在原生代码中使用POSIX创建线程，需要引入头文件pthread.h

1、初始化java虚拟机接口指针

```
// Java 虚拟机接口指针
static JavaVM* gVm = NULL;

// 对象的全局引用
static jobject gObj = NULL;
```

2、在JNI_OnLoad方法中保存gVm变量

```
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
    //缓存java虚拟机接口指针
    gVm = vm;
    return JNI_VERSION_1_4;
}
```

3、创建线程

通过 pthread_create 函数创建 POSIX 线程。

```
int pthread_create(pthread_t* thread,
    pthread_attr_t const* attr,
    void* (*start_routine)(void*),
    void* arg);
```

该函数有如下参数：

- 指向 thread_t 类型变量的指针，函数用该指针返回新线程的句柄。
- 指向 pthread_attr_t 结构的指针形式存在的新线程属性，可以通过该属性指定新线程的栈基址、栈大小、守护大小、调度策略和调度优先级等。本章后面的内容将介绍这些属性中的一部分，如果使用默认值，取值可能为 NULL。
- 指向线程启动程序的函数指针，启动程序函数签名格式如下：

```
void* start_rountine (void* args)
```

启动程序将线程参数看成 void 指针，返回 void 指针类型结果。

当线程以空指针的形式执行时，参数都需要被传递给启动程序，如果不需要传递参数，它可以为 NULL。

成功时，pthread_create 函数返回 0，否则返回一个错误代码。

4、附加线程到java虚拟机

```
JNIEnv* env = NULL;
    //将当前线程附加到java虚拟机上，并且获得JNIEnv接口指针
    if (0 == gVm->AttachCurrentThread(&env, NULL)) {

        NativeWorkerArgs* nativeWorkerArgs = (NativeWorkerArgs*) args;

        if (0 != pthread_mutex_lock(&mutex)) {
            throwException(env, "unable to lock pthread_mutex_lock");
            goto exit;
        }

        if (0 != pthread_mutex_unlock(&mutex)) {
            throwException(env, "unable to unlock pthread_mutex_unlock");
            goto exit;
        }

        //释放元素的worker线程参数
        delete nativeWorkerArgs;

        //从java虚拟机中分离出当前线程
        gVm->DetachCurrentThread();
    }
```

5、完整代码

```
#include "com_example_zzl_jni_NativeThread.h"
#include <pthread.h>

#include <stdio.h>
#include <unistd.h>

//原生worker线程参数
struct NativeWorkerArgs {
    jint id;
    jint iterations;
```

```
};
//java虚拟机接口指针
static JavaVM* gVm = NULL;
static jobject gObj = NULL;

//线程同步 互斥锁
static pthread_mutex_t mutex;

//onCallbackMsg方法id被缓存
static jmethodID callbackMsgId = NULL;

void throwException(JNIEnv* env, char* msg);
/*
 * Class:     com_example_zzl_jni_NativeThread
 * Method:    nativeInit
 * Signature: ()V
 */JNIEXPORT void JNICALL Java_com_example_zzl_jni_NativeThread_nativeInit(
        JNIEnv * env, jobject obj) {
    if (NULL == gObj) {
        //为对象创建一个新的全局引用
        gObj = env->NewGlobalRef(obj);
        if (NULL == gObj) {
            goto exit;
        }
    }

    //初始化互斥锁
    if (0 != pthread_mutex_init(&mutex, NULL)) {
        throwException(env, "unable to init pthread_mutex_init");
        goto exit;
    }

    //没有被缓存
    if (NULL == callbackMsgId) {
        //从对象中获取类
```

```
        jclass clazz = env->GetObjectClass(obj);
        callbackMsgId = env->GetMethodID(clazz, "onCallbackMsg","
(Ljava/lang/String;)V");
        //如果没有找到
        if (NULL == callbackMsgId) {
            throwException(env, "unable to find method onCallbackMsg");
        }
    }

    exit: return;
}

JNIEXPORT void JNICALL Java_com_example_zzl_jni_NativeThread_nativeFree(
        JNIEnv * env, jobject obj) {
    if (NULL != gObj) {
        //为对象创建一个新的全局引用
        env->DeleteGlobalRef(gObj);
        gObj = NULL;
    }

    //销毁互斥锁
    if (0 != pthread_mutex_destroy(&mutex)) {
        throwException(env, "unable to destroy pthread_mutex_destroy");
    }
}

/*
 * Signature: (II)V
 */JNIEXPORT void JNICALL Java_com_example_zzl_jni_NativeThread_nativeWorker(
        JNIEnv * env, jobject obj, jint id, jint iterations) {
    jint i = 0;

    for (; i < iterations; i++) {
        char msg[50];
        sprintf(msg, "工作线程id:%d,执行第 %d 次", id, i);
```

```
        jstring msgStr = env->NewStringUTF(msg);
        env->CallVoidMethod(obj, callbackMsgId, msgStr);
        if (NULL != env->ExceptionOccurred()) {
            break;
        }
        sleep(1);
    }
}


/------------------------------------------------------------下面是在native中创建线程----
-------------------------------------------------------------------------/


/*
 * POSIX线程
 * 它不是java平台一部分，虚拟机不能识别POSIX，因此要先将它附加到虚拟机上。
 */

jint JNI_OnLoad(JavaVM* vm, void* reserved) {
    //缓存java虚拟机接口指针
    gVm = vm;
    return JNI_VERSION_1_4;
}

/**
 * native附加线程
 */
static void* nativeWorkerThread(void* args) {
    JNIEnv* env = NULL;
    //将当前线程附加到java虚拟机上，并且获得JNIEnv接口指针
    if (0 == gVm->AttachCurrentThread(&env, NULL)) {

        NativeWorkerArgs* nativeWorkerArgs = (NativeWorkerArgs*) args;
```

```
        if (0 != pthread_mutex_lock(&mutex)) {
            throwException(env, "unable to lock pthread_mutex_lock");
            goto exit;
        }

        //在线程上下文中运行原生worker
        jint i = 0;
        jint iterations = nativeWorkerArgs->iterations;
        jint id = nativeWorkerArgs->id;

        for (; i < iterations; i++) {
            char msg[50];
            sprintf(msg, "ndk工作线程id:%d,执行第 %d 次", id, i);
            jstring msgStr = env->NewStringUTF(msg);
            env->CallVoidMethod(gObj, callbackMsgId, msgStr);
            if (NULL != env->ExceptionOccurred()) {
                break;
            }
            sleep(1);
        }

        if (0 != pthread_mutex_unlock(&mutex)) {
            throwException(env, "unable to unlock pthread_mutex_unlock");
            goto exit;
        }

        //释放元素的worker线程参数
        delete nativeWorkerArgs;

        //从java虚拟机中分离出当前线程
        gVm->DetachCurrentThread();
    }
    exit: return (void*) 1;
}
```

```
/*
 * Class:     com_example_zzl_jni_NativeThread
 * Method:    posixThreads
 * Signature: (II)V
 */JNIEXPORT void JNICALL Java_com_example_zzl_jni_NativeThread_posixThreads(
        JNIEnv * env, jobject obj, jint threads, jint iterations) {
    jint i = 0;

    for (; i < threads; i++) {
        //封装参数设置
        NativeWorkerArgs* workerArgs = new NativeWorkerArgs();
        workerArgs->id = i;
        workerArgs->iterations = iterations;

        //线程句柄
        pthread_t thread;
        int result = -1;
        //创建新的线程
        result = pthread_create(&thread, NULL, nativeWorkerThread,
                (void*) workerArgs);
//       pthread_join() 可以使一个函数等待线程的终止，可挂起UI线程
        //线程创建失败
        if (result != 0) {
            jclass exception = env->FindClass("java/lang/RuntimeException");
            env->ThrowNew(exception, "unable to create thread");
        }
    }

}

void throwException(JNIEnv* env, char* msg) {
    jclass exception = env->FindClass("java/lang/RuntimeException");
    env->ThrowNew(exception, msg);
}
```

```
public class NativeThread {

    private ThreadCallback callback;
    public NativeThread(ThreadCallback callback){
        this.callback = callback;
    }

    /**
     * 初始化原生代码
     */
    public native void nativeInit();
    /**
     * 是否资源
     */
    public native void nativeFree();

    /**
     *
     * @param id 工作线程id
     * @param iterations 工作线程循环次数
     */
    public native void nativeWorker(int id,int iterations);

    /**
     * native回调
     * @param msg
     * @param callback
     */
    public void onCallbackMsg(final String msg){
        callback.onCallbackMsg(msg);
    }

    /**
     * 在native层使用POSIX线程
```

```
     * @param id
     * @param iterations
     */
    public native void posixThreads(int id,int iterations);



    static{
        System.loadLibrary("nativethread");
    }

}


package com.example.zzl.jni;

import android.R.integer;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

import com.example.zzzadf.R;

public class ThreadAcitivity extends Activity implements ThreadCallback{

    private EditText et_thread_num;
    private EditText et_exec_num;
    private Button btn_android_thread;
    private Button btn_ndk_thread;
    private TextView tv_log;
    private NativeThread nativeThread = new NativeThread(this);
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.jni_native_thread_main);
    et_thread_num = (EditText) findViewById(R.id.et_thread_num);
    et_exec_num = (EditText) findViewById(R.id.et_exec_num);
    btn_android_thread = (Button) findViewById(R.id.btn_android_thread);
    btn_ndk_thread = (Button) findViewById(R.id.btn_ndk_thread);
    tv_log = (TextView) findViewById(R.id.tv_log);

    nativeThread.nativeInit();

    btn_android_thread.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            int threadnums = getNumber(et_thread_num, 0);
            int iterations = getNumber(et_exec_num, 0);
            if (threadnums>0&& iterations>0) {
                tv_log.setText("");
                startThreads(threadnums,iterations);
            }
        }
    });
    btn_ndk_thread.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            int threadnums = getNumber(et_thread_num, 0);
            int iterations = getNumber(et_exec_num, 0);
            if (threadnums>0&& iterations>0) {
                tv_log.setText("");
                nativeThread.posixThreads(threadnums,iterations);
            }
        }
    });
```

```
}



private void startThreads(int threadnums, int iterations) {
    javaThread(threadnums,iterations);
}

/**
 * 原生消息回调
 * @param msg
 */
@Override
public void onCallbackMsg(final String msg){
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            tv_log.append(msg);
            tv_log.append("\n");
        }
    });
}

private void javaThread(int threads,final int iterations){
    for (int i = 0; i < threads; i++) {
        final int id = i;
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("threadid: "+Thread.currentThread().getId());
                nativeThread.nativeWorker(id, iterations);
            }
        });
```

```
            thread.start();
        }
    }

    @Override
    protected void onDestroy() {
        nativeThread.nativeFree();
        super.onDestroy();
    }

    private int getNumber(EditText et,int defaultVal){
        int value=0;
        try {
            value = Integer.parseInt(et.getText().toString().trim());
        } catch (Exception e) {
            value = defaultVal;
        }
        return value;
    }

}



public interface ThreadCallback {
    public void onCallbackMsg(String msg);
}




#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include <pthread.h>

//消费者与生产者应用

//消费者数量
#define CONSUMER_NUM 2
//生产者数量
#define PRODUCER_NUM 1

pthread_t pids[CONSUMER_NUM+PRODUCER_NUM];

//产品队列
int ready = 0;

//互斥锁
pthread_mutex_t mutex;
//条件变量
pthread_cond_t has_product;

//生产
void* producer(void* arg){
    int no = (int)arg;
    //条件变量
    for(;;){
        pthread_mutex_lock(&mutex);
        //往队列中添加产品
        ready++;
        printf("producer %d, produce product\n",no);
        //fflush(NULL);
        //通知消费者，有新的产品可以消费了
        //会阻塞输出
        pthread_cond_signal(&has_product);
        printf("producer %d, singal\n",no);
        pthread_mutex_unlock(&mutex);
        sleep(1);
```

```
    }
}

//消费者
void* consumer(void* arg){
    int num = (int)arg;
    for(;;){
        pthread_mutex_lock(&mutex);
        //while?
        //superious wake  '惊群效应'
        while(ready==0){
            //没有产品，继续等待
            //1.阻塞等待has_product被唤醒
            //2.释放互斥锁，pthread_mutex_unlock
            //3.被唤醒时，解除阻塞，重新申请获得互斥锁pthread_mutex_lock
            printf("%d consumer wait\n",num);
            pthread_cond_wait(&has_product,&mutex);
        }
        //有产品，消费产品
        ready--;
        printf("%d consume product\n",num);
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}


void main(){
    //初始化互斥锁和条件变量
     pthread_mutex_init(&mutex,NULL);
      pthread_cond_init(&has_product,NULL);
    printf("init\n");

    int i;
    for(i=0; i<PRODUCER_NUM;i++){
```

```
        //生产者线程
        printf("%d\n",i);
        pthread_create(&pids[i],NULL,producer,(void*)i);
    }

    for(i=0; i<CONSUMER_NUM;i++){
        //消费者线程
        pthread_create(&pids[PRODUCER_NUM+i],NULL,consumer,(void*)i);
    }

    //等待
    sleep(10);
    for(i=0; i<PRODUCER_NUM+CONSUMER_NUM;i++){
        pthread_join(pids[i],NULL);
    }

    //销毁互斥锁和条件变量
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&has_product);

}



#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int i = 0;
//互斥锁
pthread_mutex_t mutex;

void* thr_fun(void* arg){
    //加锁
```

```
    pthread_mutex_lock(&mutex);
    char* no = (char*)arg;
    for(;i < 5; i++){
        printf("%s thread, i:%d\n",no,i);
        sleep(1);
    }
    i=0;
    //解锁
    pthread_mutex_unlock(&mutex);
}

void main(){
    pthread_t tid1, tid2;
    //初始化互斥锁
    pthread_mutex_init(&mutex,NULL);

    pthread_create(&tid1,NULL,thr_fun,"No1");
    pthread_create(&tid2,NULL,thr_fun,"No2");

    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);

    //销毁互斥锁
    pthread_mutex_destroy(&mutex);
}
```