在 Linux 系统中，是以进程为单位分配和管理资源的。出于保护机制，一个进程不能直接访问另一个进程的资源，也就是说，进程之间互相封闭。但是，在一个复杂的应用系统中，通常会使用多个相关的进程来共同完成一项任务，因此要求进程之间必须能够互相通信，从而共享资源和信息。所以，操作系统内核必须提供进程间的通信机制（IPC）。在 Linux 中，进程间的通信机制有很多种，例如可以采用命名管道（named pipe）、消息队列（message queue）、信号（signal）、共享内存（share memory）、socket 等方式，它们都可以实现进程间的通信。但是，在 Android 终端上的应用软件的通信几乎看不到这些 IPC 通信方式，取而代之的是 Binder 方式。Android 同时为 Java 环境和 C/C++环境提供了 Binder 机制。本章主要介绍 C/C++环境下的 Binder 机制，主要包括 Binder 驱动的实现、运作原理、IPC 机制的实现、接口等，所以本章可能会涉及部分系统运行库的源代码，我们将详细讲解。

3.1　Binder 概述

应用程序虽然是以独立的进程来运行的，但相互之间还是需要通信，比如，在多进程的环境下，应用程序和后台服务通常会运行在不同的进程中，有着独立的地址空间，但是因为需要相互协作，彼此间又必须进行通信和数据共享，这就需要进程通信来完成。在 Linux 系统中，进程间通信的方式有 socket、named pipe、message queue、signal、share memory 等；Java 系统中的进程间通信方式也有 socket、named pipe 等，所以 Android 可以选择的进程间通信的方式也很多，但是它主要包括以下几种方式：

- 标准 Linux Kernel IPC 接口
- 标准 D-BUS 接口
- Binder 接口

3.1.1　为什么选择 Binder

在上面这些可供选择的方式中，Android 使用得最多也最被认可的还是 Binder 机制。为什么会选择 Binder 来作为进程之间的通信机制呢？因为 Binder 更加简洁和快速，消耗的内存资源更小吗？不错，这些也正是 Binder 的优点。当然，也还有很多其他原因，比如传统的进程间通信可能会增加进程的开销，而且有进程过载和安全漏洞等方面的风险，Binder 正好能解决和避免这些问题。Binder 主要能提供以下一些功能：

- 用驱动程序来推进进程间的通信。
- 通过共享内存来提高性能。
- 为进程请求分配每个进程的线程池。
- 针对系统中的对象引入了引用计数和跨进程的对象引用映射。
- 进程间同步调用。

3.1.2 初识 Binder

Binder 是通过 Linux 的 Binder Driver 来实现的，Binder 操作类似于线程迁移（thread migration），两个进程间通信看起来就像是一个进程进入另一个进程去执行代码，然后带着执行的结果返回。Binder 的用户空间为每一个进程维护着一个可用的线程池，线程池用于处理到来的 IPC 以及执行进程的本地消息，Binder 通信是同步的而不是异步的。同时，Binder 机制是基于 OpenBinder[㊀]来实现的，是一个 OpenBinder 的 Linux 实现，Android 系统的运行都将依赖 Binder 驱动。

Binder 通信也是基于 Service 与 Client 的，所有需要 IBinder 通信的进程都必须创建一个 IBinder 接口。系统中有一个名为 Service Manager 的守护进程管理着系统中的各个服务，它负责监听是否有其他程序向其发送请求，如果有请求就响应，如果没有则继续监听等待。每个服务都要在 Service Manager 中注册，而请求服务的客户端则向 Service Manager 请求服务。在 Android 虚拟机启动之前，系统会先启动 Service Manager 进程，Service Manager 就会打开 Binder 驱动，并通知 Binder Kernel 驱动程序，这个进程将作为 System Service Manager，然后该进程将进入一个循环，等待处理来自其他进程的数据。因此，我们也可以将 Binder 的实现大致分为：Binder 驱动、Service Manager、Service、Client 这几个部分，下面将分别对这几个部分进行详细分析。

3.2 Binder 驱动的原理和实现

通过上一节的介绍，大家应该对 Binder 有了基本的认识了。任何上层应用程序接口和用户操作都需要底层硬件设备驱动的支持，并为其提供各种操作接口。本节首先从 Binder 的驱动实现入手，分析其原理和它提供给用户层使用的接口。

3.2.1 Binder 驱动的原理

为了完成进程间通信，Binder 采用了 AIDL（Android Interface Definition Language）来描述进程间的接口。在实际的实现中，Binder 是作为一个特殊的字符型设备而存在的，设备节点为 /dev/binder，其实现遵循 Linux 设备驱动模型，实现代码主要涉及以下文件：

- kernel/drivers/staging/binder.h
- kernel/drivers/staging/binder.c

在其驱动的实现过程中，主要通过 binder_ioctl 函数与用户空间的进程交换数据。BINDER_WRITE_READ 用来读写数据，数据包中有一个 cmd 域用于区分不同的请求。binder_thread_write 函数用于发送请求或返回结果，而 binder_thread_read 函数则用于读取结果。

㊀ 关于 OpenBinder 的更多信息，可以参考 http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html。

在 binder_thread_write 函数中调用 binder_transaction 函数来转发请求并返回结果。当收到请求时，binder_transaction 函数会通过对象的 handle 找到对象所在的进程，如果 handle 为空，就认为对象是 context_mgr，把请求发给 context_mgr 所在的进程。请求中所有的 Binder 对象全部放到一个 RB 树中，最后把请求放到目标进程的队列中，等待目标进程读取。数据的解析工作放在 binder_parse()中实现；关于如何生成 context_mgr，内核中提供了 BINDER_SET_CONTEXT_MGR 命令来完成此项功能。下面我们就来看看 Binder 驱动究竟是如何实现的。

3.2.2　Binder 驱动的实现

上面我们已经对 Binder 驱动的原理进行了分析，在开始分析驱动的实现之前，我们还是通过一个例子来说明 Binder 在实际应用中应该如何运用，以及它能帮我们解决什么样的问题。这样会更容易帮助大家理解 Binder 驱动的实现。比如，A 进程如果要使用 B 进程的服务，B 进程首先要注册此服务，A 进程通过 Binder 获取该服务的 hanlde，通过这个 handle，A 进程就可以使用该服务了。此外，你可以把 handle 理解成地址。A 进程使用 B 进程的服务还意味着二者遵循相同的协议，这个协议反映在代码上就是二者要实现 IBinder 接口。

1.“对象”与“引用”

Binder 不仅是 Android 系统中的一个完善的 IPC 机制，它也可以被当作 Android 系统的一种 RPC（远程过程调用）机制，因为 Binder 的功能就是在本地“执行”其他进程的功能。因此，进程在通过 Binder 获取将要调用的进程服务时，可以是一个本地对象，也可以是一个远程服务的“引用”。这一点可能比较难以理解，稍候就会为大家分析，这里就先记住 Binder 不仅可以与本地进程通信，还可以与远程进程通信；这里的本地进程就是我们所说的本地对象，而远程进程则是我们所说的远程服务的一个“引用”㊀。

Binder 的实质就是要把对象从一个进程映射到另一个进程中，而不管这个对象是本地的还是远程的。如果是本地对象，更好理解；如果是远程对象，就按照我们上面所讲的来理解，即将远程对象的“引用”从一个进程映射到另一个进程中，于是当使用这个远程对象时，实际上就是使用远程对象在本地的一个“引用”，类似于把这个远程对象当作一个本地对象在使用。这也就是 Binder 与其他 IPC 机制不同的地方。

这个本地“对象”与远程对象的“引用”有什么不同呢？本地“对象”表示本地进程的地址空间的一个地址，而远程对象的“引用”则是一个抽象的 32 位句柄。它们之间是互斥的：所有的进程本地对象都是本地进程的一个地址（address、ptr、binder），所有的远程进程的对象的“引用”都是一个句柄。对于发送者进程来说，不管是“对象”还是“引用”，它都会认为被发送的 Binder 对象是一个远程对象的句柄（即远程对象的“引用”）。但是，当 Binder 对象的数据被发送到远端接收进程时，远端接收进程则会认为该 Binder 对象是一个本地对象地址（即本地对象）。正如我们之前说的，当 Binder 对象被接收进程接收后，不管该 Binder 对象是

㊀ “引用”这个词并不是官方所描述的，而是笔者为了方便大家理解，将其称为引用，或许你有更好的描述。

本地的还是远程的，它都会被当作一个本地进程来处理。因此，从第三方的角度来说，尽管名称不同，对于一次完整的 Binder 调用，都将指向同一个对象，Binder 驱动则负责两种不同名称的对象的正确映射，这样才能把数据发送给正确的进程进行通信。这个映射关系也是进程间引用对象的基础，对一个对象的引用，在远程是句柄，在本地则是地址（即本地对象的地址）。

下面我们先介绍分析该机制中所使用的数据结构体，然后再对整个流程进行分析。

2．binder_work

首先来看一个最简单也是最基础的结构体 binder_work，其定义如代码清单 3-1 所示。

代码清单 3-1　binder_work 定义

```
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;
};
```

其中 entry 被定义为 list_head，用来实现一个双向链表，存储所有 binder_work 的队列；此外，还包含一个 enum 类型的 type，表示 binder_work 的类型，后文会对这些类型进行详细分析，大家就会觉得它更像是一个用来表示状态的 enum。

3．Binder 的类型

Binder 的类型是使用定义在 binder.h 头文件中的一个 enum 来表示的，定义如代码清单 3-2 所示。

代码清单 3-2　Binder 类型

```
#define B_PACK_CHARS(c1, c2, c3, c4) \
    ((((c1)<<24)) | (((c2)<<16)) | (((c3)<<8)) | (c4))
#define B_TYPE_LARGE 0x85
enum {
    BINDER_TYPE_BINDER  = B_PACK_CHARS('s', 'b', '*', B_TYPE_LARGE),
    BINDER_TYPE_WEAK_BINDER  = B_PACK_CHARS('w', 'b', '*', B_TYPE_LARGE),
    BINDER_TYPE_HANDLE  = B_PACK_CHARS('s', 'h', '*', B_TYPE_LARGE),
    BINDER_TYPE_WEAK_HANDLE  = B_PACK_CHARS('w', 'h', '*', B_TYPE_LARGE),
    BINDER_TYPE_FD  = B_PACK_CHARS('f', 'd', '*', B_TYPE_LARGE),
};
```

从上面的代码可以看出，Binder 被分成了 5 个不同的类型，但是仔细一看却是 3 个不同的大类，它们分别是：本地对象（BINDER_TYPE_BINDER、BINDER_TYPE_WEAK_BINDER）、

远程对象的“引用”（BINDER_TYPE_HANDLE、BINDER_TYPE_WEAK_HANDLE）和文件（BINDER_TYPE_FD）。前面两种都是我们刚刚分析过的，下面主要分析最后一种——文件（BINDER_TYPE_FD）。如果传递的是 BINDER_TYPE_FD 类型，其实还是会将文件映射到句柄上，根据此 fd 找到对应的文件，然后在目标进程中分配一个 fd，最后把这个 fd 赋值给返回的句柄。

4．Binder 对象

我们把进程之间传递的数据称之为 Binder 对象（Binder Object），它在对应源码中使用 flat_binder_object 结构体（位于 binder.h 文件中）来表示，其定义如代码清单 3-3 所示。

代码清单 3-3　flat_binder_object 定义

```
struct flat_binder_object {
    unsigned long    type;
    unsigned long    flags;
    union {
        void     *binder;
        signed long  handle;
    };
    void         *cookie;
};
```

该结构体中的 type 字段描述的是 Binder 的类型，传输的数据是一个复用数据联合体。对于 Binder 类型，数据就是一个 Binder 本地对象；HANDLE 类型，就是一个远程的 handle 句柄。本地 Binder 对象和远程 handle 句柄比较难以理解，这里我们再次举例说明：假如 A 有个对象 O，对于 A 来说，O 就是一个本地的 Binder 对象；如果 B 想访问 A 的 O 对象，对于 B 来说，O 就是一个 handle。因此，从根本上来说，handle 和 Binder 都指向 O。如果是本地对象，Binder 还可以带有额外的数据，这些数据将被保存到 cookie 字段中。flags 字段表示传输方式，比如同步和异步等，其值同样使用一个 enum 来表示，定义如代码清单 3-4 所示。

代码清单 3-4　transaction_flags 定义

```
enum transaction_flags {
    TF_ONE_WAY   = 0x01,
    TF_ROOT_OBJECT    = 0x04,
    TF_STATUS_CODE    = 0x08,
    TF_ACCEPT_FDS= 0x10,
};
```

其中 TF_ONE_WAY 表示单向传递，是异步的，不需要返回；TF_ROOT_OBJECT 表示内容是一个组建的根对象，对应类型为本地对象 Binder；TF_STATUS_CODE 表示内容是一个 32 位的状态码，将对应类型为远程对象的“引用”（即句柄 handle）；TF_ACCEPT_FDS 表示可以接收一个文件描述符，对应的类型为文件（BINDER_TYPE_FD），即 handle 中存储的为文件描述符。

5. binder_transaction_data

其实我们并没有从 flat_binder_object 结构体中看到 Binder 对象所传递的实际内容，因为 Binder 对象所传递的实际内容是通过另外一个结构体 binder_transaction_data 来表示的，其定义如代码清单 3-5 所示。

代码清单 3-5 binder_transaction_data 定义

```
struct binder_transaction_data {
    union {
        size_t   handle;
        void *ptr;
    } target;
    void     *cookie;
    unsigned int code;
    unsigned int flags;
    pid_t        sender_pid;
    uid_t        sender_euid;
    size_t       data_size;
    size_t       offsets_size;
    union {
        struct {
            const void   *buffer;
            const void   *offsets;
        } ptr;
        uint8_t  buf[8];
    } data;
};
```

该结构体是理解 Binder 驱动实现的关键，下面将详细地对一个重要的数据进行分析。其中 target 字段又是一个复合联合体对象，target 字段中的 handle 是要处理此事件的目标对象的句柄，根据此 handle，Binder 驱动可以找到应该由哪个进程处理此事件，并且把此事件的任务分发给一个线程，而那个线程也正在执行 ioctl 的 BINDER_WRITE_READ 操作，即正在等待一个请求（见 3.1 节和 3.2.1 节），处理方法将稍候分析。target 的 ptr 字段与 handle 对应，对于请求方，使用 handle 来指出远程对象；对于响应方，使用 ptr 来寻址，以便找到需要处理此事件的对象。所以 handle 和 ptr 是一个事物的两种表达（正如前面所说的本地对象和远程对象的“引用”），handle 和 ptr 之间的翻译（解析）关系正是 Binder 驱动需要维护的（在 binder_transaction 函数中，稍候分析）。

另外，该结构体中的 cookie 字段表示 target 对象所附加的额外数据；code 是一个命令，它描述了请求 Binder 对象执行的操作；flags 字段描述了传输的方式与 flat_binder_object 中的 flags 字段对应；sender_pid 和 sender_euid 表示该进程的 pid 和 uid；data_size 表示数据的大小字节数；offsets_size 表示数据的偏移量字节数；最后一个 union 数据 data 表示真正的数据，其中 ptr 表示与 target->ptr 对应的对象的数据，buf 表示与 handle 对象对应的数据，data 中的 ptr 中的 buffer

表示实际的数据，而 offsets 则表示其偏移量。

6．binder_write_read

上面我们说过，当 Binder 驱动找到处理此事件的进程之后，Binder 驱动就会把需要处理的事件的任务放在读缓冲（binder_write_read）里，返回给这个服务线程，该服务线程则会执行指定命令的操作；处理请求的线程把数据交给合适的对象来执行预定操作，然后把返回结果同样用 binder_transaction_data 结构封装，以写命令的方式传回给 Binder 驱动，并将此数据放在一个读缓冲（binder_write_read）里，返回给正在等待结果的原进程（线程），这样就完成了一次通信。这里所说的 BINDER_WRITE_READ 命令和读写缓冲区分别什么呢？从下面这个宏中可以看到结果：

```
#define BINDER_WRITE_READ _IOWR('b', 1, struct binder_write_read)
```

该命令是 ioctl 函数中的一个操作，从上面的宏中可以看出，其所对应的参数是 binder_write_read 结构体，该结构体正好就是我们所说的读写缓冲区，将用它来存储发送的任务信息和接收返回的结果信息，其定义如代码清单 3-6 所示。

代码清单 3-6　binder_write_read 定义

```
struct binder_write_read {
    signed long    write_size;
    signed long    write_consumed;
    unsigned long  write_buffer;
    signed long    read_size;
    signed long    read_consumed;
    unsigned long  read_buffer;
};
```

上面的代码中分别指定了读缓冲区和写缓冲区，对于写操作，write_buffer 包含了一系列请求线程执行的 Binder 命令；对于读（返回结果）操作，read_buffer 包含了一系列线程执行后填充的返回值。这些读和写的操作命令都需要遵循一定的 Binder 协议。write_size 和 read_size 分别表示写入和读取的数据的大小；write_consumed 和 read_consumed 则分别表示被消耗的写数据和读数据的大小。下面我们看看 BINDER_WRITE_READ 命令具体有哪些，如代码清单 3-7 所示。

代码清单 3-7　BINDER_WRITE_READ 命令协议

```
//BINDER_WRITE_READ 的写操作命令协议
enum BinderDriverCommandProtocol {
    BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
    BC_REPLY = _IOW('c', 1, struct binder_transaction_data),

    BC_ACQUIRE_RESULT = _IOW('c', 2, int),
    BC_FREE_BUFFER = _IOW('c', 3, int),
    BC_INCREFS = _IOW('c', 4, int),
    BC_ACQUIRE = _IOW('c', 5, int),
```

```
    BC_RELEASE = _IOW('c', 6, int),
    BC_DECREFS = _IOW('c', 7, int),
    BC_INCREFS_DONE = _IOW('c', 8, struct binder_ptr_cookie),
    BC_ACQUIRE_DONE = _IOW('c', 9, struct binder_ptr_cookie),
    BC_ATTEMPT_ACQUIRE = _IOW('c', 10, struct binder_pri_desc),
    BC_REGISTER_LOOPER = _IO('c', 11),
    BC_ENTER_LOOPER = _IO('c', 12),
    BC_EXIT_LOOPER = _IO('c', 13),
    BC_REQUEST_DEATH_NOTIFICATION = _IOW('c', 14, struct binder_ptr_cookie),
    BC_CLEAR_DEATH_NOTIFICATION = _IOW('c', 15, struct binder_ptr_cookie),
    BC_DEAD_BINDER_DONE = _IOW('c', 16, void *),
};
//BINDER_WRITE_READ 的读操作命令协议
enum BinderDriverReturnProtocol {
    BR_ERROR = _IOR('r', 0, int),
    BR_OK = _IO('r', 1),
    BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
    BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
    BR_ACQUIRE_RESULT = _IOR('r', 4, int),
    BR_DEAD_REPLY = _IO('r', 5),
    BR_TRANSACTION_COMPLETE = _IO('r', 6),
    BR_INCREFS = _IOR('r', 7, struct binder_ptr_cookie),
    BR_ACQUIRE = _IOR('r', 8, struct binder_ptr_cookie),
    BR_RELEASE = _IOR('r', 9, struct binder_ptr_cookie),
    BR_DECREFS = _IOR('r', 10, struct binder_ptr_cookie),
    BR_ATTEMPT_ACQUIRE = _IOR('r', 11, struct binder_pri_ptr_cookie),
    BR_NOOP = _IO('r', 12),
    BR_SPAWN_LOOPER = _IO('r', 13),
    BR_FINISHED = _IO('r', 14),
    BR_DEAD_BINDER = _IOR('r', 15, void *),
    BR_CLEAR_DEATH_NOTIFICATION_DONE = _IOR('r', 16, void *),
    BR_FAILED_REPLY = _IO('r', 17),
};
```

其中最重要的就是 BC_TRANSACTION 和 BC_REPLY 命令，它们被作为发送操作的命令，其数据参数都是 binder_transaction_data 结构体；前者用于翻译和解析将要被处理的事件数据，后者则是事件处理完成之后对返回“结果数据”的操作命令。我们稍后在分析 binder_ioctl 的实现时，会对这些命令的作用进行详细的分析。

7．binder_proc

binder_proc 结构体用于保存调用 Binder 的各个进程或线程的信息，比如线程 ID、进程 ID、Binder 状态信息等，定义如代码清单 3-8 所示。

代码清单 3-8　binder_proc 定义

```
struct binder_proc {
    //实现双向链表
```

```
    struct hlist_node proc_node;
    //线程队列、双向链表、所有的线程信息
    struct rb_root threads;
    struct rb_root nodes;
    struct rb_root refs_by_desc;
    struct rb_root refs_by_node;
    //进程 ID
    int pid;
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    struct files_struct *files;
    struct hlist_node deferred_work_node;
    int deferred_work;
    void *buffer;
    ptrdiff_t user_buffer_offset;

    struct list_head buffers;
    struct rb_root free_buffers;
    struct rb_root allocated_buffers;
    size_t free_async_space;

    struct page **pages;
    size_t buffer_size;
    uint32_t buffer_free;
    struct list_head todo;
    //等待队列
    wait_queue_head_t wait;
    //Binder 状态
    struct binder_stats stats;
    struct list_head delivered_death;
    //最大线程
    int max_threads;
    int requested_threads;
    int requested_threads_started;
    int ready_threads;
    //默认优先级
    long default_priority;
};
```

对于其中几个重要的字段，我们都给出了注解，在具体的使用过程中，我们还会进一步分析。其中 proc_node 字段用于实现双向链表，threads 则用于储存所有的线程信息。紧接着我们就需要分析 Binder 节点和 Binder 线程的具体定义了。

8．binder_node

该结构体表示一个 Binder 节点，其定义如代码清单 3-9 所示。

代码清单 3-9　binder_node 定义

```
struct binder_node {
    int debug_id;
    struct binder_work work;
    union {
        struct rb_node rb_node;
        struct hlist_node dead_node;
    };
    struct binder_proc *proc;
    struct hlist_head refs;
    int internal_strong_refs;
    int local_weak_refs;
    int local_strong_refs;
    void __user *ptr;
    void __user *cookie;
    unsigned has_strong_ref : 1;
    unsigned pending_strong_ref : 1;
    unsigned has_weak_ref : 1;
    unsigned pending_weak_ref : 1;
    unsigned has_async_transaction : 1;
    unsigned accept_fds : 1;
    int min_priority : 8;
    struct list_head async_todo;
};
```

9．binder_thread

我们已经知道，binder_thread 结构体用于存储每一个单独的线程的信息，其定义如代码清单 3-10 所示。

代码清单 3-10　binder_thread 定义

```
struct binder_thread {
    struct binder_proc *proc;
    struct rb_node rb_node;
    int pid;
    int looper;
    struct binder_transaction *transaction_stack;
    struct list_head todo;
    uint32_t return_error;
    uint32_t return_error2;
    wait_queue_head_t wait;
    struct binder_stats stats;
};
```

其中，proc 字段表示当前线程属于哪一个 Binder 进程（binder_proc 指针）；rb_node 是一个红黑树节点；pid 表示线程的 pid；looper 表示线程的状态信息；transaction_stack 则定义了要

接收和发送的进程和线程信息，其结构体为 binder_transaction，稍候我们会进行详细的分析；todo 用于创建一个双向链表；return_error 和 return_error2 为返回的错误信息代码；wait 是一个等待队列头；stats 用于表示 Binder 状态信息，其定义如代码清单 3-11 所示。

代码清单 3-11　binder_stats 定义

```
struct binder_stats {
    int br[_IOC_NR(BR_FAILED_REPLY) + 1];
    int bc[_IOC_NR(BC_DEAD_BINDER_DONE) + 1];
    int obj_created[BINDER_STAT_COUNT];
    int obj_deleted[BINDER_STAT_COUNT];
};
```

其中，br 用来存储 BINDER_WRITE_READ 的写操作命令协议（Binder Driver Return Protocol）；bc 则对应存储着 BINDER_WRITE_READ 的写操作命令协议（Binder Driver Command Protocol）；obj_created 用于保存 BINDER_STAT_COUNT 的对象计数，当一个对象被创建时，就需要同时调用该成员来增加相应的对象计数，而 obj_deleted 则正好相反。

另外，looper 所表示的线程状态信息主要包括以下几种，如代码清单 3-12 所示。

代码清单 3-12　Binder looper 线程状态信息

```
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
};
```

其中主要包括了注册、进入、退出、销毁、等待、需要返回这几种状态信息。

10．binder_transaction

该结构体主要用来中转请求和返回结果，保存接收和要发送的进程信息，其定义如代码清单 3-13 所示。

代码清单 3-13　binder_transaction 定义

```
struct binder_transaction {
    int debug_id;//调试相关
    struct binder_work work;
    struct binder_thread *from;
    struct binder_transaction *from_parent;
    struct binder_proc *to_proc;
    struct binder_thread *to_thread;
    struct binder_transaction *to_parent;
    unsigned need_reply : 1;
```

```
    struct binder_buffer *buffer;
    unsigned int code;
    unsigned int flags;
    long priority;
    long saved_priority;
    uid_t    sender_euid;
};
```

其中 work 为一个 binder_work；from 和 to_thread 都是一个 binder_thread 对象，用于表示接收和要发送的进程信息，另外还包括了接收和发送进程信息的父节点 from_parent 和 to_thread；to_proc 是一个 binder_proc 类型的结构体。其中还包括 flags、need_reply、优先级（priority）等数据。这里我们需要重点说明的是最后一个 sender_euid，Linux 系统中每个进程都有 2 个 ID——用户 ID 和有效用户 ID，UID 一般表示进程的创建者（属于哪个用户创建），而 EUID 表示进程对于文件和资源的访问权限，因此这里的 sender_euid 表示要发送进程对文件和资源的操作权限。最后，该结构体中还包含类型类 inder_buffer 的一个 buffer，用来表示 binder 的缓冲区信息，其定义如代码清单 3-14 所示。

代码清单 3-14　binder_buffer 的定义

```
struct binder_buffer {
    struct list_head entry;
    struct rb_node rb_node;
    unsigned free : 1;
    unsigned allow_user_free : 1;
    unsigned async_transaction : 1;
    unsigned debug_id : 29;
    struct binder_transaction *transaction;
    struct binder_node *target_node;
    size_t data_size;
    size_t offsets_size;
    uint8_t data[0];
};
```

该结构体主要用来储存 Binder 的相关信息，entry 同样用于构建一个双向链表；rb_node 为一个红黑树节点；transaction 表示上面我们刚刚说到的 binder_transaction，用于中转请求和返回结果；target_node 是一个目标节点；data_size 表示数据的大小；offsets_size 是一个偏移量；data[0]用于存储实际数据。

11．binder_init

和其他驱动一样，我们先从初始化操作开始分析，大家可以在 binder.c 中找到该初始化函数，具体实现如代码清单 3-15 所示。

代码清单 3-15　binder_init 的实现

```
static int __init binder_init(void)
```

```
{
    int ret;
    //创建文件系统根节点
    binder_proc_dir_entry_root = proc_mkdir("binder", NULL);
    //创建 proc 节点
    if (binder_proc_dir_entry_root)
        binder_proc_dir_entry_proc = proc_mkdir("proc", binder_proc_dir_entry_root);
    //注册 Misc 设备
    ret = misc_register(&binder_miscdev);
    //创建各文件
    if (binder_proc_dir_entry_root) {
        create_proc_read_entry("state", S_IRUGO, binder_proc_dir_entry_root,
            binder_read_proc_state, NULL);
        create_proc_read_entry("stats", S_IRUGO, binder_proc_dir_entry_root,
            binder_read_proc_stats, NULL);
        create_proc_read_entry("transactions", S_IRUGO, binder_proc_dir_entry_ root,
            binder_read_proc_transactions, NULL);
        create_proc_read_entry("transaction_log", S_IRUGO, binder_proc_dir_entry_ root,
            binder_read_proc_transaction_log, &binder_transaction_log);
        create_proc_read_entry("failed_transaction_log", S_IRUGO,
            binder_proc_dir_entry_root, binder_read_proc_transaction_log,
            &binder_transaction_log_failed);
    }
    return ret;
}
device_initcall(binder_init);
```

其中，binder_init 是 Binder 驱动的初始化函数，初始化函数一般需要设备驱动接口来调用。Android Binder 设备驱动接口函数是 device_initcall，这与上一章提到的设备驱动的接口函数是 module_init 和 module_exit 不一样。通常来说，使用 module_init 和 module_exit 是为了同时兼容支持静态编译的驱动模块（buildin）和动态编译的驱动模块（module），但是 Binder 选择使用 device_initcall 的目的就是不让 Binder 驱动支持动态编译，而且需要在内核（Kernel）做镜像。initcall 用于注册进行初始化的函数；如果你的确需要将 Binder 驱动修改为动态的内核模块，可以直接将 device_initcall 修改为 module_init，但不要忘了增加 module_exit 的驱动卸载接口函数。

首先，初始化函数使用 proc_mkdir 创建了一个 Binder 的 proc 文件系统的根节点（binder_proc_dir_entry_root，/proc/binder），如果根节点创建成功，紧接着为 binder 创建 binder proc 节点（binder_proc_dir_entry_proc，/proc/binder/proc）；然后，Binder 驱动使用 misc_register 把自己注册为一个 Misc 设备，其设备节点位于/dev/binder，该节点由 init 进程在 handle_device_fd(device_fd)函数中调用 handle_device_event(&uevent)函数执行其中 uevent-netlink 事件在"/dev/"目录下创建。

最后，调用 create_proc_read_entry 创建以下只读 proc 文件：

- /proc/binder/state

- /proc/binder/stats
- /proc/binder/transactions
- /proc/binder/transaction_log
- /proc/binder/failed_transaction_log

在创建这些文件的过程中，同时也指定了操作这些文件的函数及其参数，如表 3-1 所示。

表 3-1　/proc/binder/目录下对应的文件操作函数及其参数

文件名	操作函数名称	参　数
state	binder_read_proc_state	NULL
stats	binder_read_proc_stats NULL	NULL
transactions	binder_read_proc_transactions	NULL
transaction_log	binder_read_proc_transaction_log	&binder_transaction_log
failed_transaction_log	binder_read_proc_transaction_log	&binder_transaction_log_failed

另外，在注册 Binder 驱动为 Misc 设备时，指定了 Binder 驱动的 miscdevice，具体实现如代码清单 3-16 所示。

代码清单 3-16　binder_miscdev 的实现

```
static struct miscdevice binder_miscdev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "binder",
    .fops = &binder_fops
};
```

Binder 设备的主设备号为 10，此设备号是动态获得的，.minor 被设置为动态获得设备号 MISC_DYNAMIC_MINOR；.name 代表设备名称。最后，指定了该设备的 file_operations 结构体，定义如代码清单 3-17 所示。

代码清单 3-17　file_operations 的定义

```
//Binder 文件操作结构体
static struct file_operations binder_fops = {
    .owner = THIS_MODULE,
    .poll = binder_poll,
    .unlocked_ioctl = binder_ioctl,
    .mmap = binder_mmap,
    .open = binder_open,
    .flush = binder_flush,
    .release = binder_release,
};
```

任何驱动程序都有向用户空间的程序提供操作接口这一主要功能，这个接口是标准的，对于 Android Binder 驱动来说，除了提供如表 3-1 所示的操作其只读文件的接口之外，还提供了

操作设备文件（/dev/binder）的接口。正如 binder_fops 所描述的 file_operations 结构体一样，其中主要包括了 binder_poll、binder_ioctl、binder_mmap、binder_open、binder_flush、binder_release 等标准操作接口。

下面我们将分别对 Android Binder 驱动所提供的这些操作接口进行分析。

12．binder_open

binder_open 函数用于打开 Binder 设备文件/dev/binder，对于 Android 驱动，任何一个进程及其内的所有线程都可以打开一个 Binder 设备，其打开过程的实现如代码清单 3-18 所示。

代码清单 3-18　binder_open 的实现

```
static int binder_open(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc;
    if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
        printk(KERN_INFO "binder_open: %d:%d\n", current->group_leader->pid,
        current->pid);
    //为 binder_proc 分配空间
    proc = kzalloc(sizeof(*proc), GFP_KERNEL);
    if (proc == NULL)
        return -ENOMEM;
    //增加引用计数
    get_task_struct(current);
    //保存其引用计数
    proc->tsk = current;
    //初始化 binder_proc 队列
    INIT_LIST_HEAD(&proc->todo);
    init_waitqueue_head(&proc->wait);
    proc->default_priority = task_nice(current);
    mutex_lock(&binder_lock);
    //增加 BINDER_STAT_PROC
    binder_stats.obj_created[BINDER_STAT_PROC]++;
    //添加到 binder_proc 哈希表
    hlist_add_head(&proc->proc_node, &binder_procs);
    //保存 pid 和 private_data 等数据
    proc->pid = current->group_leader->pid;
    INIT_LIST_HEAD(&proc->delivered_death);
    filp->private_data = proc;
    mutex_unlock(&binder_lock);
    //创建只读文件"/proc/binder/proc/$pid"
    if (binder_proc_dir_entry_proc) {
        char strbuf[11];
        snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
        remove_proc_entry(strbuf, binder_proc_dir_entry_proc);
        create_proc_read_entry(strbuf, S_IRUGO, binder_proc_dir_entry_proc,
        binder_read_proc_proc, proc);
    }
```

```
    return 0;
}
```

从上述代码中我们可以看出：

1）需要创建并分配一个 binder_proc 空间来保存 Binder 数据。

2）增加当前线程/进程的引用计数，并赋值给 binder_proc 的 tsk 字段。

3）初始化 binder_proc 队列，其中主要包括使用 INIT_LIST_HEAD 初始化链表头 todo，使用 init_waitqueue_head 初始化等待队列 wait，设置默认优先级（default_priority）为当前进程的 nice 值（通过 task_nice 得到当前进程的 nice 值）。

4）增加 BINDER_STAT_PROC 的对象计数，并通过 hlist_add_head 把创建的 binder_proc 对象添加到全局的 binder_proc 哈希表中，这样一来，任何一个进程就都可以访问到其他进程的 binder_proc 对象了。

5）把当前进程（或线程）的线程组的 pid（pid 指向线程 id）赋值给 proc 的 pid 字段，可以理解为一个进程 id（thread_group 指向线程组中的第一个线程的 task_struct 结构），同时把创建的 binder_proc 对象指针赋值给 filp 的 private_data 对象并保存起来。

6）在 binder proc 目录中创建只读文件/proc/binder/proc/$pid，用来输出当前 binder proc 对象的状态，文件名以 pid 命名，但需要注意的是，该 pid 字段并不是当前进程/线程的 id，而是线程组的 pid，也就是线程组中第一个线程的 pid（因为我们上面是将 current->group_leader->pid 赋值给该 pid 字段的）。另外，在创建该文件时，同样也指定了操作该文件的函数接口为 binder_read_proc_proc，其参数正是我们创建的 binder_proc 对象 proc，对于该接口的实现，后文将进行分析。

13．binder_release

binder_release 函数与 binder_open 函数的功能相反，当 Binder 驱动退出时，需要使用它来释放在打开以及其他操作过程中分配的空间并清理相关的数据信息，其实现如代码清单 3-19 所示。

代码清单 3-19　binder_release 的定义

```
static int binder_release(struct inode *nodp, struct file *filp)
{
    //取得 private_data 数据
    struct binder_proc *proc = filp->private_data;
    if (binder_proc_dir_entry_proc) {
        char strbuf[11];
        //找到 pid
        snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
        //移除 pid 命名的只读文件
        remove_proc_entry(strbuf, binder_proc_dir_entry_proc);
    }
    //释放 binder_proc 对象
```

```
    binder_defer_work(proc, BINDER_DEFERRED_RELEASE);
    return 0;
}
```

该函数的实现非常简单，首先取得 private_data 数据的使用权，找到当前进程、线程的 pid，这样就能得到在 open 过程中创建的以 pid 命名的用来输出当前 binder proc 对象的状态的只读文件；然后调用 remove_proc_entry 函数来进行删除；最后通过 binder_defer_work 函数和其参数 BINDER_DEFERRED_RELEASE 来释放整个 binder_proc 对象的数据和分配的空间。你可能会问，为什么不直接在这里释放，而是通过调用一个 workqueue 来释放呢？因为 Binder 的 release 和 flush 等操作比较复杂，而且也没有必要在系统调用里完成它，因此最好的方法是延迟执行这个任务，所以这里选择使用 workqueue（deferred）来提高系统的响应速度和性能。BINDER_DEFERRED_RELEASE 参数位于一个 enum 中，如代码清单 3-20 所示。

代码清单 3-20　BINDER_DEFERRED_RELEASE 参数的定义

```
enum {
    BINDER_DEFERRED_PUT_FILES    = 0x01,
    BINDER_DEFERRED_FLUSH        = 0x02,
    BINDER_DEFERRED_RELEASE      = 0x04,
};
```

也就是说，我们除了在 release 时使用该方式之外，还可以在执行 putfile 和 flush 等操作时使用该方式。关于释放 binder_proc 对象的函数 binder_defer_work，我们将在完成对设备文件操作接口的分析之后为大家详细地分析。

14．binder_flush

flush 操作接口将在关闭一个设备文件描述符复制时被调用。该函数的实现十分简单，如代码清单 3-21 所示。

代码清单 3-21　binder_flush 的实现

```
static int binder_flush(struct file *filp, fl_owner_t id)
{
    struct binder_proc *proc = filp->private_data;
    binder_defer_work(proc, BINDER_DEFERRED_FLUSH);
    return 0;
}
```

正如上面所说的，通过调用一个 workqueue 来执行 BINDER_DEFERRED_FLUSH 操作，从而完成该 flush 操作，但是最终的处理还是交给了 binder_defer_work 函数。

15．binder_poll

poll 函数是非阻塞型 IO 的内核驱动实现，所有支持非阻塞 IO 操作的设备驱动都需要实现 poll 函数。Binder 的 poll 函数仅支持设备是否可以非阻塞地读（POLLIN），这里有两种等待任

务：一种是 proc_work，另一种是 thread_work。同其他驱动的 poll 实现一样，这里也是通过调用 poll_wait 函数来实现的：具体实现如代码清单 3-22 所示。

代码清单 3-22　binder_poll 的实现

```
static unsigned int binder_poll(struct file *filp, struct poll_table_struct *wait)
{
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread = NULL;
    int wait_for_proc_work;

    mutex_lock(&binder_lock);
    //得到当前进程的信息
    thread = binder_get_thread(proc);

    wait_for_proc_work = thread->transaction_stack == NULL &&
        list_empty(&thread->todo) && thread->return_error == BR_OK;
    mutex_unlock(&binder_lock);
    //proc_work 方式
    if (wait_for_proc_work) {
        if (binder_has_proc_work(proc, thread))
            return POLLIN;
        poll_wait(filp, &proc->wait, wait);
        if (binder_has_proc_work(proc, thread))
            return POLLIN;
    } else {//thread_work 方式
        if (binder_has_thread_work(thread))
            return POLLIN;
        poll_wait(filp, &thread->wait, wait);
        if (binder_has_thread_work(thread))
            return POLLIN;
    }
    return 0;
}
```

首先需要取得当前进程/线程的信息，由于所有的线程信息都存储在 binder_proc 结构体的 threads 队列中，所以我们可以通过 binder_get_thread(proc)来找到当前进程/线程的相关信息，稍后会详细分析查找原理。proc_work 和 thread_work 方式如代码清单 3-23 所示。

代码清单 3-23　proc_work 和 thread_work 的实现

```
static int binder_has_proc_work(struct binder_proc *proc, struct binder_thread *thread)
{
    return !list_empty(&proc->todo) || (thread->looper &
        BINDER_LOOPER_STATE_NEED_RETURN);
}
static int binder_has_thread_work(struct binder_thread *thread)
{
```

```
    return !list_empty(&thread->todo) || thread->return_error != BR_OK ||
        (thread->looper & BINDER_LOOPER_STATE_NEED_RETURN);
}
```

其中，主要是通过检测线程队列是否为空、线程的循环状态以及返回信息来判断所要采用的等待方式，最后通过调用 poll_wait 函数来实现 poll 操作。

16．binder_get_thread

该函数可以用于在 threads 队列中查找当前的进程信息，实现如代码清单 3-24 所示。

代码清单 3-24　binder_get_thread 的实现

```
static struct binder_thread *binder_get_thread(struct binder_proc *proc)
{
    struct binder_thread *thread = NULL;
    struct rb_node *parent = NULL;
    //取得队列中的线程
    struct rb_node **p = &proc->threads.rb_node;

    while (*p) {
        parent = *p;
        // 取得将要进行比较的线程
        thread = rb_entry(parent, struct binder_thread, rb_node);
        //比较
        if (current->pid < thread->pid)
            p = &(*p)->rb_left;
        else if (current->pid > thread->pid)
            p = &(*p)->rb_right;
        else
            break;
    }
    //如果不存在
    if (*p == NULL) {
        //创建一个新进程
        thread = kzalloc(sizeof(*thread), GFP_KERNEL);
        if (thread == NULL)
            return NULL;
        //初始化新进程的一系列数据信息
        binder_stats.obj_created[BINDER_STAT_THREAD]++;
        thread->proc = proc;
        thread->pid = current->pid;
        //初始化等待队列
        init_waitqueue_head(&thread->wait);
        //初始化链表头
        INIT_LIST_HEAD(&thread->todo);
        //加入到队列
        rb_link_node(&thread->rb_node, parent, p);
        rb_insert_color(&thread->rb_node, &proc->threads);
```

```
        //设置其循环状态
        thread->looper |= BINDER_LOOPER_STATE_NEED_RETURN;
        thread->return_error = BR_OK;
        thread->return_error2 = BR_OK;
    }
    return thread;
}
```

实现原理是通过在队列中比较各个线程的 pid 与当前线程的 pid 是否相同，如果找到了，把它的线程信息返回；如果没找到，新建一个线程并把它加入到队列中，然后初始化就绪线程队列等。代码的注释已经把每一个步骤都讲清楚了。

17．binder_mmap

mmap（memory map）用于把设备内存映射到用户进程地址空间中，这样就可以像操作用户内存那样操作设备内存。Binder 设备对内存映射是有限制的，比如，Binder 设备最大能映射 4MB 的内存区域，Binder 不能映射具有写权限的内存区域等。不同于一般的设备驱动，大多设备映射的设备内存是设备本身具有的，或者是在驱动初始化时由 vmalloc 或 kmalloc 等内核内存函数分配的，Binder 的设备内存是在 mmap 操作时分配的。分配的方法是：先在内核虚拟映射表上获取一个可以使用的区域，然后分配物理页，并把物理页映射到获取的虚拟空间上。由于设备内存是在 mmap 操作中实现的，因此每个进程/线程只能执行一次映射操作，其后的操作都会返回错误，具体实现如代码清单 3-25 所示。

代码清单 3-25　binder_mmap 的实现

```
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
    int ret;
    struct vm_struct *area;
    struct binder_proc *proc = filp->private_data;
    const char *failure_string;
    struct binder_buffer *buffer;
    //检测映射内存的大小
    if ((vma->vm_end - vma->vm_start) > SZ_4M)
        vma->vm_end = vma->vm_start + SZ_4M;
    if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
        printk(KERN_INFO
            "binder_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
            proc->pid, vma->vm_start, vma->vm_end,
            (vma->vm_end - vma->vm_start) / SZ_1K, vma->vm_flags,
            (unsigned long)pgprot_val(vma->vm_page_prot));
    //检测 flags
    if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
        ret = -EPERM;
        failure_string = "bad vm_flags";
        goto err_bad_arg;
```

```
    }
    vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;
    //判断是否已经 mmap 过
    if (proc->buffer) {
        ret = -EBUSY;
        failure_string = "already mapped";
        goto err_already_mapped;
    }
    //申请虚拟空间
    area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
    if (area == NULL) {
        ret = -ENOMEM;
        failure_string = "get_vm_area";
        goto err_get_vm_area_failed;
    }
    proc->buffer = area->addr;
    proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;
#ifdef CONFIG_CPU_CACHE_VIPT
    if (cache_is_vipt_aliasing()) {
        while (CACHE_COLOUR((vma->vm_start ^ (uint32_t)proc->buffer))) {
            printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p bad alignment\n",
            proc->pid, vma->vm_start, vma->vm_end, proc->buffer);
            vma->vm_start += PAGE_SIZE;
        }
    }
#endif
    //分配 pages 空间
    proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start)
    / PAGE_SIZE), GFP_KERNEL);
    if (proc->pages == NULL) {
        ret = -ENOMEM;
        failure_string = "alloc page array";
        goto err_alloc_pages_failed;
    }
    proc->buffer_size = vma->vm_end - vma->vm_start;
    vma->vm_ops = &binder_vm_ops;
    vma->vm_private_data = proc;
    //分配物理内存
    if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE,
    vma)) {
        ret = -ENOMEM;
        failure_string = "alloc small buf";
        goto err_alloc_small_buf_failed;
    }
    buffer = proc->buffer;
    INIT_LIST_HEAD(&proc->buffers);
    list_add(&buffer->entry, &proc->buffers);
    buffer->free = 1;
```

```
        binder_insert_free_buffer(proc, buffer);
        proc->free_async_space = proc->buffer_size / 2;
        barrier();
        proc->files = get_files_struct(current);
        proc->vma = vma;
        return 0;
    err_alloc_small_buf_failed:
        kfree(proc->pages);
        proc->pages = NULL;
    err_alloc_pages_failed:
        vfree(proc->buffer);
        proc->buffer = NULL;
    err_get_vm_area_failed:
    err_already_mapped:
    err_bad_arg:
        printk(KERN_ERR "binder_mmap: %d %lx-%lx %s failed %d\n", proc->pid, vma->
        vm_start, vma->vm_end, failure_string, ret);
        return ret;
    }
```

首先介绍该函数的第二个参数 vm_area_struct 结构体，它在 mmp 的实现中会用到。为了优化查找方法，内核维护了 VMA 的链表和树形结构，vm_area_struct 中很多成员函数都是用来维护这个结构的。VMA 的作用是管理进程地址空间中不同区域的数据结构。该函数首先对内存映射进行检查（主要包括：映射内存的大小、flags 以及是否已经映射过了），判断其映射条件是否合法；然后，通过内核函数 get_vm_area 从系统中申请可用的虚拟内存空间（注意：不是物理内存空间），即在内核中申请并保留一块连续的内核虚拟内存空间区域；接着，将 binder_proc 的用户地址偏移（即用户进程的 VMA 地址与 Binder 申请的 VMA 地址的偏差）存放到 proc->user_buffer_offset 中；再接着，使用 kzalloc 函数根据请求映射的内存空间大小，分配 Binder 的核心数据结构 binder_proc 的 pages 成员，它主要用来保存指向申请的物理页的指针；最后，为 VMA 指定了 vm_operations_struct 操作，并且将 vma->vm_private_data 指向了核心数据 proc。到这里，我们就可以真正地开始分配物理内存（page）了，其物理内存的分配是通过 binder_update_page_range 来实现的。binder_update_page_range 的实现主要是一些内核操作函数，我们就不再贴出代码了，只是重点介绍一下该函数所完成的工作：

- alloc_page　　分配页面
- map_vm_area　　为分配的内存做映射关系
- vm_insert_page　　把分配的物理页插入到用户 VMA 区域

最后，对 Binder 的 binder_mmap 进行一个简单的总结，实现步骤如下：

1）检查内存映射条件，包括映射内存大小（4MB）、flags、是否是第一次 mmap 等。

2）获得地址空间，并把此空间的地址记录在进程信息（buffer）中。

3）分配物理页面（pages）并记录下来。

4）将 buffer 插入到进程信息的 buffer 列表中。

5）调用 binder_update_page_range 函数将分配的物理页面和 vm 空间对应起来。

6）通过 binder_insert_free_buffer 函数把此进程的 buffer 插入到进程信息中。

这里我们再来看一下 vm_operations_struct 操作的定义，如代码清单 3-26 所示。

代码清单 3-26　binder_vm_ops 定义

```
static struct vm_operations_struct binder_vm_ops = {
    .open = binder_vma_open,
    .close = binder_vma_close,
};
```

其中只包括了一个打开操作和一个关闭操作，由于篇幅关系，这里就不再贴出代码了，它们的具体实现大家可以分别参考源码中指定的函数 binder_vma_open 和 binder_vma_close。需要说明的是，打开操作中使用了一个 dump_stack 函数，它主要用来输出内核的一些相关信息（比如堆栈信息等），这样方便调试；而在关闭函数中再次使用了 binder_defer_work 函数，但是这里使用的参数则是 BINDER_DEFERRED_PUT_FILES。

18．binder_ioctl

到这里，终于进入 Binder 最核心的部分了，Binder 的功能就是通过 ioctl 命令来实现的。Binder 的 ioctl 命令共有 7 个，定义在 ioctl.h 头文件中，如代码清单 3-27 所示。

代码清单 3-27　Binder 的 ioctl 命令

```
#define BINDER_WRITE_READ          _IOWR('b', 1, struct binder_write_read)
#define  BINDER_SET_IDLE_TIMEOUT   _IOW('b', 3, int64_t)
#define  BINDER_SET_MAX_THREADS    _IOW('b', 5, size_t)
#define  BINDER_SET_IDLE_PRIORITY  _IOW('b', 6, int)
#define  BINDER_SET_CONTEXT_MGR    _IOW('b', 7, int)
#define  BINDER_THREAD_EXIT        _IOW('b', 8, int)
#define BINDER_VERSION             _IOWR('b', 9, struct binder_version)
```

在这 7 个命令中，BINDER_SET_IDLE_TIMEOUT 和 INDER_SET_IDLE_PRIORITY 还没有被实现；BINDER_WRITE_READ 是所有 Binder 的基础，即读写操作；BINDER_SET_MAX_THREADS 用于设置最大线程数目；BINDER_SET_CONTEXT_MGR 则被 Service Manager 用于设置自己作为 context manager 节点，凡是为 0 的 handle 都是指这个 master 节点；BINDER_THREAD_EXIT 用于删除线程信息；BINDER_VERSION 很简单，用于返回版本信息。ioctl 在前面的驱动分析中我们已经提到过，这里将对其进行更深入的分析。

ioctl 是设备驱动程序中对设备的 I/O 通道进行管理的函数。所谓对 I/O 通道进行管理，就是对设备的一些特性进行控制，例如串口的传输波特率、马达的转速等。它的调用函数是“int ioctl(int fd, ind cmd, …);”，其中，fd 就是用户程序打开设备时使用 open 函数返回的文件标识符；cmd 就是用户程序对设备的控制命令；至于后面的省略号，那是一些补充参数，一般最多一个，有或没有与 cmd 的意义相关。ioctl 函数是文件结构中的一个属性分量，也就是说，如果你的驱动程序提供了对 ioctl 的支持，用户就可以在用户程序中使用 ioctl 函数控制设备的 I/O 通道。

如果不用 ioctl，也可以实现对设备的 I/O 通道的控制，但那就太复杂了。例如，我们可以在驱动程序中实现 write 的时候检查一下是否有特殊约定的数据流通过，如果有，那么后面就在后面附加控制命令（一般在 socket 编程中常常这样做）。但是，如果这样做，就会导致代码分工不明确，程序结构混乱。所以，我们就使用 ioctl 来实现控制的功能。要记住，用户程序所做的只是通过命令码告诉驱动程序它想做什么，至于怎么解释这些命令和怎么实现这些命令，这都是驱动程序要做的事情，稍候我们将分析在驱动中如何实现这些命令。

在 ioctl 函数体内有一个 switch{case}结构，每一个 case 对应一个命令码，它们用于执行相应的操作。因为在 ioctl 中，命令码是唯一联系用户程序和驱动程序的途径。下面我们来分析命令码是如何组成的。

一定要做到命令和设备是一一对应的，这样才不会将正确的命令发给错误的设备，或者是把错误的命令发给正确的设备，或者是把错误的命令发给错误的设备，这些错误都会导致不可预料的事情发生。所以，在 Linux 内核中这样定义一个命令码，如表 3-2 所示。

表 3-2　命令码构成

设备类型	序列号	方向	数据尺寸
8 bit	8 bit	2 bit	8~14 bit

这样一来，一个命令就变成了一个整数形式的命令码。但是，命令码非常不直观，所以 Linux 内核中提供了一些宏，这些宏可以根据便于理解的字符串生成命令码，或者是从命令码得到一些用户可以理解的字符串，以标明这个命令对应的设备类型、设备序列号、数据传送方向和传输数据大小。幻数是一个字母，数据长度也是 8，所以就用一个特定的字母来标明设备类型，这与使用数字的原理是一样的，只是更加利于记忆和理解。

正如上面所述，每个命令都被定义成一个_IOW 宏，关于_IOW 宏的定义大家可以在“bionic\libc\kernel\common\asm-generic\Ioctl.h”中找到，如代码清单 3-28 所示。

代码清单 3-28　_IOW 宏的定义

```
#define  _IOC_NRBITS       8     //序数（number）字段的字位宽度，8bits
#define  _IOC_TYPEBITS     8     //类型（type）字段的字位宽度，8bits
#define  _IOC_SIZEBITS     14    //大小（size）字段的字位宽度，14bits
#define  _IOC_DIRBITS      2     //方向（direction）字段的字位宽度，2bits
#define  _IOC_NRMASK      ((1 << _IOC_NRBITS)-1)     //序数字段的掩码，0x000000FF
#define  _IOC_TYPEMASK    ((1 << _IOC_TYPEBITS)-1)  //类型字段的掩码，0x000000FF
#define  _IOC_SIZEMASK    ((1 << _IOC_SIZEBITS)-1)  //大小字段的掩码，0x00003FFF
#define  _IOC_DIRMASK     ((1 << _IOC_DIRBITS)-1)   //方向字段的掩码，0x00000003
#define  _IOC_NRSHIFT    0                          //序数字段在整个字段中的位移，0
#define  _IOC_TYPESHIFT   (_IOC_NRSHIFT+_IOC_NRBITS)       //类型字段的位移，8
#define  _IOC_SIZESHIFT   (_IOC_TYPESHIFT+_IOC_TYPEBITS)   //大小字段的位移，16
#define  _IOC_DIRSHIFT    (_IOC_SIZESHIFT+_IOC_SIZEBITS)   //方向字段的位移，30
//方向标志
#define _IOC_NONE    0U     //没有数据传输
```

```
#define _IOC_WRITE   1U     //向设备中写入数据，驱动程序必须从用户空间读入数据
#define _IOC_READ    2U     //从设备中读取数据，驱动程序必须向用户空间中写入数据
//构造命令
#define _IOC(dir,type,nr,size) \
       (((dir)  << _IOC_DIRSHIFT) | \
       ((type) << _IOC_TYPESHIFT) | \
       ((nr)   << _IOC_NRSHIFT) | \
       ((size) << _IOC_SIZESHIFT))
//构造无参数的命令编号
#define _IO(type,nr)          _IOC(_IOC_NONE,(type),(nr),0)
//构造从驱动程序中读取数据的命令编号
#define _IOR(type,nr,size)    _IOC(_IOC_READ,(type),(nr),sizeof(size))
//用于向驱动程序写入数据的命令
#define _IOW(type,nr,size)    _IOC(_IOC_WRITE,(type),(nr),sizeof(size))
//用于双向传输
#define _IOWR(type,nr,size)  _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),sizeof(size))
//从命令参数中解析出数据方向，即写进还是读出
#define _IOC_DIR(nr)         (((nr) >> _IOC_DIRSHIFT) & _IOC_DIRMASK)
//从命令参数中解析出类型 type
#define _IOC_TYPE(nr)        (((nr) >> _IOC_TYPESHIFT) & _IOC_TYPEMASK)
//从命令参数中解析出序数 number
#define _IOC_NR(nr)          (((nr) >> _IOC_NRSHIFT) & _IOC_NRMASK)
//从命令参数中解析出用户数据大小
#define _IOC_SIZE(nr)        (((nr) >> _IOC_SIZESHIFT) & _IOC_SIZEMASK)
```

现在我们应该已经明白了 ioctl 的深层次含义，下面就来对这些命令的实现一一进行分析。

（1）BINDER_VERSION

该命令的实现过程如代码清单 3-29 所示：

代码清单 3-29　BINDER_VERSION 命令的实现

```
if (size != sizeof(struct binder_version)) {
     ret = -EINVAL;
     goto err;
}
//将 Binder 的版本返回给调用进程
if (put_user(BINDER_CURRENT_PROTOCOL_VERSION, &((struct binder_version *)ubuf)->
     protocol_version)) {
     ret = -EINVAL;
     goto err;
}
```

其中 BINDER_CURRENT_PROTOCOL_VERSION 表示当前的版本信息，定义于 binder.h 中。binder_version 表示版本信息的一个结构体，其成员 protocol_version 则表示具体的版本信息。

（2）BINDER_SET_MAX_THREADS

该命令用于设置进程的 Binder 对象所支持的最大线程数，进程会根据该数目来决定线程池的容量。设置的值保存在 binder_proc 结构的 max_threads 成员里，实现如代码清单 3-30 所示。

代码清单 3-30　BINDER_SET_MAX_THREADS 命令的实现

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

（3）BINDER_THREAD_EXIT

该命令用于释放相应的线程信息，其具体过程在 binder_free_thread 函数中实现，用来终止并释放 binder_thread 对象及其 binder_transaction 事务。由于该接口涉及一些事务相关的内容，因此我们在后面具体为大家分析。

（4）BINDER_SET_CONTEXT_MGR

如果一个进程（或线程）能被成功设置成 binder_context_mgr_node 对象，那么称这个进程为 Context Manager（context_mgr）。该命令就是将一个线程/进程设置为 Context Manager，也就是设置驱动中的全局变量 binder_context_ mgr_uid 为当前进程的 uid，并初始化一个 binder_node 并赋值给全局变量 binder_context_mgr_ node。该命令一般是在系统启动时初始化 Binder 驱动的过程中被调用，并且也只有创建 binder_context_mgr_node 对象的 Binder 上下文管理进程/线程才有权限重新设置这个对象。该命令的实现如代码清单 3-31 所示：

代码清单 3-31　BINDER_SET_CONTEXT_MGR 命令的实现

```
if (binder_context_mgr_node != NULL) {
    printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
    ret = -EBUSY;
    goto err;
}
if (binder_context_mgr_uid != -1) {
    if (binder_context_mgr_uid != current->cred->euid) {
        printk(KERN_ERR "binder: BINDER_SET_"
                "CONTEXT_MGR bad uid %d != %d\n",
                current->cred->euid,
                binder_context_mgr_uid);
                ret = -EPERM;
                goto err;
    }
} else  //取得进程权限
    binder_context_mgr_uid = current->cred->euid;
//创建 binder_node 节点
binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
if (binder_context_mgr_node == NULL) {
    ret = -ENOMEM;
    goto err;
}
//初始化 binder_node 节点数据
binder_context_mgr_node->local_weak_refs++;
```

```
binder_context_mgr_node->local_strong_refs++;
binder_context_mgr_node->has_strong_ref = 1;
binder_context_mgr_node->has_weak_ref = 1;
```

从上面的代码中可以看出，首先检测 binder_context_mgr_node 是否为 NULL，然后检测 binder_context_mgr_uid 是否存在。如果存在，继续检测它是否是当前进程的 uid（这里主要检查当前进程是否有操作该命令的权限）；如果不存在，就对 binder_context_mgr_uid 进行赋值，取得当前进程对文件和资源的操作权限，即 euid。最后，通过 binder_new_node 函数创建一个 binder_node 节点，并对该 binder_node 节点执行初始化操作。下面我们分析如何创建这个 binder_node 节点，其具体实现如代码清单 3-32 所示。

代码清单 3-32　binder_new_node 的实现

```
static struct binder_node *binder_new_node(struct binder_proc *proc, void __user *ptr,
    void __user *cookie)
{
    struct rb_node **p = &proc->nodes.rb_node;
    struct rb_node *parent = NULL;
    struct binder_node *node;
    //查找第一个叶节点
    while (*p) {
        parent = *p;
        node = rb_entry(parent, struct binder_node, rb_node);

        if (ptr < node->ptr)
            p = &(*p)->rb_left;
        else if (ptr > node->ptr)
            p = &(*p)->rb_right;
        else
            return NULL;
    }
    //创建 binder_node 节点
    node = kzalloc(sizeof(*node), GFP_KERNEL);
    if (node == NULL)
        return NULL;
    binder_stats.obj_created[BINDER_STAT_NODE]++;
    //插入节点
    rb_link_node(&node->rb_node, parent, p);
    rb_insert_color(&node->rb_node, &proc->nodes);
    //初始化数据
    node->debug_id = ++binder_last_id;
    node->proc = proc;
    node->ptr = ptr;
    node->cookie = cookie;
    node->work.type = BINDER_WORK_NODE;
    //初始化链表头
    INIT_LIST_HEAD(&node->work.entry);
    INIT_LIST_HEAD(&node->async_todo);
```

```
    if (binder_debug_mask & BINDER_DEBUG_INTERNAL_REFS)
        printk(KERN_INFO "binder: %d:%d node %d u%p c%p created\n",
              proc->pid, current->pid, node->debug_id,
              node->ptr, node->cookie);
    return node;
}
```

binder_proc 的成员 node 是 binder_node 的根节点，这是一棵红黑树（一种平衡二叉树）。该函数首先根据规则找到第一个叶节点作为新插入的节点的父节点，然后创建 binder_node 节点并插入。这里需要说明一下，rb_link_node 和 rb_insert_color 都是内核红黑树函数。rb_link_node 是一个内联函数，它用于将新节点插入到红黑树中的指定父节点下。rb_insert_color 则是把已经插入到红黑树中的节点调整并融合到红黑树中。最后，执行数据初始化和初始化该节点的链表头，其中 node->proc 保存着 binder_proc 对象指针。

（5）BINDER_WRITE_READ

该命令才是 Binder 最核心的部分，Binder 的 IPC 机制就是通过这个接口来实现的，具体实现如代码清单 3-33 所示。

代码清单 3-33　BINDER_WRITE_READ 命令的实现

```
struct binder_write_read bwr;
//判断数据的完整性
if (size != sizeof(struct binder_write_read)) {
    ret = -EINVAL;
    goto err;
}
//从用户空间复制数据
if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
    ret = -EFAULT;
    goto err;
}
if (binder_debug_mask & BINDER_DEBUG_READ_WRITE)
    printk(KERN_INFO "binder: %d:%d write %ld at %08lx, read %ld at %08lx\n",
            proc->pid, thread->pid, bwr.write_size, bwr.write_buffer,
            bwr.read_size, bwr.read_buffer);
//执行写操作
if (bwr.write_size > 0) {
    ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer, bwr.
    write_size, &bwr.write_consumed);
    if (ret < 0) {
        bwr.read_consumed = 0;
        if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
            ret = -EFAULT;
        goto err;
    }
}
//执行读操作
```

```
if (bwr.read_size > 0) {
    ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer, bwr.
    read_size, &bwr.read_consumed, filp->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 (binder_debug_mask & BINDER_DEBUG_READ_WRITE)
    printk(KERN_INFO "binder: %d:%d wrote %ld of %ld, read return %ld of %ld\n",
            proc->pid, thread->pid, bwr.write_consumed, bwr.write_size, bwr.
            read_consumed, bwr.read_size);
//将数据复制到用户空间
if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
    ret = -EFAULT;
    goto err;
}
```

该部分的实现很简单，首先检查其数据是否完整，然后从用户空间复制数据到 binder_write_read 结构体中，最后通过 write_size 和 bwr.read_size 来判断需要执行的操作。当然，最终的操作会通过 inder_thread_write 和 binder_thread_read 函数来实现。稍候我们将分析具体的读写操作，处理完成之后再将数据复制到用户空间即可。

现在我们已经熟悉了这些命令的操作，在 ioctl 函数中是先通过 wait_event_interruptible 函数来修改 task 的状态为 TASK_INTERRUPTIBLE，使得该进程将不会继续运行，直到它被唤醒，然后添加到等待队列 binder_user_error_wait（该函数的第一个参数）中。

wait_event_interruptible 的实现也很简单，如代码清单 3-34 所示。

代码清单 3-34　wait_event_interruptible 的实现

```
#define wait_event_interruptible(wq, condition)
({
   int __ret = 0;
   if (!(condition))
      __wait_event_interruptible(wq, condition, __ret);
   __ret;
})
```

判断是否满足 condition 条件，如果是，则直接返回 0；否则，调用函数__wait_event_interruptible()，并用__ret 来存放返回值。

现在我们就来分析前面提到的一些处理方法和其他几个比较底层的函数。首先是前面已经说过多次的释放 binder_proc 对象的函数 binder_defer_work，其定义如代码清单 3-35 所示。

代码清单 3-35　binder_defer_work 的实现

```
//释放 binder_proc 对象
static void binder_defer_work(struct binder_proc *proc, int defer)
{
    mutex_lock(&binder_deferred_lock);
    proc->deferred_work |= defer;
    if (hlist_unhashed(&proc->deferred_work_node)) {
            hlist_add_head(&proc->deferred_work_node,&binder_deferred_list);
        schedule_work(&binder_deferred_work);
    }
    mutex_unlock(&binder_deferred_lock);
}
```

操作一目了然，这里使用了 schedule_work 来调度一个线程去执行我们需要执行的操作。接下来分析释放线程的函数 binder_free_thread，其定义如代码清单 3-36 所示。

代码清单 3-36　binder_free_thread 的实现

```
static int binder_free_thread(struct binder_proc *proc, struct binder_thread *thread)
{
    struct binder_transaction *t;
    struct binder_transaction *send_reply = NULL;
    int active_transactions = 0;
    //将当前线程从红杉树上删除
    rb_erase(&thread->rb_node, &proc->threads);
    //取得 binder_transaction 数据
    t = thread->transaction_stack;
    //判断要释放的是否是“回复”进程
    if (t && t->to_thread == thread)
        send_reply = t;
    //释放所有的 binder_transaction
    while (t) {
        active_transactions++;
        if (binder_debug_mask & BINDER_DEBUG_DEAD_TRANSACTION)
            printk(KERN_INFO "binder: release %d:%d transaction %d %s, still
            active\n",
                proc->pid, thread->pid, t->debug_id, (t->to_thread == thread) ?
                "in" : "out");
        if (t->to_thread == thread) {
            t->to_proc = NULL;
            t->to_thread = NULL;
            if (t->buffer) {
                t->buffer->transaction = NULL;
                t->buffer = NULL;
            }
            t = t->to_parent;
        } else if (t->from == thread) {
```

```
                t->from = NULL;
                t = t->from_parent;
            } else
                BUG();
        }
        //需要发送失败回复
        if (send_reply)
            binder_send_failed_reply(send_reply, BR_DEAD_REPLY);
        //释放 binder_work
        binder_release_work(&thread->todo);
        //释放 binder_thread
        kfree(thread);
        //改变 Binder 的状态
        binder_stats.obj_deleted[BINDER_STAT_THREAD]++;
        return active_transactions;
    }
```

该函数所执行的操作是：首先，将当前要删除的线程从红杉树上删除；然后，取得其 binder_transaction 数据，并且通过 to_thread 和 fram 来判断线程的类型。如果需要回复，那么在释放完成之后就通过 binder_send_failed_reply 发送一个失败的回复，在释放 binder_transaction 时需要释放每个节点。这里，释放 binder_work 需要使用 binder_release_work 函数，其定义如代码清单 3-37 所示。

代码清单 3-37　binder_release_work 的实现

```
static void binder_release_work(struct list_head *list)
{
    struct binder_work *w;
    //检查是否为 NULL
    while (!list_empty(list)) {
        w = list_first_entry(list, struct binder_work, entry);
        list_del_init(&w->entry);
        switch (w->type) {
        case BINDER_WORK_TRANSACTION: {
            struct binder_transaction *t = container_of(w,
                struct binder_transaction, work);
            if (t->buffer->target_node && !(t->flags & TF_ONE_WAY))
                binder_send_failed_reply(t, BR_DEAD_REPLY);
        } break;
        case BINDER_WORK_TRANSACTION_COMPLETE: {
            kfree(w);
            binder_stats.obj_deleted[BINDER_STAT_TRANSACTION_COMPLETE]++;
        } break;
        default:
            break;
        }
    }
}
```

其中，同样对其类型进行了判断，如果是 BINDER_WORK_TRANSACTION 类型，则需要发送 BR_DEAD_REPLY 回复；如果为 BINDER_WORK_TRANSACTION_COMPLETE 类型，则直接释放，然后改变其状态即可。

通过前面的学习，我们对 Binder 的整个工作流程有了一个深入的认识，关于 Binder 的实现，还涉及有很多的细节，大家可以仔细阅读 Binder 驱动的源代码进一步了解。

3.3 Binder 的构架与实现

上一节分析了 Android 中的 IPC 机制——Binder 驱动的实现。我们知道 Binder 在 Android 中占据着重要地位，需要完成进程之间通信的所有应用程序都使用了 Binder 机制，所以 Android 也对 Binder 驱动所提供的接口进行了封装，同时还在 Android 的工具库中提供了一套 Binder 库，这正是本节需要分析的内容。

3.3.1 Binder 的系统构架

在分析 Binder 的系统构架之前，首先举例说明 Binder 的用处。在 Android 的设计中，每个 Activity 都是一个独立的进程，每个 Service 也是一个独立的进程，而 Activity 要与 Service 进行通信，就是跨进程的通信，这时就需要使用 Binder 机制了。这里可以把 Activity 看作一个客户端，把 Service 看作一个服务端，实际上也就是一个客户端与服务端之间的通信，具体的通信流程则由 Binder 来完成。

1．Binder 机制的组成

Android 的 Binder 机制就是一个 C/S 构架，客户端和服务端直接通过 Binder 交互数据，打开 Binder 写入数据，通过 Binder 读取数据，这样通讯就可以完成了。数据的读写是由上一节所介绍的 Binder 驱动完成的，除了 Binder 驱动外，整个机制还包括以下几个组成部分：

（1）Service Manager

Service Manager 主要负责管理 Android 系统中所有的服务，当客户端要与服务端进行通信时，首先就会通过 Service Manager 来查询和取得所需要交互的服务。当然，每个服务也都需要向 Service Manager 注册自己提供的服务，以便能够供客户端进行查询和获取。

（2）服务（Server）

这里的服务即上面所说的服务端，通常也是 Android 的系统服务，通过 Service Manager 可以查询和获取某个 Server。

（3）客户端

这里的客户端一般是指 Android 系统上面的应用程序。它可以请求 Server 中的服务，比如 Activity。

（4）服务代理

服务代理是指在客户端应用程序中生成的 Server 代理（proxy）。从应用程序的角度来看，

Linux公社（LinuxIDC.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

LinuxIDC.com提供包括Ubuntu，Fedora，SUSE技术，以及最新IT资讯等Linux专业类网站。

并被收录到Google 网页目录-计算机 > 软件 > 操作系统 > Linux 目录下。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

包括:

Ubuntu专题

Fedora专题

RedHat专题

SUSE专题

红旗Linux专题

Android专题