一、JNI开发流程

1. 编写java类,声明了native方法
2. 编写native代码
3. 将native代码编译成so文件
4. 在java类中引入so库,调用native方法

二、native方法命名

```
extern "C"
JNIEXPORT void JNICALL
Java_com_xfhy_jnifirst_MainActivity_callJavaMethod(JNIEnv *env, jobject thiz) {

}
```

函数命名规则:

```
Java_类全路径_方法名
```

函数参数:

1. JNIEnv*是定义任意native函数的第一个参数,是指向JNI环境的指针,可以通过它来访问JNI提供的接口方法.
2. jobject: 表示Java对象中的this. 如果是静态方法则是用jclass
3. JNIEXPORT和JNICALL: 它们是JNI中所定义的宏,可以在jni.h这个头文件中查找到.

三、JNI数据类型及与Java数据类型的映射关系

首先在Java代码里写个native方法声明,然后alt+enter让AS帮忙创建一个native方法.看看类型对应关系

```
public static native void ginsengTest(short s, int i, long l, float f, double d,
char c,
                                       boolean z, byte b, String str, Object obj,
MyClass p, int[] arr);
```

```
Java_com_xfhy_jnifirst_MainActivity_ginsengTest(JNIEnv *env, jclass clazz, jshort
s, jint i, jlong l, jfloat f, jdouble d, jchar c,  jboolean z, jbyte b, jstring
str, jobject obj, jobject p, jintArray arr) {

}
```

上面的方法定义上看,除了JNIEnv和jclass之外,其他都是一一对应的.

可以总结出如下表格:

「Java和JNI的类型对照表」

java 类型	Native 类型	符号属性	字长
boolean	jboolean	无符号	8 位
byte	jbyte	有符号	8 位
char	jchar	无符号	16 位
short	jshort	有符号	16 位
int	jint	有符号	32 位
long	jlong	有符号	64 位
float	jfloat	有符号	32 位
double	jdouble	有符号	64 位

「引用类型对照表」

java 类型	Native 类型
java.lang.Class	jclass
java.lang.Throwable	jthrowable
java.lang.String	jstring
java.lang.Object[]	jobjectArray
Boolean[]	jbooleanArray
Byte[]	jbyteArray
Char[]	jcharArray
Short[]	jshortArray
int[]	jintArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray

3.1 基本数据类型

基本数据类型其实就是将C/C++中的基本类型用typedef重新定义了一个新的名字,在JNI中可以直接访问.

```
/* Primitive types that match up with Java equivalents. */
typedef uint8_t  jboolean; /* unsigned 8 bits */
typedef int8_t   jbyte;    /* signed 8 bits */
typedef uint16_t jchar;    /* unsigned 16 bits */
typedef int16_t  jshort;   /* signed 16 bits */
typedef int32_t  jint;     /* signed 32 bits */
typedef int64_t  jlong;    /* signed 64 bits */
typedef float    jfloat;   /* 32-bit IEEE 754 */
typedef double   jdouble;  /* 64-bit IEEE 754 */
```

3.2 引用数据类型

如果使用C++语言编写,则所有引用派生自jobject.

```
class _jobject {};
class _jclass : public _jobject {};
class _jstring : public _jobject {};
class _jarray : public _jobject {};
class _jobjectArray : public _jarray {};
class _jbooleanArray : public _jarray {};
class _jbyteArray : public _jarray {};
class _jcharArray : public _jarray {};
class _jshortArray : public _jarray {};
class _jintArray : public _jarray {};
class _jlongArray : public _jarray {};
class _jfloatArray : public _jarray {};
class _jdoubleArray : public _jarray {};
class _jthrowable : public _jobject {};
```

JNI使用C语言时,所有引用类型使用jobject.

四、JNI 字符串处理

4.1 native操作JVM的数据结构

JNI会把Java中所有对象当做一个C指针传递到本地方法中,这个指针指向JVM内部数据结构,而内部的数据结构在内存中的存储方式是不可见的.只能从JNIEnv指针指向的函数表中选择合适的JNI函数来操作JVM中的数据结构.

比如native访问java.lang.String 对应的JNI类型jstring时,不能像访问基本数据类型那样使用,因为它是一个Java的引用类型,所以在本地代码中只能通过类似GetStringUTFChars这样的JNI函数来访问字符串的内容.

4.2 字符串操作

```
//调用
String result = operateString("待操作的字符串");
Log.d("xfhy", result);

//定义
public native String operateString(String str);
```

```
extern "C"
JNIEXPORT jstring JNICALL
Java_com_xfhy_jnifirst_MainActivity_operateString(JNIEnv *env, jobject thiz,
jstring str) {
    //从java的内存中把字符串拷贝出来  在native使用
    const char *strFromJava = (char *) env->GetStringUTFChars(str, NULL);
    if (strFromJava == NULL) {
        //必须空检查
        return NULL;
    }

    //将strFromJava拷贝到buff中,待会儿好拿去生成字符串
    char buff[128] = {0};
    strcpy(buff, strFromJava);
    strcat(buff, " 在字符串后面加点东西");

    //释放资源
    env->ReleaseStringUTFChars(str, strFromJava);

    //自动转为Unicode
    return env->NewStringUTF(buff);
}
```

输出为: `待操作的字符串 在字符串后面加点东西`

4.2.1 native中获取JVM字符串

operateString函数接收一个jstring类型的参数str,jstring是指向JVM内部的一个字符串,不能直接使用.首先需要将jstring转为C风格的字符串类型char*,然后才能使用,这里必须使用合适的JNI函数来访问JVM内部的字符串数据结构.(上例中使用的是GetStringUTFChars)

GetStringUTFChars(jstring string, jboolean* isCopy)参数说明:

- string : jstring,Java传递给native代码的字符串指针
- isCopy : 一般情况下传NULL. 取值是 `JNI_TRUE` 和 `JNI_FALSE` .如果是 `JNI_TRUE` 则会返回JVM内部源字符串的一份拷贝,并为新产生的字符串分配内存空间.如果是 `JNI_FALSE` 则返回JVM内部源字符串的指针,意味着可以在native层修改源字符串,但是不推荐修改,Java字符串的原则是不能修改的.

Java中默认是使用Unicode编码,C/C++默认使用UTF编码,所以在native层与java层进行字符串交流的时候需要进行编码转换.GetStringUTFChars就刚好可以把jstring指针(指向JVM内部的Unicode字符序列)的字符串转换成一个UTF-8格式的C字符串.

4.2.2 异常处理

在使用GetStringUTFChars的时候,返回的值可能为NULL,这时需要处理一下,否则继续往下面走的话,使用这个字符串的时候会出现问题.因为调用这个方法时,是拷贝,JVM为新生成的字符串分配内存空间,当内存空间不够分配的时候,会导致调用失败.调用失败就会返回NULL,并抛出OutOfMemoryError.JNI遇到未决的异常不会改变程序的运行流程,还是会继续往下走.

4.2.3 释放字符串资源

native不像Java,我们需要手动释放申请的内存空间.GetStringUTFChars调用时会新申请一块空间用来装拷贝出来的字符串,这个字符串用来方便native代码访问和修改之类的. 既然有内存分配,那么就必须手动释放,释放方法是ReleaseStringUTFChars.可以看到和GetStringUTFChars是一一对应的,配对的.

4.2.4 构建字符串

使用NewStringUTF函数可以构建出一个jstring,需要传入一个char *类型的C字符串.它会构建一个新的java.lang.String字符串对象,并且会自动转换成Unicode编码. 如果JVM不能为构造java.lang.String分配足够的内存,则会抛出一个OutOfMemoryError异常,并返回NULL.

4.2.5 其他字符串操作函数

1. GetStringChars和ReleaseStringChars: 这对函数和Get/ReleaseStringUTFChars函数功能类似,用于获取和释放的字符串是以Unicode格式编码的.
2. GetStringLength: 获取Unicode字符串(jstring)的长度. UTF-8编码的字符串是以`\0`结尾,而Unicode的不是,所以这里需要单独区分开.
3. **「GetStringUTFLength」**: 获取UTF-8编码字符串的长度,就是获取C/C++默认编码字符串的长度. 还可以使用标准C函数**「strlen」**来获取其长度.
4. strcat: 拼接字符串,标准C函数. eg: `strcat(buff, "xfhy");` 将xfhy添加到buff的末尾.
5. GetStringCritical和ReleaseStringCritical: 为了增加直接传回指向Java字符串的指针的可能性(而不是拷贝).在这2个函数之间的区域,是绝对不能调用其他JNI函数或者让线程阻塞的native函数.否则JVM可能死锁. 如果有一个字符串的内容特别大,比如1M,且只需要读取里面的内容打印出来,此时比较适合用该对函数,可直接返回源字符串的指针.
6. GetStringRegion和GetStringUTFRegion: 获取Unicode和UTF-8字符串中指定范围的内容(eg: 只需要1-3索引处的字符串),这对函数会将源字符串复制到一个预先分配的缓冲区(自己定义的char数组)内.

```
extern "C"
JNIEXPORT jstring JNICALL
Java_com_xfhy_jnifirst_MainActivity_operateString(JNIEnv *env, jobject thiz,
jstring str) {
    //方式2  用GetStringUTFRegion方法将JVM中的字符串拷贝到C/C++的缓冲区(数组)中
    //获取Unicode字符串长度
    int len = env->GetStringLength(str);
    char buff[128];
    env->GetStringUTFRegion(str, 0, len, buff);
    LOGI("-------------- %s", buff);

    //自动转为Unicode
    return env->NewStringUTF(buff);
}
```

GetStringUTFRegion会进行越界检查,越界会抛StringIndexOutOfBoundsException异常.GetStringUTFRegion其实和GetStringUTFChars有点相似,但是GetStringUTFRegion内部不会分配内存,不会抛出内存溢出异常. 由于其内部没有分配内存,所以也没有类似Release这样的函数来释放资源.

4.2.6 字符串 小结

1. Java字符串转C/C++字符串: 使用GetStringUTFChars函数,必须调用ReleaseStringUTFChars释放内存
2. 创建Java层需要的Unicode字符串,使用NewStringUTF函数
3. 获取C/C++字符串长度,使用GetStringUTFLength或者strlen函数
4. 对于小字符串,GetStringRegion和GetStringUTFRegion这2个函数是最佳选择,因为缓冲区数组可以被编译器提取分配,不会产生内存溢出的异常.当只需要处理字符串的部分数据时,也还是不错.它们提供了开始索引和子字符串长度值,复制的消耗也是非常小
5. 获取Unicode字符串和长度,使用GetStringChars和GetStringLength函数

五、数组操作

5.1 基本类型数组

基本类型数组就是JNI中的基本数据类型组成的数组,可以直接访问.下面举个简单例子,int数组求和:

```
//MainActivity.java
public native int sumArray(int[] array);
```

```
extern "C"
JNIEXPORT jint JNICALL
Java_com_xfhy_jnifirst_MainActivity_sumArray(JNIEnv *env, jobject thiz, jintArray
array) {
    //数组求和
    int result = 0;

    //方式1  推荐使用
```

```
    jint arr_len = env->GetArrayLength(array);
    //动态申请数组
    jint *c_array = (jint *) malloc(arr_len * sizeof(jint));
    //初始化数组元素内容为0
    memset(c_array, 0, sizeof(jint) * arr_len);
    //将java数组的[0-arr_len)位置的元素拷贝到c_array数组中
    env->GetIntArrayRegion(array, 0, arr_len, c_array);
    for (int i = 0; i < arr_len; ++i) {
        result += c_array[i];
    }
    //动态申请的内存 必须释放
    free(c_array);

    return result;
}
```

C层拿到jintArray之后首先需要获取它的长度,然后动态申请一个数组(因为Java层传递过来的数组长度是不定的,所以这里需要动态申请C层数组),这个数组的元素是jint类型的.malloc是一个经常使用的拿来申请一块连续内存的函数,申请之后的内存是需要手动调用free释放的.然后就是调用GetIntArrayRegion函数将Java层数组拷贝到C层数组中,然后求和.

另一种求和方式:

```
extern "C"
JNIEXPORT jint JNICALL
Java_com_xfhy_jnifirst_MainActivity_sumArray(JNIEnv *env, jobject thiz, jintArray
array) {
    //数组求和
    int result = 0;

    //方式2
    //此种方式比较危险,GetIntArrayElements会直接获取数组元素指针,是可以直接对该数组元素进行
修改的.
    jint *c_arr = env->GetIntArrayElements(array, NULL);
    if (c_arr == NULL) {
        return 0;
    }
    c_arr[0] = 15;
    jint len = env->GetArrayLength(array);
    for (int i = 0; i < len; ++i) {
        //result += *(c_arr + i); 写成这种形式,或者下面一行那种都行
        result += c_arr[i];
    }
    //有Get,一般就有Release
    env->ReleaseIntArrayElements(array, c_arr, 0);

    return result;
}
```

直接通过GetIntArrayElements函数拿到原数组元素指针,直接操作,就可以拿到元素求和.看起来要简单很多,但是这种方式我个人觉得是有点危险的,毕竟这种可以在C层直接进行源数组修改.GetIntArrayElements的第二个参数一般传NULL,传递 `JNI_TRUE` 是返回临时缓冲区数组指针(即拷贝一个副本),传递 `JNI_FALSE` 则是返回原始数组指针.

这里简单小结一下: 推荐使用「**Get/SetArrayRegion**」函数来操作数组元素是效率最高的.自己动态申请数组,自己操作,免得影响Java层.

5.2 对象数组

对象数组中的元素是一个类的实例或其他数组的引用,不能直接访问Java传递给JNI层的数组.

操作对象数组稍显复杂,下面举一个例子,在native层创建一个二维数组,且赋值并返回给Java层使用.(ps: 第二维是int[],它属于对象)

```
public native int[][] init2DArray(int size);

//交给native层创建->Java打印输出
int[][] init2DArray = init2DArray(3);
for (int i = 0; i < 3; i++) {
    for (int i1 = 0; i1 < 3; i1++) {
        Log.d("xfhy", "init2DArray[" + i + "][" + i1 + "]" + " = " +
init2DArray[i][i1]);
    }
}
```

```
extern "C"
JNIEXPORT jobjectArray JNICALL
Java_com_xfhy_jnifirst_MainActivity_init2DArray(JNIEnv *env, jobject thiz, jint
size) {
    //创建一个size*size大小的二维数组

    //jobjectArray是用来装对象数组的    Java数组就是一个对象 int[]
    jclass classIntArray = env->FindClass("[I");
    if (classIntArray == NULL) {
        return NULL;
    }
    //创建一个数组对象,元素为classIntArray
    jobjectArray result = env->NewObjectArray(size, classIntArray, NULL);
    if (result == NULL) {
        return NULL;
    }
    for (int i = 0; i < size; ++i) {
        jint buff[100];
        //创建第二维的数组 是第一维数组的一个元素
        jintArray intArr = env->NewIntArray(size);
        if (intArr == NULL) {
            return NULL;
        }
        for (int j = 0; j < size; ++j) {
            //这里随便设置一个值
            buff[j] = 666;
        }
        //给一个jintArray设置数据
        env->SetIntArrayRegion(intArr, 0, size, buff);
        //给一个jobjectArray设置数据 第i索引,数据位intArr
        env->SetObjectArrayElement(result, i, intArr);
        //及时移除引用
```

```
        env->DeleteLocalRef(intArr);
    }

    return result;
}
```

分析一下

1. 首先是利用FindClass函数找到java层int[]对象的class,这个class是需要传入NewObjectArray创建对象数组的.调用NewObjectArray函数之后,即可创建一个对象数组,大小是size,元素类型是前面获取到的class.
2. 进入for循环构建size个int数组,构建int数组需要使用NewIntArray函数.可以看到我构建了一个临时的buff数组,然后大小是随便设置的,这里是为了示例,其实可以用malloc动态申请空间,免得申请100个空间,可能太大或者太小了.整buff数组主要是拿来给生成出来的jintArray赋值的,因为jintArray是Java的数据结构,咱native不能直接操作,得调用SetIntArrayRegion函数,将buff数组的值复制到jintArray数组中.
3. 然后调用SetObjectArrayElement函数设置jobjectArray数组中某个索引处的数据,这里将生成的jintArray设置进去.
4. 最后需要将for里面生成的jintArray及时移除引用.创建的jintArray是一个JNI局部引用,如果局部引用太多的话,会造成JNI引用表溢出.

在JNI中,只要是jobject的子类就属于引用变量,会占用引用表的空间. 而基础数据类型jint,jfloat,jboolean等是不会占用引用表空间的,不需要释放.

六、native调Java方法

在JVM中,运行一个Java程序时,会先将运行时需要用到的所有相关class文件加载到JVM中,并按需加载,提高性能和节约内存.当我们调用一个类的静态方法之前,JVM会先判断该类是否已经加载,如果没有被ClassLoader加载到JVM中,会去classpath路径下查找该类.找到了则加载该类,没有找到则报ClassNotFoundException异常.

6.1 native调用Java静态方法

先写一个MyJNIClass.java类

```
public class MyJNIClass {

    public int age = 18;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public static String getDes(String text) {
        if (text == null) {
            text = "";
        }
        return "传入的字符串长度是 :" + text.length() + "  内容是 : " + text;
```

```
    }

}
```

然后去native调用getDes方法,为了复杂一点,这个getDes方法不仅有入参,还有返参.

```
extern "C"
JNIEXPORT void JNICALL
Java_com_xfhy_allinone_jni_CallMethodActivity_callJavaStaticMethod(JNIEnv *env,
jobject thiz) {
    //调用某个类的static方法

    //JVM使用一个类时,是需要先判断这个类是否被加载了,如果没被加载则还需要加载一下才能使用
    //1. 从classpath路径下搜索MyJNIClass这个类,并返回该类的Class对象
    jclass clazz = env->FindClass("com/xfhy/allinone/jni/MyJNIClass");
    //2. 从clazz类中查找getDes方法 得到这个静态方法的方法id
    jmethodID mid_get_des = env->GetStaticMethodID(clazz, "getDes", "
(Ljava/lang/String;)Ljava/lang/String;");
    //3. 构建入参,调用static方法,获取返回值
    jstring str_arg = env->NewStringUTF("我是xfhy");
    jstring result = (jstring) env->CallStaticObjectMethod(clazz, mid_get_des,
str_arg);
    const char *result_str = env->GetStringUTFChars(result, NULL);
    LOGI("获取到Java层返回的数据 : %s", result_str);

    //4. 移除局部引用
    env->DeleteLocalRef(clazz);
    env->DeleteLocalRef(str_arg);
    env->DeleteLocalRef(result);
}
```

看起来好像比较简单,native的代码其实是短小精悍,这里面涉及的知识点挺多.

1. 首先是调用FindClass函数,传入Class描述符(Java类的全类名,这里在AS中输入MyJNIClass时会有提示补全,直接enter即可补全),找到该类,得到jclass类型.
2. 然后通过GetStaticMethodID找到该方法的id,传入方法签名,得到jmethodID类型的引用(存储方法的引用).(这里输入getDes时,AS也会有补全功能,按enter直接把签名带出来了,真tm方便). 这里先使用AS的自动补全功能把方法签名带出来,后面会详细说这个方法签名是什么.
3. 构建入参,然后调用CallStaticObjectMethod去调用Java类里面的静态方法,然后传入参数,返回的直接就是Java层返回的数据. 其实这里的CallStaticObjectMethod是调用的引用类型的静态方法,与之相似的还有: CallStaticVoidMethod(无返参),CallStaticIntMethod(返参是Int),CallStaticFloatMethod,CallStaticShortMethod.他们的用法是一致的.
4. 移除局部引用

函数结束后,JVM会自动释放所有局部引用变量所占的内存空间. 这里还是手动释放一下比较安全,因为在JVM中维护着一个引用表,用于存储局部和全局引用变量. 经测试发现在Android低版本(我测试的是Android 4.1)上,这个表的最大存储空间是512个引用,,当超出这个数量时直接崩溃.当我在高版本,比如小米 8(安卓10)上,这个引用个数可以达到100000也不会崩溃,只是会卡顿一下.可能是硬件比当年更牛逼了,默认值也跟着改了.

下面是引用表溢出时所报的错误:

```
E/dalvikvm: JNI ERROR (app bug): local reference table overflow (max=512)
E/dalvikvm: Failed adding to JNI local ref table (has 512 entries)
E/dalvikvm: VM aborting
A/libc: Fatal signal 11 (SIGSEGV) at 0xdeadd00d (code=1), thread 2561
(m.xfhy.allinone)
```

6.2 native调用Java实例方法

下面演示在native层创建Java实例,并调用该实例的方法,大致上是和上面调用静态方法差不多的.

```
extern "C"
JNIEXPORT void JNICALL
Java_com_xfhy_allinone_jni_CallMethodActivity_createAndCallJavaInstanceMethod(JN
IEnv *env, jobject thiz) {
    //构建一个类的实例,并调用该实例的方法

    //特别注意 : 正常情况下,这里的每一个获取出来都需要判空处理,我这里为了展示核心代码,就不判空
了
    //这里输入MyJNIClass之后AS会提示,然后按Enter键即可自动补全
    jclass clazz = env->FindClass("com/xfhy/allinone/jni/MyJNIClass");
    //获取构造方法的方法id
    jmethodID mid_construct = env->GetMethodID(clazz, "<init>", "()V");
    //获取getAge方法的方法id
    jmethodID mid_get_age = env->GetMethodID(clazz, "getAge", "()I");
    jmethodID mid_set_age = env->GetMethodID(clazz, "setAge", "(I)V");
    jobject jobj = env->NewObject(clazz, mid_construct);

    //调用方法setAge
    env->CallVoidMethod(jobj, mid_set_age, 20);
    //再调用方法getAge 获取返回值 打印输出
    jint age = env->CallIntMethod(jobj, mid_get_age);
    LOGI("获取到 age = %d", age);

    //凡是使用是jobject的子类,都需要移除引用
    env->DeleteLocalRef(clazz);
    env->DeleteLocalRef(jobj);
}
```

这里还是使用上面的MyJNIClass类

1. 找到该类的class
2. 获取构造方法的id,获取需要调用方法的id. 其中获取构造方法时,方法名称固定写法就是`<init>`,然后后面是方法签名.
3. 利用NewObject()函数构建一个Java对象
4. 调用Java对象的setAge和getAge方法,获取返回值,打印结果.这里调用了CallIntMethod和CallVoidMethod,相信大家一看名字就能猜出来,这是干啥的.就是调用返回值是int和void的方法.和上面的CallStaticIntMethod类似.
5. 删除局部引用 !!!

6.3 方法签名

调用Java方法需要使用jmethodID,每次获取jmethodID都需要传入方法签名.明明已经传入了方法名称,感觉就已经可以调这个方法了啊,为啥还需要传入一个方法签名? 因为Java方法存在重载,可能方法名是相同的,但是参数不同.所以这里必须传入方法签名以区别.

方法签名格式为: **「(形参参数类型列表)返回值」** , 引用类型以L开头,后面是类的全路径名.

Java基本类型与方法签名中的定义关系:

Java	Field Descriptor
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
int[]	[I

示例:

```
public String[] testString(boolean a, byte b, float f, double d, int i, int[]
array)

env->GetMethodID(clazz,"testString", "(ZBFDI[I)[Ljava/lang/String;")
```

6.4 native调用Java方法 小结

1. 获取一个类的Class实例,得使用FindClass函数,传入类描述符.JVM会从classpath目录下去查找该类.
2. 创建一个类的实例是使用NewObject方法,需传入class引用和构造方法的id.
3. 局部引用(只要是jobject的子类就属于引用变量,会占用引用表的空间)一定记得及时移除掉,否则会引用表溢出.低版本上很容易就溢出了(低版本上限512).
4. 方法签名格式: **「(形参参数类型列表)返回值」**
5. 在使用FindClass和GetMethodID的时候,一定得判空.基操.
6. 获取实例方法ID,使用GetMethodID函数;获取静态方法ID,使用GetStaticMethodID函数;获取构造方法ID,方法名称使用`<init>`
7. 调用实例方法使用CallXXMethod函数,XX表示返回数据类型.`eg:CallIntMethod()`;调用静态方法使用CallStaticXXMethod函数.

七、NDK Crash错误定位

当NDK发生错误某种致命的错误的时候,会导致APP闪退.这类错误非常不好查问题,比如内存地址访问错误,使用野指针,内存泄露,堆栈溢出,数字除0等native错误都会导致APP崩溃.

虽然这些NDK错误不好排查,但是我们在NDK错误发生后,拿到logcat输出的堆栈日志,再结合下面的2款调试工具--addr2line和ndk-stack,能够精确地定位到相应发生错误的代码行数,然后迅速找到问题.

首先到ndk目录下,来到`sdk/ndk/21.0.6113669/toolchains/`目录,我本地的目录如下,可以看到NDK交叉编译器工具链的目录结构.

```
aarch64-linux-android-4.9
arm-linux-androideabi-4.9
llvm
renderscript
x86-4.9
x86_64-4.9
```

其中ndk-stack放在$NDK_HOME目录下，与ndk-build同级目录。addr2line在ndk的交叉编译器工具链目录下.NDK针对不同的CPU架构实现了多套工具.在使用addr2line工具时,需要根据当前手机cpu架构来选择.我的手机是aarch64的,则使用`aarch64-linux-android-4.9`目录下的工具. 查看手机的cpu信息的命令:`adb shell cat /proc/cpuinfo`

在介绍两款调试工具之前,我们得先写好能崩溃的native代码,方便看效果. 我在demo的`native-lib.cpp`里面写了如下代码:

```
void willCrash() {
    JNIEnv *env = NULL;
    int version = env->GetVersion();
}

extern "C"
JNIEXPORT void JNICALL
Java_com_xfhy_allinone_jni_CallMethodActivity_nativeCrashTest(JNIEnv *env,
jobject thiz) {
    LOGI("崩溃前");
    willCrash();
    //后面的代码是执行不到的,因为崩溃了
    LOGI("崩溃后");
    printf("oooo");
}
```

这段代码,是很明显的空指针错误.然后运行起来,错误日志如下:

```
2020-06-07 17:05:25.230 12340-12340/? A/DEBUG: *** *** *** *** *** *** *** ***
*** *** *** *** *** *** *** ***
2020-06-07 17:05:25.230 12340-12340/? A/DEBUG: Build fingerprint:
'Xiaomi/dipper/dipper:10/QKQ1.190828.002/V11.0.8.0.QEACNXM:user/release-keys'
2020-06-07 17:05:25.230 12340-12340/? A/DEBUG: Revision: '0'
2020-06-07 17:05:25.230 12340-12340/? A/DEBUG: ABI: 'arm64'
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG: Timestamp: 2020-06-07
17:05:25+0800
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG: pid: 11527, tid: 11527, name:
m.xfhy.allinone  >>> com.xfhy.allinone <<<
```

```
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG: uid: 10319
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG: signal 11 (SIGSEGV), code 1
(SEGV_MAPERR), fault addr 0x0
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG: Cause: null pointer dereference
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG:     x0  0000000000000000  x1
0000007fd29ffd40  x2  0000000000000005  x3  0000000000000003
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG:     x4  0000000000000000  x5
8080800000000000  x6  fefeff6fb0ce1f1f  x7  7f7f7f7fffff7f7f
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG:     x8  0000000000000000  x9
a95a4ec0adb574df  x10 0000007fd29ffee0  x11 000000000000000a
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG:     x12 0000000000000018  x13
ffffffffffffffff  x14 0000000000000004  x15 ffffffffffffffff
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG:     x16 0000006fc6476c50  x17
0000006fc64513cc  x18 00000070b21f6000  x19 000000702d069c00
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG:     x20 0000000000000000  x21
000000702d069c00  x22 0000007fd2a00720  x23 0000006fc6ceb127
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG:     x24 0000000000000004  x25
00000070b1cf2020  x26 000000702d069cb0  x27 0000000000000001
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG:     x28 0000007fd2a004b0  x29
0000007fd2a00420
2020-06-07 17:05:25.237 12340-12340/? A/DEBUG:     sp  0000007fd2a00410  lr
0000006fc64513bc  pc  0000006fc64513e0
2020-06-07 17:05:25.788 12340-12340/? A/DEBUG: backtrace:
2020-06-07 17:05:25.788 12340-12340/? A/DEBUG:       #00 pc 00000000000113e0
/data/app/com.xfhy.allinone-4VScOmUWz8wLqqwBWZCP2w==/lib/arm64/libnative-lib.so
(_JNIEnv::GetVersion()+20) (BuildId: b1130c28a8b45feda869397e55c5b6d75441oc8d)
2020-06-07 17:05:25.788 12340-12340/? A/DEBUG:       #01 pc 00000000000113b8
/data/app/com.xfhy.allinone-4VScOmUWz8wLqqwBWZCP2w==/lib/arm64/libnative-lib.so
(willCrash()+24) (BuildId: b1130c28a8b45feda869397e55c5b6d75441oc8d)
2020-06-07 17:05:25.788 12340-12340/? A/DEBUG:       #02 pc 0000000000011450
/data/app/com.xfhy.allinone-4VScOmUWz8wLqqwBWZCP2w==/lib/arm64/libnative-lib.so
(Java_com_xfhy_allinone_jni_CallMethodActivity_nativeCrashTest+84) (BuildId:
b1130c28a8b45feda869397e55c5b6d75441oc8d)
2020-06-07 17:05:25.788 12340-12340/? A/DEBUG:       #03 pc 000000000013f350
/apex/com.android.runtime/lib64/libart.so (art_quick_generic_jni_trampoline+144)
(BuildId: 2bc2e11d57f839316bf2a42bbfdf943a)
2020-06-07 17:05:25.788 12340-12340/? A/DEBUG:       #04 pc 0000000000136334
/apex/com.android.runtime/lib64/libart.so (art_quick_invoke_stub+548) (BuildId:
2bc2e11d57f839316bf2a42bbfdf943a)
```

看到这种错误,首先找到关键信息`Cause: null pointer dereference`,但是我们不知道发生在具体哪里,只知道是这个错误

7.1 addr2line

有了错误日志,现在我们使用工具addr2line来定位位置.

需要执行命令 `/Users/xfhy/development/sdk/ndk/21.0.6113669/toolchains/aarch64-linux-android-4.9/prebuilt/darwin-x86_64/bin/aarch64-linux-android-addr2line -e /Users/xfhy/development/AllInOne/app/libnative-lib.so 00000000000113e0 00000000000113b8`,这其中`-e`是指定so文件的位置,然后末尾的00000000000113e0和00000000000113b8是出错位置的汇编指令地址.

执行之后,结果如下:

```
/Users/xfhy/development/sdk/ndk/21.0.6113669/toolchains/llvm/prebuilt/darwin-
x86_64/sysroot/usr/include/jni.h:497
/Users/xfhy/development/AllInOne/app/src/main/cpp/native-lib.cpp:260
```

可以看到是`native-lib.cpp`的260行出的问题.我们只需要去找到这个位置,修复这个bug即可.完美,瞬间就找到了.

7.2 ndk-stack

还有一种更简单的方式,直接输入命令`adb logcat | ndk-stack -sym /Users/xfhy/development/AllInOne/app/build/intermediates/cmake/debug/obj/arm64-v8a`,末尾是so文件的位置.执行完命令,然后在手机上产生native错误就能在这个so文件中定位到这个错误点,如下:

```
********** Crash dump: **********
Build fingerprint:
'Xiaomi/dipper/dipper:10/QKQ1.190828.002/V11.0.8.0.QEACNXM:user/release-keys'
#00 0x00000000000113e0 /data/app/com.xfhy.allinone-oVu0tjta-
aW9LYaO8eoK1Q==/lib/arm64/libnative-lib.so (_JNIEnv::GetVersion()+20) (BuildId:
b1130c28a8b45feda869397e55c5b6d75441Oc8d)

                      _JNIEnv::GetVersion()


/Users/xfhy/development/sdk/ndk/21.0.6113669/toolchains/llvm/prebuilt/darwin-
x86_64/sysroot/usr/include/jni.h:497:14
#01 0x00000000000113b8 /data/app/com.xfhy.allinone-oVu0tjta-
aW9LYaO8eoK1Q==/lib/arm64/libnative-lib.so (willCrash()+24) (BuildId:
b1130c28a8b45feda869397e55c5b6d75441Oc8d)

                      willCrash()

                      /Users/xfhy/development/AllInOne/app/src/main/cpp/native-
lib.cpp:260:24
#02 0x0000000000011450 /data/app/com.xfhy.allinone-oVu0tjta-
aW9LYaO8eoK1Q==/lib/arm64/libnative-lib.so
(Java_com_xfhy_allinone_jni_CallMethodActivity_nativeCrashTest+84) (BuildId:
b1130c28a8b45feda869397e55c5b6d75441Oc8d)


Java_com_xfhy_allinone_jni_CallMethodActivity_nativeCrashTest

                      /Users/xfhy/development/AllInOne/app/src/main/cpp/native-
lib.cpp:267:5
```

可以看出是`willCrash()`方法出的错,它的代码行数是260行(甚至连列都打印出来了,,第24列).

八、JNI 引用

Java平时在新创建对象的时候,不需要管JVM是怎么申请内存的,使用完之后也不需要管JVM是如何释放内存的.而C++不同,需要我们手动申请和释放内存(new->delete,malloc->free). 在使用JNI时,由于本地代码不能直接通过引用操作JVM内部的数据结构,要进行这些操作必须调用相应的JNI接口间接操作JVM内部的数据内容.我们不需要关心JVM中对象的是如何存储的,只需要学习JVI中的三种不同引用即可.

8.1 JNI 局部引用

本地函数中通过NewLocalRef或调用FindClass、NewObject、GetObjectClass、NewCharArray等创建的引用,就是局部引用.

- 会阻止GC回收所引用的对象
- 不能跨线程使用
- 不在本地函数中跨函数使用
- 释放: 函数返回后局部引用所引用的对象会被JVM自动释放,也可以调用DeleteLocalRef释放

通常是在函数中创建并使用的.上面提到了: 局部引用在函数返回之后会自动释放,那么我们为啥还需要去管它(手动调用DeleteLocalRef释放)呢?

比如,开了一个for循环,里面不断地创建局部引用,那么这时就必须得使用DeleteLocalRef手动释放内存.不然局部引用会越来越多,导致崩溃(在Android低版本上局部引用表的最大数量有限制,是512个,超过则会崩溃).

还有一种情况,本地方法返回一个引用到Java层之后,如果Java层没有对返回的局部引用使用的话,局部引用就会被JVM自动释放.

8.2 JNI 全局引用

全局引用是基于局部引用创建的,调用NewGlobalRef方法.

- 会阻止GC回收所引用的对象
- 可以跨方法、跨线程使用
- JVM不会自动释放,需调用DeleteGlobalRef手动释放

8.3 JNI 弱全局引用

弱全局引用是基于局部引用或者全局引用创建的,调用NewWeakGlobalRef方法.

- 不会阻止GC回收所引用的对象
- 可以跨方法、跨线程使用
- 引用不会自动释放,只有在JVM内存不足时才会进行回收而被释放. 还有就是可以调用DeleteWeakGlobalRef手动释放.