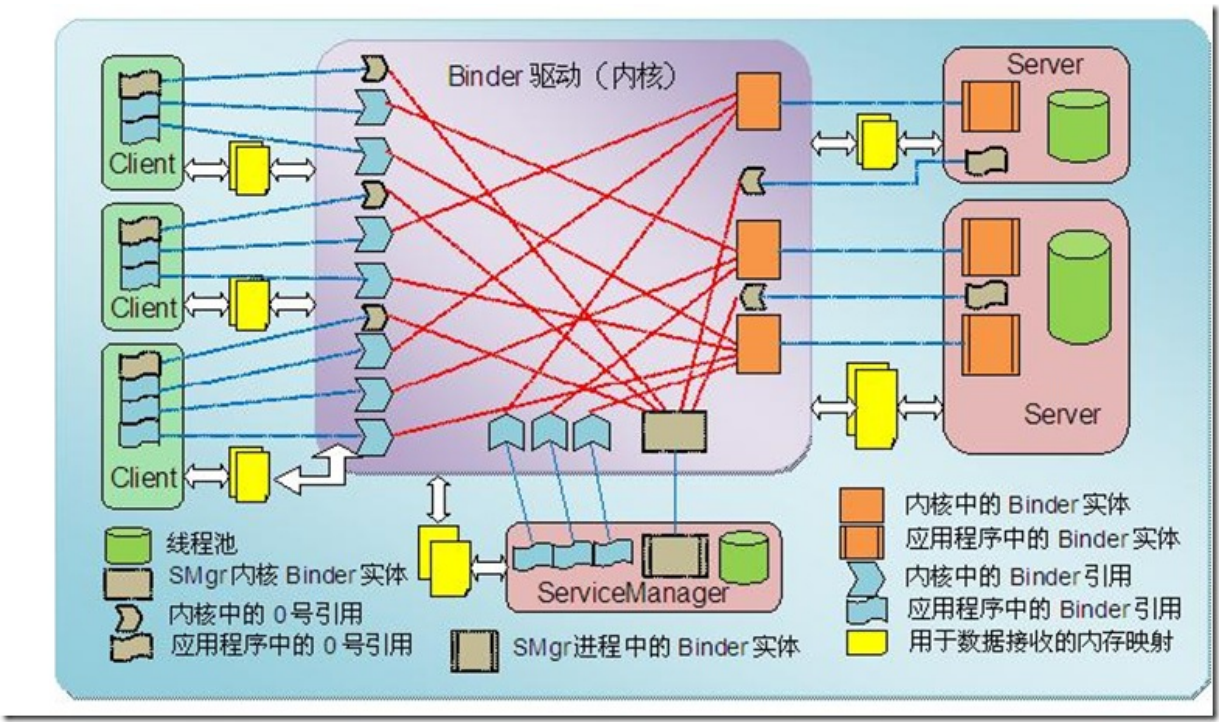
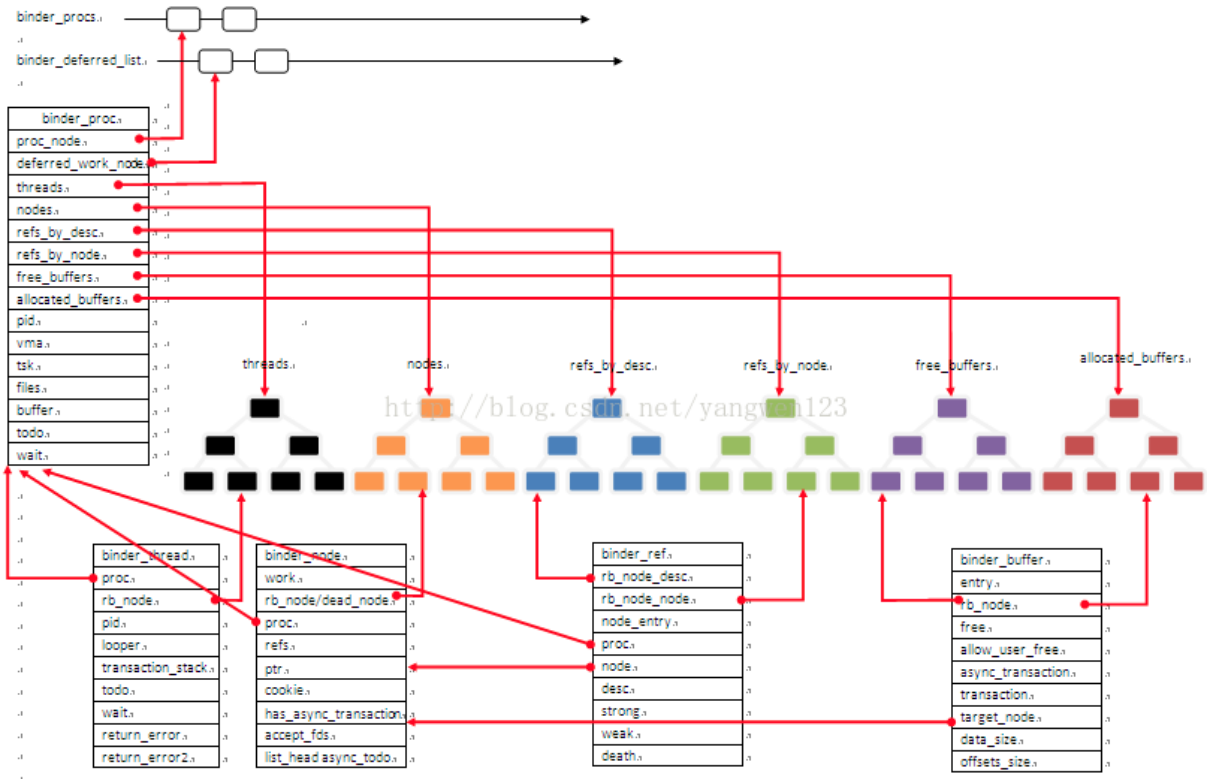
Binder 通信数据结构



图一



图二

1. struct binder_work

generated by haroopad

```
struct binder_work {
  struct list_head entry;
  enum {
          BINDER_WORK_TRANSACTION = 1,
          BINDER_WORK_TRANSACTION_COMPLETE,
          BINDER_WORK_NODE,
          BINDER_WORK_DEAD_BINDER,
          BINDER_WORK_DEAD_BINDER_AND_CLEAR,
          BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
  } type;
};
```

成员变量type用来描述工作项的类型，根据其取值，dr就可以判断一个binder_work结构体应该嵌入到什么类型的宿主结构中。

2. struct binder_node
   1）该结构体用来描述一个binder实体对象，每一个service组件在Binder驱动中都对应一个Binder实体对象，用来描述它在内核中的状态。Binder驱动通过强弱引用计数来维护binder实体对象的生命期。
   2）宿主进程通过使用一个红黑树来维护其内部所有的Binder实体对象。
   3）internal_strong_refs和local_strong_refs都是用来描述Binder实体对象的强引用计数
   4）成员变量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。通常，Binder驱动都是将一个事务保存在一个线程的一个todo队列中，表示由该线程来处理该事务,同时，每一个事务都关联者一个Binder实体对象，表示该事务的目标处理对象，即要求与该Binder实体对象对应的Service组件在指定的线程中处理该事务。
   5） 一个service组件将自己注册到Binder驱动时，可以指定这个最小线程优先级，而Binder驱动会将这个优先级保存到相应的Binder实体对象的成员变量min_priority.
3. struct binder_refs_death
   该结构体用来描述一个service组件的死亡接收通知，当service组件意外死亡时，Client进程能够在它所引用的service在死亡时获得通知，以便可以进行相应的处理。
   1) 成员变量cookie用来保存负责接收死亡通知的的对象的地址，也就是说有一个client c向驱动注册了接收死亡通知，当c关心的service意外死亡时，驱动会创建一个struct binder_ref_death对象并将其封装成一个工作项，设置好该结构体相关成员，然后将该工作项添加到Client的todo队列中等待处理。
4. struct binder_ref
   成员变量desc是一个句柄值，或者称为描述符，用来描述一个Binder引用对象。在Client进程的用户空间中，一个Binder引用对象是使用一个句柄值来描述的。这一点可以观察最开始的图一。可以看到一个应用程序Client中有多个binder实体对象的引用，通过这些句柄值就可以很方便的找到需要的通信的的是哪一个Binder实体对象。成员变量death指向一个service组件的死亡接收通知，当Client 进程向驱动注册一个它所引用的组件的死亡接收通知时，Binder驱动程序就会创建一个binder_ref_death结构体，然后保存在death中。
5. struct binder_buffer
   成员变量transaction用来描述一个内核缓冲区正在交给哪一个事务使用，target_node用来描述一个内核缓冲区正在交给哪一个Binder实体对象使用。Binder驱动程序将事务数据保存在一个内核缓冲区中，每一个事务都关联有一个目标Binder实体对象。而目标Binder实体对象再将该内核缓冲区的内容交给相应的Service组件处理。

6. struct binder_pro
   进程打开了设备文件/dev/binder后，还必须调用mmap函数将它映射到进程的地址空间来，实际上是请求Binder驱动为其分配一块内核缓冲区，以便可以用来在进程间传输数据。该内核缓冲区有两个地址，一个是内核空间空间地址,保存在buffer成员变量中，一个是用户空间地址，保存在vma成员变量中，二者相差一个固定值。注意，这两个地址都是虚地址，它们对应的物理页面保存在成员变量page中，通过虚拟地址映射关系CPU运行时即可拿到物理地址。
   1) 每一个使用了Binder进程间通信机制的进程都一个Binder线程池，用来处理进程间通讯请求，这个Binder线程池是由Binder驱动进行维护的。进程可以调用函数iotrl将一个线程注册到Binder驱动中，同事，当进程没有足够的空闲线程来处理进程间通讯请求时，Binder驱动也可以主动要求进程注册更多的线程到Binder线程池中。
   2) Binder线程池中的空闲Binder线程会睡眠在由wait等待队列(猜测：等待队列是通过信号量机制或者管程来实现)，当它们的宿主进程增加新的工作项之后，Binder驱动程序就会唤醒这些线程，区处理新的工作项。
   3)Binder驱动程序将所有的延迟执行的工作项保存在一个hash列表中，如果一个进程有延迟执行的工作项，那么成员变量deferred_work_node就刚好是该hash列表的一个节点，同时使用deferred_work来描述该延迟工作项的具体类型。
7. binder_thread
   1)当Binder驱动决定将一个事务交给一个Binder线程处理时，就会将该事务封装成一个binder_transaction结构体对象，并且将其添加到由线程结构体binder_thread的成员变量transaction_stack描述的一个事务堆栈。
   2)当一个Binder线程在处理一个事务T1时，其依赖其他Binder线程来处理另外一个事务T2,那么处理T1的线程就会睡眠在由成员变量wait描述的等待队列。
8. struct binder_transaction
   1)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，里面保存了进程间通信数据，而成员变量code和flags则是直接从进程间通讯数据中拷贝过来。
9. struct binder_write_read
   1) 成员变量write_buffer指向一个用户空间的缓冲区的地址，里面保存的内容即为要传输到Binder驱动的数据，read_buffer也指向一个用户空间地址，里面保存的内容为驱动返回给用户空间的进程间通讯结果数据。
   2)缓冲区write_buffer和read_buffer数据格式：

缓冲区write_buffer和read_buffer的数据格式如图5–2所示。

Code-1	Content of Code-1	Code-2	Content of Code-2	.....	Code-*N*	Conent of Code-*N*

图5–2 缓冲区write_buffer和read_buffer的数据格式

   数组每一个元素都由一个通信协议码和通信数据组成。
10. struct binder_transaction_data
    成员变量target是一个联合体，用来描述一个目标Binder实体对象或者目标Binder引用对象，如果描述的是一个目标Binder实体对象，则target的成员ptr就指向该Binder实体对象对应的Service 组件内部的一个弱引用计数对象的地址；如果描述的是一个目标Binder引用对象，其成员变量handle指向该Binder引用对象的句柄值。
    1)当数据缓冲区中包含有Binder对象时，那么紧跟在这个数据缓冲区后面就会有一个偏移数组offset,用于描述数据缓冲区中每一个Binder对象的位置。而数据缓冲区的每一个

Binder对象都使用一个flat_binder_object结构体来描述。如下图所示：

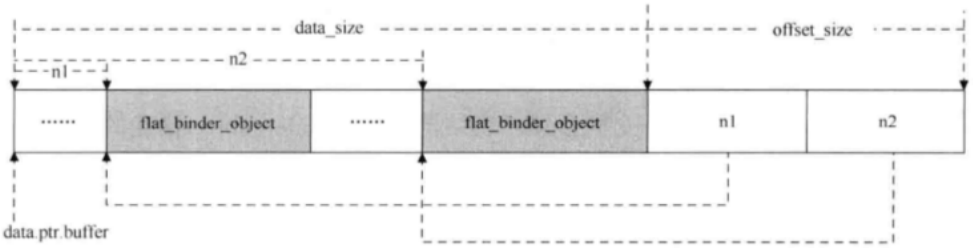


图5-3 Binder进程间通信数据缓冲区的内存布局

11. struct flat_binder_object
成员变量binder和handle组成了一个联合体，当结构体flat_binder_object描述的是一个BInder实体对象时，就使用成员变量binder来指向与该Binder实体对象对应的Service 组件内部的一个弱引用计数对象的地址，并且使用成员变量cookie来指向该service组件的地址；当其描述的是一个Binder引用对象时，使用成员变量handle来描述该Binder引用对象的句柄值。