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

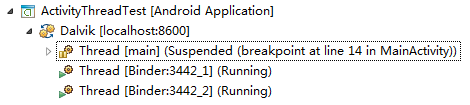
<略>

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

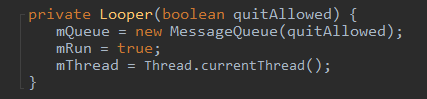
将待执行的Runnable任务包装成一个Message对象，再压入到MessageQueue中。

3.2.2 Handler指定类来处理接收到的消息

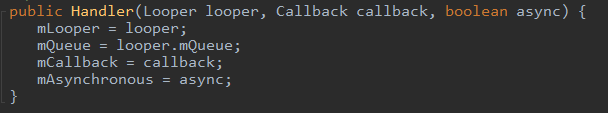
Handler可以通过在构造函数中传入一个自定义的CallBack来处理接收到的消息。该CallBack必须实现Handler内部的CallBack接口,并在重写的handleMessage(Message msg)中处理相关消息。

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

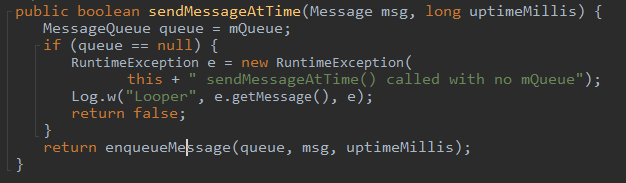
◆A) Looper本身关联了一个MessageQueue(mQueue,通过构造函数创建)，

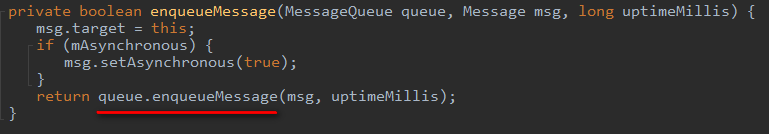


◆B) Handler需要在构造函数时传入Looper,通过传入的Looper,可以拿到Looper身上绑定的MessageQueue

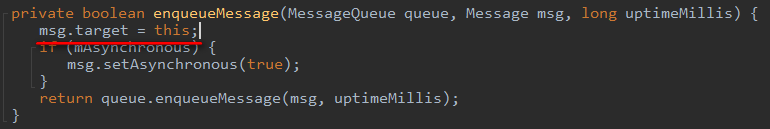


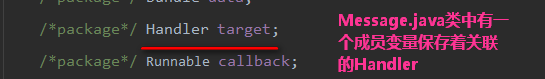
◆C) 这样Handler就可以在后续投递消息时(sendMessageAtTime)将消息投递到mQueue,也就是mLooper.mQueue中。





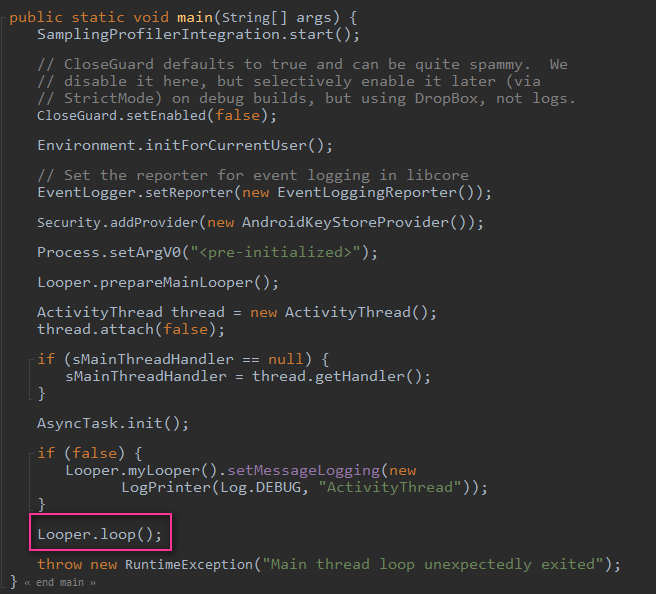
◆D) 接下来，一旦Looper处理到MessageQueue中的某个Message,就又会调用该Message身上绑定的Handler来处理



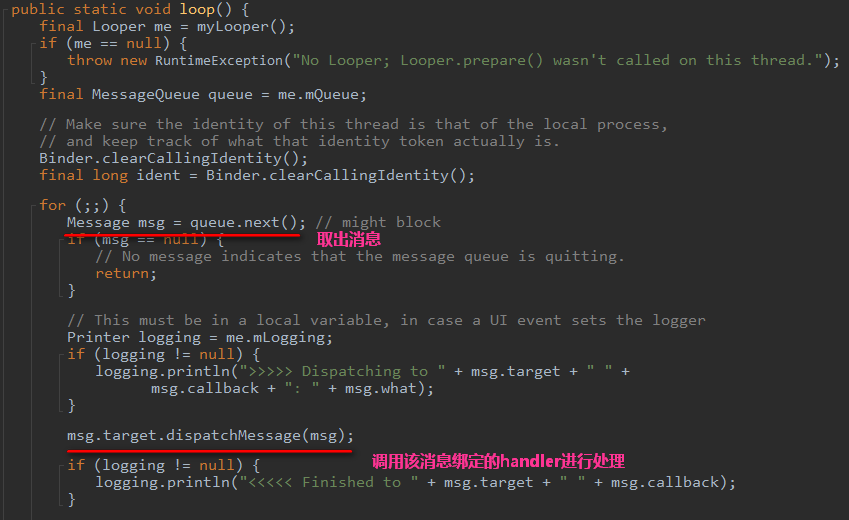


3.2.4 ActivityThread中消息的处理流程

**ActivityThread.java**



**Looper.java**

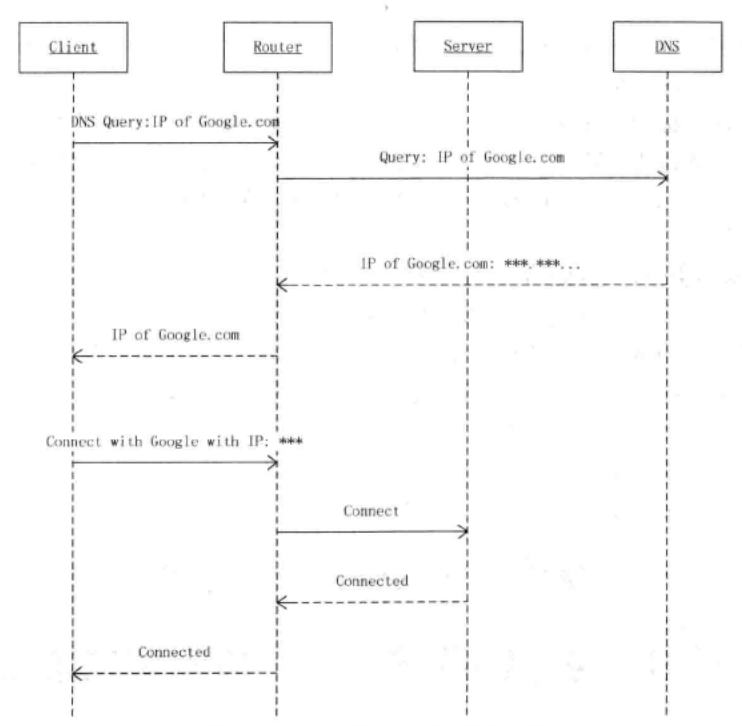


1. 进程间通信-Binder

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**

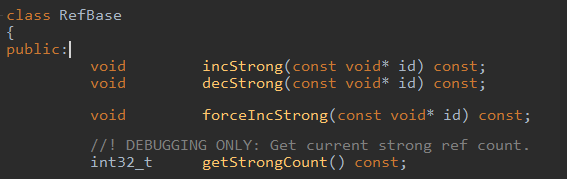
通过分析wp的源码，可以发现wp和sp在类定义上有如下重要区别：

◆ 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)**

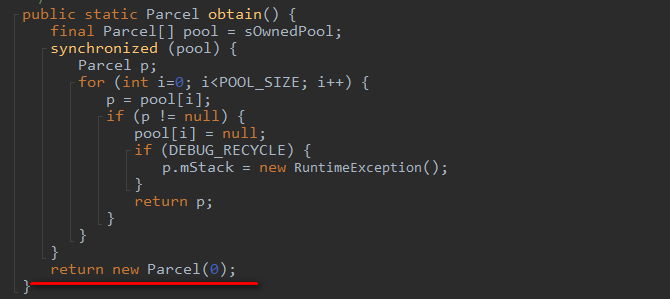
Parcel是进程间通信中数据传递的载体，用于承载希望通过IBinder发送的相关信息，Parcel可用来存储以下几种类型数据：

* **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

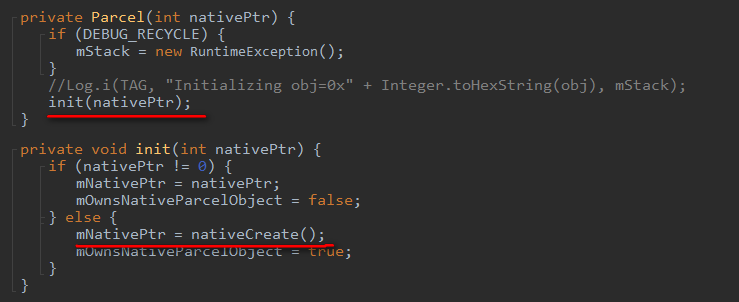
1. **首先通过Parcel.obtain()接口来获取一个Parcel对象。**

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。

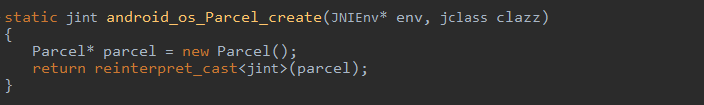
1. **调用本地方法nativeCreate()创建一个native对象，并将创建完的指针地址返回给Java层。**



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

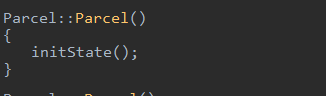


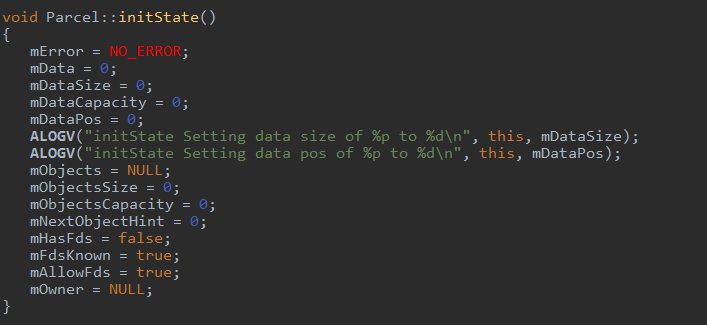
这里创建了一个Parcel对象，并将Parcel对象的指针返回。将Parcel指针转换成jint类型返回给java层。

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

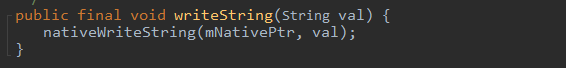
mutable size\_t mDataPos; // 数据指针

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**

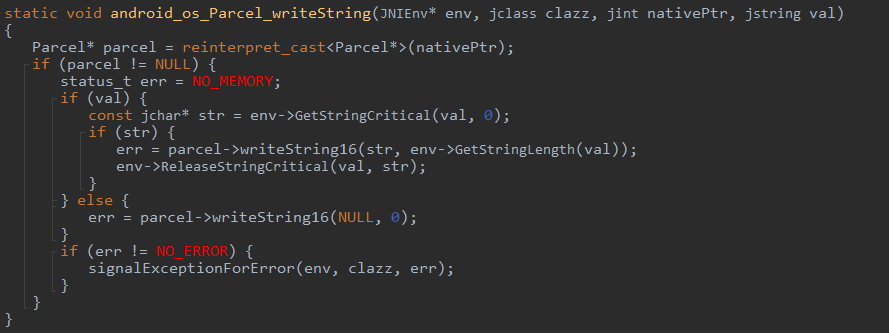
frameworks/base/core/java/android/os/Parcel..java



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：

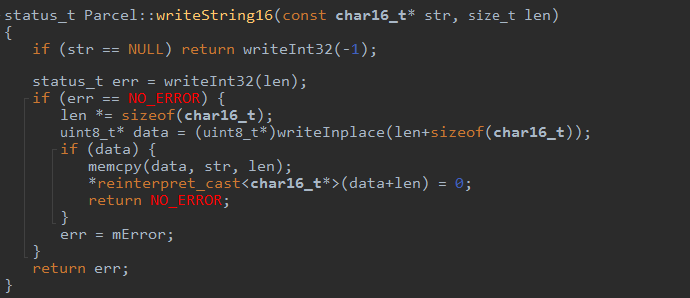
Parcel\* parcel = reinterpret\_cast<Parcel\*>(nativePtr);

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。

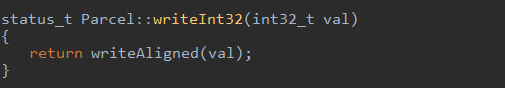
1. **调用Parcel类的writeString16方法对字符串进行写入**

frameworks/native/libs/binder/Parcel.cpp



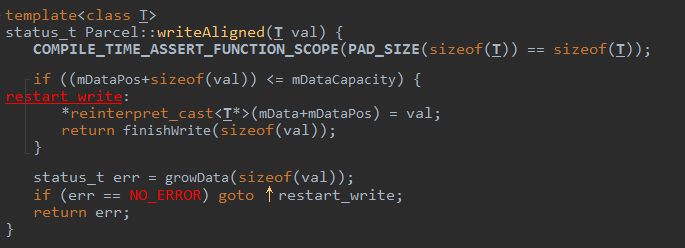
该方法被要求传递两个参数：一个是需要写入到Parcel的字符串指针，另一个则是该字符串的长度。当检测到传递过来的字符串不为NULL后，则先调用writeInt32方法写入字符串长度

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)

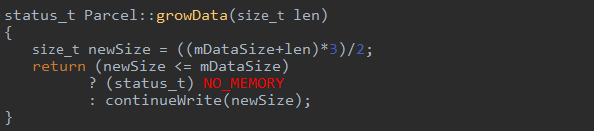
或者百度搜索：编译期ASSERT

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

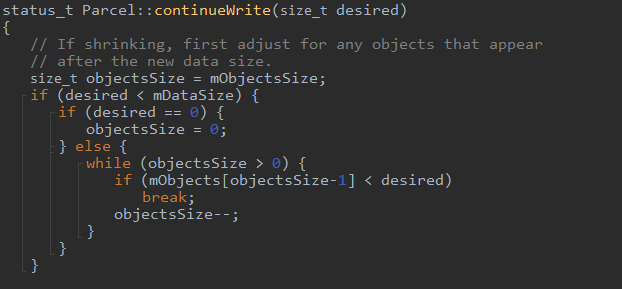
1. **若将要写入Parcel的数据长度超过了Parcel的最大存储能力，就调用growData方法扩充Parcel的存储能力**



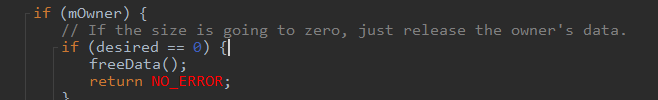
这里首先计算出一个newSize,这个newSize目前等于6：**（(0+4)\*3）/2**

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

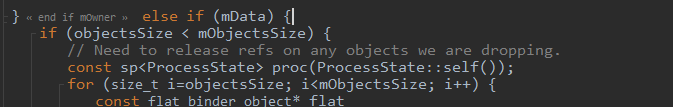
接下来分析continueWrite()方法，该方法比较长，但有效代码很短。



参数desired此时为6，大于mDataSize代码块不执行。

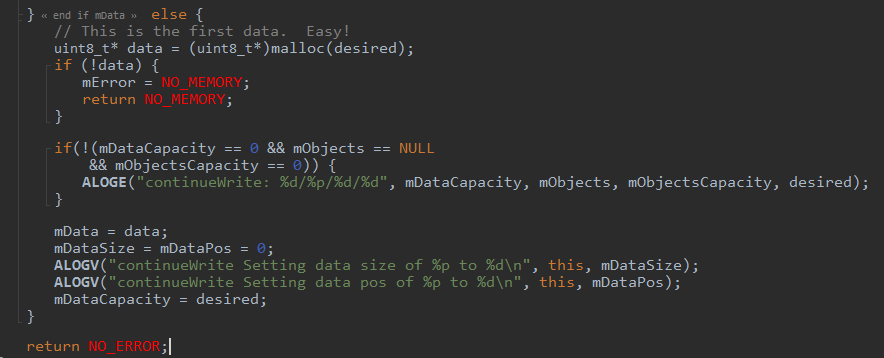


mOwner在initState方法中先初始化为NULL,代码不执行。



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;

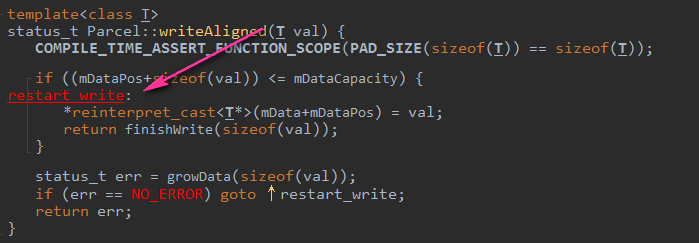
2.**将mDataSize和mDataPos置为0**。mDataSize = mDataPos = 0, 猜测可能是因为还没有开始写数据

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：



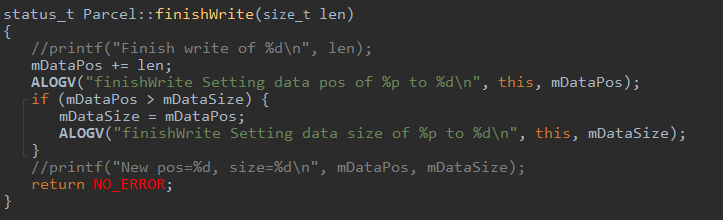
然后调用

\*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**



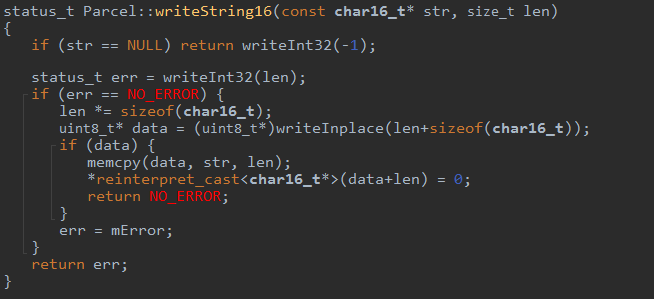
这里主要两件事情，分别是:

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

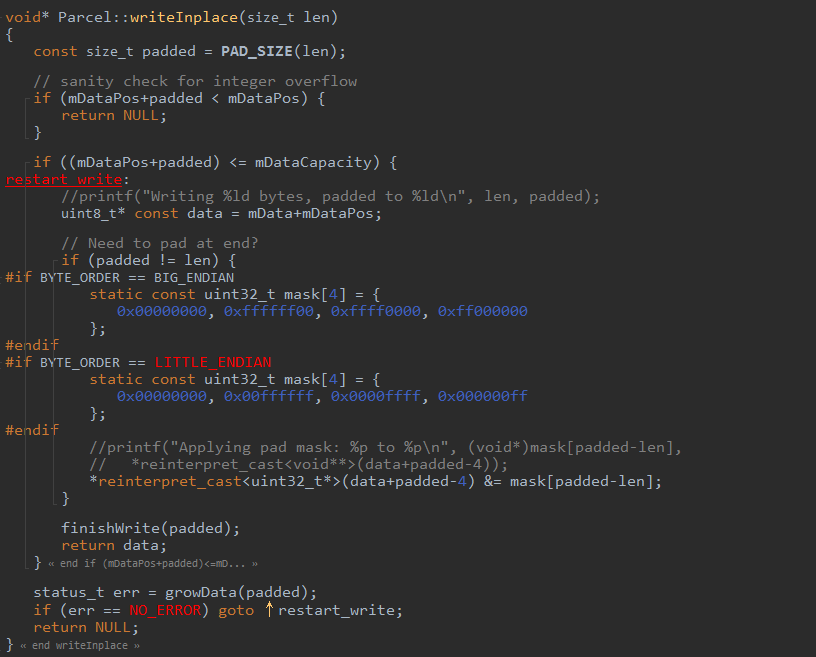
注意: 这里可能受到了Unicode字符长度的影响，JNI层获取字符串长度时调用的方法是GetStringLength()而不是GetStringUTFLength()方法

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;

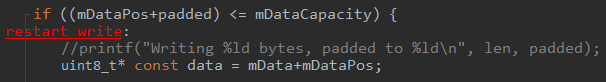
然后判断

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



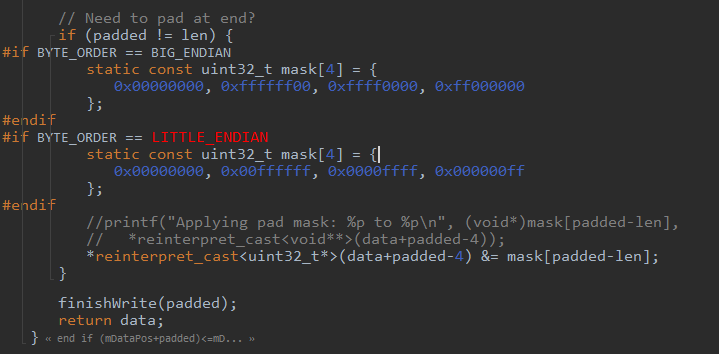
mDataPos此时为4，而4+padded很明显也要大于mDataCapacity(当前为6),所以这个代码块也不执行。当发现Parcel的数据存储能力又无法满足要求时，最终又调用growData()方法来扩充Parcel的存储能力，扩充成功后返回到goto的标志处：**restart\_write**继续执行。

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;

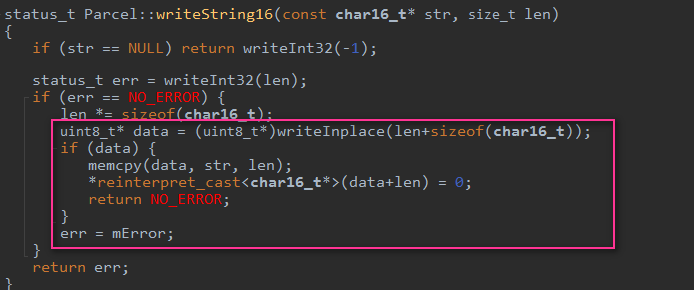
这时候还要去做一个处理就是要对新申请的内存做初始化，因为通过PAD\_SIZE计算出来的长度可能与之前传进来的len长度不一致



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**

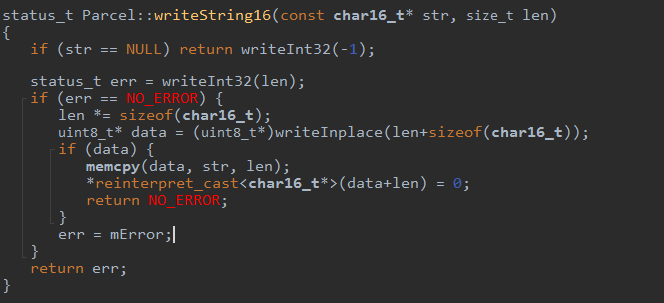


如上，首先判断之前计算的写入指针位置是否有效，然后调用memcpy函数将数据str拷贝到data所在的位置。因为拷贝的是字符串，还需要将字符串结尾置为”\0”。不必担心最后一位置0会超出Parcel的容量，因为Parcel在growData的时候会提升足够多的容量。

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**。

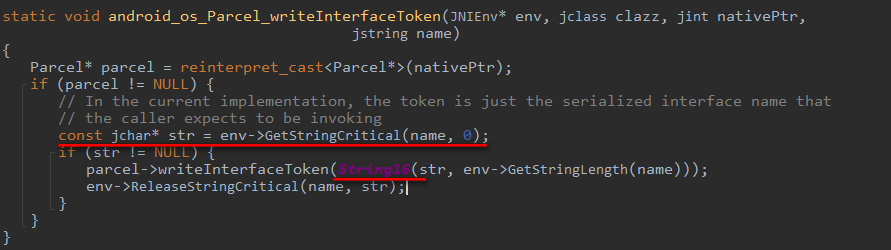


下面以java层的

data.writeInterfaceToken为例说明向Parcel中存储字符串的过程：

首先是要调用到JNI层的android\_os\_Parcel\_writeInterfaceToken方法

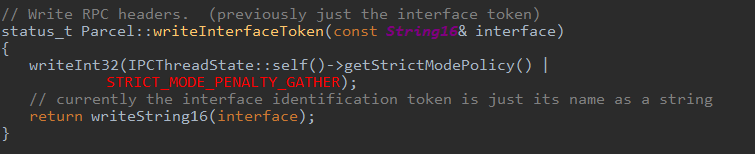
**frameworks/base/core/jni/android\_os\_Parcel.cpp**



java 本身采用unicode编码，GetStringCritical()方法用于指向字符串的Unicode字符数组指针，这所以使用这个JNI方法，而不采用GetStringChars()方法,主要是增加返回JVM中原始字符串指针的可能性。

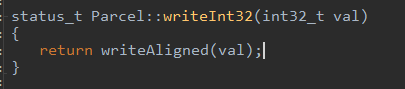
若取到的字符串指针不为NULL，则继续调用Parcel对象的writeInterfaceToken()方法，该方法重新包装了一个String16对象(因为在JNI层操作，而不是堆上，所以直接在栈上新建了一个对象，没有用new关键字。)

下面是Parcel类的writeInterfaceToken()方法：

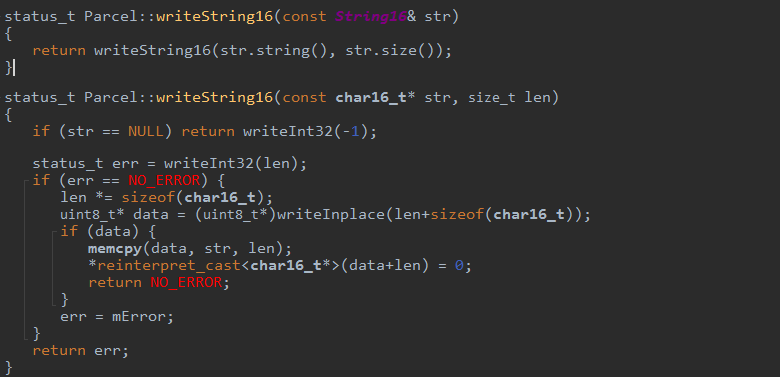


上面的

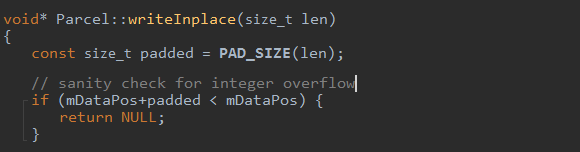
IPCThreadState::self()->getStrictModePolicy() | STRICT\_MODE\_PENALTY\_GATHER 其实就是一个int数，主要看writeInt32方法：



这个函数实现很简单，从函数名判断，它是将val值按照对齐的方式写入Parcel的存储空间中，换句话说，也就是将数据写入mDataPos起始的mData中。下面是writeString16()方法：



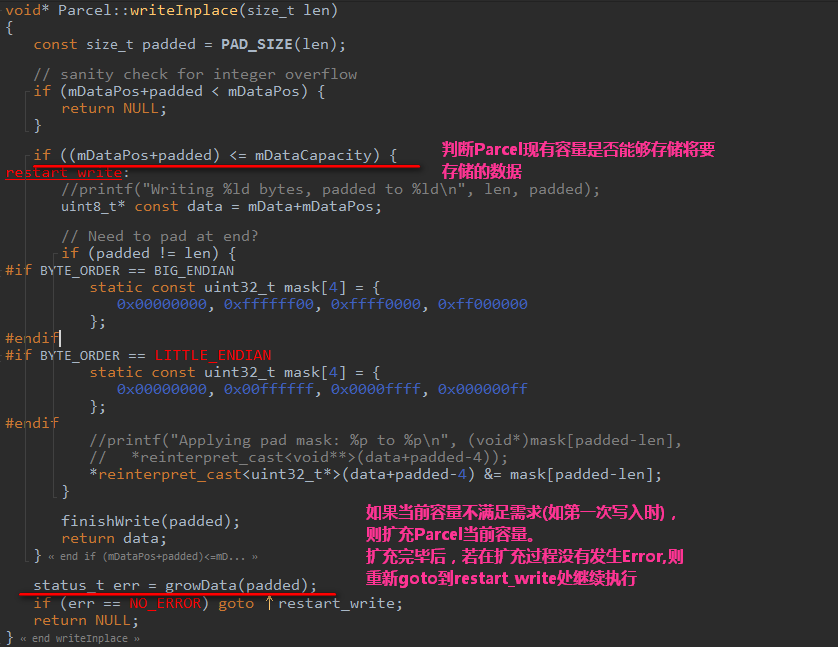
这里的writeInPlace()方法用于计算复制数据的目标地址



C:\Users\xp.chen\Desktop\2018-08-06_145141.png

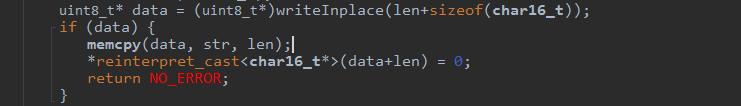
这里的PAD\_SIZE用于计算“当以4对齐时，容纳len大小的数据需要多少空间”，底下则对溢出进行了判断。如果溢出则返回NULL

接下来则对Parcel类当前的数据容量进行判断



中间的BYTE\_ORDER部分则主要是根据大小端对新增加的部分进行处理

数据扩充并且处理完毕后，调用finishWrite来调整mDataPos指针，并返回新的data指针的位置以供调用者写入真正的数据内容。



拿到新的数据指针位置后，开始将字符串memcpy到新的指针位置，最后写入字符串结束符0

综上，写入一个String(writeString16)的步骤为：

1. writeInt32(len) // 先写入数据长度
2. padding // 扩充Parcel内部容量(有时候不需要扩充)
3. memcpy // 将String拷贝到mData末尾。

4.4 **Binder驱动和Binder驱动协议(难点)**

主要了解Binder驱动支持哪些协议，以及这些协议具体有什么作用，比如：

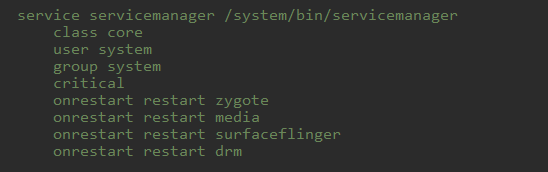
BINDER\_WRITE\_READ。该协议是用于读写操作，可以用此命令向Binder读取或写入数据

## 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)

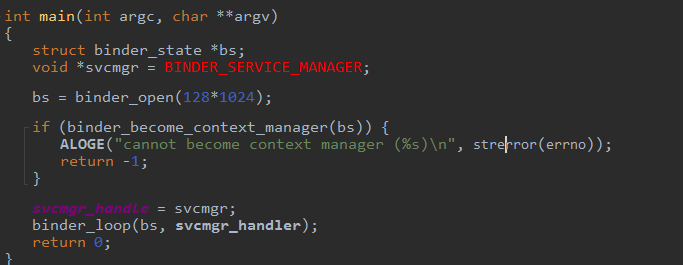
PS: 源码工程里还有很多ServiceManager.cpp文件存在，但并不属于SM程序。

### 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**



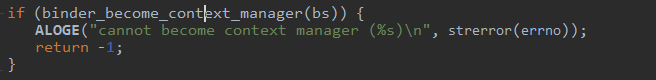
这个binder\_open()方法很重要，它返回了一个包含binder驱动信息的结构体指针。这个结构体所包含的内容包括：

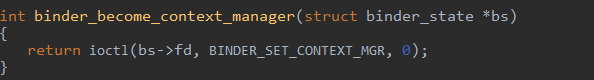
Binder驱动的设备文件描述符fd // 用于将来通过ioctl()函数向Binder设备驱动发送各种消息。

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

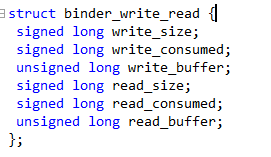


这里面有个地方很奇怪，

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

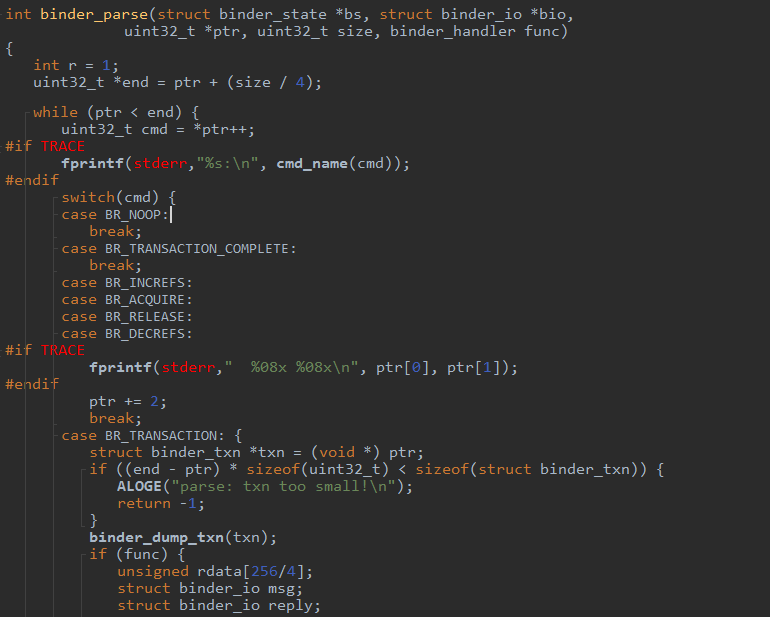
32位和64位系统下指针长度不一致(32位指针长度为4，64位指针长度为8)，所以使用了unsigned long类型来保存这个数组的指针。

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。

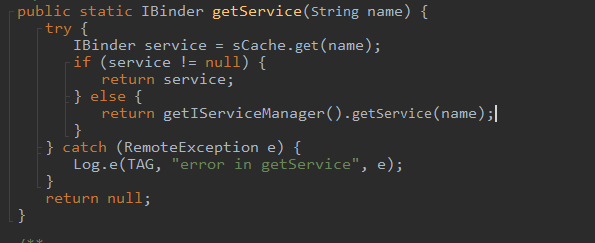
而**ServiceManagerProxy类的引入，则可以简化整个流程，直接调用这个类，就可以获取SM提供的各项服务**。

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);

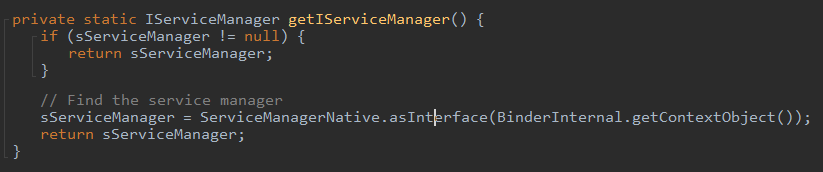
1. 通过getService()方法，返回了一个IBinder对象，这个IBinder对象又是个什么东西？

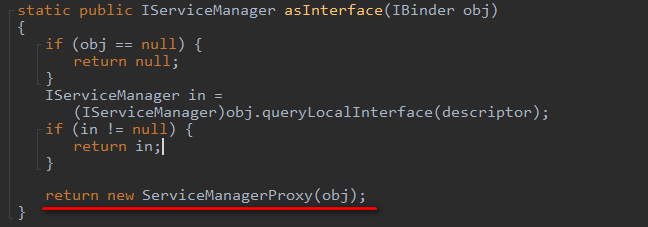
从代码中可以看出：这个IBinder对象是有可能为NULL的，如果为NULL的话，又尝试从getIServiceManager().getService()上去获取



1. 那么这个getIServiceManager()又是什么东西呢？

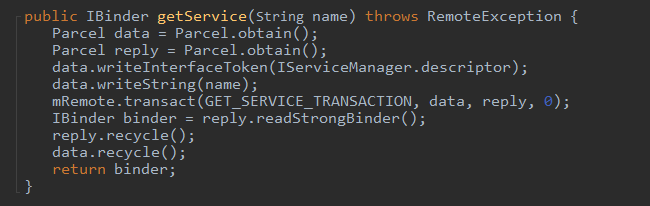
从代码上看，它是一个实现了IServiceManager接口的对象。





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？

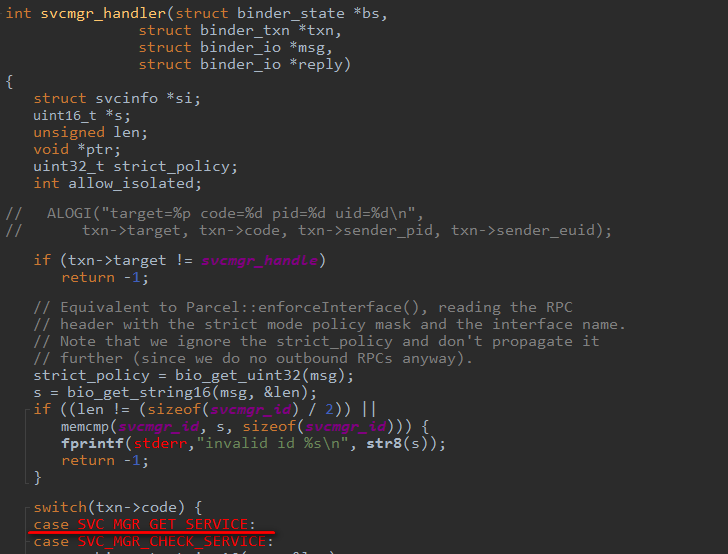
这里还有一点需要特别留意一下：

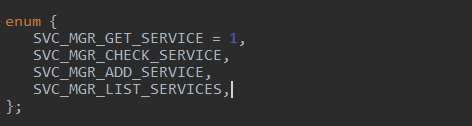
mRemote.transact(GET\_SERVICE\_TRANSACTION, data, reply, 0);

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()**



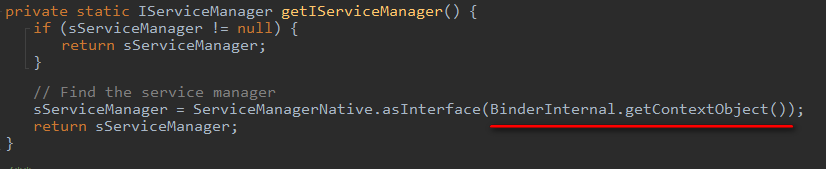


这个**SVC\_MGR\_GET\_SERVICE**的枚举值恰好是1

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。

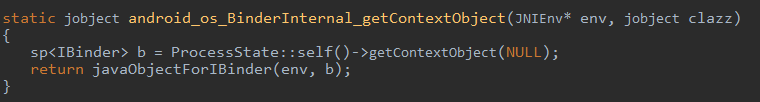
**frameworks/base/core/java/com/android/internal/os/BinderInternal.java**

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：

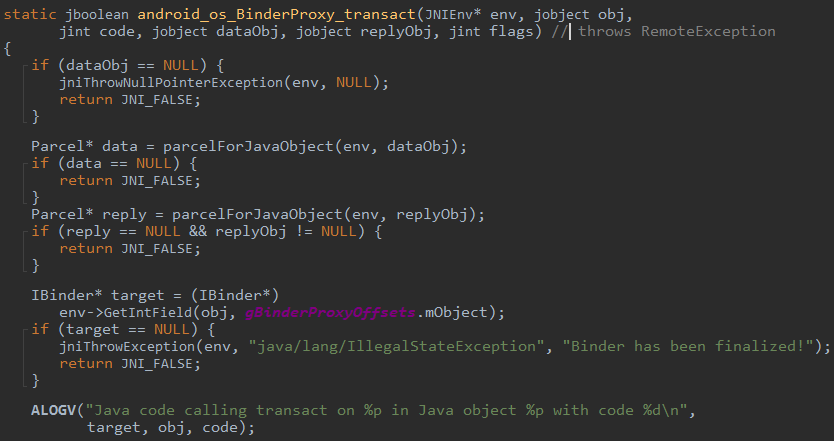
注：BinderProxy是Binder（android.os.Binder）的一个内部类。

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

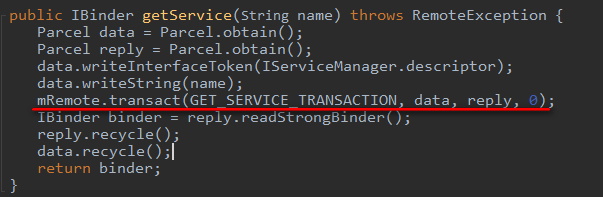
C:\Users\xp.chen\Desktop\2018-08-16_100433.png

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。

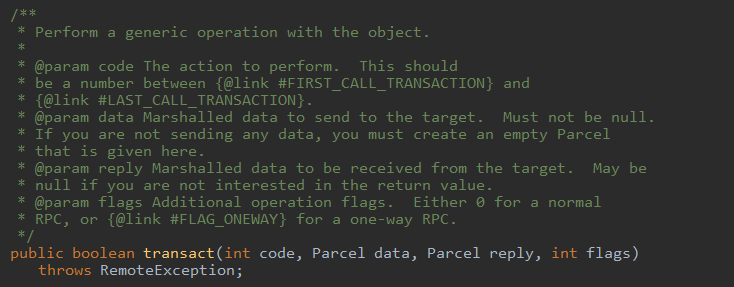


第一个参数是与Binder驱动通信的业务码；

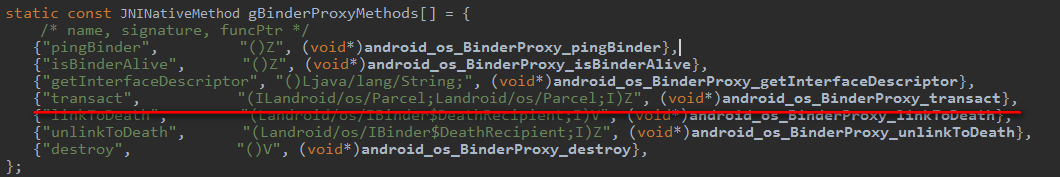
第二个参数是发给Binder驱动的Parcel对象（Parcel是约定格式）。

第三个参数是未来Binder驱动返回的Parcel对象。

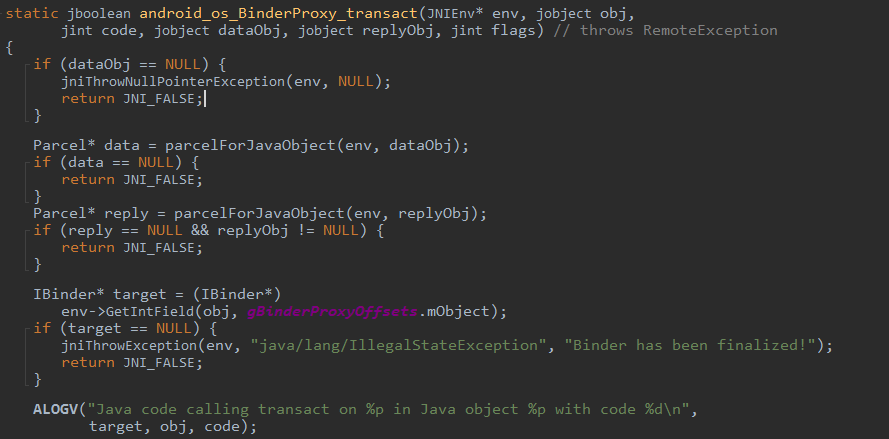
第四个参数可选，可以是0或者是1



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