JNI

JAVA本地编程接口，是一套规范，由JAVA定义，本地语言实现，

为什么要有JNI？

因为Android体系结构中的函数库层和Linux 内核层基本是使用C或者C++语言编写的，而应用框架层和应用层由JAVA编写，那么JNI就是用于JAVA与C或C++之间的相互调用，形成一个语言交互。

不直接使用JAVA来实现语言的交互是因为：JAVA语言效率略低，且无法直接驱动硬件。

交叉编译概念

在不同软硬件平台上模拟其它平台，并编译其它平台的可执行文件

如：windows平台编译出 .so文件

Google提供NDK编译工具进行交叉编译的支持。

NDK native development kit,是谷歌提供的交叉编译工具，帮助开发者快速开发C（或C++）的动态库，并能自动将so和java应用一起打包成apk。

ndk目录

指令：1、要配置环境变量

2、在cmd中找到指定的C语言源代码进行编译 命令 ndk-build

Nkd-build.cmd是从NDKr4版本开始引入的一个shell脚本。通过ndk-build可以方便的编译本地代码生成.so文件，如果想在控制台中使用ndk-build命令进行交叉编译，需要将ndk目录添加到环境变量中。

jni开发流程

1、 创建Android项目，在类中自定义一个方法让C语言去实现具体的功能

java声明本地方法 ：public native 返回值类型 方法名（）；

2、 类个给一个静态代码块： 用来加载指定的动态系统库

```
static{                    //通过系统的load方法加载系统库     与交叉编译后的 文件名保持一致
    System.loadLibrary("hello");
  }
```

3、 项目中创建一个与src平级的jni目录

4、 在jni目录中创建两个文件。（不支持中文，不能有空格）

- 一个后缀名是 .c的文件

c 文件 用于实现JAVA的本地语言方法 本地代码编写规则：

```
#include <string.h>     //已定义好的头文件
#include <jni.h>
jstring     //返回值类型    Java_包名_类名_要实现的方法名（）
Java_com_liu_jinrong_MainActivity_getStr( JNIEnv* env,jobject thiz )
{   return (*env)->NewStringUTF(env, "Hello");   }
```

```
#include <string.h>   //头文件
#include <jni.h>      //头文件
jstring  //返回值类型

//Java_包名_类名_要实现的方法名 ()
Java_com_liu_jinrong_MainActivity_getStr( JNIEnv* env, jobject thiz )
{
    return (*env)->NewStringUTF(env, "Hello,liujinrong");
}
```

结构体指针

要返回的内容

不支持中文

● 一个Android.mk的文件　　编译系统描述要编译的资源

mk 文件 用于指定jni 交叉编译的相关配置，如：交叉编译后的 .so文件名 、将要编译的目标源文件

配置信息内容：

```
LOCAL_PATH  :=  $(call  my-dir)
include  $(CLEAR_VARS)
LOCAL_MODULE :=  hello     //该名称与加载动态系统库名称一致  最终是用于加载系统库的
LOCAL_SRC_FILES:=  hello.c     //实现本地方法的C文件      名称与.c文件保存一致
include  $(BUILD_SHARED_LIBRARY)
```

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE     := hello       //该名称与加载动态系统库名称一致 最终是用于加载系统库的
LOCAL_SRC_FILES := hello.c     //实现本地方法的C文件   名称与.c文件保存一致

include $(BUILD_SHARED_LIBRARY)
```

```
# 获取jni目录的绝对路径
LOCAL_PATH := $(call my-dir)
# 清除上一次编译时的配置信息
include $(CLEAR_VARS)
# 指定编译生成的目标文件的名字
# 编译之后系统会自动加上lib前缀 生成的文件名字是libhello.so
LOCAL_MODULE     := hello
#指定要变异的c的代码叫什么名字 如果有多个.c文件共同编译 不同文件名之间用空格隔开
LOCAL_SRC_FILES := hello.c
#指定编译生成一个动态链接库 .so
include $(BUILD_SHARED_LIBRARY)
```

● 如果模拟器是x86平台的还需要创建一个Application.mk文件　　配置 arm以外其他平台的编译方式

配置信息内容：APP_ABI := all

5、 使用命令进行交叉编译 ndk-build

6、 运行看模拟器的执行结果。

jni简便开发流程

1、创建本地方法及C回调java中的方法 public native 返回值类型 方法名（）；

2、加载.SO的动态库。static { System.loadLibrary(); }

3、Android Tools------>Add Android Native Support------>给个名称（指定要编译的c的代码叫什么名字）

4、打开jni文件夹将.cpp文件改为.c ，Android.mk文件中的.cpp改.c 然后在jni文件夹中创建一个Application.mk文件 定义其内容 APP_ABI:=all

5、打开SRC文件夹-------->在当前目录进入CMD-------->输入 javah 包名.类名（有本地方法的类）- - 回车后自动生成一个.h文件

6、添加头文件自动编写方法：右键--->Properties----->C/C++ Generl------>Paths and Sysbols----->Add------->选中Add to all languages添加includen的所在目录 E:\androidndk64\android-ndk-r9\platforms\android-18\arch-arm\usr\include

7、刷新后将.H文件jni文件夹，复制.h文件的名称 粘贴到 .c文件的 #include <.h文件名> 头文件 并将.h文件打开复制c的方法到.c文件中

JNIEXPORT ~~void JNICALL(不同的本地方法返回值和类型会不同)~~ JAVA_对应的包名_类名_方法名

(JNIEnv *, jobject);

8、完善.c里的方法： (JNIEnv *指针名, jobject thiz){C的具体的方法逻辑}

9、进入打开jni文件-------->在当前目录进入CMD-------->输入 ndk-build 进行交叉编译

10、运行项目

C里的具体方法逻辑：

1、把java数据传递给C并返回数据给JAVA

java写：

```java
package com.liurong;
//由底层C语言实现本地无参方法，返回数据
import android.app.Activity;
import android.os.Bundle;
import android.widget.Toast;
public class MainActivity extends Activity {
    static {
        System.loadLibrary("hello");
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 接收返回值
        String str = getStr();
        Toast.makeText(this, str, 1).show();
    }
    // 无参有返回值类型的本地方法
    public native String getStr();
}
```

```java
package com.liurong;
import android.os.Bundle;
//由底层C语言通过实现本地方法，反射回调JAVA方法
public class MainActivity extends Activity {
    static{
        System.loadLibrary("hhelo");    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        calls();
    }//本地方法
    public native void calls();
    //C要回调的JAVA方法
    public void scall(){
        Toast.makeText(getApplicationContext(), "回调成功显示一下吧！", 1).show();
    }
```

```java
package com.liurong;
//由底层C语言实现本地有参方法，返回数据
import android.app.Activity;
public class MainActivity extends Activity {
    static{
        System.loadLibrary("addint");
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        int sum = addint(1,2);
        Toast.makeText(this, "计算结果是："+sum, 1).show();
    }
    public native int addint(int i,int j);
}
```

```java
package com.liurong;
import android.app.Activity;
public class MainActivity extends Activity {
//由底层C语言回调有参JAVA方法，本地将回调参数设置到进度条，并实时更新进度条显示数据
    static{
        System.loadLibrary("jianc");
    }            private ProgressBar pb;    //进度要条
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        pb = (ProgressBar) findViewById(R.id.pb_g);
        new Thread() {
            public void run() {
                getp();  //子线程实时回去回调数据
            };
        }.start();
    }
    public native void getp();
    public void intpd(int i){
        pb.setProgress(i);
    }
```

c写：

```c
#include <com_liurong_MainActivity.h>
JNIEXPORT jstring JNICALL Java_com_liurong_MainActivity_getStr(JNIEnv *env,
        jobject thiz) {
    return (*env)->NewStringUTF(env, "Hello,liujinrong");
}

```

```c
#include <com_liurong_MainActivity.h>
JNIEXPORT jint JNICALL Java_com_liurong_MainActivity_addint(JNIEnv* env,
        jobject thiz, jint i, jint j) {
    return i+j;
}
```

```c
#include <com_liurong_MainActivity.h>
#include<stdlib.h>
//产生随机数
int getint(){
    return rand() %101;     //产生0-100的随机数
}
JNIEXPORT void JNICALL Java_com_liurong_MainActivity_getp
  (JNIEnv *env, jobject thiz){
    int gin;
    //通过反射调用方法将数据返回给方法
    jclass clazz = (*env)->GetObjectClass(env,thiz);
    jmethodID md = (*env)->GetMethodID(env,clazz,"intpd","(I)V");
    while(1){
        //循环获取数据
        sleep(2);
        gin = getint();
        (*env)->CallVoidMethod(env,thiz,md,gin);
    }
}
```

```c
#include <com_liurong_MainActivity.h>
JNIEXPORT void JNICALL Java_com_liurong_MainActivity_calls
  (JNIEnv *env, jobject thiz){
    //获取类，通过反射
    jclass clazz = (*env)->GetObjectClass(env,thiz);
    //获取要回调的方法
    /*参一：指针  env
     *参二：获得的类  clazz
     *参三：JAVA中要回调的方法名
     *参四：回调方法的签名
    */
    jmethodID jm = (*env)->GetMethodID(env,clazz,"scall","()V");
    /**
     * 参一：指针
     * 参二:
     * 参三：获取的方法
     * 参四：回调方法的实参值
     */
    (*env)->CallVoidMethod(env,thiz,jm,...);
}
```

2、C回调java中的方法经典事例

```
package com.liurong;
import java.util.ArrayList;
public class MainActivity extends Activity {
    private TextView xin, yp;
    private ProgressBar pd;
    private int ht;
    private Handler handler;
    static {
        System.loadLibrary("myxin");  //加载动态库资源
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        xin = (TextView) findViewById(R.id.xin);
        yp = (TextView) findViewById(R.id.yp);
        pd = (ProgressBar) findViewById(R.id.pd);    //进度条
        new Thread() {
            public void run() {
                jk();  //实时获取C回调的数据
            };
        }.start();
    }
    public native void jk();
    public void getxin(int i) {
        final MediaPlayer md = MediaPlayer.create(this,R.raw.jing_bao);
        pd.setProgress(i);    //获取回调数据给进度条组件
        ht = i;     //获取回调数据赋值给int  ht
        //主线程更新UI组件
        MainActivity.this.runOnUiThread(new Thread() {
            @Override
            public void run() {
                if (ht < 120 || ht > 61) {
                    yp.setText("");
                    xin.setText("当前心率正常：" + ht);
                    md.stop();
                }
                if (ht > 120 || ht < 60) {
                    xinset();
                    md.start();
                    xin.setText("当前心率危险：" + ht);
                    handler = new Handler() {
                        @Override
                        public void handleMessage(Message msg) {
                            String string = msg.obj.toString();
                            yp.setText(string);
                        }
                    };
                }
            }
        });
    }
    //上传数据到服务器并返回数据给客户端
    public void xinset() {
        new Thread() {
            public void run() {
                try {
                    String path = "http://192.168.154.36:8080/MyPhysique/servlet/MypostGobac
                    HttpClient hc = new DefaultHttpClient();
                    HttpPost hp = new HttpPost(path);
                    BasicNameValuePair pair = new BasicNameValuePair("name",
                            "张小人");
```

```
79          BasicNameValuePair pair1 = new BasicNameValuePair("sex",
80                  "男");
81          BasicNameValuePair pair2 = new BasicNameValuePair("place",
82                  "XX路XX号XX楼110房");
83          BasicNameValuePair pair3 = new BasicNameValuePair("state",
84                  ht + "");
85          List<BasicNameValuePair> list = new ArrayList<BasicNameValuePair>();
86          list.add(pair);
87          list.add(pair1);
88          list.add(pair2);
89          list.add(pair3);
90          UrlEncodedFormEntity formEntity = new UrlEncodedFormEntity(
91                  list, "UTF-8");
92          hp.setEntity(formEntity);
93          HttpResponse response = hc.execute(hp);
94          if (response.getStatusLine().getStatusCode() == 200) {
95              HttpEntity entity = response.getEntity();
96              String content = EntityUtils.toString(entity);
97              Message message = new Message();
98              message.obj = content;
99              handler.sendMessage(message);
```

C中的回调处理

```
1#include <com_liurong_MainActivity.h>
2#include <stdlib.h>
3
4int geti() {
5    return rand() % 122;
6}
7JNIEXPORT void JNICALL Java_com_liurong_MainActivity_jk(JNIEnv *env,
8        jobject thiz) {
9    jclass clazz = (*env)->GetObjectClass(env, thiz);
10   jmethodID md = (*env)->GetMethodID(env, clazz, "getxin", "(I)V");
11   int i;
12   while (1) {
13
14       i = geti();
15       (*env)->CallVoidMethod(env, thiz, md, i);
16       sleep(5);
17   }
18}
```

回调方法的 方法参数及返回值类型签名 的获取方法：

打开要回调的类.class文件夹下运行CMD 输入 javap -s 类名

进行反编译获取so库

1、将第三方应用apk解压缩，通过反编译助手将解压缩的.dex文件转为jar包，获取到所有的JNI本地方法

2、将解压缩出来的lib文件目录下的文件夹（该文件夹存放着编译好的.SO库）拷贝到自己项目的libs目录里，并将这个文件中的.SO文件名（去掉文件名的头lib和尾的.SO）复制到自己类的静态代码块中做为动态库加载，如果没有动态库加载那么本地方法无法使用

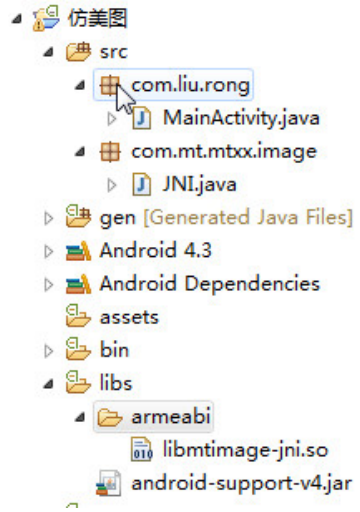
2、将所有JNI本地方法包含包名一起拷贝到自己的工程类中，并建立一个与JNI包名完全相同的一个包来保存JNI类

3、定义自己的方法，调用第三方的JNI方法即可实现第三方应用的功能。

经典案例：仿美图

```
public class MainActivity extends Activity {
    static {
        System.loadLibrary("mtimage-jni");
    }
    private Bitmap bt;
    private ImageView img;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        img = (ImageView) findViewById(R.id.img_IV);
        //通过位图工厂获取图片
        bt = BitmapFactory.decodeFile("data/sas.jpg");
        //将图片设置到UI控件
        img.setImageBitmap(bt);
    }
    public void KO1(View v) {
                //获取图片的宽及高
                int width = bt.getWidth();
                int height = bt.getHeight();


                int[] btt = new int[width * height];
                // 将图片转为数据的形式
                bt.getPixels(btt, 0, width, 0, 0, width, height);
                // 使用美图已定义好的JNI
                JNI J = new JNI();
                J.StyleLomoC(btt, width, height);
                //经美图底层处理完后重新设置到UI组件
                Bitmap nBitmap = Bitmap.createBitmap(btt, width, height,
                        bt.getConfig());
                img.setImageBitmap(nBitmap);
```



卸载应用打开浏览器案例

应用是否被卸载
运用fork函数获取应用子进程
使用execlp函数开启系统浏览器

c语言基本语法

include的作用: 包含头文件。头文件是c语言的一些核心函数库

#include<stdlib.h> //include(包含) 相当于java的 import

#include<stdio.h> //.h C的头文件 stdlib.h标准函数库 stdio.h标准输入输出流

main函数编译规则：int main(){ return 0} 前面的**int**可以不写 **main**程序开始的入口

system函数的作用: 用来调用系统命令 或可执行程序 system("pause") //system 调用系统命令

数据类型

int long char float double C 没有 boolean byte

长度（字节）：

short 2

int 4

long 4

char 1

float 4

double 8

使用sizeof(int)获取数据类型的长度 printf("long = %d \n",sizeof(long)); 获取数据类型长度

signed unsigned 有符号无符号修饰符 只能修饰整形变量

signed 有符号 最高位符号位 可以表示负数

unsigned 无符号 最高位仍然是数值位 不可以表示负数

函数

输出函数

printf("要输出的内容+占位符",.....) printf("c=%c ,s = %hd ,i = %d, f=%f \n",c,s,i,f);//变量作为可变参数进行传递

常用占位符

%d - int

%ld – long int

%lld - long long

%hd – 短整型

%c - char

%f - float

%lf – double

%u – 无符号数

%x – 十六进制输出 int或者long int或者short int

%o - 八进制输出

%s – 字符串

输入函数 **Scanf**

Int len;

Scanf(“%d”,&len); &取地址符号 内存指针

内存地址

内存地址是分配，使用及管理内存空间的数字编号，通常用16进制表示

指针：指向内存某个地址的一个标识

指针的定义格式：数据类型* 指针变量名; int* p; cha* p;

为指针赋值：int* p = & 普通变量名;

获取指针地址：　　指针变量名　　p

使用指针取指针指向的内存 的数值：　*指针变量名　　*p

指针的用途

1、 指向内存地

2、 直接访问硬件

3、 快速传递数据(指针表示地址)

4、 操作数组

5、定义及操作字符串

指针常见的错误

指针错误1：指针必须分配内存空间，方可使用！！！

指针错误2：错误使用不同数据类型的指针。

a，数组的存储规则　　数组是一段连续的内存空间

b，说明数组与指针的关系　　指针指向数组的第1个元素的内存地址

多级指针

定义格式：数据类型** 多级指针名;　　取值 ： **多级指针名

静态内存和动态内存的概念

静态内存：自动开辟内存空间，函数执行完成自动释放，位于栈空间

动态内存：由程序员手动开辟空间，函数执行完不会自动释放，位于堆空间

开辟动态内存空间： malloc(内存空间大小);　　开辟的空间地址必须给指针，而不能直接赋值给变量。

```
int* i = malloc(sizeof(int));//开辟内存空间
```

重新开辟空间：realloc(指针,新长度);

释放内存：free(指针);

静态内存和动态内存

静态内存是系统是程序编译执行后系统自动分配,由系统自动释放,静态内存是栈分配的.

动态内存是开发者手动分配的,是堆分配的.

栈内存
* 系统自动分配
* 系统自动销毁
* 连续的内存区域
* 向低地址扩展
* 大小固定
* 栈上分配的内存称为静态内存

堆内存
* 程序员手动分配
* java：new　* c：malloc
* 空间不连续
* 大小取决于系统的虚拟内存
* C：程序员手动回收free
* java：自动回收
* 堆上分配的内存称为动态内存

JNI： Java本地编程接口。
Java定义规范。
本地语言进行具体实现。

内存地址：内存空间的编号。

指针：内存地址。

指针变量：存放内存地址（指针）的变量。长度为4 (64位系统中长度为8)

指针错误：指针变量必须初始化。指针变量数据类型和变量的数据类型冲突。

指针操作数组：指针指向的是数组的第一个元素地址。int* nums = 数组地址
*(nums+i)//代替nums[角标]；

多级指针：指向 指针变量 的指针。int*** p; 取值：***p;

静态、动态内存：

静态内存：自动分配，随着函数执行完成自动释放。栈中

动态内存：程序员手动分配，手动释放。堆中

数据结构

结构体声明

```
struct per{    struct 结构体名{变量1; 变量2; .....};
   int age;
   int id;
   char* name;
};
```

```c
//结构体
struct per{
    int age;
    int id;
    char* name;
};
main(){
    //结构体的赋值及使用
    struct per p = {26,1233,"xiaoming"};
    printf("age=%d \n",p.age);
    struct per p2;
    p2.id = 3336;
    printf("id = %d \n",p2.id);
    struct per* p3 = malloc(sizeof(struct per));
    (*p3).age = 39;
    printf("age = %d \n",(*p3).age);
}
```