目录

1. 请介绍一下 NDK
2. 什么是 NDK 库?

https://blog.csdn.net/carson_ho/article/details/73250163

https://www.jianshu.com/p/fa613762f516

JNI 介绍

定义：Java Native Interface，即 Java 本地接口

作用： 使得 Java 与 本地其他类型语言（如 C、C++）交互（即在 Java 代码 里调用 C、C++等语言的代码 或 C、C++代码调用 Java 代码）

特别注意：

1) JNI 是 Java 调用 Native 语言的一种特性

2) JNI 是属于 Java 的，与 Android 无直接关系

NDK 介绍

定义：Native Development Kit，是 Android 的一个工具开发包 。NDK 是属于 Android 的，与 Java 并无直接关系

作用：快速开发 C、 C++的动态库，并自动将 so 和应用一起打包成 APK （即可通过 NDK 在 Android 中 使用 JNI 与本地代码（如 C、C++）交互）

应用场景：在 Android 的场景下 使用 JNI （即 Android 开发的功能需要本地代码（C/C++）实现）

NDK 特点

(1) 性能方面：

1) **运行效率高：**在开发要求高性能的需求中，采用 C/C++更加有效率（如使用本地代码（C/C++）执行算法，能大大提高算大的执行效率）

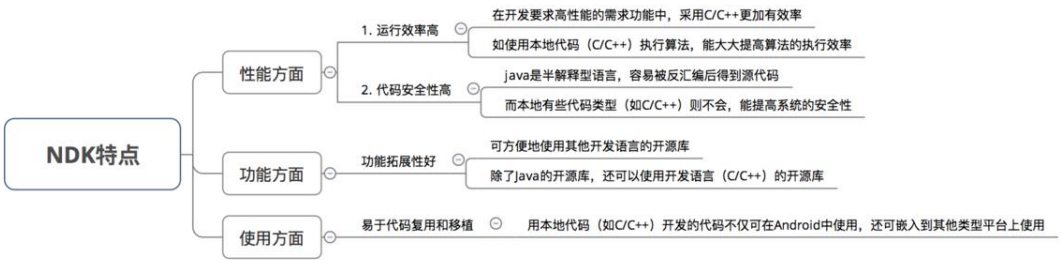
2) **代码安全性高：**java 是半解释型语言，容易被反编译后得到源代码,而本地有些代码类型（如 C/C++）则不会，能提高系统的安全性

(2) 功能方面：

功能扩展性好：可方便地使用其他开发语言的开源库，除了 java 的开源库，还可以使用开发语言（C/C++）的开源库

(3) 使用方面：

易于代码复用和移植：用本地代码（如 C/C++）开发的代码不仅可在 Android 中使用，还可嵌入其他类型平台上使用

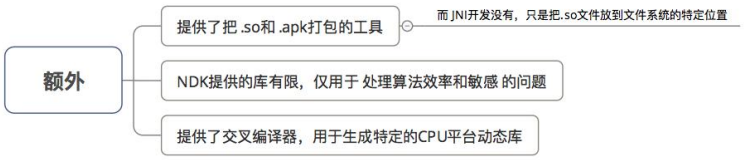


额外注意

1） 提供了吧 .so 和 .apk 打包的工具（而 JNI 开发没有，只是把.so 文件放到文件系统的特定位置）

2） NDK 提供的库有限，仅用于处理算法效率和敏感的问题

3） 提供了交叉编译器，用于生成特定的 CPU 平台动态库



NDK 与 JNI 关系

	JNI	NDK
定义	Java 中的接口	Android 中的工具开发包
作用	用于 java 与本地语言（如 C/C++）交互	快速开发 C/C++的动态库，并自动将 so 和应用一起打包成 APK
之间的关系：JNI 是实现的目的，NDK 是在 Android 中实现 JNI 的手段（即在 Android 的开发环境中（Android Studio），通过 NDK 从而实现 JNI 的功能）		

3. jni 用过吗？

https://www.jianshu.com/p/fe42aa3150a0

JNI：

JNI(Java Native Interface) Java 本地接口，又叫 Java 原生接口。它允许 Java 调用 C/C++的代码,同时也允许在 C/C++中调用 Java 的代码。可以把 JNI 理解为一个桥梁，连接 Java 和底层。其实根据字面意思，JNI 就是一个介于 Java 层和 Native 层的接口，而 Native 层就是 C/C++层面。

为什么使用 JNI：

一般情况下都是从 Java 的角度来使用 JNI，也就是说在 Java 中调用 C/C++语言来实现一些操作。所以从 Java 角度来说使用 JNI 具有以下的优点：

1) 能够重复使用一些现成的、具有相同功能的的 C/C++代码

2) 因为 C/C++是偏向底层的语言，所以使用 C/C++能够更加的高效，而且也使得 Java 能够访问操作系统中一些底层的特性。

使用 JNI

这里所说的使用 JNI 是指从 Java 层调用 C/C++代码，一般的使用步骤都是使用 Java 定义一个类，然后在该类中声明一个 native 的方法，接着使用 C/C++来实现这个方法的方法体。

1) 使用 Java 声明 native 方法
2) 编译声明的 Java 文件
3) 用 C 语言来实现函数
4) 生成动态库文件
5) 使用生成的动态库文件

4. 如何在 jni 中注册 native 函数，有几种注册方式?

5. Java 如何调用 c、c++语言？

https://www.jianshu.com/p/27404a899d88

https://blog.csdn.net/wwj_748/article/details/52347341

注册 JNI 函数的两种方法

静态方法

这种方法我们比较常见，但比较麻烦，大致流程如下：

1) 先创建 Java 类，声明 Native 方法，编译成.class 文件。

2) 使用 Javah 命令生成 C/C++的头文件，例如：javah -jni com.devilwwj.jnidemo.TestJNI，则会生成一个以.h 为后缀的文件 com_devilwwj_jnidemo_TestJNI.h。

3) 创建.h 对应的源文件，然后实现对应的 native 方法

这种方法的弊端：

1) 需要编译所有声明了 native 函数的 Java 类，每个所生成的 class 文件都得用 javah 命令生成一个头文件。

2) javah 生成的 JNI 层函数名特别长，书写起来很不方便

3) 初次调用 native 函数时要根据函数名字搜索对应的 JNI 层函数来建立关联关系，这样会影响运行效率

动态注册

我们知道 Java Native 函数和 JNI 函数时一一对应的，JNI 中就有一个叫 JNINativeMethod 的结构体来保存这个对应关系，实现动态注册方就需要用到这个结构体。举个例子，你就一下子明白了：

1) 声明 native 方法还是一样的：

```
public class JavaHello {

    public static native String hello();

}
```

2)　创建 jni 目录，然后在该目录创建 hello.c 文件，如下：

```
#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#include <jni.h>

#include <assert.h>


/**

 * 定义 native 方法

 */

JNIEXPORT jstring JNICALL native_hello(JNIEnv *env, jclass clazz)

{

    printf("hello in c native code./n");

    return (*env)->NewStringUTF(env, "hello world returned.");

}


// 指定要注册的类

#define JNIREG_CLASS "com/devilwwj/library/JavaHello"


// 定义一个 JNINativeMethod 数组，其中的成员就是 Java 代码中对应的 native 方法

static JNINativeMethod gMethods[] = {

    { "hello", "()Ljava/lang/String;", (void*)native_hello},

};


static int registerNativeMethods(JNIEnv* env, const char* className,
```

```
JNINativeMethod* gMethods, int numMethods) {
    jclass clazz;
    clazz = (*env)->FindClass(env, className);
    if (clazz == NULL) {
        return JNI_FALSE;
    }
    if ((*env)->RegisterNatives(env, clazz, gMethods, numMethods) < 0) {
        return JNI_FALSE;
    }
    return JNI_TRUE;
}

/***
 * 注册 native 方法
 */
static int registerNatives(JNIEnv* env) {
    if (!registerNativeMethods(env, JNIREG_CLASS, gMethods, sizeof(gMethods) /
sizeof(gMethods[0]))) {
        return JNI_FALSE;
    }
    return JNI_TRUE;
}

/**
 * 如果要实现动态注册，这个方法一定要实现
 * 动态注册工作在这里进行
```

```
 */
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;
    jint result = -1;

    if ((*vm)-> GetEnv(vm, (void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        return -1;
    }
    assert(env != NULL);

    if (!registerNatives(env)) { //注册
        return -1;
    }
    result = JNI_VERSION_1_4;

    return result;

}
```

先仔细看一下上面的代码，看起来好像多了一些代码，稍微解释下，如果要实现动态注册就必须实现 JNI_OnLoad 方法，这个是 JNI 的一个入口函数，我们在 Java 层通过 System.loadLibrary 加载完动态库后，

1) 紧接着就会去查找一个叫 JNI_OnLoad 的方法。如果有，就会调用它，而动态注册的工作就是在这里完成的。在这里我们会去拿

到 JNI 中一个很重要的结构体 JNIEnv，env 指向的就是这个结构体，通过 env 指针可以找到指定类名的类，并且调用 JNIEnv 的 RegisterNatives 方法来完成注册 native 方法和 JNI 函数的对应关系。

2) 我们在上面看到声明了一个 JNINativeMethod 数组，这个数组就是用来定义我们在 Java 代码中声明的 native 方法，我们可以在 jni.h 文件中查看这个结构体的声明：

```
typedef struct {
    const char* name;
    const char* signature;
    void*       fnPtr;
} JNINativeMethod;
```

结构体成员变量分别对应的是 Java 中的 native 方法的名字，如本文的 hello；Java 函数的签名信息、JNI 层对应函数的函数指针。

区别：

1) **静态注册**

优点：理解和使用方式简单，属于傻瓜式操作，使用相关工具按流程操作就行，出错率低

缺点：当需要更改类名,包名或者方法时，需要按照之前方法重

新生成头文件，灵活性不高

2) **动态注册**

优点：灵活性高，更改类名,包名或方法时，只需对更改模块进行少量修改，效率高

缺点：对新手来说稍微有点难理解，同时会由于搞错签名，方法,导致注册失败

6. jni 如何调用 java 层代码？

https://www.jianshu.com/p/4893848a3249

Android 开发中调用一个类中没有公开的方法，可以进行反射调用，而 JNI 开发中 C 调用 java 的方法也是反射调用。

C 代码回调 Java 方法步骤：

①获取字节码对象（jclass (FindClass)(JNIEnv, const char*);）

②通过字节码对象找到方法对象（jmethodID (GetMethodID)(JNIEnv, jclass, const char, const char);）

③通过字节码文件创建一个 object 对象（该方法可选，方法中已经传递一个 object，如果需要调用的方法与本地方法不在同一个文件夹则需要新创建 object（jobject (AllocObject)(JNIEnv, jclass);），如果需要反射调用的 java 方法与本地方法不在同一个

类中，需要创建该方法，但是如果是这样，并且需要跟新 UI 操作，例如打印一个 Toast 会报空指针异常，因为这时候调用的方法只是一个方法，没有 actiivty 的生命周期。）

④通过对象调用方法，可以调用空参数方法，也可以调用有参数方法，并且将参数通过调用的方法传入（void (CallVoidMethod)(JNIEnv, jobject, jmethodID, ...);）

```
/**
 调用 java 中 Int 方法
 */
JNIEXPORT void JNICALLJava_com_mao_ccalljava_JNI_callbackIntmethod(JNIEnv *env, jobject object) {
 jclass clzz=(*env)->FindClass(env,"com/mao/ccalljava/JNI");
 jmethodID methodID=(*env)->GetMethodID(env,clzz,"add","(II)I");
int result=(*env)->CallIntMethod(env,object,methodID,3,4);
 //logcat 打印相加返回的结果
 LOGD("RESLUT = %d",result);
}
```

7. Binder 机制

http://www.cnblogs.com/innost/archive/2011/01/09/1931456.html

https://blog.csdn.net/luoshengyang/article/details/6618363/

Binder 定义：

在 Android 内部，那些支撑应用的组件往往会身处于不同的进程，那么应用的底层必然会牵涉大量的跨进程通信。为了保证通信的高效性，Android 提供了 Binder 机制。

Binder 机制具有两层含义：

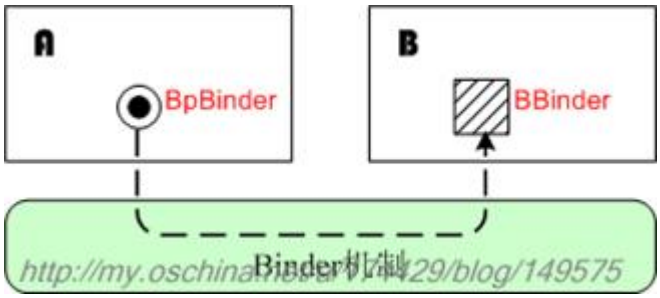
1) 是一种跨进程通信手段（IPC，Inter-Process Communication）。

2) 是一种远程过程调用手段（RPC，Remote Procedure Call）。

从实现的角度来说，Binder 核心被实现成一个 Linux 驱动程序，并运行于内核态。这样它才能具有强大的跨进程访问能力。

简述 Binder 的跨进程机制

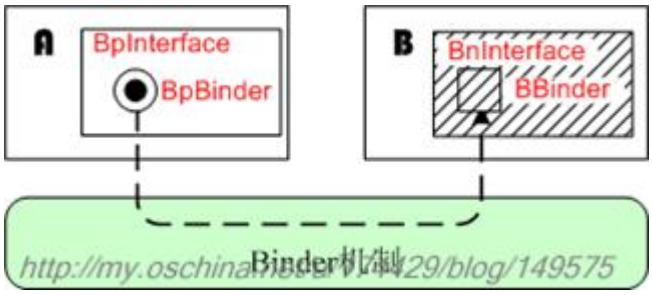
一张最简单的跨进程通信示意图：



图中 A 侧的圆形块◉，表示“Binder 代理方”，主要用于向远方发送语义，而 B 侧的方形块▨则表示“Binder 响应方”，主要用于响应语义。

在后文中，我们可以看到，Binder 代理方大概对应于 C++层次的 BpBinder 对象，而 Binder 响应方则对应于 BBinder 对象。

然而，上图的 Binder 代理方主要只负责了“传递信息”的工作，并没有起到“远程过程调用”的作用，如果要支持远程过程调用，我们还必须提供“接口代理方”和“接口实现体”。这样，我们的示意图就需要再调整一下，如下：



从图中可以看到，A 进程并不直接和 BpBinder（Binder 代理）打交道，而是通过调用 BpInterface（接口代理）的成员函数来完成远程调用的。此时，BpBinder 已经被聚合进 BpInterface 了，它在 BpInterface 内部完成了一切跨进程的机制。另一方面，与 BpInterface 相对的响应端实体就是 BnInterface（接口实现）了。需要注意的是，BnInterface 是继承于 BBinder 的，它并没有采用聚合的方式来包含一个 BBinder 对象，所以上图中 B 侧的 BnInterface 块和 BBinder 块的背景图案是相同的。

这样看来，对于远程调用的客户端而言，主要搞的就是两个东西，一个是“Binder 代理”，一个是“接口代理”。而服务端主要搞的则是“接口实现体”。因为 binder 是一种跨进程通信机制，所以还需要一个专门的管理器来为通信两端牵线搭桥，这个管理器就是 Service Manager Service。

Binder 相关接口和类

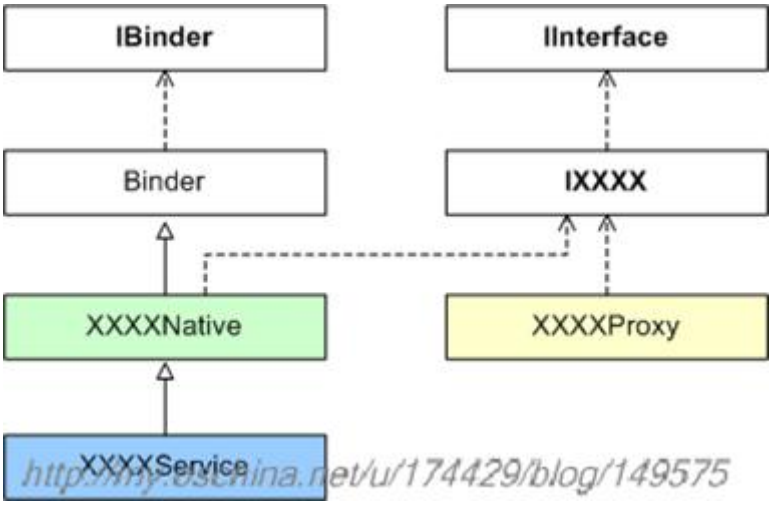
Android 的整个跨进程通信机制都是基于 Binder 的，这种机制不但会在底层使用，也会在上层使用，所以必须提供 Java 和 C++两个层次的支持。

Java 层次的 binder 元素

Java 层次里并没有我们前文图中所表示的 BpBinder、BpInterface、BBinder 等较低层次的概念，取而代之的是 IBinder 接口、IInterface 等接口。Android 要求所有的 Binder 实体都必须实现 IBinder 接口，

另外，不管是代理方还是实体方，都必须实现 IInterface 接口：

Java 层次中，与 Binder 相关的接口或类的继承关系如下：



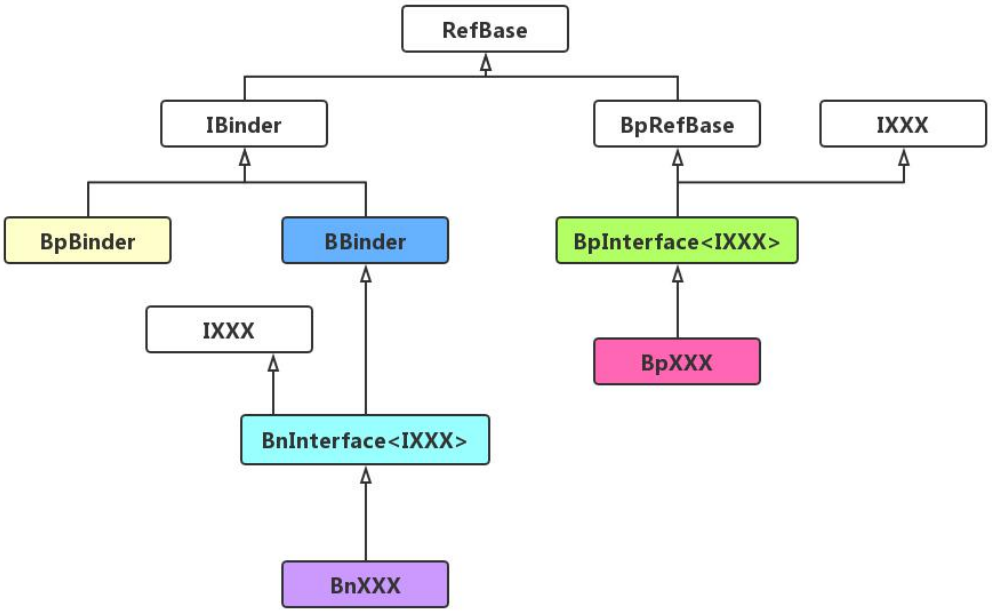
在实际使用中，我们并不需要编写上图的 XXXXNative、XXXXProxy，它们会由 ADT 根据我们编写的 aidl 脚本自动生成。用户只需继承 XXXXNative 编写一个具体的 XXXXService 即可，这个 XXXXService 就是远程通信的服务实体类，而 XXXXProxy 则是其对应的代理类。

关于 Java 层次的 binder 组件，我们就先说这么多，主要是先介绍一个大概。就研究跨进程通信而言，其实质内容基本上都在 C++层次，Java 层次只是一个壳而已。

C++层次的 binder 元素

在 C++层次，就能看到我们前文所说的 BpBinder 类和 BBinder 类了。这两个类都继承于 IBinder，IBinder 的定义截选如下：

C++层次的继承关系图如下：



其中有以下几个很关键的类：

1) BpBinder

2) BpInterface

3) BBinder

4) BnInterface

它们扮演着很重要的角色。

BpBinder

作为代理端的核心，BpBinder 最重要的职责就是**实现跨进程传输的传输机制**，至于具体传输的是什么语义，它并不关心。我们观察它的 transact()函数的参数，可以看到所有的语义都被打包成 Parcel 了。

BpInterface

BpInterface 使用了模板技术，而且因为它继承了 BpRefBase，所以先天上就聚合了一个 mRemote 成员（这个成员记录的就是前面所说的 BpBinder 对象啦）。以后，我们还需要继承 BpInterface<>实现我们自己的代理类。

在实际的代码中，我们完全可以创建多个聚合同一 BpBinder 对象的代理对象，这些代理对象就本质而言，对应着同一个远端 binder 实体。在 Android 框架中，常常把指向同一 binder 实体的多个代理称为 token，这样即便这些代理分别处于不同的进程中，它们也具有了某种内在联系。这个知识点需要大家关注。

BBinder

Binder 远程通信的目标端实体必须继承于 BBinder 类，该类和 BpBinder 相对，主要关心的只是传输方面的东西，不太关心所传

输的语义。

transact()内部会调用 onTransact()，从而走到用户所定义的子类的 onTransact()里。这个 onTransact()的一大作用就是解析经由 Binder 机制传过来的语义了。

BnInterface

远程通信目标端的另一个重要类是 BnInterface<>，它是与 BpInterface<>相对应的模板类，**比较关心传输的语义**。一般情况下，服务端并不直接使用 BnInterface<>，而是使用它的某个子类。为此，我们需要编写一个新的 BnXXX 子类，并重载它的 onTransact()成员函数。

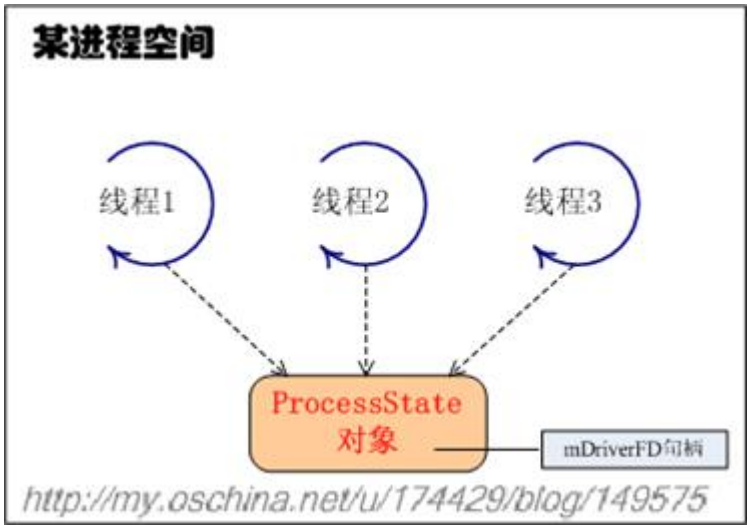
ProcessState

底层进程运用 Binder 机制来完成跨进程通信的：

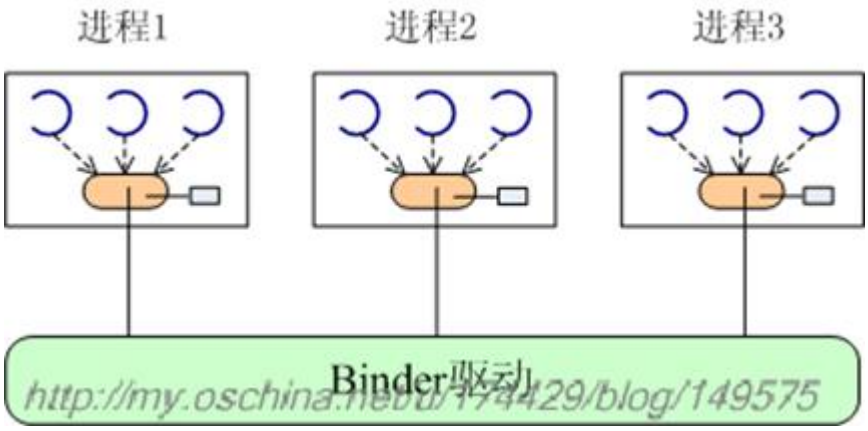
在每个进程中，会有一个全局的 ProcessState 对象。这个很容易理解，ProcessState 的字面意思不就是“进程状态”吗，当然应该是每个进程一个 ProcessState。

Binder 内核被设计成一个驱动程序，所以 ProcessState 里专门搞了个 mDriverFD 域，来记录 binder 驱动对应的句柄值，以便随时和 binder 驱动通信。ProcessState 对象采用了典型的单例模式，在一个应用进程中，只会有唯一的一个 ProcessState 对象，它将被进

程中的多个线程共用，因此每个进程里的线程其实是共用所打开的那个驱动句柄（mDriverFD）的，示意图如下：



每个进程基本上都是这样的结构，组合起来的示意图就是：



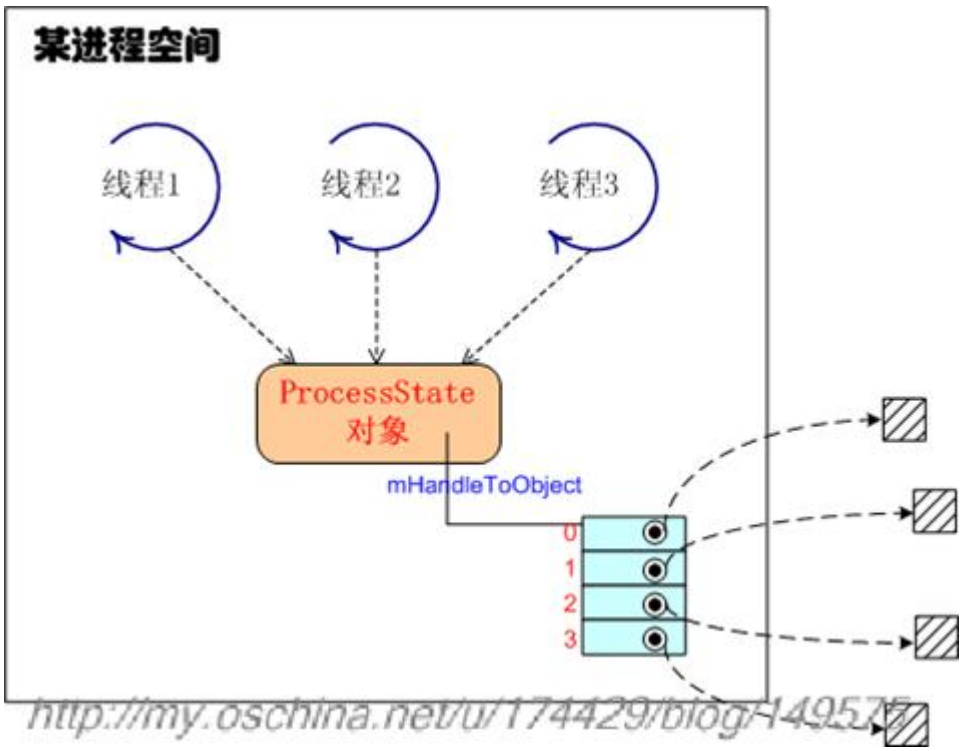
因为 ProcessState 采用的是单例模式，所以它的构造函数是 private 的，我们只能通过调用 ProcessState::self()来获取进程中唯一的一个 ProcessState 对象。

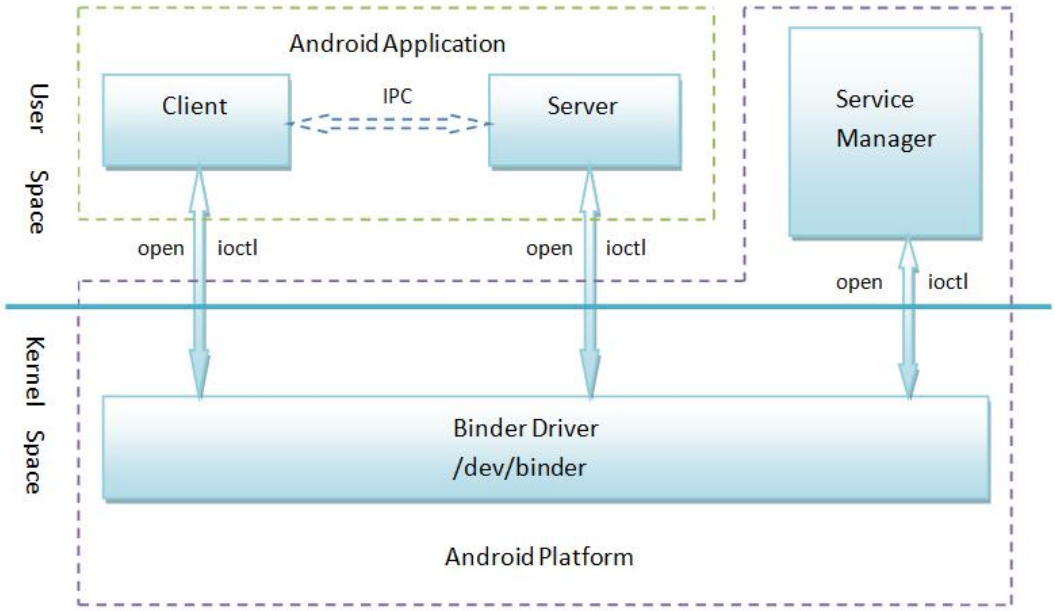
ProcessState 对象构造之时，就会打开 binder 驱动：构造函数中有句 mDriverFD(open_driver())，其中的 open_driver()就负责打开“/dev/binder”驱动：

ProcessState 中另一个比较有意思的域是 mHandleToObject：

Vector<handle_entry> mHandleToObject;它是本进程中记录所有 BpBinder 的向量表噢，非常重要。BpBinder 是代理端的核心。在 Binder 架构中，应用进程是通过“binder 句柄”来找到对应的 BpBinder 的。从这张向量表中我们可以看到，那个句柄值其实对应着这个向量表的下标。

其中的 binder 域，记录的就是 BpBinder 对象。





8. 简述 IPC？

https://blog.csdn.net/luoshengyang/article/details/6618363/

IPC 定义

IPC 是 Inter-Process-Communication 的缩写，意思是进程间通信或者跨进程通信；（按照操作系统的描述，线程是 CPU 调度的最小单元，而进程一般指一个执行单元，在移动设备上指一个程序或应用；一个进程可以包含多个线程；）

为什么要用到多进程？

在 Android 系统中一个应用默认只有一个进程，每个进程都有自己独立的资源和内存空间，其它进程不能任意访问当前进程的内存和资源，系统给每个进程分配的内存会有限制。如果一个进程占用内存

超过了这个内存限制，就会报 OOM 的问题，

很多涉及到大图片的频繁操作或者需要读取一大段数据在内存中使用时，很容易报 OOM 的问题，为了彻底地解决应用内存的问题，Android 引入了多进程的概念，它允许在同一个应用内，为了分担主进程的压力，将占用内存的某些页面单独开一个进程，比如 Flash、视频播放页面，频繁绘制的页面等。

Android 多进程使用很简单，只需要在 AndroidManifest.xml 的声明四大组件的标签中增加”android:process”属性即可，process 分私有进程和全局进程，以“：”号开头的属于私有进程，其他应用组件不可以和他跑在同一个进程中；不以“：”号开头的属于全局进程，其他应用可以通过 ShareUID 的方式和他跑在同一个进程中；

但是多进程模式出现以下问题：

1) 静态成员和单例模式完全失效

2) 线程同步机制完全失效

3) SharedPreferences 的可靠性下降

4) Application 多次创建

5)

因此为了避免这些问题 Android 中有多种 IPC 机制，如 AIDL，Messenger，Socket，ContentProvider，但是这些机制底层全部都是用了 Binder 机制来实现的

1) **bundle** ： 简单易用　但是只能传输 Bundle 支持的对象 常用于四大组件间进程间通信

2) **文件共享：**简单易用　但不适合在高并发的情况下 并且读取文件需要时间 不能即时通信　常用于并发程度不高 并且实时性要求不高的情况 3.AIDL ： 功能强大 支持一对多并发通信 支持即时通信　但是使用起来比其他的复杂 需要处理好多线程的同步问题　常用于一对多通信 且有 RPC 需求的场合(服务端和客户端通信)

3) **Messenger ：** 功能一般 支持一对多串行通信 支持实时通信　但是不能很好处理高并发情况 只能传输 Bundle 支持的类型　常用于低并发的无 RPC 需求一对多的场合

4) **ContentProvider ：** 在数据源访问方面功能强大 支持一对多并发操作 可扩展 call 方法　可以理解为约束版的 AIDL　提供 CRUD 操作和自定义函数　常用于一对多的数据共享场合

5) **Socket ：** 功能强大 可以通过网络传输字节流 支持一对多并发操作　但是实现起来比较麻烦 不支持直接的 RPC　常用于网络数据交换

总结起来

1)　当仅仅是跨进程的四大组件间的传递数据时 使用 Bundle 就可以简单方便

2)　当要共享一个应用程序的内部数据的时候使用 ContentProvider 实现比较方便

3)　当并发程度不高也就是偶尔访问一次那种 进程间通信用

Messenger 就可以

4) 当设计网络数据的共享时使用 socket

5) 当需求比较复杂高并发 并且还要求实时通信 而且有 RPC 需求时就得使用 AIDL 了

6) 文件共享的方法用于一些缓存共享 之类的功能

9. 什么是 AIDL?

https://www.jianshu.com/p/d1fac6ccee98

https://www.jianshu.com/p/a5c73da2e9be

AIDL 是 Android 中 IPC（Inter-Process Communication）方式中的一种，AIDL 是 Android Interface definition language 的缩写（对于小白来说，AIDL 的作用是让你可以在自己的 APP 里绑定一个其他 APP 的 service，这样你的 APP 可以和其他 APP 交互。）

AIDL 只是 Android 中众多进程间通讯方式中的一种方式，

AIDL 和 Messenger 的区别：

1) **Messenger 不适用大量并发的请求：**Messenger 以串行的方式来处理客户端发来的消息，如果大量的消息同时发送到服务端，服务端仍然只能一个个的去处理。

2) **Messenger 主要是为了传递消息：**对于需要跨进程调用服务端的方法，这种情景不适用 Messenger。

3) **Messenger 的底层实现是 AIDL**，系统为我们做了封装从而方便上层的调用。

4) **AIDL 适用于大量并发的请求，以及涉及到服务端端方法调用的情况**

AIDL 通信的原理：首先看这个文件有一个叫做 proxy 的类，这是一个代理类，这个类运行在客户端中，其实 AIDL 实现的进程间的通信并不是直接的通信，客户端和服务端都是通过 proxy 来进行通信的：客户端调用的方法实际是调用是 proxy 中的方法，然后 proxy 通过和服务端通信将返回的结果返回给客户端。

代码中的几个方法：

(1)DESCRIPTION

Binderd 的唯一标识，一般用当前的类名表示。

(2)asInterface(android.os.IBinder obj)

用于将服务端的 Binder 对象转换为客户端需要的 AIDL 接口类型的对象。转换区分进程，客户端服务端位于同一进程，返回服务端的 //Stub 对象本身；否则返回的是系统的封装后的 Stub.proxy 对象。

(3)asBinder

返回 Binder 对象

(4)onTransact

此方法运行在服务端中的 Binder 线程池中，当客户端发起跨进程请求时，远程请求会通过系统底层封装后交由此方法处理。

(5)Proxy#add

此方法运行在客户端，当客户端远程调用此方法时，它的内部实现是这样的：

1) 首先创建该方法所需要的输入型 Parcel 对象_data、输出型 Parcel 对象_reple 和返回值对象_result,
2) 然后将该方法的参数信息写入_data 中；
3) 接着调用 transact 方法来发 RPC 请求，同时当前线程挂起；
4) 然后服务端的 onTransact 方法会被调用，直到 RPC 过程返回后，当前线程继续执行，并从_reply 中取出 RPC 过程返回的结果，写入_result 中。

```
## 5.一些补充#####AIDL 支持的数据类型

+ 基本数据类型（int、long、char 等）

+ String 和 CharSequence

+ List：只支持 ArrayList，里面的每个元素都必须被 AIDL 支持。

+ Map：  只支持 HashMap，  里面的每个元素都必须被 AIDL 支持。

+ Parcelable：  所有实现了 Parcelable 接口的对象

+ AIDL：  所有的 AIDL 接口本身也可以在 AIDL 文件中使用
```

10. AIDL 解决了什么问题？

AIDL 出现的目的就是为了解决一对多并发及时通信了

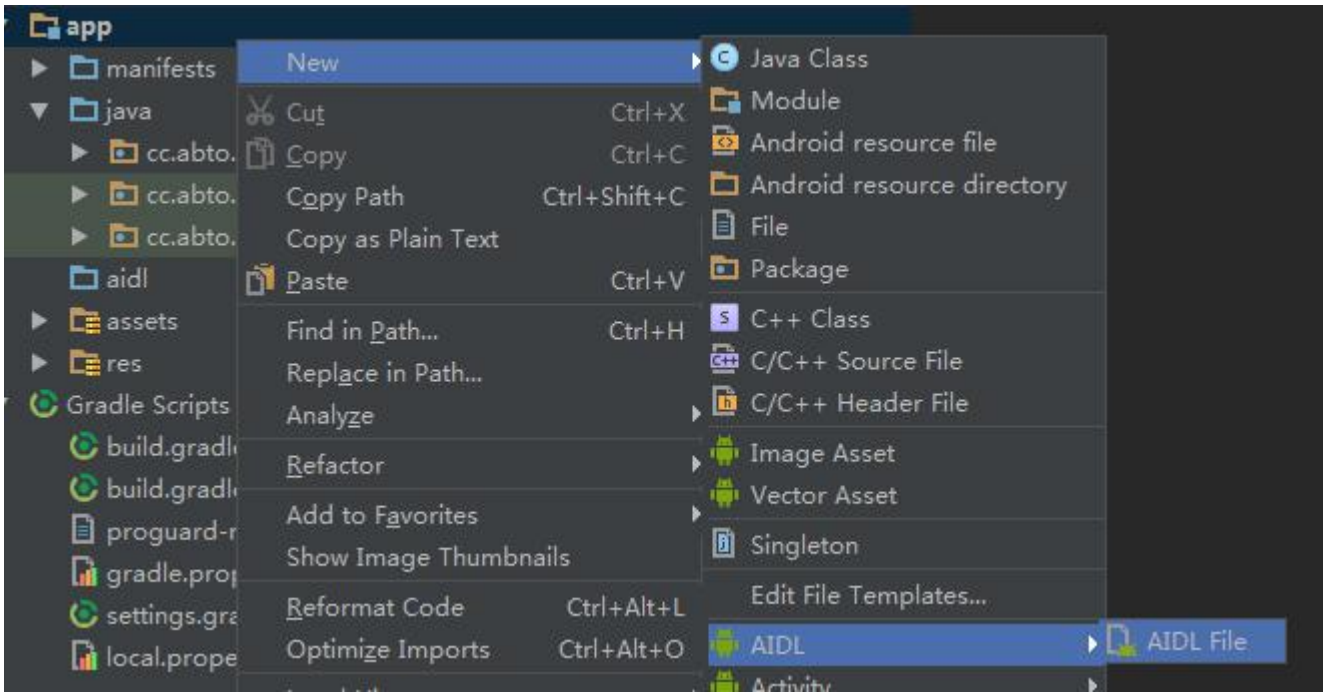
11. AIDL 如何使用？

https://www.jianshu.com/p/d1fac6ccee98
https://www.jianshu.com/p/a5c73da2e9be

AIDL 的使用

(1)在 Android Studio 中新建 AIDL 文件：

在 android studio 2.0 里面使用 AIDL，因为是两个 APP 交互么，所以当然要两个 APP 啦，我们在第一个工程目录右键



输入名称后，sutido 就帮我们创建了一个 AIDL 文件。

```
// IMyAidlInterface.aidlpackage cc.abto.demo;
```

```
// Declare any non-default types here with import statements

interface IMyAidlInterface {

    /**

     * Demonstrates some basic types that you can use as parameters

     * and return values in AIDL.

     */

    void basicTypes(int anInt, long aLong, boolean aBoolean, float aFloat,

            double aDouble, String aString);

}
```

上面就是 studio 帮我生成的 aidl 文件。basicTypes 这个方法可以无视，看注解知道这个方法只是告诉你在 AIDL 中你可以使用的基本类型（int, long, boolean, float, double, String），因为这里是要跨进程通讯的，所以不是随便你自己定义的一个类型就可以在 AIDL 使用的，这些后面会说。

(2)我们在 AIDL 文件中定义一个我们要提供给第二个 APP 使用的接口。

```
interface IMyAidlInterface {

   String getName();

}
```

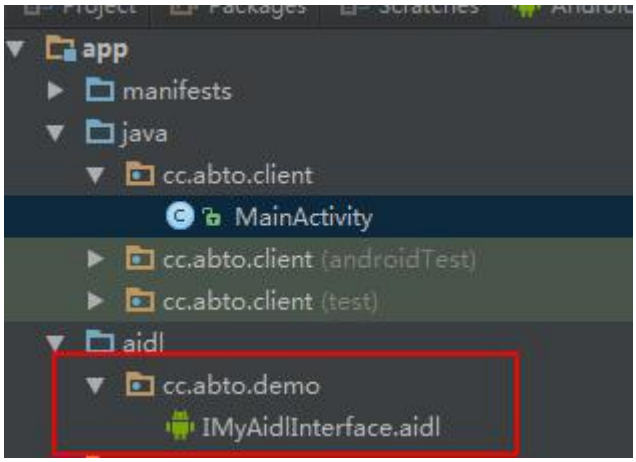
(3)定义好之后，就可以 sycn project 一下，

(4)然后新建一个 service。在 service 里面创建一个内部类，继承

你刚才创建的 AIDL 的名称里的 Stub 类，并实现接口方法，在 onBind 返回内部类的实例。

```
public class MyService extends Service{

    public MyService() { }

    @Override

    public IBinder onBind(Intent intent)

    {   return new MyBinder();   }



    class MyBinder extends IMyAidlInterface.Stub {

       @Override

        public String getName() throws RemoteException

        {  return "test"; }

    }

}
```

(5)接下来，将我们的 AIDL 文件拷贝到第二个项目，然后 sycn project 一下工程。



(6)这边的包名要跟第一个项目的一样哦,这之后在 Activity 中绑定服务。

```
public class MainActivity extends AppCompatActivity{

    private IMyAidlInterface iMyAidlInterface;

    @Override

    protected void onCreate(Bundle savedInstanceState){

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        bindService(new Intent("cc.abto.server"), new ServiceConnection() {

            @Override

            public void onServiceConnected(ComponentName name, IBinder service) {

                iMyAidlInterface = IMyAidlInterface.Stub.asInterface(service);

            }

            @Override

            public void onServiceDisconnected(ComponentName name) {

            }

        }, BIND_AUTO_CREATE);

    }

    public void onClick(View view){

        try{

            Toast.makeText(MainActivity.this, iMyAidlInterface.getName(),
Toast.LENGTH_SHORT).show();

        }catch (RemoteException e){
```

```
            e.printStackTrace();

        }

    }

}
```

这边我们通过隐式意图来绑定 service，在 onServiceConnected 方法中通过 IMyAidlInterface.Stub.asInterface(service) 获取 iMyAidlInterface 对象，然后在 onClick 中调用 iMyAidlInterface.getName()。