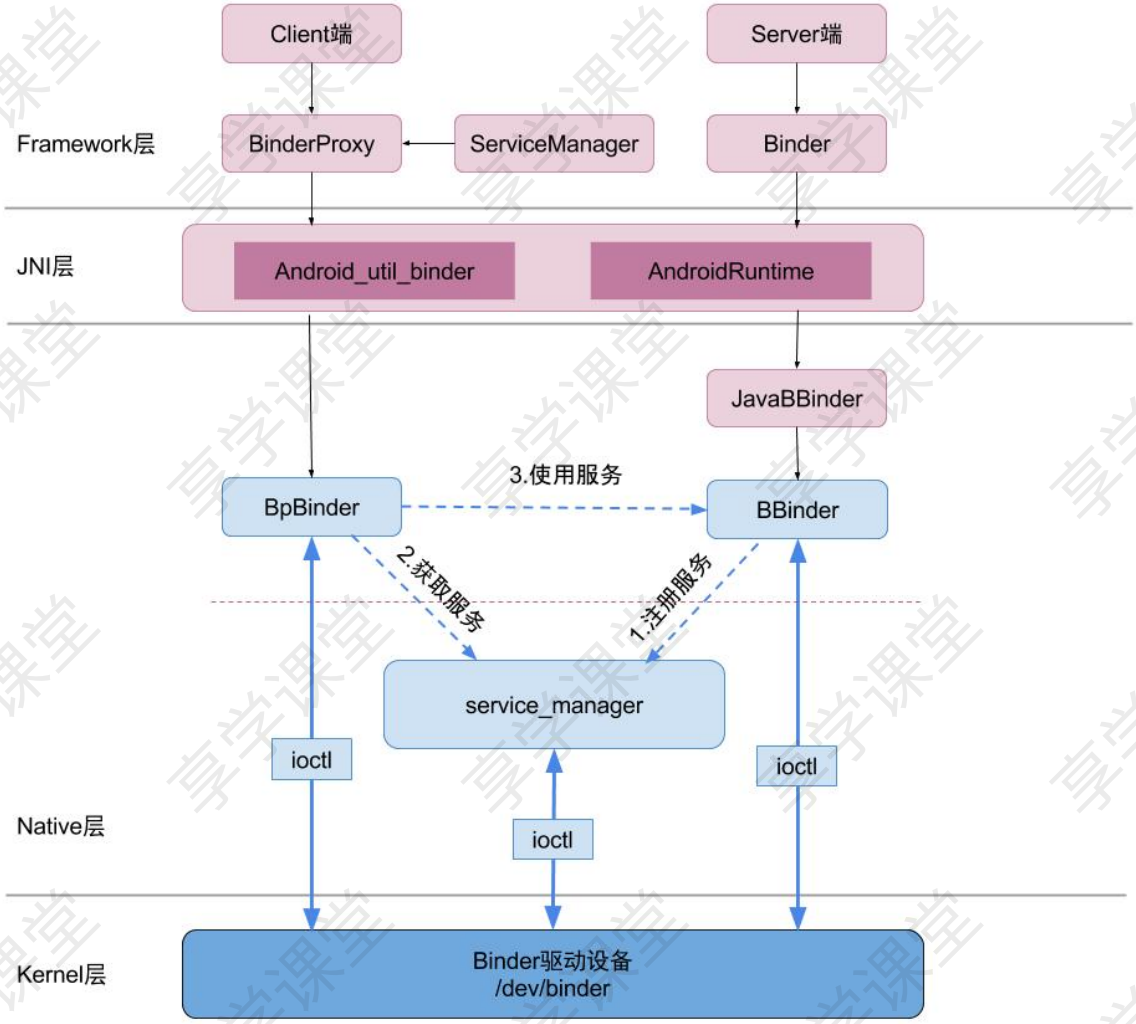
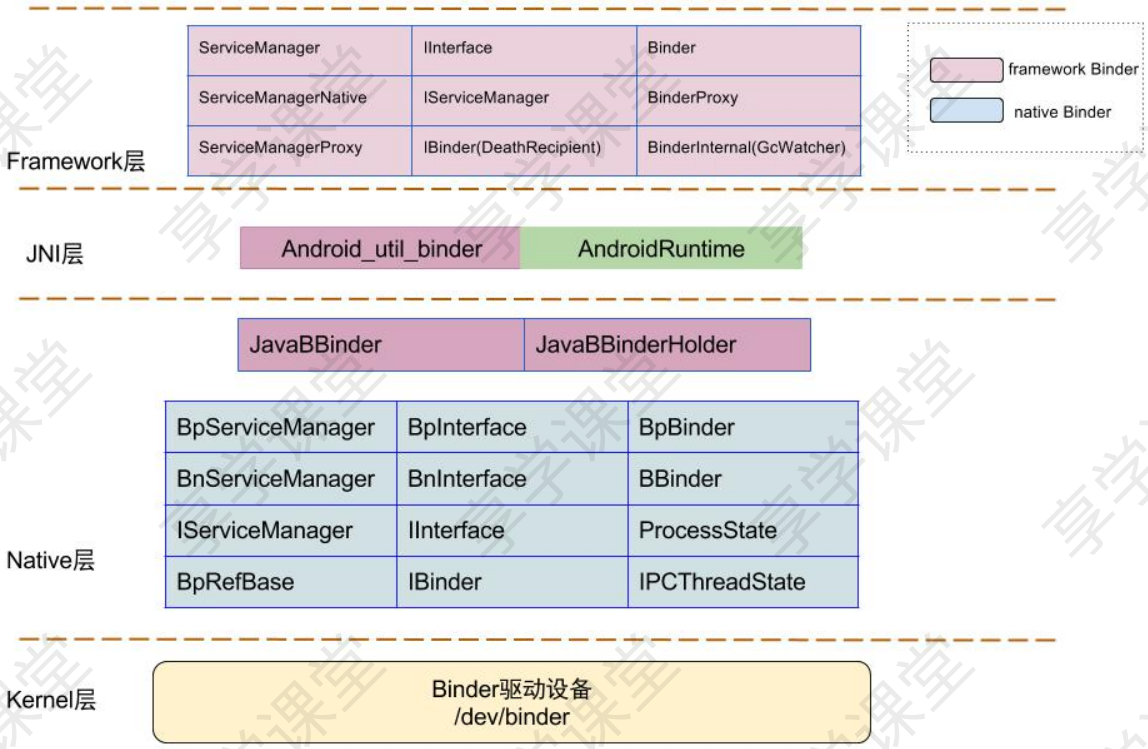
图片

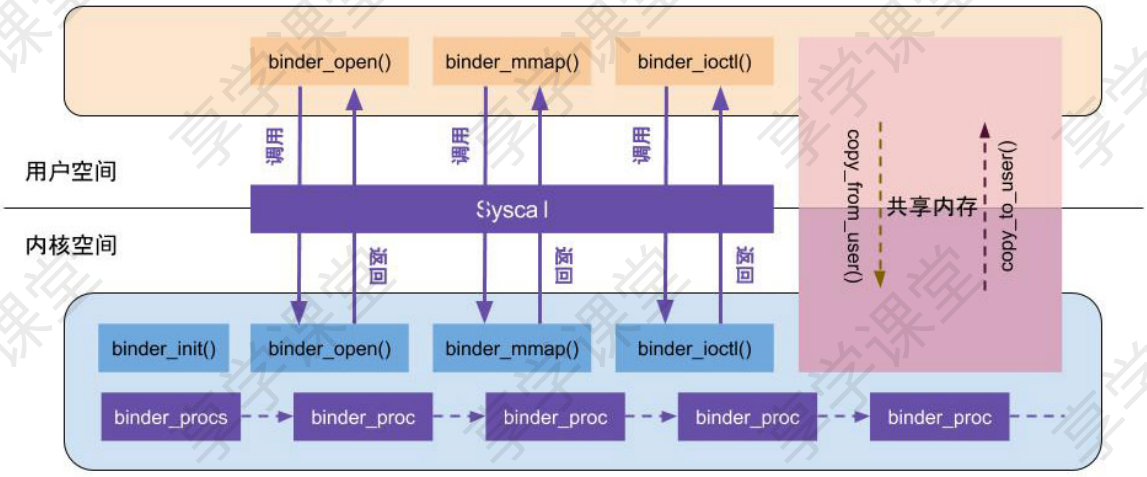
binder框架



binder涉及到的类



binder驱动



binder的jni方法注册

1.zygote启动

1-1.启动zygote进程

zygote是由init进程通过解析 `init.zygote.rc` 文件而创建的，zygote所对应的可执行程序app_process，所对应的源文件是 `app_main.cpp` ，进程名为zygote。

```
// system/core/rootdir/init.zygote32.rc

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-
system-server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
    writepid /dev/cpuset/foreground/tasks
```

1-2.执行app_main.cpp中的main方法

启动zygote的入口函数是app_main.cpp中的main方法。

```
frameworks/base/cmds/app_process/app_main.cpp

// 186
int main(int argc, char* const argv[])

// 248 将zygote标志位置为true。
if (strcmp(arg, "--zygote") == 0) {
    zygote = true;
}

// 306 运行AndroidRuntime.cpp的start方法
if (zygote) {
    runtime.start("com.android.internal.os.ZygoteInit", args, zygote);
}
```

1-3.AndroidRuntime::start

调用startReg方法来完成jni方法的注册。

```
frameworks/base/core/jni/AndroidRuntime.cpp

// 1007
void AndroidRuntime::start(const char* className, const Vector<String8>&
options, bool zygote)

// 1051
if (startReg(env) < 0) {
```

```
frameworks/base/core/jni/AndroidRuntime.cpp

1440 int AndroidRuntime::startReg(JNIEnv* env)

// 1459 注册jni方法
if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
```

```
frameworks/base/core/jni/AndroidRuntime.cpp

// 1283
static int register_jni_procs(const RegJNIRec array[], size_t count, JNIEnv*
env)
{
    // 循环注册jni方法
    for (size_t i = 0; i < count; i++) {
        if (array[i].mProc(env) < 0) {
            return -1;
        }
    }
    return 0;
}
```

```
frameworks/base/core/jni/AndroidRuntime.cpp

// 1296
static const RegJNIRec gRegJNI[] = {

    // 1312
    REG_JNI(register_android_os_Binder),

}
```

2.register_android_os_Binder

```
frameworks/base/core/jni/android_util_Binder.cpp

// 1282
int register_android_os_Binder(JNIEnv* env)
{
    if (int_register_android_os_Binder(env) < 0)
        return -1;
    if (int_register_android_os_BinderInternal(env) < 0)
        return -1;
    if (int_register_android_os_BinderProxy(env) < 0)
        return -1;
}
```

2-1.int_register_android_os_Binder

```
frameworks/base/core/jni/android_util_Binder.cpp

// 843
static const JNINativeMethod gBinderMethods[] = {
     /* name, signature, funcPtr */
    { "getCallingPid", "()I", (void*)android_os_Binder_getCallingPid },
    { "getCallingUid", "()I", (void*)android_os_Binder_getCallingUid },
    { "clearCallingIdentity", "()J",
(void*)android_os_Binder_clearCallingIdentity },
    { "restoreCallingIdentity", "(J)V",
(void*)android_os_Binder_restoreCallingIdentity },
```

```
    { "setThreadStrictModePolicy", "(I)V",
(void*)android_os_Binder_setThreadStrictModePolicy },
    { "getThreadStrictModePolicy", "()I",
(void*)android_os_Binder_getThreadStrictModePolicy },
    { "flushPendingCommands", "()V",
(void*)android_os_Binder_flushPendingCommands },
    { "init", "()V", (void*)android_os_Binder_init },
    { "destroy", "()V", (void*)android_os_Binder_destroy },
    { "blockUntilThreadAvailable", "()V",
(void*)android_os_Binder_blockUntilThreadAvailable }
};

// 857
const char* const kBinderPathName = "android/os/Binder";

// 859
static int int_register_android_os_Binder(JNIEnv* env)
{
    // 查找文件 kBinderPathName = "android/os/Binder"，返回对应Class对象
    jclass clazz = FindClassOrDie(env, kBinderPathName);

    // 通过gBinderOffsets结构体，保存Java层Binder类的信息，为JNI层访问Java层提供通道
    gBinderOffsets.mClass = MakeGlobalRefOrDie(env, clazz);
    gBinderOffsets.mExecTransact = GetMethodIDOrDie(env, clazz, "execTransact",
"(IJJI)Z");
    gBinderOffsets.mObject = GetFieldIDOrDie(env, clazz, "mObject", "J");

    // 通过RegisterMethodsOrDie，将为gBinderMethods数组完成映射关系，从而为Java层访问
JNI层提供通道
    return RegisterMethodsOrDie(
        env, kBinderPathName,
        gBinderMethods, NELEM(gBinderMethods));
}
```

2-2.int_register_android_os_BinderInternal

```
frameworks/base/core/jni/android_util_Binder.cpp

// 925
static const JNINativeMethod gBinderInternalMethods[] = {
     /* name, signature, funcPtr */
    { "getContextObject", "()Landroid/os/IBinder;",
(void*)android_os_BinderInternal_getContextObject },
    { "joinThreadPool", "()V", (void*)android_os_BinderInternal_joinThreadPool
},
    { "disableBackgroundScheduling", "(Z)V",
(void*)android_os_BinderInternal_disableBackgroundScheduling },
    { "handleGc", "()V", (void*)android_os_BinderInternal_handleGc }
};

// 933
const char* const kBinderInternalPathName =
"com/android/internal/os/BinderInternal";

// 935
static int int_register_android_os_BinderInternal(JNIEnv* env)
```

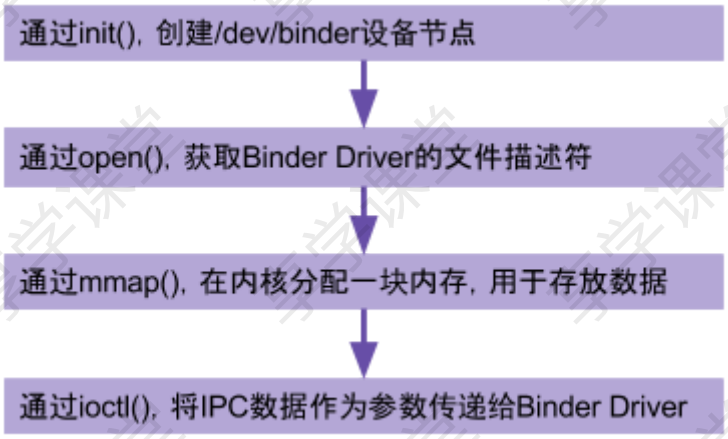
```
{
    // 查找文件kBinderInternalPathName =
"com/android/internal/os/BinderInternal"，返回Class对象
    jclass clazz = FindClassOrDie(env, kBinderInternalPathName);

    // 通过gBinderInternalOffsets，保存Java层BinderInternal类的信息，为JNI层访问java
层提供通道
    gBinderInternalOffsets.mClass = MakeGlobalRefOrDie(env, clazz);
    gBinderInternalOffsets.mForceGc = GetStaticMethodIDOrDie(env, clazz,
"forceBinderGc", "()V");

    // 通过RegisterMethodsOrDie()，将为gBinderInternalMethods数组完成映射关系，从而为
Java层访问JNI层提供通道
    return RegisterMethodsOrDie(
        env, kBinderInternalPathName,
        gBinderInternalMethods, NELEM(gBinderInternalMethods));
}
```

2-3.int_register_android_os_BinderProxy

```
frameworks/base/core/jni/android_util_Binder.cpp

// 1241
static const JNINativeMethod gBinderProxyMethods[] = {
     /* name, signature, funcPtr */
    {"pingBinder",          "()Z", (void*)android_os_BinderProxy_pingBinder},
    {"isBinderAlive",       "()Z", (void*)android_os_BinderProxy_isBinderAlive},
    {"getInterfaceDescriptor", "()Ljava/lang/String;",
(void*)android_os_BinderProxy_getInterfaceDescriptor},
    {"transactNative",      "(ILandroid/os/Parcel;Landroid/os/Parcel;I)Z",
(void*)android_os_BinderProxy_transact},
    {"linkToDeath",         "(Landroid/os/IBinder$DeathRecipient;I)V",
(void*)android_os_BinderProxy_linkToDeath},
    {"unlinkToDeath",       "(Landroid/os/IBinder$DeathRecipient;I)Z",
(void*)android_os_BinderProxy_unlinkToDeath},
    {"destroy",             "()V", (void*)android_os_BinderProxy_destroy},
};

// 1252
const char* const kBinderProxyPathName = "android/os/BinderProxy";

// 1254
static int int_register_android_os_BinderProxy(JNIEnv* env)
{
    // 查找文件 kBinderProxyPathName = "android/os/BinderProxy"，返回对应Class对象
    jclass clazz = FindClassOrDie(env, "java/lang/Error");
    gErrorOffsets.mClass = MakeGlobalRefOrDie(env, clazz);

    // 通过gBinderProxyOffsets，保存Java层BinderProxy类的信息，为JNI层访问Java提供通道
    clazz = FindClassOrDie(env, kBinderProxyPathName);
    gBinderProxyOffsets.mClass = MakeGlobalRefOrDie(env, clazz);
    gBinderProxyOffsets.mConstructor = GetMethodIDOrDie(env, clazz, "<init>", "
()V");
    gBinderProxyOffsets.mSendDeathNotice = GetStaticMethodIDOrDie(env, clazz,
"sendDeathNotice",
```

```
            "(Landroid/os/IBinder$DeathRecipient;)V");

    gBinderProxyOffsets.mObject = GetFieldIDOrDie(env, clazz, "mObject", "J");
    gBinderProxyOffsets.mSelf = GetFieldIDOrDie(env, clazz, "mSelf",

"Ljava/lang/ref/WeakReference;");
    gBinderProxyOffsets.mOrgue = GetFieldIDOrDie(env, clazz, "mOrgue", "J");

    clazz = FindClassOrDie(env, "java/lang/Class");
    gClassOffsets.mGetName = GetMethodIDOrDie(env, clazz, "getName", "
()Ljava/lang/String;");

    // 通过RegisterMethodsOrDie()，将为gBinderProxyMethods数组完成映射关系，从而为Java
层访问JNI层提供通道
    return RegisterMethodsOrDie(
        env, kBinderProxyPathName,
        gBinderProxyMethods, NELEM(gBinderProxyMethods));
}
```

binder驱动



1.binder_init

```
kernel/drivers/staging/android/binder.c

// 4290 设备驱动入口函数
device_initcall(binder_init);

// 4213
static int __init binder_init(void)

// 4220 创建名为binder的单线程的工作队列
binder_deferred_workqueue = create_singlethread_workqueue("binder");

// 4269
ret = init_binder_device(device_name);
```

```
kernel/drivers/staging/android/binder.c
```

```
// 4186
static int __init init_binder_device(const char *name)
{
    int ret;
    struct binder_device *binder_device;

    // 4191 为binder设备分配内存
    binder_device = kzalloc(sizeof(*binder_device), GFP_KERNEL);

    // 4195 初始化设备
    binder_device->miscdev.fops = &binder_fops; // 设备的文件操作结构，这是
file_operations结构
    binder_device->miscdev.minor = MISC_DYNAMIC_MINOR; //  次设备号 动态分配
    binder_device->miscdev.name = name; // 设备名,"binder"

    binder_device->context.binder_context_mgr_uid = INVALID_UID;
    binder_device->context.name = name;

    // 4202 misc驱动注册
    ret = misc_register(&binder_device->miscdev);

    // 4208 将hlist节点添加到binder_devices为表头的设备链表
    hlist_add_head(&binder_device->hlist, &binder_devices);

    return ret;
}
```

2.binder_open

```
kernel/drivers/staging/android/binder.c

// 3454
static int binder_open(struct inode *nodp, struct file *filp)

// 3462 为binder_proc结构体在kernel分配内存空间
proc = kzalloc(sizeof(*proc), GFP_KERNEL);

// 3465 将当前线程的task保存到binder进程的tsk
get_task_struct(current);
proc->tsk = current;
INIT_LIST_HEAD(&proc->todo); // 初始化todo列表
init_waitqueue_head(&proc->wait); // 初始化wait队列
proc->default_priority = task_nice(current); // 将当前进程的nice值转换为进程优先级

// 3474 同步锁，因为binder支持多线程访问
binder_lock(__func__);

binder_stats_created(BINDER_STAT_PROC); // binder_proc对象创建数加1
hlist_add_head(&proc->proc_node, &binder_procs); // 将proc_node节点添加到
binder_procs的队列头部
proc->pid = current->group_leader->pid; // 进程pid
INIT_LIST_HEAD(&proc->delivered_death); // 初始化已分发的死亡通知列表
filp->private_data = proc; // 将这个binder_proc与filp关联起来，这样下次通过filp就能找
到这个proc了
```

```
binder_unlock(__func__); // 释放同步锁
```

3.binder_mmap

```
kernel/drivers/staging/android/binder.c

// 3355
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)

// 3366 保证映射内存大小不超过4M
if ((vma->vm_end - vma->vm_start) > SZ_4M)
    vma->vm_end = vma->vm_start + SZ_4M;

// 3382 同步锁，保证一次只有一个进程分配内存，保证多进程间的并发访问
mutex_lock(&binder_mmap_lock);
// 是否已经做过映射，执行过则进入if，goto跳转，释放同步锁后结束binder_mmap方法
if (proc->buffer) {
    goto err_already_mapped;
}
// 采用 VM_IOREMAP方式，分配一个连续的内核虚拟内存，与进程虚拟内存大小一致
area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
// 内存分配不成功直接报错
if (area == NULL) {
    ret = -ENOMEM;
    failure_string = "get_vm_area";
    goto err_get_vm_area_failed;
}
// 将proc中的buffer指针指向这块内核的虚拟内存
proc->buffer = area->addr;
// 计算出用户空间和内核空间的地址偏移量。地址偏移量 = 用户虚拟内存地址 - 内核虚拟内存地址
proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;
mutex_unlock(&binder_mmap_lock); // 释放锁

// 3407 分配物理页的指针数组，数组大小为vma的等效page个数
proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start) /
PAGE_SIZE), GFP_KERNEL);

// 3418 分配物理页面，同时映射到内核空间和进程空间，先分配1个物理页。
if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE,
vma)) {
```

```
kernel/drivers/staging/android/binder.c

// 576
static int binder_update_page_range(struct binder_proc *proc, int allocate,
                    void *start, void *end,
                    struct vm_area_struct *vma)

// 609 allocate为1，代表分配内存过程。如果为0则代表释放内存过程
if (allocate == 0)
        goto free_range;

// 624 分配一个page的物理内存
*page = alloc_page(GFP_KERNEL | __GFP_HIGHMEM | __GFP_ZERO);
```

```
// 630 物理空间映射到虚拟内核空间
ret = map_kernel_range_noflush((unsigned long)page_addr,
                    PAGE_SIZE, PAGE_KERNEL, page);

// 641 物理空间映射到虚拟进程空间
ret = vm_insert_page(vma, user_page_addr, page[0]);
```

```
kernel/drivers/staging/android/binder.c

// 3355
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)

// 3425
list_add(&buffer->entry, &proc->buffers);// 将buffer连入buffers链表中
buffer->free = 1; // 此内存可用
binder_insert_free_buffer(proc, buffer);// 将buffer插入 proc->free_buffers 链表中
proc->free_async_space = proc->buffer_size / 2; // 异步的可用空闲空间大小
barrier();
proc->files = get_files_struct(current);
proc->vma = vma;
proc->vma_vm_mm = vma->vm_mm;
```

3-1.binder_insert_free_buffer

```
kernel/drivers/staging/android/binder.c

// 494
static void binder_insert_free_buffer(struct binder_proc *proc,
                      struct binder_buffer *new_buffer)

// 511
while (*p) {
    parent = *p;
    buffer = rb_entry(parent, struct binder_buffer, rb_node);

    // 计算得出空闲内存的大小
    buffer_size = binder_buffer_size(proc, buffer);

    if (new_buffer_size < buffer_size)
        p = &parent->rb_left;
    else
        p = &parent->rb_right;
}
rb_link_node(&new_buffer->rb_node, parent, p);
// 将 buffer插入 proc->free_buffers 链表中
rb_insert_color(&new_buffer->rb_node, &proc->free_buffers);
```

4.binder_ioctl

```
kernel/drivers/staging/android/binder.c

// 3241
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
```

```
// 3254 进入休眠状态，直到中断唤醒
ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error
< 2);

// 3259 根据当前进程的pid，从binder_proc中查找binder_thread,
//       如果当前线程已经加入到proc的线程队列则直接返回，
//       如果不存在则创建binder_thread，并将当前线程添加到当前的proc
thread = binder_get_thread(proc);

// 3265 进行binder的读写操作
switch (cmd) {
    case BINDER_WRITE_READ:
        ret = binder_ioctl_write_read(filp, cmd, arg, thread);
        if (ret)
```

4-1.binder_ioctl_write_read

```
kernel/drivers/staging/android/binder.c

// 3136
static int binder_ioctl_write_read(struct file *filp,
                unsigned int cmd, unsigned long arg,
                struct binder_thread *thread)

// 3150 把用户空间数据ubuf拷贝到bwr
if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {

// 3160
if (bwr.write_size > 0) { // 当写缓存中有数据，则执行binder写操作
    ret = binder_thread_write(proc, thread,
                bwr.write_buffer,
                bwr.write_size,
                &bwr.write_consumed);
}
if (bwr.read_size > 0) { // 当读缓存中有数据，则执行binder读操作
    ret = binder_thread_read(proc, thread, bwr.read_buffer,
                bwr.read_size,
                &bwr.read_consumed,
                filp->f_flags & O_NONBLOCK);
    // 进程todo队列不为空,则唤醒该队列中的线程
    if (!list_empty(&proc->todo))
        wake_up_interruptible(&proc->wait);
}

// 3192 把内核空间数据bwr拷贝到ubuf
if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
```

数据结构

file_operations

```
static const struct file_operations binder_fops = {
    .owner = THIS_MODULE,
    .poll = binder_poll,
    .unlocked_ioctl = binder_ioctl,
    .compat_ioctl = binder_ioctl,
    .mmap = binder_mmap,
    .open = binder_open,
    .flush = binder_flush,
    .release = binder_release,
};
```

binder_proc

每个进程调用open()打开binder驱动都会创建该结构体，用于管理IPC所需的各种信息。

```
struct binder_proc {
    struct hlist_node proc_node; // 进程节点
    struct rb_root threads; // binder_thread红黑树的根节点
    struct rb_root nodes; // binder_node红黑树的根节点
    struct rb_root refs_by_desc; // binder_ref红黑树的根节点(以 handle为 key)
    struct rb_root refs_by_node; // binder_ref红黑树的根节点（以 ptr为 key）
    int pid; // 相应进程 id
    struct vm_area_struct *vma; // 指向进程虚拟地址空间的指针
    struct mm_struct *vma_vm_mm; // 相应进程的内存结构体
    struct task_struct *tsk; //      相应进程的 task结构体
    struct files_struct *files; // 相应进程的文件结构体
    struct hlist_node deferred_work_node;
    int deferred_work;
    void *buffer; // 内核空间的起始地址
    ptrdiff_t user_buffer_offset; // 内核空间与用户空间的地址偏移量

    struct list_head buffers; // 所有的 buffer
    struct rb_root free_buffers; // 空闲的 buffer
    struct rb_root allocated_buffers; // 已分配的 buffer
    size_t free_async_space; // 异步的可用空闲空间大小

    struct page **pages; // 指向物理内存页指针的指针
    size_t buffer_size; // 映射的内核空间大小
    uint32_t buffer_free; // 可用内存总大小
    struct list_head todo; // 进程将要做的事
    wait_queue_head_t wait; // 等待队列
    struct binder_stats stats; // binder统计信息
    struct list_head delivered_death; // 已分发的死亡通知
    int max_threads; // 最大线程数
    int requested_threads; // 请求的线程数
    int requested_threads_started; // 已启动的请求线程数
    int ready_threads; // 准备就绪的线程个数
    long default_priority; // 默认优先级
    struct dentry *debugfs_entry;
    struct binder_context *context;
};
```

binder_node

```
struct binder_node {
    int debug_id; // 节点创建时分配，具有全局唯一性，用于调试使用
    struct binder_work work;
    union {
        struct rb_node rb_node; // binder节点正常使用，union
        struct hlist_node dead_node; // binder节点已销毁，union
    };
    struct binder_proc *proc; // binder所在的进程，见后面小节
    struct hlist_head refs; // 所有指向该节点的 binder引用队列
    int internal_strong_refs;
    int local_weak_refs;
    int local_strong_refs;
    binder_uintptr_t ptr; // 指向用户空间 binder_node的指针,对应于
flat_binder_object.binder
    binder_uintptr_t cookie; // 指向用户空间 binder_node的指针，附件数据,对应于
flat_binder_object.cookie
    unsigned has_strong_ref:1; // 占位 1bit
    unsigned pending_strong_ref:1; // 占位 1bit
    unsigned has_weak_ref:1; // 占位 1bit
    unsigned pending_weak_ref:1; // 占位 1bit
    unsigned has_async_transaction:1; // 占位 1bit
    unsigned accept_fds:1; // 占位 1bit
    unsigned min_priority:8; // 占位 8bit，最小优先级
    struct list_head async_todo; // 异步todo队列
};
```

binder_buffer

```
struct binder_buffer {
    struct list_head entry; // buffer实体的地址
    struct rb_node rb_node; // buffer实体的地址
                /* by address */
    unsigned free:1; // 标记是否是空闲buffer，占位1bit
    unsigned allow_user_free:1; // 是否允许用户释放，占位1bit
    unsigned async_transaction:1; // 占位1bit
    unsigned debug_id:29; // 占位29bit

    struct binder_transaction *transaction; // 该缓存区的需要处理的事务

    struct binder_node *target_node; // 该缓存区所需处理的Binder实体
    size_t data_size; // 数据大小
    size_t offsets_size; // 数据偏移量
    size_t extra_buffers_size;
    uint8_t data[0]; // 数据地址
};
```