# 第8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

8.10 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）~leo
这道题想考察什么?
考生应该如何回答
8.11 handler.post(Runnable) runnable是如何执行的？（字节跳动、小米）~colin
这道题想考察什么?
考察的知识点
考生应该如何回答
8.12 handler的Callback和handlemessage都存在，但callback返回true handleMessage会执行？（字节跳动、小米）~leo
这道题想考察什么?
考生应该如何回答
8.13 Handler的sendMessage和postDelay的区别？（字节跳动）~colin
这道题想考察什么?
考察的知识点
考生应该如何回答
8.14 IdleHandler是什么？怎么使用，能解决什么问题？~leo
这道题想考察什么?
考生应该如何回答
IdleHandler 是什么？怎么用?
能解决什么问题?
8.15 为什么Looper.loop不阻塞主线程？~leo
这道题想考察什么?
考生应该如何回答
进程
线程
ActivityThread
死循环问题
8.16 Looper无限循环为啥没有ANR（B站）~colin
这道题想考察什么?
考察的知识点
考生应该如何回答
8.17 Looper如何在子线程中创建？（字节跳动、小米）~leo
这道题想考察什么?
考生应该如何回答
8.18 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？（字节跳动、小米）~leo
这道题想考察什么?
考生应该如何回答
Looper相关
Handler相关
总结
8.19 Zygote进程的启动流程 ~leo
这道题想考察什么?
考生应该如何回答
1.Zygote是什么
2.Zygote启动脚本
3.Zygote进程启动流程
8.20 SystemServer进程的启动流程 ~leo
这道题想考察什么?
考生应该如何回答
1.SystemServer是干什么的
2.Zygote处理SystemServer进程
2-1.ZygoteInit.nativeZygoteInit
2-2.RuntimeInit.applicationInit
3.SystemServer.main
4.总结
8.21 子线程发消息到主线程进行更新 UI， 除了 handler 和 AsyncTask， 还有什么 ~leo
这道题想考察什么?

考生应该如何回答

8.22 子线程中能不能 new handler？ 为什么？ ~colin

这道题想考察什么?

考察的知识点

考生应该如何回答

8.23 源码层面来说下application为什么是单例? Vivo ~leo

这道题想考察什么?

考生应该如何回答

handleBindApplication

makeApplication

8.24 Intent的原理，作用，可以传递哪些类型的参数? ~leo

这道题想考察什么?

考生应该如何回答

Intent的原理

1.查找

2.信息匹配

Intent的作用

1.打开指定网页

2.打电话

3.发送信息

4.播放指定路径音乐：action+data+type

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）

6.安装程序：action+data+type

Intent传递的数据

1.传递简单的数据

2.传递数组

3.传递集合

4.传递对象

5.传递Bitmap

总结

8.25 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

考察的知识点

考生应该如何回答

Activity之间传递大量数据通常经过如下几种方式实现：

Application

使用LruCache

持久化数据

8.26 打开页面，如何实现一键退出?

这道题想考察什么?

考察的知识点

考生应该如何回答

问题本质

采用Activity启动模式：SingleTask

原理如下

优点

缺点

采用Activity启动标记位

通过系统任务栈

BroadcastReceiver

自己管理

RxBus

一键结束当前 App 进程

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

这道题想考察什么?

考生应该如何回答

初始化工作

## 8.1 Android中多进程通信的方式有哪些？~leo

**这道题想考察什么?**

Android 多进程相关问题

**考察的知识点**

Android 中有哪些多进程通信的方式

**考生应该如何回答**

Android中支持的多进程通信方式主要有以下几种：

1. AIDL：功能强大，支持进程间一对多的实时并发通信，并可实现 RPC (远程过程调用)。
2. Messenger：支持一对多的串行实时通信， AIDL 的简化版本。
3. Bundle：四大组件的进程通信方式，只能传输 Bundle 支持的数据类型。
4. ContentProvider：强大的数据源访问支持，主要支持 CRUD 操作，一对多的进程间数据共享，例如应用访问系统的通讯录数据。
5. BroadcastReceiver：即广播，但只能单向通信，接收者只能被动的接收消息。
6. 文件共享：在非高并发情况下共享简单的数据，不适合高并发场景。
7. Socket：可通过网络传输字节流，开销大，支持一对多的并发实时通信。

---

## 8.2 进程通信你用过哪些？原理是什么？（字节跳动、小米）~leo

**这道题想考察什么?**

Android 多进程相关问题

**考察的知识点**

Android 中有哪些多进程通信的方式

**考生应该如何回答**

Android中支持的多进程通信方式主要有以下几种：

1. AIDL：功能强大，支持进程间一对多的实时并发通信，并可实现 RPC (远程过程调用)。
2. Messenger：支持一对多的串行实时通信， AIDL 的简化版本。
3. Bundle：四大组件的进程通信方式，只能传输 Bundle 支持的数据类型。
4. ContentProvider：强大的数据源访问支持，主要支持 CRUD 操作，一对多的进程间数据共享，例如应用访问系统的通讯录数据。
5. BroadcastReceiver：即广播，但只能单向通信，接收者只能被动的接收消息。
6. 文件共享：在非高并发情况下共享简单的数据，不适合高并发场景。
7. Socket：可通过网络传输字节流，开销大，支持一对多的并发实时通信。

---

## 8.3 android 的 IPC 通信方式，线程（进程间）通信机制有哪些 ~leo

**这道题想考察什么?**

Android 多进程相关问题

**考察的知识点**

Android 中有哪些多进程通信的方式

**考生应该如何回答**

Android中支持的多进程通信方式主要有以下几种：

1. AIDL：功能强大，支持进程间一对多的实时并发通信，并可实现 RPC (远程过程调用)。
2. Messenger：支持一对多的串行实时通信， AIDL 的简化版本。
3. Bundle：四大组件的进程通信方式，只能传输 Bundle 支持的数据类型。
4. ContentProvider：强大的数据源访问支持，主要支持 CRUD 操作，一对多的进程间数据共享，例如应用访问系统的通讯录数据。
5. BroadcastReceiver：即广播，但只能单向通信，接收者只能被动的接收消息。
6. 文件共享：在非高并发情况下共享简单的数据，不适合高并发场景。
7. Socket：可通过网络传输字节流，开销大，支持一对多的并发实时通信。

---

## 8.4 描述下Binder机制原理？（东方头条）~ leo

**这道题想考察什么?**

对Binder通信的理解

**考生应该如何回答**

首先讲解进程间的内存关系，这样才知道为什么进程间不能直接通信，以及通信是怎么实现的。

**内存**

进程间的内存是隔离的，所以进程与进程之间是不能直接通信的。

每一个进程的内存又分为用户空间和内核空间，它们都是虚拟内存，与物理内存之间存在映射关系。为了安全考虑，用户空间与内核空间也是隔离的。

**如何实现通信**

那如何实现通信的呢?

1. 进程中的用户空间与内核空间虽然是隔离的，但是可以通过系统调用进行通信。
2. 每一个进程的内核空间都是共享的，所以不同进程的内核空间是可以互相通信的。

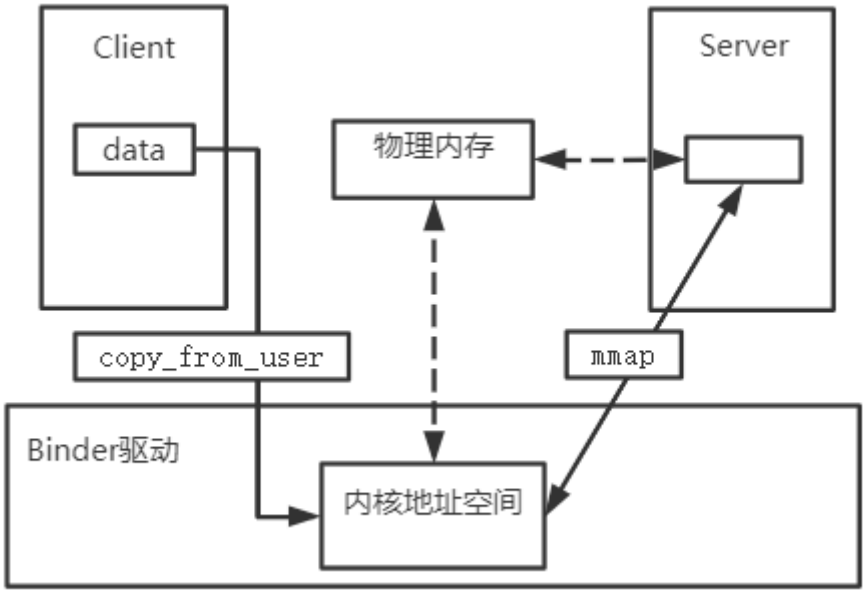
结合上面两点就可得出它们的通信方式了。例如有两个进程，一个Client，一个Server。

Client首先通过系统调用（copy_from_user）就可以将数据从用户空间拷贝到内核空间，这儿做了一次拷贝。因为内核空间是共享的，所以Server的内核空间中也得到了这个数据。接着Server的用户空间，再通过系统调用（copy_to_user）就可以将数据从内核空间拷贝到用户空间，从而得到数据，这里又发生了一次拷贝。

上面就是正常实现进程间通信的原理，而Binder对此做了优化，减少了Server进程这块的内核空间到用户空间的数据拷贝。

它是如何减少这次拷贝的呢?

将Server的用户空间和内核空间中的一块区域，通过MMAP实现了共享。这样当数据从Client的用户空间拷贝到内核空间时，由于Server的用户空间和内核空间实现了共享，所以Server的用户空间也可以直接处理数据了。




---

## 8.5 什么是 AIDL 以及如何使用 ~leo

**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 如何工作?

1. 创建 AIDL 文件, 在这个文件里面定义接口, 该接口定义了可供客户端访问的方法和属性。
2. 执行编译操作，会生成对应的 java 代码，其中类和接口总共有三个。
    1. 用户在AIDL中定义的接口
    2. Stub抽象类，它继承扩展了接口并实现了远程调用需要的几个方法。
    3. Proxy 实现了接口的一个类，是一个代理类。
3. 用户实现 onServiceConnected 回调方法，在这个方法中需执行 Stub.asInterface(service)，如果是同进程则是获取服务的对象，如果是跨进程，则是获取服务的代理对象，即Proxy的实例。
4. 然后用户就可以像同进程一样，直接使用 Proxy 实例去调用接口中的方法。
5. Proxy 类中对应方法，主要是处理了数据的打包，然后通过调用 transact 方法，进入native层，如果是同步调用当前线程会挂起。
6. 最终执行到服务端，调用 Stub 中的 onTransact，因为 Stub 是一个抽象类，并没有实现接口中的方法，服务端在 Service 中会继承 Stub 去实现接口中的方法，所以 onTransact 最终会调用到 Service 的具体实现。从而完成客户端到服务端的调用。

---

## 8.6 AIDL 的全称是什么?如何工作?能处理哪些类型的数据？~leo

**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 Android Interface Definition Language(Android接口描述语言)是一种接口描述语言。编译器可以通过aidl文件生成一段代码，通过预先定义的接口达到两个进程内部通信进程跨界访问对象的目的。

AIDL 如何工作?

1. 创建 AIDL 文件, 在这个文件里面定义接口, 该接口定义了可供客户端访问的方法和属性。
2. 执行编译操作，会生成对应的 java 代码，其中类和接口总共有三个。
    1. 用户在AIDL中定义的接口
    2. Stub抽象类，它继承扩展了接口并实现了远程调用需要的几个方法。
    3. Proxy 实现了接口的一个类，是一个代理类。
3. 用户实现 onServiceConnected 回调方法，在这个方法中需执行 Stub.asInterface(service)，如果是同进程则是获取服务的对象，如果是跨进程，则是获取服务的代理对象，即Proxy的实例。
4. 然后用户就可以像同进程一样，直接使用 Proxy 实例去调用接口中的方法。
5. Proxy 类中对应方法，主要是处理了数据的打包，然后通过调用 transact 方法，进入native层，如果是同步调用当前线程会挂起。
6. 最终执行到服务端，调用 Stub 中的 onTransact，因为 Stub 是一个抽象类，并没有实现接口中的方法，服务端在 Service 中会继承 Stub 去实现接口中的方法，所以 onTransact 最终会调用到 Service 的具体实现。从而完成客户端到服务端的调用。

实现接口时有几个原则:

1. 抛出的异常不要返回给调用者. 跨进程抛异常处理是不可取的。
2. IPC调用是同步的。如果你知道一个IPC服务需要超过几毫秒的时间才能完成地话，你应该避免在Activity的主线程中调用。也就是IPC调用会挂起应用程序导致界面失去响应. 这种情况应该考虑单起一个线程来处理。
3. 不能在AIDL接口中声明静态属性。

AIDL支持的数据类型:

1. 不需要import声明的简单Java编程语言类型(int,boolean等)
2. String, CharSequence不需要特殊声明
3. List, Map和Parcelables类型, 这些类型内所包含的数据成员也只能是简单数据类型, String等其他支持的类型.

---

## 8.7 为什么 Android 要采用 Binder 作为 IPC 机制？~leo

**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**

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。

---

## 8.8 Binder线程池的工作过程是什么样？（东方头条）~leo

**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**

**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。

#### 10.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。

---

## 8.9 Handler怎么进行线程通信，原理是什么？（东方头条）~leo

**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的使用

首先介绍Handler的使用，代码如下：

```
// 在主线程创建 Handler 对象，并重写回调方法
private final Handler handler = new Handler() {
    @Override
    public void handleMessage(@NonNull Message msg) { // 收到消息系统会回调该方法
        super.handleMessage(msg);
        // 写自己的业务逻辑
    }
};
```

然后在子线程发送消息，代码如下：

```
// 创建子线程
new Thread(new Runnable() {
    @Override
    public void run() {
        // 发送消息
        handler.sendMessage(new Message());
    }
});
```

### Handler的通信原理

接着再讲解Handler的通信原理。
当子线程执行 `handler.sendMessage()` 等发送消息的方法，通过源码可以发现，最终都会执行到下面的方法：

```
public boolean sendMessageAtTime(@NonNull Message msg, long uptimeMillis) {
    return enqueueMessage(queue, msg, uptimeMillis);
}
```

```
private boolean enqueueMessage(@NonNull MessageQueue queue, @NonNull Message
msg,
                               long uptimeMillis) {
    // message 的 target 对象保存了 当前Handler对象
    msg.target = this;
    // 执行MessageQueue 的 enqueueMessage 方法
    return queue.enqueueMessage(msg, uptimeMillis);
}
```

```
// 根据不同情况，将新的Message消息添加到Message链表中
boolean enqueueMessage(Message msg, long when) {
    synchronized (this) {
        msg.markInUse();
        msg.when = when;
        Message p = mMessages;
        boolean needWake;
        if (p == null || when == 0 || when < p.when) {
            // New head, wake up the event queue if blocked.
            msg.next = p;
            mMessages = msg;
            needWake = mBlocked;
        } else {
            needWake = mBlocked && p.target == null && msg.isAsynchronous();
            Message prev;
            for (;;) {
                prev = p;
                p = p.next;
                if (p == null || when < p.when) {
                    break;
                }
                if (needWake && p.isAsynchronous()) {
                    needWake = false;
                }
            }
            msg.next = p; // invariant: p == prev.next
            prev.next = msg;
```

```
        }

        if (needWake) {
            nativeWake(mPtr);
        }
    }
    return true;
}
```

小结：通过上面几步，成功将handler发送的消息，加入到了Message链表中。
那消息是怎么处理的呢？在App启动的时候，会执行 ActivityThread中的main方法：

```
public static void main(String[] args) {
    // 在主线程创建Looper对象
    Looper.prepareMainLooper();
    // 执行 Looper的 loop 方法
    Looper.loop();
}
```

```
public static void loop() {
    final Looper me = myLooper();

    final MessageQueue queue = me.mQueue;

    for (;;) {
        // 从 MessageQueue 中取出消息
        Message msg = queue.next(); // might block
        try {
            // msg的target实际就是Handler对象，此处就是执行Handler的dispatchMessage方
法
            msg.target.dispatchMessage(msg);
        }
    }
}
```

```
public void dispatchMessage(@NonNull Message msg) {
    // 一般情况下就会执行 下面方法，即我们写的回调方法
    handleMessage(msg);
}
```

到此我们Handler的流程就分析完了，可以看到，在子线程发送消息，消息会被MessageQueue管理，通过链表的形式用Message保存。而接受消息的处理是通过Looper.loop()方法遍历执行回调方法，因为Looper.loop()是执行在主线程中的，所以handleMessage这个回调方法也是执行在主线程中的。从而实现了线程间通信。

---

## 8.10 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）~leo

**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消息处理流程前面已经介绍过了。还不清楚的，可以看下前面章节。
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;
    }
```

MessageQueeu.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，知道有满足条件的消息到来。

---

## 8.11 handler.post(Runnable) runnable是如何执行的？（字节跳动、小米）~colin

**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**

1. 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，和sendMessage(Message msg)是一致。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。

1. 然后，我们看下Message是如何执行的。

- Message会被添加到MessageQueen中，通过queue.enqueueMessage(msg, uptimeMillis)方法，我们看下enqueueMessage的源码。

```
  boolean enqueueMessage(Message msg, long when) {
    //省略部分代码
     Message prev;
     for (;;) {
        prev = p;
        p = p.next;
        if (p == null || when < p.when) {
            break;
         }
         if (needWake && p.isAsynchronous()) {
             needWake = false;
         }
      }
      msg.next = p; // invariant: p == prev.next
      prev.next = msg;
    //省略部分代码
  }
```

- 从上面代码可以看出，MessageQueen通过循环的方式找出队列末端，然后将msg添加到Queen中，那msg什么时候被执行？我可以Looper.loop方法，在里面找到了msg.target.dispatchMessage(msg)，这样就很明了了。target是我们的Handler，我们看到相关的Handler源码。

```
/**
     * 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();
    }
```

- Handler.dispatchMessage中如果是callback不为空，最终调用的是callback.run()方法。到此为止，我们走完了整个Runnable的执行流程。

---

## 8.12 handler的Callback和handlemessage都存在，但callback返回true handleMessage会执行？（字节跳动、小米）~leo

**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**

这题的答案是：callback返回true handleMessage不会执行。原因如下：
Handler消息处理流程前面已经介绍过了。还不清楚的，一定要先看下前面章节(章节名：Handler怎么进行线程通信，原理是什么？（东方头条）)。
获取需要执行的消息之后，将调用msg.target.dispatchMessage(msg);处理消息，具体如下：

```
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);
}
```

可以看到，除了在Handler#handleMessage(...)中处理消息外，Handler 机制还提供了两个 Callback 来增加消息处理的灵活性。具体来说，若设置了Message.Callback则优先执行，否则判断Handler.Callback的返回结果，如果返回false，则最后分发到Handler.handleMessage(...)；如果返回true，handleMessage 不会执行。

---

## 8.13 Handler的sendMessage和postDelay的区别？（字节跳动）~colin

**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**

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;
}
```

- 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使用到了我们的getPostMessage函数，将我们的Runnable转化为了我们的Message，然后相较sendMessage多了一个延时操作。
- postDelayed函数使用起来会便捷一些。

---

## 8.14 IdleHandler是什么？怎么使用，能解决什么问题？~leo

**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**

Handler消息处理流程前面已经介绍过了。还不清楚的，可以看下前面章节。

**IdleHandler 是什么？怎么用?**

IdleHandler 说白了，就是 Handler 机制提供的一种，可以在 Looper 事件循环的过程中，当出现空闲的时候，允许我们执行任务的一种机制。
IdleHandler 被定义在 MessageQueue 中，它是一个接口。

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

可以看到，定义时需要实现其 queueIdle() 方法。同时返回值为 true 表示是一个持久的 IdleHandler 会重复使用，返回 false 表示是一个一次性的 IdleHandler。
既然 IdleHandler 被定义在 MessageQueue 中，使用它也需要借助 MessageQueue。在 MessageQueue 中定义了对应的 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 或 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，那么何时出现空闲？MessageQueue 是一个基于消息触发时间的优先级队列，所以队列出现空闲存在两种场景。

1. MessageQueue 为空，没有消息；
2. 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，都会尝试执行 IdleHandler。
处理 IdleHandler 的场景，就在 Message.next() 这个获取消息队列下一个待执行消息的方法中，我们跟一下具体的逻辑。

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

        synchronized (this) {
            // ...
            if (msg != null) {
                if (now < msg.when) {
                    // 计算休眠的时间
                    nextPollTimeoutMillis = (int) Math.min(msg.when - now,
Integer.MAX_VALUE);
                } else {
                    // Other code
                    // 找到消息处理后返回
```

```
                    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;
    }
}
```

我们先解释一下 next() 中关于 IdleHandler 执行的主逻辑：

1. 准备执行 IdleHandler 时，说明当前待执行的消息为 null，或者这条消息的执行时间未到；
2. 当 pendingIdleHandlerCount < 0 时，根据 mIdleHandlers.size() 赋值给 pendingIdleHandlerCount，它是后期循环的基础；
3. 将 mIdleHandlers 中的 IdleHandler 拷贝到 mPendingIdleHandlers 数组中，这个数组是临时的，之后进入 for 循环；
4. 循环中从数组中取出 IdleHandler，并调用其 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，最核心的就是在 next() 中，当队列空闲的时候，循环 mIdleHandler 中记录的 IdleHandler 对象，如果其 queueIdle() 返回值为 false 时，将其从 mIdleHander 中移除。
需要注意的是，对 mIdleHandler 这个 List 的所有操作，都通过 synchronized 来保证线程安全，这一点无需担心。

**能解决什么问题?**

IdleHander 可以保证不进入死循环。
当队列空闲时，会循环执行一遍 mIdleHandlers 数组并执行 IdleHandler.queueIdle() 方法。而如果数组中有一些 IdleHander 的 queueIdle() 返回了 true，则会保留在 mIdleHanders 数组中，下次依然会再执行一遍。
注意现在代码逻辑还在 MessageQueue.next() 的循环中，在这个场景下 IdleHandler 机制是如何保证不会进入死循环的?
有些文章会说 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重新唤醒，但这是不对的。
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 决定了下次进入 nativePollOnce() 超时的时间，它传递 0 的时候等于不会进入休眠，所以说 natievPollOnce() 进入休眠所以不会死循环是不对的。
这很好理解，毕竟 IdleHandler.queueIdle() 运行在主线程，它执行的时间是不可控的，那么 MessageQueue 中的消息情况可能会变化，所以需要再处理一遍。
实际不会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，循环开始前，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 continus；
- Step 4，重置 pendingIdleHandlerCount 为 0；

在第一次循环时，pendingIdleHandlerCount<0，会给pendingIdleHandlerCount赋值，然后执行到Step4，此时 Step3不会执行，在第二次循环时，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() 时就会进入休眠，等待再次被唤醒。
下次唤醒时，mMessage 必然会有一个待执行的 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。

---

## 8.15 为什么Looper.loop不阻塞主线程？~leo

**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**

Handler消息处理流程前面已经介绍过了。还不清楚的，一定要先看下前面章节(章节名：Handler怎么进行线程通信，原理是什么？（东方头条）)。
这里涉及线程，先说说说 Android 中每个应用所对应的 进程/线程：

**进程**

每个app运行时前首先创建一个进程，该进程是由 Zygote fork 出来的，用于承载各种 Activity/Service 等组件。进程对于上层应用来说是完全透明的，这也是 Google 有意为之，让App 程序都是运行在 Android Runtime。大多数情况一个 App 就运行在一个进程中，除非在 AndroidManifest.xml 中配置 Android:process 属性，或通过 native 代码 fork 进程。

### 线程

线程对应用来说非常常见，比如每次 new Thread().start 都会创建一个新的线程。该线程与App所在进程之间资源共享，从 Linux 角度来说进程与线程除了是否共享资源外，并没有本质的区别，都是一个 task_struct 结构体，在 CPU 看来进程或线程无非就是一段可执行的代码，CPU 采用 CFS 调度算法，保证每个task都尽可能公平的享有 CPU 时间片。

### ActivityThread

ActivityThread 是应用程序的入口，这里你可以看到写Java程序时司空见惯的 main 方法，而 main 方法正是整个Java程序的入口。ActivityThread 的 main 方法主要就是做消息循环，一旦退出消息循环，那么你的程序也就可以退出了。
Android是事件驱动的，在Loop.loop()中不断接收事件、处理事件，而Activity的生命周期都依靠于主线程的Loop.loop()来调度，所以可想而知它的存活周期和Activity也是一致的。当没有事件需要处理时，主线程就会阻塞；当子线程往消息队列发送消息，并且往管道文件写数据时，主线程就被唤醒。
ActivityThread 并不是一个 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 方法很简短，一眼就能看全，我们看到里面有 Looper 了，那么接下来就找找 ActivityThread 对应的 Handler 贴出 handleMessage 的小部分：

```
public void handleMessage(Message msg) {
    if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
    switch (msg.what) {
        case BIND_APPLICATION:
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
"bindApplication");
            AppBindData data = (AppBindData)msg.obj;
            handleBindApplication(data);
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            break;
        case EXIT_APPLICATION:
            if (mInitialApplication != null) {
                mInitialApplication.onTerminate();
            }
            Looper.myLooper().quit();
            break;
        case RECEIVER:
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
"broadcastReceiveComp");
            handleReceiver((ReceiverData)msg.obj);
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            break;
        case CREATE_SERVICE:
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, ("serviceCreate:
" + String.valueOf(msg.obj)));
            handleCreateService((CreateServiceData)msg.obj);
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            break;
        case BIND_SERVICE:
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "serviceBind");
            handleBindService((BindServiceData)msg.obj);
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            break;
```

看完这 Handler 里处理消息的内容应该明白了吧， Activity 的生命周期都有对应的 case 条件了，ActivityThread 有个 getHandler 方法，得到这个 handler就可以发送消息，然后 loop 里就分发消息，然后就发给 handler, 然后就执行到 H（Handler ）里的对应代码。所以这些代码就不会卡死～，有消息过来就能执行。举个例子，在 ActivityThread 里的内部类 ApplicationThread 中就有很多 sendMessage 的方法：

```
public final void schedulePauseActivity(IBinder token, boolean finished,
                boolean userLeaving, int configChanges, boolean dontReport) {
            sendMessage(
                    finished ? H.PAUSE_ACTIVITY_FINISHING : H.PAUSE_ACTIVITY,
                    token,
                    (userLeaving ? 1 : 0) | (dontReport ? 2 : 0),
                    configChanges);
        }

        public final void scheduleStopActivity(IBinder token, boolean
showWindow,
                int configChanges) {
           sendMessage(
                showWindow ? H.STOP_ACTIVITY_SHOW : H.STOP_ACTIVITY_HIDE,
                token, 0, configChanges);
        }

        public final void scheduleWindowVisibility(IBinder token, boolean
showWindow) {
            sendMessage(
                showWindow ? H.SHOW_WINDOW : H.HIDE_WINDOW,
                token);
        }

        public final void scheduleSleeping(IBinder token, boolean sleeping) {
            sendMessage(H.SLEEPING, token, sleeping ? 1 : 0);
        }

        public final void scheduleResumeActivity(IBinder token, int
processState,
                boolean isForward, Bundle resumeArgs) {
            updateProcessState(processState, false);
            sendMessage(H.RESUME_ACTIVITY, token, isForward ? 1 : 0);
}
```

ActivityThread 的 main 方法主要就是做消息循环，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阻塞，取到消息会做出相应的处理。如果某个消息处理时间过长，就可能会影响 UI 线程的刷新速率，造成卡顿的现象。

**死循环问题**

线程既然是一段可执行的代码，当可执行代码执行完成后，线程生命周期便该终止了，线程退出。而对于主线程，我们是绝不希望会被运行一段时间，自己就退出，那么如何保证能一直存活呢？简单做法就是可执行代码是能一直执行下去的，死循环便能保证不会被退出。
例如，binder 线程也是采用死循环的方法，通过循环方式与 Binder 驱动进行读写操作，当然并非简单地死循环，无消息时会休眠。但这里可能又引发了另一个问题，既然是死循环又如何去处理其他事务呢？通过创建新线程的方式。
真正会卡死主线程的操作是在回调方法 onCreate/onStart/onResume 等操作时间过长，会导致掉帧，甚至发生 ANR，looper.loop 本身不会导致应用卡死。

## 8.16 Looper无限循环为啥没有ANR（B站）~colin

**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)即应用无响应。
- 发生ANR的主要三种类型：KeyDispatchTimeOut即key事件的分事件超过了5秒；BroadcastTimeOut即广播的时间超时，分为FG和BG，分别是10秒和60秒；ServiceTimeOut即Service超时，超过了20秒；

1. **Looper无限循环是什么，为什么这么编写?**

- Looper无线循环是Looper不断取MessageQueen中的Message，并执行的一种机制。我们的APP中的事件，如点击、长按、滑动、Activity的生命周期切换都是依赖这种机制，这是安卓的基石。
- 当主线程的MessageQueue中没有消息，便会阻塞在Loop的queue.next()中的nativePollOnce方法里。这个时候主线程会释放CPU资源进入休眠状态，知道下一个消息到达或者有事物发生，通过往pipe管道写入数据来进行唤醒主线程工作。
- 事件是随机的、时间不确定的，所以我们需要无限循环的处理事件，这是安卓的驱动方式，如果不循环，那应用就退出了。

1. **Looper无限循环为啥没有ANR?**

- Looper的无线循环是安卓的事件机制，只要我们执行的事件得到及时的处理，是不会导致ANR的

---

## 8.17 Looper如何在子线程中创建？（字节跳动、小米）~leo

**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**

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

```
Looper.prepare();// Looper 初始化，使用Handler 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();//Handler机制的大前提，使调用线程进入死循环，没错，Android中主线程一直都在死循环

// Handler中的方法，获取Handler中缓存的Looper对象，使用Handler无参构造创建将缓存调用线程Looper对象，使用Handler(Looper looper)构造创建将缓存传入的Looper对象；
getLooper();

//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
mHandler.getLooper().quit();

//Android4.3，API Level 18 后加入的方法，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
mHandler.getLooper().quitSafely();
```

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。

---

## 8.18 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？（字节跳动、小米）~leo

**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**

Handler消息处理流程前面已经介绍过了。还不清楚的，一定要先看下前面章节(章节名：Handler怎么进行线程通信，原理是什么？（东方头条））。

**Looper相关**

从Looper.prepare()开始，要在一个线程里面处理消息，代码如下：

```
class LooperThread extends Thread{
public Handler mHandler;
public void run() {
    Looper.prepare();
    mHandler = new Handler() {
        public void handleMessage(Message msg) {
            // process incoming messages here
        }
    };
        Looper.loop();
    }
}
```

首先就必须要先调用Looper.prepare()，那这个方法做了些什么呢：

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

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

代码其实只有关键性的一句，就是sThreadLocal.set(new Looper(quitAllowed))，首先来看看sThreadLocal。

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

ThreadLocal：代表了一个线程局部的变量，每条线程都只能看到自己的值，并不会意识到其它的线程中也存在该变量。
在这里ThreadLocal的作用是保证了每个线程都有各自的Looper。
接下来看看创建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**

在为当前线程创建了Looper之后，就可以创建Handler来处理消息了，Handler是怎么跟Looper关联上的?

```
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和mQueue代表当前Handler关联的Looper和消息队列，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();
}
```

其实还是调用的线程局部变量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。

---

## 8.19 Zygote进程的启动流程 ~leo

**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**

**1.Zygote是什么**

Zygote是在init进程启动时创建的，它又称为孵化器，它可以通过fork（复制进程）的形式来创建应用程序进程和SystemServer进程。并且，Zygote进程在启动的时候会创建DVM或者ART，因此通过fork而创建的应用程序进程和SystemServer进程可以在内部获取一个DVM或者ART的实例副本。

**2.Zygote启动脚本**

init.rc文件中采用了如下所示的Import类型语句来引入Zygote启动脚本：`import /init.${ro.zygote}.rc`
这里根据属性ro.zygote的内容来引入不同的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：上面的Zygote的启动脚本都存放在system/core/rootdir目录中。
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.main 方法中会执行如下方法启动 zygote 进程。

```
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);
        }
    }
}
```

通过代码中注释的6步，最终执行到了 ZygoteInit.java中的main函数。
这里要注意，第六步之所以这里要使用JNI，是因为ZygoteInit是Java代码。最终，Zygote就从Native层进入了Java FrameWork层。在此之前，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。

---

## 8.20 SystemServer进程的启动流程 ~leo

**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**

**1.SystemServer是干什么的**

SystemServer进程主要是用于创建系统服务的，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**

```
// 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);
        }
    }
```

```
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);
        }
    }
```

```
public static final Runnable zygoteInit(int targetSdkVersion, long[]
disabledCompatChanges,
            String[] argv, ClassLoader classLoader) {
        // 1.进入native方法
        ZygoteInit.nativeZygoteInit();
        // 2.
        return RuntimeInit.applicationInit(targetSdkVersion,
disabledCompatChanges, argv,
                classLoader);
    }
```

### 2-1.ZygoteInit.nativeZygoteInit

```
// frameworks/base/core/jni/AndroidRuntime.cpp
int register_com_android_internal_os_ZygoteInit_nativeZygoteInit(JNIEnv* env)
{
    const JNINativeMethod methods[] = {
        { "nativeZygoteInit", "()V",
            (void*) com_android_internal_os_ZygoteInit_nativeZygoteInit },
    };
    return jniRegisterNativeMethods(env, "com/android/internal/os/ZygoteInit",
        methods, NELEM(methods));
}
```

上面代码使用了JNI动态注册的方式，将nativeZygoteInit()方法和native函数com_android_internal_os_ZygoteInit_nativeZygoteInit()建立了映射关系。

```
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();
    }
```

### 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);
    }
```

```
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()方法中被使用。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();
                    return;
                }
            }
        } catch (Throwable ex) {
            Log.e(TAG, "System zygote died with exception", ex);
            throw ex;
        } finally {
            if (zygoteServer != null) {
                zygoteServer.closeServerSocket();
            }
        }
```

### 3.SystemServer.main

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

```
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);
        }

        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();
    }
```

### 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. 创建SystemServiceManager，它用来对系统服务进行创建、启动和生命周期管理。

3. 启动各种系统服务：引导服务、核心服务、其他服务，共100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。

---

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

**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**

方式一:
在子线程中用Activity对象的runOnUiThread方法更新

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

其实每部也是调用的是Handler的post方法，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否为主线程，如果是当前的线程是主线程，则直接运行，是非主线程的话，调用post方法。

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自己内部也封装了自己的异步处理机制，从上面就可以看出，调用的是 HandlerActionQueue 的post方法，而在Handler内部调用post的时候，先调用的是sendMessageDelayed方法，然后调用sendMessageAtTime方法，紧接着调用enqueueMessage，最终调用的是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，其核心还是通过 Handler机制。

---

## 8.22 子线程中能不能 new handler？ 为什么？ ~colin

**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的基本知识
2. Handler内部运行的原理

**考生应该如何回答**

1. **首先回答子线程中是可以创建Handler。**

- 我们翻看Looper的源码，然后找到注释中的一部分

```
class LooperThread extend Thread {
    public Handler mHandler;

    public void run() {
        Looper.prepare();

        mHandler = new Handler() {
        //process incoming messages here
    }

    Looper.loop();
    }
}
```

我们可以看到Looper的源码注释已经给了我们答案，子线程中是可以创建Handler。那么我们之前的印象中有什么不一样，这里面在Handler的创建前面加入了Looper.prepare()方法，在后面添加了Looper.loop()方法。

1. **我们需要在创建Handler之前加上Looper.prepare()，在创建之后加上Looper.loop()。**

- Looper.prepare()的作用

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

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

从上述代码中我们可以了解到，首先判断了当前线程是否创建了Looper对象，如果有则抛出异常，如果没有则创建一个，然后存入sThreadLocal中。也就是说线程与Looper是一一对应的关系，不能重复创建。

- Looper.loop()的作用

```
public static void loop() {
    //省略部分代码
    final MessageQueue queue = me.mQueue;
    for (;;) {
        //省略部分代码
        Message msg = queue.next();
        try {
        msg.target.dispatchMessage(msg);
        } catch (Exception exception) {}
        //省略部分代码
    }
}
```

loop()是处理Message消息的开始，相当于按下了一个开始键，机器开始运作。拿到MessageQueue之后，会进入一个循环。在循环中，我们取出Message消息，然后执行操作。

1. **为什么在主线程中我们创建Handler没有类似的操作?**

- ActivityThread中的main()

```
public static void main(String[] args) {
    //省略部分代码
    Looper.prepareMainLooper();
    //省略部分代码
    if (sMainThreadHandler == null) {
        sMainThreadHandler = thread.getHandler();
    }
    //省略部分代码
    Looper.loop();
}
```

ActivityThread就是我们常说的主线程或UI线程，主线程中我们是有prepareMainLooper和loop的操作的，系统帮我们做好了而已。

1. **大部分人认为子线程不能创建Handler的误解怎么来的?**

- 子线程直接创建Handler

```
2021-04-10 16:40:57.455 7232-7282/com.example.myapplication E/AndroidRuntime: FATAL EXCEPTION: Thread-2
    Process: com.example.myapplication, PID: 7232
    java.lang.RuntimeException: Can't create handler inside thread Thread[Thread-2,5,main] that has not called Looper.prepare()
        at android.os.Handler.<init>(Handler.java:207)
        at android.os.Handler.<init>(Handler.java:119)
        at com.example.myapplication.MainActivity$1.run(MainActivity.java:20)
        at java.lang.Thread.run(Thread.java:919)
```

没有调用Looper.prepare()之前是不能创建Handler，而不是说子线程不能创建，这个误解的症结在这。

---

## 8.23 源码层面来说下application为什么是单例? Vivo ~leo

**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为什么是单例，实际就是看Application 对象的创建过程，找出它为什么是单例的。
所以我们直接来看Application是如何创建的。

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;
    }

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

通过这块代码就可以知道，当 Application为空的时候，才会创建，从而保证了它是一个单例对象。

---

## 8.24 Intent的原理，作用，可以传递哪些类型的参数? ~leo

**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**

### Intent的原理

Intent原理分为查找和匹配

#### 1.查找

Android使用Intent进行组件，进程之间的通信和跳转。Intent具有隐式Intent和显式Intent两种，Android系统通过PackageManagerService来进行系统组件的维护。
系统启动之后会注册各种系统服务，其中就包括PackageManagerService。在启动之后，PMS会扫描已安装的apk目录，解析apk包下的AndroidManifest.xml文件得到App的相关信息，而每个AndroidManifest.xml又包含了Activity，Service等组件的注册信息，当PMS扫描并且解析完信息后，就清晰地描绘出了整棵apk的信息树。
PackageManagerService的构造函数加载了系统已安装的各类apk，并加载了Framework资源和核心库。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文件进行解析，然后里面的PackageParser的parsePackage方法进行文件解析。实现这一操作的代码是scanDirLi函数。

```
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);
}
```

可以清晰地看到它解析了apk的versionName，versionCode,baseVersionCode。那么apk的各个组件是在哪里解析的呢?
打开方法parseBaseApplication，我们的一切疑问都可以得到回答。

```
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);
            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")) {
```

```
            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")) {
            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")) {
            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")) {
            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")) {
            // 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;
```

```
            }
        }
}
```

显而易见，这里系统将应用的各个组件解析出来，并且存入到了各个集合当中。
至此，整个信息树构建完毕，系统已经存储好了这个应用的组件信息。

**2.信息匹配**

我们看一下调用intent的具体方法，一般情况下，我们都是使用startActivity这个方法的。我们跟进去最终会调用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这个方法。
这个方法经过一系列调用，最终调用了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。
因此整个流程就是：
在系统启动时，PackageManagerService就会启动，PMS将解析所有已安装的应用信息，构建信息表，当用户通过Intent跳转到某个组件时，会根据Intent中包含的信息到PMS中查找对应的组件列表，最后跳转到目标组件当中。

### Intent的作用

#### 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);
```

使用这个功能需要加入权限:

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

#### 3.发送信息

方式一：打开发送信息的界面：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：打开发送短信的界面（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**

**1.传递简单的数据**

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");
```

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**

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)
```

2.List< Object>
将list强转成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.Map<String, Object>,或更复杂的
解决方法是：外层套个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

传递对象的方式有两种：将对象转换为Json字符串或者通过Serializable,Parcelable序列化 不建议使用Android内置的抠脚json解析器，可使用fastjson或者Gson第三方库!

1.将对象转换为Json字符串
写入数据：

```
Book book=new Book();
book.setTitle("Java编程思想");
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());
```

2.使用Serializable序列化对象
Serializable 是序列化的意思，表示将一个对象转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，也可以存储到本地。至于序列化的方法也很简单，只需要让一个类去实现Serializable 这个接口就可以了。
比如说有一个Person 类，其中包含了name 和age 这两个字段，想要将它序列化就可以这样写：

```
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 方法都是用于赋值和读取字段数据的，最重要的部分是在第一行。这里让Person 类去实现了Serializable 接口，这样所有的Person 对象就都是可序列化的了。
接下来在FirstActivity 中的写法非常简单：

```
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 的实例，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，所以才可以这样写。

接下来在SecondActivity 中获取这个对象也很简单，写法如下：

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

3.使用Parcelable序列化对象
除了Serializable 之外，使用Parcelable 也可以实现相同的效果，不过不同于将对象进行序列化，Parcelable 方式的实现原理是将一个完整的对象进行分解，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的功能了。
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 的实现方式要稍微复杂一些。可以看到，首先我们让Person 类去实现了Parcelable 接口，这样就必须重写describeContents()和writeToParcel()这两个方法。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 类中的字段一一写出。注意字符串型数据就调用writeString()方法，整型数据就调用writeInt()方法，以此类推。

除此之外，我们还必须在Person 类中提供一个名为CREATOR 的常量，这里创建了Parcelable.Creator 接口的一个实现，并将泛型指定为Person。接着需要重写createFromParcel()和newArray()这两个方法，在createFromParcel()方法中我们要去读取刚才写出的name 和age字段，并创建一个Person 对象进行返回，其中name 和age 都是调用Parcel 的readXxx()方法读取到的，注意这里读取的顺序一定要和刚才写出的顺序完全相同。而newArray()方法中的实现就简单多了，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加改动，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

通过上面讲解可以看到，常见的一些数据类型，Intent都是可以传递的。

---

## 8.25 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?

**考察的知识点**

1. Activity间大量数据传递及注意事项

**考生应该如何回答**

**Activity之间传递大量数据通常经过如下几种方式实现：**

- 经过Application
- LruCache
- 持久化（sqlite、share preference、file等）

**Application**

这个应该也都接触过，将数据保存早全局Application中，随整个应用的存在而存在，这样不少地方都能访问。具体使用就很少说了。

可是须要 **注意**的是：

当因为某些缘由（好比系统内存不足），咱们的app会被系统强制杀死，此时再次点击进入应用时，系统会直接进入被杀死前的那个界面，制造一种历来没有被杀死的假象。那么问题来了，系统强制中止了应用，进程死了，那么再次启动时Application天然新的，那里边的数据天然木有啦，若是直接使用极可能报空指针或者其余错误。

所以仍是要考虑好这种状况的：

（1）使用时必定要作好非空判断

（2）若是数据为空，能够考虑逻辑上让应用直接返回到最初的activity，好比用`FLAG_ACTIVITY_CLEAR_TASK`或者`BroadcastReceiver 杀掉其余的activity。`

**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等了。

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

---

## 8.26 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>
```

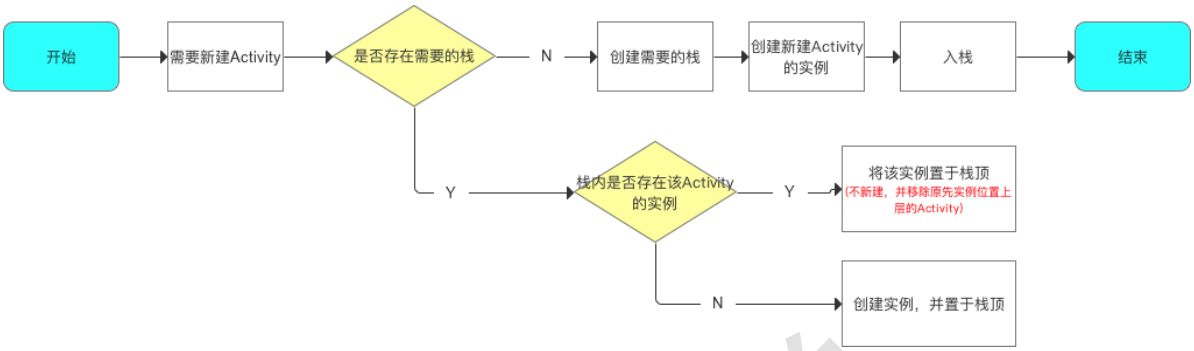
步骤2：在入口 `Activity` 重写 `onNewIntent()`

```
// 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();
```

- 效果图


```
07-12 02:27:42.786 1394-1394/scut.carson_ho.app_1shot_finsh D/ContentValues: 容器内的Activity列表如下
07-12 02:27:42.786 1394-1394/scut.carson_ho.app_1shot_finsh D/ContentValues: MainActivity
07-12 02:27:42.786 1394-1394/scut.carson_ho.app_1shot_finsh D/ContentValues: Activity2
07-12 02:27:42.786 1394-1394/scut.carson_ho.app_1shot_finsh D/ContentValues: 正逐步退出容器内所有Activity
```

- 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

---

## 8.27 startActivity(MainActivity.this，LoginActivity.class); LoginActivity配置的launchMode是何时解析的？~leo

**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.java文件的startActivityUnchecked这个方法， 我们拆解这个函数来分析一下：
注意下两个很重要的属性 mLaunchFlags，mLaunchMode，后面的源码分析也主要围绕着两个属性在讨论。
mLaunchFlags 包含启动的flag，比如FLAG_ACTIVITY_NEW_TASK,FLAG_ACTIVITY_CLEAR_TASK等，作用是规定了如何去启动一个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);
        ... ... ...
```

看 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为null，比如在一个Service启动了一个Activity，那么会为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();
        ... ... ...
```

我们顺便也跳出startActivityUnchecked方法，去看一看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为SingleInstance时，走mSupervisor.findActivityLocked;其他情况下，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,会走mSupervisor.findTaskLocked。

**最合适的可复用栈**

```
    void findTaskLocked(ActivityRecord target, FindTaskResult result) {
        ... ... ...
        // 注意看这里是倒序遍历 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 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;
                }
            }
        }
```

继续来看一下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,看两种典型情况：
1.如果是SingleTask或者是SingleInstance模式的Activity，则执行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**

我们先总结一下几个重要的结论：

1. FLAG_ACTIVITY_NEW_TASK是新建栈或者复用栈的必要条件。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。SingleInstance是很好理解的，SingleTask需要注意下，关于SingleTask我会详细说明，不用担心。
2. 新建栈或者复用已经存在栈的充要条件是什么？
    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)。解释一下，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。

---

## 8.28 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？ ~leo

**这道题想考察什么?**

考察同学是否对清单文件的解析熟悉

**考生应该如何回答**

首先apk是由PMS解析的，下面将介绍PMS如何解析APK。

**scanPackageLI()**

PMS 的构造中会通过 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();
    }
```

最终会调用到scanPackageLI()：

```
    private PackageParser.Package scanPackageLI(PackageParser.Package pkg, File
scanFile,
            final int policyFlags, int scanFlags, long currentTime, @Nullable
UserHandle user)
            throws PackageManagerException {
        // If the package has children and this is the first dive in the
function
        // we scan the package with the SCAN_CHECK_ONLY flag set to see whether
all
        // packages (parent and children) would be successfully scanned before
the
        // actual scan since scanning mutates internal state and we want to
atomically
        // install the package and its children.
        if ((scanFlags & SCAN_CHECK_ONLY) == 0) {
            if (pkg.childPackages != null && pkg.childPackages.size() > 0) {
                scanFlags |= SCAN_CHECK_ONLY;
            }
        } else {
            scanFlags &= ~SCAN_CHECK_ONLY;
        }

        // Scan the parent
        PackageParser.Package scannedPkg = scanPackageInternalLI(pkg, scanFile,
policyFlags,
                scanFlags, currentTime, user);

        // Scan the children
        final int childCount = (pkg.childPackages != null) ?
pkg.childPackages.size() : 0;
        for (int i = 0; i < childCount; i++) {
            PackageParser.Package childPackage = pkg.childPackages.get(i);
            scanPackageInternalLI(childPackage, scanFile, policyFlags,
scanFlags,
                    currentTime, user);
        }


        if ((scanFlags & SCAN_CHECK_ONLY) != 0) {
            return scanPackageLI(pkg, scanFile, policyFlags, scanFlags,
currentTime, user);
        }

        return scannedPkg;
    }
```

这里需要知道的是scanPackageInternalLI() 最终会进入：

```

```

```
    private PackageParser.Package scanPackageLI(File scanFile, int parseFlags,
int scanFlags,
            long currentTime, UserHandle user) throws PackageManagerException {
        if (DEBUG_INSTALL) Slog.d(TAG, "Parsing: " + scanFile);
        PackageParser pp = new PackageParser();
        pp.setSeparateProcesses(mSeparateProcesses);
        pp.setOnlyCoreApps(mOnlyCore);
        pp.setDisplayMetrics(mMetrics);
        pp.setCallback(mPackageParserCallback);

        if ((scanFlags & SCAN_TRUSTED_OVERLAY) != 0) {
            parseFlags |= PackageParser.PARSE_TRUSTED_OVERLAY;
        }

        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "parsePackage");
        final PackageParser.Package pkg;
        try {
            pkg = pp.parsePackage(scanFile, parseFlags); //这里就是我们所需要看的解析
接口
        } catch (PackageParserException e) {
            throw PackageManagerException.from(e);
        } finally {
            Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
        }

        // Static shared libraries have synthetic package names
        if (pkg.applicationInfo.isStaticSharedLibrary()) {
            renameStaticSharedLibraryPackage(pkg);
        }

        return scanPackageLI(pkg, scanFile, parseFlags, scanFlags, currentTime,
user);
    }
```

### parsePackage()

code 路径：frameworks/base/core/java/android/content/pm/PackageParser.java

```
    public Package parsePackage(File packageFile, int flags, boolean useCaches)
            throws PackageParserException {
        Package parsed = useCaches ? getCachedResult(packageFile, flags) : null;
        if (parsed != null) {
            return parsed;
        }

        long parseTime = LOG_PARSE_TIMINGS ? SystemClock.uptimeMillis() : 0;
        if (packageFile.isDirectory()) {
            parsed = parseClusterPackage(packageFile, flags); //解析下面所有的apk
        } else {
            parsed = parseMonolithicPackage(packageFile, flags); //解析指定的apk
        }

        long cacheTime = LOG_PARSE_TIMINGS ? SystemClock.uptimeMillis() : 0;
        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;
    }
```

上面代码中，如果参数是packageFile 是一个目录，就会调用parseClusterPackage()，否则调用parseMonolithicPackage() 来处理。对于关联的多个apk 会放到一个目录下，来看下parseClusterPackage()：

```
    private Package parseClusterPackage(File packageDir, int flags) throws
PackageParserException {
        //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
        final PackageLite lite = parseClusterPackageLite(packageDir, 0);
        if (mOnlyCoreApps && !lite.coreApp) { //如果没有核心应用就会给出exception
            throw new
PackageParserException(INSTALL_PARSE_FAILED_MANIFEST_MALFORMED,
                    "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) {
                throw new
PackageParserException(INSTALL_PARSE_FAILED_BAD_MANIFEST, e.getMessage());
            }
        } else {
            assetLoader = new DefaultSplitAssetLoader(lite, flags);
        }

        try {
            final AssetManager assets = assetLoader.getBaseAssetManager(); //需要
AssetManager 对象
            final File baseApk = new File(lite.baseCodePath);

            //对于apk 进行分析，等到Package 对象
            final Package pkg = parseBaseApk(baseApk, assets, flags);
            if (pkg == null) {
                throw new PackageParserException(INSTALL_PARSE_FAILED_NOT_APK,
                        "Failed to parse base APK: " + baseApk);
            }

            if (!ArrayUtils.isEmpty(lite.splitNames)) {
                final int num = lite.splitNames.length;
                pkg.splitNames = lite.splitNames;
                pkg.splitCodePaths = lite.splitCodePaths;
```

```
                pkg.splitRevisionCodes = lite.splitRevisionCodes;
                pkg.splitFlags = new int[num];
                pkg.splitPrivateFlags = new int[num];
                pkg.applicationInfo.splitNames = pkg.splitNames;
                pkg.applicationInfo.splitDependencies = splitDependencies;
                pkg.applicationInfo.splitClassLoaderNames = new String[num];

                for (int i = 0; i < num; i++) {
                    final AssetManager splitAssets =
assetLoader.getSplitAssetManager(i);
                    parseSplitApk(pkg, i, splitAssets, flags);
                }
            }

            pkg.setCodePath(packageDir.getAbsolutePath());
            pkg.setUse32bitAbi(lite.use32bitAbi);
            return pkg;
        } finally {
            IoUtils.closeQuietly(assetLoader);
        }
    }
```

parseClusterPackage() 方法中先调用parseClusterPackageLite() 方法对目录下的apk 文件进行初步分析，主要是区别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() 方法实际上是分析AndroidManifest.xml 文件。下面来分析下parseBaseApk()：

```
    private Package parseBaseApk(File apkFile, AssetManager assets, int flags)
            throws PackageParserException {
        final String apkPath = apkFile.getAbsolutePath();

        String volumeUuid = null;
        if (apkPath.startsWith(MNT_EXPAND)) {
            final int end = apkPath.indexOf('/', MNT_EXPAND.length());
            volumeUuid = apkPath.substring(MNT_EXPAND.length(), end);
        }

        mParseError = PackageManager.INSTALL_SUCCEEDED;
        mArchiveSourcePath = apkFile.getAbsolutePath();

        if (DEBUG_JAR) Slog.d(TAG, "Scanning base APK: " + apkPath);

        final int cookie = loadApkIntoAssetManager(assets, apkPath, flags);

        Resources res = null;
        XmlResourceParser parser = null;
        try {
            res = new Resources(assets, mMetrics, null);
            parser = assets.openXmlResourceParser(cookie,
ANDROID_MANIFEST_FILENAME);

            final String[] outError = new String[1];
            final Package pkg = parseBaseApk(apkPath, res, parser, flags,
outError);
```

```
            if (pkg == null) {
                throw new PackageParserException(mParseError,
                        apkPath + " (at " + parser.getPositionDescription() +
"): " + outError[0]);
            }

            pkg.setVolumeUuid(volumeUuid);
            pkg.setApplicationVolumeUuid(volumeUuid);
            pkg.setBaseCodePath(apkPath);
            pkg.setSignatures(null);

            return pkg;

        } catch (PackageParserException e) {
            throw e;
        } catch (Exception e) {
            throw new
PackageParserException(INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION,
                    "Failed to read manifest from " + apkPath, e);
        } finally {
            IoUtils.closeQuietly(parser);
        }
    }
```

主要看XmlResourceParser 对象，通过：

```
parser = assets.openXmlResourceParser(cookie, ANDROID_MANIFEST_FILENAME);
```

而这里的ANDROID_MANIFEST_FILENAME 为：

```
private static final String ANDROID_MANIFEST_FILENAME = "AndroidManifest.xml";
```

继续分析，最终会调用到

```
final Package pkg = parseBaseApk(apkPath, res, parser, flags, outError);
```

```
    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;
        }

        if (mCallback != null) {
            String[] overlayPaths = mCallback.getOverlayPaths(pkgName, apkPath);
            if (overlayPaths != null && overlayPaths.length > 0) {
                for (String overlayPath : overlayPaths) {
                    res.getAssets().addOverlayPath(overlayPath);
                }
            }
        }

        final Package pkg = new Package(pkgName);

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

        pkg.mVersionCode = pkg.applicationInfo.versionCode = sa.getInteger(
                com.android.internal.R.styleable.AndroidManifest_versionCode,
0);
        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);

        sa.recycle();

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

这个函数是Package 对象创建的地方，最开始的手会对xml 中的manifest 项进行解析，如：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.shift.camera.test"
    android:versionCode="1"
    android:versionName="1.0" >
```

继续分析，最终函数会调用到parseBaseApkCommon()，这个方法的source code 太长，我们来分布解读。

**parseBaseApkCommon()**

第1步继续Package 对象的解析，获取到SharedUserId 和ShareUserLabel

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

        String str = sa.getNonConfigurationString(
```

```
                com.android.internal.R.styleable.AndroidManifest_sharedUserId,
0);
        if (str != null && str.length() > 0) {
            if ((flags & PARSE_IS_EPHEMERAL) != 0) {
                outError[0] = "sharedUserId not allowed in ephemeral
application";
                mParseError =
PackageManager.INSTALL_PARSE_FAILED_BAD_SHARED_USER_ID;
                return null;
            }
            String nameError = validateName(str, true, false);
            if (nameError != null && !"android".equals(pkg.packageName)) {
                outError[0] = "<manifest> specifies bad sharedUserId name \""
                    + str + "\": " + nameError;
                mParseError =
PackageManager.INSTALL_PARSE_FAILED_BAD_SHARED_USER_ID;
                return null;
            }
            pkg.mSharedUserId = str.intern();
            pkg.mSharedUserLabel = sa.getResourceId(

 com.android.internal.R.styleable.AndroidManifest_sharedUserLabel, 0);
        }
```

第2步：一系列flag 的初始化地方，主要是针对applicationInfo

```
        pkg.installLocation = sa.getInteger(

 com.android.internal.R.styleable.AndroidManifest_installLocation,
                PARSE_DEFAULT_INSTALL_LOCATION);
        pkg.applicationInfo.installLocation = pkg.installLocation;

        final int targetSandboxVersion = sa.getInteger(

 com.android.internal.R.styleable.AndroidManifest_targetSandboxVersion,
                PARSE_DEFAULT_TARGET_SANDBOX);
        pkg.applicationInfo.targetSandboxVersion = targetSandboxVersion;

        /* Set the global "forward lock" flag */
        if ((flags & PARSE_FORWARD_LOCK) != 0) {
            pkg.applicationInfo.privateFlags |=
ApplicationInfo.PRIVATE_FLAG_FORWARD_LOCK;
        }

        /* Set the global "on SD card" flag */
        if ((flags & PARSE_EXTERNAL_STORAGE) != 0) {
            pkg.applicationInfo.flags |= ApplicationInfo.FLAG_EXTERNAL_STORAGE;
        }

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

第3步：解析xml 中Application 项。

```
if (tagName.equals(TAG_APPLICATION)) {
    if (foundApp) {
        if (RIGID_PARSER) {
            outError[0] = "<manifest> has more than one <application>";
            mParseError =
PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
            return null;
        } else {
            Slog.w(TAG, "<manifest> has more than one <application>");
            XmlUtils.skipCurrentTag(parser);
            continue;
        }
    }

    foundApp = true;
    if (!parseBaseApplication(pkg, res, parser, flags, outError)) {
        return null;
    }
}
```

到这里就找到了我们需要的，广播就是在这个里面解析的，其他的我们就暂时不分析了

**parseBaseApplication()**

第1步：解析获取ApplicationInfo

```
final ApplicationInfo ai = owner.applicationInfo;
```

这里的owner 就是上面Package 对象，Package 里面的applicationInfo 就是在这里进一步解析。
其中需要注意的是：

```
        if ((flags&PARSE_IS_SYSTEM) != 0) {
            if (sa.getBoolean(

 com.android.internal.R.styleable.AndroidManifestApplication_persistent,
                    false)) {
                // Check if persistence is based on a feature being present
                final String requiredFeature = sa.getNonResourceString(
                    com.android.internal.R.styleable.
                    AndroidManifestApplication_persistentWhenFeatureAvailable);
                if (requiredFeature == null ||
mCallback.hasFeature(requiredFeature)) {
                    ai.flags |= ApplicationInfo.FLAG_PERSISTENT;
                }
            }
        }
```

如果不是系统应用，会解析persistent 项，并将其置到flag 里。

```
            ai.processName = buildProcessName(ai.packageName, null, pname,
                    flags, mSeparateProcesses, outError);
```

processName 是需要比较确认

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);
             }
```

到此我们知道了整个广播的静态注册流程。

---

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

**这道题想考察什么?**

1. 是否了解ThreadLocal的运行机制?

**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**

1. 首先回答ThreadLocal是什么，ThreadLocal是线程局部变量，属于线程自身所有，不在多个线程间共享，我们访问的每个线程都有自己独立的初始化变量副本。

- 我们翻看ThreadLocal的源码中的注释

  This class provides thread-local variables.  These variables differ from their normal counterparts in that each thread that accesses one (via its{@code get} or {@code set} method) has its own, independently initialized copy of the variables

1. ThreadLocal在安卓中的引用场景。

- 当某些数据以线程为作用域，并且不同线程用于不同的数据副本的情况
- 复杂逻辑下对象传递，比如说监听器的传递

1. ThreadLocal的原理

- 我们先看下ThreadLocal提供的几个方法

  public T get(){}

  public void set(T value){}

  public void remove(){}

  protected T initialValue(){}

**首先看下get()方法中的源码**

```
    /**
     * Returns the value in the current thread's copy of this
     * thread-local variable.  If the variable has no value for the
     * current thread, it is first initialized to the value returned
     * by an invocation of the {@link #initialValue} method.
     *
     * @return the current thread's value of this thread-local
     */
    public T get() {
        Thread t = Thread.currentThread();
        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();
    }

    /**
     * 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;
    }
```

我们首先获得当前现成currentThread，再通过当前线程找到ThreadLocalMap，这是用于存储数据的容器，后面就是一个简单的取值，如果没有值则初始化数据返回初始值。

**ThreadLocalMap是什么，我们继续看源码**

```
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。

#### set()的源码瞧一瞧

```
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) {

    // We don't use a fast path as with get() because it is at
    // least as common to use set() to create new entries as
    // it is to replace existing ones, in which case, a fast
    // path would fail more often than not.

    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值方法一致，主要是避免hash值的冲突。再往下走是遍历Map有三种情况：第一种情况是找了存在的key有效，然后赋值；第二种情况找到了存在的key但是无效，替换掉过期的Entry；第三种情况没找到相同key值，新建一个Entry然后赋值。

#### map.remove()正确使用避免内存泄漏

在线程池中的线程存活时间很长，这种情况下Thread持有的ThreadLocalMap一直都不会被回收，ThreadLocalMap中Entry对ThreadLocal是弱引用，所以ThreadLocal结束自己生命周期后是可以被回收的。但Entry中的Value是被Entry强引用，所以我们需要在结束代码块的时候手动清理ThreadLocal中的value，调用ThreadLocal的remove方法。

1. 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) {
            throw new RuntimeException("Only one Looper may be create per
thread");
        }
        sThreadLocal.set(new Looper(quitAllowed));
    }
```

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。Looper的prepare()方法先是获取sThreadLocal的值，判断当前线程是否已经创建的Looper对象，如果有则报异常，如果没有就会创建一个Looper对象存入sThreadLocal中去。

---

## 8.30 如何通过WindowManager添加Window(代码实现)？-leo

**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);
```

---

## 8.31 为什么Dialog不能用Application的Context？ ~colin

**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. 首先我们看一下使用Application的Context出现什么问题?

```
 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看出，不能将Dialog添加到Window上，因为Token为空是非法的，最后一句，“is your activity running?”。我们可以推测，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：定义窗口样式和行为的抽象基类，作为顶层的View加到WindowManager中，它的实现类为PhoneWindow。 每个Window都需要定义一个Type（应用窗口、子窗口、系统窗口）；Activity是应用窗口，PopupWindow、ContextMenu、OptionMenu是常见的子窗口；像Toast和系统警告提示框是系统窗口。
- WM：WindowManager是用于在应用与Window之间管理接口，像窗口顺序、消息等。
- WMS：WindowManagerService管理窗口的创建、更新、删除、显示顺序，是WindowManager这个管理接口的实现类。在System_server进程中作为服务器，客户端通过IPC调用和它进行交互。
- Token：这里提到的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有一个PhoneWindow的实例，Dialog的类型是TYPE_APPLICATION，属于应用窗口类型

1. 我们看下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);
    }
```

- 从上面代码我们可以看出，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

---

## 8.32 WindowMangerService中token到底是什么？有什么区别

**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**

#### Dialog的报错情况

当我们想要在屏幕上展示一个Dialog的时候，我们可能会在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()
}
```

他的构造参数需要一个context对象，但是这个context不能是ApplicationContext等其他context，只能是ActivityContext（当然没有ApplicationContext这个类，也没有ActivityContext这个类，这里这样写只是为了方便区分context类型，下同）。这样的代码运行时没问题的，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：

```
Caused by: java.lang.IllegalStateException: You need to use a Theme.AppCompat theme (or descendant) with this activity.
   at androidx.appcompat.app.AppCompatDelegateImpl.createSubDecor(AppCompatDelegateImpl.java:843)
   at androidx.appcompat.app.AppCompatDelegateImpl.ensureSubDecor(AppCompatDelegateImpl.java:806)
   at androidx.appcompat.app.AppCompatDelegateImpl.setContentView(AppCompatDelegateImpl.java:693)
   at androidx.appcompat.app.AppCompatDialog.setContentView(AppCompatDialog.java:95)
   at androidx.appcompat.app.AlertController.installContent(AlertController.java:232)
```

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：

```
Caused by: android.view.WindowManager$BadTokenException: Unable to add window -- token null is not valid; is your activity running?
   at android.view.ViewRootImpl.setView(ViewRootImpl.java:907)
   at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:387)
   at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:95)
   at android.app.Dialog.show(Dialog.java:342)
   at androidx.appcompat.app.AlertDialog$Builder.show(AlertDialog.java:1009)
   at com.example.myandroid.MainActivity.onCreate(MainActivity.kt:50)
```

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，然后抛出异常，这样我们就能找到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?");
                ...
        }
        ...
    }
    ...
}
```

我们看到代码就是在注释1的地方抛出了异常，是根据一个变量`res`来判断的，这个`res`来自方法`addToDisplay`，那么token的判断肯定在这个方法里面了，`res`只是一个 判断的结果，那么我们需要进到这个`addToDisplay`里去看一下。mWindowSession的类型是IWindowSession，他是一个接口，那他的实现类是什么？找不到实现类就无法知道他的具体代码。这里涉及到window机制的相关内容，简单讲一下：

> 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和Session都是运行在系统进程，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方法代码这么多怎么找到关键代码？还记得viewRootImpl在判断res是什么值的情况下抛出异常吗？没错是`WindowManagerGlobal.ADD_BAD_APP_TOKEN`和`WindowManagerGlobal.ADD_BAD_SUBWINDOW_TOKEN`，我们只需要找到其中一个就可以找到token的判断位置，从代码中可以看到，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`，这样我们就顺利找到token的类型：**WindowToken。**那么根据我们这一路跟过来，终于找到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。

好了到这里关于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

那么接下来就有更多的问题需要思考了：

- Dialog在show过程中是如何拿到token并给到WMS验证的?
- 这个token在activity和application两者之间有什么不同?
- WMS怎么知道这个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，我们解决第一个问题：Dialog在show过程中是如何拿到token并给到WMS验证的?

我们知道导致两种context（activity和application）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，他是如何拿到context的token并给到WMS验证的？源码内容很多，我们需要先看一下token是封装在哪个参数被传输到了WMS,确定了参数我们的搜索范围就减小了，我们回到WMS的代码：

```
parentWindow = windowForClientLocked(null, attrs.token, false);
WindowToken token = displayContent.getWindowToken(
        hasParent ? parentWindow.mAttrs.token : attrs.token);
```

我们可以看到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是如何被获取到的：

```
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的构造函数一看究竟：

```
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);
    ...
}
```

初始化的逻辑我们看重点就好：首先判断这是不是个有主题的context，如果不是需要设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。然后获取windowManager，注意，这里是重点，也是我当初看源码的时候忽略的地方。这里的context可能是Activity或者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返回的其实是自身的WindowManager，而Application是调用ContextImpl的方法，返回的是应用服务windowManager。**这两个有什么不同，我们暂时不知道，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，这里使用的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，我们可以看到addView的参数里并不是window而只是mDecor。

我们继续看代码，，同时要注意这个`l`参数，最终token就是封装在里面。`addView`方法最终会调用到了`WindowManagerGlobal`的`addView`方法，具体调用流程可以看我文章开头的文章：

```
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。最后调用ViewRootImpl的setView方法。到这里WindowManager.LayoutParams这个参数依旧没有被设置token，那么最大的可能性就是在`adjustLayoutParamsForSubWindow`方法中了，马上进去看看：

```
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;
        }
        ...
    }
    ...
}
```

终于看到了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。

应用窗口直接获取的是与WindowManager对应的PhoneWindow的mAppToken，而子窗口是拿到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，就必须到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。这里需要对Activity的启动流程有一定的了解。追踪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);
    ...
}
```

这个方法调用了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以及对应的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。到这里我们知道Activity的PhoneWindow是拥有token的。那么Application呢?

---

Application调用的是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拥有自己PhoneWindow以及WindowManager，同时它的PhoneWindow拥有token；而Application并没有自己的PhoneWindow，他返回的WindowManager是应用服务windowManager，并没有赋值token的过程。**

那么到这里结论已经快要出来了，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来添加dialog的时候，此时Activity的DecorView已经是添加到屏幕上了，也就是我们的Activity是有界面了，这个情况下，他就是属于子窗口的类型被添加到PhoneWindow中，而他的token就是DecorView的token，此时DecorView已经被添加到屏幕上，他本身是拥有token的；

> 这里补充一点。当一个view（view树）被添加到屏幕上后，他所对应的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展示到屏幕后，后续添加的子window的token，就都是这个IWindowSession 对象了。

而如果是第一次添加，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就是null，后面到WMS也就会抛出异常了。而这也就是为什么使用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

到这里我们已经知道。我们从WMS的token判断找到了token的类型以及token的载体：WindowManager.LayoutParams，然后我们再从dialog的创建流程追到了赋值token的时候会因为windowManager的不同而不同。因此我们再去查看了两者不同的windowManager，最终得到结论**Activity的PhoneWindow拥有token，而Application使用的是应用级服务windowManager，并没有token。**

那么此时还是会有疑问：

- token到底是在什么时候被创建的?
- WMS怎么知道我这个token是合法的?

虽然到目前我们已经弄清原因，但是知识却少了一块，秉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.jav
a

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：

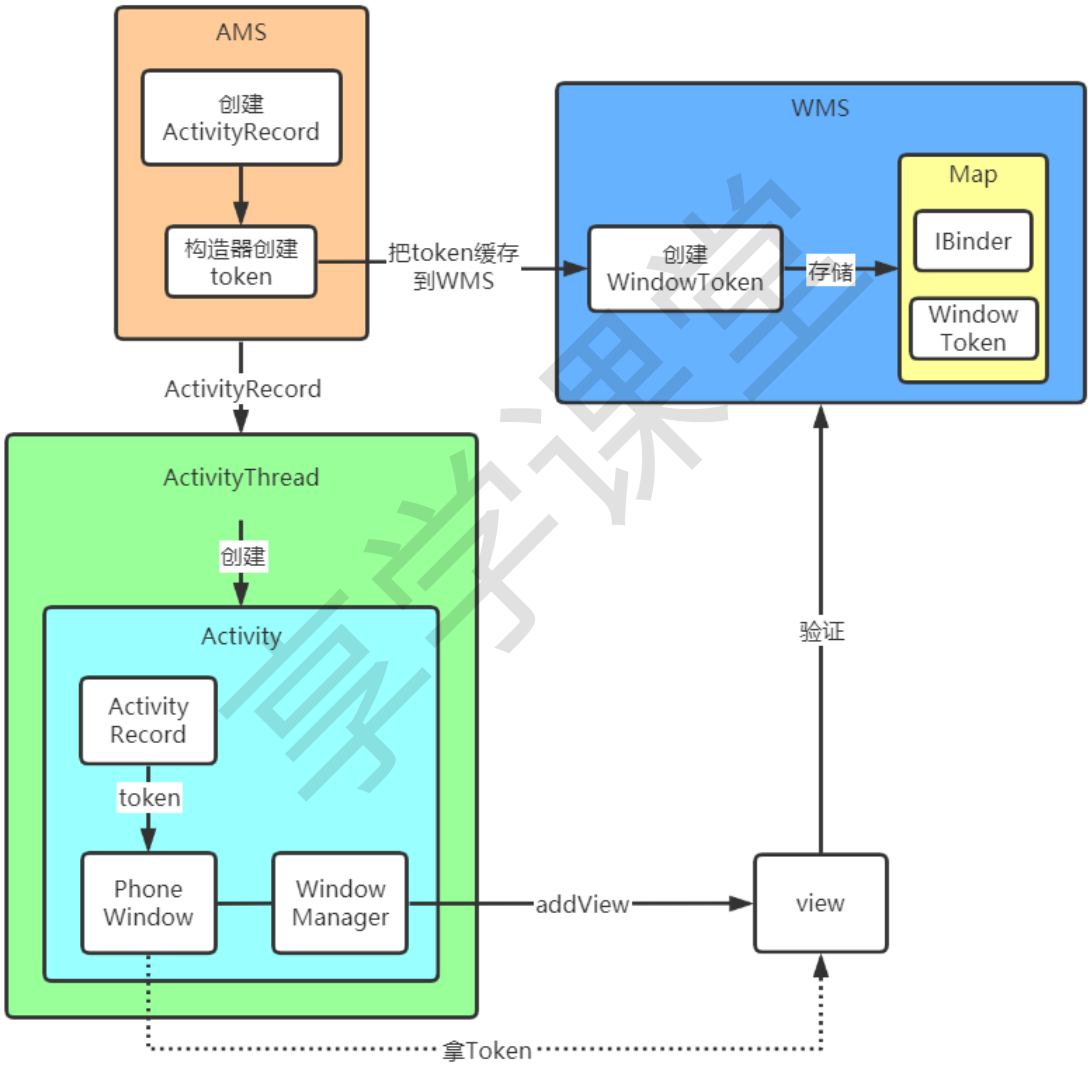
```
WindowToken.class
void onDisplayChanged(DisplayContent dc) {
    dc.reParentWindowToken(this);
    ...
}

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。

---