# Android Binder通信数据结构介绍

2013-06-27 19:32 1703人阅读 评论(1) 收藏 举报

分类：

【Android Binder通信】（8）



目录(?) [+]

## Binder通信进程描述——binder_proc

结构体binder_proc用来描述一个正在使用Binder进程间通信机制的进程。当一个进程调用函数open打开/dev/binder设备文件时，Binder驱动程序就会为该进程创建一个binder_proc结构体，并且保存在全局的binder_procs链表中。

```cpp
struct binder_proc {
    //挂载在全局binder_procs链表中的节点。
    struct hlist_node proc_node;
    //使用红黑树来保存使用Binder机制通信的进程的Binder线程池的线程ID
    struct rb_root threads;
    //使用红黑树来保存使用Binder机制通信的进程内所有Binder实体对象binder_node的成员变量ptr
    struct rb_root nodes;
    //使用红黑树来保存使用Binder机制通信的进程内所有Binder引用对象binder_ref的成员变量desc
    struct rb_root refs_by_desc;
    //使用红黑树来保存使用Binder机制通信的进程内所有Binder引用对象binder_ref的成员变量node
    struct rb_root refs_by_node;
    //保存使用Binder机制通信的进程内的pid
    int pid;
    //保存内核缓冲区在用户空间的地址
    struct vm_area_struct *vma;
    //保存使用Binder机制通信的进程信息
    struct task_struct *tsk;
    struct files_struct *files;//打开文件结构体
    //挂载在全局延迟工作项链表binder_deferred_list中的节点
    struct hlist_node deferred_work_node;
    //描述延迟工作项的具体类型
    int deferred_work;
    //表示要映射的物理内存在内核空间中的起始位置;
    void *buffer;
    //它表示的是内核使用的虚拟地址与进程使用的虚拟地址之间的差值
    ptrdiff_t user_buffer_offset;
    //指向被划分为若干小块的内核缓冲区
    struct list_head buffers;
    //指向没有分配物理页面的空闲小块内核缓冲区
    struct rb_root free_buffers;
    //指向已经分配了物理页面正在使用的小块内核缓冲区
    struct rb_root allocated_buffers;
    //保存当前可以用来保存异步事务数据的内核缓冲区的大小
    size_t free_async_space;
    //为内核缓冲区分配的物理页面
    struct page **pages;
    //保存Binder驱动程序为进程分配的内核缓冲区的大小
```

```
38.     size_t buffer_size;
39.     //保存空闲内核缓冲区的大小
40.     uint32_t buffer_free;
41.     //进程待处理工作项队列
42.     struct list_head todo;
43.     //空闲Binder线程会睡眠在wait描述的等待队列中
44.     wait_queue_head_t wait;
45.     //统计进程接收到的进程间通信请求次数
46.     struct binder_stats stats;
47.     //死亡通知队列
48.     struct list_head delivered_death;
49.     //保存Binder驱动程序最多可以主动请求进程注册的线程数量
50.     int max_threads;
51.     //记录请求注册的线程个数
52.     int requested_threads;
53.     //记录响应请求的线程个数
54.     int requested_threads_started;
55.     //保存进程当前空闲的Binder线程数目
56.     int ready_threads;
57.     //设置进程优先级
58.     long default_priority;
59.     struct dentry *debugfs_entry;
60. };
```

进程打开设备文件/dev/binder之后，还必须调用函数mmap将它映射到进程的地址空间来，实际上是请求Binder驱动程序为它分配一看内核缓冲区，以便用来接收进程间通信数据。Binder驱动程序为进程分配的内核缓冲区的大小保存在成员变量buffer_size中。这些内核缓冲区有内核空间地址和用户空间地址两个地址，内核空间地址保存在成员变量buffer中，用户空间地址保存在成员变量vma中，用户空间程序通过用户空间地址来访问内核缓冲区，内核缓冲区的内核空间地址和用户空间地址之间相差一个固定值，保存在成员变量user_buffer_offset中。这样就可以通过一个用户空间地址或内核空间地址来计算出另外一个地址的大小。这两个地址都是虚拟地址，对应的物理页面保存在成员变量pages中，Binder驱动程序开始时只为内核缓冲区分配一个物理页面。Binder驱动为了方便管理内核缓冲区，会将它划分成若干小块，使用binder_buffer来描述这些内核缓冲区，并且按地址从小到大保存在成员变量buffers指向的链表中。当使用这些内核缓冲区时就会为其分配物理页面，正在使用的内核缓冲区保存在allocated_buffers红黑树中，而空闲的内核缓冲区保存在free_buffers红黑树中。成员变量保存了空闲内核缓冲区的大小。每个使用了Binder通信机制的进程都有一个Binder线程池，用来处理进程间通信请求，这个Binder线程池由Binder驱动程序来维护，Binder线程池中的线程保存在threads红黑树上。进程可以使用ioctl函数将一个线程注册到Binder驱动程序中，当进程没有足够多的空闲线程来处理进程间通信请求时，Binder驱动程序可以主动要求进程注册更多的线程到Binder线程池中。Binder驱动程序最多可以主动请求进程注册的线程的数量保存在成员变量max_threads中，并不表示Binder线程池中的最大线程数目，成员变量ready_threads表示进程当前空闲Binder线程数目。Binder驱动程序每一次主动请求进程注册一个线程时，都会将成员变量requested_threads的值加1，当进程响应这个请求之后，Binder驱动程序就会将成员变量requested_threads的值减1，同时将requested_threads_started的值加1，表示驱动程序已经主动请求进程注册了多少个线程到Binder线程池中。当进程接收到一个进程间通信请求时，Binder驱动程序就将该请求封装成一个工作项，并且加入到进程的待处理队列todo中，Binder线程池中的空闲Binder线程会睡眠在wait所描述的一个等待队列中，当它们的宿主进程的待处理工作项队列增加了新的工作项之后，Binder驱动程序会唤醒这些线程去处理新的工作项，同时线程优先级被设置为default_priority值。一个进程包含一系列的Binder实体对象和Binder引用对象，这些Binder实体对象binder_node的成员变量ptr保存在成员变量nodes红黑树中，所有Binder引用对象的成员变量desc组织在

refs_by_desc红黑树中，所有Binder引用对象的成员变量node组织在refs_by_node红黑树中。Binder线程池中空闲Binder线程睡眠在一个等待队列中，进程可以通过调用flush函数来唤醒这些线程，以便它们可以检查进程是否有新的工作项需要处理。

## Binder线程描述——binder_thread

binder_thread用来描述Binder线程池中的一个线程

```cpp
struct binder_thread {
    //指向该binder线程的宿主进程
    struct binder_proc *proc;
    //挂载到宿主进程binder_proc的成员变量threads红黑树的节点
    struct rb_node rb_node;
    //binder线程pid
    int pid;
    //binder线程运行状态
    int looper;
    //需要线程处理的事务堆栈
    struct binder_transaction *transaction_stack;
    //binder线程待处理队列
    struct list_head todo;
    //binder线程处理出错的错误码
    uint32_t return_error; /* Write failed, return error code in read buf */
    uint32_t return_error2; /* Write failed, return error code in read */
    //Binder线程会睡眠在wait描述的等待队列中
    wait_queue_head_t wait;
    //统计该Binder线程接收到的进程间通信请求次数
    struct binder_stats stats;
};
```

成员变量proc指向该binder线程的宿主进程，成员变量rb_node用于挂载该binder线程到宿主进程的threads红黑树上，looper保存了该线程的运行状态，线程运行状态包括：

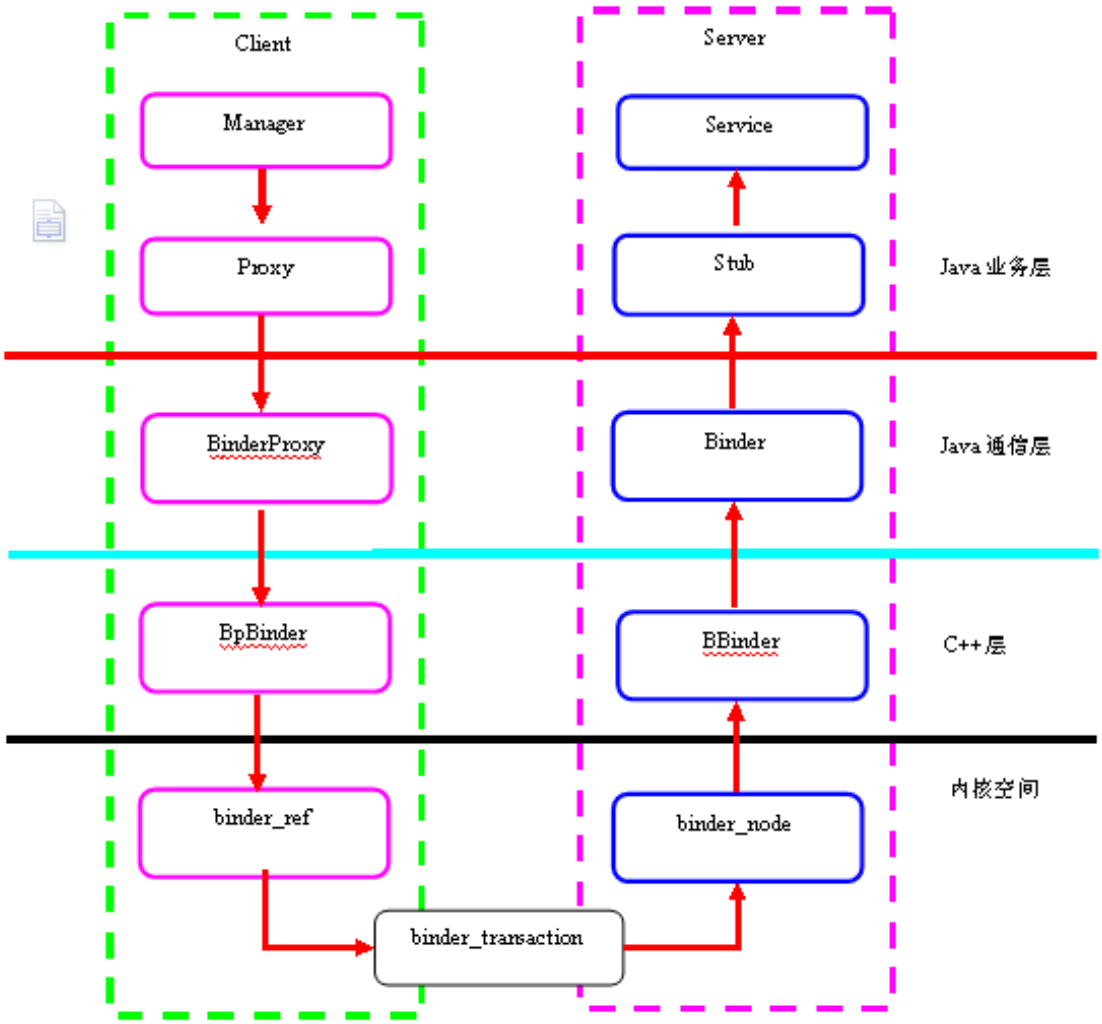
```cpp
enum {
    BINDER_LOOPER_STATE_REGISTERED  = 0x01,
    BINDER_LOOPER_STATE_ENTERED     = 0x02,
    BINDER_LOOPER_STATE_EXITED      = 0x04,
    BINDER_LOOPER_STATE_INVALID     = 0x08,
    BINDER_LOOPER_STATE_WAITING     = 0x10,
    BINDER_LOOPER_STATE_NEED_RETURN = 0x20
};
```

一个线程注册到Binder驱动时，Binder驱动程序就会为它建立一个binder_thread结构体，并且初始化线程状态为BINDER_LOOPER_STATE_NEED_RETURN，表示该线程需要马上返回到用户空间。当进程调用flush函数来刷新Binder线程池时，Binder线程池中的线程状态也会重置为BINDER_LOOPER_STATE_NEED_RETURN。如果一个线程是应用程序主动注册的，那么它就通过BC_ENTER_LOOPER协议来通知Binder驱动程序，表示该线程已经准备就绪，可以接受进程间通信请求了，同时将该Binder线程的状态设置为

BINDER_LOOPER_STATE_ENTERED；如果一个线程是Binder驱动程序请求创建的，那么它就通过BC_REGISTER_LOOPER协议来通知Binder驱动程序，这时候Binder驱动程序就会增加它所请求进程创建的Binder线程的数目，设置该Binder线程的状态为BINDER_LOOPER_STATE_REGISTERED。当一个Binder线程处于空闲状态时，Binder驱动程序就会把它的状态设置为BINDER_LOOPER_STATE_WAITING，而当Binder线程退出时，应用程序通过BC_EXIT_LOOPER协议来通知Binder驱动程序，并将线程状态设置为BINDER_LOOPER_STATE_EXITED；在线程出现异常情况下，Binder驱动程序会将该线程的状态设置为BINDER_LOOPER_STATE_INVALID。当客户进程请求要交给某一Binder线程处理时，就会将这个请求加入到该Binder线程的todo待处理队列，并唤醒该Binder线程；当Binder驱动将一个事务交个某个Binder线程处理时，就会将事务封装为binder_transaction结构体，并添加到transaction_stack线程事务堆栈中。当Binder线程空闲时，就睡眠在wait所描述的等待队列中。

## Binder实体对象——binder_node

binder_node用来描述一个Binder实体对象，每个Service组件在Binder驱动程序中都对应有一个Binder实体对象，用来描述它在内核中的状态。Android系统的Binder通信框架如下图所示：




```
[cpp]
01. struct binder_node {
02.     //调试id
03.     int debug_id;
04.     //描述一个待处理的工作项
```

```
    struct binder_work work;
    union {
        //挂载到宿主进程binder_proc的成员变量nodes红黑树的节点
        struct rb_node rb_node;
        //当宿主进程死亡，该binder实体对象将挂载到全局binder_dead_nodes链表中
        struct hlist_node dead_node;
    };
    //指向该binder线程的宿主进程
    struct binder_proc *proc;
    //保存所有引用该binder实体对象的binder引用对象
    struct hlist_head refs;
    //binder实体对象的强引用计数
    int internal_strong_refs;
    int local_strong_refs;
    unsigned has_strong_ref:1;
    unsigned pending_strong_ref:1;
    unsigned has_weak_ref:1;
    unsigned pending_weak_ref:1;
    //binder实体对象的弱引用计数
    int local_weak_refs;
    //指向用户空间service组件内部的引用计数对象wekref_impl的地址
    void __user *ptr;
    //保存用户空间的service组件地址
    void __user *cookie;
    //标示该binder实体对象是否正在处理一个异步事务
    unsigned has_async_transaction:1;
    //设置该binder实体对象是否可以接收包含有文件描述符的IPC数据
    unsigned accept_fds:1;
    //binder实体对象要求处理线程应具备的最小线程优先级
    unsigned min_priority:8;
    //异步事务队列
    struct list_head async_todo;
};
```

用户空间中的每一个Binder本地对象在Binder驱动中都对应有一个Binder实体对象，成员变量proc指向Binder实体对象的宿主进程，宿主进程使用红黑树来维护它内部的所有Binder实体对象，成员变量rb_node就是用来挂载到宿主进程proc的Binder实体对象红黑树中的节点；如果该Binder实体对象的宿主进程已经死亡，该Binder实体就通过成员变量dead_node保存到全局链表binder_dead_nodes。一个Binder实体对象可以被多个client引用，成员变量refs用来保存所有引用该Binder实体的Binder引用对象，internal_strong_refs和local_strong_refs都是用来描述Binder实体对象的强引用计数，而local_weak_refs则是用来描述Binder实体对象的弱引用计数。成员变量ptr和cookie分别指向用户空间地址，cookie指向BBinder的地址，ptr指向BBinder对象的引用计数地址。has_async_transaction用来描述一个Binder实体对象是否正在处理一个异步事务，当Binder驱动指定某个线程来处理某一事务时，首先将该事务保存到指定线程的todo队列中，表示要由该线程来处理该事务。如果是异步事务，Binder驱动程序就会将它保存在目标Binder实体对象的一个异步事务队列async_todo队列中。min_priority表示一个Binder实体对象在处理来自client进程请求时所要求处理线程的最小线程优先级。

## Binder引用对象——binder_ref

```
[cpp]

struct binder_ref {
    //调试id
```

```
    int debug_id;
    //挂载到宿主对象binder_proc的红黑树refs_by_desc中的节点
    struct rb_node rb_node_desc;
    //挂载到宿主对象binder_proc的红黑树refs_by_node中的节点
    struct rb_node rb_node_node;
    //挂载到Binder实体对象的refs链表中的节点
    struct hlist_node node_entry;
    //Binder引用对象的宿主进程binder_proc
    struct binder_proc *proc;
    //Binder引用对象所引用的Binder实体对象
    struct binder_node *node;
    //Binder引用对象的句柄值
    uint32_t desc;
    //强引用计数
    int strong;
    //弱引用计数
    int weak;
    //注册死亡接收通知
    struct binder_ref_death *death;
};
```

binder_ref用来描述一个Binder引用对象，每一个client在Binder驱动中都有一个binder引用对象。成员变量node保存该binder引用对象所引用的Binder实体对象，Binder实体对象使用链表保存了所有引用该实体对象的Binder引用对象，node_entry就是该Binder引用对象所引用的Binder实体对象的成员变量refs链表中的节点，desc是一个句柄值，用来描述一个Binder引用对象。当Client进程通过句柄值来访问某个Service服务时，Binder驱动程序可以通过该句柄值找到对应的Binder引用对象，然后根据该Binder引用对象的成员变量node找到对应的Binder实体对象，最后通过该Binder实体对象找到要访问的Service。proc执行该Binder引用对象的宿主进程，rb_node_desc和rb_node_node是binder_proc中红黑树refs_by_desc和refs_by_node的节点。

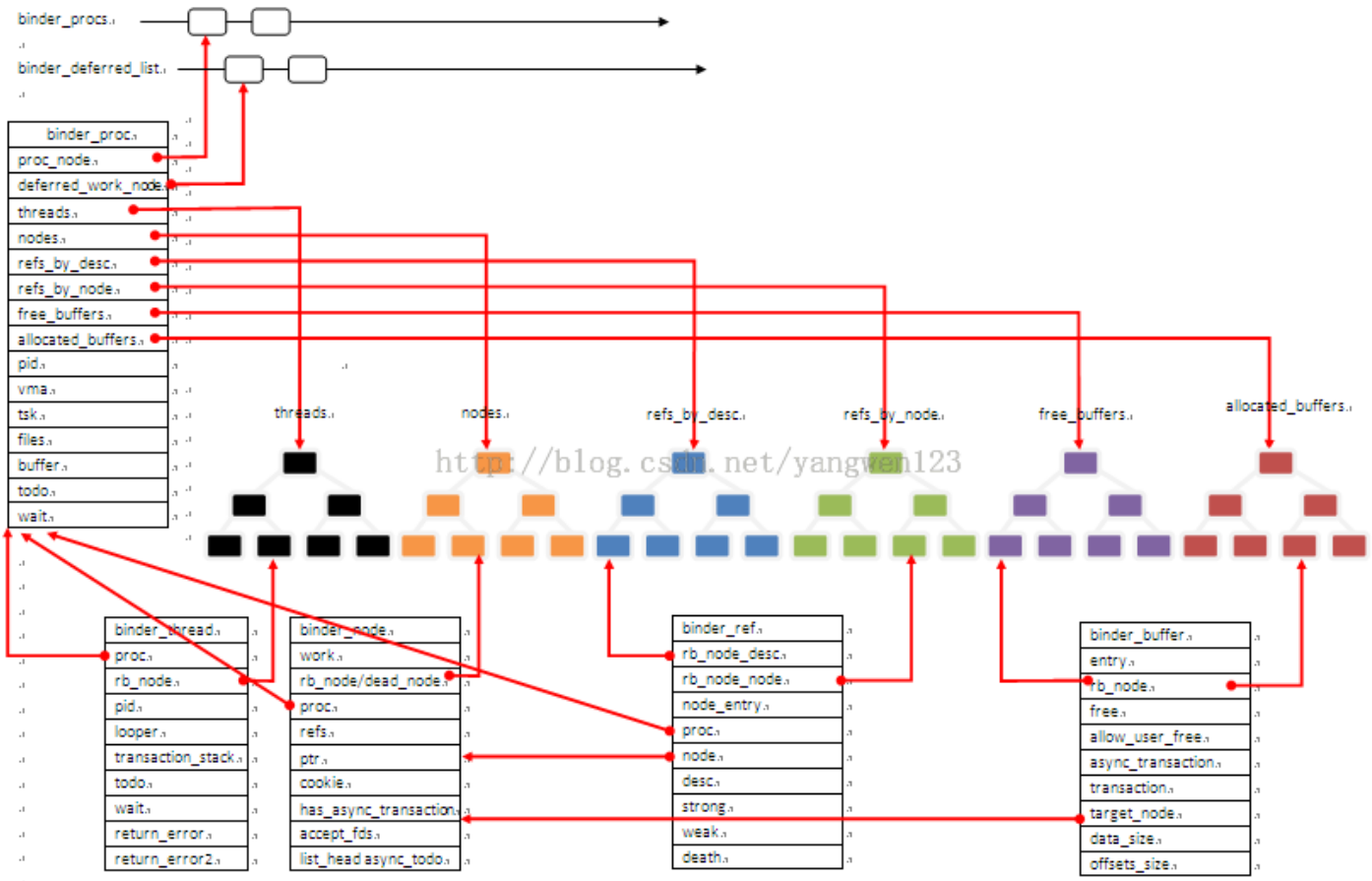
## 内核缓存区描述符——binder_buffer

```
[cpp]

struct binder_buffer {
    //binder_proc成员变量buffers链表中的节点
    struct list_head entry;
    //根据该内核缓存区是否空闲来选择挂载到binder_proc的free_buffers和allocated_buffers红黑树上
    struct rb_node rb_node;
    //标识该缓冲区是否空闲
    unsigned free:1;
    //标识内核缓冲区是否允许释放
    unsigned allow_user_free:1;
    //标识该内核缓冲区是否关联异步事务
    unsigned async_transaction:1;
    //调试id
    unsigned debug_id:29;
    //描述该内核缓冲区所属的事务
    struct binder_transaction *transaction;
    //描述该内核缓冲区所属的Binder实体对象
    struct binder_node *target_node;
    //描述数据缓冲区的大小
    size_t data_size;
    //Binder对象偏移数组的大小
    size_t offsets_size;
```

```
    //大小可变的数据缓冲区，用来保存通信数据
    uint8_t data[0];
};
```

binder_buffer用来描述一个内核缓冲区，用来在进程间传输数据。每个使用Binder进程通信机制的进程在Binder驱动中都有一个内核缓冲区列表，用来保存Binder驱动程序为它分配的内核缓冲区，成员变量entry就是这个内核缓冲区列表中的节点，binder_proc使用两个红黑树free_buffers和allocated_buffers来保存正在使用的内核缓冲区和空闲内核缓冲区，rb_node就是这两棵红黑树的节点。成员变量transaction和target_node用来描述一个内核缓冲区正在交给那个事务及那个Binder实体对象使用，data指向一块大小可变的数据缓冲区，真正用来保存通信数据，数据缓冲区保存普通数据和Binder对象，由于数据缓冲区中的普通数据和Binder对象是混合保存在一起的，没有固定顺序，在数据缓冲区后面用一个偏移数组来记录每一个Binder对象在数据缓冲区中的位置，该偏移数组的大小保存在成员变量offsets_size中，而缓冲区的大小保存在成员变量data_size中。




# 进程通信事务——binder_transaction

```
[cpp]
struct binder_transaction {
    //调试id
    int debug_id;
    //设置事务类型
    struct binder_work work;
    //描述发起事务的线程
    struct binder_thread *from;
    //该事务所依赖的事务
    struct binder_transaction *from_parent;
    //目标线程的下一个事务
    struct binder_transaction *to_parent;
```

```
    //负责处理该事务的进程
    struct binder_proc *to_proc;
    //负责处理该事务的线程
    struct binder_thread *to_thread;
    //区分同步或异步事务
    unsigned need_reply:1;
    //指向为该事务分配的一块内核缓冲区
    struct binder_buffer *buffer;
    //进程通信代码
    unsigned int    code;
    //标志位，描述进程间通信行为的特征
    unsigned int    flags;
    //发起事务的线程优先级
    long    priority;
    //保存处理该事务的线程原有优先级
    long    saved_priority;
    //发起事务的euid
    uid_t   sender_euid;
};
```

binder_transaction用来描述进程间通信过程。need_reply用来区分一个事务是同步还是异步事务，from指向发起事务的线程，to_proc指向负责处理该事务的进程，to_thread指向处理该事务的线程；当Binder驱动为目标进程或线程创建事务时，通过work设置事务类型，然后添加到进程或者线程的todo队列中等待处理；一个线程在处理一个事务时，Binder驱动程序需要修改它的优先级，在修改前，首先将线程的原有优先级保存在saved_priority中，以便线程处理完该事务后可以恢复原来的优先级。成员变量buffer指向Binder驱动程序为该事务分配的一块内核缓冲区，该缓冲区保存了通信数据。

## 进程通信数据——binder_transaction_data

```
[cpp]

struct binder_transaction_data {
    //用来描述目标Binder实体对象或者目标Binder引用对象
    union {
        size_t  handle; /* target descriptor of command transaction */
        void    *ptr;   /* target descriptor of return transaction */
    } target;
    //目标Binder本地Binder对象BBinder的地址
    void        *cookie;    /* target object cookie */              载
    //通信代码
    unsigned int    code;       /* transaction command */
    //通信标志位
    unsigned int    flags;
    //源进程的pid
    pid_t       sender_pid;
    //源进程的euid
    uid_t       sender_euid;
    //数据缓存区大小
    size_t      data_size;
    //记录Binder实体对象偏移的数组大小
    size_t      offsets_size;
    //数据缓冲区
    union {
        struct {
```

```
            //真正保存通信数据的缓冲区
            const void  *buffer;
            //记录Binder对象偏移的数组
            const void  *offsets;
        } ptr;
        uint8_t buf[8];
    } data;
};
```

binder_transaction_data用来描述进程间通信过程中传输的数据，target用来描述目标Binder实体对象或者目标Binder引用对象，如果描述的是目标Binder实体对象，那么成员变量ptr就指向该Binder本地对象BBinder内部的弱引用计数的地址，如果描述的是目标Binder引用对象，那么成员变量handle就指向该Binder引用对象的句柄值。cookie指向Binder本地对象BBinder的地址，成员变量data_size用来描述一个通信数据缓冲区，offset_size用来描述一个偏移数组的大小，data指向一个通信数据缓冲区，当通信数据较小时，就直接使用buf来传输数据，当通信数据较大时，就需要使用一块动态分配的缓冲区来传输数据。ptr指向这块动态缓冲区，buffer指向保存通信数据的数据缓冲区，当数据缓冲区中包含有Binder对象时，offsets就用来保存每一个Binder对象在数据缓冲区的偏移。