자바 네이티브 메소드를 위한 통합 개발 환경

Integrated Development Environment for Java Native Methods

김상훈
세명대학교 컴퓨터학부

Sang-Hoon Kim(kimsh@semyung.ac.kr)

요약

다양한 모바일 장비의 등장으로 인하여 해당 장비에서 실행 가능한 응용 프로그램의 요구가 증가하고 있다. 플랫폼 독립적이란 특성을 가지는 자바 언어는 이러한 환경에 최적의 프로그래밍 언어로 급부상하고 있다. 그러나 자바는 가상 기계에 의해 실행되므로 플랫폼 의존적인 기능을 제공하지 못하는 단점을 가진다. 썬 마이크로시스템즈에서는 이러한 문제점을 해결하기 위해 자바 네이티브 메소드라는 JNI 기술을 제시하였다. JNI 기술을 이용하기 위해서는 JVM의 내부구조와 JNI에 대한 풍부한 지식이 필요하다. 또한 수많은 JNI 함수의 사용으로 인하여 프로그램 개발 생산성과 품질의 저하를 초래하고 있다. JNI에 대한 지식 없이 쉽고 빠르게 네이티브 프로그래밍이 가능 하도록 도와주는 지원 도구의 필요성 대두되고 있다. 이를 위해서는 자바 언어와 C/C++ 언어 사이에 존재하는 구문과 의미상 차이점을 자동적으로 처리해 주어야 한다. 본 연구에서 자바와 C/C++언어 간에 차이점과 이를 극복하기 위한 방안을 제시하고 이를 바탕으로 개발한 JNI 통합 개발 환경을 소개한다.

■ 중심어 :ㅣ자바ㅣ자바 네이티브 인터페이스ㅣ자바 가상 기계ㅣ

Abstract

As a result of a growing demand for various mobile devices, the demand for application programs on the devices is on the rise. The Java language that is platform-independent rapidly rose as the best programming language for mobile devices. However, the Java has a problem that does not support platform-dependent features needed by the application. To solve this problem, the JNI technology was introduced by Sun Microsystems. Programmers using the JNI to write native methods need to have a lot of knowledge about the JNI and the internal structure of the JVM. Also, the increased load by using a number of JNI functions may decrease software productivity and quality. Demands for tools writing native method without understanding of JNI are progressively increasing. To develop these tools, it is necessary to translate automatically the differences between Java and C/C++. In this study, I suggested a way to overcome differences between both languages and developed JNI editor that is an integrated develope environment on the basis of this.

■ keyword :ㅣJavaㅣJava Native InterfaceㅣJava Virtual Machineㅣ

접수번호 : #100527-001
접수일자 : 2010년 05월 27일
심사완료일 : 2010년 06월 29일
교신저자 : 김상훈, e-mail : kimsh@semyung.ac.kr

I. 서 론

다양한 모바일 장비들의 증가에 비례하여 콘텐츠의 수요 또한 급속히 증가하고 있다. 플랫폼에 독립적이라는 자바 언어[9]의 특성은 이러한 상황에 최적이다. 그러나 자바는 소프트웨어적으로 이루어진 자바 가상 기계(Java Virtual Machine: JVM)[10]를 사용하므로 플랫폼 의존적인 작업 및 기존 코드의 재사용의 어려우며 실행 속도 저하를 가져온다. 이 문제를 해결하기 위해 썬 마이크로시스템즈는 자바 네이티브 인터페이스(Java Native Interface: JNI)[7]라는 기술을 제시하였다.

JNI 기술은 자바와 네이티브 언어 간의 교량 역할을 제공한다. 현재는 사용 가능한 네이티브 언어로 C와 C++가 있다. JNI를 사용하기 위해서는 JVM의 내부 구조와 JNI에 대한 지식을 요구한다. 또한 JVM과의 적절한 정보 교환을 위해 다수의 전처리와 후처리 과정이 필요하다. 따라서 JNI에 대한 충분한 지식을 가지고 있더라도 각종 JNI 함수의 적절한 선택과 처리는 시간 소모적이고 지루한 작업이며, 이는 소프트웨어 개발 생산성과 품질의 저하를 가져오고 있다. 이 문제로 고생하는 개발자들에게 도움을 줄 수 있는 통합 개발 환경(Integrated Development Environment: IDE)인 JNIEditor를 본 연구에서 제시한다.

본 도구는 자바 프로그램과 C/C++ 프로그램 상호간에 자료 및 제어 전달에 필요한 다양한 전처리 및 후처리 과정을 자동으로 처리하여 프로그래머가 두 언어 간의 정보 전달 체계를 의식하지 않고 프로그래밍에 집중할 수 있다. 또한 JNI를 사용하기 위해서는 헤더 파일 생성, 네이티브 메소드 구현, 컴파일, 동적 라이브러리 생성 등 다수의 부가 작업이 필요하다. JNIEditor는 이상에서 열거한 기능을 통합적으로 수행하여 주는 통합 개발 환경이다.

역사적으로 C 언어에서 C++로, C++에서 자바로 발전하는 과정을 가진다. 이러한 역사적 배경으로 인하여 기본 문형과 자료형 측면에서 자바와 C/C++는 매우 유사한 형태를 가진다. 그러나 자바의 클래스, 필드, 메소드란 개념은 C 언어에 존재하지 않으므로 이에 대응하는 C 언어의 개념이 필요하다. 본 연구에서는 클래스를 파일로 변환하고, 필드와 메소드를 파일 영역을 가지는 외부 정적 변수와 외부 함수로 변환한다. 자바의 기본 자료형과 참조형은 C 언어의 기본형과 포인터로 대응시켜 변환한다. C++ 언어는 C 언어에 비해 엄격한 형 검사(type checking)를 수행한다는 점을 제외하면 C 언어는 C++ 언어의 진부분집합(proper subset)이다. 이 성질을 이용하여 네이티브 언어로는 C와 C++를 모두 가능하도록 하였다.

논문의 구성은 다음과 같다. 기존 환경에서 네이티브 메소드 작성의 어려움과 이를 해결하기 위한 관련 연구를 2장에서 살펴본다. 3장에서는 자바 언어와 C/C++ 언어 간의 차이를 극복하고 상호 연결시기위한 방안을 제시하고, 3장에서 제안한 방법에 따라 구현된 JNIEditor 및 사용 사례를 4장에서 설명한다. 5장 평가 및 결론에서는 연구의 성과와 제약사항을 알아보도록 한다.

II. 연구 배경

연구의 필요성에 대해 알아보도록 하자. 우선 명령어 창 또는 통합 개발 환경에서 JNI를 사용하는데 어려움을 알아보고, 이를 해결하기 위한 기존 연구와 지원 라이브러리들의 문제점에 대해 살펴본다.

최근 가장 많이 사용되는 자바를 위한 통합 개발 환경에는 Eclipse[3]와 NetBeans[1]가 있다. 이들은 프로그램 작성, 컴파일, 테스트, 디버깅, 실행 등을 통합적으로 지원하는 개발 환경으로 일반 자바 프로그램 개발에 집중되어 있다. 명령어 창 또는 기존 통합 개발 환경에서 네이티브 메소드를 구현하는 과정과 그의 어려움을 알아본다. 네이티브 메소드 선언 sumArray()를 포함한 SumTester 클래스인 [그림 1]과, 그 선언에 해당하는 네이티브 메소드 구현인 [그림 2]를 살펴보자. 네이티브 메소드를 구현한 함수는 JNI 명세에 따라 변형되어 네이티브 메소드 선언과 다른 형태의 함수 원형을 사용하여야 한다. 그리고 그 함수의 매개 변수 arr로 전달된 정수 배열은 JVM 내부에 유지되므로 C 코드에서 직접 접근할 수 없다. 배열 객체를 접근하기 위해서는 버퍼

를 할당 후, 그 버퍼로 정수형 배열을 얻어오는 JNI 함수 GetIntArrayRegion을 사용하거나, 또는 [그림 2]와 같이 배열의 원소로 직접 접근할 수 있는 포인터를 얻어오는 GetIntArrayElements라는 JNI 함수를 사용하여야 한다. 전달된 배열을 가지고 작업을 수행한 후, 더 이상 네이티브 코드가 자바의 기본형 배열을 접근하지 않는 다는 것을 JVM 알리기 위해 ReleaseArrayElements를 호출하여야 한다. 이를 행하지 않으면 변경된 배열의 내용이 JVM에 반영되지 않는다.

```
import java.util.*;
public class SumTester {
    private native int sumArray(int[] arr, size);
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        SumTester st = new SumTester();
        int[] arr = new int[5];
        for(int i = 0; i < arr.length; i++)
            arr[i] = scan.nextInt();
        scan.close();
        System.out.println(st.sumArray(arr));
    }
    static { System.loadLibrary("SumTester"); }
}
```

그림 1. class SumTester

```
#include "SumTester.h"

JNIEXPORT jint JNICALL
Java_SumTester_sumArray(JNIEnv *env, jobject obj,
    jintArray arr, jint size){
    jint * carr;
    jint sum = 0;
    int i;
    carr = (*env)->GetIntArrayElements(env, arr, NULL);
    if(carr == NULL) return 0;
    for(i=0; i < size; i ++)
        sum += carr[i];
    (*env)->ReleaseIntArrayElements(env, arr, carr, 0);
    return sum;
}
```

그림 2. 네이티브 메소드 sumArray

단순히 배열의 합을 구하는 프로그램에서 조차도 JNI에서 요구하는 형태로 변형된 네이티브 함수 이름의 작성 및 JNI가 사용하는 자료형의 이해가 필요하며 자바 배열을 접근하기 위한 전처리와 후처리 과정을 요구하고 있다. 이는 JVM과의 인터페이스인 JNI를 충분히 습득한 프로그래머에게 조차도 귀찮고 지루한 작업이며 이로 인하여 코딩 오류를 유발시킬 가능성이 크고 개발 생산성 저하를 야기한다.

기존 개발 환경을 사용한 네이티브 메소드 구현의 어려움을 해소하기 위해 이러한 복잡한 처리 과정을 C++의 template 기능을 이용하여 캡슐화를 시도한 연구 "JNI - C++ inegration made easy"가 있다[4]. [그림 3]은 이 연구에서 제시한 JNI 캡슐화 프레임워크를 사용하여 네이티브 메소드 구현 예이다.

```
// JniExample.java
public class JniExample {
    public int intField = 17;
    public int[] intArray = new int[2];
    ...
    private static native void native_call(JniExample x);
    ...
}

// jni_example.cpp
JNIEXPORT void JNICALL Java_JniExample_native_1call( //㉮
    JNIEnv* env, jclass clazz, jobject obj) {
    // Lookup the Java fields in 'obj'
    JNIField<jint> intField(env, obj, "intField"); //㉯
    JNIStringUTFChars str(env, "JniExample", "stringField"); //㉰

    // Set new values
    intField = 0;
    arr[0]=0; arr[1]=0;
}
```

그림 3. JIN 캡슐화 프레임워크를 사용한 코드

[그림 3]의 "jni_example.cpp"의 함수 원형(㉮)을 살펴보면 기존 방식과 동일하게 변형된 함수 원형을 사용하여야 한다. 다시 말해서 JNI 명세를 인지해야 한다는 것이다. C++의 constructor와 destructor가 각각 전처리와 후처리를 담당하고 있다. 따라서 네이티브 코드에서 자바 객체를 접근하기 위해서는 네이티브 코드 구현자가 접근하는 멤버 단위로 [그림 3]의 ㉯, ㉰와 같은 코드를 삽입하여 주어야 한다. 이 방법은 JNI 함수 복잡한 사용 부담을 덜어주는 장점은 가지나 대신 위 연구에서

제시한 템플릿의 사용이란 새로운 부담이 생기고 있다. 마지막으로 C++ 템플릿을 기반으로 하고 있어 네이티브 언어로 C 언어를 사용할 수 없다는 한계점을 가진다.

이외에 현재 기존 네이티브 코드로 작성된 라이브러리의 수월한 호출을 가능하도록 도와주는 JNA[8], HawtJNI[5], GlueGen[6] 등이 있다. 이들 도구들은 자바 환경에서 JNI의 존재를 의식하지 않고 네이티브 함수들을 자유로이 호출할 수 있도록 도와주는 라이브러리 또는 도구들이다. 그러나 C 환경에서 자바 객체의 접근에 대한 고려는 없거나 또는 빈약하다는 단점을 가진다. 또한 JNI를 대신하는 새로운 클래스와 메소드를 습득해야하는 추가 부담을 가진다.

본 연구에서는 네이티브 언어로 C와 C++ 모두 사용 가능하며 변형되지 않은 네이티브 메소드 선언을 그대로 사용하고자 한다. 또한 자바 언어와 네이티브 언어 상호간에 JNI의 존재를 의식하지 않고 접근 할 수 있도록 하고자 한다. 더구나 기존 통합 개발 환경은 JNI 네이티브 메소드 번역 및 실행을 위해 많은 부가적인 작업을 수동으로 해야 한다. 이 또한 매우 불편한 일이다. 따라서 네이티브 코드 작성, javah를 사용한 헤더파일 생성, 동적 라이브러리 생성, 실행을 통합적으로 수행할 수 있는 도구를 개발하고자 한다.

Ⅲ. 자바와 네이티브 언어의 연결

JNIEditor는 자바 환경과 네이티브 코드 환경간의 차이를 극복해주는 코드를 자동으로 생성하여 주어야 한다. 이를 지원하기 위한 네이티브 코드 환경의 기본 구조와 수행 작업에 대해 알아보도록 하자.

자바 객체와 네이티브 메소드 상호간의 접근은 다음 두 가지 관점으로 나누어 고려해야 한다. 첫째는 자바 객체에서 네이티브 함수를 호출하는 것이고, 두 번째는 네이티브 함수에서 자바 객체의 필드를 접근하고 자바 메소드를 호출하는 것이다. JNI의 존재를 의식하지 않고 상호간에 접근 또는 호출을 가능하도록 하기 위해서는 두 언어 간에 구문 구조 및 의미의 차이를 해결하여 주어야 한다. 자바의 기본 자료형과 C/C++의 기본 자료형은 유사하나 동일하지 않다. 구조 자료형(structured data type)은 다른 모양을 가진다. 따라서 자료형 간의 변환이 필요하다. 매개 변수의 전달에서도 차이를 보인다. 기본 자료형에서는 두 언어 모두 값 전달 방식(call by value)을 취하고 있으나 구조 자료형의 경우 자바는 C/C++의 포인터 또는 참조 전달의 형태만을 취하고 있다. JNIEditor를 사용하는 개발자는 C/C++와 자바 언어를 모두 알고 있는 프로그래머를 대상으로 한 것이므로 이러한 차이점의 극복은 문제가 되지 않는다.

본 논문에서는 유사하나 역할에 있어 중요한 차이를 보이는 몇 가지 메소드(함수)가 등장한다. 이를 간단히 그리고 명료하게 설명하기 위해 용어를 명확히 정의가 필요하다. 자바 클래스에 나타나는 네이티브 메소드 선언에 대응되며, JNI 명세에 따른 함수 헤더를 가지는 함수를 자바 네이티브 메소드 구현이라 하겠다. 그리고 네이티브 메소드 구현의 실재 실행 코드를 가지고 있는 함수를 C 언어에서는 메소드라는 용어 대신 함수란 용어를 사용하고 있으므로 네이티브 함수라 기술할 것이다. 이 네이티브 함수에서 자바 메소드를 직접 호출할 수 없다. 호출하기 위해서는 메소드 ID를 얻어와 Call<Type>Method를 사용하여 간접적으로 호출하게 된다. 또한 매개변수의 변환이 필요하다. 이러한 과정을 대리로 처리하여 호출해 주는 함수를 프락시 함수(proxy function)라 하겠다.

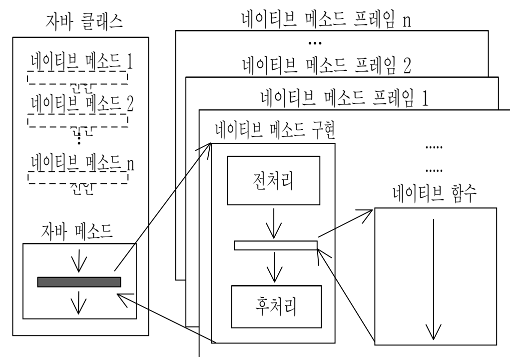

그림 4. 자바 메소드와 네이티브 메소드 간의 관계

네이티브 함수에서 필드와 메소드를 자유로이 접근

하기 위해서는 그의 참조가 네이티브 언어 환경 내에 미리 정의되고 초기화되어 있어야 한다. 이러한 참조는 네이티브 메소드 단위로 존재한다. 이러한 참조와 네이티브 함수 그리고 부가 선언들을 포함하고 있는 파일 단위의 구조를 네이티브 메소드 프레임이라 하겠다. 네이티브 코드 환경은 네이티브 메소드의 개수만큼의 네이티브 메소드 프레임을 가지며 이는 파일로 만들어진다. 네이티브 메소드와 자바 객체가 상호간에 접근을 처리하여 주는 다양한 변수 및 함수 선언으로 이루어진 네이티브 메소드 프레임은 파일 영역을 갖는다. [그림 4]는 이상에서 설명한 자바 메소드, 네이티브 메소드 선언, 네이티브 메소드 구현, 네이티브 함수 들 간의 관계를 그림으로 표현한 것이다.

3.1 네이티브 함수의 호출

자바 메소드에서 네이티브 함수의 호출은 네이티브 메소드 구현 부분을 경유하여 간접 호출이 이루어진다. 이를 그림으로 도시하면 [그림 5]이며 그림의 원 숫자는 호출 순서이다. 자바 메소드는 네이티브 메소드 선언을 참조하여 호출을 한다. [표 1]은 네이티브 메소드 선언, 네이티브 메소드 구현, 네이티브 함수의 헤더를 보여주고 있다. 네이티브 메소드 선언과 달리 JNI 명세 규칙에 따라 작성된 네이티브 메소드 구현은 아주 상이한 형태를 가진다. 그러나 네이티브 메소드 선언과 네이티브 함수는 유사한 형태를 보여 주고 있다. 네이티브 메소드 구현자는 네이티브 함수의 원형을 참고하여 네이티브 프로그램을 작성하게 된다.

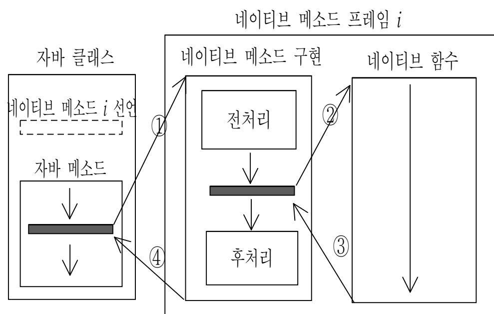

그림 5. 네이티브 함수 호출

표 1. 네이티브 메소드의 종류별 헤더

메소드 종류	메소드 헤더
네이티브 메소드 선언	public native int getPort(short num, String str)
네이티브 메소드 구현	JNIEXPORT jint JNICALL Java_NMSim_getPort(JNIEnv *_env, jclass _cls, jshort num, jstring str)
네이티브 함수	long getPort(short num, const char * str)

자바 메소드가 네이티브 함수를 호출하는 경우 매개변수로 전달되는 자료와 반환으로 돌려받는 값에 대해 고려하여 보자. 기본 자료형은 두 언어 간에 유사하다. 매개 변수 전달 방식은 모두 값 전달 방식을 사용하고 있으므로 [표 2]와 같이 대응하는 자료형으로의 단순 변환만으로 직접 접근하여 사용가능하다. 자바에서 구조 자료형은 C의 입장에서 보면 모두 포인터이다. 또한 직접 접근은 불가능하며 JNI을 경유하여 접근하여야 한다. 따라서 자바에서 전달받은 구조 자료형의 객체는 C 언어에서 접근할 수 있는 형태로 변환해 주어야 한다. 자바는 구조 자료형으로 배열, 클래스, 인터페이스를 제공하고 있다. 그러나 네이티브 코드에서 복잡한 구조를 가지는 객체를 접근하는 경우는 드물며 다단계의 전후 처리 부담으로 인하여 비효율적인 구현이 되기 쉽다. 따라서 본 연구에서는 자바 참조형 중에서 접근 가능한 자료형은 스트링과 기본형 배열로 제한하고 있다.

표 2. 기본 자료형 매핑

자바	C/C++ 언어	자바	C/C++ 언어
boolean	unsigned char	long	long long
byte	signed char	float	float
char	unsigned short	double	double
short	short	void	void
int	long		

매개 변수로 전달된 자바 스트링은 JNI 함수인 GetStringUTFChars을 사용하여 C 스트링으로 변환하여 읽어 온다. 이때 자바 스트링의 자료형은 jstring이

며 C 스트링의 자료형은 const char* 이다. 반환된 스트링의 사용을 종료하면 ReleaseStringUTFChars를 호출하여 C 스트링을 위해 사용된 메모리를 free해 주어야 한다. 스트링과 마찬가지로 배열도 C 환경에서 직접 접근하여 사용할 수 없다. C 환경에서 기본형 배열을 직접 접근할 수 있도록 배열로의 포인터를 얻어오기 위해 JNI 함수 Get<Type>ArrayElements를 사용하고, 사용을 종료한 후 이 사실을 JVM에 알리고 변경사항을 반영하기 위해 Release<Type>ArrayEmements를 호출해야 한다. C 언어의 배열과는 달리 자바 배열은 원소의 개수를 나타내는 length 속성을 가진다. 이 차이점을 해소하기 위해 매개 변수로 전달된 각 배열 마다 length 변수가 추가적으로 필요하다. 메소드 이름과 배열 명 그리고 length를 연결한 methodName_arrayName_length 형태의 정적 외부 변수를 두어 처리한다. 매개변수로 전달된 기본형 배열과 이에 대응하는 C 언어의 배열 형 간의 대응관계는 byte[]는 char *, short[]는 short *, int[]는 long * 등이며 [표 2]의 나머지 자료형도 유사한 형태로 대응된다. 지금까지 매개변수를 통하여 자바 환경에서부터 네이티브 언어 환경으로 전달된 객체의 접근 방법을 자료형별로 나누어 살펴보았다.

3.2 자바 객체의 필드 접근

네이티브 함수에서 자바 객체의 필드를 접근하는 방법에 대해 살펴보도록 하자. 자바의 필드는 매개변수와는 달리 기본 자료형도 직접 접근이 불가능하다. 필드의 값을 참조하기 위해서는 GetFieldID와 Get<Type>Field를 사용하여 필드의 값을 네이티브 언어 환경으로 가져와야 한다. 또한 네이티브 함수에서 필드의 값을 변경하였다면 Set<Type>Field를 사용하여 변경 사항을 자바 객체에 반영시켜야 한다. 이와 같이 기본 자료형에서조차도 전처리와 후처리 과정이 요구된다. 자바 객체의 필드에 해당하는 변수를 네이티브 언어 환경인 네이티브 메소드 프레임 파일의 정적 외부 변수로 등장 시키고 전 후처리하는 과정이 필요하다.

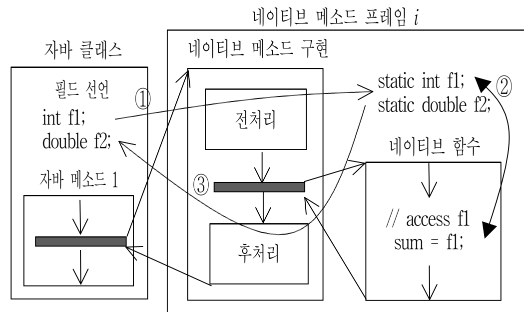

그림 6. 필드의 접근과 변경 반영

필드의 접근 및 변경 반영 과정을 [그림 6]에서 보여주고 있다. JNIEditor는 네이티브 함수 구현자가 접근이 필요한 필드를 선택하도록 하고 있다. 모든 필드를 네이티브 메소드 프레임 영역으로 이동 시킨다면 사용하지 않는 필드가 발생할 수 있다. 사용하지 않는 필드에 대한 전처리와 후처리 과정은 불필요한 자원의 낭비를 초래한다. 따라서 필요한 필드를 사용자가 선택하도록 하였다. 사용자에 의해 선택된 각 필드는 그에 해당하는 정적 외부 변수로 C 환경에 생성되고, 전처리 과정을 통하여 정적 외부 변수의 값을 해당 필드의 값으로 초기화된다. 이후 네이티브 함수에서 이 정적 외부 변수를 자바의 필드로 고려하여 자유로이 접근할 수 있다. 사용이 종료 된 후, 즉 네이티브 함수의 종료 직 후 변경된 필드의 내용은 후 처리 과정을 통하여 자바 객체의 실제 필드에 반영하게 된다. 나머지 자료형들 간의 대응관계와 접근 방법은 매개변수와 동일하다. 본 방법에 제약사항이 존재한다. 네이티브 함수에서 필드의 값을 변경한 후 자바 메소드를 호출하여 그 필드의 값을 참조한다면 변경된 값을 접근할 수 없다. 이는 네이티브 함수의 종료 후 수행하는 후처리 과정에서 변경에 대한 반영을 수행하기 때문에 발생하는 문제이다. 이를 해결하기 위해서는 모든 자바 메소드 호출 전에 변경에 대한 반영을 수행해야 하나 이는 전체 환경에 너무 많은 부담을 초래하므로 이를 제한하기로 하였다.

3.3 자바 메소드의 호출

네이티브 함수에서 자바 객체의 메소드를 호출하는

call-back에 대해 알아보도록 하자. 네이티브 언어 환경에 존재하는 네이티브 함수에서 자바 메소드를 호출하기 위해서는 다음 과정이 필요하다. 첫째 GetObjectClass와 GetStaticMethodID를 호출하여 메소드 식별자를 얻어야 한다. 그 다음 반환 자료형이 Type인 메소드를 호출하기 위해서는 Call<Type>Method를 사용하여 호출하게 된다. 이 때 네이티브 함수에서 자바 메소드로 전달하는 자료와 자바 메소드에서 반환 받는 자료의 변환이 필요하다. 자료의 변환과 자바 메소드의 호출을 함께 모아 처리하여 주는 프락시 함수 개념을 도입하여 처리하도록 하였다. 프락시 함수는 매개 변수로 전달된 자료의 변화, 자바 메소드 호출, 반환된 자료의 변환이란 3가지 작업을 수행하게 된다. 첫 단계는 프락시 함수가 전달 받은 매개 변수를 자바 메소드가 접근할 수 있는 자바 객체로 변환하는 전처리 과정이다. 기본 자료형은 [표 2]의 자료형 매핑 테이블에 따라 단순 형 변환으로 직접 전달 가능하다. 그러나 구조 자료형에 대해서는 자바 객체를 생성하는 과정을 수행해야 한다. C 스트링의 경우는 자바 스트링으로 변환하기 위해 NewStringUTF를 사용해야 되며, 배열의 경우는 New<Type>Array를 사용하여 새로운 자바 배열 객체를 생성하여야 한다. 이 때 C 언어에서 배열은 배열 원소의 개수에 대한 정보를 가지고 있지 않다. 따라서 call-back에 의해 호출 가능한 메소드의 매개 변수가 배열을 포함하고 있다면 methodName_arrayName_length형태의 미리 정의된 정적 외부 변수를 가지고 있어야 하며, 자바 메소드 호출 전에 네이티브 함수에서 초기화해 주어야 한다. 두 번째 단계는 Call<Type>Method를 사용하여 호출하고 반환 값이 있으면 이를 반환 받는 단계이다. 마지막 후처리 단계는 반환받은 값을 네이티브 함수에서 접근 가능한 형태로 변환한 후, 프락시 함수를 호출한 네이티브 함수로 값을 되돌려 보내는 작업으로 네이티브 메소드 구현의 전 처리 단계와 유사하다. 이상에서 설명한 내용을 네이티브 함수에서 자바 메소드의 호출 과정을 [그림 7]에서 보여주고 있다.

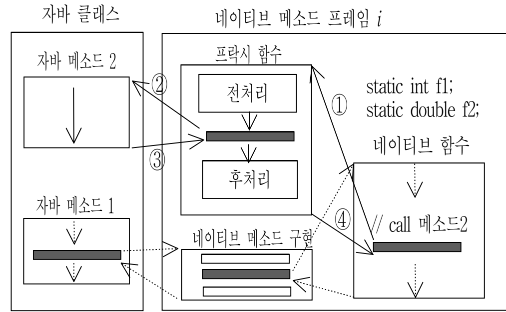

그림 7. 자바 메소드의 호출

Ⅳ. JNIEditor의 구현과 사용사례

자바 객체와 C/C++ 함수 간에 JNI를 의식하지 않고 자유로이 상호 접근하기 위해서는 네이티브 메소드 프레임이 어떠한 구조를 가져야 하는지 3장에서 살펴보았다. 본 장에서는 JNIEditor의 구성과 실제 사용 사례를 통해 본 시스템의 이해와 필요성, 그리고 방법의 옳음을 보이고자한다.

이 시스템에서 네이티브 메소드 프레임을 생성하기 위해서는 클래스 파일로부터 필드, 메소드, 매개 변수 등의 주요 정보를 얻어야 한다. 이를 위해 Apache Jakarta Project 중에 하나인 BCEL(Byte Code Engineering Library)[2]을 사용하였다. BCEL 라이브러리를 사용하여 얻어진 클래스 정보를 바탕으로 3장에서 언급한 네이티브 메소드 프레임을 생성한다. 주변 환경은 GUI 패키지인 swing을 사용하여 자바로 구현하였으며 전체 구조는 [그림 8]과 같다.

[그림 8]에서 막대 인간이 지시하는 자바 클래스와 네이티브 함수 틀 부분을 제외하면 나머지 부분은 자동으로 지원되는 부분이다. 사용자는 네이티브 메소드 선언을 포함한 자바 클래스를 작성하고 네이티브 함수 프레임 내부에 위치한 네이티브 함수 틀에 C/C++를 사용하여 코드를 작성한다. 완성된 네이티브 함수를 포함한 각각의 네이티브 메소드 프레임들은 생성되어 파일로 만들어진다. 이 파일들은 C/C++ 컴파일러에 의해 한 개의 동적 라이브러리로 구성된다. 마지막으로 자바 클래스 파일과 라이브러리는 JVM에 함께 적재되어 실행된

다. 이상의 과정은 JNIEditor에 의해 자동으로 이루어진다.

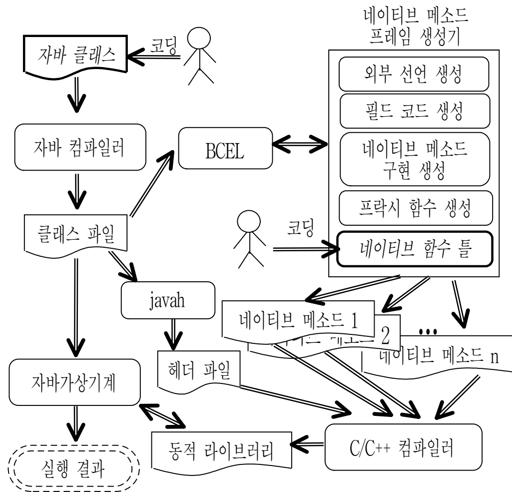

그림 8. JNIEditor의 구조

네이티브 메소드 선언을 포함한 자바 클래스를 예제로 하여 본 도구를 사용하는 방법과 생성된 프로그램을 살펴보도록 하자. [그림 9]는 [그림 10]의 class NTester의 번역된 클래스 파일을 읽어 들인 상태이다. 영역 ①은 필드 선택 창으로 자바 클래스의 필드를 보여주고, 영역 ②는 자바 클래스의 메소드를 보여주고 있다. 영역 ③은 명령어 라인 매개변수를 직접 입력할 수 있는 창이 있다. 영역 ④ 리스트 박스는 자바 클래스에 포함된 네이티브 메소드 선언을 가지고 있다. 사용자가 리스트 박스에서 네이티브 메소드 선언을 선택하면 영역 ⑥에 그에 해당하는 네이티브 메소드 프레임이 나타나며, 그 내부에 C/C++ 코드를 작성할 수 있는 네이티브 함수 틀인 영역 ⑤가 포함되어 있다 영역 ⑦은 표준 출력 또는 표준 오류를 출력하기 위한 윈도우를 가지고 있다.

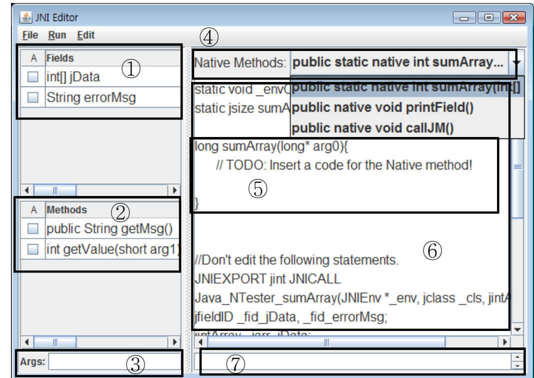

그림 9. JNIEditor의 실행

```
public class NTester {
    int jData[] = {3, 5, 7, 9};
    String errorMsg;

    public native static int sumArray(int[] data);
    public native void printField();
    public native void callJM();

    public String getMsg() { ... }
    int getValue(short index){ ... }
    public static void main(String[] args) {
        int data[] = {3, 5, 7, 9};
        NTester nt = new NTester();
        System.out.println(sumArray(data));
        nt.printField();
        nt.callJM();
        System.out.print("In Java: ");
        for(int val: nt.jData) System.out.print(val + ",");
        System.out.println();
    }
    static { System.loadLibrary("NTester"); }
}
```

그림 10. Class NTester

[그림 10]의 첫 번째 네이티브 메소드인 sumArray는 매개변수로 넘어온 정수형 배열의 내용을 합하여 결과를 반환하는 함수라 가정하자. C 언어에서 배열은 원소의 개수를 포함하고 있지 않다. 따라서 배열의 길이에 대한 정보가 변수 sumArray_arg0_length에 유지하고 있다. 이 변수의 값은 네이티브 메소드 구현의 전처리 단계에서 초기화 된다. 배열의 길이 정보를 이용하여 구현된 네이티브 함수는 [그림 11]과 같다.

```
long sumArray(long* arg0){
    long sum = 0;
    int i = 0;
    for(i = 0; i < sumArray_arg0_length; i++)
        sum += arg0[i];
    return sum;
}
```

그림 11. sumArray() 함수의 구현

다음으로 자바 필드를 접근하는 예를 알아보자. [그림 10]의 두 번째 네이티브 메소드에서 필드 jData의 내용을 그대로 출력하고 값을 변경하여주는 네이티브 함수를 작성하여 보자. 불필요한 전처리 부담을 경감하기 위해 필드 선택 창에서 접근할 필드를 사용자가 선택하여야 한다. 이는 [그림 12]의 ①과 같이 네이티브 환경으로 이동된 배열과 배열 길이 선언한 코드의 생성을 야기 시킨다. 선택을 해제한다면 ① 코드 부분을 자동으로 삭제될 것이다. 필드 또한 매개 변수 처리와 마찬가지로 네이티브 메소드 구현의 전처리 부분에서 초기화되고 변경사항은 후처리 부분에 의해 반영될 것이다.

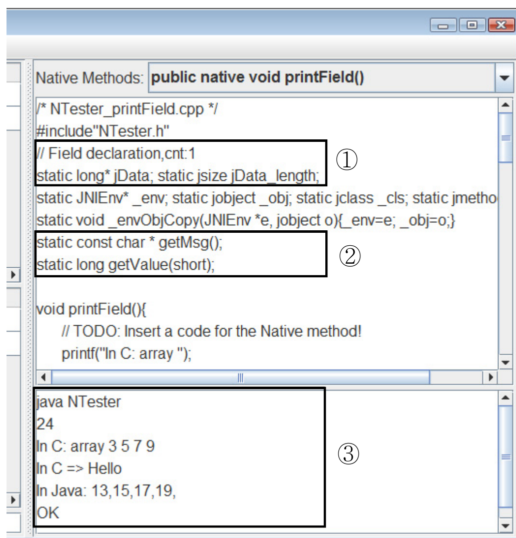

그림 12. 네이티브 메소드 printField()

[그림 13]은 필드의 값을 모두 출력하고 각 원소에 10을 더하는 프로그램이다. 네이티브 함수 printField()의 실행 결과인 “In C: array 3 5 7 9”와 배열을 갱신한 내용을 자바에서 출력한 결과인 “In Java:13,15,17,19,”를 [그림 12]의 ③에서 확인할 수 있다.

```
void printField(){
    // TODO: Insert a code for the Native method!
    printf("In C: array ");
    for(int i = 0; i < jData_length; i++) {
        printf("%d ", jData[i]);
        jData[i] += 10;
    }
    printf("\n");
}
```

그림 13. printField() 함수의 구현

```
void callJM(){
    // TODO: Insert a code for the Native method!
    printf("In C => %s\n", getMsg());
}
```

그림 14. callJM() 함수의 구현

마지막으로 네이티브 함수에서 자바 메소드를 호출하는 Call-back에 대해 알아보도록 하자. [그림 12]의 ②는 일반 자바 메소드에 해당하는 프락시 함수 선언이다. 이 메소드의 존재는 [그림 9]의 ②와 [그림 10]에서 각각 확인할 수 있다. 이 프락시 함수 선언은 정적 네이티브 메소드 sumArray의 프레임 영역에서 나타나지 않는다. 정적 네이티브 메소드에서는 인스턴스 메소드를 호출할 수 없기 때문이다. 그러나 printField()는 인스턴스 네이티브 메소드이므로 프락시 함수 선언부가 필요하고, 또한 프락시 함수 정의가 네이티브 메소드 프레임 하단에 포함되어 있다. 네이티브 함수에서 자바 메소드를 호출하여 문자열 자료를 얻어 출력하는 프로그램은 [그림 14]와 같다. 프로그램의 실행 결과는 [그림 12]의 ③에서 "In C:=>Hello"로 알 수 있다. 이와 같이 자바 메소드를 대리로 호출하여 주는 프락시 함수를 두어 네이티브 함수에서 일반 C 함수를 호출하는 방법과 동일하게 자바 함수를 호출할 수 있음을 보여주었다.

이상에서 설명한 방법으로 각 네이티브 메소드 선언 당 한 개의 네이티브 메소드 프레임이 생성되며 이는 파일 형태로 저장되어진다. 자바 클래스 NTester는 3

개의 네이티브 메소드 프레임 파일이 생성되어지며 공유 라이브러리 형태로 번역되고 링크되어 JVM에 의해 사용된다. [그림 15]는 JNIEditor에 의해 자동으로 생성된 네이티브 메소드 프레임 파일 중 "public native void printField()"를 위한 프레임 파일이다.

```
/* NTester_printField.cpp */
#include"NTester.h"
// Field declaration,cnt:1
static long* jData; static jsize jData_length;
static JNIEnv* _env;
... 중략 ...
static long getValue(short);

void printField(){
    // TODO: Insert a code for the Native method!
    ... 중략...
}

//Don't edit the following statements.
JNIEXPORT void JNICALL
Java_NTester_printField(JNIEnv *_env, jobject _obj){
jclass _cls = _env->GetObjectClass(_obj);
jfieldID _fid_jData, _fid_errorMsg;
jintArray _jarr_jData;
jstring _jstr_errorMsg;
_envObjCopy(_env, _obj);
//s,f,0,4
_fid_jData = _env->GetFieldID(_cls, "jData", "[I");
... 중략 ...
//<-
    printField();
//->
//e,f,0,1
_env->ReleaseIntArrayElements(_jarr_jData, jData, 0);
}
const char * getMsg(){
jstring _retVal;
const char * _nRetVal;
_cls = _env->GetObjectClass(_obj);
_mid = _env->GetMethodID(_cls, "getMsg",
"()Ljava/lang/String;");
_retVal = static_cast<jstring>(_env->CallObjectMethod(_obj,
_mid));
_nRetVal = _env->GetStringUTFChars(_retVal, NULL);
return _nRetVal;
}
long getValue(short arg1){
    ... 중략 ...
}
```

그림 15. 자동 생성된 메소드 프레임

V. 평가 및 결론

플랫폼이 다양해짐에 자바의 사용분야가 급속히 증가하고 있는 추세이다. 그러나 플랫폼 의존적인 작업의 한계를 극복하기 위해 JNI을 사용해야 한다. JNI 기술을 이용하기 위해서는 부가적 지식과 수많은 JNI 함수의 사용으로 인하여 프로그램 개발 생산성과 품질의 저하를 초래하고 있다. 이러한 문제점은 기존의 통합 개발 환경에서 해결하지 못한다. C++ template class에 JNI 전처리 및 후처리 캡슐화를 시도한 연구에서는 이상의 문제점은 어느 정도 완화시키고 있다. 그러나 template class의 사용으로 인하여 C 컴파일러의 사용이 불가능하고 또한 template class의 사용법 습득이란 부담이 발생한다. HawtJNI, GlueGen, JNA 등 은 자바 환경에서 네이티브 함수로의 호출에 중점을 두고 있어 C 환경에서 자바 객체의 필드와 메소드 접근이 어려움을 가진다. 이에 비해 본 JNIEditor은 두 언어의 양 방향 접근이 가능하며 추가적 지식 습득의 부담이 없음을 4장에 있는 [그림 11][그림 13][그림 14]의 사용자 작성 네이티브 코드를 보면 확인할 수 있다. 본 아이디어의 옳음을 확인하기 위해 4장에서 실제 구현된 JNIEditor를 사용하여 얻어진 실행 결과를 보여주었다.

본 도구가 지원하는 자료형은 기본 자료형과 배열, 그리고 스트링으로 제한된다. JNI의 일반적 사용은 기존 코드의 재사용, 실행 속도 향상, 플랫폼 의존적인 작업의 수행이다. 우선 네이티브 코드로 작성된 코드의 재사용은 자바 클래스 제작 이전에 존재하는 코드이므로 해당 객체의 내부를 접근하는 코드는 존재하지 않는다. 두 번째로 실행 속도 향상을 위해 네이티브 메소드를 사용하는 경우이다. 복잡한 객체의 잦은 내부 접근은 오히려 실행 속도의 저하요인으로 등장한다. 이는 이전 연구에서 실험을 통하여 입증하였다. 마지막은 플랫폼 의존적인 작업을 하고자 네이티브 메소드를 사용한 경우이다. 이는 대부분 하드웨어나 운영체제를 직접 제어하거나 또는 자료를 교환하는 경우이므로 기본 자료형 또는 단순 배열로 작업이 가능하다.

참 고 문 헌

[1] Amit Kumar Saha, "Beginning JNI with NetBeans C/C++ Pack 6.0, Linux," http://cnd.netbeans.org/docs/jni/nb6-linux/beginning-jni-linux.html

[2] BCEL. http://jakarta.apache.org/bcel/index.html

[3] Eclipse. http://www.eclipse.org

[4] Evgeniy Gabrilovich, Lev Finkelstein, "JNI - C++ integration made easy", C/C++ Users Journal, Vol.19, Issue 1. 10-21 CMP Media, Inc. 2001.

[5] HawtJNI, http://fusesource.org/forge/projects/HAWTJNI

[6] GlueGen, https//gluegen.dev.java.net

[7] Java Native Interface Specification. http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html

[8] JNA, http://jna.dev.java.net

[9] Ken Arnold, James Gosling, and David Holmes, *The Java Programming Language* Fourth Edition, Addition Wesley, 2005.

[10] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification* Second Edition, Addison Wesley, 1999.

저 자 소 개

김 상 훈(Sang-Hoon Kim) 정회원

- 1989년 2월 : 동국대학교 컴퓨터공학과(공학석사)
- 1996년 8월 : 동국대학교 컴퓨터공학과(공학박사)
- 1997년 3월 ~ 현재 : 세명대학교 컴퓨터학부 부교수

<관심분야> : 프로그래밍언어, 컴파일러, 소프트웨어공학