**LINUX内核中的Binder处理**

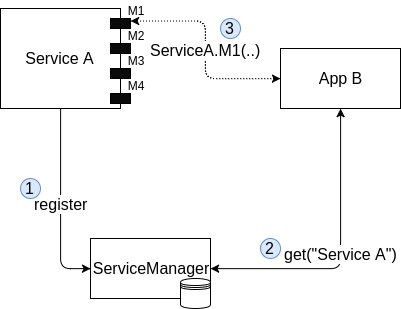
Binder是Android中主要的IPC / RPC（进程间通信）系统。它允许应用程序彼此通信，它是Android环境中几种重要机制的基础。例如，Android服务是建立在Binder之上的。与Binder交换的消息称为Binder处理(transactions)，它们可以传输简单数据（例如整数），还可以处理更复杂的结构，例如文件描述符，内存缓冲区或对象上的弱/强引用。Internet上有很多有趣的Binder文档，但是关于如何将消息从一个进程转换到另一个进程的细节很少。本文试图描述Binder如何处理消息以及如何在不同进程之间执行复杂对象（文件描述符，指针）的转换。这里将跟踪一个Binder调用，从用户区到Binder内核。

**用户区的Binder**

在探索Binder内核模块如何工作之前，让我们看一下在调用Android Service的情况下如何在用户区中准备调用。

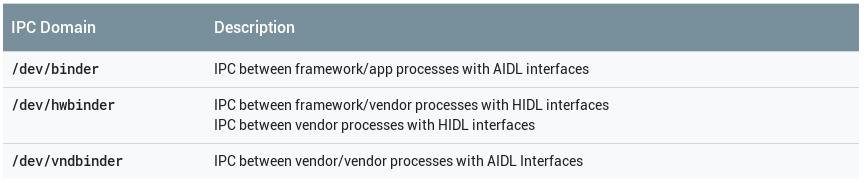
**Android服务概述**

服务是在后台运行并为其他应用程序提供功能的Android组件。其中一些是Android框架的一部分，但已安装的应用程序也可以公开其自身的功能。当应用程序要公开新服务时，它首先注册到“Service Manager” （1）其中包含并更新所有正在运行的服务的列表。稍后，客户端向处理程序请求Service Manager （2）与该服务进行通信，获得该服务之后就能够调用公开的函数（3）。



服务交互

从Android 8.0开始，存在三个不同的Binder域。每个域都有其自己的Service Manager，它们访问/dev/下对应的设备。下表描述了每个Binder域对应的设备：

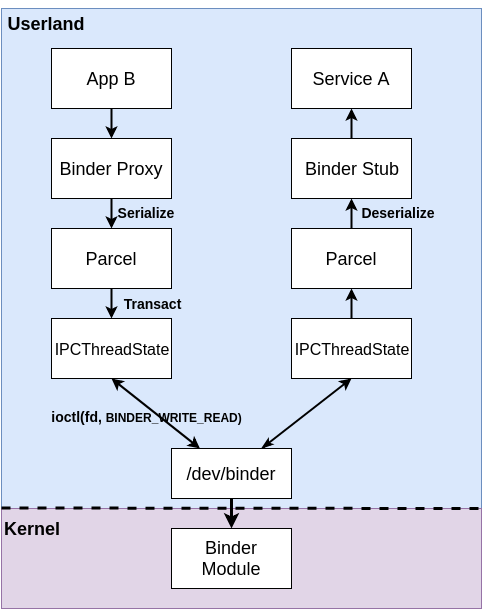


Binder域，摘自Android文档网站

为了使用Binder系统，一个进程需要打开这些设备之一并执行一些初始化步骤，然后再发送或接收Binder调用。

**准备Binder调用**

Android框架在Binder设备的上面包含多个抽象层。通常，当开发人员实施新服务时，他们会描述以高级语言公开的接口。在框架应用程序的情况下，描述是使用AIDL语言编写的，而由供应商开发的硬件服务则具有以HIDL语言编写的接口描述。这些描述被编译到Java / C ++文件中，在其中使用Parcel组件对参数进行反序列化。生成的代码包含两个类，一个Binder Proxy（代理）和一个Binder Stub（存根）。代理类用于请求远程服务，而存根则用于接收传入呼叫，如下图所示：



Binder Layers

在最低级别，使用域对应的设备（比如aidl对应/dev/Binder/）将应用程序连接到Binder内核模块。他们使用ioctl syscall发送和接收Binder消息。

序列化步骤使用Parcel类完成，该类提供了在Binder消息中读取和写入数据的功能。有两种不同的Parcel类：

1、在/dev/binder和/dev/vndbinder域是基于AIDL描述语言，使用的Parcel定义在frameworks/native/include/binder/Parcel.h。这个Parcel允许发送基本类型和文件描述符。例如，以下代码摘自SHELL\_COMMAND\_TRANSACTION命令的默认代理实现。该命令准备并写入远程服务使用的标准输入，输出和错误流的文件描述符。

// Extract from frameworks/base/core/java/Android/os/Binder.java

public void shellCommand(FileDescriptor in, FileDescriptor out, FileDescriptor err,

String[] args, ShellCallback callback,

ResultReceiver resultReceiver) throws RemoteException {

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeFileDescriptor(in);

data.writeFileDescriptor(out);

data.writeFileDescriptor(err);

data.writeStringArray(args);

ShellCallback.writeToParcel(callback, data);

resultReceiver.writeToParcel(data, 0);

try {

transact(SHELL\_COMMAND\_TRANSACTION, data, reply, 0);

reply.readException();

} finally {

data.recycle();

reply.recycle();

}

}

1. 在/dev/hwbinder域，使用另一个Parcel实现，它是基于上面那个AIDL Parcel的，头文件为system/libhwbinder/include/hwbinder/Parcel.h。这个Parcel实现允许发送数据缓冲区，例如C结构。数据缓冲区可以嵌套，并包含指向其他结构的指针。在以下示例中，结构hild\_memory包含一个嵌入式结构（hild\_string）和一个内存指针（mHandle）：

// Extract from system/libhidl/transport/include/hidl/HidlBinderSupport.h

// ---------------------- hidl\_memory

status\_t readEmbeddedFromParcel(const hidl\_memory &memory, const Parcel &parcel, size\_t parentHandle, size\_t parentOffset);

status\_t writeEmbeddedToParcel(const hidl\_memory &memory, Parcel \*parcel, size\_t parentHandle, size\_t parentOffset);

// [...]

// Extract from system/libhidl/base/include/hidl/HidlSupport.h

struct hidl\_memory {

// ...

private:

hidl\_handle mHandle \_\_attribute\_\_ ((aligned(8)));

uint64\_t mSize \_\_attribute\_\_ ((aligned(8)));

hidl\_string mName \_\_attribute\_\_ ((aligned(8)));

};

这两种Parcel能够发送文件描述符和带有内存地址的复杂数据结构。因为这些元素包含特定于调用者进程的数据，所以Parcel组件将Binder对象写入调用消息中。

**Binder对象**

除了简单类型（字符串，整数等）之外，还可以发送Binder对象。Binder对象是一种类型值为以下之一的结构：

// Extract from : drivers/staging/Android/uapi/binder.h

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),

//下面两种在HIDL里才有

BINDER\_TYPE\_FDA = B\_PACK\_CHARS('f', 'd', 'a', B\_TYPE\_LARGE),

BINDER\_TYPE\_PTR = B\_PACK\_CHARS('p', 't', '\*', B\_TYPE\_LARGE),

};

下面是类型为BINDER\_TYPE\_PTR的Binder对象示例：

struct binder\_object\_header {

\_\_u32 type;

};

struct binder\_buffer\_object {

struct binder\_object\_header hdr;

\_\_u32 flags;

binder\_uintptr\_t buffer;

binder\_size\_t length;

binder\_size\_t parent;

binder\_size\_t parent\_offset;

};

下方的属性hdr就是用来指定类型的。

不同的Binder对象可以描述如下：

BINDER\_TYPE\_BINDER和BINDER\_TYPE\_WEAK\_BINDER：这些类型是对本地对象的强引用和弱引用。

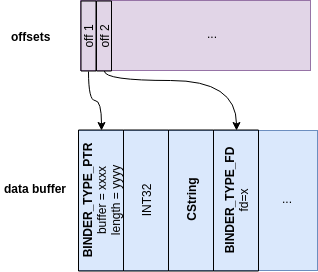
BINDER\_TYPE\_HANDLER和BINDER\_TYPE\_WEAK\_HANDLE：这些类型是对远程对象的强引用和弱引用。

BINDER\_TYPE\_FD ：此类型用于发送文件描述符号。这通常用于发送ashmem共享内存以传输大量数据。实际上，Binder传输消息被限制为1 MB。但是，可以使用任何文件描述符类型（文件，Socket，标准输入等）。

BINDER\_TYPE\_FDA：描述文件描述符数组的对象。

BINDER\_TYPE\_PTR：用于使用内存地址及其大小发送缓冲区的对象。

当Parcel类写入缓冲区或文件描述符时，它将在数据缓冲区中添加Binder对象（图上为蓝色）。Binder对象和简单类型混合在数据缓冲区中。每次写入对象时，其相对位置都会插入到偏移缓冲区中（紫色）。



Binder消息缓冲区和偏移量

一旦data和offets缓冲区被填充好，一个binder\_transaction\_data就准备将传递给内核。我们可以注意到它包含上述指针，数据缓冲区和偏移量数组的大小。字段handler用于设置先前由Service Manager检索出来的目标进程。另一个有趣的属性是code，它包含要执行的远程服务的方法ID。

// file : development/ndk/platforms/android-9/include/linux/binder.h

struct binder\_transaction\_data {

union { //这个用来表示传输的目标，本地是ptr，远程是handle

size\_t handle;

void \*ptr;

} target;

void \*cookie;

unsigned int code; //这个是远程服务方法ID

unsigned int flags;

pid\_t sender\_pid;

uid\_t sender\_euid;

size\_t data\_size;

size\_t offsets\_size;

union { //这个表示上面说的parcle buffer和offset

struct {

const void \*buffer;

const void \*offsets;

} ptr;

uint8\_t buf[8];

} data;

};

在调用ioctl之前，必须填充最后一个结构（binder\_write\_read），它包含读写命令缓冲区，并指向上一个缓冲区：

// file : development/ndk/platforms/android-9/include/linux/binder.h

struct binder\_write\_read {

signed long write\_size;

signed long write\_consumed;

unsigned long write\_buffer;

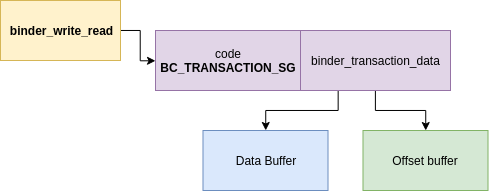
signed long read\_size;

signed long read\_consumed;

unsigned long read\_buffer;

};

发Binder调用所需的数据结构可以用下面的图总结：



binding\_write\_read结构

我们可以注意到，write\_buffer并没有直接指向binder\_transaction\_data结构。它以命令标识符为前缀。如果是调用，其值为BC\_TRANSACTION\_SG。

需要注意的是，除了BC\_TRANSACTION\_SG还存在很多命令，例如BC\_ACQUIRE和BC\_RELEASE是增加或减少一个强句柄引用，又或者BC\_REQUEST\_DEATH\_NOTIFICATION是提示远程服务被停止。

现在所有执行Binder调用的准备都做好了，调用者需要调用ioctl的BINDER\_WRITE\_READ命令，内核模块将处理该消息并为目标进程转换所有的Binder对象：强/弱句柄，文件描述符和缓冲区。

在下一部分中，让我们继续在内核方面进行分析！

**Binder内核模块**

现在，调用者进程已准备好其数据并执行了一个ioctl来发送调用。所有Binder对象都将被翻译，并且消息将被复制到目标内存中。

函数binder\_ioctl\_write\_read用于执行ioctl命令，它会执行安全的数据参数复制。

// file : drivers/android/binder.c

static long binder\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)

{

// [...]

switch (cmd) {

case BINDER\_WRITE\_READ:

ret = binder\_ioctl\_write\_read(filp, cmd, arg, thread);

if (ret)

goto err;

break;

// file : drivers/android/binder.c

static int binder\_ioctl\_write\_read(struct file \*filp,

unsigned int cmd, unsigned long arg,

struct binder\_thread \*thread)

{

// [...]

if (copy\_from\_user(&bwr, ubuf, sizeof(bwr))) {

ret = -EFAULT;

goto out;

}

// [...]

if (bwr.write\_size > 0) {

ret = binder\_thread\_write(proc, thread,

bwr.write\_buffer,

bwr.write\_size,

&bwr.write\_consumed);

在写调用的情况下，函数binder\_thread\_write被调用，然后根据命令码调用相应的处理程序。

// file : drivers/android/binder.c

switch (cmd) {

case BC\_INCREFS:

case BC\_ACQUIRE:

case BC\_RELEASE:

case BC\_DECREFS:

// [...]

case BC\_TRANSACTION\_SG:

case BC\_REPLY\_SG: {

struct binder\_transaction\_data\_sg tr;

if (copy\_from\_user(&tr, ptr, sizeof(tr)))

return -EFAULT;

ptr += sizeof(tr);

binder\_transaction(proc, thread, &tr.transaction\_data,

cmd == BC\_REPLY\_SG, tr.buffers\_size);

break;

}

// [...]

对于命令BC\_TRANSACTION\_SG，在用户区中准备好的binder\_transaction\_data缓冲，会被binder\_transaction函数处理。

**BINDER TRANSACTION**

binder\_transaction函数位于文件drivers/staging/Android/binder.c中。

这个重要的函数执行以下任务：在目标进程中（在Binder保留的内存中）分配一个缓冲区，验证所有数据对象并执行转换，在目标内存进程中复制数据和偏移缓冲区。

为了验证Binder对象，内核查看offsets缓冲区中所有包括相对位置的对象。取决于对象类型，内核执行不同的转换。

// file : drivers/android/binder.c

static void binder\_transaction(struct binder\_proc \*proc,

struct binder\_thread \*thread,

struct binder\_transaction\_data \*tr, int reply,

binder\_size\_t extra\_buffers\_size){

// [...]

// Object validation in binder\_transaction function.

// offp is a pointer to the offsets buffer

for (; offp < off\_end; offp++) {

struct binder\_object\_header \*hdr;

size\_t object\_size = binder\_validate\_object(t->buffer, \*offp);

if (object\_size == 0 || \*offp < off\_min) {

binder\_user\_error("%d:%d got transaction with invalid offset (%lld, min %lld max %lld) or object.\n",

proc->pid, thread->pid, (u64)\*offp,

(u64)off\_min,

(u64)t->buffer->data\_size);

return\_error = BR\_FAILED\_REPLY;

return\_error\_param = -EINVAL;

return\_error\_line = \_\_LINE\_\_;

goto err\_bad\_offset;

}

hdr = (struct binder\_object\_header \*)(t->buffer->data + \*offp);

off\_min = \*offp + object\_size;

switch (hdr->type) {

case BINDER\_TYPE\_BINDER:

case BINDER\_TYPE\_WEAK\_BINDER: {

// [..] Validation and Translation

case BINDER\_TYPE\_HANDLE:

case BINDER\_TYPE\_WEAK\_HANDLE: {

// [..] Validation and Translation

}

case BINDER\_TYPE\_FD:{

// [..] Validation and Translation

}

case BINDER\_TYPE\_FDA:{

// [..] Validation and Translation

}

case BINDER\_TYPE\_PTR: {

// [..] Validation and Translation

}

**Weak/Strong Binder/Handler**

Binder对象引用可以是指向本地对象的虚拟内存地址（Binder引用），也可以是标识另一个进程的远程对象（Handler引用）。

当内核获取对象引用（本地或远程）时，它将更新内部表，该表为每个进程提供一个真实虚拟内存地址和handlers之间的映射（binder <=>handler）。

这里有两种转换：

将虚拟内存地址转换为handler： binder\_translate\_binder

将handler转换为虚拟内存地址： binder\_translate\_handle

Binder内核模块保留共享对象的引用计数。与新进程共享引用时，其计数器值将增加。当不再使用引用时，将通知所有者释放它。

**binder -> handler translation**

// file : drivers/android/binder.c

static int binder\_translate\_binder(struct flat\_binder\_object \*fp,

struct binder\_transaction \*t,

struct binder\_thread \*thread)

{

// [...]

node = binder\_get\_node(proc, fp->binder); //先去node表取一下

if (!node) { //取不到，就建一个new node

node = binder\_new\_node(proc, fp);

if (!node)

return -ENOMEM;

}

if (fp->cookie != node->cookie) {

// [...] ERROR

}

// SELinux check

if (security\_binder\_transfer\_binder(proc->tsk, target\_proc->tsk)) {

// [...] ERROR

}

ret = binder\_inc\_ref\_for\_node(target\_proc, node, //增加引用计数

fp->hdr.type == BINDER\_TYPE\_BINDER,

&thread->todo, &rdata);

if (ret)

goto done;

if (fp->hdr.type == BINDER\_TYPE\_BINDER) //binder 转成 handle

fp->hdr.type = BINDER\_TYPE\_HANDLE;

else

fp->hdr.type = BINDER\_TYPE\_WEAK\_HANDLE;

fp->binder = 0;

fp->handle = rdata.desc;

fp->cookie = 0;

// [..]

}

该函数获取与binder值（虚拟地址）对应的节点，或者如果不存在该节点，则创建一个新节点。该节点在本地对象和远程对象（rdata.desc）之间具有关联。在SELinux安全检查之后，引用计数器将增加，并且binder对象中的引用值将更改，并由引用handler替换。

**handler -> binder translation**

// file : drivers/android/binder.c

static int binder\_translate\_handle(struct flat\_binder\_object \*fp,

struct binder\_transaction \*t,

struct binder\_thread \*thread)

{

// [...]

node = binder\_get\_node\_from\_ref(proc, fp->handle,

fp->hdr.type == BINDER\_TYPE\_HANDLE, &src\_rdata);

if (!node) {

// [...] Error

}

// SELinux security check

if (security\_binder\_transfer\_binder(proc->tsk, target\_proc->tsk)) {

ret = -EPERM;

goto done;

}

binder\_node\_lock(node);

if (node->proc == target\_proc) {

if (fp->hdr.type == BINDER\_TYPE\_HANDLE) //handle 转成 binder

fp->hdr.type = BINDER\_TYPE\_BINDER;

else

fp->hdr.type = BINDER\_TYPE\_WEAK\_BINDER;

fp->binder = node->ptr;

fp->cookie = node->cookie;

// [...]

binder\_inc\_node\_nilocked(node,

fp->hdr.type == BINDER\_TYPE\_BINDER,

0, NULL);

// [...]

} else {

struct binder\_ref\_data dest\_rdata;

ret = binder\_inc\_ref\_for\_node(target\_proc, node,

fp->hdr.type == BINDER\_TYPE\_HANDLE,

NULL, &dest\_rdata);

// [...]

fp->binder = 0;

fp->handle = dest\_rdata.desc;

fp->cookie = 0;

}

done:

binder\_put\_node(node);

return ret;

}

此转换函数与上一个函数非常相似。但是，我们可以注意到，handler引用可以在不同的进程之间共享。当目标进程与节点匹配的时候才会进行handler到binder的转换。

**文件描述符**

当binder对象类型为BINDER\_TYPE\_FD或BINDER\_TYPE\_FDA时，内核需要检查文件描述符是否正确（与打开的结构文件相关联）并在目标进程中将其复制。转换由binder\_translate\_fd函数完成。详情如下：

// file : drivers/android/binder.c

static int binder\_translate\_fd(int fd,

struct binder\_transaction \*t,

struct binder\_thread \*thread,

struct binder\_transaction \*in\_reply\_to)

{

// [...]

// 1 : Check if the target allows file descriptors

if (in\_reply\_to)

target\_allows\_fd = !!(in\_reply\_to->flags & TF\_ACCEPT\_FDS);

else

target\_allows\_fd = t->buffer->target\_node->accept\_fds;

if (!target\_allows\_fd) {

binder\_user\_error("%d:%d got %s with fd, %d, but target does not allow fds\n",

proc->pid, thread->pid,

in\_reply\_to ? "reply" : "transaction",

fd);

ret = -EPERM;

goto err\_fd\_not\_accepted;

}

// 2 : Get file struct corresponding to the filedescriptor number

file = fget(fd);

if (!file) {

binder\_user\_error("%d:%d got transaction with invalid fd, %d\n",

proc->pid, thread->pid, fd);

ret = -EBADF;

goto err\_fget;

}

// 3 : SELinux check

ret = security\_binder\_transfer\_file(proc->tsk, target\_proc->tsk, file);

if (ret < 0) {

ret = -EPERM;

goto err\_security;

}

// 4 : Get a 'free' filedescriptor number in the target process.

target\_fd = task\_get\_unused\_fd\_flags(target\_proc, O\_CLOEXEC);

if (target\_fd < 0) {

ret = -ENOMEM;

goto err\_get\_unused\_fd;

}

// 5 : This inserts the 'file' into the target process with the target\_fd filedescriptor number.

task\_fd\_install(target\_proc, target\_fd, file);

return target\_fd;

// [...]

}

经过一些验证后，最后一次调用task\_fd\_install将在目标进程中添加与调用方文件描述符关联的文件。在内部，它使用内核API函数\_\_fd\_install在进程fd数组中安装文件指针。

Buffer对象

Buffer对象是最有趣的。它们由硬件服务的Parcel类使用（特指HIDL），允许传输内存Buffer。Buffer对象具有一种层次结构机制，可用于修补父对象的偏移量。这对于发送包含指针的结构非常有用。Binder Buffer对象由以下结构定义：

struct binder\_buffer\_object {

struct binder\_object\_header hdr;

\_\_u32 flags;

binder\_uintptr\_t buffer;

binder\_size\_t length;

binder\_size\_t parent;

binder\_size\_t parent\_offset;

};

让我们看一个例子：我们有以下代码，我们想使用Binder 发送hidl\_string结构的实例。

struct hidl\_string {

// copy from a C-style string. nullptr will create an empty string

hidl\_string(const char \*);

// ...

private:

details::hidl\_pointer<const char> mBuffer; // Pointer to the real char string

uint32\_t mSize; // NOT including the terminating '\0'.

bool mOwnsBuffer; // if true then mBuffer is a mutable char \*

};

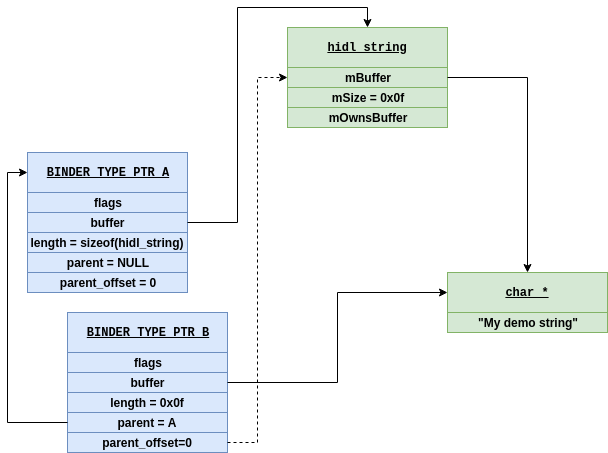
hidl\_string my\_obj("My demo string");

创建my\_obj时，将执行堆分配以存储给定的字符串，并设置属性mBuffer。要将这个对象发送到另一个进程，需要两个BINDER\_TYPE\_PTR对象：

第一个binder\_buffer\_offset缓冲区字段指向my\_obj结构

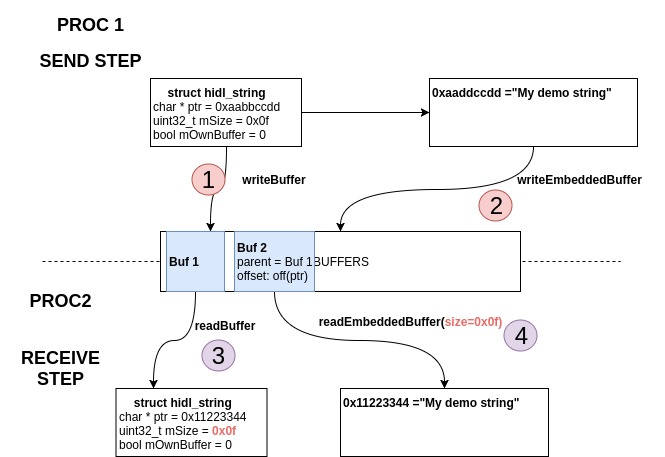
第二个指向堆中的字符串。该对象必须是先前对象的子对象，并将parent\_offset属性设置为char \* str在结构中的位置

下图详细说明了所需的两个Binder对象的配置：



Binder消息缓存

当内核转换这些对象时，它将修补子缓冲区中描述的偏移量，并将不同的缓冲区（[object.buffer，object.buffer + object.length]）复制到目标内存进程中。



Binder消息缓存

为了解析my\_obj数据，目标进程读取第一个缓冲区以获取hidl\_struct （3），而下一个缓冲区的预期大小为mSize，确保结构中描述的大小（mSize）与包含字符串（4）的缓冲区的大小相同。

**结论**

Binder是一个复杂而强大的IPC / RPC系统，它可以使整个Android生态系统正常工作。即使内核组件很旧，也很少有有关其工作原理的文档。此外，对象类型BINDER\_TYPE\_FDA和BINDER\_TYPE\_PTR是最近添加到Android内核中的（https://lore.kernel.org/patchwork/patch/757477/）。这些新类型是Android 8.0中新引入的HAL架构中的通信（HIDL）的基础（project Treble）。

附记1：

Android Binder 详解，说明binder和handle的关系。

<https://blog.csdn.net/pwl999/article/details/84978109>



衍生出的原则如下：

service\_server类的进程只有binder对象，没有handle(除了handle0)，因为它所有操作都必须本地执行，引用远程对象毫无意义；

service\_client类的进程只有handle，没有binder对象，因为它需要远程执行service不需要本地执行；

service\_mannager进程同时有binder对象和handle，它本地binder对象的作用就是操作所有其他进程的handle；

附记2：

在通用 3.10、3.18、4.4、4.9 版内核和上游中

在之前的 Android 版本中，Binder 调用中的每条数据都会被复制 3 次：

一次是在调用进程中将数据序列化为 Parcel

一次是在内核驱动程序中将 Parcel 复制到目标进程

一次是在目标进程中对 Parcel 进行反序列化

Android O 使用分散-集中优化机制将复制次数从 3 次减少到了 1 次。数据保留在其原始结构和内存布局中且 Binder 驱动程序会立即将数据复制到目标进程中，而不是先在 Parcel 中对数据进行序列化。在目标进程中，这些数据的结构和内存布局保持不变，并且，在无需再次复制的情况下即可读取这些数据。