要开启线程

        // Zygote goes into its own process group.
        try {
            Os.setpgid(0, 0);  //为zygote进程设置pgid (Process Group ID)，详见:`https://stackoverflow.com/questions/41498383/what-do-the-identifiers-pid-ppid-sid-pgid-uid-euid-mean`
        } catch (ErrnoException ex) {
            throw new RuntimeException("Failed to setpgid(0,0)", ex);
        }

        Runnable caller;
        try {
            // Report Zygote start time to tron unless it is a runtime restart
            if (!"1".equals(SystemProperties.get("sys.boot_completed"))) { //获取系统属性，判断系统重启完成
                MetricsLogger.histogram(null, "boot_zygote_init",
                        (int) SystemClock.elapsedRealtime());
            }

            String bootTimeTag = Process.is64Bit() ? "Zygote64Timing" : "Zygote32Timing";//判断当前进程是64位程序还是32位程序，并设置标记
            TimingsTraceLog bootTimingsTraceLog = new TimingsTraceLog(bootTimeTag,
                    Trace.TRACE_TAG_DALVIK);
            bootTimingsTraceLog.traceBegin("ZygoteInit");
            RuntimeInit.enableDdms();//注册到ddms服务端，内部调用`DdmServer.registerHandler()`

            boolean startSystemServer = false;
            String zygoteSocketName = "zygote";
            String abiList = null;
            boolean enableLazyPreload = false;
            //对参数进行解析
            for (int i = 1; i < argv.length; i++) {
                if ("start-system-server".equals(argv[i])) { //参数重包含`start-system-server`
                    startSystemServer = true; //设置标志为位true
                } else if ("--enable-lazy-preload".equals(argv[i])) {
                    enableLazyPreload = true;
                } else if (argv[i].startsWith(ABI_LIST_ARG)) { //获取支持的架构列表
                    abiList = argv[i].substring(ABI_LIST_ARG.length());
                } else if (argv[i].startsWith(SOCKET_NAME_ARG)) {
                    zygoteSocketName = argv[i].substring(SOCKET_NAME_ARG.length());
                } else {
                    throw new RuntimeException("Unknown command line argument: " + argv[i]);
                }
            }

            final boolean isPrimaryZygote = zygoteSocketName.equals(Zygote.PRIMARY_SOCKET_NAME);//根据socketName判断是否是primaryZygote，可能还有secondZygote

            if (abiList == null) { //如果支持架构为空，直接抛出异常
```

```
                throw new RuntimeException("No ABI list supplied.");
            }

            // In some configurations, we avoid preloading resources and classes eagerly.
            // In such cases, we will preload things prior to our first fork.
            if (!enableLazyPreload) {
                bootTimingsTraceLog.traceBegin("ZygotePreload");
                EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
                        SystemClock.uptimeMillis());
                preload(bootTimingsTraceLog); //加载各种系统res资源，类资源
                EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
                        SystemClock.uptimeMillis());
                bootTimingsTraceLog.traceEnd(); // ZygotePreload
            } else {
                Zygote.resetNicePriority();
            }

            // Do an initial gc to clean up after startup
            bootTimingsTraceLog.traceBegin("PostZygoteInitGC");
            gcAndFinalize(); //调用ZygoteHooks.gcAndFinalize()进行垃圾回收
            bootTimingsTraceLog.traceEnd(); // PostZygoteInitGC

            bootTimingsTraceLog.traceEnd(); // ZygoteInit
            // Disable tracing so that forked processes do not inherit stale tracing tags
from Zygote.
            Trace.setTracingEnabled(false, 0);


            Zygote.initNativeState(isPrimaryZygote);//jni调用初始化zygote的状态，是否为
isPrimaryZygote

            ZygoteHooks.stopZygoteNoThreadCreation(); //结束zygote创建，其实内部是调用`runtime`给
`zygote_no_threads_`赋值为false,为创建本地线程做准备

            zygoteServer = new ZygoteServer(isPrimaryZygote); //创建zygoteServer，为其他进程初始
化创建时与zygote通信做准备

            if (startSystemServer) { //判断是否需要startSystemServer
                Runnable r = forkSystemServer(abiList, zygoteSocketName, zygoteServer);//通过
fork的方式开启zygote的子进程，systemServer，并返回一个Runnale对象
                // {@code r == null} in the parent (zygote) process, and {@code r != null}
in the
                // child (system_server) process.
                if (r != null) {//如果是zygote进程，则r==null,如果不是zygote进程，也就是
systemServer进程，则执行下面的代码
                    r.run();
                    return;
                }
            }

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

            // The select loop returns early in the child process after a fork and
```

```
            // loops forever in the zygote.
            caller = zygoteServer.runSelectLoop(abiList); //zygote进程进入死循环中，来获取子进程发
送的消息
        } catch (Throwable ex) {
            Log.e(TAG, "System zygote died with exception", ex);
            throw ex;
        } finally {
            if (zygoteServer != null) {
                zygoteServer.closeServerSocket(); //如果发生异常，则说明zygote初始化失败，
zygoteServer也需要关闭
            }
        }

        // We're in the child process and have exited the select loop. Proceed to execute
the command.
        if (caller != null) {
            caller.run();
        }
    }
```

zygote java层的大概流程我们已经梳理完了，现在我们来总结一下：

1. 基于传参解析对应的`zygote.rc`脚本，设置进程名为zygote等信息。
2. 创建ZygoteServer ，这个server存在的目的是让zygote接收来自socket的请求，进而fork进程，zygoteServer里面封装了socket。
3. preload(bootTimingsTraceLog)：加载系统资源res，加载 Android sdk class资源 和其他libc。
4. forkSystemServer，就是fork 产生了SystemServer进程
5. 调用`runSelectionLoop()`，接收其他进程发送的socket消息，进而创建子进程。

特别说明

调用preload完成系统资源的预加载，主要包括preloadClasses，preloadResources，preloadDrawables，preloadSharedLibraries。这些资源并不是给zygote自己实际显示使用的，它之所以存在，是为后面fork 出来的App去运行而准备的。因为app 进程需要运行就需要framework 层的系统资源，而这些资源就会在这里得到加载。

总结

zygote进程的启动分为两大部分，第一部分：执行Native层面的代码，这个过程主要包含：虚拟机启动，JNI资源函数的注册，启动zygote的java层；第二部分：执行java层面的代码，这个过程主要包含：zygote脚本解读，加载公用的各种资源，创建socket服务器并在runSelectionLoop中死循环等待socket消息，fork 了systemServer进程等操作。

10.21 Android中进程的优先级

这道题想考察什么?

这道题想考察同学对进程的理解

考生应该如何回答

总的来说，android系统是基于进程的优先级来决定它们的被回收的顺序。它们的回收顺序从先到后分别是：空进程，后台进程，服务进程，可见进程，前台进程。

前台进程 这个进程是最重要的，是最后被销毁的。前台进程是目前正在屏幕上显示的进程和一些系统进程，也就是和用户正在交互的进程。例如，我正在使用qq跟别人聊天，在我的Android手机上这个进程就应该是前台进程。

可见进程 可见进程指部分程序界面能够被用户看见，却不在前台与用户交互的进程。例如，我们在一个界面上弹出一个对话框（该对话框是一个新的Activity），那么在对话框后面的原界面是可见的，但是并没有与用户进行交互，那么原界面就是可见进程。

服务进程 服务进程是通过 startService() 方法启动的进程，但不属于前台进程和可见进程。例如，在后台播放音乐或者在后台下载就是服务进程。

后台进程 后台进程指的是目前对用户不可见的进程。例如我正在使用qq和别人聊天，这个时候qq是前台进程，但是当我点击Home键让qq界面消失的时候，这个时候它就转换成了后台进程。当内存不够的时候，可能会将后台进程回收。

空进程 空进程指的是在这些进程内部，没有任何东西在运行。保留这种进程的的唯一目的是用作缓存，以缩短该应用下次在其中运行组件所需的启动时间。

现在市面上最难的问题是要提升进程的优先级，然后确保自己的进程能够一直在后台进行存活，但是国内各大厂商对进程的管理进行了很大的改进，就是避免出现进程一直存在于后台的情况出现，所以进程保活一直是一个各app开发者的难点问题。

10.22 SystemServer进程的启动流程

详细讲解

享学课堂移动开发课程：Framework专题 系统启动流程分析部分

这道题想考察什么?

这道题想考察同学对SystemServer进程的了解。

考生应该如何回答

SystemServer进程是由zygote进程启动的，它作为zygote进程的大儿子，它被赋予了重要的职责，启动并管理app运行所需要的绝大多数服务，其中包含：AMS ，ATMS，WMS，PKMS，PMS等等大概有90多个服务，这些服务的启动和管理流程的细节请看后面的分析。

1.SystemServer是做什么的

SystemServer进程主要是用于创建和管理系统服务的，例如AMS、WMS、PMS。由于SystemServer进程和应用进程由Zygote进程启动，所以接下来分析Zygote如何处理的SystemServer进程。

2.Zygote处理SystemServer进程

继上一题《Zygote进程的启动流程》的代码，在zygote#main() 函数中通过调用forkSystemServer()函数 fork SystemServer 进程，那么我们看一下下面的代码：

```
// frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
private static Runnable forkSystemServer(String abiList, String socketName,
            ZygoteServer zygoteServer) {
```

```
        try {
            // 1.创建SystemServer进程，并返回当前进程的pid
            pid = Zygote.forkSystemServer(
                    parsedArgs.mUid, parsedArgs.mGid,
                    parsedArgs.mGids,
                    parsedArgs.mRuntimeFlags,
                    null,
                    parsedArgs.mPermittedCapabilities,
                    parsedArgs.mEffectiveCapabilities);
        } catch (IllegalArgumentException ex) {
            throw new RuntimeException(ex);
        }

        // 2.如果pid==0则说明Zygote进程创建SystemServer进程成功，当前运行在SystemServer进程中
        if (pid == 0) {
            // 3.由于SystemServer进程fork了Zygote进程的地址空间，所以会得到Zygote进程创建的Socket，
            // 而这个Socket对于SystemServer进程是无用的，因此，在此处关闭了该Socket。
            zygoteServer.closeServerSocket();
            // 4.启动SystemServer进程
            return handleSystemServerProcess(parsedArgs);
        }
    }
```

上面的代码执行过程会涉及到进程的fork，在代码1 处，Zygote.forkSystemServer()执行的过程中会创建一个进程，这个进程就是systemServer进程。那么fork完成后，此处的代码会返回两次,因为fork 会返回两次，一次返回是执行fork新创建的代码，另外一次是zygote自己的代码。第一次返回值如果是0 ，说明当前运行的是新的创建的进程的代码也就是systemServer进程的代码，第二次返回的值是一个大于0的值，说明当前运行的代码是zygote进程的代码。因此就有了后面的if判断也就是代码2处，如果 pid 等于0 ，说明当前进程是运行在systemServer进程中。systemServer进程是不需要zygoteServer的，于是就有了 closeServerSocket代码的存在。然后，再执行handleSystemServerProcess，运行systemServer进程。

```
private static Runnable handleSystemServerProcess(ZygoteArguments parsedArgs) {
        if (parsedArgs.mInvokeWith != null) {
            ...
        } else {
            ClassLoader cl = null;
            if (systemServerClasspath != null) {
                // 1.使用了systemServerClassPath和targetSdkVersion创建了一个PathClassLoader
                cl = createPathClassLoader(systemServerClasspath,
parsedArgs.mTargetSdkVersion);
            }

            // 2.调用 ZygoteInit.zygoteInit 方法
            return ZygoteInit.zygoteInit(parsedArgs.mTargetSdkVersion,
                    parsedArgs.mDisabledCompatChanges,
                    parsedArgs.mRemainingArgs, cl);
        }
    }
```

以上代码做了两件事：1）基于sdk版本创建了一个ClassLoader；2）执行zygoteInit方法。

```
public static final Runnable zygoteInit(int targetSdkVersion, long[] disabledCompatChanges,
            String[] argv, ClassLoader classLoader) {
        // 1.进入native方法
        ZygoteInit.nativeZygoteInit();
        // 2.获取封装了SystemServer的main函数的runnable
        return RuntimeInit.applicationInit(targetSdkVersion, disabledCompatChanges, argv,
                classLoader);
    }
```

在zygoteInit函数中，首先是调用nativeZygoteinit函数来初始化进程环境，然后再执行applicationInit函数获取封装了SystemServer的main函数的Runnable。

2-1.ZygoteInit.nativeZygoteInit

在上一题《zygote进程的启动流程》中我们知道有一个流程，那就是在zygote初始化的时候会注册JNI，其中就有将nativeZygoteInit()方法和native函数com_android_internal_os_ZygoteInit_nativeZygoteInit()建立了映射关系。所以此时调用nativeZygoteInit()会调用到AndroidRuntime的下面的代码。

```
static AndroidRuntime* gCurRuntime = NULL;

static void com_android_internal_os_ZygoteInit_nativeZygoteInit(JNIEnv* env, jobject clazz)
{
    gCurRuntime->onZygoteInit();
}
```

gCurRuntime是AndroidRuntime的指针，具体指向的是其子类AppRuntime，它在app_main.cpp中定义。代码如下：

```
// frameworks/base/cmds/app_process/app_main.cpp
virtual void onZygoteInit()
    {
        // 1.创建了一个ProcessState实例
        sp<ProcessState> proc = ProcessState::self();
        // 2.启动了一个Binder线程池
        proc->startThreadPool();
    }
```

在这个代码中执行的是进程的初始化，首先在代码1处，创建了一个进程ProcessState，其中就会创建一个binder；然后在代码2处启动了Binder线程池，为Binder线程的运行提供了动力。

到目前为止SystemServer进程已经创建完成，那么接下来就需要运行systemServer进程了，所以我们回到RuntimeInit.applicationInit函数中进行分析。

2-2.RuntimeInit.applicationInit

```
// frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
protected static Runnable applicationInit(int targetSdkVersion, long[]
disabledCompatChanges,
            String[] argv, ClassLoader classLoader) {
        return findStaticMain(args.startClass, args.startArgs, classLoader);
}
```

```
protected static Runnable findStaticMain(String className, String[] argv,
            ClassLoader classLoader) {
        Class<?> cl;
        try {
            // 1.通过反射得到了SystemServer类
            cl = Class.forName(className, true, classLoader);
        }

        Method m;
        try {
            // 2.找到了 SystemServer中的main方法
            m = cl.getMethod("main", new Class[] { String[].class });
        }

        // 3.将main()方法传入MethodAndArgsCaller()方法中
        return new MethodAndArgsCaller(m, argv);
    }
```

以上代码就是运用反射，拿到systemServer的main 函数，并且调用MethodAndArgsCaller，将main函数的调用封装到一个Runnable中，方便后期的执行。

```
static class MethodAndArgsCaller implements Runnable {
        private final Method mMethod;
        public MethodAndArgsCaller(Method method, String[] args) {
            mMethod = method;
            mArgs = args;
        }

        public void run() {
            try {
                // 执行SystemServer的main()方法
                mMethod.invoke(null, new Object[] { mArgs });
            }
        }
    }
```

这个Runnable的run方法是在ZygoteInit的main()方法中被执行。代码如下：

```
// frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
public static void main(String argv[]) {
    try {
            if (startSystemServer) {
                Runnable r = forkSystemServer(abiList, zygoteSocketName, zygoteServer);
                if (r != null) {
                    r.run();  //1 执行run
                    return;
                }
            }
        } catch (Throwable ex) {
            Log.e(TAG, "System zygote died with exception", ex);
            throw ex;
        } finally {
```

```
            if (zygoteServer != null) {
                zygoteServer.closeServerSocket();
            }
        }
```

如上面代码1处，执行run方法，这个时候就是执行方法main(),所以，接下来我们看systemServer main函数的执行。

3. SystemServer进程执行

```
// frameworks/base/services/java/com/android/server/SystemServer.java
public static void main(String[] args) {
        new SystemServer().run();
}
```

main函数就一行代码执行SystemServer进程的润函数：

```
private void run() {
        try {
            Looper.prepareMainLooper();

            // 1.加载动态库libandroid_servers.so
            System.loadLibrary("android_servers");

            // 2.创建SystemServerManager
            mSystemServiceManager = new SystemServiceManager(mSystemContext);

            // 3.初始化系统上下文
            createSystemContext();

        }

        try {
            // 3.使用SystemServiceManager启动引导服务
            startBootstrapServices(t);
            // 4.启动核心服务
            startCoreServices(t);
            // 5.启动其他服务
            startOtherServices(t);
        }

        // Loop forever.
        Looper.loop();
}
```

通过上面的代码不难发现，启动过程中，在大流程上systemServer和app进程的启动很像，初始化了很多资源，也加载了很多资源，还创建了上下文，同时构建了进程的Looper 死循环确保进程不会退出。当然，在SystemServer中最重要的内容还是通过代码3，4，5启动了非常多的服务，大概有90多个服务，这些服务包括AMS，ATMS，WMS，PKMS，BMS等等一系列的服务。另外，在代码2处创建了一个systemServiceManager对象，这个对象是用于在SystemServer进程中管理90多个服务的启动的。

4.总结

SystemService进程被创建后，主要的处理如下：

1. 启动Binder线程池，这样就可以与其他进程进行Binder跨进程通信。
2. SystemServer在启动过程中，先初始化一些系统变量，加载类库，创建Context对象。
3. 创建SystemServiceManager，它用来对系统服务进行创建、启动和生命周期管理。
4. 启动各种系统服务：引导服务、核心服务、其他服务，共90多种。应用开发主要关注引导服务ActivityManagerService、PackageManagerService和其他服务WindowManagerService、InputManagerService即可。
5. SystemServer在启动服务前，会尝试与Zygote建立Socket通信，通信成功后才去启动服务。
6. 启动的服务都单独运行在SystemServer的各自线程中，同属于SystemServer进程

10.23 AMS启动流程

详细讲解

享学课堂移动开发课程：Framework专题 AMS&SystemServer进程分析部分

启动源码解析：https://wx59a7e2633f445ef7.wx.finezb.com/share/recording/de17eb75126929454782d381645585330849a546b82e51bc79aa56c53c2dc08a

这道题想考察什么?

学员对AMS是什么，它的启动方式，以及AMS的管理方式

考生应该如何回答

先从进程的角度分析AMS，然后再结合系统启动流程讲解AMS的启动过程，最后讲解AMS如何给App提供服务的。

1. AMS与ATMS 是什么

ActivityManagerService简称 AMS，在Android 10 之前，AMS主要负责对四大组件进行管理和调度，同时，AMS也对进程、电池、内存、权限等进行管理。但在Android 10开始，系统发现AMS要承载的事务太多，就将Activity的管理迁移到了ActivityTaskManagerService中，ActivityTaskManagerService也被缩写为ATMS。

AMS与ATMS都是SystemServer进程中的系统服务，他们并不是独立的进程，所以，他们的启动和管理全部都是在SystemServer进程中进行的。

2. ATMS&AMS的启动

由于AMS和ATMS的启动流程基本一致，这里就以ATMS为例，对他们的启动流程进行分析。

在SystemServer进程启动后会执行main函数，main函数里面会执行run方法，代码如下：

```
private void run() {
    //...
    try {
        //1、进入ActivityThread的attch，初始化了Instrumentation、
        //context和调用Application的onCreate
        createSystemContext();
```

```
    }
    try {
        //...
        // 启动各种服务，包括AMS，ATMS等
        startBootstrapServices();
        startCoreServices();
        startOtherServices();
    }
}
```

上面的代码会调用三个函数 startBootstrapServices()、startCoreServices()、startOtherServices()，这三个函数分别用于启动各类系统服务的，其中AMS&ATMS的启动如下面的代码所示：

```
private void startBootstrapServices() {
    //......

    //1、启动ATMS服务
    ActivityTaskManagerService atm = mSystemServiceManager.startService(
            ActivityTaskManagerService.Lifecycle.class).getService();
    //2、启动AMS服务
    mActivityManagerService = ActivityManagerService.Lifecycle.startService(
            mSystemServiceManager, atm);
    //3、将SystemServiceManager设置给AMS
    mActivityManagerService.setSystemServiceManager(mSystemServiceManager);
    //3、将installer设置给AMS
    mActivityManagerService.setInstaller(installer);
    //4、初始化电源管理功能
    mActivityManagerService.initPowerManagement();
    //5、设置系统进程
    mActivityManagerService.setSystemProcess();
    //6、初始化看门狗
    watchdog.init(mSystemContext, mActivityManagerService);
}
```

大家通过代码1 不难发现，ATMS并不是直接启动，而是SystemServiceManager通过 Lifecycle类来间接实现。SystemServiceManager是一个辅助类，它用于辅助SystemServer进程启动和管理SystemServer进程中的各类服务Service。由于需要管理的服务非常多，所以SystemServiceManager是通过管理所有实现了SystemService接口的类的方式来统一的运用反射启动这些服务。Lifecycle 是 ATMS 里的静态内部类，静态内部类不依赖于外部类，它封装了ATMS对象，同时实现了SystemService接口。Lifecycle 构造方法调用时，会初始化内部成员变量 mService，即调用ATMS的构造方法，构造方法中会执行一些初始化操作。

下面我们分析一下SystemServiceManager,通过startService函数启动AMTS的流程。

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

        final T service;
        try {
            Constructor<T> constructor = serviceClass.getConstructor(Context.class);
            service = constructor.newInstance(mContext);
```

```
        } catch (InstantiationException ex) {
        ...
        }
        startService(service);
        return service;
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
    }
}
```

通过上面的代码我们不难发现，startService()方法内部是运用了反射获取到了serviceClass类的对象实例service，然后再调用startService（service）如下所示：

```
public void startService(@NonNull final SystemService service) {
    // Register it.
    mServices.add(service);
    // Start it.
    long time = SystemClock.elapsedRealtime();
    try {
        service.onStart(); //code1 统一执行服务的onStart函数
    }
    ...
}
```

在上面代码中会执行到code1，通过code1会执行到 ATMS的Lifecycle中的 onStart函数，通过这个onStart函数会执行ATMS的启动。那么ATMS启动onStart的流程又是怎样的呢?

具体Lifecycle的代码如下：

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

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

    @Override
    public void onStart() {
        publishBinderService(Context.ACTIVITY_TASK_SERVICE, mService); //code1
        mService.start(); //code2 ATMS start ()
    }

    @Override
    public void onUnlockUser(int userId) {
        synchronized (mService.getGlobalLock()) {
            mService.mStackSupervisor.onUserUnlocked(userId);
        }
    }

    @Override
    public void onCleanupUser(int userId) {
```

```
        synchronized (mService.getGlobalLock()) {
            mService.mStackSupervisor.mLaunchParamsPersister.onCleanupUser(userId);
        }
    }

    public ActivityTaskManagerService getService() {
        return mService;
    }
}
```

首先分析一下code1代码块，publishBinderService函数最终会执行到下面的代码：

```
ServiceManager.addService(name, service, allowIsolated, dumpPriority)
```

这就表面，ATMS将自己的binder 发布到了ServcieManager这个进程上面了，发布上去的目的是方便后期其他进程调用ATMS的服务，具体的逻辑大家可以在Binder相关的章节学会。

其次，分析一下code2，上面代码中onStart()函数执行实则是在执行ATMS的start函数，如下代码所示：

```
private void start() {
    LocalServices.addService(ActivityTaskManagerInternal.class, mInternal);
}

@VisibleForTesting(visibility = VisibleForTesting.Visibility.PACKAGE)
public ActivityTaskManagerService(Context context) {
    //保存先前SystemServer 创建的SystemContext，感觉这个也可以通过mSystemThread 获取
    mContext = context;
    mFactoryTest = FactoryTest.getMode();
    mSystemThread = ActivityThread.currentActivityThread();// 获取 ActivityThread
    mUiContext = mSystemThread.getSystemUiContext();// 获取 mUiContext
    mLifecycleManager = new ClientLifecycleManager();
    mInternal = new LocalService();
    GL_ES_VERSION = SystemProperties.getInt("ro.opengles.version",
                    GL_ES_VERSION_UNDEFINED);
    mWindowOrganizerController = new WindowOrganizerController(this);
    mTaskOrganizerController = mWindowOrganizerController.mTaskOrganizerController;
}
```

实际上，start函数就是在将ATMS的本地服务保存到LocalServices列表中，当然，ATMS的本地服务大家理解为就是ATMS的服务就好了。

小结

AMS&ATMS的启动流程进行小结：

首先，他们都是SystemServer进程中的服务，都是在SystemServer进程启动的时候进行启动；

其次，在他们启动的过程中，并不是直接自己启动的，而是由SystemServer进程通过SystemServiceManager对象进行统一的启动；

然后，ATMS&AMS 启动后会将自己的本地服务公布到LocalServices列表，将来进程内别的服务需要使用他们的时候可以去LocalServices中找到；

最后，ATMS&AMS 会将自己的binder 公布给ServiceManager进程，由这个进程去进程存储，方便后面其他进程获取AMS和ATMS的binder 代理。

3. ATMS与AMS的管理

ATMS与AMS的管理是非常典型的android系统对systemServer进程创建的各类服务管理案例。具体的逻辑我们进行如下的分析：

1）SystemServer进程会启动大概90多项的系统服务，为了能够更好的启动和管理这么多服务，那么SystemServer进程就需要依靠SystemServiceManager 的对象来协助管理。因此各项服务基本都是采用了如下的代码模板进行启动：

```
mSystemServiceManager.startService(ActivityTaskManagerService.Lifecycle.class)
```

具体的启动流程在上面的环节已经介绍。

2）SystemServiceManager并非直接去启动AMS等服务，而是采用启动这些服务的Lifecycle的方式，原因是什么呢？方便管理。其实，读者们你们细细想想，如果要启动一个服务，一般是不是需要先创建这个服务的对象，然后再进行这个对象的服务的启动，如果systemServer直接去操作这些服务，比如会出现非常多的样板代码。因此，systemServer就借助SystemServcieManager 去完成这个庞大的重复工作。当然，为了SystemServcieManager 更加内聚，于是就让每个服务都自动包含一个Lifeycle类，这个类里面会去创建自己的服务的对象，同时会启动服务。而且所有服务的LifeCycle都拓展了SystemService类，这样所有统一的操作都可以在SystemService基类中完成。具体的代码在上面的章节中有比较详细的介绍。

3）当AMS与ATMS等服务启动后，为了方便将服务提供给第三方使用，此时系统进行了下面的管理：

1. 调用`LocalServices.addService()`，将服务存储在SystemServer本地，以便SystemServer进程内部快速的实现服务的调用；
2. 调用`publishBinderService`进而调用`ServiceManager.addService`函数，将服务的IBinder对象公布到ServiceManager进程，以便app进程通过ServiceManager调用这些服务。

4. 总结

AMS是由SystemServer进程通过SystemServiceManager启动并管理的一个对象，它和所有其他被SystemServer进程启动的服务一样并不是独立的进程，它只是一个服务。这个服务启动后会在SystemServer进程本地有一份存储，目的是方便当前进程共享该服务。同时AMS在ServiceManager进程也会进行发布，将自己的IBinder对象存在在SM中，方便其他第三方的进程去获取AMS的服务。

10.24 SystemServer进程为什么要在Zygote中fork启动，而不是在init 进程中直接启动

详细讲解

享学课堂移动开发课程：Framework专题 zygote部分

这道题想考察什么？

考察我们对fork的理解以及对zygote进程的价值的理解

考生应该如何回答

分析什么是fork，然后再分析zygote fork并启动进程和 init启动进程的区别。

fork()机制

一个进程，包括代码、数据和分配给进程的资源。fork（）函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。 一个进程调用fork（）函数后，系统先给新的进程分配资源，例如存储数据和代码的空间，然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同，相当于克隆了一个自己。

所以fork()机制执行后会得到一个新进程，这个过程具有下面的特点：

1）这两个进程代码一致，而且代码执行到的位置也一致；

2）区别是进程ID（PID）不一样；

3）一次调用，两次返回。父进程返回的是子进程的PID，从而让父进程可以跟踪子进程的状态，以子进程PID作为函数参数，子进程返回的是0。

SystemServer进程的孵化和启动为什么在Zygote进程，那么我们就需要知道zygote进程做了哪些事，而这些事情是SystemServer进程需要的，而且在init进程中没有的。

总结

通过上面问题中关于《zygote进程的启动流程》的回答，大家不难发现Zygote 作为一个孵化器，可以提前加载一些资源，这样 fork() 时基于 Copy-On-Write 机制创建的其他进程就能直接使用这些资源，而不用重新加载。比如 system_server 就可以直接使用 Zygote 中的 JNI函数、共享库、常用的类、以及主题资源。然而，init进程缺没有这些资源提供，所以不能用init进程创建SystemServer。

10.25 为什么要专门使用Zygote进程去孵化app进程，而不是让SystemServer去孵化

详细讲解

享学课堂移动开发课程：Framework专题 zygote部分

这道题想考察什么?

考察大家对zygote工作内容的理解，fork的理解，还有systemServer进程的工作内容。

考生应该如何回答

通过上面的题目《Zygote进程的启动流程》&《SystemServer进程的启动流程的分析》的内容学习，大家不难发现zygote和SystemServer进程的工作重点。第一：zygote进程的运行主要做以下几件事：虚拟机初始化与启动，JNI函数的注册，加载公用的各种资源，创建socket服务器并在runSelectionLoop中死循环等待socket消息，fork 了systemServer进程等操作；第二：SystemServer进程作为Zygote进程的大儿子，主要工作主要是启动和管理了引导服务、核心服务、其他服务，等共90多种服务，这些服务是专门提供给app进程使用的。

那么一个app的运行需要什么：1）每个app必须有虚拟机；2）必须能够调用framework中的资源和函数，比如要使用ImageView,那么这个类谁来加载？Zygote帮忙加载了；3）需要AMS WMS等服务给app运行和管理提供支持。

那么，在App出生的时候，如何让它天生的具备虚拟机和 各种android的资源和函数的引用呢？那就是通过fork zygote得到。Zygote做的事情就是加载资源加载通用的库，创建虚拟机等操作，当从zygote fork产生进程后，这个进程就自然的具有了和zygote一模一样的所有的资源和代码了，那么这个时候就不需要app再去自己创建虚拟机自己加载各种资源了，因此app进程的fork是通过Zygote进程。

SystemServer所创建和启动的各种Service，会将这些服务的binder发送给ServiceManager进程进程统一的管理，我们app可以通过ServcieManager拿到这些服务的binder，然后通过binder来访问这些服务。

10.26 Zygote 为什么不采用Binder机制进行IPC通信呢?

详细讲解

享学课堂移动开发课程：Framework专题 zygote&binder部分

这道题想考察什么?

考察学员对binder的理解和zygote fork的理解

考生应该如何回答

这个很重要的原因是如果zygote采用binder 会导致 fork出来的进程产生死锁。

在UNIX上有一个 程序设计的准则：多线程程序里不准使用fork。

这个规则的原因是：如果程序支持多线程，在程序进行fork的时候，就可能引起各种问题，最典型的问题就是，fork出来的子进程会复制父进程的所有内容，包括父进程的所有线程状态。那么父进程中如果有子线程正在处于等锁的状态的话，那么这个状态也会被复制到子进程中。父进程的中线程锁会有对应的线程来进行释放锁和解锁，但是子进程中的锁就等不到对应的线程来解锁了，所以为了避免这种子进程出现等锁的可能的风险，UNIX就有了不建议在多线程程序中使用fork的规则。

在Android系统中，Binder是支持多线程的，Binder线程池有可以有多个线程运行，那么binder 中就自然会有出现子线程处于等锁的状态。那么如果Zygote是使用的binder进程 IPC机制，那么Zygote中将有可能出现等锁的状态，此时，一旦通过zygote的fork去创建子进程，那么子进程将继承Zygote的等锁状态。这就会出现子进程一创建，天生的就在等待线程锁，而这个锁缺没有地方去帮它释放，子进程一直处于等待锁的状态。

10.27 Android app进程是怎么启动的?

详细讲解

享学课堂移动开发课程：Framework专题 AMS部分

这道题想考察什么?

app进程的创建和启动流程。

考生应该如何回答

分析app如何被fork出来，然后，讲解app启动过程中的重要细节，我们首先分析app启动的方案，然后再重点讲解相关的启动流程。

什么是冷启动和热启动

1. **冷启动** 当启动应用时，后台没有该应用的进程，这时系统会重新创建一个新的进程分配给该应用，这个启动方式就是冷启动，也就是先实例化Application。
2. **热启动** 当启动应用时，后台已有该应用的进程（例：按back键、home键，应用虽然会退出，但是该应用的进程是依然会保留在后台，可进入任务列表查看），所以在已有进程的情况下，这种启动会从已有的进程中来启动应用，也就是直接从进程中启动,不需要重新创建Application，这个方式叫热启动。

由于热启动是从已有的后台进程中启动，所以热启动并不会走Application这步，它一般是直接走MainActivity，所以热启动过程只需要创建和初始化一个MainActivity即可，不必创建和初始化Application。

冷启动的启动流程分析

1）启动APP进程

当我们点击Launcher桌面程序的APP图标时，Launcher程序会调用startActivity()函数，通过Binder跨进程通信，发送消息给system_server进程中的AMS。在system_server进程中，由AMS通过socket通信告知Zygote进程fork出一个子进程(APP进程)。

对上面流程中需要进行几点说明：1） AMS不是一个独立的进程，它只是system_server进程的一个服务；2）Launcher进程要启动另外一个app就必须发消息给AMS，由AMS发送App进程创建的命令；3）Launcher是通过去ServiceManager进程中获取AMS的Binder的方案得到AMS的Binder，然后通过这个Binder调用AMS的服务；4）当AMS收到Launcher启动进程的事件后，会在AMS体系中进行各种检测和判断，最终发现被启动的App进程还未创建；5）此时AMS会给Zygote发送一个 socket消息，而zygote中的socket服务器会接收到这个消息，然后会执行Zygote 的fork，并产生一个子进程，这个子进程就是APP进程。

具体流程如下图所示：


activity启动流程

1）launcher进程不能直接去启动app进程，所以它会调用SM拿到ATMS的代理对象ActivityTaskManagerProxy，然后执行ActivityTaskManagerProxy的startActivity方法。

2）ActivityTaskManager收到执行startActivity方法的请求后会执行将请求转交给AMS，然后AMS会去判断当前Activity所对于的Application进程是否存在，如果进程存在，那么就是热启动，如果进程不存在那么就是冷启动。在冷启动的时候，AMS将发送socket请求给到Zygote进程，通知zygote去创建app进程。

3）Zygote进程收到socket请求就会去执行fork函数，fork会产生一个子进程，这个进程就是需要启动的app进程了。这个子进程同时就具备了Zygote进程的一切，这个 说明在zygote相关的内容中有详细介绍。

4）开启APP主线程

APP进程启动后，会实例化一个ActivityThread，并在native层面通过反射执行其main函数，同时触发执行attach函数，同时会创建ApplicationThread、Looper、Handler对象，开启主线程消息循环Looper.loop()。

对于上面流程的需要进行几点说明：

1）App进程创建后，Zygote会执行反射相关的代码，通过反射执行ActivityThread的main函数，此时代码进入ActivityThread类，并完成了ActivityThread的初始化。

2）ActivityThread执行main函数的代码如下所示：

```
public static void main(String[] args) {

    ...
    //核心代码1
    Looper.prepareMainLooper();

    ...

    //核心代码2
    ActivityThread thread = new ActivityThread();
    thread.attach(false, startSeq);

    ...
    //核心代码3
    Looper.loop();

    ...
}
```

首先，创建了主线程的Looper，并且开启了主线程的Looper.loop(),对线程开始执行消息循环。这样的设计可以让android所有的主线程的事务都封装成为Message，然后通过MessageQueue的循环来完成消息轮询和执行。

其次，核心代码2中创建了ActivityThread对象，同时调用了thread.attach()函数，这个函数会获取AMS服务的binder，然后调用AMS的attachApplication()方法，这个方法会将App的 IApplicationThread 对象传递给AMS，AMS这时候就持有了App的Binder对象，当AMS需要向App进行通信的时候就能够非常的便利了。

5）创建并初始化Application和Activity

ActivityThread的main函数通过调用attach方法进行 Binder 通信，通知system_server进程执行AMS的attachApplication方法。在attachApplication方法中，AMS分别通过bindApplication、scheduleLaunchActivity方法，并且将这些操作封装成Message并通知APP进程的主线程Handler，对APP进程的Application和Activity进行初始化，并执行Application、Activity的生命周期。最后在Activity的生命周期中执行UI的创建和显示。

至此，应用启动流程完成，这里对源码细节不做过多分析，感兴趣的同学可以学习后面章节中《Activity启动流程分享》的内容，或者大家可以自行阅读android 11 源码。

10.28 Android Application为什么是单例

这道题想考察什么?

这道题想考察同学对 Application 的创建这块的源码是否熟悉。

考生应该如何回答

从源码层面来说下application凭什么是单例，我们查看下Application 对象的创建过程，找出它为什么是单例的原因。 所以我们直接来看Application是怎么一步步创建的。

handleBindApplication

```
// ActivityThread.java
private void handleBindApplication(AppBindData data) {
    Application app;

    // 创建 Application 对象的代码
    app = data.info.makeApplication(data.restrictedBackupMode, null);
}
```

makeApplication

```
// LoadedApk.java
public Application makeApplication(boolean forceDefaultAppClass,
            Instrumentation instrumentation) {
    // 如果 mApplication 不等于空，则直接返回
    if (mApplication != null) {
        return mApplication;
    }

    // 创建 Application对象通过反射
    app = mActivityThread.mInstrumentation.newApplication(
                    cl, appClass, appContext);
}
```

从上面代码可知，当 Application为空的时候，才会创建，从而保证了它是一个单例对象。这道题相对简单，就是考察大家是否了解application的创建流程。

Application是一个单例，即每一个app启动的时候都会创建一个Application的实例，它用来存储和处理整个全局变量的一些事情，它同时还是一个Context，Application也像Activity一样有自己的生命周期，但是它并没有那么复杂，因为Application只在打开app，确切的说是启动app的时候才会创建，所以说可以在application创建的时候对app全局进行一些配置，比如配置插件

10.29 Intent的原理，作用，可以传递哪些类型的参数?

这道题想考察什么?

这道题想考察同学对 Intent 的理解。

考生应该如何回答

Android中的Intent是一个非常重要且常用的类，可以用来在一个组件中启动App中的另一个组件或者是启动另一个App的组件，这里所说的组件指的是Activity、Service以及Broadcast。

Intent中文意思指"意图"，按照Android的设计理念,Android使用Intent来封装程序的"调用意图",不管启动Activity、Service、BroadcastReceiver,Android都使用统一的Intent对象来封装这一"启动意图"。

那么如何实现Intent的这个功能了，我们一起来分析一下intent的原理。

Intent的原理

在手机系统启动的时候，PMS会扫描所有已安装的apk目录，解析apk包中的AndroidManifest.xml文件得到App的相关信息，并且将所有的这些信息存储起来构建一个完整的apk的信息树。在Intent去进行各组件间通信的时候，会调用PMS去查找apk信息表，找到相关的组件进行类似于启动的操作。

Intent原理分为查找和匹配，

1.查找

Android使用Intent组件，用于进程之间的通信和跳转。Intent具有隐式、显式两种。我们知道Android系统通过PackageManagerService来进行系统组件的维护。系统启动之后会有各种系统服务的注册，其中就含有PackageManagerService。在启动之后，PMS会扫描所有已安装的apk目录，解析apk包中的AndroidManifest.xml文件得到App的相关信息，而每个AndroidManifest.xml清单文件又包含了Activity，Service等组件的声明信息，当PMS扫描并且解析完信息后，就清晰地绘制出了整棵apk的信息树。 PackageManagerService的构造函数加载了系统已安装的各种apk，并加载了Framework资源内容和核心库。加载了资源和核心库之后才开始对扫描的指定目录下的apk文件进行分析，然后里面的PackageParser的parsePackage方法进行文件分析。

```
private Package parseBaseApk(String apkPath, Resources res, XmlResourceParser parser, int
flags,
        String[] outError) throws XmlPullParserException, IOException {
    final String splitName;
    final String pkgName;

    try {
        Pair<String, String> packageSplit = parsePackageSplitNames(parser, parser);
        pkgName = packageSplit.first;
        splitName = packageSplit.second;

        if (!TextUtils.isEmpty(splitName)) {
            outError[0] = "Expected base APK, but found split " + splitName;
            mParseError = PackageManager.INSTALL_PARSE_FAILED_BAD_PACKAGE_NAME;
            return null;
        }
    } catch (PackageParserException e) {
        mParseError = PackageManager.INSTALL_PARSE_FAILED_BAD_PACKAGE_NAME;
        return null;
    }

    final Package pkg = new Package(pkgName);

    TypedArray sa = res.obtainAttributes(parser,
            com.android.internal.R.styleable.AndroidManifest);

    pkg.mVersionCode = sa.getInteger(
            com.android.internal.R.styleable.AndroidManifest_versionCode, 0);
    pkg.mVersionCodeMajor = sa.getInteger(
            com.android.internal.R.styleable.AndroidManifest_versionCodeMajor, 0);
    pkg.applicationInfo.setVersionCode(pkg.getLongVersionCode());
    pkg.baseRevisionCode = sa.getInteger(
            com.android.internal.R.styleable.AndroidManifest_revisionCode, 0);
    pkg.mVersionName = sa.getNonConfigurationString(
            com.android.internal.R.styleable.AndroidManifest_versionName, 0);
```

```
    if (pkg.mVersionName != null) {
        pkg.mVersionName = pkg.mVersionName.intern();
    }

    pkg.coreApp = parser.getAttributeBooleanValue(null, "coreApp", false);

    final boolean isolatedSplits = sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifest_isolatedSplits, false);
    if (isolatedSplits) {
        pkg.applicationInfo.privateFlags |=
ApplicationInfo.PRIVATE_FLAG_ISOLATED_SPLIT_LOADING;
    }

    pkg.mCompileSdkVersion = sa.getInteger(
            com.android.internal.R.styleable.AndroidManifest_compileSdkVersion, 0);
    pkg.applicationInfo.compileSdkVersion = pkg.mCompileSdkVersion;
    pkg.mCompileSdkVersionCodename = sa.getNonConfigurationString(
            com.android.internal.R.styleable.AndroidManifest_compileSdkVersionCodename, 0);
    if (pkg.mCompileSdkVersionCodename != null) {
        pkg.mCompileSdkVersionCodename = pkg.mCompileSdkVersionCodename.intern();
    }
    pkg.applicationInfo.compileSdkVersionCodename = pkg.mCompileSdkVersionCodename;

    sa.recycle();

    return parseBaseApkCommon(pkg, null, res, parser, flags, outError);
}
```

可以明确地看到它解析了App的versionName，versionCode,baseVersionCode。那么App的各个组件是在哪里解析的呢？ 那么我们继续分析方法parseBaseApplication，一切都十分清晰了。

```
private boolean parseBaseApplication(Package owner, Resources res,
        XmlResourceParser parser, int flags, String[] outError)
    throws XmlPullParserException, IOException {

    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() > innerDepth)) {
        String tagName = parser.getName();
        if (tagName.equals("activity")) {
            Activity a = parseActivity(owner, res, parser, flags, outError, cachedArgs,
false, owner.baseHardwareAccelerated);//解析activity标签
            if (a == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            hasActivityOrder |= (a.order != 0);
            owner.activities.add(a);

        } else if (tagName.equals("receiver")) {//解析广播标签
            Activity a = parseActivity(owner, res, parser, flags, outError, cachedArgs,
                    true, false);
```

```
            if (a == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            hasReceiverOrder |= (a.order != 0);
            owner.receivers.add(a);

        } else if (tagName.equals("service")) {//解析服务标签
            Service s = parseService(owner, res, parser, flags, outError, cachedArgs);
            if (s == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            hasServiceOrder |= (s.order != 0);
            owner.services.add(s);

        } else if (tagName.equals("provider")) {//解析provider标签
            Provider p = parseProvider(owner, res, parser, flags, outError, cachedArgs);
            if (p == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            owner.providers.add(p);

        } else if (tagName.equals("activity-alias")) {//解析activity-alias
            Activity a = parseActivityAlias(owner, res, parser, flags, outError,
cachedArgs);
            if (a == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            hasActivityOrder |= (a.order != 0);
            owner.activities.add(a);

        } else if (parser.getName().equals("meta-data")) {//解析meta-data
            // note: application meta-data is stored off to the side, so it can
            // remain null in the primary copy (we like to avoid extra copies because
            // it can be large)
            if ((owner.mAppMetaData = parseMetaData(res, parser, owner.mAppMetaData,
                    outError)) == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }
        }
}
```

通过上面的代码十分明显的发现这里会将App manifest文件中定义的各个组件解析出来，并且存入对应的集合当中。至此，整个信息树绘制完毕，已经存储好了这个应用的组件信息。

2.信息匹配

我们看一下执行intent的具体方法，一般情况下，我们都是使用startActivity方法。我们点进去最终会调用startActivityForResult方法。

```
public void startActivityForResult(@RequiresPermission Intent intent, int requestCode,
        @Nullable Bundle options) {
    if (mParent == null) {
        options = transferSpringboardActivityOptions(options);
        Instrumentation.ActivityResult ar =
            mInstrumentation.execStartActivity(
                this, mMainThread.getApplicationThread(), mToken, this,
                intent, requestCode, options);
        if (ar != null) {
            mMainThread.sendActivityResult(
                mToken, mEmbeddedID, requestCode, ar.getResultCode(),
                ar.getResultData());
        }
```

其中核心代码是execStartActivity方法。 这个方法经过一系列执行，最终调用Intent类中的 resolveActivityInfo 方法

```
public ActivityInfo resolveActivityInfo(@NonNull PackageManager pm,
        @PackageManager.ComponentInfoFlags int flags) {
    ActivityInfo ai = null;
    if (mComponent != null) {
        try {
            ai = pm.getActivityInfo(mComponent, flags);
        } catch (PackageManager.NameNotFoundException e) {
            // ignore
        }
    } else {
        ResolveInfo info = pm.resolveActivity(
            this, PackageManager.MATCH_DEFAULT_ONLY | flags);
        if (info != null) {
            ai = info.activityInfo;
        }
    }

    return ai;
}
```

这个方法会根据传进来的flags在PMS所存储的组件列表中挑选最合适的系统组件，进行回传。

整个流程就是:

在安卓系统启动时，PackageManagerService(PMS)就会启动，PMS将分析所有已安装的应用信息，构建相对应的信息表，当用户需要通过Intent跳转到某个组件时，会根据Intent中包含的信息，然后从PMS中查找对应的组件列表，最后跳转到目标组件。

Intent的作用

Intent可以实现界面间的切换，过程中可以包含动作和动作数据，它是连接四大组件的纽带。具体的功能如下：

1.打开指定网页

```
button1.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent();
                // 指定了Intent的action是 Intent.ACTION_VIEW，表示查看的意思，这是一个Android系统内置的动作；
                intent.setAction(Intent.ACTION_VIEW);//方法:android.content.Intent.Intent(String action)
                Uri data = Uri.parse("http://www.baidu.com");
                intent.setData(data);
                startActivity(intent);
            }
        });
```

2.打电话

方式一：打开拨打电话的界面：

```
Intent intent = new Intent(Intent.ACTION_DIAL);
intent.setData(Uri.parse("tel:10086"));
startActivity(intent);
```

方式二：直接拨打电话：

```
Intent intent = new Intent(Intent.ACTION_CALL);
intent.setData(Uri.parse("tel:10086"));
startActivity(intent);
```

我们使用这项功能需要添加权限:

```
<uses-permission android:name="android.permission.CALL_PHONE"/>
```

3.发送信息

方式一：跳转信息发送的界面：action+type

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setType("vnd.android-dir/mms-sms");
intent.putExtra("sms_body", "具体短信内容"); //"sms_body"为固定内容
startActivity(intent);
```

方式二：跳转发送短信的界面（同时指定电话号码）：action+data

```
Intent intent = new Intent(Intent.ACTION_SENDTO);
intent.setData(Uri.parse("smsto:18780260012"));
intent.putExtra("sms_body", "具体短信内容"); //"sms_body"为固定内容
startActivity(intent);
```

4.播放指定路径音乐:

action+data+type

```
Intent intent = new Intent(Intent.ACTION_VIEW);
Uri uri = Uri.parse("file:///storage/sdcard0/平凡之路.mp3"); ////路径也可以写
成："/storage/sdcard0/平凡之路.mp3"
intent.setDataAndType(uri, "audio/mp3"); //方法：Intent
android.content.Intent.setDataAndType(Uri data, String type)
startActivity(intent);
```

5.卸载程序:

action+data（例如点击按钮，卸载某个应用程序，根据包名来识别）

注：无论是安装还是卸载，应用程序是根据包名package来识别的。

```
Intent intent = new Intent(Intent.ACTION_DELETE);
Uri data = Uri.parse("package:com.example.smyh006intent01");
intent.setData(data);
startActivity(intent);
```

6.安装程序：action+data+type

```
Intent intent = new Intent(Intent.ACTION_VIEW);
Uri data = Uri.fromFile(new File("/storage/sdcard0/AndroidTest/smyh006_Intent01.apk"));
//路径不能写成："file:///storage/sdcard0/···"
intent.setDataAndType(data, "application/vnd.android.package-archive");  //Type的字符串为固定内
容
startActivity(intent);
```

Intent传递的数据类型

Intent基本上可以涵盖大多数的数据类型，当然使用者也可以基于需要自定义数据类型，不过这些数据类型必须实现Parcelable接口或是Serializable接口。具体的数据类型大家可以在下面一一学习到：

1. **传递简单的数据**

1.1 存取一个数据

```
// 存数据
Intent i1 = new Intent(A.this,B.class);
i1.putExtra("key",value);
startActivity(i1);

// 取数据
Intent i2 = getIntent();
getStringExtra("key");
```

1.2 存取多个数据

```
// 存数据
Intent i1 = new Intent(A.this,B.class);
Bundle bundle = new Bundle();
bundle.putInt("num",1);
bundle.putString("detail","haha");
i1.putExtras(bundle);
startActivity(i1);

// 取数据
Intent i2 = getIntent();
Bundle bundle = i2.getExtras();
int i = bundle.getInt("num");
String str = bundle.getString("detail");
```

2. **传递数组**

```
bundle.putStringArray("StringArray",new String[]{"hehe","呵呵"});
```

读取数组：

```
String[] str = bundle.getStringArray("StringArray");
```

3. **传递集合**

3.1 List<基本数据类型或String>

```
intent.putStringArrayListExtra(name, value)
intent.putIntegerArrayListExtra(name, value)
```

读取集合

```
intent.getStringArrayListExtra(name)
intent.getIntegerArrayListExtra(name)
```

3.2 List< Object> 将list强制类型转换成Serializable类型,然后传入(可用Bundle做媒介) 写入集合：

```
putExtras(key, (Serializable)list)
```

读取集合：

```
(List<Object>) getIntent().getSerializable(key)
```

注意：Object类需要实现Serializable接口

3.3 Map<String, Object>或许更复杂的 解决方案是：外层包装个List

```
//传递复杂些的参数
Map<String, Object> map1 = new HashMap<String, Object>();
map1.put("key1", "value1");
map1.put("key2", "value2");
List<Map<String, Object>> list = new ArrayList<Map<String, Object>>();
list.add(map1);

Intent intent = new Intent();
intent.setClass(MainActivity.this,ComplexActivity.class);
Bundle bundle = new Bundle();

//须定义一个list用于在budnle中传递需要传递的ArrayList<Object>,这个是必须要的
ArrayList bundlelist = new ArrayList();
bundlelist.add(list);
bundle.putParcelableArrayList("list",bundlelist);
intent.putExtras(bundle);
startActivity(intent);
```

4. 传递对象

有两种传递对象的方式：通过Serializable,Parcelable序列化或者将对象转换成Json字符串， 不推荐使用Android内置的Json解析器，可使用Gson第三方库!

4.1 将对象转换为Json字符串 写入数据：

```
Book book=new Book();
book.setTitle("Java编程大法");
Author author=new Author();
author.setId(1);
author.setName("Bruce Eckel");
book.setAuthor(author);
Intent intent=new Intent(this,SecondActivity.class);
intent.putExtra("book",new Gson().toJson(book));
startActivity(intent);
```

读取数据

```
String bookJson=getIntent().getStringExtra("book");
Book book=new Gson().fromJson(bookJson,Book.class);
Log.d(TAG,"book title->"+book.getTitle());
Log.d(TAG,"book author name->"+book.getAuthor().getName());
```

4.2 运用Serializable序列化对象 Serializable 是序列化的意思，表示将一个对象换成可存储或可传输的状态。序列化后的对象可以在网络上进行传输，或存储到本地。至于序列化的方法是很简单，只需要让一个类去实现Serializable这个接口。 比如说有一个Person 类，其中包含两个字段name 和age，可以这样写：

```
public class Person implements Serializable{
private String name;
private int age;
public String getName() {
        return name;
    }
public void setName(String name) {
        this.name = name;
    }
public int getAge() {
        return age;
        }
public void setAge(int age) {
        this.age = age;
    }
}
```

其中get、set 方法用于赋值和读取字段，第一行是最重要。这里让Person 类去实现了Serializable 接口，这样一来所有的Person 都可序列化。 接下来在FirstActivity 中的写法非常容易：

```
Person person = new Person();
person.setName("Tom");
person.setAge(20);
Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
intent.putExtra("person_data", person);
startActivity(intent);
```

可以看到，这里我们创建了一个Person 的对象，然后就直接将它传入到putExtra()方法中了。由于Person 类实现了Serializable 接口。

接下来在SecondActivity 中获取这个对象也很容易，写法如下：

```
Person person = (Person) getIntent().getSerializableExtra("person_data");
```

4.3 使用Parcelable序列化对象 除了Serializable 以外，使用Parcelable 也可以实现同样的效果，不过和将对象进行序列化不同，Parcelable 方式的实现原理是分解完整的对象，而分解后的每一部分都是Intent 所支持的数据类型，这样也就实现传递对象。 下面我们来看一下Parcelable 的实现方式，修改Person 中的代码，如下所示：

```
public class Person implements Parcelable {
    private String name;
    private int age;

    @Override
    public int describeContents() {
        // TODO Auto-generated method stub
        return 0;
    }
```

```
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        // TODO Auto-generated method stub
        dest.writeString(name);
        dest.writeInt(age);
    }
    public static final Parcelable.Creator<Person> CREATOR=new Parcelable.Creator<Person>()
{

        @Override
        public Person createFromParcel(Parcel source) {
            // TODO Auto-generated method stub
            Person person=new Person();
            person.name=source.readString();
            person.age=source.readInt();
            return person;
        }

        @Override
        public Person[] newArray(int size) {
            // TODO Auto-generated method stub
            return new Person[size];
        }
    };
}
```

Parcelable 的实现方式是要复杂一些。可以看出，首先我们让Person 类去实现了Parcelable 接口，这样就必须重写两个方法，describeContents()和writeToParcel()。其中describeContents()方法直接返回0就可以了，而writeToParcel()方法中我们需要调用Parcel的writeXxx()方法将Person 类中的字段逐个写出。注意writeString()方法是字符串型数据调用，writeInt()方法是整型数据调用，以此类推。

除此之外，我们还必须在Person 类中声明一个名为CREATOR 的常量，这里创建了Parcelable.Creator 接口的一个实现类，并将泛型类型指定为Person。接着需要重写两个方法createFromParcel()和newArray()，在createFromParcel()方法中去获取刚才写出的name 和age字段，并创建一个Person 对象并进行返回，其中name 和age 都是调用Parcel 的readXxx()方法获取到，注意这里获取的顺序一定要和刚才写出的顺序完全一致。而newArray()方法中的实现就容易多了，只需要new 出一个Person 数组，并使用方法中传入的size 作为数组大小就可以了。

接下来在FirstActivity 中我们仍然可以使用相同的代码来传递Person 对象，只不过在SecondActivity 中获取对象的时候需要稍微做些改动，如下所示：

```
Person person = (Person) getIntent().getParcelableExtra("person_data");
```

5. **传递Bitmap**

bitmap默认实现Parcelable接口,直接传递即可

```
Bitmap bitmap = null;
Intent intent = new Intent();
Bundle bundle = new Bundle();
bundle.putParcelable("bitmap", bitmap);
intent.putExtra("bundle", bundle);
```

总结

Android都使用统一的Intent对象来封装这一"启动意图"，Intent在启动四大组件的过程中，会去查找PMS中存储的四大组件的信息来进行数据传递。需要注意的是Intent采用了隐士和显示两种调用方案。它所能传递的数据类型基本上包含了所有的类型，只是自定义的数据类型需要必须实现Parcelable接口或是Serializable接口。

10.30 Activity启动流程分析

详细讲解

享学课堂移动开发课程：Framework专题 AMS部分

这道题想考察什么

考核activity 的启动过程，以及启动过程中各大类的关系

考生应该如何回答

Activity的启动过程一般有两种情况：第一种，activity所在的进程没有创建，那么这个过程就会涉及到App进程的创建，我们可以在《Android app进程是怎么启动的》的章节去得到详细说明，在这里就不赘述了；第二种，App进程存在，那么对应的Activity启动流程将是下面分析的重点。

下文会分析整个activity 启动的流程，同时分析一下它的生命周期的切换过程，最后再分析一下管理activity的几个类的基本情况。

1.Activity启动流程

平时我们开发的应用都是展示在Android系统桌面上，这个系统桌面其实也是一个Android应用，它叫Launcher。所以本文通过源码层面从Launcher调用ATMS，ATMS再通过AMS发送socket请求给zygote，由zygote fork出app进程，然后再通过反射调用ActivityThread 的main函数，开始运行app进程的代码。在ActivityThread main函数中，会执行AMS的 attachApplication方法，将Application的binder赋值给AMS，然后再由AMS通过这个IBinder去调用ApplicationThread的bindApplication函数执行application的生命周期,紧接着AMS再直线ATMS的attachApplication方法，进而启动Activity并执行Activity的相关生命周期。

下文我们将重点对Activity启动流程进行分析。（文中源码基于Android 11 ）。

1.1 Launcher 调用Activity的过程

在这个阶段，Launcher只是一个app，当用户点击Launcher上app icon的时候就会执行Launcher中的代码，在这个流程里面，Launcher主要会通过Instrumentation类跨进程调用ATMS（android 10之前是AMS）的代码去启动Activity，具体的其他细节不是我们关注的重点，有感兴趣的同学可以自行查看源码进行了解。我们将分析的重点放到ATMS启动Activity的流程里面来。

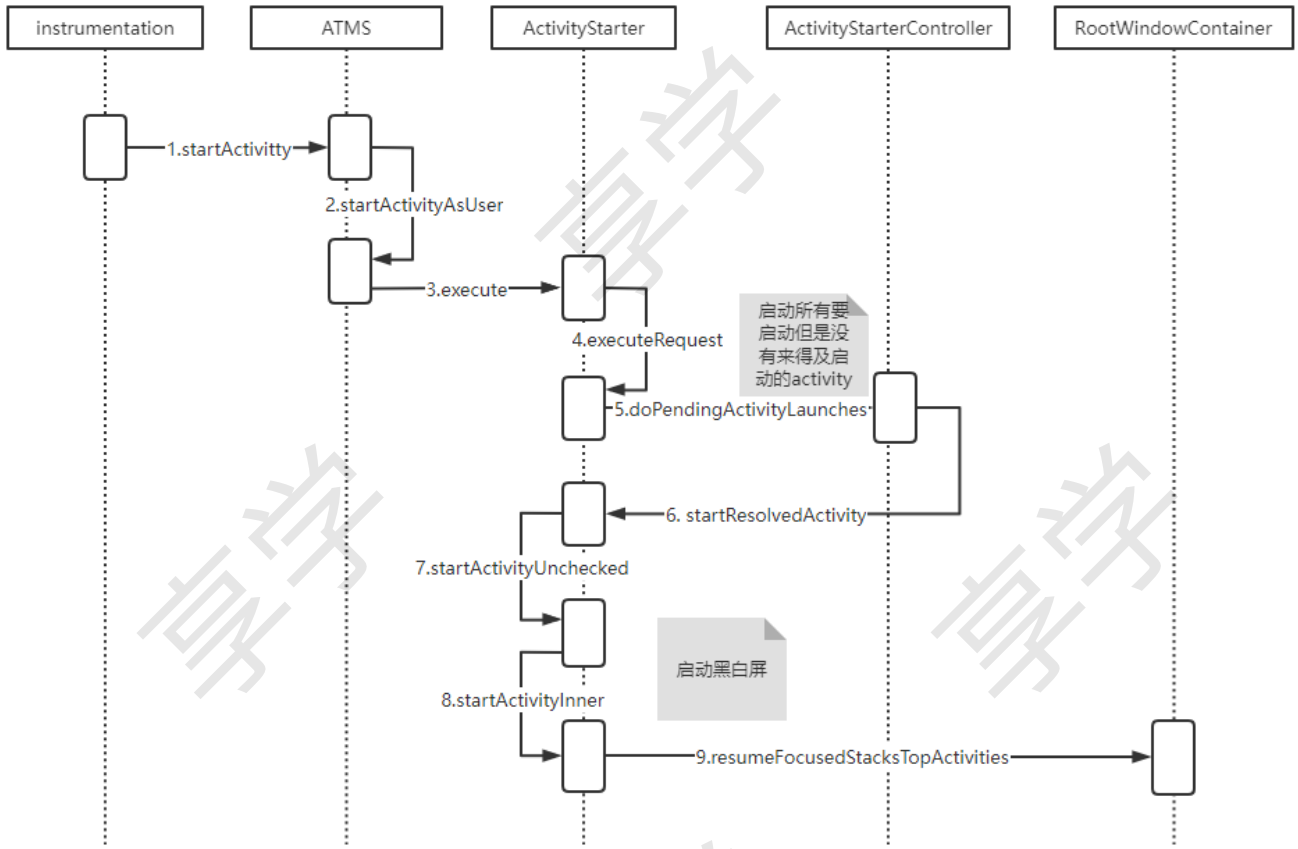


图 29-1

从上图29-1，流程图可以看到，ActivityTaskManagerService是通过ActivityStarter来启动Activity，为什么存在ActivityStarter类呢?

1.1.1 ActivityStarter类的说明

ActivityStarter它是一个用于解释如何启动一个Activity的控制器，用来配置actiivty的各种熟悉并加载启动 Activity 的类，此类记录所有逻辑，用于确定如何将意图和标志转换为Activity以及关联的任务和堆栈。

在ActivityStarter中包含了一个静态内部类Request，这个类或许非常恰当的说明ActivityStarter的作用。它的部分代码如下：

```
static class Request {
    ...
    IApplicationThread caller;
    Intent intent;
    NeededUriGrants intentGrants;
    // A copy of the original requested intent, in case for ephemeral app launch.
    Intent ephemeralIntent;
    String resolvedType;
    ActivityInfo activityInfo;
    ResolveInfo resolveInfo;
    IVoiceInteractionSession voiceSession;
    IVoiceInteractor voiceInteractor;
    IBinder resultTo;
    String resultWho;
    int requestCode;
    int callingPid = DEFAULT_CALLING_PID;
```

```
    int callingUid = DEFAULT_CALLING_UID;
    String callingPackage;
    @Nullable String callingFeatureId;
    int realCallingPid = DEFAULT_REAL_CALLING_PID;
    int realCallingUid = DEFAULT_REAL_CALLING_UID;
    int startFlags;
    SafeActivityOptions activityOptions;
    boolean ignoreTargetSecurity;
    boolean componentSpecified;
    boolean avoidMoveToFront;
    ActivityRecord[] outActivity;
    Task inTask;
    String reason;
    ProfilerInfo profilerInfo;
    Configuration globalConfig;
    int userId;
    WaitResult waitResult;
    int filterCallingUid;
    PendingIntentRecord originatingPendingIntent;
    boolean allowBackgroundActivityStart;
    ...
}
```

上面的代码大家不能发现，这个类里面有非常多的参数，而这些参数就包含了启动activity的相关信息和被启动Activity的相关信息。我们不难知道，在启动activity之前把所有信息都准备全，这个工作就需要交给ActivityStarter类来做。另外，ActivityA启动ActivityB时，代码是这样的：startActivity(new Intent(ActivityA.this,ActivityB.class))，参数信息只有三个Intent，context和ActivityB.class，这些信息就会在ActivtyStarter类中被封装成为一个request，有了这些信息，才能去进行启动的下一步工作。

我们一起来看一下图29-1中的step 4 executeRequest的代码：

```
private int executeRequest(Request request) {
        ActivityInfo aInfo = request.activityInfo;
        ResolveInfo rInfo = request.resolveInfo;
        String resultWho = request.resultWho;
        Task inTask = request.inTask;

        ActivityRecord sourceRecord = null;//code 1
        ActivityRecord resultRecord = null;//code 2

        if (resultTo != null) {
            sourceRecord = mRootWindowContainer.isInAnyStack(resultTo); //code 3
            ......
            if (sourceRecord != null) {
                if (requestCode >= 0 && !sourceRecord.finishing) {
                    resultRecord = sourceRecord;
                }
            }
        }
        ......
        //code 4
        final ActivityRecord r = new ActivityRecord(mService,callerApp, callingPid,
                callingUid, callingPackage, callingFeatureId, intent, resolvedType,
```

```
                aInfo,mService.getGlobalConfiguration(), resultRecord, resultWho,
                requestCode,request.componentSpecified, voiceSession != null,
                mSupervisor, checkedOptions, sourceRecord);
        //code 5
        mLastStartActivityResult = startActivityUnchecked(r, sourceRecord, voiceSession,
                request.voiceInteractor, startFlags, true /* doResume */, checkedOptions,
                inTask,restrictedBgActivity, intentGrants);
}
```

1)上面代码中code 1 &code 2 定义了两个ActivityRecord；

2）code3处初始化sourceRecord，这个sourceRecord，就是构建了当前启动Activity的activity在AMS中的ActivtyRecord。比如ActivityA启动ActivityB，那么这个ActivityRecord就是ActivityA 在AMS中的存在形式。当然这也就是说要在启动新Activity之前要知道sourceReord是谁。

3）code4 new了一个ActivityRecord，它就是要被启动的Activity。

4）code5 调了startActivityUnchecked()方法，执行下一步的生命周期流程的调度。

1.1.2 ActivityStartController类

这个类相对较简单，看名字知道它是一个ActivityStart的控制器，这个类主要是接收Activity启动的各种请求，并将这些请求封装成为一个可以被ActivityStarter处理的活动。所以，ActivityStarter对象的产生，是由ActivityStartController提供的。同时，它还会负责处理围绕活动启动流程开始的环节的逻辑，如图29-1中的第5步，当app中有一系列的activity处于pending需要启动的时候，这个时候就会调用doPendingActivityLaunches方法，处理所有的pending的activity。

```
void doPendingActivityLaunches(boolean doResume) {
    //循环获取所有的pending状态的Activity，并执行启动逻辑
    while (!mPendingActivityLaunches.isEmpty()) {
        final PendingActivityLaunch pal = mPendingActivityLaunches.remove(0);
        final boolean resume = doResume && mPendingActivityLaunches.isEmpty();
        final ActivityStarter starter = obtainStarter(null /* intent */,
                "pendingActivityLaunch");
        try {
            starter.startResolvedActivity(pal.r, pal.sourceRecord, null, null,
            pal.startFlags, resume, pal.r.pendingOptions, null, pal.intentGrants);
        } catch (Exception e) {
            Slog.e(TAG, "Exception during pending activity launch pal=" + pal, e);
            pal.sendErrorResult(e.getMessage());
        }
    }
}
```

通过上面的代码，大家不难发现在图29-1中的第5步，所走的代码，其实就是将所有要启动而没有启动的Activity们进行统一的启动管理，只是真正启动的过程仍旧交由AcitivityStarter的startResolvedActivity去完成。

1.1.3 启动期间的黑白屏现象

在app启动的时候，在AcitivityStarter启动activity的时候会有一个特殊的过程，这个过程就是启动一个黑白屏，用于提醒用户，正在启动新的app。那么这个启动黑白屏的过程是怎样的的呢?

我们一起来看一下图29-1中的第8步的代码startActivityInner。

```
int startActivityInner(final ActivityRecord r, ActivityRecord sourceRecord,
        IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
        int startFlags, boolean doResume, ActivityOptions options, Task inTask,
        boolean restrictedBgActivity, NeededUriGrants intentGrants) {
    //初始化配置，mStartActivity、mLaunchMode等
    setInitialState(r, options, inTask, doResume, startFlags, sourceRecord, voiceSession,
            voiceInteractor, restrictedBgActivity);

     // 计算要启动的Activity的task标志，也就是计算启动模式
    computeLaunchingTaskFlags();
    computeSourceStack();

    //将mLaunchFlags设置给Intent，也就是设置启动模式
    mIntent.setFlags(mLaunchFlags);

    final Task reusedTask = getReusableTask();

    ......

    // Compute if there is an existing task that should be used for.
    final Task targetTask = reusedTask != null ? reusedTask : computeTargetTask();
    final boolean newTask = targetTask == null;
    mTargetTask = targetTask;

    computeLaunchParams(r, sourceRecord, targetTask);

    ...

    // code1
    // 创建启动黑白屏window
    mTargetStack.startActivityLocked(mStartActivity,
            topStack != null ? topStack.getTopNonFinishingActivity() : null, newTask,
            mKeepCurTransition, mOptions);
    if (mDoResume) {
        final ActivityRecord topTaskActivity =
                mStartActivity.getTask().topRunningActivityLocked();
        if (!mTargetStack.isTopActivityFocusable()
                || (topTaskActivity != null && topTaskActivity.isTaskOverlay()
                && mStartActivity != topTaskActivity)) {
            mTargetStack.ensureActivitiesVisible(null /* starting */,
                    0 /* configChanges */, !PRESERVE_WINDOWS);
            mTargetStack.getDisplay().mDisplayContent.executeAppTransition();
        } else {
            if (mTargetStack.isTopActivityFocusable()
                    && !mRootWindowContainer.isTopDisplayFocusedStack(mTargetStack)) {
                mTargetStack.moveToFront("startActivityInner");
            }
            //code2 将启动流程交给RootWindowContainer去执行，并通过
            mRootWindowContainer.resumeFocusedStacksTopActivities(
                    mTargetStack, mStartActivity, mOptions);
        }
    }
    ......
```

```
    return START_SUCCESS;
}
```

函数一开始先初始化启动activity需要的配置，然后再基于配置参数计算出启动模式，并将启动模式设置给Intent作为后期备用的变量。接着会运行到code1，此时便是创建一个黑白屏。在分析黑白屏创建方式前，我们先分析一下黑白屏为什么在Activity进程创建之前会启动?

当用户点击Launcher界面app icon的时候，为了让点击者在第一时间能够感受到点击的响应，此时必须要有界面切换来证明变化的存在。在app启动流程中，如果在app进程创建后才显示这个黑白屏，那么Launcher界面将出现一个比较长的等待时间，这将会被用户错误的认知为点击没有及时响应。因此，在app进程还没有创建的时候，在启动activity的过程中，一旦设置了activity的启动模式就立刻创建一个黑白屏，用于衔接点击app icon到app真正显示这中间的时间间隙，这也是在activity启动过程中的一个巧妙设计。关于黑白屏的具体显示流程，感兴趣的朋友可以去阅读其他相关章节补充学习。

1.1.4 RootWindowContainer类的说明

Android10新增的类，当activity启动的时候，会将整个启动流程转交给RootWindowContainer去执行，为什么会这样去做了？我们接下来分析。

RootWindowContainer是窗口容器（WindowContainer）的根容器，管理了所有窗口容器，设备上所有的窗口（Window）、显示（Display）都是由它来管理的。resumeFocusedStacksTopActivities函数会恢复对应任务栈顶部的Activity。这个方法会检查一些可见性相关的属性，如果当前需要resume的activityStack 是可见的，这个时候才resume，而可见性是由RootWindowContainer中的窗口控制的。所以，每个activity都有自己的窗口，为了控制activity窗口的可见性，Activity的启动必须经过RootWindowContainer。

1.1.5 小结

从图29-1可以看出，Activity的启动会有一个相对漫长的前期准备阶段，这个阶段所做的事情如下：

1）ATMS发出启动activity的请求；

2）将启动Activity的动作所需要准备的参数全部封装成为ActivityStarter里面的参数，也就是说ActivityStarter存储了一个activity启动所需要的所有的参数；

3）由于可能存在一次启动多个Activity的状况，或者积累了多个activity需要启动的情况，所以此时需要将所有可以启动的activity进行启动，因此启动流程需要去走到ActivityStarterController里面进行处理；

4）当真正启动Activity的时候，如果是启动的application的launcher activity，那么我们需要先提供一个黑白屏，因为在用户点击启动application的时候，用户需要对启动App这个动作有感知，然而此时App启动可能比较缓慢没办法立刻显示，因此就需要先创建一个黑白屏。

5）由于启动activity会改变activity的窗口的可见性，而这个可见性的管理是由RootWindowContainer进行，因此在启动路上需要经过这个类来进行可见性控制的管理。

接下来，我们继续分析Activity的启动流程

1.2 Activity启动流程在AMS中的执行

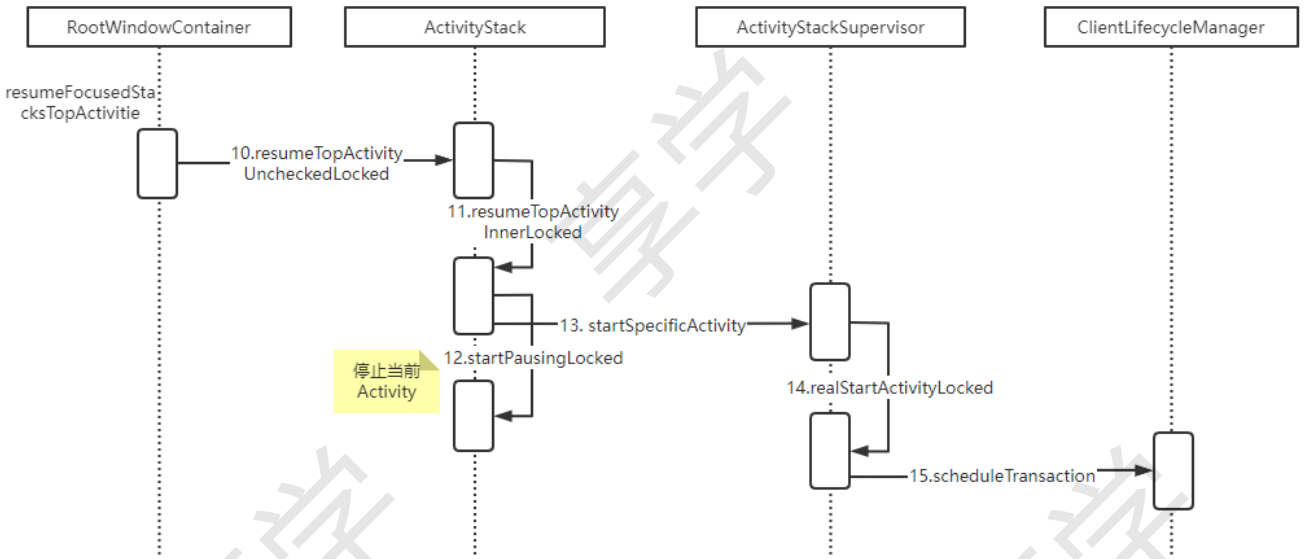


图29-2

1.2.1 ActivityRecord，Task，ActivityStack，ActivityStackSupervisor类说明

我们要理解Activity启动流程，首先是要了解一些关键类的信息。

ActivityRecord：一个ActivityRecord对应着一个Activity，保存着一个Activity的所有信息；但是一个Activity可能会有多个ActivityRecord，因为Activity可能会被启动多次，主要是取决于Activity的启动模式。

Task：Android系统中的每个Activity都位于一个Task中。一个Task能够包括多个Activity，同一个Activity也可能有多个实例。 在AndroidManifest.xml中，我们能够通过 android:launchMode 来控制Activity在Task中的实例。Task管理的意义还在于近期任务列表以及Back栈。 当你通过多任务键（有些设备上是长按Home键。有些设备上是专门提供的多任务键）调出多任务时，事实上就是从ActivityManagerService获取了近期启动的Task列表。

ActivityStack：Task是它的父类，是一个管理类，管理着所有`Activity`，内部维护了`Activity`的所有状态、特殊状态的Actvity和Activity以及相关的列表数据。

ActivityStackSupervisor：`ActivityStack`是由`ActivityStackSupervisor`来进行管理，而这个是在ATMS的`initialize`中被创建出来的。ATMS初始化的时候，会创建一个ActivityStackSupervisor对象用于统一管理一些事务：1）将显示层次结构相关的内容移动到RootWindowContainer中；2）将与activity生命周期相关的事务的处理转交给ActivityLifeCyler处理；3）处理点击物理按键Menu键后任务栈的处理。

注意：在Android 11版本中上面的类的划分已经和之前的版本进行了非常大的修改，所以职责也各不相同，编者注意到在Android12版本中，已经去掉了ActivityStack类和ActivityStackSupervisor类，替代他们的是Task类和ActivityTaskSupervisor类，感兴趣的读者可以拿源码进行阅读。

小结： 在AMS启动的时候，会创建一个ActivityStackSupervisor对象，ActivityStackSupervisor创建和管理Android系统中所有应用的ActivityStack，一个ActivityStack对应和包含一个应用中所有的栈。

有了以上对类的基本理解，我们就很容易理解图29-2的流程了。

一个app所有的Activity都会被ActivityStack进行管理，因此启动Activity自然也是在ActivityStack中进行，所以在完成启动流程的前9步准备工作后，自然就在第10步迈入了ActivityStack中。ActivityStack开始真正的启动Activity的时候，需要先执行12步，第12步会先让当前正在显示的Activity执行它的pause生命周期，然后再去执行13步：准备启动指定的Activity。

大家可以看一下ActivityStack中下面的核心代码

```
private boolean resumeTopActivityInnerLocked(ActivityRecord prev, ActivityOptions options) {
    ...

    // 停止当前的Activity
    if (mResumedActivity != null) {
        if (DEBUG_STATES) Slog.d(TAG_STATES,
                "resumeTopActivityLocked: Pausing " + mResumedActivity);
        // 停止当前的Activity
        pausing |= startPausingLocked(userLeaving, false /* uiSleeping */, next);//code1
    }
    ...

    if (next.attachedToProcess()){
    ...

    } else {
        ...
        //启动指定的Activity
        mStackSupervisor.startSpecificActivity(next, true, true); //code2
    }

    return true;
}
```

通过上面的code1，此处代码是用于pause 当前activity的代码，code2 则是启动特定Activity的代码。

关于startPausingLocked() 方法去触发当前Activity执行pause生命周期的具体流程，我们就不做系统分析了，大致流程会和执行activity onCreate的流程一致，感兴趣的读者可以自行阅读源码。我们将重点放到Activity启动上面来，所以请大家看下面的ClientTransaction传递流程。

1.3 Activity启动中事件的跨进程通信

图 29-4

从Android 8之后，activity的启动流程就加入了触发器的机制，这个机制出现的目的是为了更加友好的管理activity的生命周期，但是，本人感觉它更让人费解了。不管怎样，我们一起分析一下它的流程和类的关系，用于帮助我们了解这个流程。

ClientLifecycleManager 是管理 Activity 生命周期的，在 ActivityTaskManagerService 里面提供 getLifecycleManager 来获取此对象，其中 mLifecycleManager 是在 ActivityTaskManagerService 的构造方法里面初始化的。

```
public class ActivityTaskManagerService extends IActivityTaskManager.Stub {
    ......
    private final ClientLifecycleManager mLifecycleManager;
    ......
    public ActivityTaskManagerService(Context context) {
        mContext = context;
        mFactoryTest = FactoryTest.getMode();
        mSystemThread = ActivityThread.currentActivityThread();
        mUiContext = mSystemThread.getSystemUiContext();
```

```
        mLifecycleManager = new ClientLifecycleManager();//mLifecycleManager初始化
        mInternal = new LocalService();
        ......
    }
    ......
    //获取mLifecycleManager对象
    ClientLifecycleManager getLifecycleManager() {
        return mLifecycleManager;
    }
    ......
}
```

相关类功能说明：

ClientTransactionItem 对象，一个回调消息，client 端可以调用执行，实现了 BaseClientRequest 接口，在接口里面定义了3个方法：preExecute，execute，poseExecute。

ClientTransaction 是一个容器，持有一系列可以发送给 client 的消息（比如声明周期的状态），包括有 mActivityCallbacks 列表和一个目标状态 mLifecycleStateRequest。

TransactionExecutor 用来执行 ClientTransaction，以正确的顺序管理事务执行 execute()，定义在 ActivityThread 应用端。

ClientTransactionHandler是一个抽象类，定义了一系列生命周期处理Activity生命周期的接口，由ActivityThread 实现。

另外，有一个类也在这里和大家一起介绍一下：

ActivityLifecycleItem 继承自 ClientTransctionItem，主要的子类有 ResumeActivityItem、PauseActivityItem、StopActivityItem、DestoryActivityItem。

ActivityThread 它管理应用程序进程中主线程中执行的调度和执行活动、广播以及活动管理器请求的其他操作。

小结

图29-4展示的代码执行流程可以知道，第15步，生命周期的执行是ActivityStackSupervisor这个activity栈的大管家拿着在ATMS中所创建的ClientLifecycleManager去执行生命周期的触发器。而生命周期的处理必须要经过生命周期事物存储的容器来分发，也就是第16步走ClientTransaction。然而Activity生命周期的执行到目前为止还是在SystemServer进程也就是在AMS所在的进程中，最终Activity的生命周期的执行还必须是在App进程中，所以就有了第17步，通过binder调用将生命周期的执行分发给Activity所在的进程去执行。当IApplicationThread 收到AMS所在进程发送过来的生命周期处理消息的时候，开始发送一个handler事件来处理这个生命周期，这就是第18步的执行。最后，app进程会去执行生命周期的触发器来处理ClientTransation事件。

请大家重点关注一个细节，也就是第17步，进程间完成了一次从systemServer进程到Activity所在的app进程进行切换的过程。

1.4 应用进程中生命周期的执行流程

生命周期的执行到目前这一步算是一个重要的分水岭，因为接下来的执行过程全部是在App进程中进行，接下来的执行流程我们可以看一下下面的流程图。

 start4

图 29-5

在执行启动流程中，我们先来看第14步ActivityStackSupervisor中的realStartActivityLocked函数一段核心代码：

```
boolean realStartActivityLocked(ActivityRecord r, WindowProcessController proc,
        boolean andResume, boolean checkConfig) throws RemoteException {

    ......
    // Create activity launch transaction.
    // 创建 clientTransaction 对象
    final ClientTransaction clientTransaction = ClientTransaction.obtain(
            proc.getThread(), r.appToken);

    final DisplayContent dc = r.getDisplay().mDisplayContent;
    //code1
    // 添加LaunchActivityItem
    clientTransaction.addCallback(LaunchActivityItem.obtain(new Intent(r.intent),
            System.identityHashCode(r), r.info,
            // TODO: Have this take the merged configuration instead of separate global
            // and override configs.
            mergedConfiguration.getGlobalConfiguration(),
            mergedConfiguration.getOverrideConfiguration(), r.compat,
            r.launchedFromPackage, task.voiceInteractor, proc.getReportedProcState(),
            r.getSavedState(), r.getPersistentSavedState(), results, newIntents,
            dc.isNextTransitionForward(), proc.createProfilerInfoIfNeeded(),
            r.assistToken, r.createFixedRotationAdjustmentsIfNeeded()));

    // Set desired final state.
    final ActivityLifecycleItem lifecycleItem;
    if (andResume) {
        //需要Rusume的话，设置ResumeActivityItem到clientTransaction中
        lifecycleItem = ResumeActivityItem.obtain(dc.isNextTransitionForward());
    } else {
        lifecycleItem = PauseActivityItem.obtain();
    }
    //code 2
    clientTransaction.setLifecycleStateRequest(lifecycleItem);

    // code3
    // Schedule transaction.
    // 获取生命周期管理类 ClientLifecycleManager，并执行事务
    mService.getLifecycleManager().scheduleTransaction(clientTransaction);
    ......
    return true;
}
```

以上代码创建了一个clientTransaction对象，并设置了clientTransaction对象的各参数。从上面的代码code1，我们构建了一个LaunchActivityItem，并且将它添加到了clientTransaction的Callback列表中；然后在code2中往clientTransaction中设置了当前阶段最终想要执行的生命周期状态，也就是到生命周期的resume状态，所以再code2 代码处向clientTransation中设置了LifecycleStateRequest，也就是本次transaction希望执行的activity的生命周期的最终的状态，也就是添加了ResumeActivityItem，最终状态是activity的resume；最后在code3 把这个封装的clientTransaction 传递给了ClientLifecycleManager，这个clientTransaction对象被第17步的跨进程通信传给了app进程，所以接下来生命周期的执行就会按照AMS中构建的生命周期管理想法开始执行，也就是会按照图29-5的流程开始执行。

在Activity启动流程执行到TransactionExecutor中的excute的时候，此时的代码已经执行到了app进程里面了。我们一起来看一下第19步execute函数的执行流程，流程中我们只展示核心代码：

```
public void execute(ClientTransaction transaction) {

    ......

    executeCallbacks(transaction);  //code1

    executeLifecycleState(transaction);  //code2
    mPendingActions.clear();
    if (DEBUG_RESOLVER) Slog.d(TAG, tId(transaction) + "End resolving transaction");
}
```

在第19步，也就是在执行上面的excute函数的时候，会执行两个非常重要的代码：code1&code2，code1就是我们流程图中的第20步，而code2 就是我们流程图中的第24步。我们一起来分析一下这两个流程：

1.4.1 执行executeCallbacks函数

code1所引发的第20步，这个过程是执行在14步ActivityStackSupervisor中的realStartActivityLocked里面添加的Callback的这个阶段，我们通过上面代码的分析可以得到一个结论，这次从第20步执行到第23步，整个流程是围绕当前Activity的launch流程进行。当大家阅读第22步handleLaunchActivity源码的时候，不难发现，在代码执行中会执行Activity的创建以及attach的生命周期函数，然后再执行onCreate生命周期，具体的代码如下：

```
ActivityThread.java

public Activity handleLaunchActivity(ActivityClientRecord r,
        PendingTransactionActions pendingActions, Intent customIntent) {
    ......

    WindowManagerGlobal.initialize(); //code1


    final Activity a = performLaunchActivity(r, customIntent);  //code2

    ......
}

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ......

    ContextImpl appContext = createBaseContextForActivity(r);
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = appContext.getClassLoader();
        //创建一个activity
        activity = mInstrumentation.newActivity(  //code 3
                cl, component.getClassName(), r.intent);
        StrictMode.incrementExpectedActivityCount(activity.getClass());
        r.intent.setExtrasClassLoader(cl);
        r.intent.prepareToEnterProcess();
        if (r.state != null) {
```

```
            r.state.setClassLoader(cl);
        }
    } catch (Exception e) {
        if (!mInstrumentation.onException(activity, e)) {
            throw new RuntimeException(
                "Unable to instantiate activity " + component
                + ": " + e.toString(), e);
        }
    }

    try {
        //LoadedApk 构建 makeApplication
        Application app = r.packageInfo.makeApplication(false, mInstrumentation);

        ......

        if (activity != null) {
            ......

            // Activity resources must be initialized with the same loaders as the
            // application context.
            appContext.getResources().addLoaders(
                    app.getResources().getLoaders().toArray(new ResourcesLoader[0]));

            appContext.setOuterContext(activity);

            //code4
            //会在这个方法中创建Activity的PhoneWindow，并绑定对应的WindowManager。
            activity.attach(appContext, this, getInstrumentation(), r.token,
                    r.ident, app, r.intent, r.activityInfo, title, r.parent,
                    r.embeddedID, r.lastNonConfigurationInstances, config,
                    r.referrer, r.voiceInteractor, window, r.configCallback,
                    r.assistToken);

            if (customIntent != null) {
                activity.mIntent = customIntent;
            }
            r.lastNonConfigurationInstances = null;
            checkAndBlockForNetworkAccess();
            activity.mStartedActivity = false;
            int theme = r.activityInfo.getThemeResource();
            if (theme != 0) {
                activity.setTheme(theme);
            }

            activity.mCalled = false;

            //code5
            // 设置 mLifecycleState 为 ON_CREATE
            if (r.isPersistable()) {
                mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);
            } else {
                mInstrumentation.callActivityOnCreate(activity, r.state);
```

```
            }
            if (!activity.mCalled) {
                throw new SuperNotCalledException(
                    "Activity " + r.intent.getComponent().toShortString() +
                    " did not call through to super.onCreate()");
            }
            r.activity = activity;
            mLastReportedWindowingMode.put(activity.getActivityToken(),
                    config.windowConfiguration.getWindowingMode());
        }
        // 设置 mLifecycleState 为 ON_CREATE
        r.setState(ON_CREATE);

        // updatePendingActivityConfiguration() reads from mActivities to update
        // ActivityClientRecord which runs in a different thread. Protect modifications to
        // mActivities to avoid race.
        synchronized (mResourcesManager) {
            mActivities.put(r.token, r);
        }

    } catch (SuperNotCalledException e) {
        throw e;

    } catch (Exception e) {
        if (!mInstrumentation.onException(activity, e)) {
            throw new RuntimeException(
                "Unable to start activity " + component
                + ": " + e.toString(), e);
        }
    }

    return activity;
}
```

上面代码code1处是启动windowManagerGlobal，由于windowManagerGlobal是单例模式，所以，一般认为windowManagerGlobal的初始化就是在此时，其实就是为activity准备WindowManagerService。在code2处，将执行performLaunchActivity，大家仔细阅读performLaunchActivity的函数会不难发现 code3处创建了一个activity实例对象。然后执行到code4处，此处执行了activity的attach 生命周期，在这个生命周期里面，构建了activity的唯一的phoneWindow对象，并且并绑定对应的WindowManager方便后期使用。然后代码会执行到code5，在code5里面就是调用Activity的onCreate生命周期的过程，同时mLifecycleState被设置为ON_CREATE，这个状态在后面执行第26步的时候将会用到。

1.4.2 执行executeLifecycleState函数

上面主要介绍了 TransactionExecutor#execute执行executeCallbacks的过程，下面我们一起介绍一下执行executeLifecycleState的过程，也就是执行第24步到31步的过程。

```
private void executeLifecycleState(ClientTransaction transaction) {
        //获取ActivityLifecycleItem，我们知道这里获取的是我们之前添加的ResumeActivityItem
        //其值是由第14步ActivityStackSupervisor中的realStartActivityLocked函数
        final ActivityLifecycleItem lifecycleItem =
```

```
            transaction.getLifecycleStateRequest();
        if (lifecycleItem == null) {
            // No lifecycle request, return early.
            return;
        }

        final IBinder token = transaction.getActivityToken();
        final ActivityClientRecord r = mTransactionHandler.getActivityClient(token);
        if (DEBUG_RESOLVER) {
            Slog.d(TAG, tId(transaction) + "Resolving lifecycle state: "
                    + lifecycleItem + " for activity: "
                    + getShortActivityName(token, mTransactionHandler));
        }

        if (r == null) {
            // Ignore requests for non-existent client records for now.
            return;
        }

        // Cycle to the state right before the final requested state.
        //code1 关键点,ResumeActivityItem的getTargetState 是 ON_RESUME
        cycleToPath(r, lifecycleItem.getTargetState(), true /* excludeLastState */,
                    transaction);

        // Execute the final transition with proper parameters.
        //code2 执行 ResumeActivityItem 的 execute
        lifecycleItem.execute(mTransactionHandler, token, mPendingActions);
        lifecycleItem.postExecute(mTransactionHandler, token, mPendingActions);
}
```

从上面的代码，我们发现code1处有一个非常重要的函数cycleToPath(),其中的 lifecycleItem.getTargetState() 返回值是 ON_RESUME。

```
private void cycleToPath(ActivityClientRecord r, int finish, boolean excludeLastState,
            ClientTransaction transaction) {
        //这里在performLaunchActivity设置了LifecycleState为ON_CREATE,即：start是ON_CREATE，
        final int start = r.getLifecycleState();
        if (DEBUG_RESOLVER) {
            Slog.d(TAG, tId(transaction) + "Cycle activity: "
                    + getShortActivityName(r.token, mTransactionHandler)
                    + " from: " + getStateName(start) + " to: " + getStateName(finish)
                    + " excludeLastState: " + excludeLastState);
        }
        //这里的 start 是 ON_CREATE,finish 是 ON_RESUME
        //通过 mHelper 调用 getLifecyclePath 返回的 path 是 ON_START，下面会有解析
        //这里是 Activity 执行 onStart 函数的关键所在
        final IntArray path = mHelper.getLifecyclePath(start, finish, excludeLastState);
        //执行path中的相关的生命周期函数
        performLifecycleSequence(r, path, transaction);
}

private void performLifecycleSequence(ActivityClientRecord r, IntArray path,
```

```
            ClientTransaction transaction) {
       //通过mHelper调用getLifecyclePath返回的path 是 ON_START
        final int size = path.size();
        for (int i = 0, state; i < size; i++) {
            state = path.get(i);
            if (DEBUG_RESOLVER) {
                Slog.d(TAG, tId(transaction) + "Transitioning activity: "
                        + getShortActivityName(r.token, mTransactionHandler)
                        + " to state: " + getStateName(state));
            }
            switch (state) {
                case ON_CREATE:
                    mTransactionHandler.handleLaunchActivity(r, mPendingActions,
                            null /* customIntent */);
                    break;
                case ON_START:
                    mTransactionHandler.handleStartActivity(r, mPendingActions);
                    break;
                case ON_RESUME:
                    mTransactionHandler.handleResumeActivity(r.token, false /*
finalStateRequest */,
                            r.isForward, "LIFECYCLER_RESUME_ACTIVITY");
                    break;
                case ON_PAUSE:
                    mTransactionHandler.handlePauseActivity(r.token, false /* finished */,
                            false /* userLeaving */, 0 /* configChanges */, mPendingActions,
                            "LIFECYCLER_PAUSE_ACTIVITY");
                    break;
                case ON_STOP:
                    mTransactionHandler.handleStopActivity(r.token, false /* show */,
                            0 /* configChanges */, mPendingActions, false /*
finalStateRequest */,
                            "LIFECYCLER_STOP_ACTIVITY");
                    break;
                case ON_DESTROY:
                    mTransactionHandler.handleDestroyActivity(r.token, false /* finishing
*/,
                            0 /* configChanges */, false /* getNonConfigInstance */,
                            "performLifecycleSequence. cycling to:" + path.get(size - 1));
                    break;
                case ON_RESTART:
                    mTransactionHandler.performRestartActivity(r.token, false /* start */);
                    break;
                default:
                    throw new IllegalArgumentException("Unexpected lifecycle state: " +
state);
            }
        }
}

//TransactionExecutorHelper.java
//excludeLastState 为 true, start 为 ON_CREATE (1) ,finish 为 ON_RESUME (3)
public IntArray getLifecyclePath(int start, int finish, boolean excludeLastState) {
```

```
        if (start == UNDEFINED || finish == UNDEFINED) {
            throw new IllegalArgumentException("Can't resolve lifecycle path for undefined
state");
        }
        if (start == ON_RESTART || finish == ON_RESTART) {
            throw new IllegalArgumentException(
                    "Can't start or finish in intermittent RESTART state");
        }
        if (finish == PRE_ON_CREATE && start != finish) {
            throw new IllegalArgumentException("Can only start in pre-onCreate state");
        }

        mLifecycleSequence.clear();
        if (finish >= start) {//走到此分支
            // just go there
            for (int i = start + 1; i <= finish; i++) {
            //把  ON_START  和 ON_RESUME 添加到 mLifecycleSequence 中
                mLifecycleSequence.add(i);
            }
        } else { // finish < start, can't just cycle down
            ......
        }

        // Remove last transition in case we want to perform it with some specific params.
        //  关键点:因为 excludeLastState 为 true,所以删除掉 ON_RESUME 状态
        if (excludeLastState && mLifecycleSequence.size() != 0) {
            mLifecycleSequence.remove(mLifecycleSequence.size() - 1);
        }

        return mLifecycleSequence;
    }
```

通过以上代码逻辑结构，大家不难发现，整个过程是基于activity 的lifecycle的初始状态以及目标状态来进行运算，找到中间需要响应的其他生命周期状态，也就是计算从ON_CREATE作为start的状态（在ActivityThread 中执行performLaunchActivity的过程中设置的），到以ON_RESUME作为生命周期结束状态，然后计算其中还需要经历的生命周期过程，也就是ON_START状态的过程。其实这也就是为什么activity生命周期被封装成为触发器Transaction的过程。

然后，我们再重新回到executeLifecycleState的执行，当这个函数执行完成cycleToPath函数之后（里面会触发ActivityThread的handleStartActivity去执行acitivty的onStart 生命周期）就会运行到code2部分的代码进行运行lifecycleItem.execute(mTransactionHandler, token, mPendingActions)。lifecycleItem在executeLifecycleState 中我们知道lifecycleItem的值是ResumeActivityItem，此时代码就会运行到ResumeActivityItem#execute(),也就是第29步。最后会调用ActivityThread的handleResumeActivity(),进而执行Activity的onResume生命周期。

1.4.3 小结

在本阶段，主要是在Application进程通过跨进程接收从AMS封装好的ClientTransaction对象，ClientTransaction对象里面封装了需要App进程执行Activity生命周期的所有的事件，然后在activity所在的进程中，按照触发器的触发逻辑，顺序的执行activity的 创建->attach->onCreate->onStart->onResume生命周期。

在AMS进程中的会通过ActivityStackSupervisor将activity启动相关的事件封装成为一个ClientTransaction对象，然后由ClientLifecycleManager通过ClientTransaction的函数将ClientTransaction对象以跨进程通信的方式传递给ApplicationThread。在完成了跨进程后，这个触发器所携带的命令就会在Application进程中得到触发。Application所在的进程会通过Handler来启动触发器TransactionExecutor的执行。TransactionExecutor会按照AMS中设置的逻辑，逐步去分发Activity的生命周期，直到整个触发器所携带的生命周期状态被执行完成为止，也就是onResume状态得到执行为止。

1.5 启动Activity的Activity onPause生命周期的运行

在Activity A 启动Activity B的时候，当Activity B启动流程执行到onResume生命周期之后，这个时候Activity A才会去执行它的onPause生命周期，这个逻辑是怎样的呢？大家看下面的代码。

```
ActivityThread.java
public void handleResumeActivity(IBinder token, boolean finalStateRequest, boolean
isForward,
        String reason) {
    ......
    // 回调 onResume
    final ActivityClientRecord r = performResumeActivity(token, finalStateRequest, reason);
    ......
    final Activity a = r.activity;
    ......
    if (r.window == null && !a.mFinished && willBeVisible) {
        ......
        if (a.mVisibleFromClient) {
            if (!a.mWindowAdded) {
                a.mWindowAdded = true;
                // 添加 decorView 到 WindowManager
                wm.addView(decor, l);
            } else {
                a.onWindowAttributesChanged(l);
            }
        }
    } else if (!willBeVisible) {
        ......
    }
    ......

    // 主线程空闲时会执行 Idler
    Looper.myQueue().addIdleHandler(new Idler());
}
```

在代码最后有一句代码：Looper.myQueue().addIdleHandler(new Idler())。IdleHandler 不知道大家是否熟悉，它提供了一种机制，当主线程消息队列空闲时，会执行 IdleHandler 的回调方法，如果不懂这个逻辑的可以找一下其他handler章节的内容进行学习。messageQueue 中在正常的消息处理机制之后，额外对 IdleHandler 进行了处理。当从Messagequeue中调用next函数去取Message时，在本次取到的 Message 为空或者需要延时处理的时候，就会去执行 mIdleHandlers 数组中的 IdleHandler 对象。所以，不出意外的话(主线程很忙)，当新的 Activity 完成页面绘制并显示之后，主线程就可以停下歇一歇，来执行 IdleHandler 了。再回来 handleResumeActivity() 中来，Looper.myQueue().addIdleHandler(new Idler())，这里的 Idler 是 IdleHandler 的一个具体实现类。

```
private class Idler implements MessageQueue.IdleHandler {
    @Override
    public final boolean queueIdle() {
        ActivityClientRecord a = mNewActivities;
        boolean stopProfiling = false;
        if (mBoundApplication != null && mProfiler.profileFd != null
                && mProfiler.autoStopProfiler) {
            stopProfiling = true;
        }
        if (a != null) {
            mNewActivities = null;
            IActivityTaskManager am = ActivityTaskManager.getService();
            ActivityClientRecord prev;
            do {
                if (localLOGV) Slog.v(
                    TAG, "Reporting idle of " + a +
                    " finished=" +
                    (a.activity != null && a.activity.mFinished));
                if (a.activity != null && !a.activity.mFinished) {
                    try {
                        // code1 调用 AMS.activityIdle()
                        am.activityIdle(a.token, a.createdConfig, stopProfiling);
                        a.createdConfig = null;
                    } catch (RemoteException ex) {
                        throw ex.rethrowFromSystemServer();
                    }
                }
                prev = a;
                a = a.nextIdle;
                prev.nextIdle = null;
            } while (a != null);
        }
        if (stopProfiling) {
            mProfiler.stopProfiling();
        }
        applyPendingProcessState();
        return false;
    }
}
```

上面的代码在code1处调用AMS 的activityIdle()，这个时候就顺利的将代码的执行转移到了AMS所在的进程中。

图 29-5

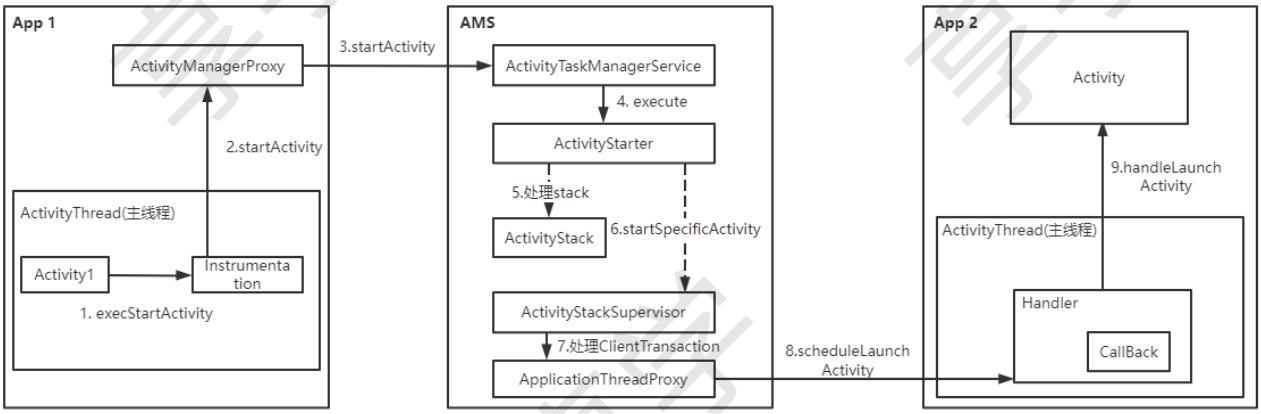
通过上图大家不难发现：上面的流程证明，我们activity的stop和destroy生命周期在此处执行。也就是说此时会执行启动activity之前的上一个activity的onStop生命周期，或者onDestroy生命周期。当然，如果我们详细的分析ActivityStackSupervisor(ASS)的代码，我们会不难发现一个细节，等待销毁的Activity被保存在了 ASS 的 mStoppingActivities 集合中，它是一个 ArrayList<ActivityRecord>。

1.5.1 小结

Activity 的 onStop/onDestroy 是依赖 IdleHandler 来回调的，正常情况下当主线程空闲时会调用。但是由于某些特殊场景下的问题，导致主线程迟迟无法空闲，onStop/onDestroy 也会迟迟得不到调用。但这并不意味着 Activity 永远得不到回收，系统提供了一个兜底机制，当 onResume 回调 10s 之后，如果仍然没有得到调用，会主动触发。

1.6 总结

Activity的启动交由ATMS触发处理，在Activity启动前需要先在ActivityStarter类中解读包括Activity的启动模式在内的各种参数信息。确定好启动信息后通过创建一个黑白屏的方式反馈给用户一个信息：我们正在响应启动app的过程中。然后将启动流程转交给ActivityStack处理，因为在ActivityStack中存储了一个Application的所有Activity，因此在此时应用的各Activity应该如何响应需要由ActivityStack来统一安排，在这个过程中就必然涉及到Activity的pause和需要启动的Activity的"realstart"。为了更好的管理Activity启动过程中的各生命周期，在ActivityStackSupervisor里面会将生命周期的事件封装成为一个ClientTransaction，并将这个ClientTransaction的对象通过跨进程的方式发送给App进程，然后由app进程的TransactionExecutor统一去触发执行，直到完成Activity的onResume生命周期为止，总流程如下图所示。



10.31 Activity A启动 ActivityB，activity的生命周期调度流程

详细讲解

享学课堂移动开发课程：Framework专题 AMS部分

这道题想考察什么

这道题主要考察Activity的启动流程应用

考生应该如何回答

在上一题的回答基础上，我们可以很容易的理解如果是Activity A启动Activity B，他们的生命周期变化如下：

1）Activity A 通过 startActivity 启动Activity B 的时候，Activity A 会先执行onPause生命周期；

2）当AMS调用触发器TransactionExecutor 执行execute函数的时候会有下面的代码：

```
public void execute(ClientTransaction transaction) {
    ...

    final IBinder token = transaction.getActivityToken();
    ...
    // 执行回调消息 LaunchActivityItem
    executeCallbacks(transaction);
    // 执行回调消息 ResumeActivityItem
    executeLifecycleState(transaction);
    ...
}
```

在 TransactionExecutor 的 execute() 函数中会分别执行回调消息 LaunchActivityItem 和 ResumeActivityItem，这里便会执行Activity B的launch 流程，这个时候就会执行Actvity B 的onCreate 生命周期， 同时在执行 executeLifecycleState(transaction) 的时候，这个函数会触发 Activity onStart 生命周期和 onResume生命周期的执行。

3）当Activity B 执行完OnResume生命周期的时候，Activity B的界面开始展示到用户面前，这个时候才会执行Activity A的onStop生命周期函数，因为只有B显示了A才能退出。

10.32 如果需要在Activity间传递大量的数据怎么办?

这道题想考察什么

1. 是否了解Activity间大量数据传递？是否考虑过Intent传递数据的大小限制问题？

考生应该如何回答

Intent 传递数据的大小是有限制的，它大概能传的数据是1M-8K，原因是Binder锁映射的内存大小就是1M-8K.一般activity间传递数据会要使用到binder，因此这个就成为了数据传递的大小的限制。那么当activity间要传递大数据采用什么方式呢？其实方式很多，我们就举几个例子给大家说明一下，但是无非就是使用数据持久化，或者内存共享方案。一般大数据的存储不适宜使用SP， MMKV，DataStore。

Activity之间传递大量数据主要有如下几种方式实现：

- LruCache
- 持久化（sqlite、file等）
- 匿名共享内存

使用LruCache

LruCache是一种缓存策略，可以帮助我们管理缓存，想具体了解的同学可以去Glide章节中具体先了解下。在当前的问题下，我们可以利用LruCache存储我们数据作为一个中转，好比我们需要Activity A向Activity B传递大量数据，我们可以Activity A先向LruCache先写入数据，之后Activity B从LruCache读取。

首先我们定义好写入读出规则：

```
public interface IOHandler {
    //保存数据
    void put(String key, String value);
```

```
    void put(String key, int value);
    void put(String key, double value);
    void put(String key, float value);
    void put(String key, boolean value);
    void put(String key, Object value);

    //读取数据
    String getString(String key);
    double getDouble(String key);
    boolean getBoolean(String key);
    float getFloat(String key);
    int getInt(String key);
    Object getObject(String key);
}
```

我们可以根据规则也就是接口，写出具体的实现类。实现类中我们保存数据使用到LruCache，这里面我们一定要设置一个大小，因为内存中数据的最大值是确定，我们保存数据的大小最好不要超过最大值的1/8.

```
LruCache<String, Object> mCache = new LruCache<>( 10 * 1024*1024);
```

写入数据我们使用比较简单：

```
@Override
public void put(String key, String value) {
    mCache.put(key, value);
}
```

好比上面写入String类型的数据，只需要接收到的数据全部put到mCache中去。

读取数据也是比较简单方便：

```
@Override
public String getString(String key) {
    return String.valueOf(mCache.get(key));
}
```

持久化数据

那就是sqlite、file等方式。将需要传递的数据写在临时文件或者数据库中，再跳转到另外一个组件的时候再去读取这些数据信息，这种处理方式会由于读写文件较为耗时导致程序运行效率较低。这种方式特点如下：

优势：

（1）应用中全部地方均可以访问

（2）即便应用被强杀也不是问题了

缺点：

（1）操做麻烦

（2）效率低下

匿名共享内存

在跨进程传递大数据的时候，我们一般会采用binder传递数据，但是Binder只能传递1M一下的数据，所以我们需要采用其他方式完成数据的传递，这个方式就是匿名共享内存。

「Anonymous Shared Memory 匿名共享内存」是 Android 特有的内存共享机制，它可以将指定的物理内存分别映射到各个进程自己的虚拟地址空间中，从而便捷的实现进程间内存共享。

Android 上层提供了一些内存共享工具类，就是基于 Ashmem 来实现的，比如 MemoryFile、 SharedMemory。

10.33 打开页面，如何实现一键退出?

这道题想考察什么?

1. 是否了解Activity栈相关的知识?

考察的知识点

1. Activity栈
2. 事件总线
3. 启动模式

考生应该如何回答

问题本质

一键退出 App 其实是 两个需求：

1. 一键结束当前App所有的Activity
2. 一键结束当前App进程

采用Activity启动模式：SingleTask

步骤1：将 App的入口 Activity 设置成 SingleTask 启动模式 // AndroidMainifest.xml中的Activity配置进行设置

```
<activity

android:launchMode="singleTask"
//属性
//standard：标准模式
//singleTop：栈顶复用模式
//singleTask：栈内复用模式
//singleInstance：单例模式
//如不设置，Activity的启动模式默认为 标准模式（standard）
</activity>
```

步骤2：在入口 `Activity` 重写 `onNewIntent()`

```
// 在该方法传入一标志位标识是否要退出App & 关闭自身
  @Override
    protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        if (intent != null) {
            // 是否退出App的标识
            boolean isExitApp = intent.getBooleanExtra("exit", false);
            if (isExitApp) {
                // 关闭自身
                this.finish();
            }
        }
    }
```

步骤3：在需要退出时调用 `exitApp()`

```
private void exitApp() {
        Intent intent = new Intent(context, MainActivity.class);
        intent.putExtra("exit", true);
        context.startActivity(intent);

    }
```

原理如下

优点

使用简单 & 方便

缺点

1. 规定 **App的入口Activity采用SingleTask启动模式**
2. 使用范围局限：只能结束当前任务栈的Activity，若出现多任务栈（即采用 `SingleInstance` 启动模式）则无法处理

应用场景 Activity单任务栈

采用Activity启动标记位

- 原理：对入口Activity采用 2 标记位：
    1. `Intent.FLAG_ACTIVITY_CLEAR_TOP`：销毁目标 `Activity` 和它之上的所有 `Activity`，重新创建目标 `Activity`
    2. `Intent.FLAG_ACTIVITY_SINGLE_TOP`：若启动的 `Activity` 位于任务栈栈顶，那么此 `Activity` 的实例就不会重建，而是重用栈顶的实例(调用 `onNewIntent()`)
- 具体使用（从 `MainActivity`（入口 `Activity`） 跳转到 `Activity2` & 一键退出）

步骤1：在`MainActivity`中设置 重写`onNewIntent()` *MainActivity.java*

```
  // 设置 按钮 跳转到Activity2
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                startActivity(new Intent(MainActivity.this, Activity2.class));

            }

        });
    }

    // 在onNewIntent () 传入一标识符
    // 作用：标识是否要退出App
    @Override
    protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        if (intent != null) {
            // 是否退出App的标识
            boolean isExitApp = intent.getBooleanExtra("exit", false);
            if (isExitApp) {
                // 关闭自身
                this.finish();
            }
        }
        // 结束进程
        // System.exit(0);
    }
}
```

步骤2：在需要退出的地方（`Activity2`）启动`MainActivity` & 设置标记位

```
// 当需要退出时，启动入口Activity
                Intent intent = new Intent();
                intent.setClass(Activity2.this, MainActivity.class);

                // 设置标记位
                intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
                // 步骤1：该标记位作用：销毁目标Activity和它之上的所有Activity，重新创建目标Activity

                intent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
                // 步骤2：若启动的Activity位于任务栈栈顶，那么此Activity的实例就不会重建，而是重用栈顶的实例( 调用实例的 onNewIntent() )

                // 在步骤1中：MainActivity的上层的Activity2会被销毁，此时MainActivity位于栈顶；由于步骤2的设置，所以不会新建MainActivity而是重用栈顶的实例&调用实onNewIntent()

                // 传入自己设置的退出App标识
                intent.putExtra("exit", true);
```

```
                startActivity(intent);
```

- 优点 使用简单 & 方便
- 缺点 使用范围局限：只能结束当前任务栈的Activity，若出现多任务栈（即采用`SingleInstance`启动模式）则无法处理
- 应用场景 Activity单任务栈

通过系统任务栈

原理：通过 ActivityManager 获取当前系统的任务栈 & 把栈内所有Activity逐个退出 具体使用

```
    @TargetApi(Build.VERSION_CODES.LOLLIPOP)

        // 1. 通过Context获取ActivityManager
        ActivityManager activityManager = (ActivityManager)
context.getApplicationContext().getSystemService(Context.ACTIVITY_SERVICE);

        // 2. 通过ActivityManager获取任务栈
        List<ActivityManager.AppTask> appTaskList = activityManager.getAppTasks();

        // 3. 逐个关闭Activity
        for (ActivityManager.AppTask appTask : appTaskList) {
            appTask.finishAndRemoveTask();
        }
        // 4. 结束进程
        // System.exit(0);
```

- 优点 使用简单、方便
- 缺点
- 使用范围局限：只能结束当前任务栈的Activity，若出现多任务栈（即采用SingleInstance启动模式）则无法处理 对 Android 版本要求较高：Android 5.0以上
- 应用场景 Android 5.0以上的 Activity单任务栈

BroadcastReceiver

即使用 `BroadcastReceiver` 广播监听

- 原理：在每个 `Activity` 里注册广播接收器（响应动作 = 关闭自身）；当需要退出 `App` 时 发送广播请求即可
- 具体实现

步骤1：自定义广播接收器

```
public class ExitAppReceiver extends BroadcastReceiver {
    private Activity activity;

    public ExitAppReceiver(Activity activity){
        this.activity = activity;
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        activity.finish();
    }
}
```

步骤2：在每个 `Activity` 里注册广播接收器（响应动作 = 关闭自身）

```
public class Activity extends AppCompatActivity {

private  ExitAppReceiver mExitAppReceiver;

// 1. 在onCreate () 中注册广播接收器
 protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mExitAppReceiver = new ExitAppReceiver(this);
        registerReceiver(mExitAppReceiver,new IntentFilter(BaseApplication.EXIT));
    }

// 1. 在onDestroy () 中注销广播接收器
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(mExitAppReceive);
    }
```

步骤3：当需要退出App时 发送广播请求

```
context.sendBroadcast(new Intent(BaseApplication.EXIT));
// 注：此处不能使用：System.exit(0);结束进程
// 原因：发送广播这个方法之后，不会等到广播接收器收到广播，程序就开始执行下一句System.exit(0)，然后就直接变成执行System.exit(0)的效果了。
```

- 优点 应用场景广泛：兼顾单 / 多任务栈 & 多启动模式的情况
- 缺点 实现复杂：需要在每个 `Activity` 里注册广播接收器
- 应用场景 任意情况下的一键退出 App，但无法终止 `App` 进程 所以该方法仅仅是在用户的角度来说 **“一键退出App”**

自己管理

- 原理：通过在`Application`子类中建立一个`Activity`链表：保存正在运行的`Activity`实例；当需要一键退出`App`时把链表内所有`Activity`实例逐个退出即可
- 具体使用

步骤1：在`BaseApplication`类的子类里建立`Activity`链表

Carson_BaseApplicaiton.java

```
public class Carson_BaseApplicaiton extends Application {

    // 此处采用 LinkedList作为容器，增删速度快
    public static LinkedList<Activity> activityLinkedList;


    @Override
    public void onCreate() {
        super.onCreate();

        activityLinkedList = new LinkedList<>();

        registerActivityLifecycleCallbacks(new ActivityLifecycleCallbacks() {
            @Override
            public void onActivityCreated(Activity activity, Bundle savedInstanceState) {
                Log.d(TAG, "onActivityCreated: " + activity.getLocalClassName());
                activityLinkedList.add(activity);
                // 在Activity启动时（onCreate()） 写入Activity实例到容器内
            }

            @Override
            public void onActivityDestroyed(Activity activity) {
                Log.d(TAG, "onActivityDestroyed: " + activity.getLocalClassName());
                activityLinkedList.remove(activity);
                // 在Activity结束时（Destroyed () ） 写出Activity实例
            }

            @Override
            public void onActivityStarted(Activity activity) {
            }

            @Override
            public void onActivityResumed(Activity activity) {
            }

            @Override
            public void onActivityPaused(Activity activity) {
            }

            @Override
            public void onActivityStopped(Activity activity) {
            }

            @Override
            public void onActivitySaveInstanceState(Activity activity, Bundle outState) {
```

```
            }


        });
    }

    public  void exitApp() {

        Log.d(TAG, "容器内的Activity列表如下 ");
        // 先打印当前容器内的Activity列表
        for (Activity activity : activityLinkedList) {
            Log.d(TAG, activity.getLocalClassName());
        }

        Log.d(TAG, "正逐步退出容器内所有Activity");

        // 逐个退出Activity
        for (Activity activity : activityLinkedList) {
            activity.finish();
        }

        //  结束进程
        // System.exit(0);
    }
}

// 记得在Manifest.xml中添加
<application
        android:name=".Carson_BaseApplicaiton"
        ....
</application>
```

步骤2：需要一键退出 `App` 时，获取该 `Applicaiton` 类对象 & 调用 `exitApp()`

```
                private Carson_BaseApplicaiton app;

                app = (Carson_BaseApplicaiton)getApplication();
                app.exitApp();
```

- 优点 应用场景广泛：兼顾单 / 多任务栈 & 多启动模式的情况
- 缺点 需要 `Activity` 经历正常的生命周期，即创建时调用 `onCreate ()` ，结束时调用 `onDestroy ()` 因为只有这样经历正常的生命周期才能将 `Activity` 正确写入 & 写出 容器内
- 应用场景 任意情况下的一键退出 `App` 实现

RxBus

- 原理：使用 `RxBus` 当作事件总线，在每个 `Activity` 里注册 `RxBus` 订阅（响应动作 = 关闭自身）；当需要退出App时 发送退出事件请求即可。
- 具体使用

步骤1：在每个 `Activity` 里注册 `RxBus` 订阅（响应动作 = 关闭自身）

```
public class Activity extends AppCompatActivity {
  private Disposable disposable;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity2);

        // 注册RxBus订阅
        disposable = RxBus.getInstance().toObservable(String.class)
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String s) throws Exception {
                        // 响应动作 = 关闭自身
                        if (s.equals("exit")){
                            finish();
                        }
                    }
                });
    }

// 注意一定要取消订阅
 @Override
    protected void onDestroy() {
        if (!disposable.isDisposed()){
              disposable.dispose();;
        }
}
```

步骤2：当需要退出App时 发送退出事件

```
        RxBus.getInstance().post("exit");
        System.exit(0);
```

- 优点 可与 `RxJava` & `RxBus` 相结合
- 缺点 实现复杂：`RxBus` 本身的实现难度 & 需要在每个 `Activity` 注册和取消订阅 `RxBus` 使用
- 应用场景 需要与 `RxJava` 结合使用时 若项目中没有用到 `RxJava` & `RxBus` 不建议使用
- 至此，一键结束当前 `App` 的所有 `Activity` 的 方法 讲解完毕。
- 注：上述方法仅仅只是结束当前 `App` 所有的 `Activity` （在用户的角度确实是退出了 `App`），但实际上该 `App` 的进程还未结束

一键结束当前 App 进程

主要采用 `Dalvik VM` 本地方法

- 作用 结束当前 `Activity` & 结束进程 即 在 （步骤1）结束当前 `App` 所有的 `Activity` 后，调用该方法即可一键退出 `App`（更多体现在结束进程上）
- 具体使用

```
// 方式1：android.os.Process.killProcess ()
  android.os.Process.killProcess(android.os.Process.myPid()) ;

// 方式2：System.exit()
// System.exit() = Java中结束进程的方法：关闭当前JVM虚拟机
  System.exit(0);

// System.exit(0)和System.exit(1)的区别
  // 1. System.exit(0)：正常退出；
  // 2. System.exit(1)：非正常退出，通常这种退出方式应该放在catch块中。
```

- 特别注意 假设场景：当前 `Activity` ≠ 当前任务栈最后1个 `Activity` 时，调用上述两个方法会出现什么情况呢？（即 `Activity1` - `Activity2` - `Activity3` （在 `Activity3` 调用上述两个方法））

答： 1. 结束 `Activity3` （当前 `Activity` ）& 结束进程 2. 再次重新开启进程 & 启动 `Activity1` 、 `Activity2`

即在 `Android` 中，调用上述 `Dalvik VM` 本地方法结果是： 1. 结束当前 `Activity` & 结束进程 2. 之后再重新开启进程 & 启动 之前除当前 `Activity` 外的已启动的 `Activity`

- 原因：** `Android` 中的 `ActivityManager` 时刻监听着进程**。一旦发现进程被非正常结束，它将会试图去重启这个进程。
- 应用场景 当任务栈只剩下当前 `Activity` （即退出了其余 `Activity` 后），调用即可退出该进程，即**在（步骤1）结束当前 App 所有的 Activity 后，调用该方法即可一键退出App（更多体现在结束进程上）** 注： 与 “在最后一个 `Activity` 调用 `finish ()` ”的区别： `finish ()` 不会结束进程，而上述两个方法会

10.34 startActivity(MainActivity.this，LoginActivity.class); LoginActivity配置的launchMode是何时解析的?

详细讲解

享学课堂移动开发课程：Framework专题 AMS部分

这道题想考察什么?

考察同学是否Activity的launchMode熟悉。

考生应该如何回答

启动模式相关源码都在ActivityStarter#startActivityUnchecked()， 我们对这个函数来分析一下： 注意下两个主要的属性 mLaunchFlags，mLaunchMode，后面的源码也是主要对着两个属性分析。 mLaunchFlags 包括启动的flag，比如FLAG_ACTIVITY_NEW_TASK,FLAG_ACTIVITY_CLEAR_TASK等，作用是表明如何去启动一个Activity，对栈的选择和处理。 mLaunchMode 表示启动模式，比如LAUNCH_SINGLE_INSTANCE,LAUNCH_SINGLE_TASK等。

初始化工作

```
private int startActivityUnchecked(final ActivityRecord r, ActivityRecord sourceRecord,
            IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
            int startFlags, boolean doResume, ActivityOptions options, TaskRecord inTask,
            ActivityRecord[] outActivity) {

    // 初始化  mLaunchFlags  mLaunchMode
        setInitialState(r, options, inTask, doResume, startFlags, sourceRecord,
voiceSession,
                voiceInteractor);
        // 一.计算 mLaunchFlags
        computeLaunchingTaskFlags();
        //赋值 mSourceTask
        computeSourceStack();
        mIntent.setFlags(mLaunchFlags);
        ... ... ...
```

我们看下 computeLaunchingTaskFlags 方法

```
   private void computeLaunchingTaskFlags() {

        ... ... ...

        // mSourceRecod 指的是 启动者，(注意区别于 被启动者 mStartActivity) mSourceRecord为null，表
示我们不是从一个Activity来启动的
        // 可能是从 一个Service 或者 ApplicationContext 来的
            if (mSourceRecord == null) {
                if ((mLaunchFlags & FLAG_ACTIVITY_NEW_TASK) == 0 && mInTask == null) {
                     //mInTask mSourceRecord 都为null，表示 不是从一个Activity 去启动另外一个
Activity，所以不管什么
            //都加上 FLAG_ACTIVITY_NEW_TASK
                    mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
                }
            } else if (mSourceRecord.launchMode == LAUNCH_SINGLE_INSTANCE) {
        // 如果 启动者 自己是 SINGLE_INSTANCE ，那么不管被启动的Activity是什么模式，mLaunchFlags 都加
上 FLAG_ACTIVITY_NEW_TASK，
        // 这个新 Activity 需要运行在 自己的 栈内
                mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
            } else if (isLaunchModeOneOf(LAUNCH_SINGLE_INSTANCE, LAUNCH_SINGLE_TASK)) {
        //如果launchMode是  SINGLE_INSTANCE 或者 SINGLE_TASK;  mLaunchFlags 添加
FLAG_ACTIVITY_NEW_TASK
                mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
            }
    }
```

如果启动者mSourceRecord=null，例如在一个Service启动了一个Activity，那么mLaunchFlags增加 FLAG_ACTIVITY_NEW_TASK 如果启动者mSourceRecord是一个 SingleInstance 类型的Activity，那么被启动者的 mLaunchFlags就会加上 FLAG_ACTIVITY_NEW_TASK 如果被启动者mStartActivity是SINGLE_INSTANCE或者 SINGLE_TASK 类型的Activity，被启动者的mLaunchFlags都会加上FLAG_ACTIVITY_NEW_TASK

getResuableIntentActivity

这一部分还是startActivityUnchecked方法的一个片段，紧接着第一部分的源码

```
        //还是在 startActivityUnchecked 里面
        ... ... ...
        // 1. 为SINGLE_INSTANCE查找可以复用的Activity
        // 2. 为 只有FLAG_ACTIVITY_NEW_TASK并且没有MULTI_TASK的
        // 3. SINGLE_TASK 查找可以添加的栈
        ActivityRecord reusedActivity = getReusableIntentActivity();
        ... ... ...
```

我们先跳出startActivityUnchecked方法，去查看下getResuableIntentActivity方法。

```
    private ActivityRecord getReusableIntentActivity() {

    //putIntoExistingTask为true的条件
    //1.当启动模式为SingleInstance;
    //2.当启动模式为SingleTask;
    //3.使用了Intent.FLAG_ACTIVITY_NEW_TASK标签，并且没有使用FLAG_ACTIVITY_MULTIPLE_TASK标签
        boolean putIntoExistingTask = ((mLaunchFlags & FLAG_ACTIVITY_NEW_TASK) != 0 &&
                (mLaunchFlags & FLAG_ACTIVITY_MULTIPLE_TASK) == 0)
                || isLaunchModeOneOf(LAUNCH_SINGLE_INSTANCE, LAUNCH_SINGLE_TASK);

        putIntoExistingTask &= mInTask == null && mStartActivity.resultTo == null;
        ActivityRecord intentActivity = null;

        if (mOptions != null && mOptions.getLaunchTaskId() != -1) {

            ... ... ...

        } else if (putIntoExistingTask) { //putIntoExistingTask为true时的策略
            if (LAUNCH_SINGLE_INSTANCE == mLaunchMode) {

        // SINGLE_INSTANCE 模式下去寻找,这里目的是findActivityRecord
               intentActivity = mSupervisor.findActivityLocked(mIntent, mStartActivity.info,
                       mStartActivity.isActivityTypeHome());
            } else if ((mLaunchFlags & FLAG_ACTIVITY_LAUNCH_ADJACENT) != 0) {
                ... ...
            } else {
                //这里要区别于singleInstance调用的方法！！！这里目的是findTaskRecord
                // Otherwise find the best task to put the activity in.
                intentActivity = mSupervisor.findTaskLocked(mStartActivity,
mPreferredDisplayId);
            }
        }
        return intentActivity;
    }
```

mLaunchMode是SingleInstance时，走mSupervisor.findActivityLocked;剩余情况下，比如我们的mStartActivity是一个standard模式的Activity，且只加上了FLAG_ACTIVITY_NEW_TASK的flag,会执行mSupervisor.findTaskLocked。

最合适的可重用栈

```
   void findTaskLocked(ActivityRecord target, FindTaskResult result) {
        ... ... ...
```

```
        // 注意这是倒序遍历 mTaskHistory
        for (int taskNdx = mTaskHistory.size() - 1; taskNdx >= 0; --taskNdx) {

            ... ... ...

            } else if (!isDocument && !taskIsDocument
                    && result.r == null && task.rootAffinity != null) {
                //检查 是不是 相同的  taskAffinity
                if (task.rootAffinity.equals(target.taskAffinity)) {
                    //当我们找到taskAffinity符合的栈之后，并没有立马break，而是继续去寻找，说明task的
index越小，表示更适合
                    result.r = r;
                    result.matchedByRootAffinity = true;
                }
            } else if (DEBUG_TASKS) Slog.d(TAG_TASKS, "Not a match: " + task);
        }
    }
```

最合适的栈 需要满足两个条件 1.Activity的taskAffinity和我们的task的rootAffinity相等 2.不同的task的rootAffinity可能是相等的，倒序遍历找到index最小的，也是最合适的

reusedActivity的处理

我们接着分析startActivityUnchecked代码

```
        // 三. 利用可以 复用的Activity 或者 复用栈
        if (reusedActivity != null) {

            ... ... ...

            // 如果是 SingleInstance 或者 SingleTask 或者  含有 FLAG_ACTIVITY_CLEAR_TOP 标识
            //我们可以判断出来  SingleInstance 或者 SingleTask  含有 FLAG_ACTIVITY_CLEAR_TOP 的效
果
            if ((mLaunchFlags & FLAG_ACTIVITY_CLEAR_TOP) != 0
                    || isDocumentLaunchesIntoExisting(mLaunchFlags)
                    || isLaunchModeOneOf(LAUNCH_SINGLE_INSTANCE, LAUNCH_SINGLE_TASK)) {

                //拿 reuseActivity 的栈
                final TaskRecord task = reusedActivity.getTask();

        // 比如 SingleTask  移除要启动的Activity之前的所有Activity
                final ActivityRecord top =
task.performClearTaskForReuseLocked(mStartActivity,
                        mLaunchFlags);

                if (reusedActivity.getTask() == null) {
                    reusedActivity.setTask(task);
                }

                if (top != null) {
                    if (top.frontOfTask) {
                        // Activity aliases may mean we use different intents for the top
activity,
```

```
                        // so make sure the task now has the identity of the new intent.
                        top.getTask().setIntent(mStartActivity);
                    }
                //这里是 SingleInstance 或者 SingleTask ，会执行onNewIntent
                    deliverNewIntent(top);
                }
            }

            ... ... ...

            //启动者和被启动者是同一个
            if ((mStartFlags & START_FLAG_ONLY_IF_NEEDED) != 0) {
                // We don't need to start a new activity, and the client said not to do
anything
                // if that is the case, so this is it!  And for paranoia, make sure we have
                // correctly resumed the top activity.
                resumeTargetStackIfNeeded();
                return START_RETURN_INTENT_TO_CALLER;
            }

            if (reusedActivity != null) {
                //这里会去判断几种情况 singleTask singleInstance 和 singleTop
                setTaskFromIntentActivity(reusedActivity);

                if (!mAddingToTask && mReuseTask == null) {

                    //singleInstance singleTask 都会走这里
                    //1.比如要启动的Activity是singleTask，且刚好在reusedActivity的栈内
                    //2.或者一个singleInstance模式的Activity再次被启动
                    resumeTargetStackIfNeeded();
                    if (outActivity != null && outActivity.length > 0) {
                        outActivity[0] = reusedActivity;
                    }
                    return mMovedToFront ? START_TASK_TO_FRONT : START_DELIVERED_TO_TOP;
                }
            }
        }
```

继续来查看下setTaskFromIntentActivity这个方法

```
private void setTaskFromIntentActivity(ActivityRecord intentActivity) {

        if ((mLaunchFlags & (FLAG_ACTIVITY_NEW_TASK | FLAG_ACTIVITY_CLEAR_TASK))
                == (FLAG_ACTIVITY_NEW_TASK | FLAG_ACTIVITY_CLEAR_TASK)) {

            //如果是 FLAG_ACTIVITY_NEW_TASK + FLAG_ACTIVITY_CLEAR_TASK
            final TaskRecord task = intentActivity.getTask();
            //清空task
            task.performClearTaskLocked();
            mReuseTask = task;
            mReuseTask.setIntent(mStartActivity);
        } else if ((mLaunchFlags & FLAG_ACTIVITY_CLEAR_TOP) != 0
                || isLaunchModeOneOf(LAUNCH_SINGLE_INSTANCE, LAUNCH_SINGLE_TASK)) {
```

```
            //如果是 SingleInstance 或者 SingleTask 走这里，清空栈内要启动的Activity之前的所有
Activity们。
            ActivityRecord top =
intentActivity.getTask().performClearTaskLocked(mStartActivity,
                    mLaunchFlags);
            //如果top == null 继续走，不为null，就结束了这个方法
            if (top == null) {
                ... ... ...
            }
        } else if
(mStartActivity.realActivity.equals(intentActivity.getTask().realActivity)) {
            // 判断是否是 SingleTop 模式
            // 这种情况如何复现？ SingleTop + FLAG_ACTIVITY_NEW_TASK + taskAffinity。
            // FLAG_ACTIVITY_NEW_TASK + taskAffinity去指定一个特定存在的栈，且栈顶是我们要启动的
singleTop模式的activity
            if (((mLaunchFlags & FLAG_ACTIVITY_SINGLE_TOP) != 0
                        || LAUNCH_SINGLE_TOP == mLaunchMode)
                    && intentActivity.realActivity.equals(mStartActivity.realActivity)) {
                if (intentActivity.frontOfTask) {
                    intentActivity.getTask().setIntent(mStartActivity);
                }
                deliverNewIntent(intentActivity);
            } else if (!intentActivity.getTask().isSameIntentFilter(mStartActivity)) {
                ... ... ...
            }
        } else if ((mLaunchFlags & FLAG_ACTIVITY_RESET_TASK_IF_NEEDED) == 0) {
            ... ... ...
        } else if (!intentActivity.getTask().rootWasReset) {
            ... ... ...
        }
    }
```

前提是reusedActivity不为null，看两种情况： 1.如果是SingleTask、SingleInstance模式的Activity，则调用performClearTaskLocked方法，把要启动的Activity之前的所有Activity都清除掉。 2.reusedActivity的启动模式恰好是SingleTop，且也是我们需要启动的Activity，执行 deliverNewIntent。 上述的两种情况能够满足下面的if判断 !mAddingToTask && mReuseTask == null ，然后return结束。

判断SingleTop模式

继续看 startActivityUnchecked 后面的代码，这一部分是针对SingleTop模式的处理。

```
        ... ... ...
        //下面这段英文解释的很好（SingleTop模式），当我们要启动的Actiivty恰好是当前栈顶的Activity，检查
是否只需要被启动一次
        // If the activity being launched is the same as the one currently at the top, then
        // we need to check if it should only be launched once.
        final ActivityStack topStack = mSupervisor.mFocusedStack;
        final ActivityRecord topFocused = topStack.getTopActivity();
        final ActivityRecord top = topStack.topRunningNonDelayedActivityLocked(mNotTop);
        final boolean dontStart = top != null && mStartActivity.resultTo == null
                && top.realActivity.equals(mStartActivity.realActivity)
                && top.userId == mStartActivity.userId
```

```
                && top.app != null && top.app.thread != null
                && ((mLaunchFlags & FLAG_ACTIVITY_SINGLE_TOP) != 0
                || isLaunchModeOneOf(LAUNCH_SINGLE_TOP, LAUNCH_SINGLE_TASK));
        // SINGLE_TOP  SINGLE_TASK,要启动的Activiy恰好在栈顶
        // dontStart为true，表示不会去启动新的Activity，复用栈顶的Activity
        if (dontStart) {
            // For paranoia, make sure we have correctly resumed the top activity.
            topStack.mLastPausedActivity = null;
            if (mDoResume) {
                // resume  Activity
                mSupervisor.resumeFocusedStackTopActivityLocked();
            }
            ActivityOptions.abort(mOptions);
            if ((mStartFlags & START_FLAG_ONLY_IF_NEEDED) != 0) {
                // We don't need to start a new activity, and the client said not to do
                // anything if that is the case, so this is it!
                return START_RETURN_INTENT_TO_CALLER;
            }

            deliverNewIntent(top);

            // Don't use mStartActivity.task to show the toast. We're not starting a new
activity
            // but reusing 'top'. Fields in mStartActivity may not be fully initialized.
            mSupervisor.handleNonResizableTaskIfNeeded(top.getTask(),
preferredWindowingMode,
                    preferredLaunchDisplayId, topStack);

            return START_DELIVERED_TO_TOP;
        }
```

栈的复用和新建

继续分析startActivityUnChecked方法的最后一部分，这部分主要是有关于栈是否需要新建。

```
    ... ... ...
        //必要条件是有FLAG_ACTIVITY_NEW_TASK
        if (mStartActivity.resultTo == null && mInTask == null && !mAddingToTask
                && (mLaunchFlags & FLAG_ACTIVITY_NEW_TASK) != 0) {
            //要建立新栈或者使用已经存在的栈，FLAG_ACTIVITY_NEW_TASK是必要条件
            newTask = true;
            result = setTaskFromReuseOrCreateNewTask(taskToAffiliate, topStack);
        } else if (mSourceRecord != null) {
            //把被启动的mStartActivity放在启动者mSourceRecord所在的栈上
            result = setTaskFromSourceRecord();
        }
     ... ... ...
```

setTaskFromReuseOrCreateNewTask 方法

```
private int setTaskFromReuseOrCreateNewTask(
            TaskRecord taskToAffiliate, ActivityStack topStack) {
```

```
        mTargetStack = computeStackFocus(mStartActivity, true, mLaunchFlags, mOptions);

        if (mReuseTask == null) {
            //新建一个栈
            final TaskRecord task = mTargetStack.createTaskRecord(
                    mSupervisor.getNextTaskIdForUserLocked(mStartActivity.userId),
                    mNewTaskInfo != null ? mNewTaskInfo : mStartActivity.info,
                    mNewTaskIntent != null ? mNewTaskIntent : mIntent, mVoiceSession,
                    mVoiceInteractor, !mLaunchTaskBehind /* toTop */, mStartActivity,
mSourceRecord,
                    mOptions);
            addOrReparentStartingActivity(task, "setTaskFromReuseOrCreateNewTask -
mReuseTask");
            updateBounds(mStartActivity.getTask(), mLaunchParams.mBounds);

        } else {
            //用旧栈
            addOrReparentStartingActivity(mReuseTask, "setTaskFromReuseOrCreateNewTask");
        }
        ... ... ...
```

总结

1. 新建栈或者复用栈的必要条件是FLAG_ACTIVITY_NEW_TASK。SingleTask,SingleInstance会为mLaunchFlags自动添加FLAG_ACTIVITY_NEW_TASK。也就是说他们都有存在不使用当前栈的可能。
2. 新建栈或者复用已经存在栈的充分必要条件是什么?
    1. FLAG_ACTIVITY_NEW_TASK + taskAffinity(taskAffinity必须与当前显示的栈的rootAffinity不相同，taskAffinity默认是包名)。
    2. FLAG_ACTIVITY_NEW_TASK + FLAG_ACTIVITY_MULTIPLE_TASK 这是必定会新建一个栈的。
3. SingleTask可以理解成FLAG_ACTIVITY_NEW_TASK + FLAG_ACTIVITY_CLEAR_TOP + (taskAffinity == 该应用包名)。补充一下，FLAG_ACTIVITY_CLEAR_TOP的作用是啥？比如我们要启动的SingleTask模式的Activity已经在栈内，且不在栈的头部，会把之前的Activity全部出栈。
4. SingleInstance比较特殊，首先SingleInstance模式的Activity会被加上FLAG_ACTIVITY_NEW_TASK，这一点和SingleTask一样。特殊的地方其一在于如果启动过了，会去遍历找相等的Activity，查找过程不一样。而不像SingleTask是去找"合适的"栈，根据taskAffinity来查找。其二在于SingleInstance一个栈只能存放一个Activity，能做到这个的原因是我们在根据taskAffinity找到合适的栈的时候，如果发现是SingleInstance模式Activity的栈，直接忽略。

10.35 在清单文件中配置的receiver，系统是何时会注册此广播接受者的?

这道题想考察什么?

考察同学是否对清单文件的解析熟悉，以及PMS相关的知识的理解。

考生应该如何回答

清单文件Manifest 一般被解析的时间有两处：1）手机启动的时候，pkms会按照 core app, system app,other app 的优先级方案扫码APK，解析AndroidManifest.xml文件，最后得到清单文件中的标签，然后存储到settings对象中并持久化；2）在动态部署apk的时候，apk的安装中也必然会调用PackageManagerService来解析apk包中的AndroidManifest文件。所以系统注册广播接收者的时间基本上就是这两个时段，那么具体的细节大家可以参考后面的分析。

首先apk是由PMS解析的，下面将介绍PMS如何解析APK：

1）android系统启动之后会解析系统特定目录下的apk文件，并执行解析； 2）解析Manifest流程：Zygote进程 –> SystemServer进程 –> PackgeManagerService服务 –> scanDirLI方法 –> scanPackageLI方法 –> PackageParser.parserPackage方法； 3）解析完成Manifest之后会将apk的Manifest信息保存在Settings对象中并持久化，删除软件时，信息会被删除，新安装的apk会重复调用scanDirLi。

我们一起来分析一下具体的代码的调度流程：

scanPackageLI()函数说明

PMS 的构造函数中会通过 scanDirTracedLI() 对各个指定的目录进行扫描：

```
    private void scanDirTracedLI(File dir, final int parseFlags, int scanFlags, long
currentTime) {
        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "scanDir [" + dir.getAbsolutePath() +
"]");
        try {
            scanDirLI(dir, parseFlags, scanFlags, currentTime);
        } finally {
            Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
        }
    }
```

```
    private void scanDirLI(File dir, int parseFlags, int scanFlags, long currentTime) {
        final File[] files = dir.listFiles();
        if (ArrayUtils.isEmpty(files)) { //不能是空目录
            Log.d(TAG, "No files in app dir " + dir);
            return;
        }

        if (DEBUG_PACKAGE_SCANNING) {
            Log.d(TAG, "Scanning app dir " + dir + " scanFlags=" + scanFlags
                    + " flags=0x" + Integer.toHexString(parseFlags));
        }
        ParallelPackageParser parallelPackageParser = new ParallelPackageParser(
                mSeparateProcesses, mOnlyCore, mMetrics, mCacheDir,
                mParallelPackageParserCallback);

        // Submit files for parsing in parallel
        int fileCount = 0;
        for (File file : files) {
            final boolean isPackage = (isApkFile(file) || file.isDirectory()) //判断是否为应用
文件
```

```
                    && !PackageInstallerService.isStageName(file.getName());
            if (!isPackage) {
                // Ignore entries which are not packages
                continue; //忽略非应用文件
            }

            parallelPackageParser.submit(file, parseFlags);
            fileCount++;
        }

        // Process results one by one
        for (; fileCount > 0; fileCount--) {
            ParallelPackageParser.ParseResult parseResult = parallelPackageParser.take();
            Throwable throwable = parseResult.throwable;
            int errorCode = PackageManager.INSTALL_SUCCEEDED;

            if (throwable == null) {
                // Static shared libraries have synthetic package names
                if (parseResult.pkg.applicationInfo.isStaticSharedLibrary()) {
                    renameStaticSharedLibraryPackage(parseResult.pkg);
                }
                try {
                    if (errorCode == PackageManager.INSTALL_SUCCEEDED) {
                        scanPackageLI(parseResult.pkg, parseResult.scanFile, parseFlags,
scanFlags,
                                currentTime, null); //最终会调用到这里
                    }
                } catch (PackageManagerException e) {
                    errorCode = e.error;
                    Slog.w(TAG, "Failed to scan " + parseResult.scanFile + ": " +
e.getMessage());
                }
            } else if (throwable instanceof PackageParser.PackageParserException) {
                PackageParser.PackageParserException e =
(PackageParser.PackageParserException)
                        throwable;
                errorCode = e.error;
                Slog.w(TAG, "Failed to parse " + parseResult.scanFile + ": " +
e.getMessage());
            } else {
                throw new IllegalStateException("Unexpected exception occurred while parsing
"
                        + parseResult.scanFile, throwable);
            }

            // Delete invalid userdata apps
            if ((parseFlags & PackageParser.PARSE_IS_SYSTEM) == 0 &&
                    errorCode == PackageManager.INSTALL_FAILED_INVALID_APK) {
                logCriticalInfo(Log.WARN,
                        "Deleting invalid package at " + parseResult.scanFile);
                removeCodePathLI(parseResult.scanFile);
            }
        }
```

```
        parallelPackageParser.close();
    }
```

最后会执行到scanPackageLI()，当然有些不同的android api版本，代码是有不同的，但是整体大方向是一致的，那么我们一起来看一下scanPackageLI的执行代码：

```
private AndroidPackage scanPackageLI(File scanFile, int parseFlags, int scanFlags,
            long currentTime, UserHandle user) throws PackageManagerException {
        if (DEBUG_INSTALL) Slog.d(TAG, "Parsing: " + scanFile);

        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "parsePackage");
        final ParsedPackage parsedPackage;
        try (PackageParser2 pp = new PackageParser2(mSeparateProcesses, mOnlyCore, mMetrics,
null,
                mPackageParserCallback)) {
            parsedPackage = pp.parsePackage(scanFile, parseFlags, false);//代码1
        } catch (PackageParserException e) {
            throw PackageManagerException.from(e);
        } finally {
            Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
        }

        // Static shared libraries have synthetic package names
        if (parsedPackage.isStaticSharedLibrary()) {
            renameStaticSharedLibraryPackage(parsedPackage);
        }

        return addForInitLI(parsedPackage, parseFlags, scanFlags, currentTime, user);
    }
```

上面的代码会将主要的操作转交给PackageParser2#parsePackage 进行，具体的逻辑如下：

parsePackage()函数说明

code 路径：frameworks/base/core/java/android/content/pm/PackageParser2.java

```
 public ParsedPackage parsePackage(File packageFile, int flags, boolean useCaches)
            throws PackageParserException {
        if (useCaches && mCacher != null) {
            ParsedPackage parsed = mCacher.getCachedResult(packageFile, flags);
            if (parsed != null) {
                return parsed;
            }
        }

        long parseTime = LOG_PARSE_TIMINGS ? SystemClock.uptimeMillis() : 0;
        ParseInput input = mSharedResult.get().reset();
        ParseResult<ParsingPackage> result = parsingUtils.parsePackage(input, packageFile,
flags);  //code 1
        if (result.isError()) {
            throw new PackageParserException(result.getErrorCode(),
result.getErrorMessage(),
```

```
                    result.getException());
        }

        ParsedPackage parsed = (ParsedPackage) result.getResult().hideAsParsed();

        long cacheTime = LOG_PARSE_TIMINGS ? SystemClock.uptimeMillis() : 0;
        if (mCacher != null) {
            mCacher.cacheResult(packageFile, flags, parsed);
        }
        if (LOG_PARSE_TIMINGS) {
            parseTime = cacheTime - parseTime;
            cacheTime = SystemClock.uptimeMillis() - cacheTime;
            if (parseTime + cacheTime > LOG_PARSE_TIMINGS_THRESHOLD_MS) {
                Slog.i(TAG, "Parse times for '" + packageFile + "': parse=" + parseTime
                        + "ms, update_cache=" + cacheTime + " ms");
            }
        }

        return parsed;
    }
```

上面的代码会调用带code 1处，然后在code1处执行的代码就非常简单了，如下所示：

```
public ParseResult<ParsingPackage> parsePackage(ParseInput input, File packageFile,
            int flags)
            throws PackageParserException {
        if (packageFile.isDirectory()) {
            return parseClusterPackage(input, packageFile, flags);
        } else {
            return parseMonolithicPackage(input, packageFile, flags);
        }
}
```

上面代码中，如果参数是packageFile 是一个目录，就会执行parseClusterPackage()，否则执行parseMonolithicPackage() 来处理。对于关联的多个apk 会放到一个目录下，来查看parseClusterPackage()：

```
private ParseResult<ParsingPackage> parseClusterPackage(ParseInput input, File packageDir,
            int flags) {
//获取应用目录的PackageLite 对象，这个对象中分开保存了目录下的核心应用名称以及其他非核心应用的名称
    ParseResult<PackageParser.PackageLite> liteResult =
            ApkLiteParseUtils.parseClusterPackageLite(input, packageDir, 0);
    if (liteResult.isError()) {
        return input.error(liteResult);
    }

    final PackageParser.PackageLite lite = liteResult.getResult();
    if (mOnlyCoreApps && !lite.coreApp) {
        return input.error(INSTALL_PARSE_FAILED_ONLY_COREAPP_ALLOWED,
                "Not a coreApp: " + packageDir);
    }

    // Build the split dependency tree.
```

```
    SparseArray<int[]> splitDependencies = null;
    final SplitAssetLoader assetLoader;
    if (lite.isolatedSplits && !ArrayUtils.isEmpty(lite.splitNames)) {
        try {
            splitDependencies =
SplitAssetDependencyLoader.createDependenciesFromPackage(lite);
            assetLoader = new SplitAssetDependencyLoader(lite, splitDependencies, flags);
        } catch (SplitAssetDependencyLoader.IllegalDependencyException e) {
            return input.error(INSTALL_PARSE_FAILED_BAD_MANIFEST, e.getMessage());
        }
    } else {
        assetLoader = new DefaultSplitAssetLoader(lite, flags);
    }

    try {
        //需要AssetManager 对象
        final AssetManager assets = assetLoader.getBaseAssetManager();
        final File baseApk = new File(lite.baseCodePath);
        ParseResult<ParsingPackage> result = parseBaseApk(input, baseApk,
                lite.codePath, assets, flags);
        if (result.isError()) {
            return input.error(result);
        }
        //对于apk 进行分析，得到Package 对象
        ParsingPackage pkg = result.getResult();
        if (!ArrayUtils.isEmpty(lite.splitNames)) {
            pkg.asSplit(
                    lite.splitNames,
                    lite.splitCodePaths,
                    lite.splitRevisionCodes,
                    splitDependencies
            );
            final int num = lite.splitNames.length;

            for (int i = 0; i < num; i++) {
                final AssetManager splitAssets = assetLoader.getSplitAssetManager(i);
                parseSplitApk(input, pkg, i, splitAssets, flags);
            }
        }

        pkg.setUse32BitAbi(lite.use32bitAbi);
        return input.success(pkg);
    } catch (PackageParserException e) {
        return input.error(INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION,
                "Failed to load assets: " + lite.baseCodePath, e);
    } finally {
        IoUtils.closeQuietly(assetLoader);
    }
}
```

parseClusterPackage() 方法中先执行parseClusterPackageLite() 方法对目录下的apk 文件进行初步分析，主要是区分出核心应用和非核心应用。核心应用只有一个，非核心应用可以没有或者多个，非核心应用的作用是用来保存资源和代码。 接下来调用parseBaseApk() 方法对核心应用进行分析，并生成Package 对象，对非核心的应用调用parseSpliteApk() 方法来分析，分析的结果会放到前面的 Package 对象中。 parseBaseApk() 方法实际上是主要是分析AndroidManifest.xml 文件：

```
private ParseResult<ParsingPackage> parseBaseApk(ParseInput input, File apkFile,
        String codePath, AssetManager assets, int flags) {
    final String apkPath = apkFile.getAbsolutePath();

    ...

    final int cookie = assets.findCookieForPath(apkPath);
    ...
    try (XmlResourceParser parser = assets.openXmlResourceParser(cookie,
            PackageParser.ANDROID_MANIFEST_FILENAME)) { // 1
        final Resources res = new Resources(assets, mDisplayMetrics, null);

        ParseResult<ParsingPackage> result = parseBaseApk(input, apkPath, codePath, res,
                parser, flags);
        ...
        final ParsingPackage pkg = result.getResult();
        ..
        pkg.setVolumeUuid(volumeUuid);
        ...
        return input.success(pkg);
    } catch (Exception e) {
        return input.error(INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION,
                "Failed to read manifest from " + apkPath, e);
    }
}
```

在代码1处主要分析XmlResourceParser 对象，也就是获得一个 XML 资源解析对象，该对象解析的是 APK 中的AndroidManifest.xml 文件

```
parser = assets.openXmlResourceParser(cookie, ANDROID_MANIFEST_FILENAME);
```

其中这个地方的ANDROID_MANIFEST_FILENAME 为：

```
private static final String ANDROID_MANIFEST_FILENAME = "AndroidManifest.xml";
```

继续分析，最终会执行到

```
final Package pkg = parseBaseApk(apkPath, res, parser, flags, outError);
```

```
private ParseResult<ParsingPackage> parseBaseApk(ParseInput input, String apkPath,
        String codePath, Resources res, XmlResourceParser parser, int flags)
        throws XmlPullParserException, IOException, PackageParserException {
    final String splitName;
    final String pkgName;
```

```
    ParseResult<Pair<String, String>> packageSplitResult =
            ApkLiteParseUtils.parsePackageSplitNames(input, parser, parser);
    if (packageSplitResult.isError()) {
        return input.error(packageSplitResult);
    }

    Pair<String, String> packageSplit = packageSplitResult.getResult();
    pkgName = packageSplit.first;
    splitName = packageSplit.second;

    if (!TextUtils.isEmpty(splitName)) {
        return input.error(
                PackageManager.INSTALL_PARSE_FAILED_BAD_PACKAGE_NAME,
                "Expected base APK, but found split " + splitName
        );
    }

    final TypedArray manifestArray = res.obtainAttributes(parser,
R.styleable.AndroidManifest);
    try {
        final boolean isCoreApp =
                parser.getAttributeBooleanValue(null, "coreApp", false);
        final ParsingPackage pkg = mCallback.startParsingPackage(
                pkgName, apkPath, codePath, manifestArray, isCoreApp);
        final ParseResult<ParsingPackage> result =
                parseBaseApkTags(input, pkg, manifestArray, res, parser, flags);//1
        if (result.isError()) {
            return result;
        }

        return input.success(pkg);
    } finally {
        manifestArray.recycle();
    }
}
```

上面的代码1 处会执行parseBaseApkTags 函数。

```
private ParseResult<ParsingPackage> parseBaseApkTags(ParseInput input, ParsingPackage pkg,
        TypedArray sa, Resources res, XmlResourceParser parser, int flags)
        throws XmlPullParserException, IOException {
    ...
    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG
            || parser.getDepth() > depth)) {
        ...

        // TODO(b/135203078): Convert to instance methods to share variables
        // <application> has special logic, so it's handled outside the general method
        if (PackageParser.TAG_APPLICATION.equals(tagName)) {
            if (foundApp) {
                if (PackageParser.RIGID_PARSER) {
```

```
                    result = input.error("<manifest> has more than one <application>");
                } else {
                    Slog.w(TAG, "<manifest> has more than one <application>");
                    result = input.success(null);
                }
            } else {
                foundApp = true;
                result = parseBaseApplication(input, pkg, res, parser, flags);//1
            }
        } else {
            result = parseBaseApkTag(tagName, input, pkg, res, parser, flags);//2
        }

        if (result.isError()) {
            return input.error(result);
        }
    }
    ...
    convertNewPermissions(pkg);

    convertSplitPermissions(pkg);

    ...

    return input.success(pkg);
}
```

到这个地方就找到了我们需要的内容，广播就是在这个里面解析的，其他的内容就暂时不分析了,重点分析代码1 和代码2处的逻辑。

parseBaseApplication()

大家看parseBaseApplication中的下面的代码块，大家不难发现，正在解析我们要的receiver属性，解析出来后会讲这个属性存于pkg信息里面进行保存。

```
switch (tagName) {
    case "activity":
        isActivity = true;
        // fall-through
    case "receiver":
        ParseResult<ParsedActivity> activityResult =
                ParsedActivityUtils.parseActivityOrReceiver(mSeparateProcesses, pkg,
                        res, parser, flags, PackageParser.sUseRoundIcon, input);

        if (activityResult.isSuccess()) {
            ParsedActivity activity = activityResult.getResult();
            if (isActivity) {
                hasActivityOrder |= (activity.getOrder() != 0);
                pkg.addActivity(activity);
            } else {
                hasReceiverOrder |= (activity.getOrder() != 0);
                pkg.addReceiver(activity);
            }
```

```
        }

        result = activityResult;
        break;
    case "service":
        ParseResult<ParsedService> serviceResult =
                ParsedServiceUtils.parseService(mSeparateProcesses, pkg, res, parser,
                        flags, PackageParser.sUseRoundIcon, input);
        if (serviceResult.isSuccess()) {
            ParsedService service = serviceResult.getResult();
            hasServiceOrder |= (service.getOrder() != 0);
            pkg.addService(service);
        }

        result = serviceResult;
        break;
    case "provider":
        ParseResult<ParsedProvider> providerResult =
                ParsedProviderUtils.parseProvider(mSeparateProcesses, pkg, res, parser,
                        flags, PackageParser.sUseRoundIcon, input);
        if (providerResult.isSuccess()) {
            pkg.addProvider(providerResult.getResult());
        }

        result = providerResult;
        break;
    case "activity-alias":
        activityResult = ParsedActivityUtils.parseActivityAlias(pkg, res,
                parser, PackageParser.sUseRoundIcon, input);
        if (activityResult.isSuccess()) {
            ParsedActivity activity = activityResult.getResult();
            hasActivityOrder |= (activity.getOrder() != 0);
            pkg.addActivity(activity);
        }

        result = activityResult;
        break;
    default:
        result = parseBaseAppChildTag(input, tagName, pkg, res, parser, flags);
        break;
}
```

receiver的注册如下：

```
            else if (tagName.equals("receiver")) {
                Activity a = parseActivity(owner, res, parser, flags, outError, cachedArgs,
                        true, false);
                if (a == null) {
                    mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                    return false;
                }
                owner.receivers.add(a);
             }
```

到这个地方为止整个广播的静态注册流程我们就分析完了。

总结

清单文件的解析过程，一般是由PKMS来完成，触发PKMS的执行的分为两个部分：1）在系统启动的过程中，会启动SystemServer 进程，而SystemServer进程会启动各种服务，这些服务包括PKMS，在启动PKMS的时候就会扫码apk安装路径下面的apk，然后解析AndroidManifest文件，并做持久化存储；2）app 安装的过程中，也会触发PKMS对apk进行检测，调用类似的流程解析 AndroidManifest文件。Receiver的解析，都只是整个AndroidManifest解析过程中的一个环节。

10.36 如何通过WindowManager添加Window(代码实现)?

详细讲解

享学课堂移动开发课程：Framework专题 WMS部分

这道题想考察什么?

考察同学是否知道窗口创建方法。

考生应该如何回答

1.设置权限

```
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
```

```
// 设置权限
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    if (!Settings.canDrawOverlays(this)) {
        Intent intent = new Intent(Settings.ACTION_MANAGE_OVERLAY_PERMISSION,
Uri.parse("package:" + getPackageName()));
        startActivityForResult(intent, OVERLAY_PERMISSION_CODE);
    }
}
```

2.创建窗口

```
// 获取 WindowManager
WindowManager wm = (WindowManager) getApplicationContext().getSystemService(WINDOW_SERVICE);
// 获取需要添加的View
View view = View.inflate(MainActivity.this, R.layout.item, null);
WindowManager.LayoutParams params = new WindowManager.LayoutParams();
params.type = WindowManager.LayoutParams.TYPE_APPLICATION_OVERLAY;
// 设置不拦截焦点
params.flags = WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE |
WindowManager.LayoutParams.FLAG_NOT_TOUCH_MODAL;
params.width = (int) (60 * getResources().getDisplayMetrics().density);
params.height = (int) (60 * getResources().getDisplayMetrics().density);
// 且设置坐标系 左上角
params.gravity = Gravity.LEFT | Gravity.TOP;
params.format = PixelFormat.TRANSPARENT;
int width = wm.getDefaultDisplay().getWidth();
int height = wm.getDefaultDisplay().getHeight();
params.y = height / 2 - params.height / 2;
wm.addView(view, params);
```

小结

在添加Window的过程中，有三个比较关键的节点：

关键点1：获取WindowManagerService服务的代理对象，不过对于Application而言，获取到的其实是一个封装过的代理对象，一个WindowManagerImpl实例，Application 的getSystemService() 源码其实是在ContextImpl中，有兴趣的可以看看APP启动时Context的创建过程。

关键点2：设置Window的各种属性，他们决定了window显示的内容 view，另外就是布局相关的熟悉，其中最关键的一点是WindowManager.LayoutParams，主要看一个type参数，这个参数决定了窗口的类型，这里我们定义成一个系统正式统一开发者使用悬浮窗的类型窗口，属于系统窗口。

关键点3：利用WindowManagerImpl的addView方法添加View到WMS，然后由wms去进行显示处理。更多相关的细节大家可以去阅读 UI相关的章节的内容。

10.37 为什么Dialog不能用Application的Context?

详细讲解

享学课堂移动开发课程：Framework专题 WMS部分

这道题想考察什么?

1. 是否了解Dialog的运行机制?

考察的知识点

1. Window、WindowManager、WindowMangerService之间的关系

2. Dialog使用Activity的Token的原因

考生应该如何回答

1. 首先我们看一下如果用Application的Context出现什么状况?

```
 Caused by: android.view.WindowManager$BadTokenException: Unable to add window -- token null
is not valid; is your activity running?
        at android.view.ViewRootImpl.setView(ViewRootImpl.java:907)
        at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:387)
        at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:95)
        at android.app.Dialog.show(Dialog.java:342)
```

- 从上面代码我们可以得出，不能把Dialog添加到Window上，因为Token为空是非法的，末尾一句，“is your activity running?”。我们可以猜测，Dialog需要一个合法的Token，而且这个在Activity运行后是可以获得的。

1. Window、WM、WMS、Token的概念?

- Window：Window是窗口的意思，对应屏幕上的一块显示区域，它的实现类是PhoneWindow。 Window有一个属性值Type（应用窗口、子窗口、系统窗口）；应用窗口有Activity，子窗口有PopupWindow、ContextMenu、OptionMenu；系统窗口有Toast和系统警告提示框。
- WM：WM全称是WindowManager，WindowManager是在应用与Window之间管理接口，像窗口顺序、消息等。
- WMS：WMS全称是WindowManagerService，WindowManagerService是窗口的管理者，它负责窗口的启动、添加和删除。另外窗口的大小和层级也是由 **WMS** 进行管理的。
- Token：这里讲到的Token主要是指窗口令牌（Window Token），是一种特殊的Binder令牌，WMS用它唯一标识系统中的一个窗口

1. Dialog的窗口属于什么类型?

- Dialog是拥有一个PhoneWindow的实例，是应用窗口类型TYPE_APPLICATION。

1. 来看下Dialog的构造方式

```
Dialog(@NonNull Context context, @StyleRes int themeResId, boolean
createContextThemeWrapper) {
        if (createContextThemeWrapper) {
            if (themeResId == Resources.ID_NULL) {
                final TypedValue outValue = new TypedValue();
                context.getTheme().resolveAttribute(R.attr.dialogTheme, outValue, true);
                themeResId = outValue.resourceId;
            }
            mContext = new ContextThemeWrapper(context, themeResId);
        } else {
            mContext = context;
        }

        mWindowManager = (WindowManager) context.getSystemService(Context.WINDOW_SERVICE);
```

```
        final Window w = new PhoneWindow(mContext);
        mWindow = w;
        w.setCallback(this);
        w.setOnWindowDismissedCallback(this);
        w.setOnWindowSwipeDismissedCallback(() -> {
            if (mCancelable) {
                cancel();
            }
        });
        w.setWindowManager(mWindowManager, null, null);
        w.setGravity(Gravity.CENTER);

        mListenersHandler = new ListenersHandler(this);
    }
```

- 从上述代码我们可以得出，Dialog在构造方法中设置WindowManager传入的appToken为空，那Dialog的appToken在什么时候获取的了。
- 如果用Application或者Service的Context去获取这个WindowManager服务的话，会得到一个WindowManagerImpl的实例，这个实例里token也是空的。
- 如果我们使用Activity作为Context，会拿到Activity的mWindowManager，这个mWindowManager在Activity的attach方法被创建，Token指向此Activity的Token

---

10.38 WindowMangerService中token到底是什么？token的存在意义是什么?

详细讲解

享学课堂移动开发课程：Framework专题 WMS部分

这道题想考察什么?

1. 是否了解WindowManagerService的知识?

考察的知识点

1. 什么是Token
2. WMS如何验证Token
3. ActivityThread、ApplicationThread的理解
4. WMS的整体流程

考生应该如何回答

我们先来看一个常见的错误情况

Dialog的报错情况

当我们想要在屏幕上显示一个Dialog的时候，我们也许会在Activity的onCreate方法里这么写：

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val dialog = AlertDialog.Builder(this)
    dialog.run{
        title = "我是标题"
        setMessage("我是内容")
    }
    dialog.show()
}
```

他的构造参数需要传入一个context对象，这个context的要求不能是ApplicationContext等其他context，只能是ActivityContext。如果我们使用Application传入会怎么样呢?

```
override fun onCreate(savedInstanceState: Bundle?) {
...
    // 注意这里换成了ApplicationContext
    val dialog = AlertDialog.Builder(applicationContext)
...
}
```

运行一下：

报错了，原因是`You need to use a Theme.AppCompat theme (or descendant) with this activity.`，那我们给他添加一个Theme：

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    // 注意这里添加了主题
    val dialog = AlertDialog.Builder(applicationContext,R.style.AppTheme)
    ...
}
```

好了再次运行：

嗯嗯？又崩溃了，原因是：`Unable to add window -- token null is not valid; is your activity running?` token为null？这个token是什么？为什么同样是context，使用activity没问题，用ApplicationContext就出问题了？他们之间有什么区别？那么这篇文章就围绕这个token来展开讨论一下。

> 文章采用思考问题的思路来展开讲述，我会根据我学习这部分内容时候的思考历程进行复盘。希望这种解决问题的思维可以帮助到你。 对token有一定了解的读者可以看到最后部分的整体流程把握，再选择想阅读的部分仔细阅读。

什么是token

首先我们看到报错是在`ViewRootImpl.java:907`，这个地方肯定有进行token判断，然后才抛出异常，这样我们就能寻找到token了，那我们直接去这个地方看看：

```
ViewRootImpl.class(api29)
public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
    ...
    int res;
    ...
    res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
                        getHostVisibility(), mDisplay.getDisplayId(), mTmpFrame,
                        mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
                        mAttachInfo.mOutsets, mAttachInfo.mDisplayCutout, mInputChannel,
                        mTempInsets);
    ...
    if (res < WindowManagerGlobal.ADD_OKAY) {
        ...
        switch (res) {
            case WindowManagerGlobal.ADD_BAD_APP_TOKEN:
            case WindowManagerGlobal.ADD_BAD_SUBWINDOW_TOKEN:
                /*
                *   1
                */
                throw new WindowManager.BadTokenException(
                    "Unable to add window -- token " + attrs.token
                    + " is not valid; is your activity running?");
                ...
        }
        ...
    }
    ...
}
```

我们可以快速看出在注释1的地方抛出了异常，是根据一个变量`res`来判别的，这个`res`出自方法`addToDisplay`，那么token的判别肯定在这个方法里面了，`res`只是一个 判别的结果，那么我们是必须进入这个`addToDisplay`里去看一下。mWindowSession的类型是IWindowSession，他是一个接口，那他的实现类在哪里？找不到实现类就无法知道他的实际代码。这里涉及到window机制的相关内容，简单说一下：

> WindowManagerService是系统服务进程，应用进程跟window联系需要通过跨进程通信：AIDL，这里的IWindowSession只是一个Binder接口，他的具体实现类在系统服务进程的Session类。所以这里的逻辑就跳转到了Session类的`addToDisplay`方法中。

那我们继续到Session的方法中看一下：

```
Session.class(api29)
class Session extends IWindowSession.Stub implements IBinder.DeathRecipient {
    final WindowManagerService mService;
    public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams attrs,
            int viewVisibility, int displayId, Rect outFrame, Rect outContentInsets,
            Rect outStableInsets, Rect outOutsets,
            DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel,
            InsetsState outInsetsState) {
        return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId,
outFrame,
                outContentInsets, outStableInsets, outOutsets, outDisplayCutout,
outInputChannel,
                outInsetsState);
    }
}
```

可以看到，Session确实是继承自接口IWindowSession，因为WMS与Session全部是运行在系统进程，所以不需要跨进程通信，直接调用WMS的方法：

```
public int addWindow(Session session, IWindow client, int seq,
        LayoutParams attrs, int viewVisibility, int displayId, Rect outFrame,
        Rect outContentInsets, Rect outStableInsets, Rect outOutsets,
        DisplayCutout.ParcelableWrapper outDisplayCutout, InputChannel outInputChannel,
        InsetsState outInsetsState) {
    ...
    WindowState parentWindow = null;
    ...
    // 获取parentWindow
    parentWindow = windowForClientLocked(null, attrs.token, false);
    ...
    final boolean hasParent = parentWindow != null;
    // 获取token
    WindowToken token = displayContent.getWindowToken(
        hasParent ? parentWindow.mAttrs.token : attrs.token);
    ...
    // 验证token
    if (token == null) {
    if (rootType >= FIRST_APPLICATION_WINDOW && rootType <= LAST_APPLICATION_WINDOW) {
          Slog.w(TAG_WM, "Attempted to add application window with unknown token "
                        + attrs.token + ".  Aborting.");
            return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
        }
       ...//各种验证
    }
    ...
}
```

WMS的addWindow方法代码这么多如何找到关键代码？还记得viewRootImpl在判断res是什么值的情况下抛出异常吗？是`WindowManagerGlobal.ADD_BAD_APP_TOKEN`和`WindowManagerGlobal.ADD_BAD_SUBWINDOW_TOKEN`，我们找到其中一个就可以找到token的判断位置，从代码中可以得出，当token==null的时候，会进行各种判断，第一个返回的就是`WindowManagerGlobal.ADD_BAD_APP_TOKEN`，这样我们就快速找到token的类型：**WindowToken。**那么根据我们这一路跟过来，最终找到token的类型了。再看一下这个类：

```
class WindowToken extends WindowContainer<WindowState> {
    ...
    // The actual token.
    final IBinder token;
}
```

官方告诉我们里面的token变量才是真正的token，而这个token是IBinder对象。

到此为止关于token是什么已经弄清楚了：

> - token是一个IBinder对象
> - 只有利用token才能成功添加dialog

那么接下来需要思考更多的问题：

- 在show过程中Dialog是如何拿到token并给到WMS验证的?
- 在activity和application两者之间这个token有什么不同?
- WMS如何知道这个token是合法的，换句话说，WMS怎么验证token的?

dialog如何获取到context的token的?

首先，第一个问题我们需要解决：在show过程中Dialog是如何拿到token并给到WMS验证的?

我们知道弹出dialiog的不同结果，原因在于token的问题。那么在弹出Dialog的过程中，他是怎么拿到context的token并给到WMS验证的？源码内容很多，我们需要首先看一下token是封装在哪个参数被传输到了WMS，确定了参数我们的缩小搜索范围，我们回到WMS的代码：

```
parentWindow = windowForClientLocked(null, attrs.token, false);
WindowToken token = displayContent.getWindowToken(
        hasParent ? parentWindow.mAttrs.token : attrs.token);
```

从上述代码可以看到token和一个`attrs.token`关系非常密切，而这个attrs从调用栈一路往回走到了viewRootImpl中：

```
ViewRootImpl.class(api29)
public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
   ...
}
```

可以看到这是一个WindowManager.LayoutParams类型的对象。那我们接下来需要从最开始`show()`开始，追踪这个token是如何被d到到的：

```
Dialog.class(api30)
public void show() {
    ...
    WindowManager.LayoutParams l = mWindow.getAttributes();
    ...
    mWindowManager.addView(mDecor, l);
    ...
}
```

这里的`mWindow`和`mWindowManager`是什么？我们到Dialog的构造函数瞧一瞧：

```
Dialog(@NonNull Context context, @StyleRes int themeResId, boolean
createContextThemeWrapper) {
    // 如果context没有主题，需要把context封装成ContextThemeWrapper
    if (createContextThemeWrapper) {
        if (themeResId == Resources.ID_NULL) {
            final TypedValue outValue = new TypedValue();
            context.getTheme().resolveAttribute(R.attr.dialogTheme, outValue, true);
            themeResId = outValue.resourceId;
        }
        mContext = new ContextThemeWrapper(context, themeResId);
    } else {
        mContext = context;
    }
    // 初始化windowManager
    mWindowManager = (WindowManager) context.getSystemService(Context.WINDOW_SERVICE);
    // 初始化PhoneWindow
    final Window w = new PhoneWindow(mContext);
    mWindow = w;
    ...
    // 把windowManager和PhoneWindow联系起来
    w.setWindowManager(mWindowManager, null, null);
    ...
}
```

我们看到初始化的逻辑中的重点：首先判断这是不是个有主题的context，如果不是需要配置主题并封装成一个ContextThemeWrapper对象，这也是为什么我们文章一开始使用application但是没有设置主题会抛异常。然后得到windowManager，注意，这里是非常主要的部分，也是我当初看源码的时候忽视的地方。这里的context可能是Activity，也可能是Application，他们的`getSystemService`返回的windowManager是一样的吗，看代码：

```
Activity.class(api29)
public Object getSystemService(@ServiceName @NonNull String name) {
    if (getBaseContext() == null) {
        throw new IllegalStateException(
                "System services not available to Activities before onCreate()");
    }
    if (WINDOW_SERVICE.equals(name)) {
        // 返回的是自身的WindowManager
        return mWindowManager;
    } else if (SEARCH_SERVICE.equals(name)) {
        ensureSearchManager();
```

```
        return mSearchManager;
    }
    return super.getSystemService(name);
}

ContextImpl.class(api29)
public Object getSystemService(String name) {
    return SystemServiceRegistry.getSystemService(this, name);
}
```

Activity返回的其实是自己的WindowManager，而Application是执行ContextImpl的方法，返回的是应用服务windowManager。这两个有什么不一样，我们暂时不清楚，先留意着，再继续把源码看下去寻找答案。我们回到前面的方法，看到`mWindowManager.addView(mDecor, l)`;我们知道一个PhoneWindow对应一个WindowManager，这里用到的WindowManager并不是Dialog自己创建的WindowManager，而是参数context的windowManager，也意味着并没有使用自己创建的PhoneWindow。Dialog创建PhoneWindow的目的是为了使用DecorView模板，我们可以了解到addView的参数里并不是window而只是mDecor。

我们继续看源码，同时要注意`l`这个参数，最终token就是封装在里面。`addView`方法最终会执行到了`WindowManagerGlobal`的`addView`方法，具体调用流程可以看本文开头：

```
public void addView(View view, ViewGroup.LayoutParams params,
        Display display, Window parentWindow) {
    ...
    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams) params;
    if (parentWindow != null) {
        parentWindow.adjustLayoutParamsForSubWindow(wparams);
    }
    ...
    ViewRootImpl root;
    ...
    root = new ViewRootImpl(view.getContext(), display);
    ...
    try {
        root.setView(view, wparams, panelParentView);
    }
    ...
}
```

这里我们只需要看WindowManager.LayoutParams参数，parentWindow是与windowManagerPhoneWindow，所以这里肯定不是null，进入到`adjustLayoutParamsForSubWindow`方法进行调整参数。最后执行ViewRootImpl的setView方法。到这里WindowManager.LayoutParams这个参数还没有被设置token，那么最有可能是在`adjustLayoutParamsForSubWindow`方法中了，我们进去代码看看：

```
Window.class(api29)
void adjustLayoutParamsForSubWindow(WindowManager.LayoutParams wp) {
    CharSequence curTitle = wp.getTitle();
    if (wp.type >= WindowManager.LayoutParams.FIRST_SUB_WINDOW &&
            wp.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW) {
        // 子窗口token获取逻辑
        if (wp.token == null) {
            View decor = peekDecorView();
```

```
            if (decor != null) {
                wp.token = decor.getWindowToken();
            }
        }
        ...
    } else if (wp.type >= WindowManager.LayoutParams.FIRST_SYSTEM_WINDOW &&
                wp.type <= WindowManager.LayoutParams.LAST_SYSTEM_WINDOW) {
        // 系统窗口token获取逻辑
        ...
    } else {
        // 应用窗口token获取逻辑
        if (wp.token == null) {
            wp.token = mContainer == null ? mAppToken : mContainer.mAppToken;
        }
        ...
    }
    ...
}
```

最终看到了token的赋值了，这里分为三种情况：应用层窗口、子窗口和系统窗口，分别进行token赋值。

应用窗口直接得到的是与WindowManager对应的PhoneWindow的mAppToken，而子窗口是得到DecorView的token，系统窗口属于比较特殊的窗口，使用Application也可以弹出，但是需要权限，这里不深入讨论。而这里的关键就是：**这个dialog是什么类型的窗口？以及windowManager对应的PhoneWindow中有没有token?**

而这个判断跟我们前面赋值的不同WindowManagerImpl有直接的关系。那么这里，就需要到Activity和Application创建WindowManager的过程一看究竟了。

Activity与Application的WindowManager

首先我们看到Activity的window创建流程。这里需要了解Activity的启动流程。跟踪Activity的启动流程，最终会到ActivityThread的`performLaunchActivity`：

```
ActivityThread.class(api29)
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    // 最终会调用这个方法来创建window
    // 注意r.token参数
    activity.attach(appContext, this, getInstrumentation(), r.token,
        r.ident, app, r.intent, r.activityInfo, title, r.parent,
        r.embeddedID, r.lastNonConfigurationInstances, config,
        r.referrer, r.voiceInteractor, window, r.configCallback,
        r.assistToken);
    ...
}
```

这个方法执行了activity的attach方法来初始化window，同时我们看到参数里有了`r.token`这个参数，这个token最终会给到哪里，我们赶紧继续看下去：

```
Activity.class(api29)
final void attach(Context context, ActivityThread aThread,
        Instrumentation instr, IBinder token, int ident,
```

```
        Application application, Intent intent, ActivityInfo info,
        CharSequence title, Activity parent, String id,
        NonConfigurationInstances lastNonConfigurationInstances,
        Configuration config, String referrer, IVoiceInteractor voiceInteractor,
        Window window, ActivityConfigCallback activityConfigCallback, IBinder assistToken) {
    ...
    // 创建window
    mWindow = new PhoneWindow(this, window, activityConfigCallback);
    ...
    // 创建windowManager
    // 注意token参数
    mWindow.setWindowManager(
            (WindowManager)context.getSystemService(Context.WINDOW_SERVICE),
            mToken, mComponent.flattenToString(),
            (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0);
    mWindowManager = mWindow.getWindowManager();
    ...
}
```

attach方法里创建了PhoneWindow以及相应的WindowManager，再把创建的windowManager给到activity的mWindowManager属性。我们看到创建WindowManager的参数里有token，我们继续看下去：

```
public void setWindowManager(WindowManager wm, IBinder appToken, String appName,
        boolean hardwareAccelerated) {
    mAppToken = appToken;
    mAppName = appName;
    mHardwareAccelerated = hardwareAccelerated;
    if (wm == null) {
        wm = (WindowManager)mContext.getSystemService(Context.WINDOW_SERVICE);
    }
    mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);
}
```

这里利用应用服务的windowManager给Activity创建了WindowManager，同时把token保存在了PhoneWindow内。到这里我们明白Activity的PhoneWindow是拥有token的。那么Application呢?

Application执行的是ContextImpl的getSystemService方法，而这个方法返回的是应用服务的windowManager，Application本身并没有创建自己的PhoneWindow和WindowManager，所以也没有给PhoneWindow赋值token的过程。

因此，**Activity有自己PhoneWindow、WindowManager，同时它的PhoneWindow含有token；而Application并没有自己的PhoneWindow，它返回的WindowManager是应用服务windowManager，并没有赋值token的过程。**

那么到这里结论马上就可以出来了，还差最后一步，我们回到赋值token的那个方法中：

```
Window.class(api29)
void adjustLayoutParamsForSubWindow(WindowManager.LayoutParams wp) {
    if (wp.type >= WindowManager.LayoutParams.FIRST_SUB_WINDOW &&
            wp.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW) {
        // 子窗口token获取逻辑
```

```
        if (wp.token == null) {
            View decor = peekDecorView();
            if (decor != null) {
                wp.token = decor.getWindowToken();
            }
        }
        ...
    } else {
        // 应用窗口token获取逻辑
        if (wp.token == null) {
            wp.token = mContainer == null ? mAppToken : mContainer.mAppToken;
        }
        ...
    }
    ...
}
```

当我们使用Activity来弹出dialog的时候，此时Activity的DecorView已经是显示到屏幕上了，也就是我们的Activity是有界面了，这个情况下，它就是属于子窗口的类型被添加到PhoneWindow中，而它的token就是DecorView的token，此时DecorView已经被显示到屏幕上，它本身是拥有token的；

> 这里补充一点。当一个view（view树）被添加到屏幕上后，它所对应的viewRootImpl有一个token对象，这个token来自WindowManagerGlobal，他是一个IWindowSession 对象。从源码中可以看到，当我们的PhoneWindow的DecorView添加到屏幕后，后续添加的子window的token，就都是这个IWindowSession 对象了。

而如果是第一次显示，也就是应用界面，那么他的token就是Activity初始化传入的token。

但是如果使用的是Application，因为它内部并没有token，那么这里获取到的token等于null，后面到WMS也就会抛出异常了。而这也就使用Activity可以弹出Dialog而Application不可以的原因，因为受到了token的限制。

WMS是如何验证token的

到现在我们可以知道。我们从WMS的token判断找到了token的类型、token的载体：WindowManager.LayoutParams，然后我们再从dialog的创建流程寻找到了赋值token的时候会因为windowManager的不同而不同。所以我们再去查看了两者不同的windowManager，最终得到结论**Activity的PhoneWindow是有token，而Application使用的是应用级服务windowManager，并不存在token。**

那么此时还有疑问：

- 在什么时候被创建的token?
- token是合法的还是非法的WMS怎么判定?

虽然到目前我们已经搞清楚了原因，但是知识还是缺一角，本着探索知识的好奇心我们继续研究下去。

我们从前面Activity的创建window过程知道token来自于`r.token`，这个`r`是ActivityRecord，是AMS启动Activity的时候传进来的Activity信息。那么要追踪这个token的创建就必须顺着这个`r`的传递路线一路回溯。同样这涉及到Activity的完整启动流程，我不会解释详细的调用栈情况，默认你清楚activity的启动流程，如果不清楚，可以先去学习Activity的启动流程相关的知识。首先看到这个ActivityRecord是在哪里被创建的：

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

这样我们需要继续往前回溯，看看这个token是在哪里被获取的：

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

public void execute(ClientTransaction transaction) {
    ...
    executeCallbacks(transaction);
    ...
}
public void executeCallbacks(ClientTransaction transaction) {
    ...
        final IBinder token = transaction.getActivityToken();
        item.execute(mTransactionHandler, token, mPendingActions);
    ...
}
```

可以看到我们的token在ClientTransaction对象获取到。ClientTransaction是AMS传来的一个事务，负责控制activity的启动，里面包含两个item，一个负责执行activity的create工作，一个负责activity的resume工作。那么这里我们就需要到ClientTransaction的创建过程一看究竟了。下面我们的逻辑就要进入系统进程了：

```
ActivityStackSupervisor.class(api28)
final boolean realStartActivityLocked(ActivityRecord r, ProcessRecord app,
    boolean andResume, boolean checkConfig) throws RemoteException {
    ...
    final ClientTransaction clientTransaction = ClientTransaction.obtain(app.thread,
            r.appToken);
    ...
}
```

这个方法创建了ClientTransaction，但是token并不是在这里被创建的，我们继续往上回溯（注意代码的api版本，不同版本的代码会不同）：

```
ActivityStarter.java(api28)
private int startActivity(IApplicationThread caller, Intent intent, Intent ephemeralIntent,
        String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
        IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
        IBinder resultTo, String resultWho, int requestCode, int callingPid, int callingUid,
```

```
        String callingPackage, int realCallingPid, int realCallingUid, int startFlags,
        SafeActivityOptions options,
        boolean ignoreTargetSecurity, boolean componentSpecified, ActivityRecord[]
outActivity,
        TaskRecord inTask, boolean allowPendingRemoteAnimationRegistryLookup) {
    ...

    //记录得到的activity信息
    ActivityRecord r = new ActivityRecord(mService, callerApp, callingPid, callingUid,
            callingPackage, intent, resolvedType, aInfo, mService.getGlobalConfiguration(),
            resultRecord, resultWho, requestCode, componentSpecified, voiceSession != null,
            mSupervisor, checkedOptions, sourceRecord);
   ...
}
```

我们一路回溯,终于看到了ActivityRecord的创建，我们进去构造方法中看看有没有token相关的构造：

```
ActivityRecord.class(api28)
ActivityRecord(... Intent _intent,...) {
    appToken = new Token(this, _intent);
    ...
}

static class Token extends IApplicationToken.Stub {
   ...
    Token(ActivityRecord activity, Intent intent) {
        weakActivity = new WeakReference<>(activity);
        name = intent.getComponent().flattenToShortString();
    }
    ...
}
```

可以看到确实这里进行了token创建。而这个token看接口就知道是个Binder对象，他持有ActivityRecord的弱引用，这样可以访问到activity的所有信息。到这里token的创建我们也找到了。那么WMS是怎么知道一个token是否合法呢？每个token创建后，会在后续发送到WMS ，WMS对token进行缓存，而后续对于应用发送来的token只需要在缓存拿出来匹配一下就知道是否合法了。那么WMS是怎么拿到token的?

---

activity的启动流程后续会走到一个方法：`startActivityLocked`,这个方法在我前面的activity启动流程并没有讲到，因为它并不属于“主线”，但是他有一个非常重要的方法调用，如下：

```
ActivityStack.class(api28)
void startActivityLocked(ActivityRecord r, ActivityRecord focusedTopActivity,
        boolean newTask, boolean keepCurTransition, ActivityOptions options) {
    ...
    r.createWindowContainer();
    ...
}
```

这个方法就把token送到了WMS 那里，我们继续看下去：

```
ActivityRecord.class(api28)
void createWindowContainer() {
    ...
    // 注意参数有token，这个token就是之前初始化的token
    mWindowContainerController = new AppWindowContainerController(taskController, appToken,
            this, Integer.MAX_VALUE /* add on top */, info.screenOrientation, fullscreen,
            (info.flags & FLAG_SHOW_FOR_ALL_USERS) != 0, info.configChanges,
            task.voiceSession != null, mLaunchTaskBehind, isAlwaysFocusable(),
            appInfo.targetSdkVersion, mRotationAnimationHint,
            ActivityManagerService.getInputDispatchingTimeoutLocked(this) * 1000000L);
    ...
}
```

注意参数有token，这个token就是之前初始化的token，我们进入到他的构造方法看一下：

```
AppWindowContainerController.class(api28)
public AppWindowContainerController(TaskWindowContainerController taskController,
        IApplicationToken token, AppWindowContainerListener listener, int index,
        int requestedOrientation, boolean fullscreen, boolean showForAllUsers, int
configChanges,
        boolean voiceInteraction, boolean launchTaskBehind, boolean alwaysFocusable,
        int targetSdkVersion, int rotationAnimationHint, long inputDispatchingTimeoutNanos,
        WindowManagerService service) {
    ...
    synchronized(mWindowMap) {
        AppWindowToken atoken = mRoot.getAppWindowToken(mToken.asBinder());
       ...
        atoken = createAppWindow(mService, token, voiceInteraction,
task.getDisplayContent(),
                inputDispatchingTimeoutNanos, fullscreen, showForAllUsers, targetSdkVersion,
                requestedOrientation, rotationAnimationHint, configChanges,
launchTaskBehind,
                alwaysFocusable, this);
        ...
    }
}
```

还记得我们在一开始看WMS的时候他验证的是什么对象吗？WindowToken，而AppWindowToken是WindowToken的子类。那么我们继续追下去：

```
AppWindowContainerController.class(api28)
AppWindowToken createAppWindow(WindowManagerService service, IApplicationToken token,
        boolean voiceInteraction, DisplayContent dc, long inputDispatchingTimeoutNanos,
        boolean fullscreen, boolean showForAllUsers, int targetSdk, int orientation,
        int rotationAnimationHint, int configChanges, boolean launchTaskBehind,
        boolean alwaysFocusable, AppWindowContainerController controller) {
    return new AppWindowToken(service, token, voiceInteraction, dc,
            inputDispatchingTimeoutNanos, fullscreen, showForAllUsers, targetSdk,
orientation,
            rotationAnimationHint, configChanges, launchTaskBehind, alwaysFocusable,
            controller);
}
```

```
AppWindowToken(WindowManagerService service, IApplicationToken token, ...) {
    this(service, token, voiceInteraction, dc, fullscreen);
    ...
}

WindowToken.class
WindowToken(WindowManagerService service, IBinder _token, int type, boolean persistOnEmpty,
        DisplayContent dc, boolean ownerCanManageAppTokens, boolean roundedCornerOverlay) {
    token = _token;
    ...
    onDisplayChanged(dc);
}
```

createAppWindow方法调用了AppWindow的构造器，然后再调用了父类WindowToken的构造器，我们可以看到这里最终对token进行了缓存，并调用了一个方法，我们看看这个方法做了什么：

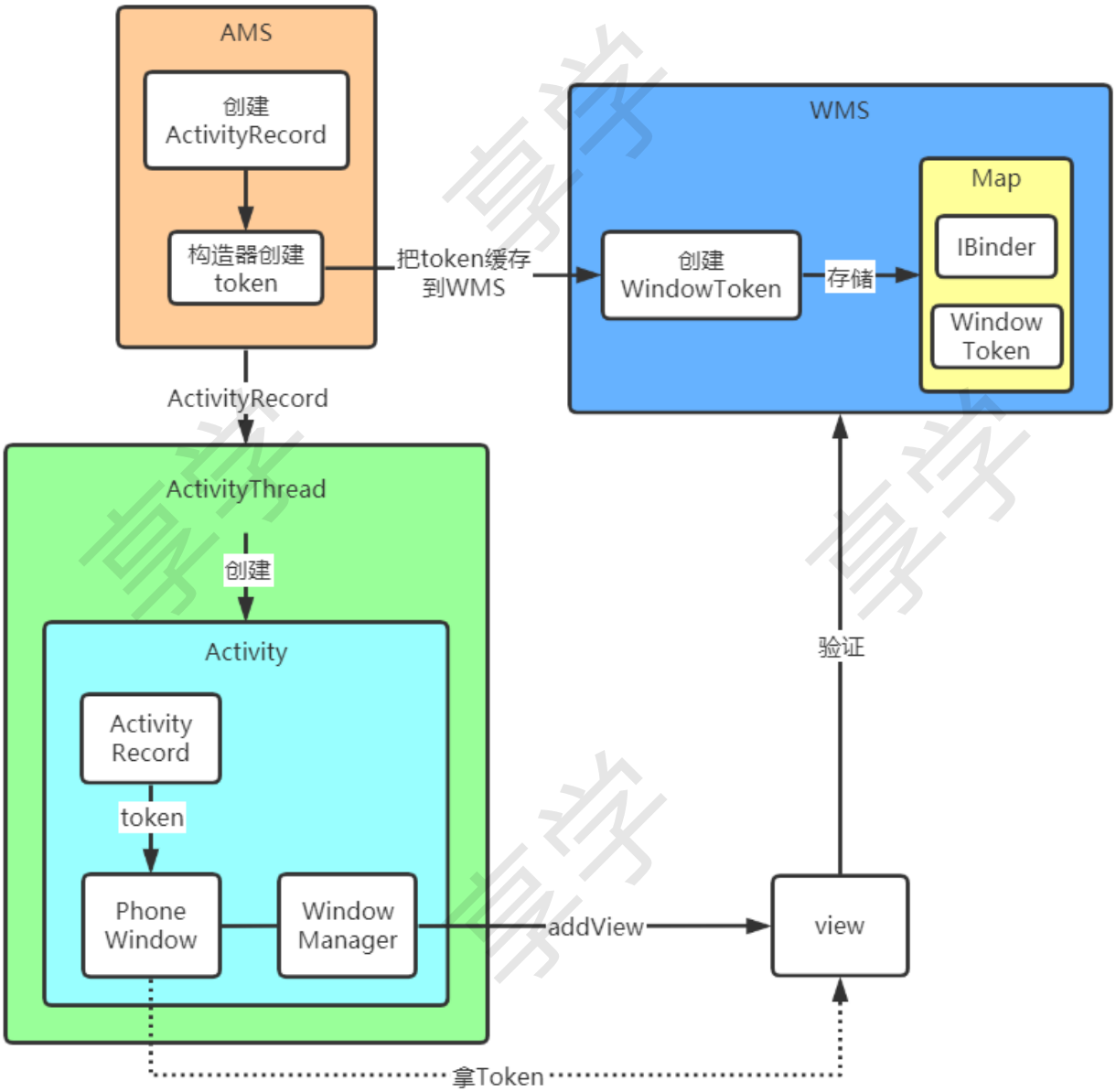
```
WindowToken.class
void onDisplayChanged(DisplayContent dc) {
    dc.reParentWindowToken(this);
    ...
}

DisplayContent.class(api28)
void reParentWindowToken(WindowToken token) {
    addWindowToken(token.token, token);
}
private void addWindowToken(IBinder binder, WindowToken token) {
    ...
    mTokenMap.put(binder, token);
    ...
}
```

mTokenMap 是一个 HashMap<IBinder, WindowToken> 对象，这里就可以保存一开始初始化的token以及后来创建的windowToken两者的关系。这里的逻辑其实已经在WMS中了，所以这个也是保存在WMS中。AMS和WMS都是运行在系统服务进程，因而他们之间可以直接调用方法，不存在跨进程通信。WMS就可以根据IBinder对象拿到windowToken进行信息比对了。至于怎么比对，代码位置在一开始的时候已经有涉及到，读者可自行去查看源码，这里就不讲了。

那么，到这里关于整个token的知识就全部走了一遍了，AMS怎么创建token，WMS怎么拿到token的流程也根据我们回溯的思路走了一遍。

整体流程把握

前面根据我们思考问题的思维走完了整个token流程，但是似乎还是有点乱，那么这一部分，就把前面讲的东西整理一下，对token的知识有一个整体上的感知，同时也当时前面内容的总结。先来看整体图：



1. token在创建ActivityRecord的时候一起被创建，他是一个IBinder对象，实现了接口IApplicationToken。
2. token创建后会发送到WMS，在WMS中封装成WindowToken，并存在一个HashMap<IBinder,WindowToken>。
3. token会随着ActivityRecord被发送到本地进程，ActivityThread根据AMS的指令执行Activity启动逻辑。
4. Activity启动的过程中会创建PhoneWindow和对应的WindowManager，同时把token存在PhoneWindow中。
5. 通过Activity的WindowManager添加view/弹出dialog时会把PhoneWindow中的token放在窗口LayoutParams中。
6. 通过viewRootImpl向WMS进行验证，WMS在LayoutParams拿到IBinder之后就可以在Map中获取WindowToken。
7. 根据获取的结果就可以判断该token的合法情况。

这就是整个token的运作流程了。而具体的源码和细节在上面已经解释完了，读者可自行选择重点部分再次阅读源码。

从源码设计看token

不同的context拥有不同的职责，系统对不同的context限制了不同的权利，让在对应情景下的组件只能做对应的事情。其中最明显的限制就是UI操作。

token看着是属于window机制的领域内容，其实是context的知识范畴。我们知道context一共有三种最终实现类：Activity、Application、Service，context是区分一个类是普通Java类还是android组件的关键。context拥有访问系统资源的权限，是各种组件访问系统的接口对象。但是，三种context，只有Activity允许有界面，而其他的两种是不能有界面的，也没必要有界面。**为了防止开发者乱用context造成混乱，那么必须对context的权限进行限制，这也就是token存在的意义。**拥有token的context可以创建界面、进行UI操作，而没有token的context如service、Application，是不允许添加view到屏幕上的（这里的view除了系统窗口）。

为什么说这不属于window机制的知识范畴？WMS控制每一个window，是通过viewRootImpl中的IWindowSession来进行通信的，token在这个过程中只充当了一个验证作用，且当PhoneWindow显示了DecorView之后，后续添加的View使用的token都是ViewRootImpl的IWindowSession对象。这表示当一个PhoneWindow可以显示界面后，那么对于后续其添加的view无需再次进行权限判断。因而，**token真正限制的，是context是否可以显示界面，而不是针对window。**

而我们了解完底层逻辑后，不是要去知道怎么绕过他的限制，动一些“大胆的想法”，而是要知道官方这么设计的目的。我们在开发的时候，也要针对不同职责的context来执行对应的事务，**不要使用Application或Service来做UI操作。**

总结

文章采用思考问题的思路来表述，通过源码分析，讲解了关于token的创建、传递、验证等内容。同时，token在源码设计上的思想进行了总结。

android体系中各种机制之间是互相联系，彼此连接构成一个完整的系统框架。token涉及到window机制和context机制，同时对activity的启动流程也要有一定的了解。阅读源码各种机制的源码，可以从多个维度来帮助我们对一个知识点的理解。同时阅读源码的过程中，不要局限在当前的模块内，思考不同机制之间的联系，系统为什么要这么设计，解决了什么问题，可以帮助我们从架构的角度去理解整个android源码设计。阅读源码切忌无目标乱看一波，要有明确的目标、验证什么问题，针对性寻找那一部分的源码，与问题无关的源码暂时忽略，不然会在源码的海洋里游着游着就溺亡了。