JNI技术之动态注册

一、静态注册的缺点

前面我们看了一下JNI静态注册的机制，实现机制很简单，JAVA层先加载动态库，然后在初次调用"native"方法时，就去动态库中找对应的函数，如果有则建立联系，然后调用，没有则失败。

通过前面学习，我们可以总结静态注册的缺点:
(1)C/C++实现的函数名太长了,书写起来很不方便
(2)C/C++实现的函数名，必须通过JAVA提供的工具javah提起对应的函数名
(3)初次调用'native'函数时，采用的是在动态库中寻找，如果'native'函数很多，效率将很低

其实，我们可以发现，**如果我们能在JAVA层将所有的'native'函数直接将与C/C++实现的函数关联起来，那我们在调用C/C++实现的函数时，我们就不需要到动态库中去寻找对应的函数接口了,从而提高运行效率**。如何实现这一机制了呢?

二、动态注册

通过上面的分析，我们知道，动态注册就是直接将JAVA "natvie"函数和C/C++函数关联起来，在JNI技术中，它提供了一个结构体来描述他们的关系，结构体如下所示:

```
/*
 * used in RegisterNatives to describe
 * native method name, signature,and function pointer.
 */

typedef struct {
    char *name;//JAVA层"native"函数名
    char *signature;//签名信息,用字符串表示，是返回值和参数的组合
    void *fnPtr;//记录JNI层函数的入口地址
} JNINativeMethod;
```

具体怎么使用?我们看下面JNI层(C/C++)的代码

```
1 #include <jni.h>
2 #include "HelloWorld.h"
```

```c
#include <stdio.h>

extern int jni_sayHello();

static JNINativeMethod gMethods[] = {
    {
        "sayHello",//JAVA层“native”函数
        "()V",     //签名信息
        (void *)jni_sayHello//JNI层需要实现的函数名，即函数的入口地址
    },
};

int jni_sayHello(JNIEnv *env,jobject thiz)
{
    printf("Hello word,I com from JNI\n");
    return -1;
}
```

从上面我们可以看到，在JNI层，我们首先要填充JNINativeMethod类型的结构体，JAVA层可能有多个"native"函数，所以这里是以结构体数组的形式呈现的，你可以添加所有JAVA层"native"函数和JNI层函数的一一对应关系。

在这里大家可以清楚的看到，在JNI层(C/C++)的函数接口，可以按照程序员的喜好自定指定了，不需要向静态注册那样写很长很奇怪的名字了。唯一看不明白的就是签名信息。

签名信息，实际上市由JAVA层"native"函数的返回值类型和参数类型组成的信息。为什么要这个签名信息呀，还是因为JAVA支持函数重载呀，函数名都是一样的，怎么区分呢？当然是通过返回值类型和参数类型不同来区分了，现在整明白这里为什么要搞一个签名信息了吧!

小知识:JNI签名信息

JNI的签名信息格式:
(参数1类型标识参数2类型标识.....参数n类型标识)返回值类型标识

例如:JAVA层的"native"函数
void testFunction(int a,short b,char c,String str,HelloWord obj);

它对应的JNI层签名信息如下:
(ISCLjava/lang/String;LHelloWorld;)V

其中括号内部是参数类型的标识，最右边是返回值类型标识别。

JNI规定的常用类型标识:

JAVA类型	类型标识	JAVA类型	类型标识
void	V	String	L/java/lang/String;
int	I	类	L类名
short	S	int[]	[I
char	C	Object[]	L/java/lang/object;
long	J		
float	F		
double	D		
byte	B		
boolean	Z		

注意:

(1)当参数类型是引用类型时，其格式是"L包名;",其中包名中的"."换成"/"。上面例子中的Ljava/lang/String;表示是一个Java String类型。

(2)如果JAVA类型是数组，则标识会有一个"["

(3)引用类型(除基本类型的数组外)的标识最后都一个";"

哈哈，是不是签名信息好变态呀，写起来还真麻烦，不过没关系，JAVA提供了一个叫javap的工具能帮助生成函数或变量的签名信息，它的用法如下:

javap -s -p xxx

其中xxx为编译后的class文件名，s表示输出内部数据类型的签名信息，p表示打印所函数和成员的签名信息，默认只会打印public成员和函数的签名信息。

有了javap，我们在也不需要担心写不对签名信息了。

好了，我们知道如何描述JAVA层"native"函数和JNI层函数间的关系，那如何把他们的关系告诉JAVA虚拟机呢?我们接着看代码。

```
42 //typedef const struct JNIInvokeInterface_ *JavaVM;
43 jint JNI_OnLoad(JavaVM *vm,void *reserved)
44 {
45     //typedef const struct JNINativeInterface *JNIEnv;
46     JNIEnv *env = NULL;
47     jint result = -1;
48
49     //获得JNIEnv 环境
50     if((*vm)->GetEnv(vm,(void **)&env,JNI_VERSION_1_4) != JNI_OK)
```

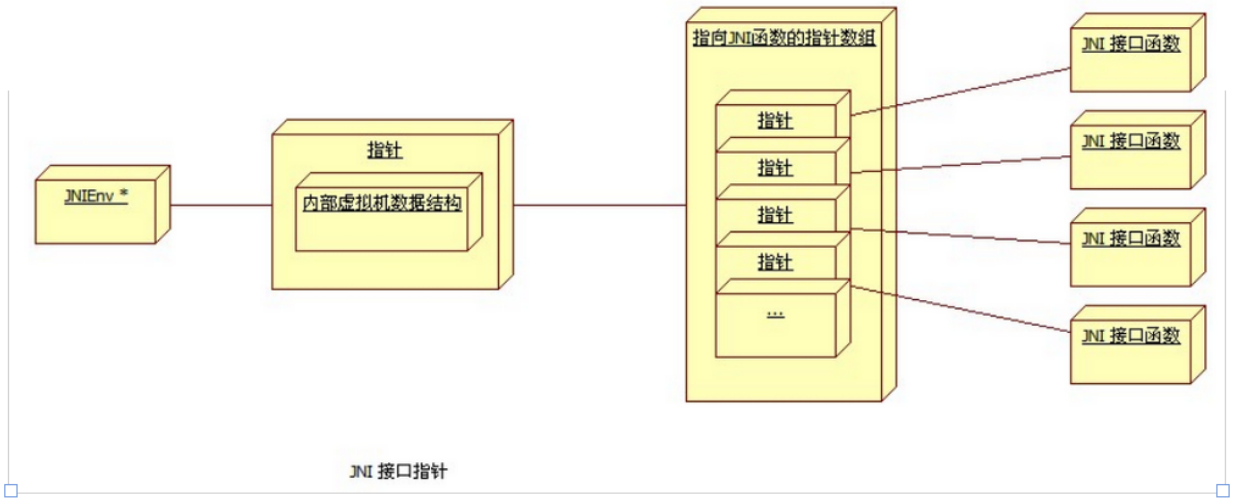
```
51      {
52          printf("JavaVM fail to get JNIEnv\n");
53          return -1;
54      }
55
56      printf("JNI_OnLoad\n");
57
58      //动态注册JNI函数
59      if(register_jni_function(env) < 0)
60      {
61          printf("Fail to register jni function");
62          return -1;
63      }
64
65      return JNI_VERSION_1_4;
66  }
```

我们看到一个JNI_onLoad函数，这个函数什么时候会调用呢?其实在Java层调用System.loadLibrary()加载动态库后，JAVA虚拟机就会去该库中寻找JNI_OnLoad 这个函数，如果有则调用它，我们的动态注册也是在这里完成的。所以如果想实现动态注册，就必须实现JNI_OnLoad这个函数，只有在这个函数中我们才有机会完成动态注册，而静态注册则没有这样的要求。

在JNI_OnLoad函数中我们看到一些不熟悉的类型，我们挨个来分析一下他们。

jint　　：JNI层，整型的标示方法
JavaVM　：用来描述Java虚拟机的数据类型
JNIEnv　：用来描述JNI层代码运行的环境，其实这个结构体中包含了很多函数指针，通过这些函数指针

我们可以操作JAVA层的对象、完成动态注册等。例如，创建Java类中的对象，调用Java对象的方法，获取Java对象中的属性等等。



当JNI层的代码被调用时，JAVA虚拟机会将JNIEnv的指针传入到JNI本地方法实现对Java端的代码进行操作。细心的读者会发现，JAVA层的"native"函数在JNI层实现的时候，总是会多两个参数，其中一个就是JNIEnv 指针，另一个是jobject 表示JAVA层的对象。

通过上面的分析，我们知道当JNI层对应的JAVA层"native"方法被调用时，JAVA虚拟机会将JNIEnv的指针传给JNI本地方法。现在并不是在调用它的时候，而是**完成注册，要想完成注册，还必须要获取JNIEnv指针**，怎么获取?

不用当心，当JNI_OnLoad函数被调用的时候，JAVA虚拟机会将JavaVm 指针传递进来，在JavaVm中有一个GetEnv函数指针可以用来获取JNIEnv指针。

回过头在看一下上面的代码，先通过JavaVM的GetEnv()获得JNIEnv指针，然后调用了register_jni_function（ ）函数完成注册，我们接着看代码

```
22 int register_jni_function(JNIEnv *env)
23 {
24     jclass clazz;
25 
26     //通过JNIEnv获取HelloWord类
27     clazz = (*env)->FindClass(env,"HelloWorld");
28     if(clazz == NULL){
29         return -1;
30     }
31     printf("FIndClass\n");
32     //调用JNIEnv的RegisterNatives函数完成注册
33     if((*env)->RegisterNatives(env,clazz,gMethods,1) < 0)
34     {
35         return -1;
36     }
37 
38     printf("RegisterNatives\n");
39     return 0;
40 }
```

通过上面的代码可以看出，先通过JNIEnv 的 FindClass()找出到了"HelloWorld"类，然后在通过JNIEnv 的RegisterNatives()完成注册。
其中jclass是在JNI层描述类的。

为什么要先知道到类呢?
我们知道Java是通过加载类来运行的，由于Java 虚拟机中有多个类，每个类都可能有"native"方法，所以在注册的时候，我们就必须告诉JAVA虚拟机，我们注册的是哪一个类的"native"方法。

好了，我们来总结一下动态注册的过程:

1.在JNI层需要填充JNINativeMethod结构体
2.在JNI层实现JAVA层"native"方法

3.在JNI层实现JNI_OnLoad函数
4.在JNI_OnLoad函数中完成动态注册