《深入理解 JNI》：JNI 函数调用

调用本地方法

在JNI中，从Java代码调用本地方法是一个核心功能。这个过程涉及到Java端的声明、本地方法的实现，以及两者之间的连接。以下是如何在JNI中调用本地方法的详细步骤。

在Java中声明本地方法

首先，在Java类中声明本地方法。使用`native`关键字标记这些方法，但不需要提供方法体。这些方法的具体实现在本地代码中完成。

```
public class NativeMethodExample {
    // 声明本地方法
    public native void nativeMethod();

    // 加载包含本地方法实现的共享库
    static {
        System.loadLibrary("native-lib");
    }

    // 主方法，用于测试本地方法
    public static void main(String[] args) {
        new NativeMethodExample().nativeMethod();
    }
}
```

在上面的代码中，`nativeMethod`是一个本地方法，它将在本地代码中实现。`System.loadLibrary`方法用于加载包含本地方法实现的共享库。

生成头文件

使用`javac`命令编译Java类，然后使用`javah`工具生成C语言的头文件。这个头文件包含了本地方法的C语言声明。

```
javac NativeMethodExample.java
javah NativeMethodExample
```

这将生成一个名为`NativeMethodExample.h`的头文件，其中包含了`nativeMethod`方法的C语言声明。

实现本地方法

接下来，编写C代码来实现头文件中声明的函数。创建一个C文件，例如`native-lib.c`，并实现`nativeMethod`函数。

```
#include <jni.h>
#include <stdio.h>
#include "NativeMethodExample.h"

JNIEXPORT void JNICALL Java_NativeMethodExample_nativeMethod(JNIEnv *env, jobject thisObj) {
    printf("Native method called!
");
}
```

在这个C函数中，我们使用了`JNIEXPORT`和`JNICALL`宏来确保函数可以被Java虚拟机正确调用。`JNIEnv *env`参数提供了访问JNI函数表的指针，而`jobject thisObj`参数是对调用该本地方法的Java对象的引用。

编译共享库

现在我们需要编译C代码并生成共享库。这个步骤会根据操作系统的不同而有所变化。

在Linux上：

```
 gcc -shared -fPIC -o libnative-lib.so native-lib.c -I${JAVA_HOME}/include -I${JAVA_HOME}/include/linux
```

在macOS上：

```
 gcc -dynamiclib -o libnative-lib.dylib native-lib.c -I${JAVA_HOME}/include -I${JAVA_HOME}/include/darwin
```

在Windows上：
使用MinGW或Visual Studio的命令行工具来编译生成DLL文件。

运行Java程序

最后，我们可以运行Java程序来测试本地方法。确保共享库位于Java的库路径中，然后运行Java类。

java NativeMethodExample

如果一切设置正确，你应该会在控制台看到输出：

Native method called!

通过这个过程，我们可以看到从Java调用本地方法的基本步骤。这个简单的示例展示了JNI的核心功能，即允许Java代码调用本地代码。在实际的项目中，本地方法可能会更加复杂，涉及到更多的数据类型转换、对象操作和异常处理等。但这个基础示例为理解JNI的工作原理提供了一个良好的起点。

调用Java方法

在前面的内容中，我们了解了如何从Java调用本地方法。现在，我们将探讨如何从本地代码（C/C++）中调用Java方法。这是JNI功能的另一个重要方面，它允许本地代码与Java对象进行交互，调用它们的方法。

获取JNIEnv指针

在本地方法中，可以通过函数参数获得一个`JNIEnv`指针。这个指针是访问JNI函数表的入口，通过它我们可以调用JNI提供的各种函数，包括调用Java方法的函数。

获取MethodID

要调用Java方法，首先需要获取该方法的`MethodID`。这可以通过调用`JNIEnv`指针的`GetMethodID`函数来实现。`GetMethodID`需要三个参数：Java对象的类、方法的名称和方法的签名。

类：可以通过`GetObjectClass`函数获取调用本地方法的Java对象的类。
方法名称：是Java方法的名字，例如"toString"。
方法签名
：是一个描述方法参数和返回类型的字符串。例如，对于没有参数且返回类型为String的方法，

其签名为`()Ljava/lang/String;`。

调用Java方法

一旦我们有了`MethodID`，就可以使用`JNIEnv`指针的相应函数来调用Java方法。根据方法的返回类型，有不同的函数可供选择，例如：

- `CallVoidMethod`：调用无返回值的Java方法。
- `CallObjectMethod`：调用返回对象引用的Java方法。
- `CallIntMethod`：调用返回int类型的Java方法。
- ...等等，对于每种返回类型都有对应的调用函数。

示例代码：从C/C++调用Java方法

假设我们有一个Java类`JavaMethodExample`，其中有一个方法`sayHello`，我们想要从C/C++代码中调用它。

```
public class JavaMethodExample {
    public void sayHello() {
        System.out.println("Hello from Java!");
    }

    public native void callSayHelloFromNative();

    static {
        System.loadLibrary("method-example");
    }

    public static void main(String[] args) {
        new JavaMethodExample().callSayHelloFromNative();
    }
}
```

```
#include <jni.h>
#include <stdio.h>
#include "JavaMethodExample.h"

JNIEXPORT void JNICALL Java_JavaMethodExample_callSayHelloFromNative(JNIEnv *env, jobject thisObj) {
    // 获取Java对象的类
    jclass cls = (*env)->GetObjectClass(env, thisObj);
    // 获取sayHello方法的MethodID
```

```
    jmethodID mid = (*env)->GetMethodID(env, cls, "sayHello", "()V");
    if (mid == 0) {
        return; // 方法未找到
    }
    // 调用sayHello方法
    (*env)->CallVoidMethod(env, thisObj, mid);
}
```

在这个示例中，我们首先获取了Java对象的类，然后获取了`sayHello`方法的`MethodID`，最后使用`CallVoidMethod`函数调用了这个方法。当运行这个程序时，它将在控制台打印出"Hello from Java!"。

调用Java方法时需要注意以下几点：

- 确保方法名称和签名正确无误。
- 处理可能出现的异常，例如方法未找到或非法访问。
- 在调用完成后，释放局部引用，以避免局部引用表溢出。

通过这种方式，JNI使得本地代码能够调用Java对象的方法，从而实现了双向的交互能力。这是JNI强大功能的体现之一，它允许Java和本地代码紧密协作，共同完成复杂的任务。