Android应用程序启动Binder线程源码分析

标签： Binder Android spawnPooledThread startThreadPool joinThreadPool

2013-07-11 14:39 1786人阅读 评论(0) 收藏 举报

分类：

【Android Binder通信】（8）

版权声明：本文为博主原创文章，未经博主允许不得转载。

Android的应用程序包括Java应用及本地应用，Java应用运行在davik虚拟机中，由zygote进程来创建启动，而本地服务应用在Android系统启动时，通过配置init.rc文件来由Init进程启动。Zygote启动Android应用程序的过程请查看文章Zygote孵化应用进程过程的源码分析，关于本地应用服务的启动过程在Android Init进程源码分析中有详细的介绍。无论是Android的Java应用还是本地服务应用程序，都支持Binder进程间通信机制，本文将介绍Android应用程序是如何启动Binder线程来支持Binder进程间通信的相关内容。

在zygote启动Android应用程序时，会调用zygoteInit函数来初始化应用程序运行环境，比如虚拟机堆栈大小，Binder线程的注册等

[cpp]

```cpp
public static final void zygoteInit(int targetSdkVersion, String[] argv)
        throws ZygoteInit.MethodAndArgsCaller {
    redirectLogStreams();
    commonInit();
    //启动Binder线程池以支持Binder通信
    nativeZygoteInit();
    applicationInit(targetSdkVersion, argv);
}
```

nativeZygoteInit函数用于创建线程池，该函数是一个本地函数，其对应的JNI函数为

frameworks\base\core\jni\AndroidRuntime.cpp

[cpp] 载

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

变量gCurRuntime的类型是AndroidRuntime，AndroidRuntime类的onZygoteInit()函数是一个虚函数，在AndroidRuntime的子类AppRuntime中被实现

frameworks\base\cmds\app_process\App_main.cpp

[cpp] 载

```cpp
virtual void onZygoteInit()
{
    sp<ProcessState> proc = ProcessState::self();
    ALOGV("App process: starting thread pool.\n");
    proc->startThreadPool();
}
```

函数首先得到ProcessState对象，然后调用它的startThreadPool()函数来启动线程池。

[cpp]

```cpp
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);
    if (!mThreadPoolStarted) {
        mThreadPoolStarted = true;
        spawnPooledThread(true);
    }
}
```

mThreadPoolStarted是线程池启动标志位，在startThreadPool()函数中被设置为true

[cpp]

```cpp
void ProcessState::spawnPooledThread(bool isMain)
{
    if (mThreadPoolStarted) {
        //统计启动的Binder线程数量
        int32_t s = android_atomic_add(1, &mThreadPoolSeq);
        char buf[16];
        snprintf(buf, sizeof(buf), "Binder_%X", s);
        ALOGV("Spawning new pooled thread, name=%s\n", buf);
        //创建一个PoolThread线程
        sp<Thread> t = new PoolThread(isMain);
        //启动线程
        t->run(buf);
    }
}
```

PoolThread是Thread的子类，PoolThread类的定义如下

[cpp]

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

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

    const bool mIsMain;
};
```

通过t->run(buf)来启动该线程，并且重写了线程执行函数threadLoop()，当线程启动运行后，threadLoop()被调用执行

[cpp]

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

直接执行joinThreadPool(mIsMain)函数将线程注册到Binder驱动程序中，mIsMain = true表示当前线程是主线程

[cpp]

```cpp
void IPCThreadState::joinThreadPool(bool isMain)
{

    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
    //设置线程组
    androidSetThreadSchedulingGroup(mMyThreadId, ANDROID_TGROUP_DEFAULT);
    status_t result;
    do {
        int32_t cmd;
        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());
                }
                mPendingWeakDerefs.clear();
            }

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

        //通知Binder驱动线程进入循环执行
        result = talkWithDriver();
        if (result >= NO_ERROR) {
            size_t IN = mIn.dataAvail();
            if (IN < sizeof(int32_t)) continue;
            //读取并执行Binder驱动返回来的命令
            cmd = mIn.readInt32();
            result = executeCommand(cmd);
        }
        androidSetThreadSchedulingGroup(mMyThreadId, ANDROID_TGROUP_DEFAULT);
        // 如果该线程不是主线程并且不在需要该线程时，线程退出
        if(result == TIMED_OUT && !isMain) {
            break;
        }
    } while (result != -ECONNREFUSED && result != -EBADF);
    //通知Binder驱动线程退出
    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
```

函数首先向IPCThreadState对象的mOut Parcel对象中写入BC_ENTER_LOOPER Binder协议命，该命令告诉Binder驱动该线程进入循环执行状态

[cpp]

```
mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
```

然后调用函数result = talkWithDriver()将mOut中的数据发送到Binder驱动程序中

[cpp]

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    ALOG_ASSERT(mProcess->mDriverFD >= 0, "Binder driver is not opened");

    binder_write_read bwr;
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

    bwr.write_size = outAvail;
    bwr.write_buffer = (long unsigned int)mOut.data();

    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (long unsigned int)mIn.data();
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }

    // Return immediately if there is nothing to do.
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;

    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
#if defined(HAVE_ANDROID_OS)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        else
            err = -errno;
#else
        err = INVALID_OPERATION;
#endif
    } while (err == -EINTR);

    if (err >= NO_ERROR) {
        if (bwr.write_consumed > 0) {
            if (bwr.write_consumed < (ssize_t)mOut.dataSize())
                mOut.remove(0, bwr.write_consumed);
            else
                mOut.setDataSize(0);
        }
        if (bwr.read_consumed > 0) {
            mIn.setDataSize(bwr.read_consumed);
            mIn.setDataPosition(0);
        }
        return NO_ERROR;
    }
    return err;
}
```

通过ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr)进入Binder驱动中，此时执行的Binder命令为BINDER_WRITE_READ，发送给Binder驱动的数据保存在binder_write_read结构体中

发送的数据为

bwr.write_size = outAvail;

bwr.write_buffer = (long unsigned int)mOut.data();

bwr.read_size = mIn.dataCapacity();

bwr.read_buffer = (long unsigned int)mIn.data();

在执行binder_ioctl()函数时先执行Binder驱动写在执行Binder驱动读操作

[cpp]

```cpp
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int ret;
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread;
    unsigned int size = _IOC_SIZE(cmd);
    void __user *ubuf = (void __user *)arg;
    /*printk(KERN_INFO "binder_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);*/
    ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret)
        return ret;

    mutex_lock(&binder_lock);
    thread = binder_get_thread(proc);
    if (thread == NULL) {
        ret = -ENOMEM;
        goto err;
    }

    switch (cmd) {
    case BINDER_WRITE_READ: {
        struct binder_write_read bwr;
        if (size != sizeof(struct binder_write_read)) {
            ret = -EINVAL;
            goto err;
        }
        if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
            ret = -EFAULT;
            goto err;
        }
        if (bwr.write_size > 0) {
            ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer, bwr.write_si
            if (ret < 0) {
                bwr.read_consumed = 0;
                if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                    ret = -EFAULT;
                goto err;
            }
        }
        if (bwr.read_size > 0) {
            ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer, bwr.read_size,
>f_flags & O_NONBLOCK);
            if (!list_empty(&proc->todo))
                wake_up_interruptible(&proc->wait);
            if (ret < 0) {
                if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                    ret = -EFAULT;
                goto err;
            }
        }
        if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
            ret = -EFAULT;
            goto err;
        }
        break;
    }
    default:
        ret = -EINVAL;
        goto err;
    }
    ret = 0;
err:
    if (thread)
        thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
    mutex_unlock(&binder_lock);
    wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret && ret != -ERESTARTSYS)
        printk(KERN_INFO "binder: %d:%d ioctl %x %lx returned %d\n", proc->pid, current->pid, cmd, arg, ret);
    return ret;
}
```

在内核数据发送缓冲区中保存了BC_ENTER_LOOPER命令，因此在执行binder_thread_write函数时，只处理BC_ENTER_LOOPER命令

[cpp]

```
int binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
            void __user *buffer, int size, signed long *consumed)
{
```

```cpp
    uint32_t cmd;
    void __user *ptr = buffer + *consumed;
    void __user *end = buffer + size;

    while (ptr < end && thread->return_error == BR_OK) {
        if (get_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
        if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
            binder_stats.bc[_IOC_NR(cmd)]++;
            proc->stats.bc[_IOC_NR(cmd)]++;
            thread->stats.bc[_IOC_NR(cmd)]++;
        }
        switch (cmd) {
        case BC_ENTER_LOOPER:
            if (thread->looper & BINDER_LOOPER_STATE_REGISTERED) {
                thread->looper |= BINDER_LOOPER_STATE_INVALID;
            }
            thread->looper |= BINDER_LOOPER_STATE_ENTERED;
            break;
        default:
            printk(KERN_ERR "binder: %d:%d unknown command %d\n",
                   proc->pid, thread->pid, cmd);
            return -EINVAL;
        }
        *consumed = ptr - buffer;
    }
    return 0;
}
```

BC_ENTER_LOOPER命令下的处理非常简单，仅仅是将当前线程binder_thread的状态标志位设置为BINDER_LOOPER_STATE_ENTERED，binder_thread_write函数执行完后，由于bwr.read_size > 0，因此binder_ioctl()函数还会执行Binder驱动读

[cpp]

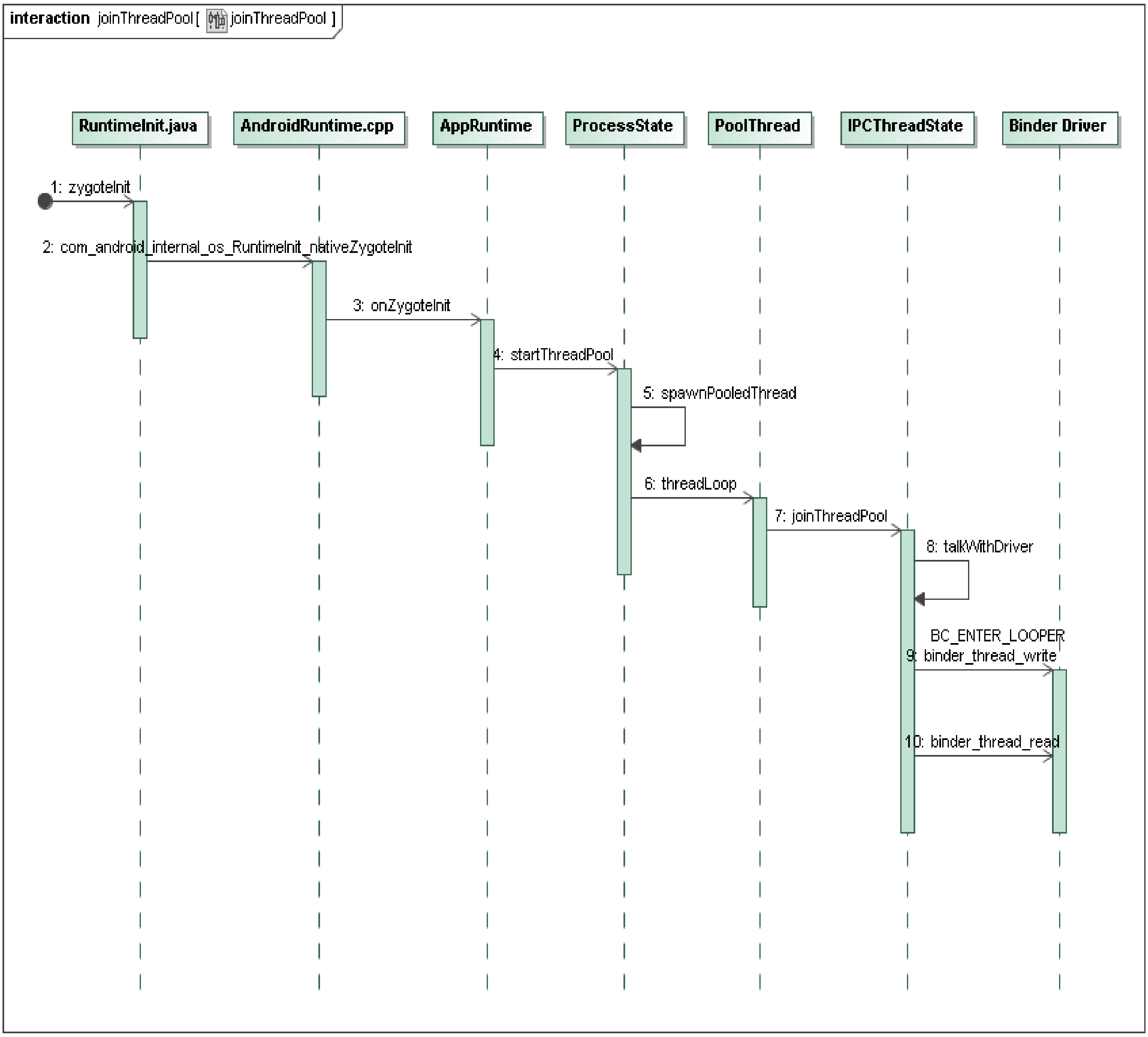
```cpp
static int binder_thread_read(struct binder_proc *proc,
                  struct binder_thread *thread,
                  void  __user *buffer, int size,
                  signed long *consumed, int non_block)
{
    void __user *ptr = buffer + *consumed;
    void __user *end = buffer + size;

    int ret = 0;
    int wait_for_proc_work;
    //向用户空间发送一个BR_NOOP
    if (*consumed == 0) {
        if (put_user(BR_NOOP, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
    }
retry:
    //由于当前线程首次注册到Binder驱动中，因此事务栈和待处理队列都为空，wait_for_proc_work = true
    wait_for_proc_work = thread->transaction_stack == NULL && list_empty(&thread->todo);
    //在初始化binder_thread时，return_error被初始化为BR_OK，因此这里为false
    if (thread->return_error != BR_OK && ptr < end) {
        if (thread->return_error2 != BR_OK) {
            if (put_user(thread->return_error2, (uint32_t __user *)ptr))
                return -EFAULT;
            ptr += sizeof(uint32_t);
            if (ptr == end)
                goto done;
            thread->return_error2 = BR_OK;
        }
        if (put_user(thread->return_error, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
        thread->return_error = BR_OK;
        goto done;
    }
    //设置当前线程的运行状态为BINDER_LOOPER_STATE_WAITING
    thread->looper |= BINDER_LOOPER_STATE_WAITING;
    if (wait_for_proc_work)
        proc->ready_threads++;
    mutex_unlock(&binder_lock);
    if (wait_for_proc_work) {
        //在注册Binder线程时已经设置为BINDER_LOOPER_STATE_ENTERED，因此这里的条件为false
        if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED | BINDER_LOOPER_STATE_ENTERED))) {
            wait_event_interruptible(binder_user_error_wait,binder_stop_on_user_error < 2);
        }
        //设置线程默认优先级
        binder_set_nice(proc->default_priority);
        if (non_block) {
            if (!binder_has_proc_work(proc, thread))
                ret = -EAGAIN;
        } else
            //在为当前线程创建binder_thread时，线程状态标志位被初始化为BINDER_LOOPER_STATE_NEED_RETURN，因此binder_has_proc_work函数返回true，当前线程睡眠在当前进程的等待队列中
            ret = wait_event_interruptible_exclusive(proc->wait, binder_has_proc_work(proc, thread));
    } else {
        ...
    }
    mutex_lock(&binder_lock);
    if (wait_for_proc_work)
        proc->ready_threads--;
    thread->looper &= ~BINDER_LOOPER_STATE_WAITING;
    if (ret)
        return ret;
    while (1) {
        uint32_t cmd;
        struct binder_transaction_data tr;
        struct binder_work *w;
        struct binder_transaction *t = NULL;
        if (!list_empty(&thread->todo))
            w = list_first_entry(&thread->todo, struct binder_work, entry);
        else if (!list_empty(&proc->todo) && wait_for_proc_work)
            w = list_first_entry(&proc->todo, struct binder_work, entry);
        else {
            if (ptr - buffer == 4 && !(thread-
>looper & BINDER_LOOPER_STATE_NEED_RETURN)) /* no data added */
                goto retry;
            break;
        }
        if (end - ptr < sizeof(tr) + 4)
            break;

        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:
            break;
    }

done:
    *consumed = ptr - buffer;
    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++;
        if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer))
            return -EFAULT;
    }
    return 0;
}
```

这样就将当前线程注册到了Binder驱动中，同时该线程进入睡眠等待客户端请求，当有客户端请求到来时，该Binder线程被唤醒，接收并处理客户端的请求。因此Android应用程序通过注册Binder线程来支持Binder进程间通信机制。


interaction joinThreadPool [joinThreadPool]
RuntimeInit.java
AndroidRuntime.cpp
AppRuntime
ProcessState
PoolThread
IPCThreadState
Binder Driver
1: zygoteInit
2: com_android_internal_os_RuntimeInit_nativeZygoteInit
3: onZygoteInit
4: startThreadPool
5: spawnPooledThread
6: threadLoop
7: joinThreadPool
8: talkWithDriver
BC_ENTER_LOOPER
9: binder_thread_write
10: binder_thread_read