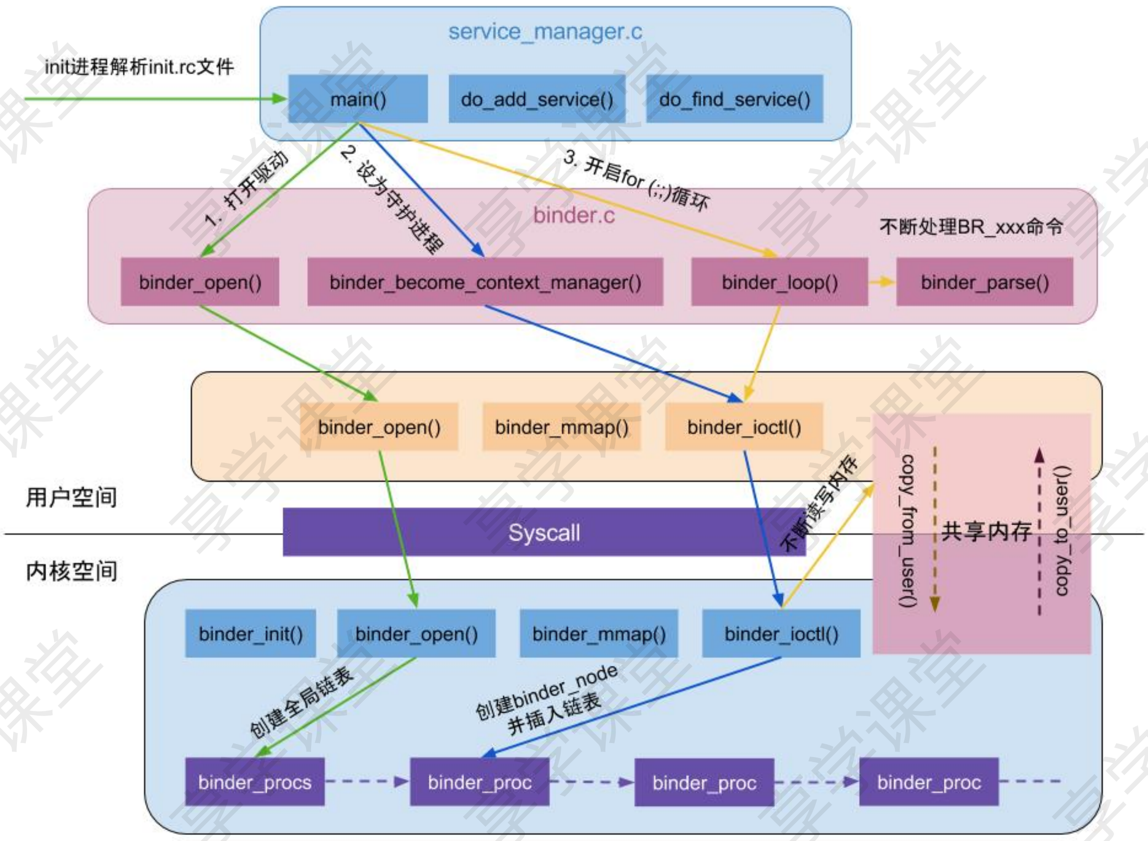
启动service_manager



1.启动servicemanager进程

ServiceManager是由init进程通过解析init.rc文件而创建的，其所对应的可执行程序servicemanager，所对应的源文件是service_manager.c，进程名为servicemanager。

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

// 602
service servicemanager /system/bin/servicemanager
    class core
    user system
    group system
    critical
    onrestart restart healthd
    onrestart restart zygote
    onrestart restart media
    onrestart restart surfaceflinger
    onrestart restart drm
```

2.main

启动ServiceManager的入口函数是`service_manager.c`中的main()方法。

```
frameworks/native/cmds/servicemanager/service_manager.c

// 354
int main(int argc, char **argv)

// 358 打开 binder驱动，申请 128k字节大小的内存空间---见后面小节
bs = binder_open(128*1024);

// 364 设为守护进程，成为 binder大管理者---见后面小节
if (binder_become_context_manager(bs)) {

// 391 进入无限循环，处理client端发来的请求---见后面小节
binder_loop(bs, svcmgr_handler);
```

2-1.binder_open

```
frameworks/native/cmds/servicemanager/binder.c

// 96
struct binder_state *binder_open(size_t mapsize)

// 98 这个结构体记录了 service_manager 中有关于 binder 的所有信息
struct binder_state *bs;

// 107 打开 binder驱动，得到文件描述符
bs->fd = open("/dev/binder", O_RDWR);

// 123
bs->mapsize = mapsize; // service_manager自己设置的，大小为 128kb
/*通过系统调用，mmap内存映射，mmap必须是 page的整数倍(即 4kb的整数倍)*/
bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
```

2-2.binder_become_context_manager

```
frameworks/native/cmds/servicemanager/binder.c

// 146
int binder_become_context_manager(struct binder_state *bs)
{
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

2-2-1.binder_ioctl

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

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

// 3277
case BINDER_SET_CONTEXT_MGR:
    ret = binder_ioctl_set_ctx_mgr(filp);
```

2-2-2.binder_ioctl_set_ctx_mgr

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

// 3200
static int binder_ioctl_set_ctx_mgr(struct file *filp)

// 3208 保证只创建一次 mgr_node对象，不为 null就直接返回
if (context->binder_context_mgr_node) {

// 3216
/* uid是否有效，当前是无效的 */
if (uid_valid(context->binder_context_mgr_uid)) {

} else {
    /* 设置当前线程 euid作为 service_manager的 uid */
    context->binder_context_mgr_uid = curr_euid;
}
// 创建 service_manager实体
context->binder_context_mgr_node = binder_new_node(proc, 0, 0);

// 3233 将 binder_context_mgr_node的强弱引用各加 1
context->binder_context_mgr_node->local_weak_refs++;
context->binder_context_mgr_node->local_strong_refs++;
context->binder_context_mgr_node->has_strong_ref = 1;
context->binder_context_mgr_node->has_weak_ref = 1;
```

2-2-2-1.binder_new_node

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

// 923
static struct binder_node *binder_new_node(struct binder_proc *proc,
                       binder_uintptr_t ptr,
                       binder_uintptr_t cookie)

// 931 首次进来为空
while (*p) {

// 943 给新创建的binder_node 分配内核空间
node = kzalloc(sizeof(*node), GFP_KERNEL);

// 947 将新创建的 node对象添加到 proc红黑树
rb_link_node(&node->rb_node, parent, p);
rb_insert_color(&node->rb_node, &proc->nodes);

// 950 初始化 binder_node
node->proc = proc;
node->ptr = ptr;
node->cookie = cookie;
node->work.type = BINDER_WORK_NODE; // 设置 binder_work的 type
INIT_LIST_HEAD(&node->work.entry);
INIT_LIST_HEAD(&node->async_todo);
```

2-3.binder_loop

```
frameworks/native/cmds/servicemanager/binder.c
```

```
// 372
void binder_loop(struct binder_state *bs, binder_handler func)

// 378
bwr.write_size = 0; // 初始化为 0
bwr.write_consumed = 0;
bwr.write_buffer = 0;

readbuf[0] = BC_ENTER_LOOPER; // 读写要处理的命令
binder_write(bs, readbuf, sizeof(uint32_t)); // 设置线程的 looper状态为循环状态

for (;;) {
    bwr.read_size = sizeof(readbuf); // 不为 0，进入 binder_thread_read
    bwr.read_consumed = 0;
    bwr.read_buffer = (uintptr_t) readbuf;

    /* 不断地 binder读数据，没有数据会进入休眠状态 */
    res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
```

2-3-1.binder_write

```
frameworks/native/cmds/servicemanager/binder.c

// 151
int binder_write(struct binder_state *bs, void *data, size_t len)

// 156
bwr.write_size = len; // 大于 0，进入 binder_thread_write
bwr.write_consumed = 0;
bwr.write_buffer = (uintptr_t) data; // 此处 data为 BC_ENTER_LOOPER
bwr.read_size = 0; // read 不会进去
bwr.read_consumed = 0;
bwr.read_buffer = 0;
/* 设置线程的 looper状态为循环状态 */
res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
```

2-3-2.binder_thread_write

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

// 2250
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)

// 2262 获取命令，即 BC_ENTER_LOOPER
if (get_user(cmd, (uint32_t __user *)ptr))

// 2472
case BC_ENTER_LOOPER:

// 2481 设置该线程的 looper状态
thread->looper |= BINDER_LOOPER_STATE_ENTERED;
```

2-3-3.binder_thread_read

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

// 2652
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)

// 2664 设置命令为 BR_NOOP
if (*consumed == 0) {
    if (put_user(BR_NOOP, (uint32_t __user *)ptr))

// 2671 wait_for_proc_work 为 true
wait_for_proc_work = thread->transaction_stack == NULL &&
            list_empty(&thread->todo);

// 2694 准备就绪的线程个数加 1
if (wait_for_proc_work)
    proc->ready_threads++;

// 2702
if (wait_for_proc_work) {
    if (non_block) { // 非阻塞操作，service_manager是阻塞的，所以 if不命中
    } else // 进入 else，开始等待
        ret = wait_event_freezable_exclusive(proc->wait,
binder_has_proc_work(proc, thread));
}
```

获取service_manager

获取Service Manager是通过defaultServiceManager()方法来完成。

1.defaultServiceManager

```
frameworks/native/libs/binder/IServiceManager.cpp

// 33
sp<IServiceManager> defaultServiceManager()
{
    /* 单例模式，如果不为空直接返回 */
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

    {
        AutoMutex _l(gDefaultServiceManagerLock);
        /* 创建或者获取 SM时，SM可能未准备就绪，这时通过 sleep 1秒后，循环尝试获取直到成功
*/
        while (gDefaultServiceManager == NULL) {
            /* 分为三块分析 */
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
            if (gDefaultServiceManager == NULL)
                sleep(1);
        }
    }

    return gDefaultServiceManager;
}
```

1-1.ProcessState::self

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

// 70
sp<ProcessState> ProcessState::self()
{
    /* 单例模式 */
    if (gProcess != NULL) {
        return gProcess;
    }
    gProcess = new ProcessState; // 实例化 ProcessState
    return gProcess;
}
```

1-1-1.ProcessState::ProcessState

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

// 339
ProcessState::ProcessState()
    : mDriverFD(open_driver())

// 358 采用内存映射函数 mmap，给 binder分配一块大小为 (1M-8K)的虚拟地址空间,用来接收事务
mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE,
mDriverFD, 0);
```

1-1-1-1.open_driver

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

// 311
static int open_driver()

// 313 打开 /dev/binder设备，建立与内核的 Binder驱动的交互通道
int fd = open("/dev/binder", O_RDWR);

// 328 通过 ioctl设置 binder驱动，能支持的最大线程数
size_t maxThreads = DEFAULT_MAX_BINDER_THREADS;
result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
```

1-2.ProcessState::getContextObject

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

// 85
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& /*caller*/)
{
    // 参数为0，获取service_manager服务
    return getStrongProxyForHandle(0);
}
```

1-2-1.ProcessState::getStrongProxyForHandle

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

// 179
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)

// 查找 handle对应的资源项
handle_entry* e = lookupHandleLocked(handle);

// 192 当handle值所对应的IBinder不存在或弱引用无效时
if (b == NULL || !e->refs->attemptIncWeak(this)) {

// 214 通过ping操作测试binder是否准备就绪
status_t status = IPCThreadState::self()->transact(
        0, IBinder::PING_TRANSACTION, data, NULL, 0);

// 220 创建BpBinder对象
b = new BpBinder(handle);
```

1-2-2.BpBinder::BpBinder

```
frameworks/native/libs/binder/BpBinder.cpp

// 89
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
{
    /* 支持强弱引用计数，OBJECT_LIFETIME_WEAK表示目标对象的生命周期受弱指针控制 */
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);
    /* handle所对应的 bindle弱引用 + 1 */
    IPCThreadState::self()->incWeakHandle(handle);
}
```

1-3.interface_cast

```
frameworks/native/include/binder/IInterface.h

// 41
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    // 等价于：IServiceManager::asInterface
    return INTERFACE::asInterface(obj);
}
```

1-3-1.IServiceManager::asInterface

对于asInterface()函数，通过搜索代码，你会发现根本找不到这个方法是在哪里定义这个函数的, 其实是通过模板函数来定义的。

```
frameworks/native/include/binder/IInterface.h

// 74
#define DECLARE_META_INTERFACE(INTERFACE)                               \
    static const android::String16 descriptor;                          \
    static android::sp<I##INTERFACE> asInterface(                       \
            const android::sp<android::IBinder>& obj);                  \
    virtual const android::String16& getInterfaceDescriptor() const;    \
    I##INTERFACE();                                                     \
    virtual ~I##INTERFACE();                                            \

// 83
#define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)                       \
    const android::String16 I##INTERFACE::descriptor(NAME);             \
    const android::String16&                                            \
            I##INTERFACE::getInterfaceDescriptor() const {              \
        return I##INTERFACE::descriptor;                                \
    }                                                                   \
    android::sp<I##INTERFACE> I##INTERFACE::asInterface(                \
            const android::sp<android::IBinder>& obj)                   \
    {                                                                   \
        android::sp<I##INTERFACE> intr;                                 \
        if (obj != NULL) {                                              \
            intr = static_cast<I##INTERFACE*>(                          \
                obj->queryLocalInterface(                               \
                        I##INTERFACE::descriptor).get());               \
            if (intr == NULL) {                                         \
                intr = new Bp##INTERFACE(obj);                          \
            }                                                           \
        }                                                               \
        return intr;                                                    \
    }                                                                   \
    I##INTERFACE::I##INTERFACE() { }                                    \
    I##INTERFACE::~I##INTERFACE() { }                                   \
```

1-3-2.DECLARE_META_INTERFACE

```
frameworks/native/include/binder/IServiceManager.h

// 33
DECLARE_META_INTERFACE(ServiceManager)
```

展开即可得:

```
static const android::String16 descriptor;

static android::sp< IServiceManager > asInterface(const
android::sp<android::IBinder>& obj)

virtual const android::String16& getInterfaceDescriptor() const;

IServiceManager ();
virtual ~IServiceManager();
```

该过程主要是声明asInterface(),getInterfaceDescriptor()方法。

1-3-3.IMPLEMENT_META_INTERFACE

```
frameworks/native/libs/binder/IServiceManager.cpp

// 185
IMPLEMENT_META_INTERFACE(ServiceManager,"android.os.IServiceManager")
```

展开即可得:

```
const android::String16
IServiceManager::descriptor(“android.os.IServiceManager”);

const android::String16& IServiceManager::getInterfaceDescriptor() const
{
     return IServiceManager::descriptor;
}

 android::sp<IServiceManager> IServiceManager::asInterface(const
android::sp<android::IBinder>& obj)
{
       android::sp<IServiceManager> intr;
        if(obj != NULL) {
           intr = static_cast<IServiceManager *>(
               obj->queryLocalInterface(IServiceManager::descriptor).get());
           if (intr == NULL) {
                // 等价于 new BpServiceManager(BpBinder)
                intr = new BpServiceManager(obj);
            }
        }
       return intr;
}

IServiceManager::IServiceManager () { }
IServiceManager::~ IServiceManager() { }
```

1-4.BpServiceManager

```
frameworks/native/libs/binder/IServiceManager.cpp

// 126
class BpServiceManager : public BpInterface<IServiceManager>

// 129
BpServiceManager(const sp<IBinder>& impl)
    : BpInterface<IServiceManager>(impl)
```

1-4-1.BpInterface::BpInterface

```
frameworks/native/include/binder/IInterface.h

// 134
template<typename INTERFACE>
inline BpInterface<INTERFACE>::BpInterface(const sp<IBinder>& remote)
    : BpRefBase(remote)
```

1-4-2.BpRefBase

```
frameworks/native/libs/binder/Binder.cpp

// 241 mRemote指向 new BpBinder(0)，从而 BpServiceManager能够利用 Binder进行通过通信
BpRefBase::BpRefBase(const sp<IBinder>& o)
    : mRemote(o.get()), mRefs(NULL), mState(0)
```