《深入理解 JNI》：JNI 数据类型与类型转换

基本数据类型映射

在JNI中，Java的基本数据类型和C/C++的基本数据类型之间有一一对应的关系。这种映射关系是JNI能够实现Java与本地代码之间数据交换的基础。以下是Java基本数据类型与C/C++基本数据类型之间的映射：

整数类型

- byte：在Java中，`byte`是有符号的8位整数。在C/C++中，它映射为`jbyte`，实际上就是一个`signed char`。
- short：Java中的`short`是有符号的16位整数，对应于C/C++中的`jshort`，即`short`类型。
- int：Java的`int`是有符号的32位整数，在C/C++中映射为`jint`，与`int`类型相同。
- long：Java的`long`是有符号的64位整数，对应C/C++中的`jlong`，在32位系统上是`long long`，在64位系统上通常是`long`。
- char：Java的`char`是无符号的16位字符，映射到C/C++中为`jchar`，实际上是一个`unsigned short`。

浮点类型

- float：Java中的`float`是单精度32位浮点数，在C/C++中映射为`jfloat`，与`float`类型相同。
- double：Java的`double`是双精度64位浮点数，对应C/C++中的`jdouble`，与`double`类型相同。

布尔类型

- boolean：Java中的`boolean`类型在JNI中有特殊的处理。它不直接映射到C/C++的任何基本类型，而是通常映射为`jboolean`，在内部实现中可能是一个字节的大小，但具体的实现依赖于JVM。

其他类型

- void：Java中的`void`类型在JNI中也有对应的表示，即`void`，用于表示没有返回值的方法。

数据类型转换注意事项

在使用JNI时，正确地处理这些数据类型的转换是非常重要的。以下是一些转换时的注意事项：

- 类型匹配：确保Java代码中的数据类型与C/C++代码中的数据类型匹配，否则可能会导致数据错误或者程序崩溃。
- 字节序问题：在不同的硬件平台上，字节序（大端或小端）可能不同。JNI提供了一些函数来处理字节序问题，如`GetBooleanArrayElements`和`ReleaseBooleanArrayElements`等。
- 符号扩展：对于有符号整数类型，从Java到C/C++的转换需要注意符号扩展问题，以避免数据错误。

示例代码：基本数据类型的传递

下面是一个简单的JNI示例，展示了如何在Java和C/C++之间传递基本数据类型：

```
public class PrimitiveTypes {
    static {
        System.loadLibrary("primitive-types");
    }

    public native void printTypes(byte b, short s, int i, long l, char c, float f, double d, boolean bo
ol);

    public static void main(String[] args) {
        new PrimitiveTypes().printTypes((byte)1, (short)2, 3, 4L, 'a', 5.0f, 6.0, true);
    }
}
```

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

JNIEXPORT void JNICALL Java_PrimitiveTypes_printTypes(JNIEnv *env, jobject obj, jbyte b, jshor
t s, jint i, jlong l, jchar c, jfloat f, jdouble d, jboolean bool) {
    printf("byte: %d", b);
    printf("short: %d", s);
    printf("int: %d", i);
    printf("long: %lld", l);
    printf("char: %c", c);
    printf("float: %f", f);
    printf("double: %lf", d);
    printf("boolean: %s", bool ? "true" : "false");
}
```

在这个示例中，Java方法`printTypes`声明了一个本地方法，该方法接受各种基本数据类型作为参数。C代码实现了这个本地方法，并打印出接收到的值。通过这个示例，可以看到如何在Java和C/C++之间传递基本数据类型，并确保数据的正确性。

正确理解和处理JNI中的基本数据类型映射是进行JNI编程的基础，它对于确保数据在Java和本地代码之间正确无误地传递至关重要。

复杂数据类型映射

在JNI中，除了基本数据类型之外，还涉及到复杂数据类型的映射，这些复杂数据类型包括数组、字符串和对象等。正确处理这些类型的数据映射是在Java和本地代码之间进行有效通信的关键。

数组

Java中的数组在JNI中有对应的表示方式。对于任何类型的数组，JNI都提供了一种统一的访问方法。

- 一维数组：在C/C++中，Java的一维数组通过`jarray`类型的指针来访问。例如，`jintArray`对应Java的`int[]`数组。
- 多维数组：对于多维数组，JNI将其视为一维数组的数组。在C/C++中，可以通过多次调用`GetArrayElements`来访问多维数组的各个维度。

示例代码：传递和返回数组

```
// Java代码
public class ArrayExample {
    static {
        System.loadLibrary("array-example");
    }

    public native int[] getArray();
    public native void setArray(int[] array);
}
```

```
// C代码
JNIEXPORT jintArray JNICALL Java_ArrayExample_getArray(JNIEnv *env, jobject obj) {
    jintArray array = (*env)->NewIntArray(env, 5);
    jint fill[5] = {1, 2, 3, 4, 5};
    (*env)->SetIntArrayRegion(env, array, 0, 5, fill);
    return array;
}
```

```
JNIEXPORT void JNICALL Java_ArrayExample_setArray(JNIEnv *env, jobject obj, jintArray array) {
    jint length = (*env)->GetArrayLength(env, array);
    jint *body = (*env)->GetIntArrayElements(env, array, 0);
    // 对数组进行操作...
    (*env)->ReleaseIntArrayElements(env, array, body, 0);
}
```

字符串

Java中的字符串是UTF-16编码的，而C/C++中的字符串通常是ASCII或UTF-8编码。JNI提供了专门的函数来处理这两种编码之间的转换。

- Java字符串到C字符串：使用`GetStringUTFChars`函数可以将Java的`String`转换为C的`char*`。
- C字符串到Java字符串：使用`NewStringUTF`函数可以将C的`char*`转换为Java的`String`。

示例代码：字符串的编码与解码

```
// Java代码
public class StringExample {
    static {
        System.loadLibrary("string-example");
    }

    public native String getStringFromC();
    public native void setStringToC(String str);
}
```

```
// C代码
JNIEXPORT jstring JNICALL Java_StringExample_getStringFromC(JNIEnv *env, jobject obj) {
    return (*env)->NewStringUTF(env, "Hello from C!");
}

JNIEXPORT void JNICALL Java_StringExample_setStringToC(JNIEnv *env, jobject obj, jstring str) {
    const char *cStr = (*env)->GetStringUTFChars(env, str, 0);
    // 对字符串进行操作...
    (*env)->ReleaseStringUTFChars(env, str, cStr);
}
```

对象

Java对象在JNI中通过`jobject`类型的指针来表示。对于特定的对象类型，JNI提供了相应的类型签名。

- 获取对象字段：使用`GetFieldID`和相应的`GetField`函数可以获取对象的字段值。
- 设置对象字段：使用`SetField`函数可以设置对象的字段值。
- 调用对象方法：使用`GetMethodID`和相应的`CallMethod`函数可以调用对象的方法。

处理复杂数据类型时，需要注意以下几点：

- 局部引用管理：JNI中的所有Java对象引用都是局部引用，它们只在当前线程的当前本地方法调用中有效。使用完毕后，应该通过`DeleteLocalRef`函数释放这些引用，以避免局部引用表溢出。
- 异常处理：在访问Java对象时，可能会抛出异常。应该使用JNI提供的异常检查函数，如`ExceptionOccurred`，并在必要时进行处理。

正确处理复杂数据类型的映射，可以帮助开发者更有效地在Java和本地代码之间传递和操作数据，从而充分利用JNI的强大功能。