MICROOH 麦可网

Android-从程序员到架构师之路

出品人：Sundy
讲师：高焕堂（台湾）

http://www.microoh.com

C05_a

JNI：多个Java线程进入本地函数(a)

By 高煥堂

内容

1. 介绍JNI线程模式
2. 从Session概念认识JNIEnv对象
3. 细说JNIEnv对象
4. 本地函数的线程安全

1、认识JNI线程模式

Android线程的特性

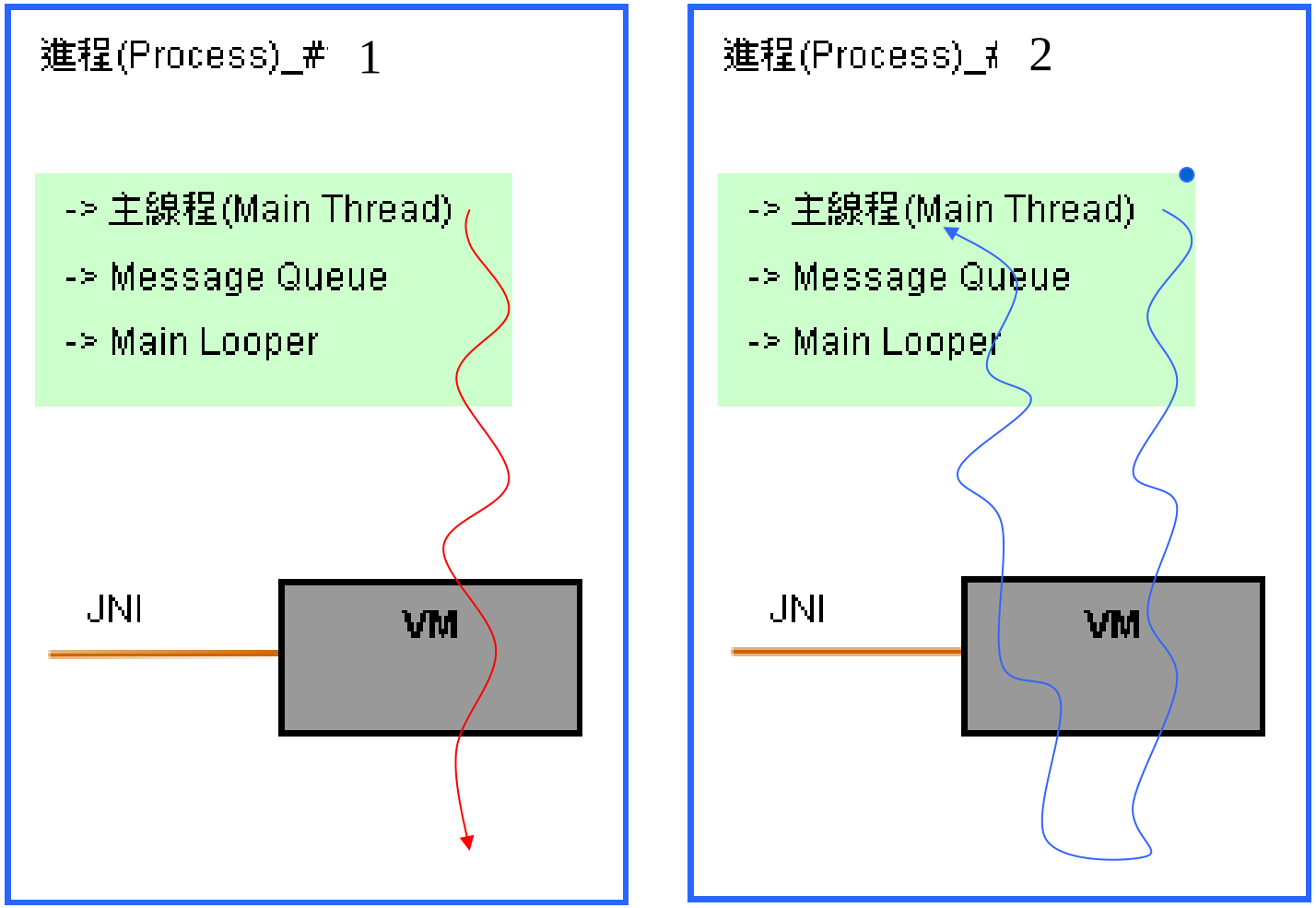
- 线程(Thread)又称为「执行绪」。
- 于默认(Default)下，一个App的各类别(如Activity、BroadcastReceiver等)都在同一个进程(Process)里执行，而且由该进程的主线程负责执行。
- 如果有特别指示，也可以让特定类在不同的进程里执行。

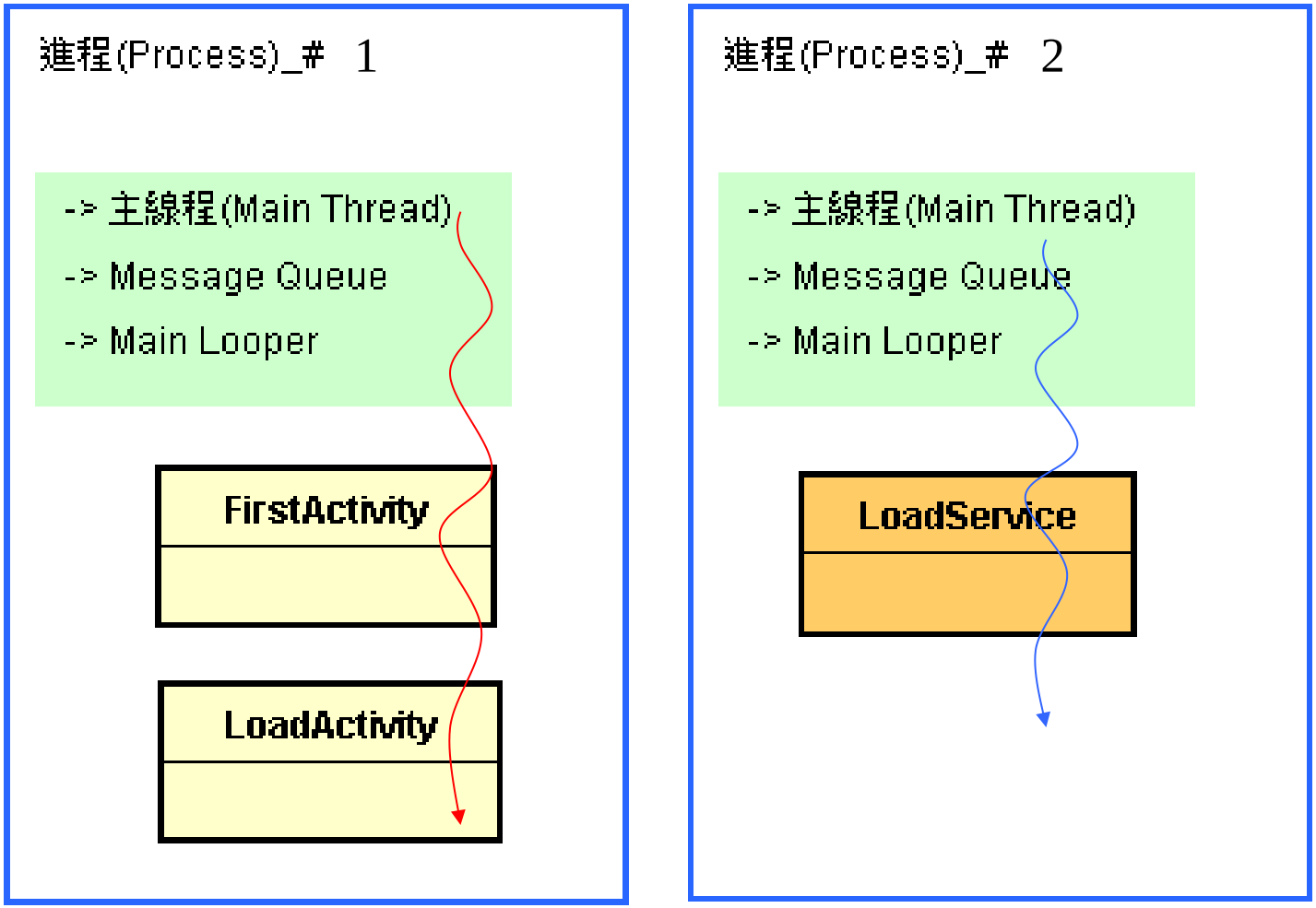
- 例如由一个Activity启动一个Service，在默认情形下，两者都在同一个进程里执行。
- 主线程除了要处理Activity类别的UI事件，又要处理Service幕后服务工作，通常会忙不过来。
- 该如何化解这种困境呢?

- 主线程可以诞生多个子线程来分担其工作，尤其是比较冗长费时的幕后服务工作，例如播放动画的背景音乐、或从网络下载映片等。
- 于是，主线程就能专心于处理UI画面的事件了。

线程往返Java与C/C++

- 由于每一个进程里都有一个主线程。
- 每一个进程里，都可能有Java程序码，也有C/C++本地程序码。
- Java层的主线程经常从Java层进入JNI层的C函数里执行；此外，当反向调用Java函数时，又返回进入Java函数里执行。


進程(Process)_# 1
-> 主線程(Main Thread)
-> Message Queue
-> Main Looper
JNI
VM
進程(Process)_# 2
-> 主線程(Main Thread)
-> Message Queue
-> Main Looper
JNI
VM


進程(Process)_# 1
-> 主線程(Main Thread)
-> Message Queue
-> Main Looper
FirstActivity
LoadActivity
進程(Process)_# 2
-> 主線程(Main Thread)
-> Message Queue
-> Main Looper
LoadService

- 无论是主线程，或是子线程，都可以从Java层进入C/C++层去执行，也能从C/C++层进入Java层。
- 在本节里，就来说明跨越JNI的线程模式，以及如何化解线程的冲突问题。

VM对象与JavaVM指针

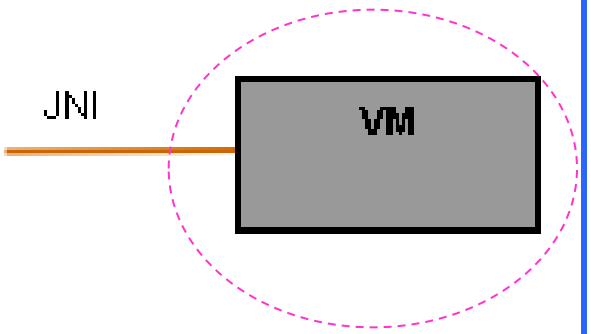
- 在进程里，有一个虚拟机(Virtual Machine，简称VM)的对象，可执行Java代码，也引导JNI本地程序的执行，实现Java与C/C++之间的沟通。

進程(Process)_#1

-> 主線程(Main Thread)

-> Message Queue

-> Main Looper

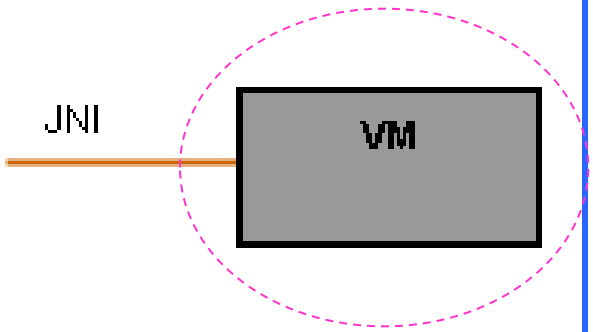


進程(Process)_#1

-> 主線程(Main Thread)

-> Message Queue

-> Main Looper



- 当VM执行到System.loadLibrary()函数去加载C模块时会时，就会立即先调用JNI_OnLoad()函数。
- VM调用JNI_OnLoad()时，会将VM的指标(Pointer)传递给它，其参数如下：

```
/* com.misoo.counter.CounterNative.cpp */
// ……
JavaVM *jvm;
// ……

jint JNI_OnLoad( JavaVM* vm, void* reserved){
    jvm = vm;
    return JNI_VERSION_1_4;
}
```

- 指令：jvm = vm;

  将传来的VM指针储存于这本地模块(*.so)的公用变量jvm里。让本地函数随时能使用jvm来与VM交互。

- 例如，当你创建一个本地C层的新线程时，可以使用指令：

```
jvm->AttachCurrentThread(&env, NULL);
```

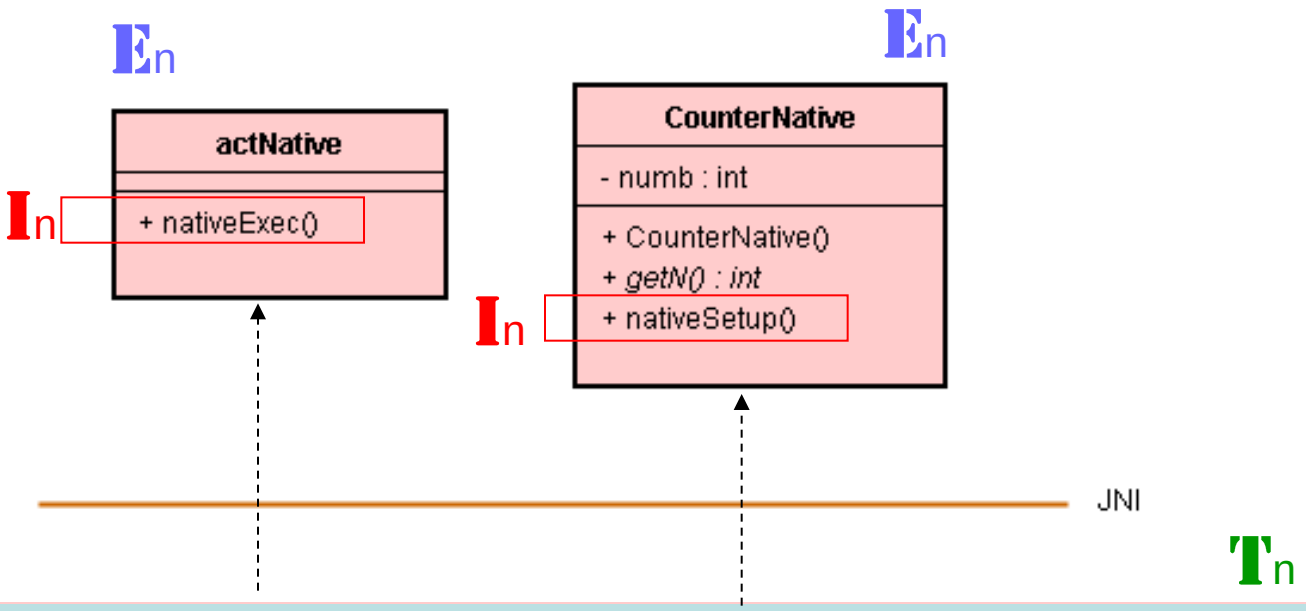
- 就向VM登记，要求VM诞生JNIEnv对象，并将其指针值存入env里。有了env值，就能执行指令：

```
env->CallStaticVoidMethod(mClass, mid, sum);
```

- 其调用Java层的函数了。

为什么需要JNIEnv对象呢?

- 本地C函数的第1个参数就是JNIEnv对象的指针，例如：



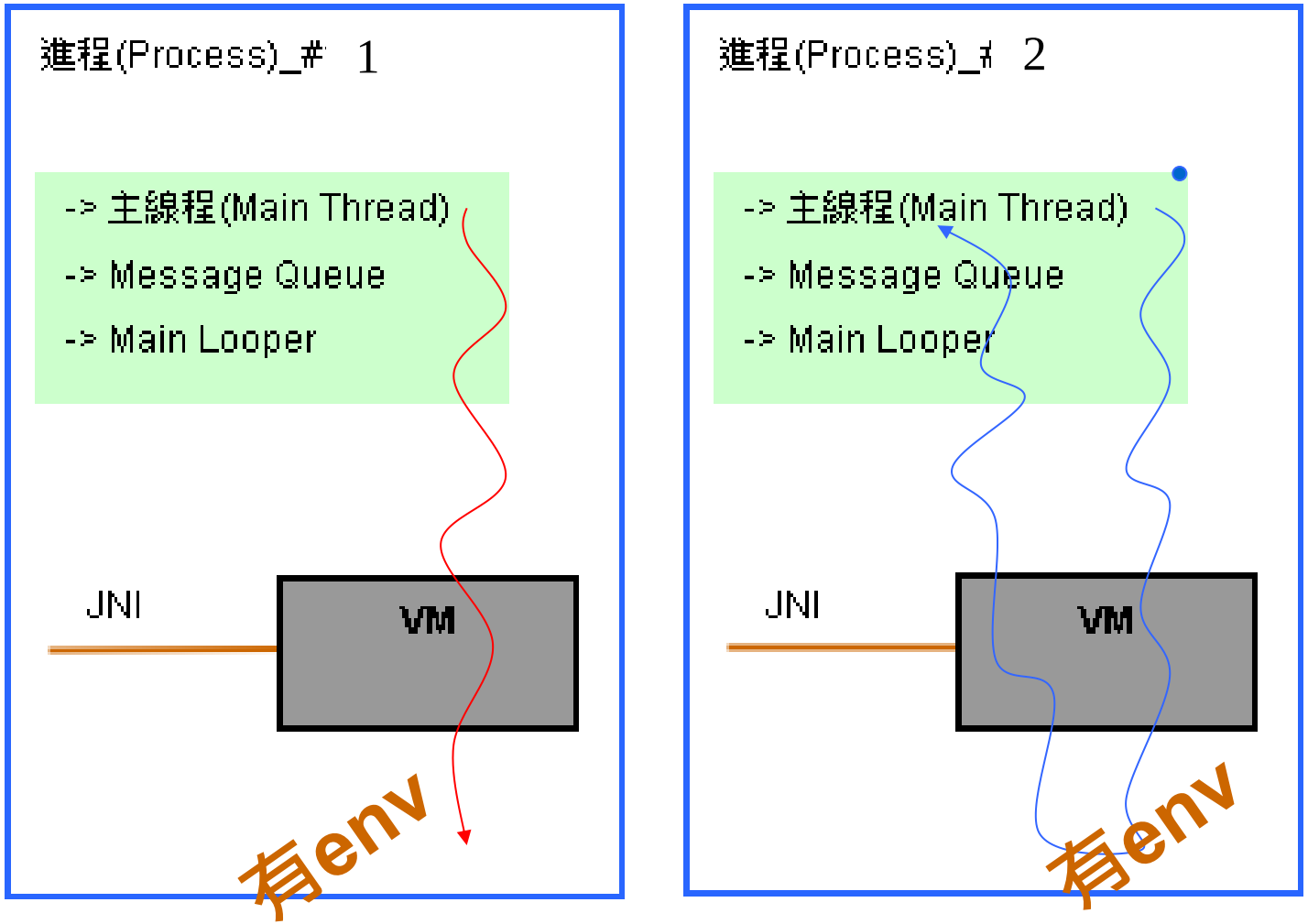

```
/* com.misoo.counter.Counter.c */
// …….
JNIEXPORT void JNICALL
Java_com_misoo_counter_CounterNative_nativeSetup
    (JNIEnv *env, jobject thiz) {    // …….    }

JNIEXPORT jobject JNICALL
Java_com_misoo_counter_actNative_nativeExec
      (JNIEnv *env, jclass clazz) {   // ……   }
}
```

- 这是Java线程透过VM进入C函数时，VM替线程而创建的对象，是该线程专属的私有对象。
- 线程透过它来要求VM协助进入Java层去取得Java层的资源，包括：取得函数或属性ID、调用Java函数或存取Java对象属性值等。

- 例如，有了env值，就能执行指令：

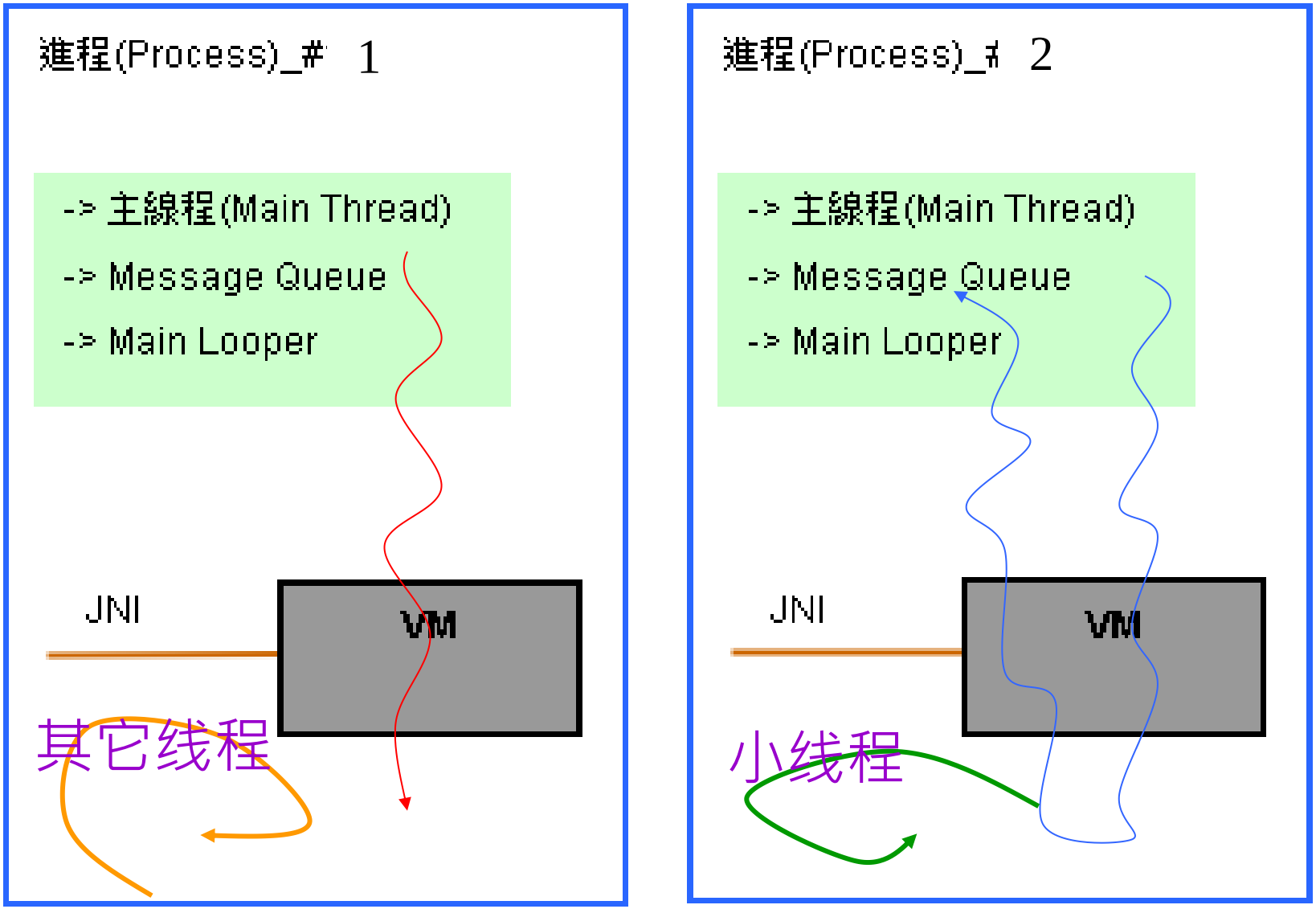
```
env->CallStaticVoidMethod(mClass, mid, sum);
```

- 其调用Java层的函数了。


進程(Process)_# 1
-> 主線程(Main Thread)
-> Message Queue
-> Main Looper
JNI
VM
有env
進程(Process)_# 2
-> 主線程(Main Thread)
-> Message Queue
-> Main Looper
JNI
VM
有env

议题

- 在C/C++层所创建的子线程，没有经过VM，所以没有JNIEnv对象，该如何要求VM协助进入Java层去取得Java层的资源，例如取得函数ID、调用Java函数呢?


進程(Process)_# 1
-> 主線程(Main Thread)
-> Message Queue
-> Main Looper
JNI
VM
其它线程
進程(Process)_# 2
-> 主線程(Main Thread)
-> Message Queue
-> Main Looper
JNI
VM
小线程

- 使用指令：

```
jvm->AttachCurrentThread(&env, NULL);
```

- 就向VM登记，要求VM诞生JNIEnv对象，并将指针存入env里。有了env值，就能执行指令：

```
env->CallStaticVoidMethod(mClass, mid, sum);
```

- 其调用Java层的函数了。

~ Continued ~