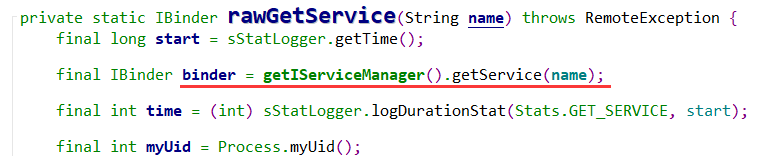
**Binder通信部分源码分析**

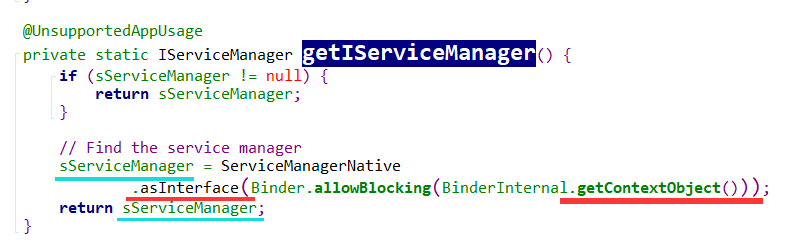
**在调用getSystemService(String)后会调用到ServiceManager.getService中:**



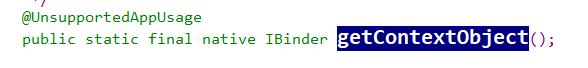
**调用了rawGetServuce：**



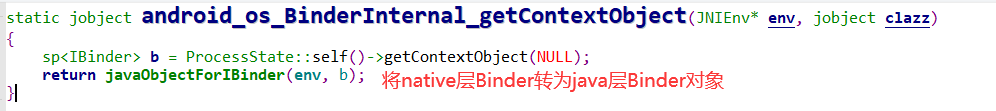
**这里调用getIserviceManager().getService，可以看这里就跟IPC通信相关了:**



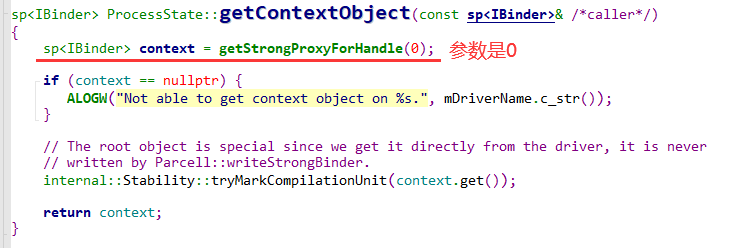
**可以看到，调用到了native方法：**



**该方法会返回一个java层的IBinder对象：**



**getContextObject:**

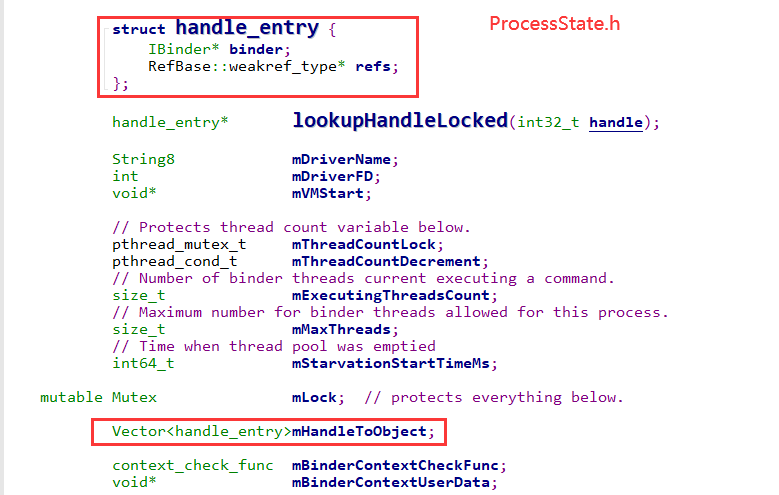


**传入的句柄为 0**



**在这里说一下这里的结构体handle\_entry.**

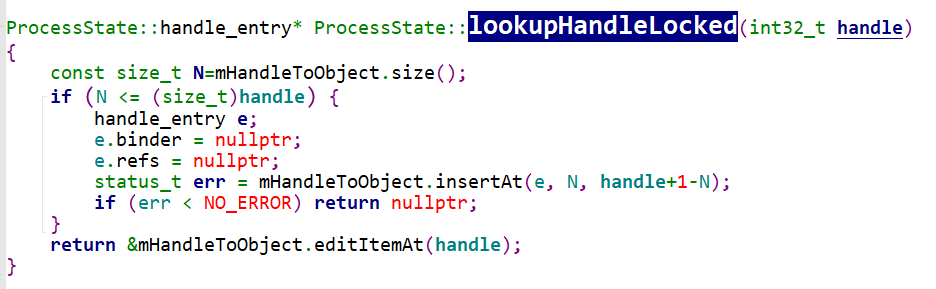
**Binder 为每一个进程维护了一个handle\_entry类型的Binder代理对象列表，他一句柄作为关键字来维护进程内部所有的Binder代理对象，这个Binder代理对象列表保存在ProcessState类的成员变量mHandlerToObject中，定义如下:**



**每一个Binder代理对象都使用一个handle\_entry的结构体来描述，handle\_entry的两个成员变量binder与refs分别指向一个Binder代理对象和它内部的一个弱引用计数对象。**

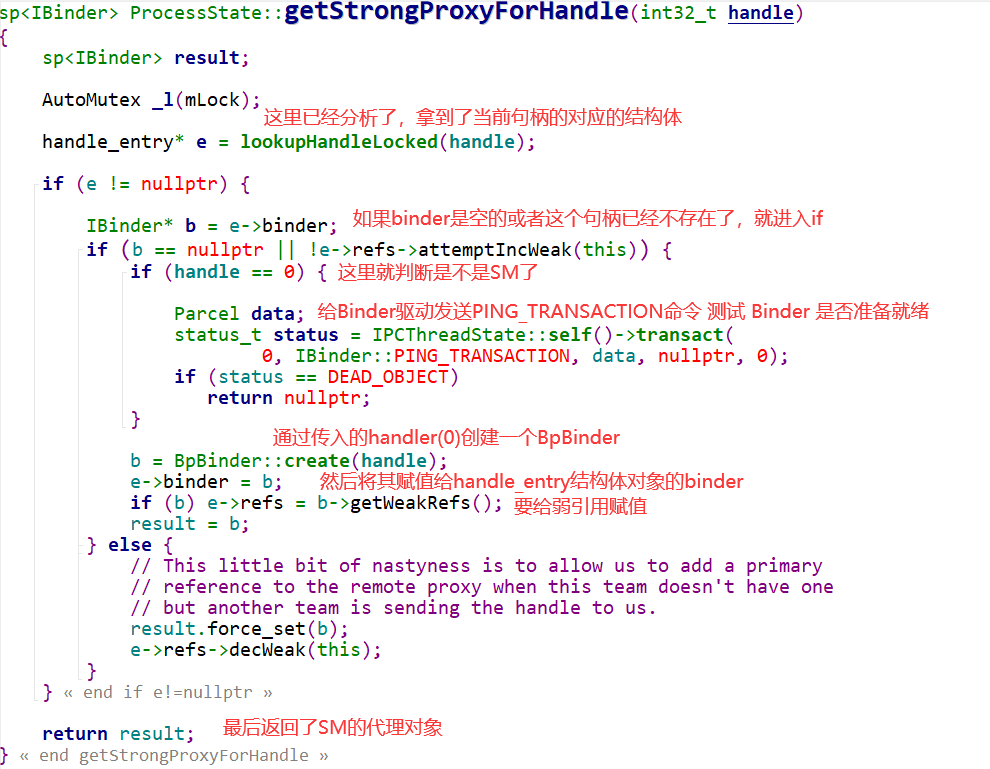
**一个Binder代理对象的句柄同时也是它在列表 mHandleToObject中索引值。**

**接着查看lookupHandleLocked ：**

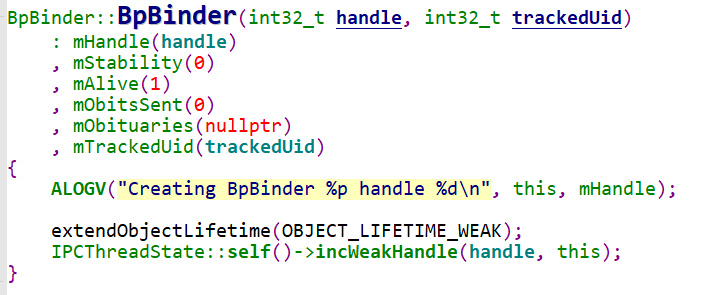


**如有N<handle句柄值，说明这个句柄并没有在数组中找到对应的结构体，那么就创建一个新的结构体对象并插入到数组中，但是要注意的是结构体的变量都是NULL的。**

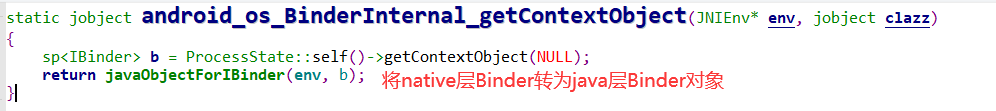
**返回方法继续看:**



**BpBinder对象：保存了句柄值，对变量进行初始化，然后给句柄添加了弱引用**



**返回了SM的代理对象，接着看上层调用：**

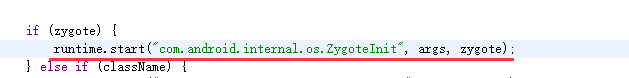


**调用了javaObjectForIBinder对象，将native层的BpBinder对象转为java端的BinderProxy对象**

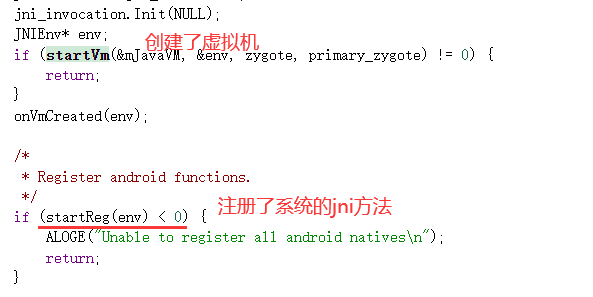
**具体怎么转的？那就要从一个遥远的地方开始说起：**

作图！！！！！！！！

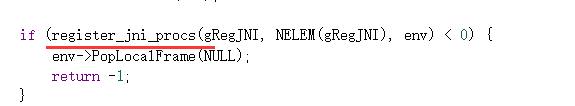
**我们知道zygote进程启动后就进入到了app\_main.cpp:**



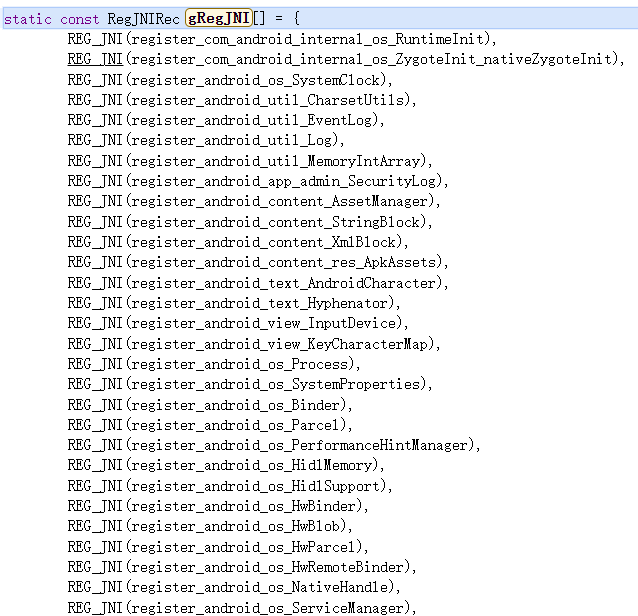
**调用了start方法：**



**进入到starReg:**

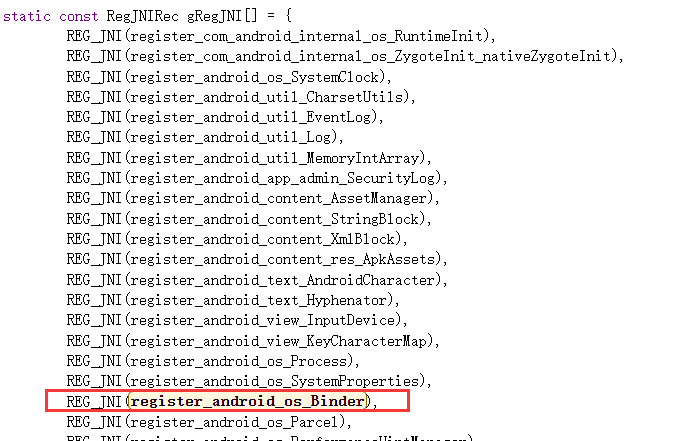


**register\_jni\_procs方法翻入了gRegJNI数组，和 长度：**





**这里重点来了，查看这里对Binder注册的jni方法：**



**register\_android\_os\_Bidner:**



**先看第一个方法:int\_register\_android\_os\_BinderInternal:**



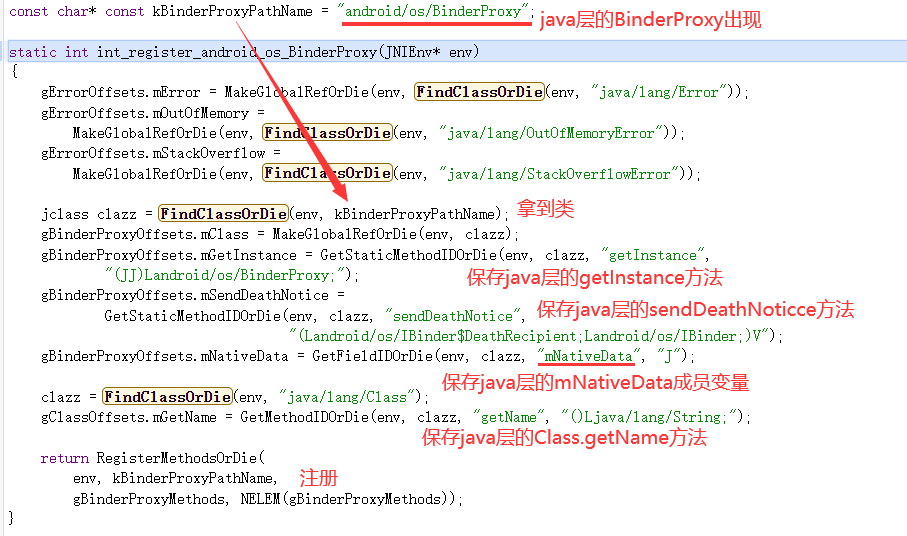
**这样，native层就可以访问java层的这个类的这些方法以及变量了**

**继续看第二个方法：**



**该方法同方法一类似，**

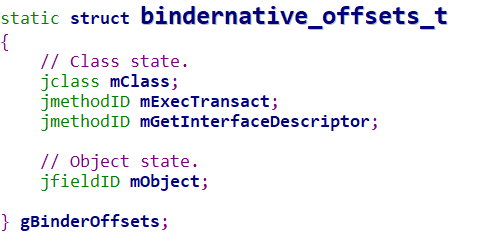
**最后看第三个方法，也是我们所关心的跟上文讲native层的BpBinder转java层BinderProxy相关：**



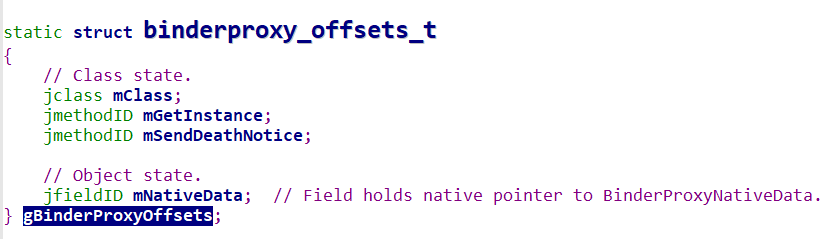
**这里注册之后，我们在native层就可以访问java层的BinderProxy类对象以及他的方法。**

**这里我们先来看看上面讲到的结构体的定义：**

**第一个方法的结构体：**



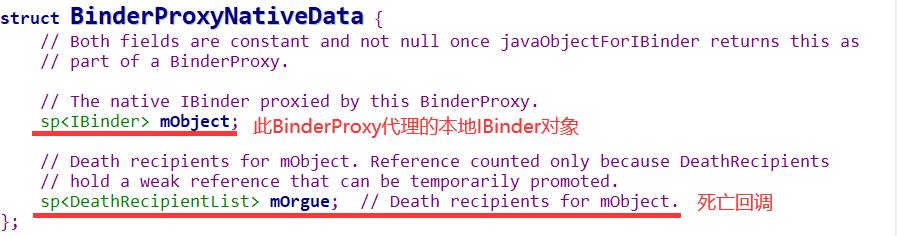
**第三个方法的结构体：**



**也就是说，通过这2个结构体，我们可以在native层访问到java层对应的类以及方法**



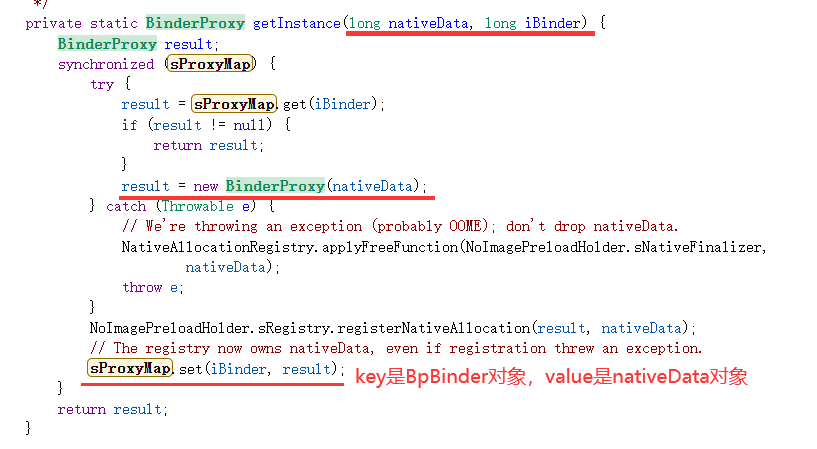
**①：这里创建了BinderProxyNativeData,从名字看是BinderProxy在native层的数据：**



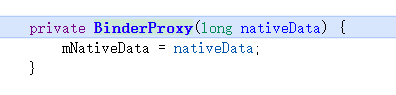
**可以看到这是一个结构体，mObject是保存了native层的BpBinder。**

**②：从之前我们分析第三个方法时，知道这里就是调用的java层的BinderProxy类的getInstance方法，传入的参数时nativeData,val.get()也就是BpBinder对象**

**我们到java层查看该getInstance:**

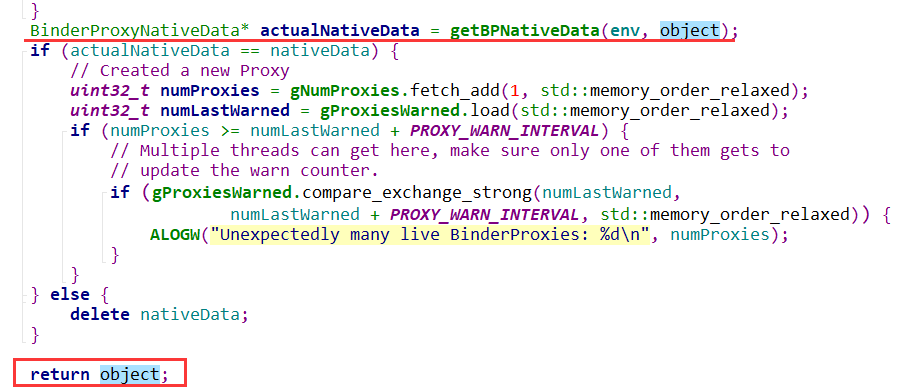


**从这里可以看到，根据我们传入进来的nativeData，创建了BinderProxy,而nativeData对象里面是保存了native层的BpBinder对象的**

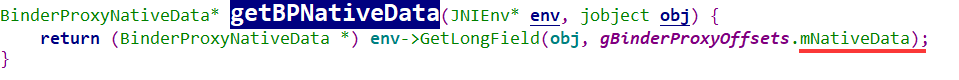


**将该变量保存到了成员变量mNativeData中，getInstance最后是返回了BinderProxy对象，mNativeData 在哪里出现过？**

**②处返回了BinderProxy对象，继续看③：**

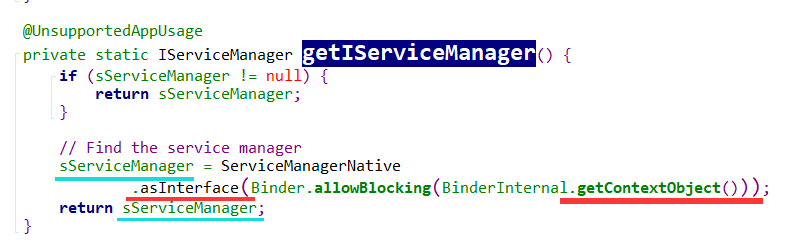


**调用了getBpNativeData方法，并传入了BinderProxy对象：**

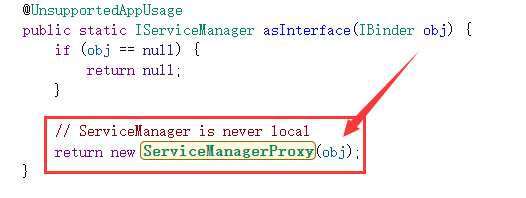


**这里就是把mNativerData取了出来。然后做了一些记录操作，最后返回了BinderProxy，这样，就将Native层的BpBinder对象转为了Java层的BindProxy对象。**

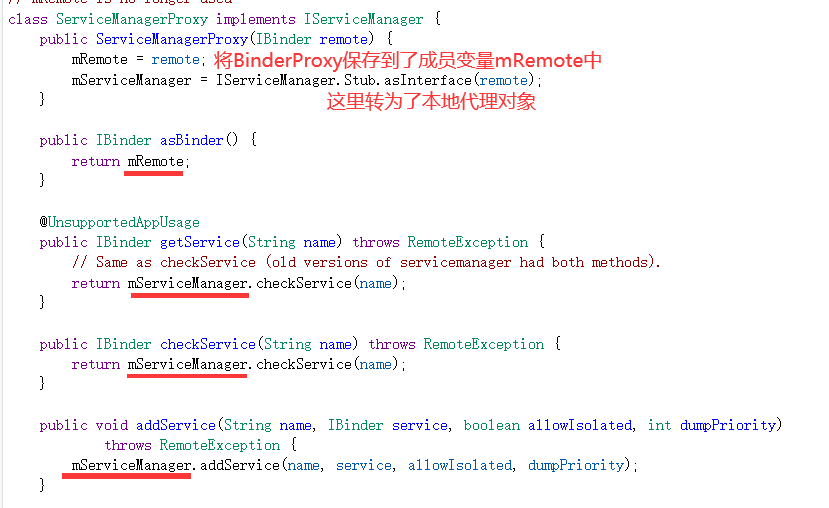
**最后回到上层：**



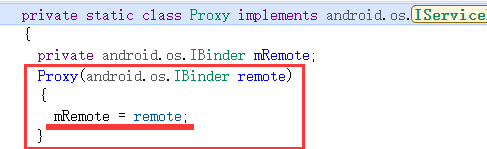
**拿到BinderProxy后调用了ServiceManagerNative.asInterface:**



**这里可以看到再次对BinderProxy进行了封装：**

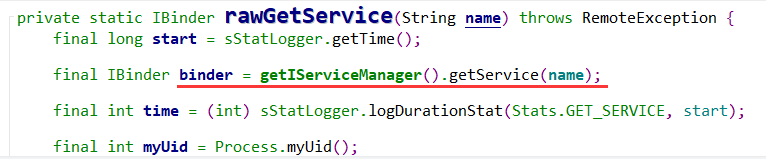


**可以看到，这里熟悉的方法调用都是通过BinderProxy的本地代理对象实现的：**

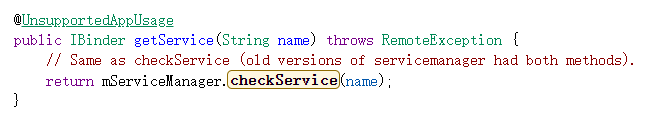


**也就是我们经常看到的mRemote.transact(xxx)，mRemote就是我们这里讲到的BinderProxy**

**继续讲解，这里最后返回了本地代理对象BinderProxy之后就回到了这里：**



**这里调用了getService，也就是我们刚才所看到的方法：**



**IServiceManager.java**



**最后调用了mRemote.transact方法像驱动发送命令去处理了，最后返回Binder代理对象**