Android Framework

1. 准备工作
   1. 下载Android源代码

□ apt-get install git-core curl #先下载这两个工具

□ mkdir android-froyo #建立下载目录

□ cd android-froyo #进入下载目录

□ curl https://mirrors.tuna.tsinghua.edu.cn/git/git-repo -o repo #下载repo脚本,通过该脚本可下载整套源码

□ chmod a+x repo #设置该脚本为可执行

□ vi repo #编辑repo文件，修改REPO\_UR为：REPO\_URL='https://mirrors.tuna.tsinghua.edu.cn/git/git-repo/'

□ ./repo init -u https://aosp.tuna.tsinghua.edu.cn/platform/manifest -b froyo #初始化git库

□ ./repo sync #同步源码树，下载源码，大小约为2GB,如果网速快，估计需要2个小时

* 1. 安装samba

用于在windows下使用souceinsight访问linux中的android源代码。

* apt-get install samba samba-common #安装samba
* sudo mkdir /home/share #创建共享目录,可使用现有目录
* sudo chmod 777 /home/share #修改共享目录访问权限
* vi /etc/samba/smb.conf #修改samba配置,在末尾添加：

[share]

path = /home/yongdaimi/share

available = yes

browsealbe = yes

public = yes

writeable = yes

* sudo apt-get install libtalloc2 #安装libtalloc2,解决没有权限访问网络资源的问题
* sudo service smbd restart #重启samba服务

1. 操作系统基础

2.1 进程间通信的经典实现

2.1.1 管道

管道通信的编程实现(略)

2.1.2 Unix Domain Socket

Socket通信的编程实现(按书上的代码敲,未实现效果)

2.1.3 RPC(Remote Procedure Calls)

2.1.4 共享内存

2.2 同步机制的经典实现

2.2.1信号量(Semaphore)

2.2.2 Mutex

2.2.3管程(Monitor)

Android的音频子系统里的AudioTrack和AudioFlinger就用到了生产者/消费者模型。

2.2.4 Android中的同步机制

2.2..4.1 Mutex

头文件在：framewokrs/native/include/utils./Mutex.h

2.2..4.2 Condition

头文件在：frameworks/native/include/utils/Condition.h

2.2.4.3 Barrier

头文件在：frameworks/native/services/surfaceflinger/Barrier.h

2.2.5 操作系统内存管理基础

<略>

2.2.6 Android中的Low Memory Killer

<略>

2.2.7 Android中的匿名共享内存

<略>

2.2.8 JNI

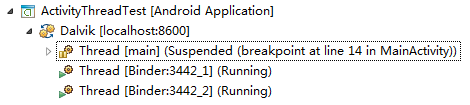
<略>

1. Android进程/线程管理

3.1 Android的进程和线程

核心API：**frameworks/base/core/java/android/app/ActivityThread.java**

* + - 应用程序启动后，将会创建ActivityThread主线程，该线程的main函数为入口点；
    - 同一个包中的所有组件(Activity,Service,Broadcast…)将运行在相同的进程空间;
    - 不同包中的组件可以通过一定的方式(android:process)运行在一个进程空间中；
    - 一个Activity应用启动后至少会有**3**个线程：即一个主线程和两个Binder线程



3.2 Handler,MessageQueue,Runnable与Looper

核心API：**frameworks/base/core/java/android/os/Handler.java**

**frameworks/base/core/java/android/os/MessageQueue.java**

**frameworks/base/core/java/android/os/Looper.java**

3.2.1 Handler.的post(Runnable r)方法

该方法传入的参数虽然是Runnable类型，但是实际执行时,会调用getPostMessage方法

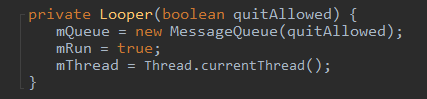
将待执行的Runnable任务包装成一个Message对象，再压入到MessageQueue中。

3.2.2 Handler指定类来处理接收到的消息

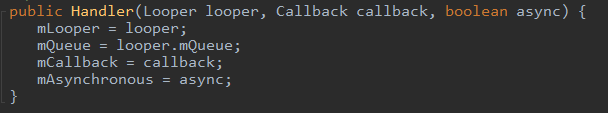
Handler可以通过在构造函数中传入一个自定义的CallBack来处理接收到的消息。该CallBack必须实现Handler内部的CallBack接口,并在重写的handleMessage(Message msg)中处理相关消息。

3.2.3 Handler,Looper,MessageQueue三者之间的通信关系

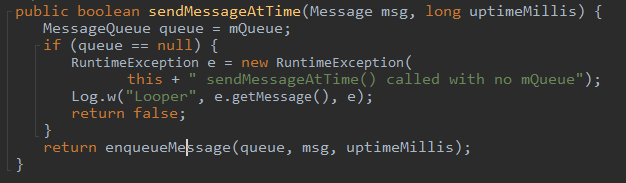
◆A) Looper本身关联了一个MessageQueue(mQueue,通过构造函数创建)，

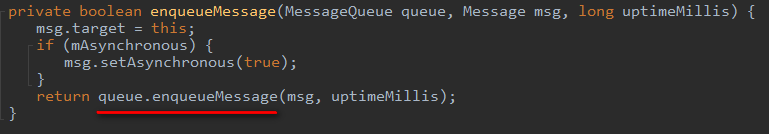


◆B) Handler需要在构造函数时传入Looper,通过传入的Looper,可以拿到Looper身上绑定的MessageQueue

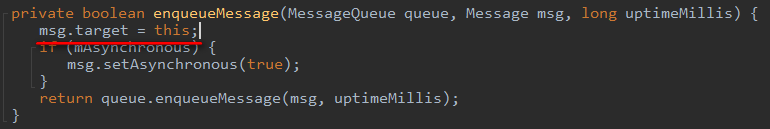


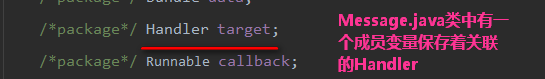
◆C) 这样Handler就可以在后续投递消息时(sendMessageAtTime)将消息投递到mQueue,也就是mLooper.mQueue中。





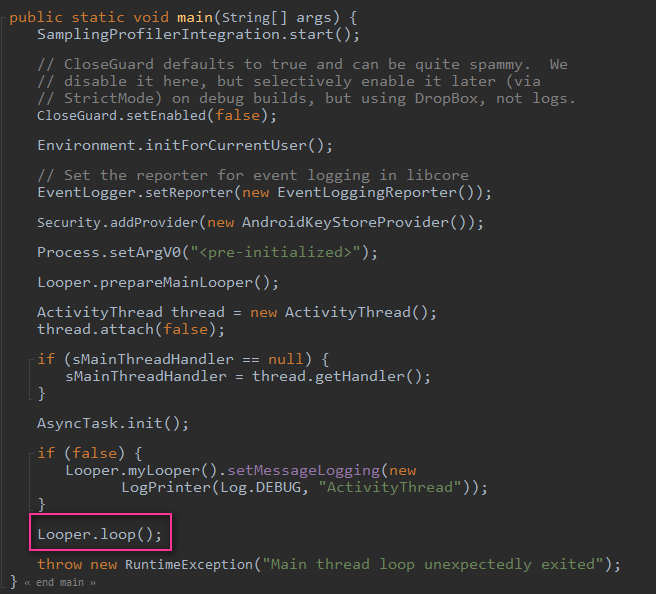
◆D) 接下来，一旦Looper处理到MessageQueue中的某个Message,就又会调用该Message身上绑定的Handler来处理



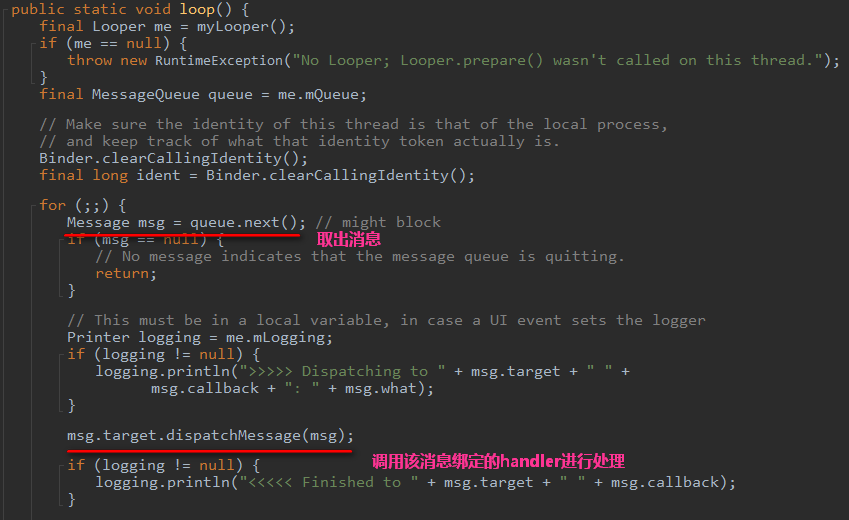


3.2.4 ActivityThread中消息的处理流程

**ActivityThread.java**



**Looper.java**

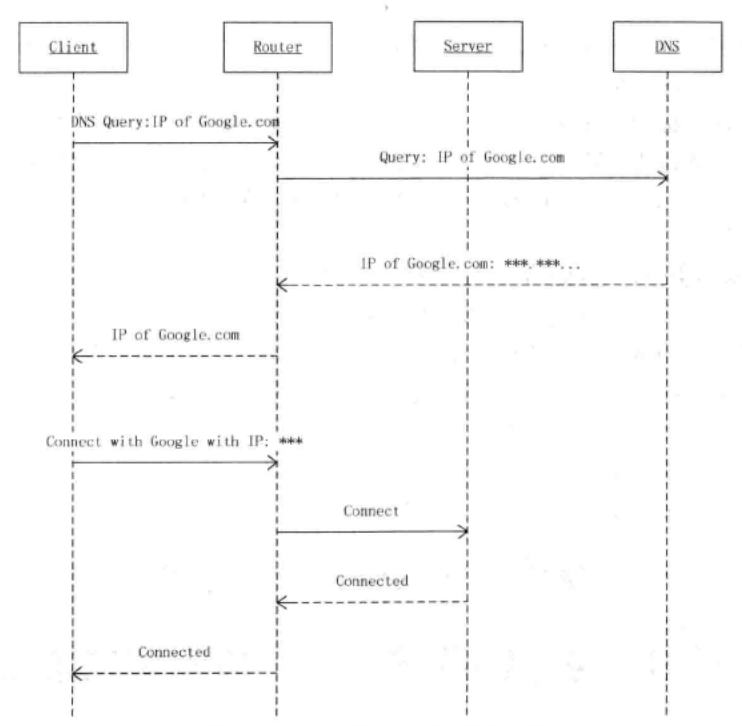


1. 进程间通信-Binder

4.1 Binder的作用及在Android中的具体实现

Binder的主要作用是用来解决Android中的进程间通信问题。因为不同的进程处于不同的地址空间，一个进程无法顺利访问到另一个地址空间的函数名，变量。Android中Binder的通信流程类似于传统的TCP/IP网络通信，进程1与进程2之前的通信类似于客户端与服务器间的通信

传统的TCP/IP网络通信流程如下: (以用户访问Google为例)



* Client向DNS查询Google.com的IP地址(这里假定Client事先不知道Google的IP地址)
* DNS将查询结果返回Client
* Client拿到IP地址后发起连接

其中Router的作用在于根据用户填写的目标IP地址将数据包正确发送到位。

Binder驱动

进程1

进程2

Service Manager

* 进程1与进程2通信相当于Clinet与Server通信；
* 进程1与进程2因为是跨进程的，处于不同的地址空间，因此无法直接通信，所以要借助第三者——**Binder驱动**,相当于网络通信中的路由。
* 进程1要与进程2通信，还必须知道对方叫什么，网络通信中有IP，可以用来标识彼此身份，Android中则为每个进程提供了相应的**Binder标志**。
* 每个进程的Binder标志都是动态的，不可能在某个进程中写死，因此进程1还需要每次都能正确获得进程2的Binder标志。Android提供了**ServiceManager**类来向进程提供查询功能。该类则相当于网络通信中DNS。

PS：ServiceManager在Binder通信过程中的唯一标志永远都是0

4.2 **智能指针(重点)**

4..2.1 传统指针的问题(引入智能指针的必要性)

a. 指针没有初始化化

b. new了之后没有及时delete

c．野指针(做了delete，但没有把指针置为NULL,再次使用出错)

4..2.2 智能指针解决了哪些问题

a. 确保指针在创建时即置为NULL

b. new了之后就一定会delete

4..2.3 智能指针如何实现

典型实现：**LightRefBase**

位于：**frameworks/native/include/utils/LightRefBase.h**

具体实现：

**4.2.3.1** **智能指针是一个类，其内部应有一个指针变量指向objectt**

class SmartPoint {

private:

void \*m\_ptr; // 指向object对象

};

**4.2.3.2** **智能指针是一个模板类，不针对某种特定类型的对象**

template<typename T>

class SmartPoint {

private:

T \*m\_ptr; // 指向object对象

};

**4.2.3.3** 确保指针在创建时即置为NULL

template<typename T>

class SmartPoint {

private:

T \*m\_ptr; // 指向object对象

public:

inline SmartPoint() :m\_ptr(0){}

};

**4.2.3.4** new了之后一定会delete

简单的说就是要让智能指针知道什么时候需要去释放内存对象。一般来说，若一个指针指向了object,那这个object就仍然有用；若这个指针不再指向这个object，那么这个object就没用。

当然了,也会有两个以上的指针会同时指向同一个内存对象。在这种情况下，内存对象的释放就必须等到所有指向该对象的指针都不再指向这个对象后,才能够去delete掉这个内存对象。那么该如何实现这种需求呢？

可以使用一个**计数器**记录该内存对象的“被需要”个数(引用次数)，当这个计数器减为0的时候，就代表没有指针再指向这个内存对象了。换言之，就可以将这个内存对象delete掉了。iOS开发里经常提到的ARC就是这种东西

那么这个计数器该谁持有？

1. 由智能指针拥有？

这种情况肯定是不对的。假设有两个智能指针：SmartPoint1、SmartPoint2，若SmartPoint1释放了自己与object的连接，则会将mCount减1，此时发现计数值已为0，就去删除object。但如果SmartPoint2还在使用object呢？

所以：

**计数器必须由object自身持有**。

**4.2.3.5** 创建一个对象，使之具备计数器的功能

template<typename T>

class LightRefBase {

private:

mutable volatile int32\_t mCount; /\*引用计数值\*/

public:

inline LightRefBase() :mCount(0){}

void incStrong() const {} /\*增加引用计数\*/

void decStrong() const {} /\*减少引用计数\*/

protected:

inline ~LightRefBase(){}

};

**4.2.3.6** 确保对象在被引用的时候,计数器能够正常工作(正常调用incStrong和decStrong)

4..2.4 Android中智能指针的实现——强指针

Android中的强指针一般都命名为**sp**(**StrongPointer**),强指针所在的头文件：

**StrongPoint.h**

**frameworks/native/include/utils/StrongPointer.h**

4..2.5 Android中智能指针的实现——弱指针

◆**强指针带来的问题**

强指针的引入虽然最大程度的解决了指针没有及时delete和野指针的问题，但也带来了新的问题。如：有两个类(CDad,CChild)，他们都具有引用计数的功能，但是CDad指向了CChild，CChild又指向了CDad，这样由于这两个类相互指向，所以他们的引用计数器永远不为0。内存回收者发现这两个类都处于被“需要”的状态，所以也不会去释放，这样就会导致内存泄露。

◆**弱指针的引入**

为解决上述问题，开发人员又引入了一种新的思路：就是CDad使用强指针来引用CChild，而CChild只使用**弱引用**来指向父类。双方规定：**当强引用计数为0时，不论弱指针是否为0都可以delete自己**。这样只要一方得到了释放，就可以成功避免死锁。

◆**弱指针存在的其它问题**

上述方案虽然可行，但是又带来了另外一个新的问题，如：CDad对象因为强指针计数已经到0，根据规则这个对象应该被delete掉，但此时CChild还持有其父类的弱引用，显然如果CChild此时用这个指针来访问CDad将造成致命问题。鉴于此，我们特别规定：

**弱指针必须先升级为强指针，才能访问它所指向的目标对象。**

Android中弱指针被命名为wp,位置在,

**wp**

**frameworks/native/include/utils/RefBase.h**

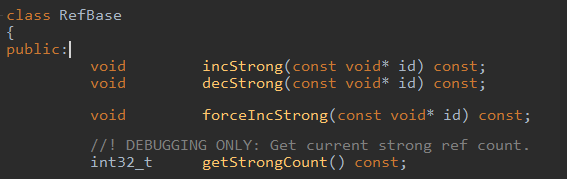
通过分析wp的源码，可以发现wp和sp在类定义上有如下重要区别：

◆ 除了指向目标对象的m\_ptr外，wp另外有一个m\_refs指针，类型为**weakref\_type;**

◆ 有一个prmote方法来将wp提升为sp;

### 4..2.6 强指针和弱指针的具体实现

首先，强指针或是弱指针指向的对象必须**实现引用计数**的功能。Android中提供了这样一个基类，它的名字是RefBase(在RefBase.h)，只要继承了该类，目标对象就具备了引用计数的功能：

****

1. 源码工程中强指针、弱指针的定义位置

**强指针**：frameworks/native/include/utils/StrongPointer.h

**弱指针**：RefBase.h

**实现引用计数功能的模板类**：RefBase.h

4.3 **进程间通信的数据载体Parcel(重点)**

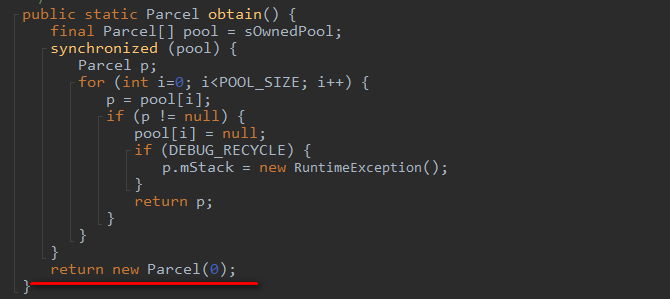
Parcel是进程间通信中数据传递的载体，用于承载希望通过IBinder发送的相关信息，Parcel可用来存储以下几种类型数据：

* **基本数据类型**。如(int, byte, float, double等)，对应的实现方法有(writeByte(b), writeDouble(double))等。
* **基本数据类型数组**。如byte[], boolean[] 对应的实现方法有writeByteArray(), writeBooleanArray等。
* **实现了Parcelable接口的对象**。只要是实现了Parcelable接口的对象也可以使用Parcel来存储，对应的方法有writeParcelable等。
* **Bundle**。Bundle也实现了Parcelable接口。
* **其它Java容器**。如Object[], List, 对应的方法有writeArray(Object[]), writeList(List)
* **其它原始对象**。如Binder, Linux中的FileDescriptor,对应的方法有writeStrongBinder(), writeFileDescriptor。

4.3.1 Parcel对象的创建过程

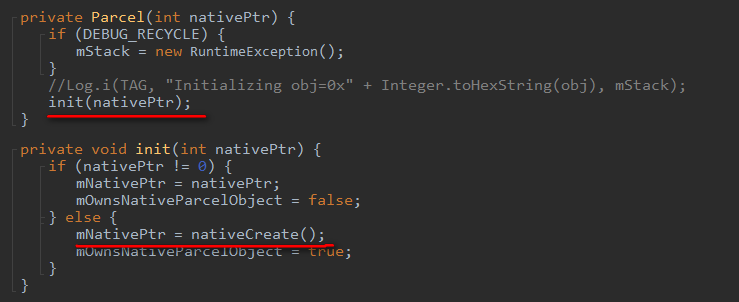
1. **首先通过Parcel.obtain()接口来获取一个Parcel对象。**

frameworks/base/core/java/android/os/Parcel.java



最终调用到了Parcel的构造函数，并为其传入了一个nativePtr为0的指针地址。

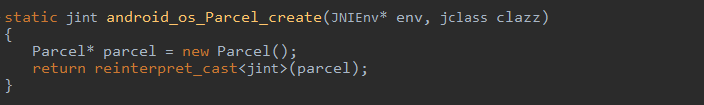
1. **调用本地方法nativeCreate()创建一个native对象，并将创建完的指针地址返回给Java层。**



可以看到，先是调用私有构造方法，然后在私有构造方法中调用init方法，并将默认指针地址0传入，在init方法中当检测到nativePtr是0的时候，就调用nativeCreate()方法创建一个本地的Pacel对象,并将创建完成后的指针地址返回给java层。

1. **进入JNI层的android\_os\_Parcel\_create()方法中进行Parcel对象的创建。**

frameworks/base/core/jni/android\_os\_Parcel.cpp

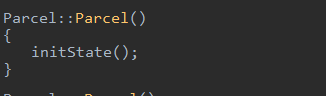


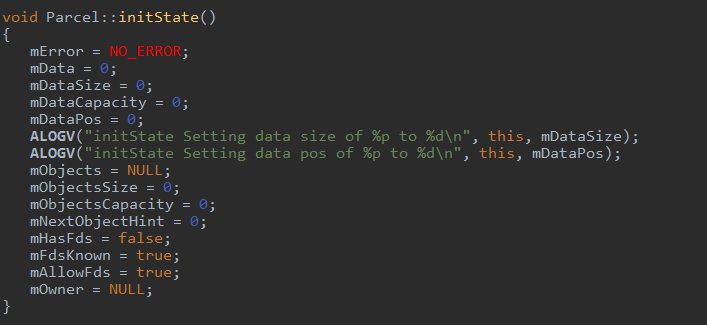
这里创建了一个Parcel对象，并将Parcel对象的指针返回。将Parcel指针转换成jint类型返回给java层。

1. **调用Parcel类的构造方法进行Parcel对象的创建。**

frameworks/native/incluce/binder/Parcel.h

frameworks/native/libs/binder/Parcel.cpp





Parcel类的构造方法中只有一个函数initState(), 该函数内部主要对Parcel类的一些成员变量进行了初始化。这些成员变量包括：

status\_t mError; // 错误码

uint8\_t\* mData; // Parcel中存储的数据，它是一个uint8\_t类型的指针

size\_t mDataSize; // Parcel中已经存储的数据大小

size\_t mDataCapacity; // Parcel的最大存储能力

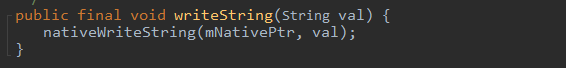
mutable size\_t mDataPos; // 数据指针

4.3.2 Parcel对象的写入过程(Parcel对象写入String)

我们以Parcel传递String对象为例来推演Parcel的传递过程。

1. **首先调用Parcel的writeString()方法**

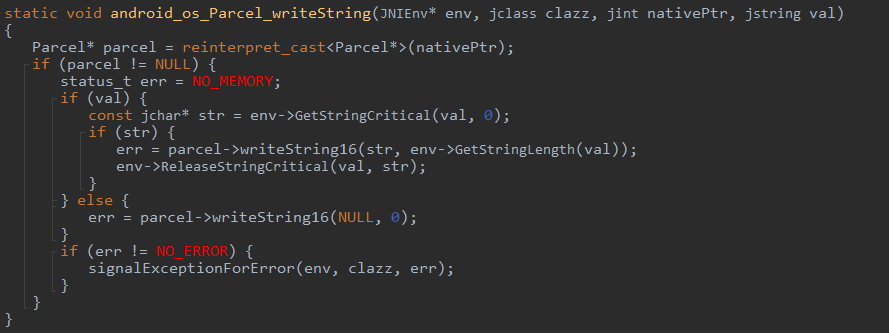
frameworks/base/core/java/android/os/Parcel..java



这个方法很简单，就是直接调用了native层的nativeWriteString方法。注意这里传入了一个mNativePtr,它就是之前我们创建Parcel对象返回的指针。

1. **然后调用到JNI层的android\_os\_Parcel\_writeString()方法对字符串进行写入**

frameworks/base/core/jni/android\_os\_Parcel.cpp



这里先通过强制类型转换：

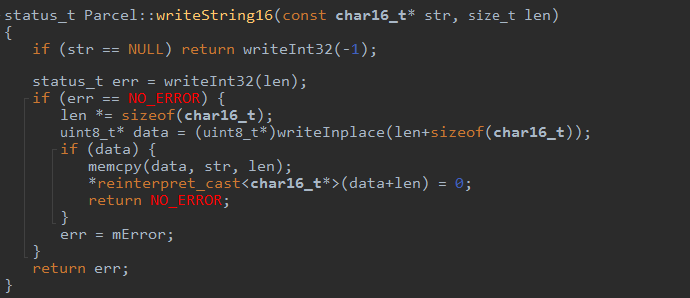
Parcel\* parcel = reinterpret\_cast<Parcel\*>(nativePtr);

将先前返回给java层的jint类型指针强转为Parcel类型的指针

判断传过来的Parcel指针和字符串不为空，紧接着调用Parcel类的**writeString16()**方法对字符串进行写入。如果Parcel指针或字符串为NULL，就返回相应的错误码，最后如果检测到有错误码就报异常。

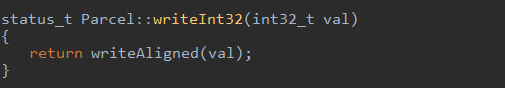
1. **调用Parcel类的writeString16方法对字符串进行写入**

frameworks/native/libs/binder/Parcel.cpp



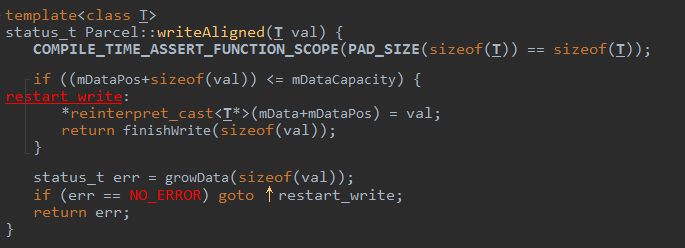
该方法被要求传递两个参数：一个是需要写入到Parcel的字符串指针，另一个则是该字符串的长度。当检测到传递过来的字符串不为NULL后，则先调用writeInt32方法写入字符串长度

1. **调用writeInt32()方法先写入字符串长度**



可以看到，这里又调用了writeAligned方法进行写入。

1. **调用writeAligned()方法写入数据长度**



这里定义的宏用到了C++的一个重要特性:编译期Assert:

COMPILE\_TIME\_ASSERT\_FUNCTION\_SCOPE(PAD\_SIZE(sizeof(T)) == sizeof(T));

可以先不用理会，详情可参考：

[浅析C++ Compile-time Assertion技术](https://blog.csdn.net/shenya1314/article/details/53895720)

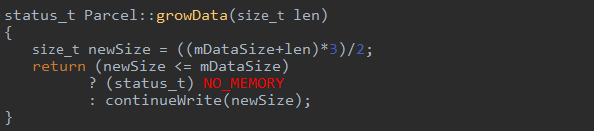
或者百度搜索：编译期ASSERT

首先判断当前数据位置加上要写入数据的长度有没有超过了Parcel的存储能力

if ((mDataPos+sizeof(val)) <= mDataCapacity)

首次写入时由于mDataPos是0，mDataCapacity也是0,而字符串len的长度是**int32\_t**,所以0+4>0的，也就是说必然会超过Parcel的存储能力，因此if代码块的内容就不会执行，而是跳转到下面的growData方法中

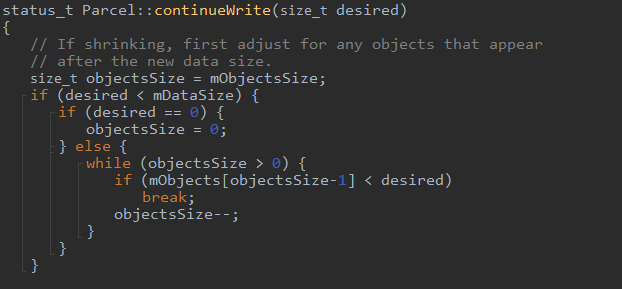
1. **若将要写入Parcel的数据长度超过了Parcel的最大存储能力，就调用growData方法扩充Parcel的存储能力**



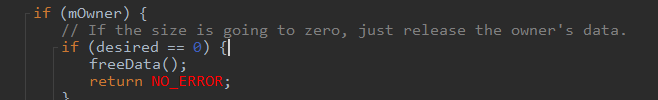
这里首先计算出一个newSize,这个newSize目前等于6：**（(0+4)\*3）/2**

然后对newSize和mDataSize比较大小，很明显newSize此时是大于mDataSize(**此时为0**)的，然后调用continueWrite()方法

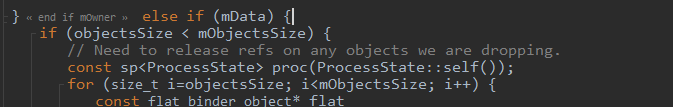
接下来分析continueWrite()方法，该方法比较长，但有效代码很短。



参数desired此时为6，大于mDataSize代码块不执行。

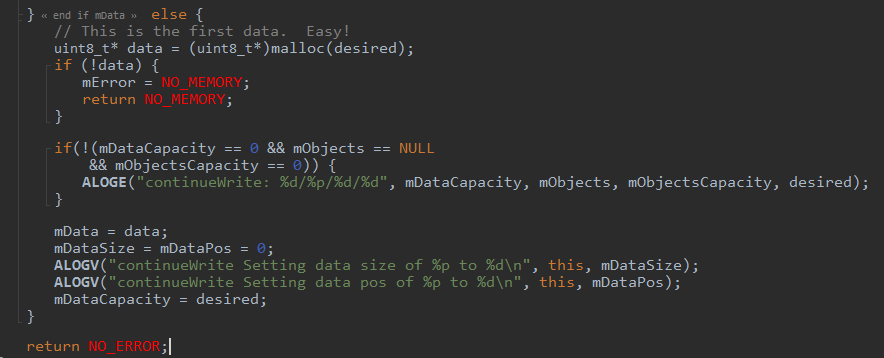


mOwner在initState方法中先初始化为NULL,代码不执行。



mData此时也被初始化为NULL，代码不执行。

所以真正有效的代码只有下面一段：



这里首先调用malloc()方法分配了6个字节的内存，然后做了如下工作：

1.**将返回的数据指针赋给mData**。mData = data;

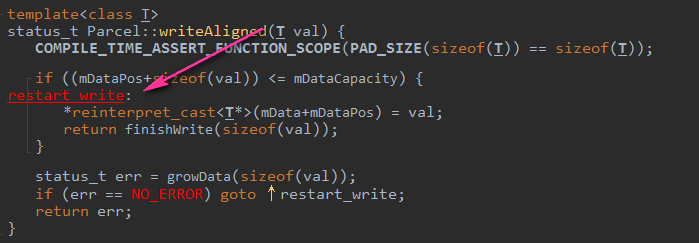
2.**将mDataSize和mDataPos置为0**。mDataSize = mDataPos = 0, 猜测可能是因为还没有开始写数据

3.**更新Parcel存储容量**。mDataCapacity = desired。

所有操作完成之后，返回一个状态值：NO\_ERROR。

紧接着，在writeAligned方法中，如果检测到growData的结果是NO\_ERROR，则继续跳转到**restart\_write**处去执行，也就是之前的goto标志。

也就是下面这个地方：



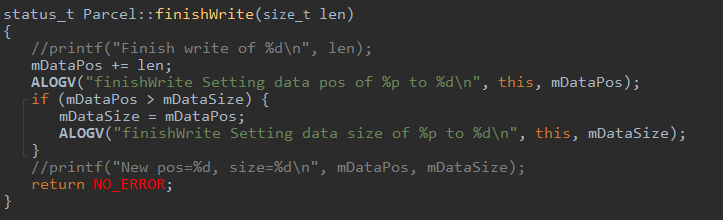
然后调用

\*reinterpret\_cast<T\*>(mData+mDataPos) = val;

将数据(这里是长度)写入到指定位置。

最终调用finishWrite()方法完成本次写入。

1. **调用finishWrite()方法完成本次写入**



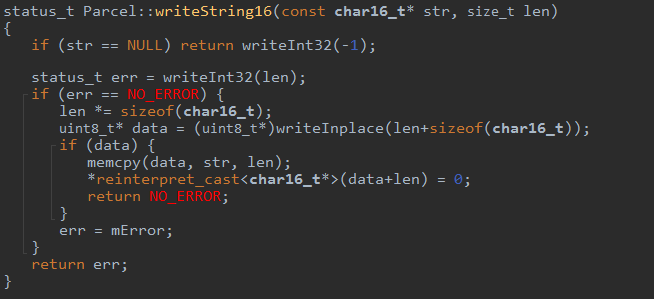
这里主要两件事情，分别是:

1.**更新mDataPos的值**。mDataPos**+=**len; // 将mDataPos更新为当前已写入的数据长度总和。

2.**更新mDataSize的值**。mDataSize = mDataPos.。

然后返回状态码：NO\_ERROR，本次处理完毕。

1. **字符串长度写完之后，然后开始写字符串**



若检测到之前写入字符串长度无误(**NO\_ERROR)**，这里首先计算出字符串的占用空间（**长度\*单位大小=占用的空间**）：

len \*= sizeof(char16\_t) // char16\_t 长度未知

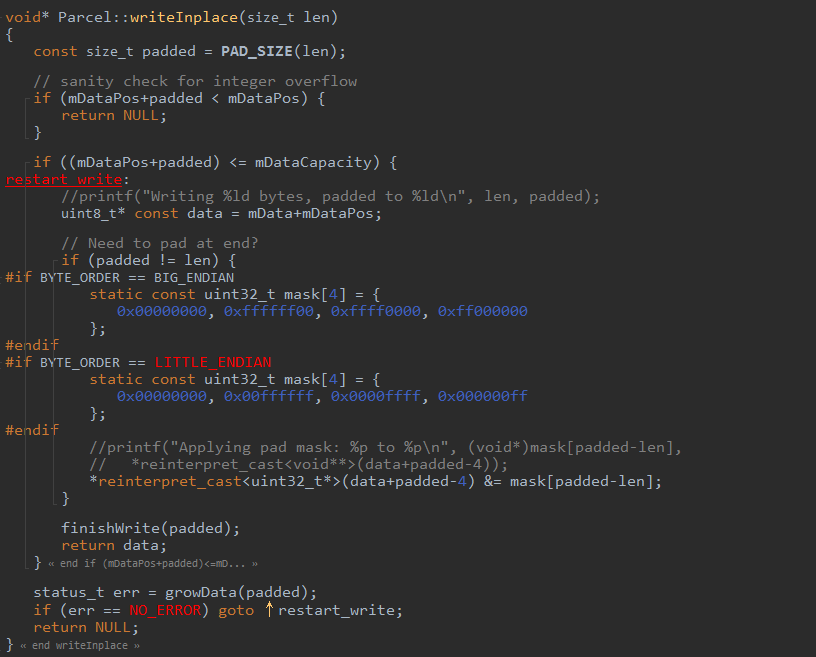
注意: 这里可能受到了Unicode字符长度的影响，JNI层获取字符串长度时调用的方法是GetStringLength()而不是GetStringUTFLength()方法

详情可参考：[从字节理解Unicode(UTF8/UTF16)](https://www.cnblogs.com/zizifn/p/4716712.html)

然后调用writeInplace()来计算复制数据的目标地址。

1. **调用writeInplace()方法来获取待写入数据的目标地址(数据究竟应该写到哪个位置)**

下面是writeInplace()方法的实际内容：



先看第一句代码：

const size\_t padded = PAD\_SIZE(len)， 这里首先定义了一个宏：**PAD\_SIZE**

C:\Users\xp.chen\Desktop\2018-09-04_100241.png

这个宏的主要作用是用于计算“**当以4对齐时,容纳len大小的数据需要多少空间**”比如：

len = 3, padded = 4;

len = 4, padded = 4;

len = 5, padded = 8;

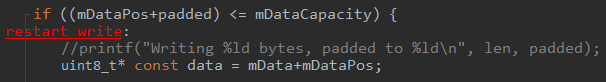
然后判断

if (mDataPos+padded < mDataPos) {

return NULL;

}

mDataPos现在为4，加上padded很明显要大于现有的mDataP os,所以此时不会返回NULL，接着执行下面的代码



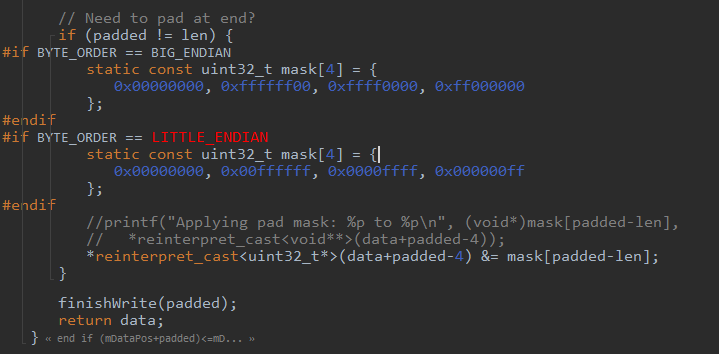
mDataPos此时为4，而4+padded很明显也要大于mDataCapacity(当前为6),所以这个代码块也不执行。当发现Parcel的数据存储能力又无法满足要求时，最终又调用growData()方法来扩充Parcel的存储能力，扩充成功后返回到goto的标志处：**restart\_write**继续执行。

C:\Users\xp.chen\Desktop\2018-09-04_100241.png

Parcel扩充完毕后就可以拿到当前准备写入数据的头指针，也就是**data**

uint8\_t\* const **data** = mData+mDataPos;

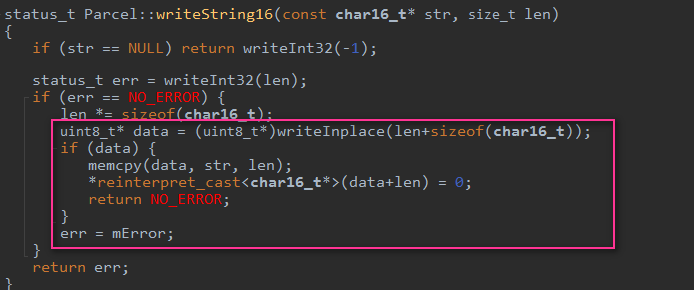
这时候还要去做一个处理就是要对新申请的内存做初始化，因为通过PAD\_SIZE计算出来的长度可能与之前传进来的len长度不一致



这里定义了两个静态数组，通过判断当前是大端还是小端，再与该数组进行&操作，然后调用finishWrite方法更新**mDataPos**和**mDataSize**的值，最终返回data指针，该指针就是下份数据即将要写入的位置。

通过前面的计算，我们得到了字符串将要写入的位置，最后一步就是调用memcpy函数将字符串拷贝到指定位置。

1. **调用memcpy函数将字符串拷贝到指定位置**



如上，首先判断之前计算的写入指针位置是否有效，然后调用memcpy函数将数据str拷贝到data所在的位置。因为拷贝的是字符串，还需要将字符串结尾置为”\0”。不必担心最后一位置0会超出Parcel的容量，因为Parcel在growData的时候会提升足够多的容量。

最后，返回NO\_ERROR，本次写入结束。

**小结：Parcel写入字符串的过程**

* **首先写入字符串的长度(len)**。writeInt32()
* **其次计算要将写入字符串的起始位置**(这一步还包含对Parcel长度的padding,若Parcel容量不足，还要对其进行扩展，扩展数为4的倍数)。
* **调用memcpy函数将字符串拷贝到指定位置**。

4.4 **Binder驱动和Binder驱动协议(难点)**

进程间通信肯定少不了通信协议，Binder通信协议分为两类，一类是**Binder Command** 协议，描述用户空间向Binder驱动程序发送通信请求，通过binder\_driver\_command\_protocol枚举来定义；另一类是**Binder Return**协议，描述Binder驱动程序返回通信请求处理结果到用户空间，通过binder\_driver\_retrun \_protocol来定义。

4.4.1 Binder驱动协议

**1. enum binder\_driver\_command\_protocol**

| **协议** | **功能** |
| --- | --- |
| BC\_TRANSACTION | 请求 Binder 驱动将通信数据传递给 Server 进程 |
| BC\_REPLY | 请求 Binder 驱动将处理结果传递给 Client 进程 |
| BC\_FREE\_BUFFER | 请求 Binder 驱动程序释放内核缓存区 |
| BC\_INCREFS | 增加一个 Binder 引用对象的弱引用计数 |
| BC\_ACQUIRE | 增加一个 Binder 引用对象的强引用计数 |
| BC\_RELEASE | 减少一个 Binder 引用对象的强引用计数 |
| BC\_DECREFS | 减少一个 Binder 引用对象的弱引用计数 |
| BC\_REGISTER\_LOOPER | 线程通知 Binder 驱动已经准备就绪 |
| BC\_ENTER\_LOOPER | 线程通知 Binder 驱动已经准备就绪 |
| BC\_EXIT\_LOOPER | 线程通知 Binder 驱动需要退出 |

BC\_TRANSACTION 和 BC\_REPLY 协议后面带的通信数据使用结构体 binder\_transaction\_data 来描述。

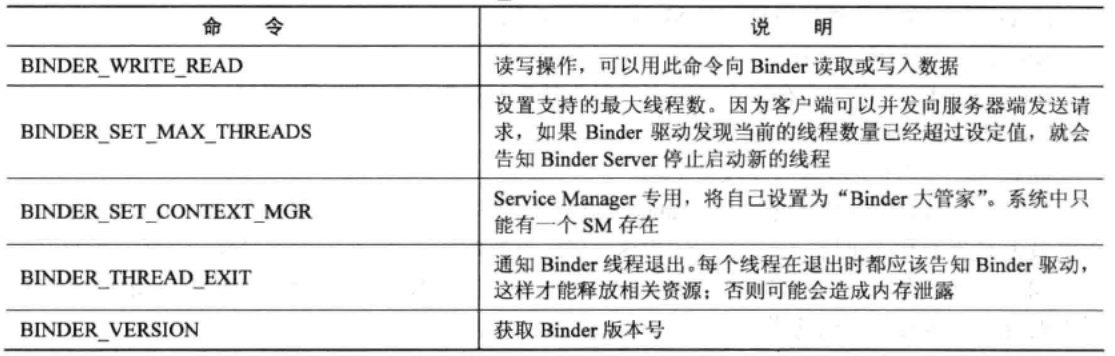
强引用计数可以控制对象的生命周期，而弱引用计数不能控制对象的生命周期，因此使用弱引用计数所引用对象时需要先把弱引用计数升级为强引用计数，如果升级失败，说明所引用对象已经被释放了。

**2. enum binder\_driver\_return\_protocol**

| **协议** | **功能** |
| --- | --- |
| BR\_ERROR | 通知应用程序出现异常情况 |
| BR\_OK | 通知应用程序请求处理完成 |
| BR\_TRANSACTION | Binder 驱动通知 Server 进程处理通信请求 |
| BR\_REPLY | Binder 驱动将请求处理结果通知 Client 进程 |
| BR\_TRANSACTION\_COMPLETE | 通知应用程序 BR\_TRANSACTION 和 BR\_REPLY 协议内容已经被 Binder 驱动接收 |
| BR\_INCREFS | 增加一个 Service 组件的弱引用计数 |
| BR\_ACQUIRE | 增加一个 Service 组件的强引用计数 |
| BR\_RELEASE | 减少一个 Service 组件的强引用计数 |
| BR\_DECREFS | 减少一个 Service 组件的弱引用计数 |
| BR\_SPAWN\_LOOPER | 通知 Server 进程增加一个新线程到 Binder 线程池 |

BR\_TRANSACTION 和 BR\_REPLY 协议后面带的通信数据也使用结构体 binder\_transaction\_data 来描述。

4.4.1 Binder ioctl 所支持的命令

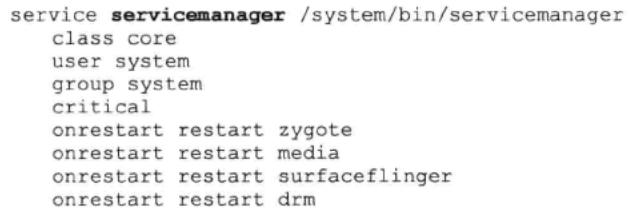


4.5 **DNS服务器——ServiceManager(重点)**

ServcieManager在整个Binder通信中充当了“DNS服务器”的角色。它同时也是一个标准的Binder Server

4.5.1 ServcieManager的启动时机(可略读)

SM为了能够提前给其它应用程序提供服务，因此它启动的优先级比较高，它在init程序解析init.rc时启动的。



ServiceManager的源码工程位于：

**frameworks/native/cmds/servicemanager**目录中

另外，虽然源码工程中还有其它的ServiceManager.cpp文件存在，但是真正属于SM的只有以上目录下的**service\_manager.c**和**binder.c**两个文件。

4.5.2 ServcieManager的启动过程

ServiceManager本身也是一个Binder Server, 在Android系统中，Binder Server的另一个常见称谓是”XX Service”,如：Media Service

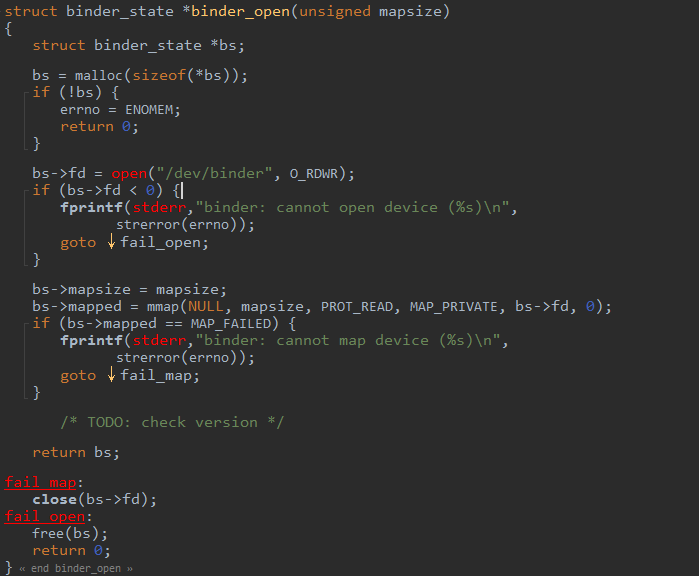
1. **在main()函数中调用binder\_open()函数打开Binder驱动设备**

frameworks/native/cmds/servicemanager/service\_manager.c

frameworks/native/cmds/servciemanager/binder.c

struct binder\_state \*bs;

bs = **binder\_open**(128\*1024);



首先利用malloc函数给binder\_state这个结构体分配了内存，然后调用open()函数打开binder驱动设备，并将该设备文件映射到内存，最后分别给结构体中的各个成员变量进行赋值。binder\_state结构体中的各项成员变量的涵义如下：

int fd // 打开binder驱动设备所返回的文件描述符

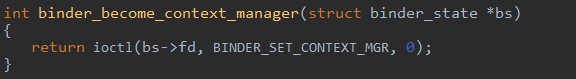
void \*mapped // 映射内存块的起始位置

unsigned mapsize // 映射内存块的大小

所有操作完成后，返回该结构体指针。

1. **调用binder\_become\_context\_manager()函数将SM程序注册到Binder驱动**

frameworks/native/cmds/servicemaanger/service\_manager.c



该函数很简单，就是利用ioctl()函数向Binder驱动发送一条“BINDER\_SET\_CONTEXT\_MSG”的命令。可以看到，程序还对ioctl的返回值做了处理。其目的在于防止其它程序再次调用这个函数，因为**整个Android系统只允许一个ServiceManager存在**。

1. **调用binder\_loop()函数进入循环，等待客户端请求**



该函数要求传入两个参数：一个是包含内存映射信息的binder\_state结构体，另一个是binder\_hander这个函数指针。

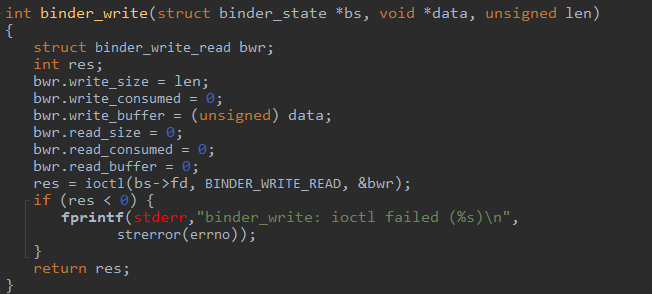
可以看到，这里首先定义了一个名为bwr的binder\_write\_read结构体，这个结构体是**执行BINDER\_WRITE\_READ命令所需的数据格式**，

然后又定义了一个readbuf[32],它是一次性**从Bnder驱动读取的容量**。

在正式开始循环前，还需要事先告诉Binder驱动这一状态变化：

readbuf[0] = **BC\_ENTER\_LOOPER**;

**binder\_write**(bs, readbuf, sizeof(unsigned));



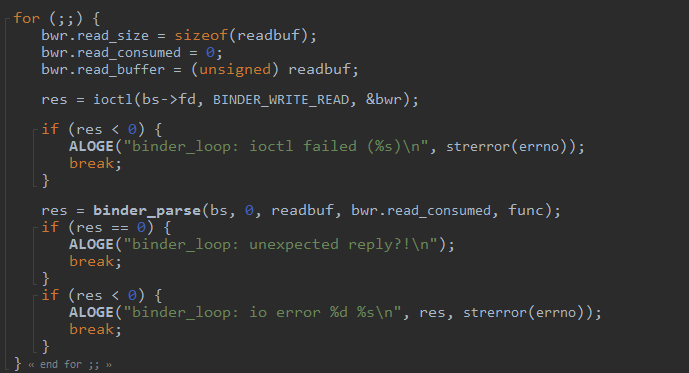
这里告诉了Binder驱动消息的内容(**BC\_ENTER\_LOOPER**),和消息的长度(unsigned)

这里猜测：

Binder的通信协议是一个unsigned int 类型的数组，这个数组的第一个元素是**Binder驱动协议的编号**，后面才是协议真正传输的数据。

当前ioctl执行的命令是**BIND\_WRITE\_READ**, 调用的通信协议是**BC\_ENTER\_LOOPER**。

下面就正式开始循环向Binder驱动发送消息，并对binder驱动返回的结果进行处理。



首先是填写协议的内容

bwr.**write\_size** = 0; // 表示需要写入Binder驱动中的数据大小

bwr.**write\_consumed** = 0; // 表示已经写入Binder驱动中的数据大小

bwr.**write\_buffer** = 0; // 保存用户空间缓存区的地址，该缓存区内容为需要写入Binder驱动的通信数据

bwr.**read\_size** = sizeof(readbuf); // 表示需要从Binder驱动中读取的数据大小

bwr.**read\_consumed** = 0; // **重要！表示已经从Binder驱动中读取的数据大小**

bwr.**read\_buffer** = (unsigned) readbuf; // 保存用户空间缓存区的地址，该缓存区保存从Binder驱动返回用户空间的数据

这里有一点需要注意：

向Binder驱动发送**BINDER\_WRITE\_READ**命令既可以实现读取也可以实现写入。至于究竟是读取还是写入就要看bwr.write\_size和bwr.read\_size这两个结构体中的成员变量。**若write\_size值为0，read\_size值不为0，则Binder驱动只执行读取操作，反之则只执行写入操作**。

在本段代码中write\_size为0，而read\_size为sizeof(readbuf)，则Binder驱动只执行读取操作；而上面给Binder驱动发送**BC\_ENTER\_LOOPER**消息时,read\_size就为0，此时就是写入操作。

Binder驱动在收到SM发送的读取指令后，如果处理成功，则会向readbuf填充数据，接下来SM就要调用binder\_parse()函数来处理这些数据。

1. **调用binder\_parse()函数来处理Binder驱动返回的数据**

frameworks/native/cmds/servicemanger/binder.c

C:\Users\xp.chen\Desktop\2018-09-05_144321.png

该函数要求传入5个参数，分别是：

**bs** // 描述Binder驱动文件描述符和内存映射信息的结构体

**0** // binder\_io结构体,默认值0。

**readbuf** // 先前从binder驱动返回的数据

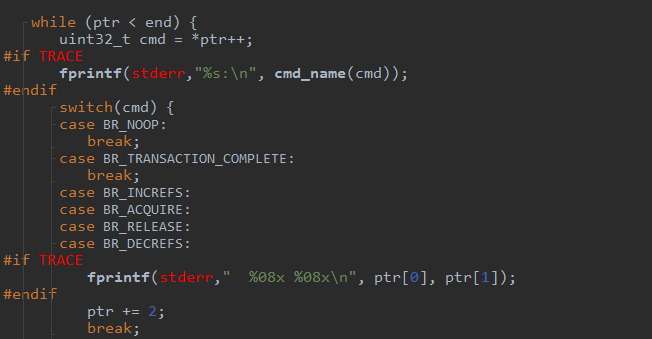
**bwr.read\_consumed** // 已经从binder驱动中读取的数据大小

**func** // binder\_handler函数指针

下面是对binder\_parse()函数的具体分析，该函数很长，我们分段分析

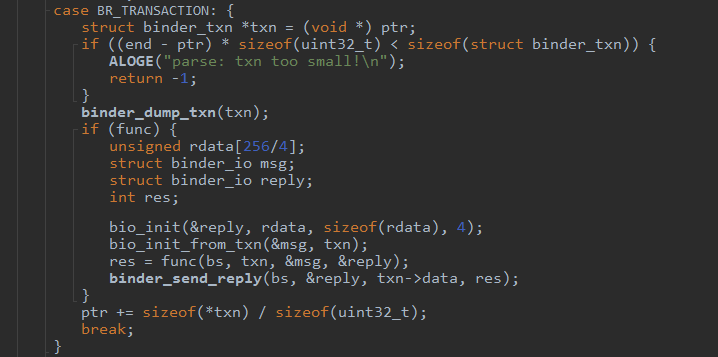
C:\Users\xp.chen\Desktop\2018-09-05_144321.png

代码开头首先获取了返回数据(从Binder驱动中返回的数据)的结束指针，这个size是从Binder中读取的数据大小，除以4是因为一个uint32占用4个字节，接下来若头指针小于末位指针，则进入while 循环：

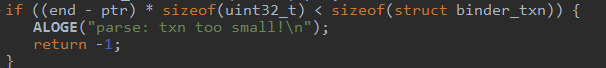


第一步，利用\*ptr++先取得指针内容。注意：这里是先取得指针内容,然后指针再下移。

上面说过，返回数据的第一项是Binder 驱动协议的编号，如果检测到是BR\_NOOP, BR\_TRANSACTION\_COMPLETE则直接break;这里需要重点留意**BR\_TARNSACTION**和**BR\_REPLY**两个协议命令。根据前面的参照表可知，前者是“**Binder驱动通知Server进程处理通信请求**”，后者是“**Binder驱动将请求处理结果告诉Client进程**”先来看BR\_TRANSACTION:

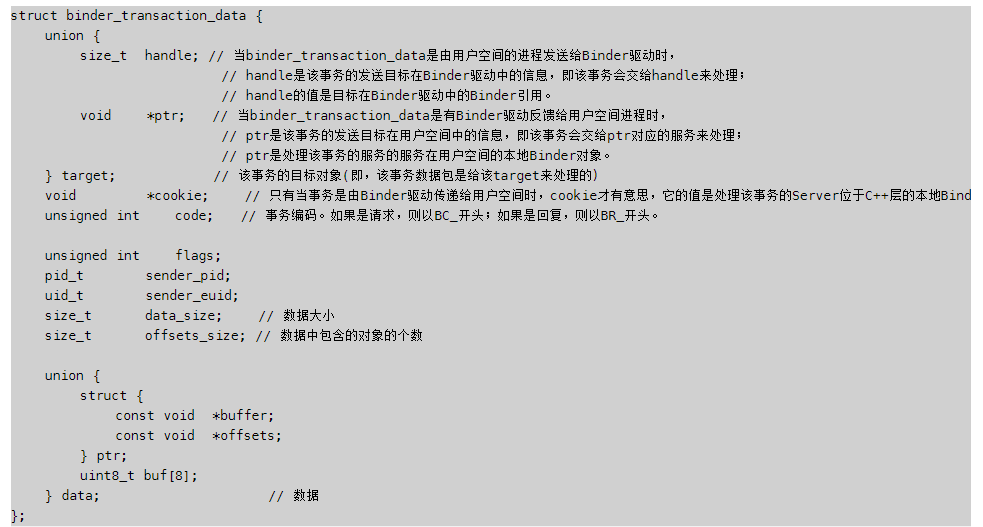


这里首先将ptr的值强转成binder\_txn这个结构体。binder\_txn这个数据结构中的内容是什么意思暂不清楚。



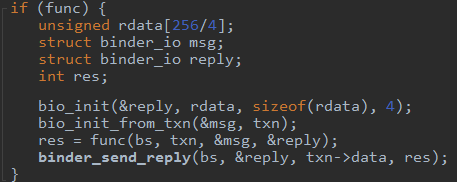
binder\_txn是什么意思目前尚不清楚，网上有个例子可以参考：

下面的**binder\_transaction\_data**是描述Binder事务交互的数据结构体。它也属于内核空间和用户空间的通信结构体。

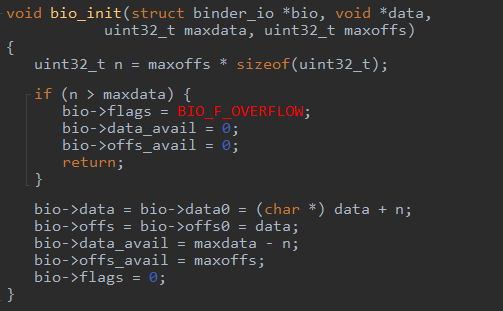


**因此可以将bind\_txn暂时理解成Binder驱动传给SM的数据结构体。**

接下来去判断函数指针fun是否为空，很明显，之前我们设置其不为NULL。



代码中声明了两个**binder\_io结构体**，作用尚不清楚，接下来调用bio\_init()函数对其进行初始化：



这里先说明一下传给bio\_init函数的各项参数值是多少。

C:\Users\xp.chen\Desktop\2018-09-05_144321.png

**&reply,** 是名为reply的biner\_io结构钵指针。

**rdata**, 上面声明了一个unsigned rdata[256/4]类型的buffer数组，也就是一个包含64个unsigned int类型的数组；

**sizeof(rdata),** 长度=64\*4,也就是256, 也是**maxdata**;

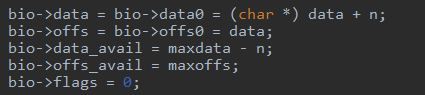
**4**， 就是bio\_init()函数的最后一个参数，也就是maxoffs。

上面看该函数的内容：

**uint32\_t n = maxoffs \* sizeof(uint32\_t);**

这里的n = 4 \* 4 = 16

很明显n < maxdata(值是256)， 因此执行上面各项代码：



bio\_data, // 指向读/写位置的指针，这里指针向偏移了16个位置，不清楚为什么。

bio\_offs. // 偏移数组，这里未偏移；

bio\_data\_avail // 数组缓冲区中可用的字节数，这里是maxdata-n,也就是256-16 = 240,不清楚什么意思。

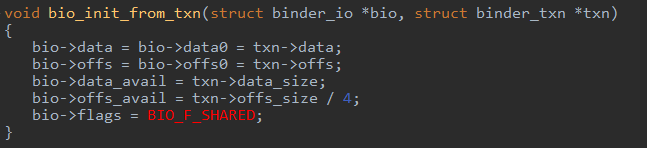
bio\_offs\_avail // 偏移数组中可用的条目，不知道什么意思

bio\_flags // 不清楚什么意思。

接下来用biner\_txn结构体中的数据去填充msg的binder\_io结构体。

bio\_init\_from\_txn(&msg, txn);

该函数如下：

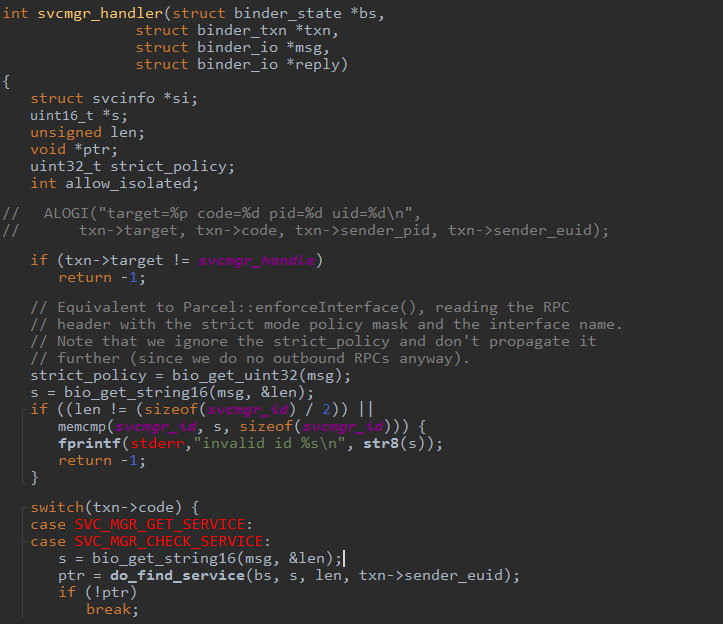


bio->data\_avail // 整个数据包中数据的大小

bio->offs\_avail // 整个数据包中包含的对象的个数

经过上面的填充，reply和msg这两个binder\_io结构体就都有值了。下一步调用func函数，也就是之前传过来的svcmgr\_handler函数对数据进行处理。

1. **调用svcmgr\_handler()函数对数据进行处理**



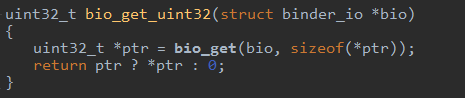
if (txn->target != **svcmgr\_handle**)

return -1;

来看这个方法，首先该方法第一行先去判断处理这段数据的targe是否是**svcmgr\_handle，**我们之前在看binder\_txn这个结构体的时候提到过：

binder\_txn结构体中的target是用来**标识该事务的目标对象**(即:该事务是由哪个程序来处理)，这里判断如果该数据不由SM处理，则return。

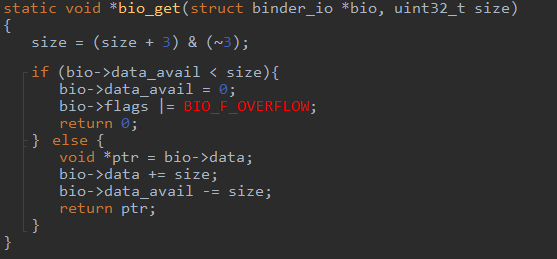
C:\Users\xp.chen\Desktop\2018-09-05_144321.png



这里先取得strict\_policy

bio\_XX系列函数为取出各种类型的数据提供了便利。下面的各种bio\_xx函数都是为了能够在binder\_io结构体中取出各式各样的数据。

最终调用到bio-get()函数



size = (size + 3) & (~3); 指明**未来的size必须是4的倍数**(4,8,12,14,16,etc…)，这里我们假设指针长度为4(32位系统下)

接下来判断bio->data\_avail < size

判断整个数据包中数据的大小是否小于size，若小于则将标记置为**BIO\_F\_OVERFLOW**,很明显不小于。

接下来执行以下操作：

void \*ptr = bio->data; // 取得binder\_io结构体中的数据指针

bio->data += size; // 将数据指针的位置右移size个长度

bio->data\_avail -= size; // 更新结构体中data\_avail的值。将data\_avail的值-size。

return ptr; // 将数据指针返回

最后回到bio\_get\_uint32方法

C:\Users\xp.chen\Desktop\2018-09-05_144321.png

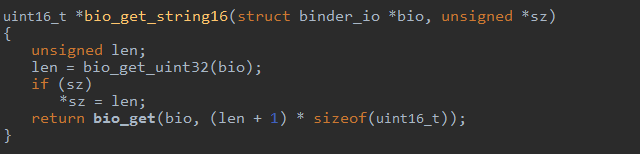
这里先判断头指针是否为NULL，如果不为NULL,则取出头指针的值，否则返回0。

这里的bio\_get\_uint32()函数用于**从binder\_io结构体中获取一个uint32\_t类型的数据，且每获取一次，bio结构体中的数据都偏移一个uint32\_t个长度**。

由上述分析看出：

☞ binder\_io结构体中data指针指向的数组中第一个元素是strict\_policy,占uint32\_t个类型的长度

接下来调用bio\_get\_string16(msg,&len)函数



这里先调用len = bio\_get\_uint32(bio);获得len的长度，接下来把它赋给sz指针指向的变量：\*sz = len;

最后调用bio\_get()函数取得字符串，len+1可能是为了包含‘\0’，后面的

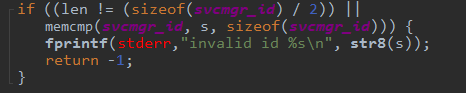
(len+1)\*sizeof(uint16\_t),只是为了计算出字符串的长度，方便接下来指针进行偏移。

从上面代码可以看出：

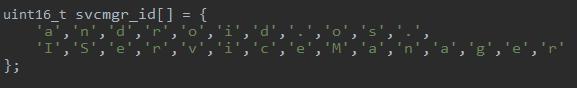
☞ binder\_io结构体中data指针指向的数组中第二个元素是字符串的长度，该字符串具体指什么尚不得知

☞ binder\_io结构体中data指针指向的数组中第三个元素就是实际的字符串，可能是SM进程的interfaceToken。

接下来对len进行判断：



这里先取出svcmgr\_id的长度



这个地方没太看懂，明明按照char类型存储就可以表达255个ASCII字符了，不知道Android为什么把数组定义成uint16\_t(unsigned short)类型，也许是为了容纳更多的字符？

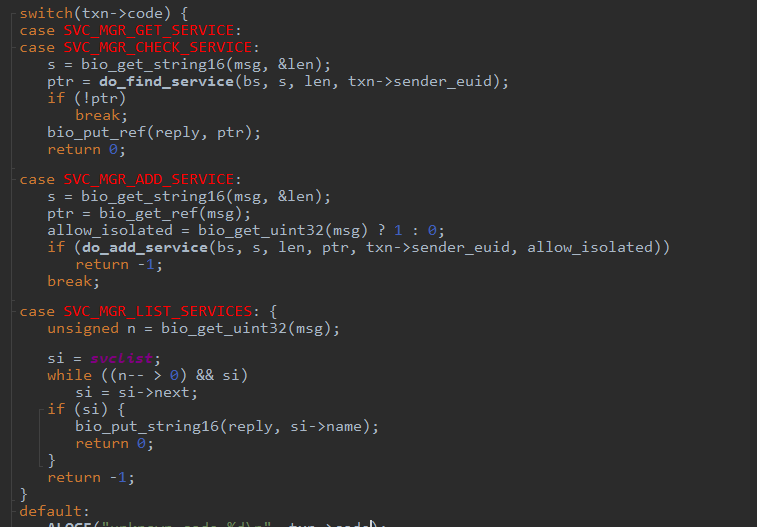
然后判断

1. 该字符串的长度是否不等于 svcmgr\_id的长度
2. 调用memcmp()函数判断这两个字符串是否相同

若上述两个字符串长度不同或内容不同，则认为这个Data不是给SM的，然后打印错误信息，返回。

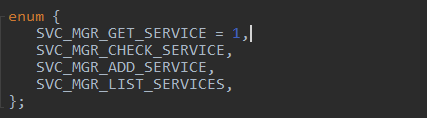
这里初步猜测data指针中的字符串是 之前看到过的 **interfaceToken**, 这个interfaceToken就是每个Binder Server的名称，这里的二级判断主要是看数据到底是不是给SM的。

下面则是根据txn的业务码针对不同的情况进行不同处理：



txn的业务码定义在

frameworks/native/cmds/servicemanager/bind.h 头文件中。



SVC\_MGR\_GET\_SERVICE // 根据Server名称查找它对应的handle值

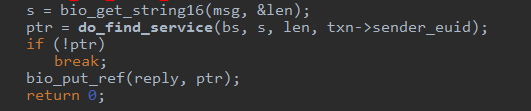
SVC\_MGR\_CHECK\_SERVICE // 同上

SVC\_MGR\_ADD\_SERVICE // 用于注册一个Binder Server

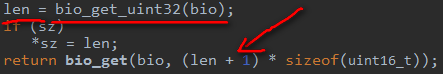
SVC\_MGR\_LIST\_SERVICE // 获取列表中对应的Server

这个业务码其实就是Binder Client 对SM程序的业务编号，或者可以理解成Binder Client想让SM做什么事情。

先看前两个命令：



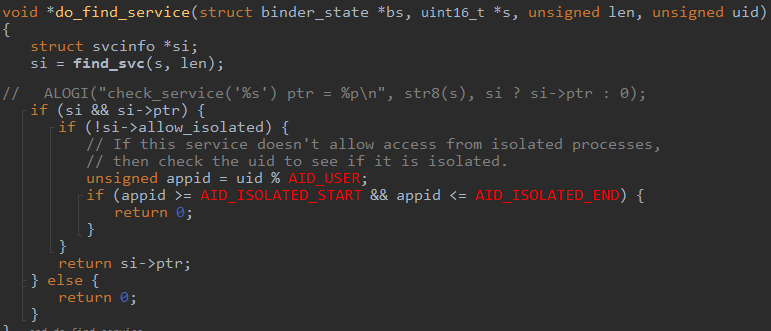
接下来继续取字符串，bio中取字符串很独特，总是先取长度，再取字符串。



s = bio\_get\_string16(msg, &len);

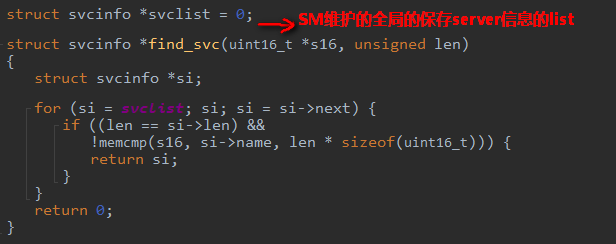
这个地方取得的字符串尚不明了，不过猜测可能是**目标进程的interfaceToken**。SM在拿到该token后就调用do\_find\_service()方法来查找目标进程的对应句柄

1. **调用do\_find\_service()函数查找目标进程的对应句柄**



该函数首先根据传入的目标进程的token和长度调用find\_svc()函数去查找，并返回一个scinfo指针，这里需要注意的是：

**SM本身维护有一个全局的svclist变量，用于保存所有server的注册信息**。find\_svc()就是具体的查找逻辑。



该函数很简单，就是遍历svclist中的各个节点，判断依据有两个：

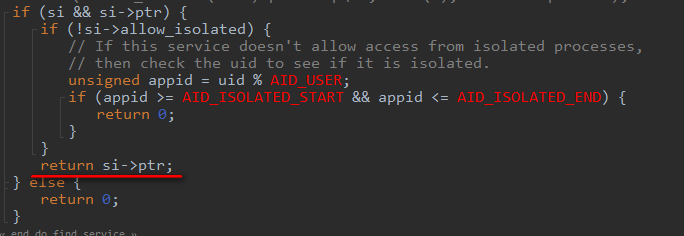
一是判断si的长度是否与传入参数的长度相同

二是判断si的name是否与传入参数的name相同

若在svclist中查找到了同名的si，则将si返回，否则返回0，下面对svcinfo这个结构体做一个简单说明：

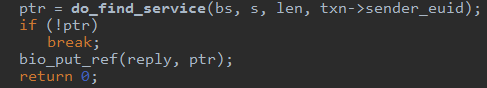
1. struct svcinfo
2. {
3. struct svcinfo \*next; *// 下一个"服务的信息"*
4. void \*ptr; *// 服务在Binder驱动中的Binder引用的描述*
5. struct binder\_death death;
6. int allow\_isolated;
7. unsigned len; *// 服务的名称长度*
8. uint16\_t name[0]; *// 服务的名称*
9. };

接下来判断si中Binder引用是否也不为空，若引用不为空，则最终将Binder引用返回。



回到srcmgr\_handler中，若Binder引用为NULL，则break,并否则调用

bio\_put\_ref(reply, ptr);



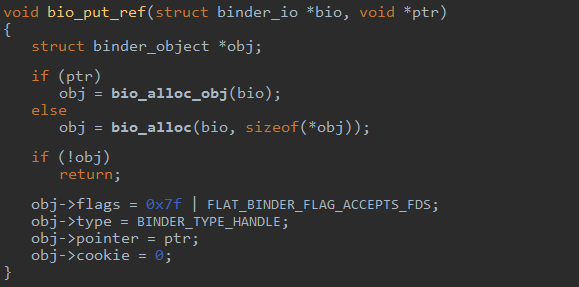
从字面意思可以看出该函数的主要作用是把上面提到的Binder引用放到reply这个返回的结构体中，这个结构体的类型也是

**struct binder\_io**, 该结构体曾在binder\_parse()方法中被初始化过。

bio\_init(&reply, rdata, sizeof(rdata), 4);

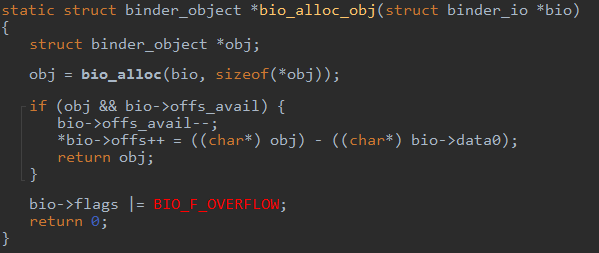
其data所指向的数据指针是一个unsigned[64]类型的数组，且已经偏移了16位

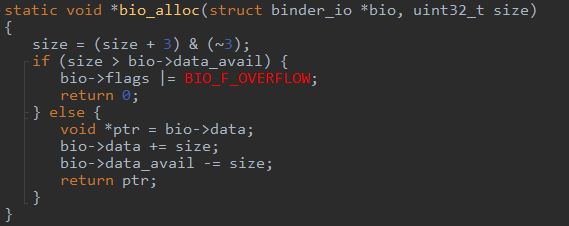
下面看bio\_put\_ref函数



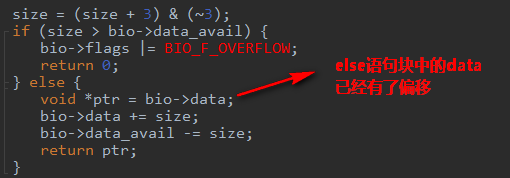
由于我们传过来的Binder引用不为空，所以函数接下来会调用

obj = bio\_alloc\_obj(bio);方法

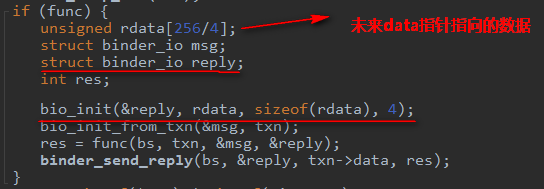


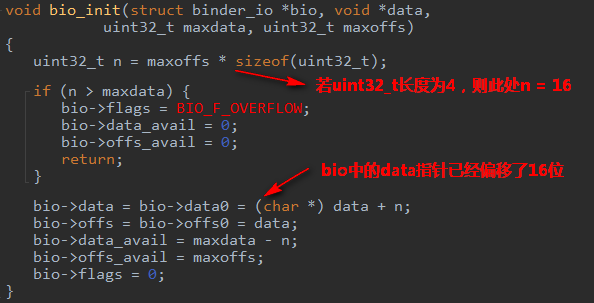


可以看到辗转到了bio\_alloc()方法，该方法和我们之前分析过的bio\_get\_uint32()方法类似，不过需要注意的是：



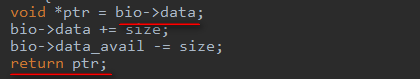
bio->data, 它所指向的data已经发生了偏移，偏移为16位，详情参见binder\_parse()中的bio\_init()方法



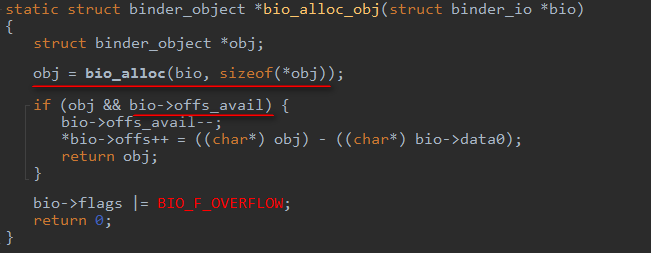


同时，此时的data\_avail的值为：maxdata-n,也就是256-16 = 240;

回到刚才的bio\_alloc()方法，其目的就是返回data当前的指针，也就是这个已经偏移了16位的指针。



然后回到bio\_alloc\_ojb()方法中



此时返回的obj肯定不为NULL，bio->offs\_avail上面提到过，当前为maxoffs，也就是4，因此接下来就执行if块内的语句：

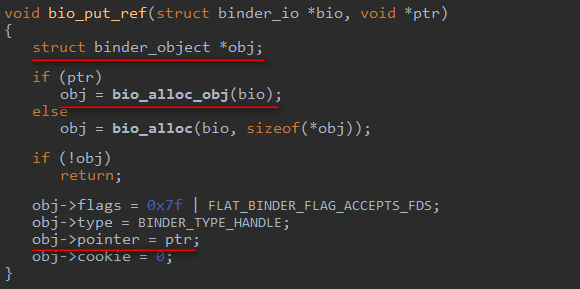
bio->offs\_avail--;

\*bio->offs++ = ((char\*) obj) - ((char\*) bio->data0);

然后再将obj返回。

通过上面的分析可以看出：目前只是在reply的这个binder\_io结构体中的data域上找到了一个位置来存放内容，但是到目前为止，还没有真正的去存放内容。

最后我们再回到bio\_put\_ref()方法中



可以看到，当拿到该指针后，接下来就正式为该指针赋值，函数调用了

obj->pointer = ptr;

正式将Binder引用放到了用于返回的reply结构体中。然后return 0;

至此，SM完成了它的工作，Binder Clinet向它查询名为xxx的Binder Server的Binder引用，SM找到后将该引用放到名为

reply的binder\_io结构体中，返回给Binder驱动，Binder驱动再将该引用返回给Binder Client。

下一步就是要把该结果返回给Binder驱动，由驱动程序将该结果反馈给Binder Client。

1. **调用binder\_send\_reply()函数将查询结果返回给Binder驱动**

我们回到binder\_parse()方法中，继续向下执行：

这下到了binder\_send\_reply()函数中

C:\Users\xp.chen\Desktop\2018-09-05_144321.png



该函数一共需要接收4个参数：

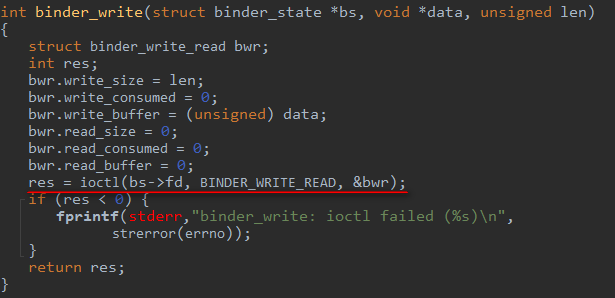
struct binder\_state \*bs // 描述Binder驱动设备信息的结构体

struct binder\_io \*reply // 由SM反馈的包含查询结果的binder\_io结构体

void \*buffer\_to\_free // txn->data, 不清楚。猜测可能是之前Binder驱动传给SM的数据指针

int status // SM程序处理Binder驱动请求的结果，上面SM处理完后,return了一个0

详细的内容就不分析了，就是封装了一个结构体，然后调用binder\_write()函数。



可以看到，最终还是调用了ioctl()函数将结果发送给了Binder驱动。

4.5.3 ServcieManager的具体使用

从上面的分析来看，要想从SM这里拿到对应Binder Server的Binder句柄，可真够麻烦呀，最起码需要经历以下4个步骤：

1. 打开Binder设备；
2. 执行mmap;
3. 通过Binder驱动向SM发送请求(SMr的handle为0)；
4. 获得结果。

远的不说，要是每个Binder Client都要这样搞才能获得Binder 引用，那这代码也够恶心的了，而且还不能保证代码是否正确。另外，要是每次使用SM服务或其它Binder Server服务都要打开一次Binder驱动，执行mmap，后果就是消耗的系统资源会越来越多，直到崩溃。

一个有效的办法就是：**每个进程只允许打开一次Binder设备，且只做一次内存映射，所有使用Binder驱动的线程共享这一资源**。

那如何改进？

Android 的做法：

1. **建立一个专门的类来管理每个进程中的Binder操作**

既然与Binder之间的操作如此繁琐，且容易出错，那很容易想到可以建立一个专门的类来负责进程中的Binder操作，而且这个类执行Binder驱动的一系列命令对上层用户还是“透明”的。该类就是——**ProcessState**。另外进程中的每一个线程也应该有与Binder驱动沟通的权利，且基于Binder的IPC是阻塞的，在线程中与Binder驱动沟通则不会卡死整个应用程序。

与Binder驱动进行实际命令通信的是**IPCThreadState**。

以上两个类都是C++版的实现，Java层要使用的话就必须借助另一个类——**BpBinder**

**二．建立一个代理类封装SM提供的各项服务**

上述的两个类封装了与Binder驱动交互的动作，但是我们最终的目的是从SM获得或是添加Binder的引用，也就是给SM发送“SVC\_MGR\_GET\_SERVICE”,“SVC\_MGR\_ADD\_SERVICE”这些指令，让SM能够响应。因此最理想的写法就是SM.getService()这样一行代码搞定就可能拿到或添加Binder引用，为了简化SM的使用流程，最好就是给SM添加一个代理。这个代理和SM提供了相同的接口，但是却封装了SM的内部实现，使用这个代理能够达到和使用SM本身一样的效果。

这个代理本身要提供和SM相同的接口，这个接口就是——**IServiceManager**

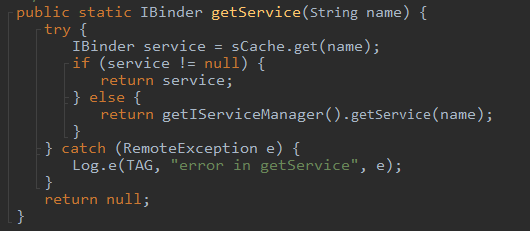
使用这个代理要保证和使用SM一样，这个代理就是——**ServiceManagerProxy**

事实上Android又是ServiceManagerProxy又做了一层封装，使得SM更加易用。 这个类就是——**ServcieManager**，它提供了SM所有的服务接口，且它的这些服务接口全是static的。

下面从ServiceManager的getService()函数开始分析，看看Android是如何封装SM的。

1. **首先进入getService()函数**

frameworks/base/core/java/android/os/ServiceManager.java

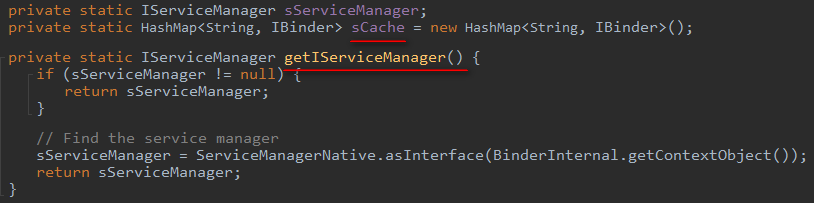


这个函数很简单，先从sCache(sCache是一个HashMap)是去找指定name的Bnder引用，如果有就直接返回,没有就先调用

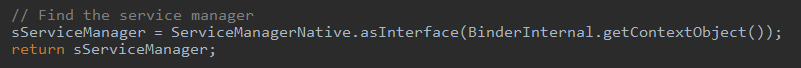
getIserviceManager().返回一个IServiceManager对象，再调用getService()方法，返回Binder引用。

我们先去看getIServiceManager()函数。

1. **进入getIServiceManager()函数**



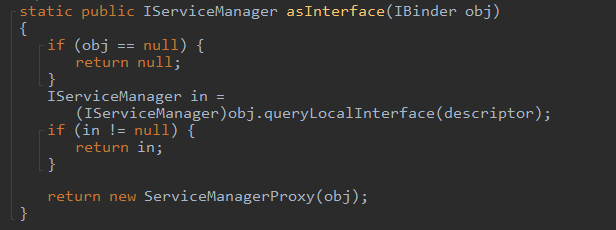
该函数同样很简单，先判断本地是否存在IServiceManager的全局变量，如果没有就继续调用下面这段代码：



这里的IBinder对象是通过BinderInternal.getContextObject()得到的，我们先不深究其内部实现，把它当成与底层Binder沟通的工具即可。我们直接绕过它，看这个asInterface（）方法

1. **进入到ServiceManagerNative.asInterface()函数**

frameworks/base/core/java/android/os/ServiceManagerNative



这里假设传入的Binder引用不为空，那么它是通过在Binder引用上调用querylocalInterface(descriptor)来得到SM对象的，可能有人对IServiceManager接口存在疑惑，这里说明一下，这个IServiceManager就是我们上面提到过的：定义了SM代理类需要具有服务函数的接口：打开IServiceManger，发现它果然定义了如下接口方法：

public IBinder getService(String name) throws RemoteException;

public IBinder checkService(String name) throws RemoteException;

public void addService(String name, IBinder service, boolean allowIsolated)

throws RemoteException;

public String[] listServices() throws RemoteException;

static final String descriptor = "android.os.IServiceManager";

因此，这里的**IServiceManger对象就可以理解成是SM的代理类**。

上面的obj.querylocalInterface()函数要求传入一个descriptor，虽然还没有分析这段代码，但是大概可以猜出该函数的作用：

该函数用于通过Binder驱动向底层的SM程序查询SM的Binder引用是多少，我们之前在分析SM时也看到过：**要查询一个Bnder Server的Binder引用，必须给出这个Binder Server的descriptor**。

若没有查询到，则新建一个ServiceManagerProxy,因为作为SM的代理程序，必须要与Binder驱动进行通信，所以在ServiceManagerPorxy的构造函数中传入了IBInder对象。

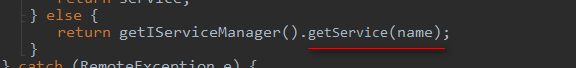
1. **查看ServiceManagerProxy的构造函数**

frameworks/base/core/java/android/os/ServiceManagerNative

C:\Users\xp.chen\Desktop\2018-09-05_144321.png

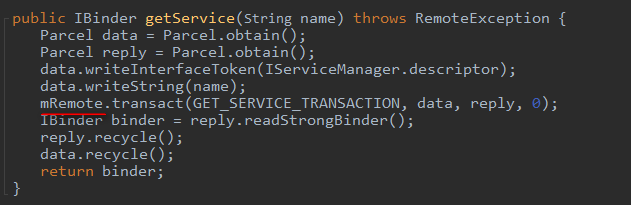
这样SM的代理程序ServiceManagerProxy就持有了IBinder对象，未来如果有需要，可以随时使用这个IBinder对象。

到此，我们总算拿到了SM的代理对象ServiceManagerProxy,不过我们的目的是拿到目标Binder Server的binder引用，也就是调用getService()后返回的那个东西，我们的代码还没有分析完。



从上面可以看出，这个getIServiceManager()最终就是那个**ServiceManagerProxy**,调用getService()其实就是调用这个ServiceManagerProxy上的方法。

我们打开ServcieMangerProxy类上的getService()方法

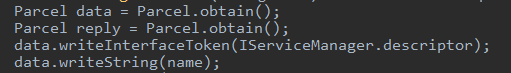


这个mRemote就是刚才通过构造函数传入的IBinder引用。

从这个函数可以看出，上层通过getServcie()方法向SM获取Binder引用主要分为以下几个步骤：

1. **封装Parcel**

也就是通过Parcel打包数据。具体代码实现为：



**二 调用IBinder.transact**

利用IBinder.transact将请求发送出去，而不用理会Binder驱动的open, mmap以及一大堆具体的Binder协议中的命令。所以这个IBinder在内部一定会使用ProcessState和IPCThreadState来与Binder驱动进行通信。

**三．获取结果**

调用完transact之后，我们就可以直接获取结果了。需要注意的是上面的代码在调用transact()方法后没有拿到结果前，代码会一直阻塞，类似于Socket，而一旦拿到结果之后就可以从结果上拿到所需的IBinder对象，最后再将Parcel recycle掉，整个过程结束。

不过有一点还需要注意，那就是给SM发送命令的ID号必须一致。上面的命令ID号是GET\_SERVICE\_TRANSACTION

C:\Users\xp.chen\Desktop\2018-09-14_151444.png

C:\Users\xp.chen\Desktop\2018-09-14_151444.png

也就是0x00000001,

而在service\_manager.c中定义的用于接收的命令ID中：

enum {

SVC\_MGR\_GET\_SERVICE = 1,

SVC\_MGR\_CHECK\_SERVICE,

SVC\_MGR\_ADD\_SERVICE,

SVC\_MGR\_LIST\_SERVICES,

};

可以看到SVC\_MGR\_GET\_SERVICE刚好也为1，这样上层的命令业务码就和底层处理该命令时所要接收的业务码保持一致。

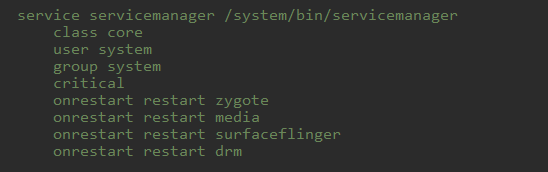
PS：SM这个Binder Server比较特殊——它主动提供了上面这个enum结构来定义业务码。通常情况下，**使用AIDL产生的Binder Server会自动生成这些业务码，不需要手动编写**。

## 4.5 DNS服务器—ServiceManager[Binder Server]

### 4.5.1 ServiceManager的启动

ServiceManager充当了DNS的作用，所以必须在使用Binder机制前就应该处于正常的工作状态。事实确实如此，其在init程序解析init.rc文件时就已经启动了。

**system/core/rootdir**



与ServiceManager相关的文件位于：

**frameworks/native/cmds/servicemanager**目录中：

根据该目录下的Android.mk文件提示，核心文件应该是service\_manager.c和binder.c

LOCAL\_PATH:= $(call my-dir)

#include $(CLEAR\_VARS)

#LOCAL\_SRC\_FILES := bctest.c binder.c

#LOCAL\_MODULE := bctest

#include $(BUILD\_EXECUTABLE)

include $(CLEAR\_VARS)

LOCAL\_SHARED\_LIBRARIES := liblog

LOCAL\_SRC\_FILES := **service\_manager.c binder.c**

LOCAL\_MODULE := servicemanager

include $(BUILD\_EXECUTABLE)

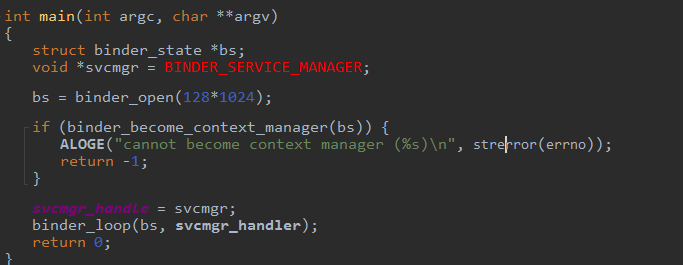
PS: 源码工程里还有很多ServiceManager.cpp文件存在，但并不属于SM程序。

### 4.5.2 ServiceManager的构建

其中的ioctl函数用于向设备发送控制和配置命令，也可以用该命令读取终端设备的一些数据。

首先找到ServiceManager的启动入口：

**frameworks/native/cmds/servicemanager/service\_manager.c**





从截图中可以看到，首先调用了**binder\_open()**函数打开了Binder设备，传入的参数是进行内存共享映射时，映射的内存块长度。

biner\_open()函数位于：

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



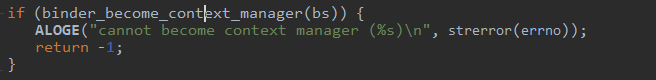
这个binder\_open()方法很重要，它返回了一个包含binder驱动信息的结构体指针。这个结构体所包含的内容包括：

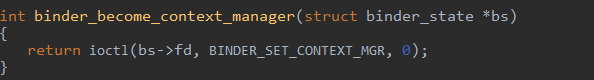
Binder驱动的设备文件描述符fd // 用于将来通过ioctl()函数向Binder设备驱动发送各种消息。

Binder 驱动的内存映射首地址 // 不知道要这个干啥

Binder 驱动的内存映射长度 // 也不知道要干啥

这里还需要注意一点的就是：这段内存的映射是私有的：**MAP\_PRIVATE**，也就是说对映射区的改变不需要保存到文件,也就不会影响到Binder驱动。





这一行代码的主要作用是通过ioctl()函数向Binder驱动发送“**BINDER\_SET\_CONTEXT\_MGR**”消息将该进程本身注册成为ServiceManager.,其中的fd就是Binder驱动的设备文件描述符。因为**整个Android系统中只允许有一个ServiceManager存在，所以再次向Binder驱动发送此消息，会返回错误**。

**重点**：接下来调用binder\_looper()函数开始从Binder驱动读取和写入消息

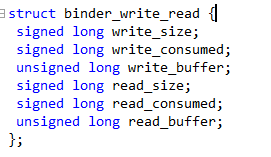


这里面有个地方很奇怪，

unsigned readbuf[32];

bwr.read\_buffer = (unsigned)readbuf;

但是通过查看NDK中的bind.h头文件，发现：



read\_buffer的类似却是unsigned long类型的。这里的结构体中的read\_buffer其实保存的是readbuf这个数组的指针(**数组名就是指针**)，但是由于在

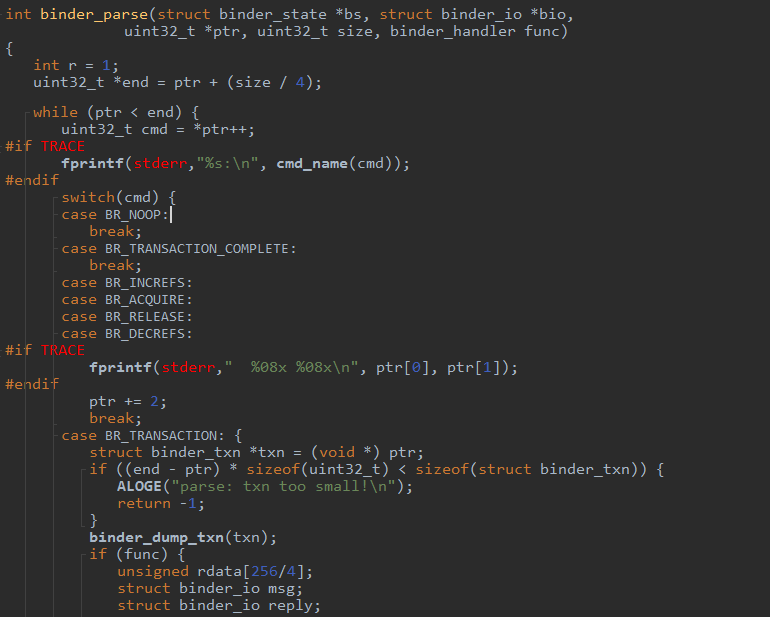
32位和64位系统下指针长度不一致(32位指针长度为4，64位指针长度为8)，所以使用了unsigned long类型来保存这个数组的指针。

紧接着通过向Binder驱动发送BINDER\_WRITE\_READ消息来实现读取和写入。这里有几点需要注意：

1. ServiceManager进程中并没有消息队列，它的“消息”是从Binder驱动那里获取的；
2. struct binder\_write\_read这个结构体是执行BINDER\_WRITE\_READ命令时所必须的数据格式。详情可参考《Binder协议所要求的数据格式》；
3. 究竟ServiceManger进程是向Binder驱动中读取还是写入消息取决于bwr.read\_size和bwr.write\_size。若write\_size初始值是0，但read\_size为

sizeof(readbuf),则Binder驱动只执行读取操作。

接下来进入**binder\_parse**()方法：



首先通过

uint32\_t \*end = ptr+(size/4);计算出数据的终点。这个方法传入的size是：

C:\Users\xp.chen\Desktop\2018-08-13_144204.png

目前尚不清楚binder驱动返回来后，这个size值是几

接下来开始进入while循环，先取出cmd

C:\Users\xp.chen\Desktop\2018-08-13_144204.png

这里需要注意的是：一个cmd占用一个uint32\_t, 因而取得cmd后跳过这段空间。每个cmd都代表程序接下来要处理的具体命令，不过有些命令不需要特别处理：

如：

BR\_NOOP,BR\_TRANSACTION\_COMPLETE

上面的代码涉及到一些Binder驱动相关的知识，不过大概的作用就是为了**从Binder驱动中拿到消息，然后调用bind\_parse()方法来处理消息**。

1. 传统IPC通信的弊端

性能、稳定性、安全性

1. Android为什么要采用Binder作为IPC通信？

Android解决了哪些问题

1. Android是如何实现的？Binder的架构设计对比传统IPC通信有哪些优势？

架构图对比

传统IPC通信(pipe/fifo)须经历两次拷贝(数据先从发送端的内存缓冲区拷贝到内核的内核缓冲区，再由内核缓冲区拷贝到接收端的内存缓冲区)，Binder只需要经历一次拷贝

Binder更像是（从发送端的内存缓冲区拷贝到内核的数据缓冲区，再由内核的数据缓冲区mmap到接收端的内存缓冲区）

四. Binder驱动的引入

首先跨进程通信肯定需要需要内核空间做支持。因为两个进程分别处于不同的虚拟地址空间。Binder又不是Linux系统内核的一部分，如何实现与内核交互

跨进程通信是需要内核空间做支持的。传统的 IPC 机制如管道、Socket 都是内核的一部分，因此通过内核支持来实现进程间通信自然是没问题的。但是 Binder 并不是 Linux 系统内核的一部分，那怎么办呢？这就得益于 Linux 的**动态内核可加载模块**（Loadable Kernel Module，LKM）的机制；模块是具有独立功能的程序，它可以被单独编译，但是不能独立运行。它在运行时被链接到内核作为内核的一部分运行。这样，Android 系统就可以通过动态添加一个内核模块运行在内核空间，用户进程之间通过这个内核模块作为桥梁来实现通信。在 Android 系统中，这个运行在内核空间，负责各个用户进程通过 Binder 实现通信的内核模块就叫 **Binder 驱动**（Binder Dirver）。

五. Binder改进了通信方式，使用mmap替代了传统的拷贝方式，但是mmap通常需要物理介质文件，Binder又如何解决？

Binder并未引入任何物理介质，而是在内核空间新建了一个**数据接收缓冲区**。建立**内核缓冲区**和**内核中数据接收缓冲区**之间的映射，

以及**内核中数据接收缓冲区**和**接收进程用户空间地址**的映射。通过此双向映射实现进程间通信。

六．底层实现

七. 如何与Binder驱动进行交互

通过open(),mmap(),.ioctl()来访问设备文件/dev/binder, 它们之间约定好了一个Binder协议，协议基本格式是(命令+数据)，详情可参考：

[Android Binder设计与实现](https://blog.csdn.net/universus/article/details/6211589)

=======================================================

如何设计一个良好的Binder Client与Binder驱动交互？

1. ServiceManagerProxy的引入

ServiceManagerProxy的主要作用用于：对SM提供的服务进行封装。封装的必要性在于：如果此时应用程序要与SM进行通信，原则上还需要通过向Binder驱动发送BINDER\_WRITE\_READ等Binder支持的命令与其交互，从而进一步得到SM提供的服务。

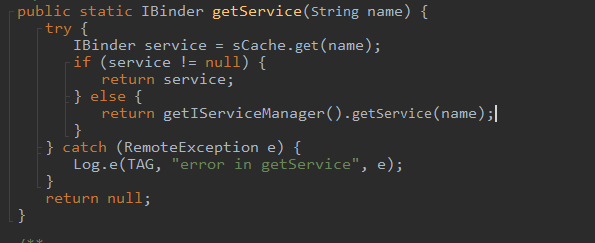
而**ServiceManagerProxy类的引入，则可以简化整个流程，直接调用这个类，就可以获取SM提供的各项服务**。

Android对ServcieManagerProxy又做了一层封装，即：**ServiceManager**.java,这样应用程序使用SM就更加方便了：

ServiceManager.getService(name);

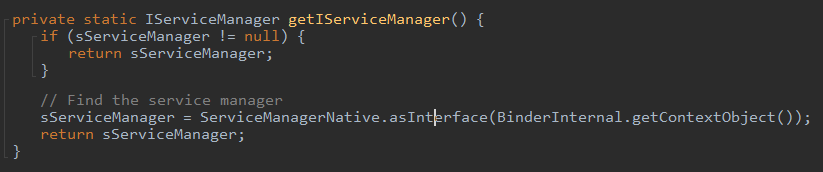
1. 通过getService()方法，返回了一个IBinder对象，这个IBinder对象又是个什么东西？

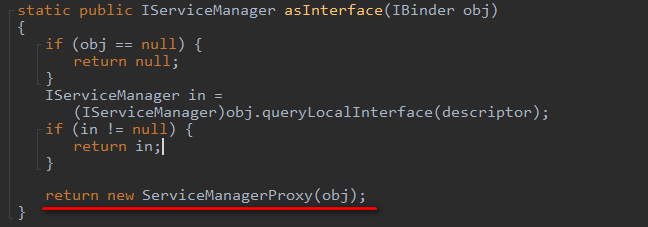
从代码中可以看出：这个IBinder对象是有可能为NULL的，如果为NULL的话，又尝试从getIServiceManager().getService()上去获取



1. 那么这个getIServiceManager()又是什么东西呢？

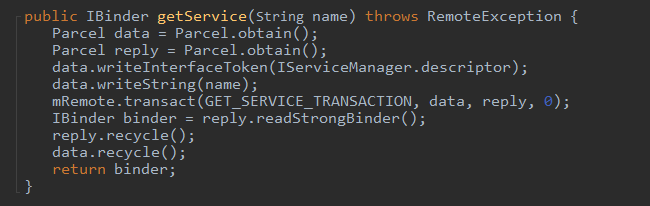
从代码上看，它是一个实现了IServiceManager接口的对象。





再通过打开asInterfadce（）方法，发现IServiceManager接口指向的正是SM的代理对象：**ServiceManagerProxy。**

这个类既然是SM的代理对象，那它必然也能通过service的名字，获取到对应的IBinder引用。具体获取的步骤为：



上面的代码其实就是Java层获取Service的核心代码

先获取一个Parcel对象，再给Parcel对象设置一些数据，最后利用mRemote将Parcel对象发送出去。最后获取结果。

这里的做法很简单，先给进程间通信的载体——Parcel对象身上设置了InterfaceToken和string这两个东西，再通过某个类**mRemote**将Parcel发出去。查看一下mRemote，发现竟然又是IBinder对象！那这个IBinder对象到底是什么东西？

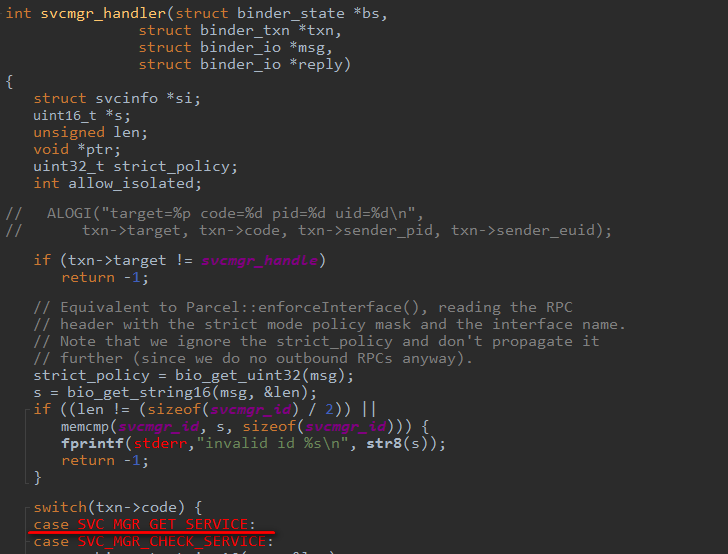
这里还有一点需要特别留意一下：

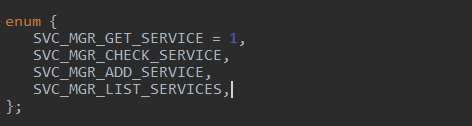
mRemote.transact(GET\_SERVICE\_TRANSACTION, data, reply, 0);

getsService发送出去的业务码为GET\_SERVICE\_TRANSACTION，也就是1，因为它是用来获取服务的，所以猜测SM中必定有对1这个业务码的处理：

事实上也确实是这样：

**frameworks/native/cmds/servciemanager/service\_manager.c/svcmgr\_handler()**



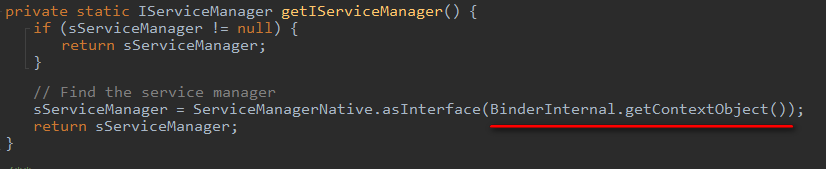


这个**SVC\_MGR\_GET\_SERVICE**的枚举值恰好是1

值得一提的是：SM这个Binder Server主动提供了上面这个enum结构来定义业务码。通常情况下，使用AIDL产生的Binder Server会自动生成这些业务码，而不需要手工编写。

1. IBinder和BPBinder

IBinder这种东西，目前看来似乎能够与Binder驱动进行通信。而且这个接口最初是在获取ServcieManagerProxy的过程中出现的。当时，它是通过下列代码获取到的。



就是BinderInternal.getContextObject(),那这个BinderInternal又是什么东西呢？

可以看到，IBinder这个东西起初是通过getContextObject()方法获取的。

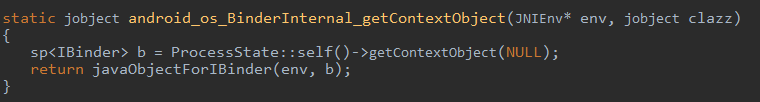
**frameworks/base/core/java/com/android/internal/os/BinderInternal.java**

C:\Users\xp.chen\Desktop\2018-08-14_154443.png

这个方法是一个Native方法。因为要与Binder驱动打交道，最终都得通过JNI调用本地代码来实现。这样我们还得参考JNI层的BinderInternal。

其对应的JNI层的代码位置在:

framworks/base/core/jni/android\_util\_Binder.cpp



IBinder只是一个接口类，还会有具体的实现类继承于它，在Native层，就是BpBinder(BpBinder.cpp),而在java层，则是Binder.java中的BinderProxy。

* BinderInternal.getContextObject(). // 获取一个用于通信的IBinder对象。具体实现在android\_util\_Binder.cpp中
* android\_os\_BinderInternal\_getContextObject; // 向java层返回一个IBinder对象
* 在android\_os\_BinderInernal.\_getContextObject方法中通过调用javaObjectForIBinder()来返回一个IBinder对象。// **BinderProxy**和**BpBinder**分别继承于Java层的Native层的IBinder接口。这里通过javaObjectForIBinder返回的其实就是一个BinderProxy对象。
* 也就是说在ServiceManagerProxy类的getService()方法中调用了transact方法进行通信的实现IBinder接口的那个类就是**BinderProxy**.

接下来看一下BinderProxy类中的transact()方法：

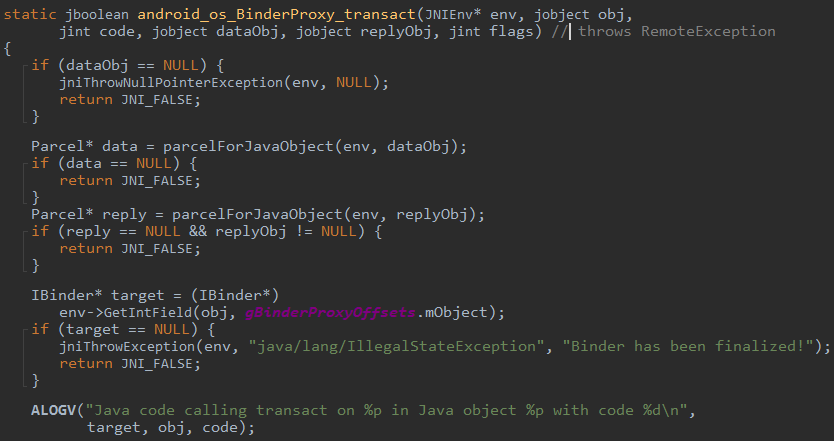
注：BinderProxy是Binder（android.os.Binder）的一个内部类。

ServiceManagerProxy是ServiceManagerNative（android.os.ServiceManagerNative）的一个内部类。

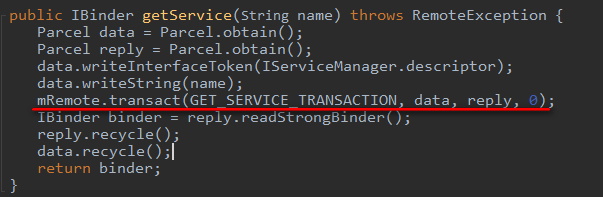
C:\Users\xp.chen\Desktop\2018-08-16_100433.png

这个方法是一个native方法。其对应的native方法在：

**android\_util\_Bnder.cpp:: android\_os\_BinderProxy\_transact**



从参数可以看出：它与ServiceManagerProxy类中的transact方法传递的参数完全一致。

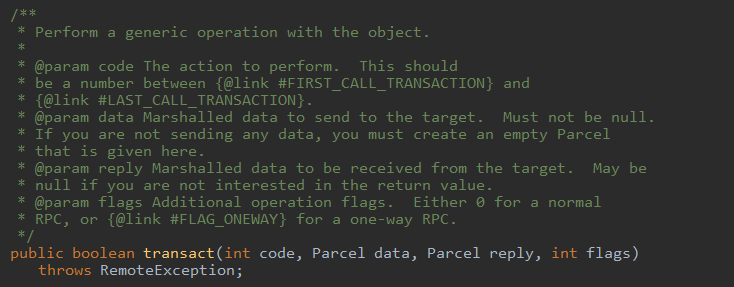


第一个参数是与Binder驱动通信的业务码；

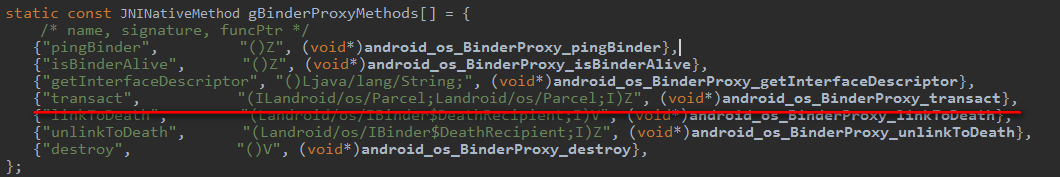
第二个参数是发给Binder驱动的Parcel对象（Parcel是约定格式）。

第三个参数是未来Binder驱动返回的Parcel对象。

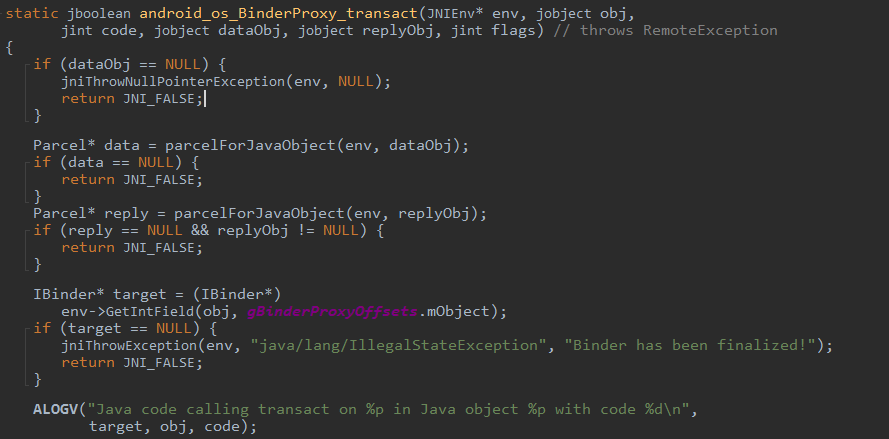
第四个参数可选，可以是0或者是1



然后接着回来看android\_util\_Binder.cpp中的android\_os\_BinderProxy\_transact()方法的内容。



从JNI的注册函数表中可知，其对应的是android\_os\_BinderProxy\_transact()这个本地方法，这个JNI方法的返回值是Z，也就是jboolean类型。



首先对传入的Parcel对象进行判断，如果对象为NULL，则向上层抛出一个异常：jniThrowNULLPointerException

注意：这个文件包含了JNIHelp.h头文件。jniThrowNULLPointerException函数就定义在该头文件中。

该头文件位于：AOSP/libnativehelper/include/nativehelper/JNIHelp.h

实际文件位于: AOSP/libnativehelper/JNIHelp.cpp

JNI GetIntField

◆ Q：如何获得一个IBinder对象？

A：通过BinderInternal类的getContextObject()方法。该方法是一个native方法,返回一个IBinder对象。

◆ Q：从Native层获取到的IBinder对象到底是个什么东西？

A：是一个继承Native层IBinder类的一个派生类，本质上是BpBinder

◆ Q：BpBinder是如何与Binder驱动通信的？

A：调用BpBinder.cpp的transact()方法，最终会调用IPCThreadState的transact方法。

ContextImpl frameworks/base/core/java/android/content/ContextImpl

1. startActivity(), startService(), bindService(), sendBroadcast()都是进程间通信的一种形式。
2. bindService()是ContextWrapper类上的一个方法。
3. ServiceManager对外提供的服务接口是IServiceManager。**接口描述了一个Binder Server能够提供的服务**。客户端只能使用Binder Server提供的这些服务。举例：客户端使用的ServiceManagerProxy和ServcieManagerNative都继承IServiceManger，具体服务端的service\_manager.c则实现了这些接口。另外Android里面Binder Server提供的接口均**以大写字母“I”打头**。
4. 其它进程访问某个Binder Server的方式。常见的途径是**调用者需要通过ServiceManager.getService(SERVICE\_NAME)先获取到这个Binder Server的本地代理，然后利用这个本地代理再与之通信**。
5. 其它进程访问某个Binder Server的另一种形式。还可以通过一个已经在ServiceMnager注册的Binder Server来访问另一个Binder Server.
6. Android 给开发人员提供了一个快速创建 Binder Server的方式—— **AIDL**。AIDL的其中一个作用就是可以保证接口一致性。
7. AIDL中的所有服务接口均继承于Iinterface, 然后在此基础上声明与此Server相关的方法。

**以IWindowManager.aidl举例分析AIDL的实现原理**

目录：**framworks/base/core;/java/android/view/IWindowManager.aidl**

*Question:*

1. ProcessState::self()的实现是什么鬼？有点像单例的味道。
2. android\_util\_Binder.cpp中的android\_os\_BinderInternal\_getContextObject()方法。(向java层返回一个IBinder对象)
3. android\_util\_Binder.cpp中的android\_os\_BinderProxy\_transact()方法（与Binder驱动通信）。
4. ContextWrapper和ContextImpl为什么要这么写？装饰者模式