JNI技术之手写JNIEnv与静态缓存与native异常（NDK第二十二节课）

JNI本身就是 JVM的，异常信息很像Java 合理吗？ 答：很合理

00-通用布局 与 通用native代码：

activity_layout.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="sortAction"
        android:text="数组排序"
        />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="staticCacheAction"
        android:text="静态缓存策略"
        />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="exceptionAction"
        android:text="native异常捕获"
        />

</LinearLayout>
```

native-lib.cpp：

```
#include <jni.h>
#include <string>
// 日志输出
#include <android/log.h>
#define TAG "Derry"
// __VA_ARGS__ 代表 ...的可变参数
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG, TAG,  __VA_ARGS__);
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, TAG,  __VA_ARGS__);
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, TAG,  __VA_ARGS__);

extern "C" JNIEXPORT jstring JNICALL
```

```
Java_com_derry_as_1jni_1project_MainActivity_stringFromJNI(
        JNIEnv *env,
        jobject /* this */) {
    std::string hello = "十年磨一剑";
    return env->NewStringUTF(hello.c_str());
}
```

01.数组排序

MainActivity.kt:

```
package com.derry.as_jni_project

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import android.view.View
import android.widget.Toast

// TODO 01.数组排序
class MainActivity : AppCompatActivity() {

    companion object {
        init {
            System.loadLibrary("native-lib")
        }
    }

    external fun stringFromJNI(): String // 默认提供的native函数
    external fun sort(arr: IntArray) // 数组排序

    /**
     * 数组排序
     */
    fun sortAction(view: View) {
        val arr = intArrayOf(11, 22, -3, 2, 4, 6, -15)
        sort(arr)
        for (element in arr) {
            Log.e("Derry",element.toString() + "\t")
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        Toast.makeText(this, stringFromJNI(), Toast.LENGTH_LONG).show()
    }
}
```

native-lib.cpp：

```
// 比较的函数
int compare(const jint *number1, const jint *number2){
    return *number1 - *number2;
}

extern "C"
```

```cpp
JNIEXPORT void JNICALL
Java_com_derry_as_1jni_1project_MainActivity_sort(JNIEnv *env, jobject thiz, jintArray arr) {
    // 对 arr 进行排序 （sort）
    jint* intArray = env->GetIntArrayElements(arr, nullptr);

    int length = env->GetArrayLength(arr);

    // 第一个参数：void* 数组的首地址
    // 第二个参数：数组的大小长度
    // 第三个参数：数组元素数据类型的大小
    // 第四个参数：数组的一个比较方法指针（Comparable）
    qsort(intArray, length, sizeof(int),
          reinterpret_cast<int (*)(const void *, const void *)>(compare));

    // 同步数组的数据给 java 数组 intArray 并不是 arr ，可以简单的理解为 copy
    // 0 : 既要同步数据给 arr ,又要释放 intArray，会排序
    // JNI_COMMIT:  会同步数据给 arr ，但是不会释放 intArray，会排序
    // JNI_ABORT: 不同步数据给 arr ，但是会释放 intArray，所以上层看到就并不会排序
    env->ReleaseIntArrayElements(arr, intArray, JNI_COMMIT);
}
```

02-静态缓存

MainActivity2：

```java
package com.derry.as_jni_project;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import androidx.annotation.Nullable;
import androidx.appcompat.app.AppCompatActivity;

// TODO 02.静态缓存
public class MainActivity2 extends AppCompatActivity {

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    static {
        System.loadLibrary("native-lib");
    }

    // 假设这里定义了一大堆变量
    static String name1 = "T1";
    static String name2 = "T2";
    static String name3 = "T3";
    static String name4 = "T4";
    static String name5 = "T5";
    static String name6 = "T6";
    // 省略更多 ....

    public static native void localCache(String name); // 普通局部缓存 弊端演示

    // 下面是静态缓存区域
```

```
public static native void initStaticCache(); // 初始化静态缓存
public static native void staticCache(String name); // 静态缓存
public static native void clearStaticCache(); // 清除静态缓存

/**
 * 静态缓存策略
 */
public void staticCacheAction(View view) {
    // 下面是局部缓存 的演示
    localCache("李元霸");
    Log.e("Derry", "name1:" + name1);


    // TODO 下面是静态缓存区域 ==============================
    initStaticCache(); // 先初始化静态缓存 (注意：如果是一个类去调用，就需要在构造函数中初始化)

    staticCache("李白"); // 再执行...
    Log.e("Derry", "静态缓存区域 name1:" + name1);
    Log.e("Derry", "静态缓存区域 name2:" + name2);
    Log.e("Derry", "静态缓存区域 name3:" + name3);
    Log.e("Derry", "静态缓存区域 name4:" + name4);
    Log.e("Derry", "静态缓存区域 name5:" + name5);
    Log.e("Derry", "静态缓存区域 name6:" + name6);

    staticCache("李小龙");
    Log.e("Derry", "静态缓存区域 name1:" + name1);
    Log.e("Derry", "静态缓存区域 name2:" + name2);
    Log.e("Derry", "静态缓存区域 name3:" + name3);
    Log.e("Derry", "静态缓存区域 name4:" + name4);
    Log.e("Derry", "静态缓存区域 name5:" + name5);
    Log.e("Derry", "静态缓存区域 name6:" + name6);

    staticCache("李连杰");
    Log.e("Derry", "静态缓存区域 name1:" + name1);
    Log.e("Derry", "静态缓存区域 name2:" + name2);
    Log.e("Derry", "静态缓存区域 name3:" + name3);
    Log.e("Derry", "静态缓存区域 name4:" + name4);
    Log.e("Derry", "静态缓存区域 name5:" + name5);
    Log.e("Derry", "静态缓存区域 name6:" + name6);

    staticCache("李贵");
    Log.e("Derry", "静态缓存区域 name1:" + name1);
    Log.e("Derry", "静态缓存区域 name2:" + name2);
    Log.e("Derry", "静态缓存区域 name3:" + name3);
    Log.e("Derry", "静态缓存区域 name4:" + name4);
    Log.e("Derry", "静态缓存区域 name5:" + name5);
    Log.e("Derry", "静态缓存区域 name6:" + name6);

    staticCache("李逵");
    Log.e("Derry", "静态缓存区域 name1:" + name1);
    Log.e("Derry", "静态缓存区域 name2:" + name2);
    Log.e("Derry", "静态缓存区域 name3:" + name3);
    Log.e("Derry", "静态缓存区域 name4:" + name4);
    Log.e("Derry", "静态缓存区域 name5:" + name5);
    Log.e("Derry", "静态缓存区域 name6:" + name6);

    staticCache("李鬼");
    Log.e("Derry", "静态缓存区域 name1:" + name1);
    Log.e("Derry", "静态缓存区域 name2:" + name2);
    Log.e("Derry", "静态缓存区域 name3:" + name3);
    Log.e("Derry", "静态缓存区域 name4:" + name4);
```

```
        Log.e("Derry", "静态缓存区域 name5:" + name5);
        Log.e("Derry", "静态缓存区域 name6:" + name6);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        clearStaticCache(); // 必须要清除静态缓存
    }
}
```

native-lib.cpp:

```
// 普通局部缓存 弊端演示
extern "C"
JNIEXPORT void JNICALL
Java_com_derry_as_1jni_1project_MainActivity2_localCache(JNIEnv *env, jclass clazz, jstring
name) {
    // 像 OpenCV WebRtc 等 大量使用局部缓存
    // name属性 赋值操作
    static jfieldID f_id = nullptr; // 局部缓存，这个方法会被多次调用，不需要反复的去获取 jfieldID
    if (f_id == nullptr) {
        f_id = env->GetStaticFieldID(clazz, "name1", "Ljava/lang/String;");
    }else {
        LOGD("fieldID是空的啊");
    }
    env->SetStaticObjectField(clazz, f_id, name);
}

// TODO ====================== 下面是全局静态缓存区域
// 全局静态缓存，在构造函数中初始化的时候会去缓存
static jfieldID f_name1_id = nullptr;
static jfieldID f_name2_id = nullptr;
static jfieldID f_name3_id = nullptr;
static jfieldID f_name4_id = nullptr;
static jfieldID f_name5_id = nullptr;
static jfieldID f_name6_id = nullptr;

// 1 先初始化静态缓存（类似于 在构造方法里面先初始化缓存）
extern "C"
JNIEXPORT void JNICALL
Java_com_derry_as_1jni_1project_MainActivity2_initStaticCache(JNIEnv *env, jclass clazz) {
    // 初始化全局静态缓存
    f_name1_id = env->GetStaticFieldID(clazz, "name1", "Ljava/lang/String;");
    f_name2_id = env->GetStaticFieldID(clazz, "name2", "Ljava/lang/String;");
    f_name3_id = env->GetStaticFieldID(clazz, "name3", "Ljava/lang/String;");
    f_name4_id = env->GetStaticFieldID(clazz, "name4", "Ljava/lang/String;");
    f_name5_id = env->GetStaticFieldID(clazz, "name5", "Ljava/lang/String;");
    f_name6_id = env->GetStaticFieldID(clazz, "name6", "Ljava/lang/String;");
    // 省略 ....
}

// 2 然后再 执行时，不会重复的去获取 jfieldID
extern "C"
JNIEXPORT void JNICALL
Java_com_derry_as_1jni_1project_MainActivity2_staticCache(JNIEnv *env, jclass clazz, jstring
name) {
    // 如果这个方法会反复的被调用，那么不会反复的去获取 jfieldID，因为是先初始化静态缓存，然后再执行此函
数的
    env->SetStaticObjectField(clazz, f_name1_id, name);
```

```
    env->SetStaticObjectField(clazz, f_name2_id, name);
    env->SetStaticObjectField(clazz, f_name3_id, name);
    env->SetStaticObjectField(clazz, f_name4_id, name);
    env->SetStaticObjectField(clazz, f_name5_id, name);
    env->SetStaticObjectField(clazz, f_name6_id, name);
}

// 3 最后要清除静态缓存
extern "C"
JNIEXPORT void JNICALL
Java_com_derry_as_1jni_1project_MainActivity2_clearStaticCache(JNIEnv *env, jclass clazz) {
    f_name1_id = nullptr;
    f_name2_id = nullptr;
    f_name3_id = nullptr;
    f_name4_id = nullptr;
    f_name5_id = nullptr;
    f_name6_id = nullptr;
    LOGD("静态缓存清除完毕...");
}
```

03-native异常捕获：

MainActivity3：

```
package com.derry.as_jni_project;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import androidx.annotation.Nullable;
import androidx.appcompat.app.AppCompatActivity;

import java.nio.file.NoSuchFileException;

// TODO 03.native异常捕获
public class MainActivity3 extends AppCompatActivity {

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    static {
        System.loadLibrary("native-lib");
    }

    // 这里定义变量
    static String name1 = "T1";

    // 下面是异常处理
    public static native void exception();
    public static native void exception2() throws NoSuchFieldException;
    public static native void exception3();

    /**
     * native异常捕获
     */
```

```
    public void exceptionAction(View view) {
        exception();

        try {
            exception2();
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
            Log.d("Derry", "Java层的 exception2 异常被我捕获到了...");
        }

        exception3();
    }

    // 此函数是 让 native层来调用的函数
    public static void show() throws Exception {
        Log.d("Derry", "show: 111");
        Log.d("Derry", "show: 222");
        Log.d("Derry", "show: 333");
        Log.d("Derry", "show: 444");
        Log.d("Derry", "show: 555");

        throw new NullPointerException("我是java中的抛出的异常，我的show方法里面发送了Java语法错
误");
    }
}
```

native-lib.cpp:

```
// TODO 03.native异常捕获  ======================
// 异常方式一：  【C++处理时异常】 扭转乾坤
extern "C"
JNIEXPORT void JNICALL
Java_com_derry_as_1jni_1project_MainActivity3_exception(JNIEnv *env, jclass clazz) {
    // 假设现在想操作name999，没有name999就会在native层奔溃掉
    jfieldID f_id = env->GetStaticFieldID(clazz, "name999", "Ljava/lang/String;");

    // 共两种方式 之 方式一
    // 补救措施 ，name999 拿不到报错的话，那么我就拿 name0

    jthrowable throwable = env->ExceptionOccurred(); // 检测本次函数执行，到底有没有异常
    if (throwable){
        // 补救措施，先把异常清除
        LOGD("native层：检测到 有异常...");
        // 清除异常
        env->ExceptionClear();
        // 重新获取 name1 属性
        f_id = env->GetStaticFieldID(clazz, "name1", "Ljava/lang/String;");
    }
}

// 异常方式二：【C++处理时异常】 往Java层抛出异常
extern "C"
JNIEXPORT void JNICALL
Java_com_derry_as_1jni_1project_MainActivity3_exception2(JNIEnv *env, jclass clazz) {
    // 假设现在想操作name999，没有name999就会在native层奔溃掉
    jfieldID f_id = env->GetStaticFieldID(clazz, "name999", "Ljava/lang/String;");

    // 共两种方式 之 方式二
    // 补救措施 ，name999 拿不到报错的话，想给 java 层抛一个异常
```

```
    jthrowable throwable = env->ExceptionOccurred();  // 检测本次函数执行，到底有没有异常
    // 想给 java 层抛一个异常
    if (throwable){
        // 清除异常
        env->ExceptionClear();
        // Throw 抛一个 java 的 Throwable 对象
        jclass no_such_clz = env->FindClass("java/lang/NoSuchFieldException");
        env->ThrowNew(no_such_clz,"NoSuchFieldException 实在是找不到 name999 啊，没办法，奔溃了!");
    }
}

// 异常方式三：【Java处理时异常】 Java的方法抛出了异常，然后native去清除
// 注意：Java的异常 native层无法捕获
extern "C"
JNIEXPORT void JNICALL
Java_com_derry_as_1jni_1project_MainActivity3_exception3(JNIEnv *env, jclass clazz) {
    jmethodID showMID = env->GetStaticMethodID(clazz, "show", "()V");
    env->CallStaticVoidMethod(clazz, showMID);

    // 按道理来说，上面的这句话：env->CallStaticVoidMethod(clazz, showMID);，就已经奔溃了，但是事实是否如此呢？
    LOGI("native层：>>>>>>>>>>>>>>>>>>>>>>>>>>>>>.1");
    LOGI("native层：>>>>>>>>>>>>>>>>>>>>>>>>>>>>>.2");
    LOGI("native层：>>>>>>>>>>>>>>>>>>>>>>>>>>>>>.3");
    LOGI("native层：>>>>>>>>>>>>>>>>>>>>>>>>>>>>>.4");
    LOGI("native层：>>>>>>>>>>>>>>>>>>>>>>>>>>>>>.5");
    LOGI("native层：>>>>>>>>>>>>>>>>>>>>>>>>>>>>>.6");

    // 证明不是马上就奔溃了，而是预料了时间，给我们处理呀
    if (env->ExceptionCheck()) {
        env->ExceptionDescribe();// 输出描述
        env->ExceptionClear();// 清除异常
    }
}

// TODO JNI异常处理总结：================================================
/*
 Native层出错了，没有办法再Java层去try的，处理的方式一般有两种：
  第一种：补救，例如：name999获取检测到发生异常了，就再去获取name0
  第二种：抛出，例如：name999获取检测到发生异常了，把此异常抛给Java层，让Java层去捕获异常

 Java层出错了，Native层可以去 监测到 然后清除Java的异常，具体业务逻辑自己去处理哦
 */
```

04-C++异常简介：

T1.cpp:

```
// TODO C++异常

#include <iostream>
#include <string>
using namespace std;

void exceptionMethod01()
{
    throw "我报废了";
}
```

```
// 更加简单的写法 自定义异常
class Student {
public:
    char * getInfo() {
        return "自定义";
    }
};

void exceptionMethod02() {
    Student s;
    throw s;
}

int main()
{
    try {
        exceptionMethod01();
    } catch (const char * &msg) {
        cout << "捕获到异常信息:" << msg << endl;
    }

    try {
        exceptionMethod02();
    } catch (Student &s) {
        cout << "捕获到异常信息: " << s.getInfo() << endl;
    }

    return 0;
}
```

05-手写JNIEnv

```
// TODO 手写JNIEnv

#include <iostream>
#include <string>
using namespace std;

// 如果是C语言
typedef const struct JNINativeInterface * JNIEnv; // 定义一个结构体指针的别名

// 模拟一个结构体
struct JNINativeInterface{
    // 结构体的函数指针
    char*(*NewStringUTF)(JNIEnv*, char*); // 函数指针的定义，实现在库中，我们这里还看不到
    // 省略 300多个 函数指针
    // ...
};

typedef char * jstring; // 简化了
typedef char * jobject; // 简化了

// 函数指针 对应的 函数实现 （这里只是简写，真正复杂的代码，就不去考虑了）
jstring NewStringUTF(JNIEnv* env, char* c_str){
    // 注意：在真正的源码中，这里需要写很多复杂代码来转换的（毕竟涉及到跨语言操作了C<-->Java），这里我们就简写了
    // c_str -> jstring
```

```
    return c_str;
}

// 模拟 我们的 JNI 函数，重点关注形参   JNIEnv * env
char* Java_j07_Demo02_getCStringPwd(JNIEnv * env, jobject jobject){
    // JNIEnv * 其实已经是一个二级指针了，所以 -> 调用的情况下必须是一级指针 *取值
    // C语言就是，就是二级指针，所以需要*取出一级指针才能使用->，   ->代表操作一级指针
    return (*env)->NewStringUTF(env, "9527");
}

// 下面是测试端 -- 其实就是 模拟 JNIEnv 内部执行的过程
int main() {
    // 构建 JNIEnv* 对象
    struct JNINativeInterface nativeInterface;

    // 给结构方法指针进行赋值(实现)
    nativeInterface.NewStringUTF = NewStringUTF;

    // 传给  Java_j07_Demo02_getCStringPwd 的参数是 JNIEnv*
    JNIEnv env = &nativeInterface;// 一级指针
    JNIEnv * jniEnv = &env;// 二级指针

    // 把 jniEnv 对象传给 Java_j07_Demo02_getCStringPwd  Java层
    char* jstring = Java_j07_Demo02_getCStringPwd(jniEnv, "com/derry/jni/MainActivity");

    // jstring 通过 JNIEnv桥梁 传给 java 层  （这个过程也省略了. . .   直接打印了）
    printf("Java层就拿到了 C++ 给我们的 jstring = %s", jstring);

    return 0;
}
```