【安卓源码】Binder机制3 -- Binder线程池

原创 蜘蛛侠不会飞 于 2023-04-26 21:16:44 发布 阅读量1.4k 收藏 2 点赞数
分类专栏： 安卓源码解析 文章标签： android binder 安卓源码 binder线程池 framework

Binder 本身是C/S架构，就可能存在多个Client会同时访问Server的情况。 在这种情况下，如果Server只有一个线程处理响应，就会导致客户端的请求要排队而导致响应过慢的现象发生。解决这个问题的方法就是引入多线程。【多个客户端不同线程去请求，服务端需要使用多线程机制，binder线程池一个线程去回复多个客户端的请求】

Binder机制 的设计从最底层–驱动层，就考虑到了对于多线程的支持。具体内容如下：

> a. 使用 Binder 的进程在启动之后，通过 BINDER_SET_MAX_THREADS 告知驱动其支持的最大线程数量
> b. 驱动会对线程进行管理。在 binder_proc 结构中，这些字段记录了进程中线程的信息：max_threads，requested_threads，requested_threads_started，ready_threads
> c. binder_thread 结构对应了 Binder 进程中的线程
> d. 驱动通过 BR_SPAWN_LOOPER 命令告知进程需要创建一个新的线程
> c. 进程通过 BC_ENTER_LOOPER 命令告知驱动其主线程已经ready
> d. 进程通过 BC_REGISTER_LOOPER 命令告知驱动其子线程（非主线程）已经ready
> e. 进程通过 BC_EXIT_LOOPER 命令告知驱动其线程将要退出
> f. 在线程退出之后，通过 BINDER_THREAD_EXIT 告知Binder驱动。驱动将对应的 binder_thread 对象销毁

1. 最大的binder 数量

在每个进程启动时候，都会创建 ProcessState 对象，获得ProcessState对象是单例模式，从而保证每一个进程只有一个ProcessState对象。因此一个binder设备一次,其中ProcessState的成员变量mDriverFD记录binder驱动的fd，用于访问binder设备。

/frameworks/native/libs/binder/ProcessState.cpp

```
// 在创建 ProcessState 对象的时候，会去打开driver
sp<ProcessState> ProcessState::self()
{
    Mutex::Autolock _l(gProcessMutex);
    if (gProcess != nullptr) {
        return gProcess;
    }
    gProcess = new ProcessState(kDefaultDriver);
    return gProcess;
}

// 打开驱动设备
static int open_driver(const char *driver)
{
    int fd = open(driver, O_RDWR | O_CLOEXEC);
    if (fd >= 0) {
        int vers = 0;
        status_t result = ioctl(fd, BINDER_VERSION, &vers);
        if (result == -1) {
            ALOGE("Binder ioctl to obtain version failed: %s", strerror(errno));
            close(fd);
            fd = -1;
        }
        if (result != 0 || vers != BINDER_CURRENT_PROTOCOL_VERSION) {
          ALOGE("Binder driver protocol(%d) does not match user space protocol(%d)! ioctl() return value: %d",
                vers, BINDER_CURRENT_PROTOCOL_VERSION, result);
            close(fd);
            fd = -1;
        }

```

```
// 设置最大的线程数量为：15                32 // #define DEFAULT_MAX_BINDER_THREADS 15
        size_t maxThreads = DEFAULT_MAX_BINDER_THREADS;

// 与binder 驱动交互，设置驱动的线程数量为 15个
        result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
        if (result == -1) {
            ALOGE("Binder ioctl to set max threads failed: %s", strerror(errno));
        }
    } else {
        ALOGW("Opening '%s' failed: %s\n", driver, strerror(errno));
    }
    return fd;
}
```

与binder 驱动交互，设置最大线程数量

/drivers/staging/android/binder.c

```
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;


    trace_binder_ioctl(cmd, arg);

    ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret)
        goto err_unlocked;

    binder_lock(__func__);

// 这里可以通过进程获取对应的线程
    thread = binder_get_thread(proc);
    if (thread == NULL) {
        ret = -ENOMEM;
        goto err;
    }

    switch (cmd) {

。。。
    case BINDER_SET_MAX_THREADS:

// 保存到 proc->max_threads
        if (copy_from_user(&proc->max_threads, ubuf, sizeof(proc->max_threads))) {
            ret = -EINVAL;
            goto err;
        }
        break;
```

设置 binder_proc 结构体的 max_threads 为 15，结构体为如下：

```
struct binder_proc {
    struct hlist_node proc_node;

// 使用红黑树保存 threads 线程
    struct rb_root threads;

// 进程的pid 号
    int pid;

// 进程需要做的事项
    struct list_head todo;

```

```
// 保存最大的线程数量    int max_threads;

// 请求的线程数量
    int requested_threads;
    int requested_threads_started;
    int ready_threads;

};
```

2. binder 主线程的创建

进程调用下列 startThreadPool 方法，去启动binder 主线程

ProcessState::self()->startThreadPool();

/frameworks/native/libs/binder/ProcessState.cpp

```
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);

// mThreadPoolStarted 初始值为 false
    if (!mThreadPoolStarted) {

// 设置为true，走到 spawnPooledThread(true)
        mThreadPoolStarted = true;
        spawnPooledThread(true);
    }
}
```

走到 spawnPooledThread(true)

```
void ProcessState::spawnPooledThread(bool isMain)
{

// 如果没有执行：startThreadPool，则 mThreadPoolStarted 为false，不走下列的代码
// 此时是为 true 的
    if (mThreadPoolStarted) {

// 设置binder thread 的名字
        String8 name = makeBinderThreadName();
        ALOGV("Spawning new pooled thread, name=%s\n", name.string());

// 创建一个线程PoolThread，isMain 为true 表示是主线程
        sp<Thread> t = new PoolThread(isMain);
// run 这个线程
        t->run(name.string());
    }
}

==========
String8 ProcessState::makeBinderThreadName() {
    int32_t s = android_atomic_add(1, &mThreadPoolSeq);
    pid_t pid = getpid();
    String8 name;

// 主线程的binder 名字为：Binder:pid号_1，如：Binder:9320_1
    name.appendFormat("Binder:%d_%X", pid, s);
    return name;
}
```

创建一个线程PoolThread，isMain 为true 表示是主线程

PoolThread 继承了 Thread：

/system/core/libutils/include/utils/threads.h

```
28  #include <utils/AndroidThreads.h>
29
30  #ifdef __cplusplus
31  #include <utils/Condition.h>
32  #include <utils/Errors.h>
33  #include <utils/Mutex.h>
34  #include <utils/RWLock.h>
35  #include <utils/Thread.h>
36  #endif
37
38  #endif // _LIBS_UTILS_THREADS_H
```

/system/core/libutils/Threads.cpp

```
status_t Thread::run(const char* name, int32_t priority, size_t stack)
{
    LOG_ALWAYS_FATAL_IF(name == nullptr, "thread name not provided to Thread::run");

    Mutex::Autolock _l(mLock);

    if (mRunning) {
        // thread already started
        return INVALID_OPERATION;
    }

    // reset status and exitPending to their default value, so we can
    // try again after an error happened (either below, or in readyToRun())
    mStatus = OK;
    mExitPending = false;
    mThread = thread_id_t(-1);

    // hold a strong reference on ourself
    mHoldSelf = this;

    mRunning = true;

    bool res;
    if (mCanCallJava) {
        res = createThreadEtc(_threadLoop,
                this, name, priority, stack, &mThread);
    } else {
        res = androidCreateRawThreadEtc(_threadLoop,
                this, name, priority, stack, &mThread);
    }

======
int Thread::_threadLoop(void* user)
{
    Thread* const self = static_cast<Thread*>(user);

    sp<Thread> strong(self->mHoldSelf);
    wp<Thread> weak(strong);
    self->mHoldSelf.clear();

#if defined(__ANDROID__)
    // this is very useful for debugging with gdb
    self->mTid = gettid();
#endif

    bool first = true;

    do {
        bool result;
        if (first) {
            first = false;
            self->mStatus = self->readyToRun();
            result = (self->mStatus == OK);

            if (result && !self->exitPending()) {
```

```
                // Binder threads (and maybe others) rely on threadLoop
                // running at least once after a successful ::readyToRun()
                // (unless, of course, the thread has already been asked to exit                // at that point).
                // This is because threads are essentially used like this:
                //   (new ThreadSubclass())->run();
                // The caller therefore does not retain a strong reference to
                // the thread and the thread would simply disappear after the
                // successful ::readyToRun() call instead of entering the
                // threadLoop at least once.
                result = self->threadLoop();
            }
        } else {
            result = self->threadLoop();
        }
```

执行run 方法，循环回调 threadLoop 方法

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

protected:
    virtual bool threadLoop()
    {

// 一个线程只有一个 IPCThreadState 实例，调用 joinThreadPool 方法
        IPCThreadState::self()->joinThreadPool(mIsMain);
        return false;
    }

    const bool mIsMain;
};
```

调用 joinThreadPool 方法

/frameworks/native/libs/binder/IPCThreadState.cpp

```
void IPCThreadState::joinThreadPool(bool isMain)
{
    LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREAD POOL\n", (void*)pthread_self(), getpid());


// 如果 isMain 为true ，则为 BC_ENTER_LOOPER
// false 为：BC_REGISTER_LOOPER
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);

    status_t result;
    do {
        processPendingDerefs();
        // now get the next command to be processed, waiting if necessary

// talkwithdriver 去驱动设备交互，执行命令
        result = getAndExecuteCommand();

        if (result < NO_ERROR && result != TIMED_OUT && result != -ECONNREFUSED && result != -EBADF) {
            ALOGE("getAndExecuteCommand(fd=%d) returned unexpected error %d, aborting",
                  mProcess->mDriverFD, result);
            abort();
        }

        if(result == TIMED_OUT && !isMain) {
            break;
        }
```

```
    } while (result != -ECONNREFUSED && result != -EBADF);
```

binder 驱动对：BC_ENTER_LOOPER 的处理

```
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) {
        if (get_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
        trace_binder_command(cmd);

        switch (cmd) {

。。。。。。。。。
        case BC_ENTER_LOOPER:

// 如果 thread->looper 有设置 binder 普通线程的值：BINDER_LOOPER_STATE_REGISTERED，则回复error
            if (thread->looper & BINDER_LOOPER_STATE_REGISTERED) {
                thread->looper |= BINDER_LOOPER_STATE_INVALID;
                binder_user_error("%d:%d ERROR: BC_ENTER_LOOPER called after BC_REGISTER_LOOPER\n",
                    proc->pid, thread->pid);
            }

// 设置 thread->looper 为：BINDER_LOOPER_STATE_ENTERED：0x02，位运算来保存
            thread->looper |= BINDER_LOOPER_STATE_ENTERED;
            break;

// 如果是退出线程的话，则设置为：BINDER_LOOPER_STATE_EXITED 0x04
        case BC_EXIT_LOOPER:
            binder_debug(BINDER_DEBUG_THREADS,
                     "%d:%d BC_EXIT_LOOPER\n",
                     proc->pid, thread->pid);
            thread->looper |= BINDER_LOOPER_STATE_EXITED;
            break;
```

3. binder 普通线程的创建

线程池是在service端，用于响应处理client端的众多请求。binder线程池中的线程都是由Binder驱动来控制创建的。

> 创建binder 普通线程是由binder 驱动控制的，**驱动通过 BR_SPAWN_LOOPER 命令告知进程需要创建一个新的线程**，然后**进程通过 BC_REGISTER_LOOPER 命令告知驱动其子线程**（非主线程）已经ready

service 端创建线程的2种情况：

> - BC_TRANSACTION：client进程向binderDriver发送IPC调用请求的时候。
> - BC_REPLY：client进程收到了binderDriver的IPC调用请求，逻辑执行结束后发送返回值。

首先客户端调用 IPCThreadState::transact：

- **客户端进程**

/frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::transact(int32_t handle, 2                                    uint32_t code, const Parcel& da
                                  Parcel* reply, uint32_t flags)
{
    status_t err;

    flags |= TF_ACCEPT_FDS;

// 封装data 值
    err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, nullptr);

        if (reply) {

// 与binder 驱动交互 waitForResponse
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
```

// 与binder 驱动交互 waitForResponse

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    uint32_t cmd;
    int32_t err;

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;

        cmd = (uint32_t)mIn.readInt32();

=======================
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    if (mProcess->mDriverFD <= 0) {
        return -EBADF;
    }

    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 = (uintptr_t)mOut.data();

    // This is what we'll read.
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (uintptr_t)mIn.data();
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }


    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(__ANDROID__)50
// 与binder 驱动交互
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
```

与binder 驱动交互：BC_TRANSACTION

```
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;

。。。。。

// 只有cmd 命令是 BC_TRANSACTION 和 BC_REPLY 才会调用 binder_transaction 函数
        case BC_TRANSACTION:
        case BC_REPLY: {
            struct binder_transaction_data tr;

            if (copy_from_user(&tr, ptr, sizeof(tr)))
                return -EFAULT;
            ptr += sizeof(tr);
            binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
            break;
        }
```

只有cmd 命令是 BC_TRANSACTION 和 BC_REPLY 才会调用 binder_transaction 函数

```
static void binder_transaction(struct binder_proc *proc,
                   struct binder_thread *thread,
                   struct binder_transaction_data *tr, int reply)
{
    struct binder_transaction *t;
    struct binder_work *tcomplete;
    binder_size_t *offp, *off_end;
    binder_size_t off_min;

。。。。。。
    }

// 设置工作类型为 BINDER_WORK_TRANSACTION，增加到工作的双向链表中
    t->work.type = BINDER_WORK_TRANSACTION;
    list_add_tail(&t->work.entry, target_list);
    tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
    list_add_tail(&tcomplete->entry, &thread->todo);

// 唤醒对应的进程处理
    if (target_wait)
        wake_up_interruptible(target_wait);
    return;
```

- **service 服务端处理消息**

```
static int binder_thread_read(struct binder_proc *proc,
                  struct binder_thread *thread,
                  binder_uintptr_t binder_buffer, size_t size,
                  binder_size_t *consumed, int non_block)
{
    void __user *buffer = (void __user *)(uintptr_t)binder_buffer;
    void __user *ptr = buffer + *consumed;
    void __user *end = buffer + size;

// 如果返回有 error，则有可能 执行到 done
    if (thread->return_error != BR_OK && ptr < end) {
```

```
        if (thread->return_error2 != BR_OK) {
                return -EFAULT;
            ptr += sizeof(uint32_t);
            binder_stat_br(proc, thread, thread->return_error2);
            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);
        binder_stat_br(proc, thread, thread->return_error);
        thread->return_error = BR_OK;
        goto done;
    }

。。。。。。
// 执行while 循环
    while (1) {
        uint32_t cmd;
        struct binder_transaction_data tr;
        struct binder_work *w;

// 初始值 t 为 null
        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 {
            /* no data added */
            if (ptr - buffer == 4 &&
                !(thread->looper & BINDER_LOOPER_STATE_NEED_RETURN))
                goto retry;
            break;
        }

        if (end - ptr < sizeof(tr) + 4)
            break;

        switch (w->type) {

// 执行 BINDER_WORK_TRANSACTION，t 不为空，不走cotinue，回走到 done 分支
        case BINDER_WORK_TRANSACTION: {
            t = container_of(w, struct binder_transaction, work);
        } break;
。。。。。
// 如果命令是 BR_DEAD_BINDER，也会走到 done
            if (cmd == BR_DEAD_BINDER)
                goto done; /* DEAD_BINDER notifications can cause transactions */
        } break;
        }

// 如果 t 为null，则执行 continue，不退出循环
        if (!t)
            continue;
。。。
        } else {
            t->buffer->transaction = NULL;
            kfree(t);
            binder_stats_deleted(BINDER_STAT_TRANSACTION);
        }
        break;

// 下列括号是 while 的括号
    }

done:
```

13

```
            if (put_user(thread->return_error2, (uint32_t __user *)p
```

```
84
85      *consumed = ptr - buffer;
86
87  // 需要满足 3 个条件：
88  // 1. 当前进程没有可请求的线程，也没有已经ready可用的线程
89  // 2. 启动的线程要小于 15；
90  // 3. 对应的client中的线程不能已经启动过。
91      if (proc->requested_threads + proc->ready_threads == 0 &&
92          proc->requested_threads_started < proc->max_threads &&
93          (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
94           BINDER_LOOPER_STATE_ENTERED)) /* the user-space code fails to */
95           /*spawn a new thread if we leave this out */) {
96
97  // 设置请求线程的数量 + 1
98          proc->requested_threads++;
99
100 // 拷贝 BR_SPAWN_LOOPER 到用户空间
101         if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer))
102             return -EFAULT;
103         binder_stat_br(proc, thread, BR_SPAWN_LOOPER);
104     }
105     return 0;
106 }
```

当发生以下3种情况之一，便会进入done分支：

> 1. 当前线程的return_error发生error的情况；
> 2. 当Binder驱动向client端发送死亡通知的情况；
> 3. 当类型为BINDER_WORK_TRANSACTION(即收到命令是BC_TRANSACTION或BC_REPLY)的情况；

线程的含义：

> 1. ready_threads： 表示当前线程池中有多少可用的空闲线程。
> 2. requested_threads：请求开启线程的数量。
> 3. requested_threads_started：表示当前已经接受请求开启的线程数量。

创建 Binder 普通线程的条件有3个：

> 1. 当前进程没有可请求的线程，也没有已经ready可用的线程
> 2. 启动的线程要小于 15；
> 3. 对应的client中的线程不能已经启动过

拷贝 BR_SPAWN_LOOPER 到用户空间，执行用户空间的代码创建普通线程：

/frameworks/native/libs/binder/IPCThreadState.cpp

```
1  status_t IPCThreadState::getAndExecuteCommand()
2  {
3      status_t result;
4      int32_t cmd;
5
6      result = talkWithDriver();
7      if (result >= NO_ERROR) {
8          size_t IN = mIn.dataAvail();
9          if (IN < sizeof(int32_t)) return result;
10         cmd = mIn.readInt32();
11         IF_LOG_COMMANDS() {
12             alog << "Processing top-level Command: "
13                  << getReturnString(cmd) << endl;
14         }
15
```

```
        pthread_mutex_lock(&mProcess->mThreadCountLock);
        if (mProcess->mExecutingThreadsCount >= mProcess->mMaxThreads &&
                mProcess->mStarvationStartTimeMs == 0) {
            mProcess->mStarvationStartTimeMs = uptimeMillis();
        }
        pthread_mutex_unlock(&mProcess->mThreadCountLock);

        result = executeCommand(cmd);
```

```
        mProcess->mExecutingThreadsCount++;
```

executeCommand

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;

    switch ((uint32_t)cmd) {

    case BR_SPAWN_LOOPER:
        mProcess->spawnPooledThread(false);
        break;
```

执行 mProcess->spawnPooledThread(false)

/frameworks/native/libs/binder/ProcessState.cpp

```
void ProcessState::spawnPooledThread(bool isMain)
{

// isMain 为false
    if (mThreadPoolStarted) {

// 设置binder 的名字：
        String8 name = makeBinderThreadName();
        ALOGV("Spawning new pooled thread, name=%s\n", name.string());

// 创建 PoolThread对象，指向run 方法
        sp<Thread> t = new PoolThread(isMain);
        t->run(name.string());
    }
}

=======
String8 ProcessState::makeBinderThreadName() {

// 递增加1
    int32_t s = android_atomic_add(1, &mThreadPoolSeq);
    pid_t pid = getpid();
    String8 name;

// 这里为：Binder:9032_2
    name.appendFormat("Binder:%d_%X", pid, s);
    return name;
}
```

创建 PoolThread对象，指向run 方法

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

protected:
```

```
    virtual bool threadLoop()
                                11      {
        IPCThreadState::self()->joinThreadPool(mIsMain);
        return false;
    }

    const bool mIsMain;
};
```

又回到：IPCThreadState::self()->joinThreadPool(false)，此 IPCThreadState 对象是个新的对象，与主线程的 IPCThreadState 是不同的。

/frameworks/native/libs/binder/IPCThreadState.cpp

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

// isMain  为false，与binder 驱动交互的命令是 BC_REGISTER_LOOPER
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);

    status_t result;
    do {

        result = getAndExecuteCommand();

        if (result < NO_ERROR && result != TIMED_OUT && result != -ECONNREFUSED && result != -EBADF) {
            ALOGE("getAndExecuteCommand(fd=%d) returned unexpected error %d, aborting",
                  mProcess->mDriverFD, result);
            abort();
        }

        // Let this thread exit the thread pool if it is no longer
        // needed and it is not the main process thread.

// 如果普通线程返回的结果是 TIMED_OUT，则回收该普通线程
        if(result == TIMED_OUT && !isMain) {
            break;
        }
    } while (result != -ECONNREFUSED && result != -EBADF);

    LOG_THREADPOOL("**** THREAD %p (PID %d) IS LEAVING THE THREAD POOL err=%d\n",
        (void*)pthread_self(), getpid(), result);

// 通知binder 驱动线程退出了：BC_EXIT_LOOPER
    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
```

isMain 为false，与binder 驱动交互的命令是 BC_REGISTER_LOOPER

```
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) {
        if (get_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);

        switch (cmd) {

。。。。。。。
        case BC_REGISTER_LOOPER:

```

```
// 如果是主线程，则报错。
                    if (thread->looper & BINDER_LOOPER_STATE_ENTERED) {
                thread->looper |= BINDER_LOOPER_STATE_INVALID;
                binder_user_error("%d:%d ERROR: BC_REGISTER_LOOPER called after BC_ENTER_LOOPER\n",
                    proc->pid, thread->pid);

// 如果binder 驱动都没有请求创建线程，则报错
            } else if (proc->requested_threads == 0) {
                thread->looper |= BINDER_LOOPER_STATE_INVALID;
                binder_user_error("%d:%d ERROR: BC_REGISTER_LOOPER called without request\n",
                    proc->pid, thread->pid);
            } else {

// requested_threads减1， requested_threads_started开启的线程增加为 1；
                proc->requested_threads--;
                proc->requested_threads_started++;
            }

// 设置looper 模式
            thread->looper |= BINDER_LOOPER_STATE_REGISTERED;
            break;
```

Binder机制中的收发消息及线程池 - 腾讯云开发者社区-腾讯云

进程的Binder线程池工作过程-移动端开发

Android、java面试技巧及常见性面试题型精编汇总.zip

Android、java面试技巧及常见性面试题型精编汇总.zip Java；基础知识点面试专题 java；深入源码级的面试题 大厂高端技术面试专题（有独立项目） 多线程面试专题及答

Android 跨进程通信-（十）Binder机制传输数据限制—罪魁祸首Binder线程池 nihaomabmt的专

前言 在Android 跨进程通信-（三）Binder机制之Client中提到2.APP进程初始化在通过ProcessState来获取驱动设备文件"/dev/binder/"文件描述符，并且在用户和内核的虚

Android跨进程通信:图文详解 Binder机制 原理_android binder-CSDN...

Server进程会创建很多线程来处理Binder请求 Binder模型的线程管理 采用Binder驱动的线程池,并由Binder驱动自身进行管理 而不是由Server进程来管理的 一个进程的Binc

Android 面试必备:高工必问Binder机制~_binder机制原理面试

Binder线程池是ServiceManager提供的,利用的是Binder内核机制。 Binder机制是安卓为了提供更高效、稳定、可靠的方式实现的一套基于内核的IPC机制。 为什么zygote

APP性能设计及优化专题——影响性能的不良实现 software_test010的

本文将重点介绍影响性能的不良实现，主要包含Binder共享内存耗尽、Binder线程池耗尽、创建大量BpBinder或Binder对象等方面。为了更好地了解下文内容，我们先简单

由浅入深 学习 Android Binder（十一) binder线程池 许佳佳的博

Android Binder系列文章： 由浅入深 学习 Android Binder（一）- AIDL 由浅入深 学习 Android Binder（二）- bindService流程 由浅入深 学习 Android Binder（三）- java I

Binder系列10 Binder线程池管理_binder_set_max_threads

且本次创建的是 Binder 主线程,以变量 isMain 为 true 为标志. 其余 Binder 线程池中的线程都是由 Binder 驱动通过发送 BR_SPAWN_LOOPER 命令来通知应用进程创建的

Android Binder原理(一)学习Binder前必须要了解的知识点

Binder是基于内存映射来实现的,在前面我们知道内存映射通常是用在有物理介质的文件系统上的,Binder没有物理介质,它使用内存映射是为了跨进程传递数据。 Binder通信

binder线程池demo

demo 是根据开发艺术探索这本书写的、主要讲解了线程池的使用

Android Binder 线程池和线程池工作流程 低头赶路，敬事如

介绍了android binder线程池的常见类和线程池的工作方式

Android Binder机制浅谈以及使用Binder进行跨进程通信的俩种方式(AI...

Binder模型的线程管理 采用Binder驱动的线程池,并由Binder驱动自身进行管理,而不是由Server进程来管理的 一个进程的Binder线程数默认最大是16,超过的请求会被阻塞等

Binder机制深入理解

Binder基于Client-Server通信模式,传输过程只需一次拷贝,为发送发添加UID/PID身份,既支持实名Binder也支持匿名Binder,安全性高。 02.Binder工作流程 2.1 Binder运行机

进程的Binder线程池工作过程 weixin_33709609的

基于Android 6.0源码剖析，分析Binder线程池以及binder线程启动过程。 frameworks/base/cmds/app_process/app_main.cpp frameworks/native/libs/binder/ProcessState

Framework详解面试题之 为什么Android 要采用 Binder 作为 IPC 机制？和Binder线程池的工作过程是什么样？ bugyinyin的

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bind

Android 进阶——Binder IPC之Binder IPC架构及原理概述(九)