JPP(JNI 전처리기)의 설계 및 구현

이 창 환† · 오 세 만††

요 약

JNI는 자바가 플랫폼 의존적인 특수한 기능을 이용할 수 있게 하고, 기존의 라이브러리와 프로그램을 재사용할 수 있게 하기 위해 C/C++와 같은 다른 프로그래밍 언어와 연결하는 방법이다. 그러나 JNI를 사용하기 위해서는 복잡하고 어려운 과정을 거쳐야 하며 자바 소스와 C/C++ 소스를 따로 다루어 불편하였다. 본 논문에서는 JNI를 쉽게 사용하기 위해서 동일한 파일에 자바 소스와 C/C++ 소스를 함께 작성할 수 있게 하고 JNI 사용에 필요한 단계들을 줄인 JPP(Java PreProcessor)를 설계하고 구현하였다.

Design and Implementation of JPP(JNI PreProcessor)

Changhwan Yi† · Seman Oh††

ABSTRACT

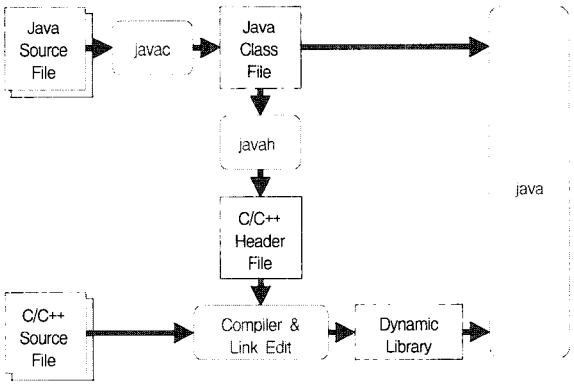
JNI is a linkage method to other languages such as C/C++ which enables the Java to do the platform-dependent specific tasks and also, it can be used to reuse the existing libraries and programs. However, the complex and difficult steps are required to use JNI and it is inconvenient to manipulate Java source and C/C++ source separately. We design and implement the JPP (Java PreProcessor) that enables the Java source and C/C++ source to handle in a same source file and reduces the required steps so as to use JNI easily.

키워드 : 자바(Java), JNI, 전처리기(Preprocessor), JPP, JNI 전처리기(JNI Preprocessor)

1. 서 론

자바 프로그램은 자바가상기계(JVM : Java Virtual Machine)에서 실행되기 때문에 플랫폼 독립적으로 실행될 수 있는 장점을 갖는다. 그러나 다양한 플랫폼에서 실행이 가능하도록 JVM이 구현되었기 때문에 여러 플랫폼이 가진 기능 중 공통적인 기능만을 가지게 되었다. 그래서 플랫폼에 의존적인 부분은 자바로 직접 구현하지 못하는 문제가 발생하였다. 더욱이, 자바가 공개되기 이전에 작성된 많은 라이브러리와 프로그램을 자바에서 직접 사용할 수 있는 방법을 요구하게 되었다. 이에 부응하기 위해 선(Sun Microsystems)사에서는 JNI(Java Native Interface)라는 방법을 JDK 1.1부터 제공하고 있다[2]. JNI는 C/C++와 연결하는 방법으로 자바가 플랫폼에 의존적인 특수한 기능을 사용할 수 있도록 하기 때문에 프로그램의 성능을 높일 수 있다. 또한 기존에 개발된 많은 라이브러리들을 재사용할 수도 있다.

그러나 JNI를 사용하기 위해서는 (그림 1)과 같이 사용자가 javac과 javah, compiler & link edit를 직접 실행하는 복잡한 과정을 거쳐야 한다[3, 4]. 또한 (그림 2)의 예제 코드에서 보듯이 개발자가 자바 소스코드와 C/C++ 소스코드를 별도의 파일로 작성하기 때문에 관리하기가 불편한 단점이 있다.



(그림 1) JNI 사용과정

† 준 회 원 : 동국대학교 대학원 컴퓨터공학과
†† 종신회원 : 동국대학교 컴퓨터공학과 교수
논문접수 : 2001년 11월 27일, 심사완료 : 2001년 12월 18일

```
<AClassWithNativeMethod.java>
public class AClassWithNativeMethod {
    public native void theNativeMethod() ;
    public void aJavaMethod() {
        theNativeMethod() ;
    }
}

<AClassWithNativeMethods.c>
# include <stdio.h>
# include "AClassWithNativeMethods.h"
JNIEXPORT void JNICALL
Java_AClassWithNativeMethods_theNativeMethod(
    JNIEnv* env, jobject thisObj) {
    printf("Hello JNI World\n") ;
}
```

(그림 2) JNI 예제 코드

이러한 단점을 해결하기 위해서, 본 논문에서는 JPP(Java PreProcessor)를 설계하고 구현하였다. JPP는 자바 소스코드와 C/C++ 코드를 동일한 파일에 작성할 수 있게 하고 JNI를 사용하기 위해 단계별로 처리하던 작업을 모두 제거하여 프로그래머가 편리하고 쉽게 관리할 수 있도록 해주는 자바의 전처리기이다.

2. 배경 연구

2.1 JNI(Java Native Interface)

JNI는 자바와 C/C++를 연결하는 방법을 정의한 것이다. 자바와 C/C++를 연결하려면 자바에서 C/C++와의 연결 지점(Entry Point)을 선언할 수 있어야 하고, C/C++에서는 실행할 내용을 이 지점에 기술할 수 있어야 한다.

자바에서 C/C++와의 연결 지점을 선언하기 위해서는 네이티브 메소드(native method)라 불리는 특별한 메소드를 사용한다. 이 메소드는 (그림 2)처럼 **native** 키워드를 수정자(Modifier)에 추가하여 선언한다. 자바에서 메소드를 네이티브 메소드로 선언을 하면, 자바 컴파일러는 해당 메소드에 대한 정의가 없어도 오류를 발생시키지 않고 클래스 파일의 메소드 정보에 네이티브 메소드라는 정보만을 추가한다. 이 정보는 (그림 1)에서 보이는 것처럼 javah가 JNI를 위한 C/C++ 헤더 파일을 생성할 때와 java 실행 환경에서 네이티브 메소드를 호출할 때 사용된다.

네이티브 메소드에서 실행할 C/C++ 코드를 본 논문에서는 네이티브 코드라 부를 것이다. 또한 본 논문에서는 네이티브라는 용어를 자바가 실행이 되는 환경을 나타낼 때 사용할 것이다. 이런 네이티브 코드를 네이티브 메소드와 연결하기 위해서는 javah가 생성한 원형을 가지는 함수에 코드를 기술해야 한다. 또한 네이티브 코드에서 자바 환경에 있는 객체에 대한 연산이 필요한 경우에는 JNI에서 정의한 방법에 따라 연산을 수행해야 한다.

2.2 JNI C/C++ 헤더파일

자바 메소드와 C/C++ 함수를 연결하기 위해서는 정해진 규칙에 따라 자료형, 함수 원형등이 선언되어야 한다. 이런 규칙에 따라 자바의 네이티브 메소드에 맞는 C/C++ 함수 원형을 만드는 일을 하는 것이 (그림 1)에서의 javah이고, 생성되는 파일이 JNI C/C++ 헤더 파일이다. 이 C/C++ 헤더 파일은 javah에 클래스 파일을 입력으로 사용해서 만들어지기 때문에, 클래스와 같은 이름을 가진다. 이 헤더 파일의 구조를 설명한 공식 문서는 없으나, (그림 3)과 같은 예를 통해서 알아볼 수 있다.

```
<AClassWithNativeMethods.h>
/* DO NOT EDIT THIS FILE - it is machine generated */
# include <jni.h>
/* Header for class AClassWithNativeMethods */

# ifndef _Included_AClassWithNativeMethods
# define _Included_AClassWithNativeMethods
# ifdef __cplusplus
extern "C" {
# endif
/*
 * Class :      AClassWithNativeMethods
 * Method :     theNativeMethod
 * Signature :  ()V
 */
JNIEXPORT void JNICALL Java_AClassWithNativeMethods_the
                    NativeMethod
  (JNIEnv *, jobject) ;

# ifdef __cplusplus
}
# endif
# endif
```

(그림 3) javah가 생성하는 헤더파일 예

헤더 파일은 공통적인 부분과 클래스와 클래스의 메소드에 따라 달라지는 부분으로 구성되어 있다. 공통적인 부분은 jni.h를 포함하고, C++에서 C에서 정의된 함수를 사용하기 위한 extern "C" { … } 문장이 정의되어 있다. 클래스나 클래스의 메소드에 따라 달라지는 부분은 파일 이름이나 3번 줄의 주석에 있는 클래스 이름, # ifndef 문장의 조건식 상수와 같이 클래스 이름이 들어가는 곳이다. (그림 3)의 예에서 "AClassWithNativeMethods" 이름이 들어가는 곳이 이에 해당한다. 그리고 클래스에서 정의된 네이티브 메소드의 수만큼의 C/C++ 함수에 대한 주석과 원형이 있다.

함수 원형에 대해서는 2.4에서 설명할 것이고, 주석을 살펴보면 구조는 클래스 이름, 메소드 이름, 자바형 시그네춰

(signature)로 구성되어 있다. <표 1>은 이와 같은 시그네쳐를 정리한 것이다.

〈표 1〉 자바형 시그네쳐

형 시그네쳐	자바형
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
V	void
L*fully-qualified-class* ;	fully-qualified-class
[*type*	type[]
(*arg-types*)*ret-type*	method type

2.3 자바형과 JNI 네이티브 형

네이티브 메소드를 C/C++ 함수와 연결하기 위해서는 다음절에서 설명할 이름만이 아니라, 자바형과 네이티브 형간의 변환도 필요하다. 형 변환에 관한 규칙을 정리하면 <표 2>와 <표 3>과 같다. <표 2>는 자바 기본형과 네이티브 형의 변환 규칙을 정리한 것이고, <표 3>은 자바 참조형을 정리한 것이다[2, 5, 6].

〈표 2〉 자바 기본형과 네이티브 형간 변환 규칙

자바형	네이티브형
byte	jbyte
short	jshort
int	jint
long	jlong
float	jfloat
double	jdouble
char	jchar
boolean	jboolean
void	void

〈표 3〉 자바 참조형과 네이티브형간 변환 규칙

자바형	네이티브형
Object (all Java objects)	jobject
Class (java.lang.Class objects)	jclass
String (java.lang.String objects)	jstring
Generic array (arrays)	jarray
Object[] (object arrays)	jobjectarray
boolean[] (boolean arrays)	jbooleanarray
byte[] (byte arrays)	jbytearray
char[] (char arrays)	jchararray
short[] (short arrays)	jshortarray
int[] (int arrays)	jintarray
long[] (long arrays)	jlongarray
float[] (float arrays)	jfloatarray
double[] (double arrays)	jdoublearray
Throwable (java.lang.Throwable objects)	jthrowable

<표 3>의 jarray는 개념상으로만 사용되는 형이다. 실제로 변환이 되는 형은 j<*type*>array와 같은 형태를 가지고 있다.

2.4 JNI 함수 원형

C/C++ 함수 원형은 자바 네이티브 메소드의 정보를 통해 만들어진다. 함수 원형을 만드는 것은 C/C++ 헤더 파일과 C/C++ 소스파일에서 함수 원형이 사용된다. 그러므로 헤더 파일과 소스 파일을 만드는데 중요한 부분이다. C/C++ 소스 파일의 경우에는 인자 부분에 형 정보만이 아니라, 인자의 이름도 들어가는 것만이 헤더 파일과 다르다. 그러므로 헤더 파일을 위한 함수 원형을 만드는 것에 인자에 대한 이름을 추가만 하면 C/C++ 소스 파일에서 사용하는 네이티브 C/C++ 함수 원형을 정의 할 수 있다. 이런 C/C++ 함수 원형은 다음과 같은 형식으로 구성이 되어 있다.

```
JNIEXPORT <return_type> JNICALL
Java_<class_name>_<method_name>
(JNIEnv *, <jobject or jclass>, <arg_type_list>)
```

(그림 4) 네이티브 메소드의 C/C++ 함수 원형 형식

위에서 <return_type>과 <arg_type_list>에서는 자바의 형을 규칙에 따라 네이티브 형으로 변환을 한다. <jobject or jclass>라고 되어 있는 부분은 네이티브 메소드가 인스턴스 메소드이면 jobject가 사용되고, 클래스 메소드이면 jclass가 사용된다.

만약 네이티브 메소드가 오버로딩되어 있다면, 메소드 이름은 맹글링(mangling)된 이름이 사용된다. 이 맹글링은 인자의 자바 시그네쳐가 사용되고, 메소드 이름과 시그네쳐 사이에는 "__"가 추가된다.

3. JPP (JNI PreProcessor)

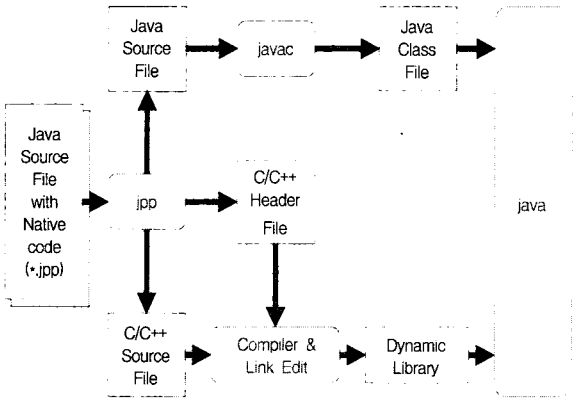
3.1개요

JPP는 JNI의 복잡한 사용 과정을 단순화하고, JNI를 사용할 때 필요한 여러 파일을 하나의 파일로 통합하여 소스 파일 관리의 편의성 제공을 위한 목적으로 설계되고 구현이 되었다.

JNI의 복잡한 사용 과정의 단순화한 JPP의 사용과정은 (그림 5)과 같다. (그림 1)의 JNI의 경우에는 자바 클래스 파일을 생성한 후, *javah* 프로그램을 실행하여 C/C++헤더 파일을 생성하고 있다. 새로 추가된 이 과정이 클래스 파일을 생성과 클래스 파일 실행 사이에 이루어지기 때문에 사용자는 JNI 사용과정을 복잡하게 느끼게 된다. 또한 네이티브 메소드의 실행 내용을 정의한 C/C++파일을 컴파일하기

위해서는 위의 과정을 통해 C헤더 파일을 생성하는 작업이 필요하다.

그러나 JPP의 경우에는 한 번의 JPP 실행을 통해 JNI를 사용하는데 필요한 모든 파일을 생성하기 때문에 사용과정이 단순화되었을 뿐 아니라 편리하게 유지할 수 있게 된다.



(그림 5) JPP 사용과정

소스 파일 관리의 편의성은 (그림 5)의 JPP 입력 파일의 예와 같이 자바 코드와 C/C++ 코드를 한 파일에 같이 기술을 하여 얻을 수 있다. 한 파일에 기술하기 때문에 여러 파일을 관리할 때 생기는 문제를 줄일 수 있다. 또한 네이티브 메소드의 시그네춰(Signature)나 복귀형(Return Type)이 변하는 경우에도 JPP의 실행만으로 변경된 내용을 갖는 자바 소스파일과 C/C++ 헤더파일, 소스 파일을 생성할 수 있다. 이전 방법으로는 변경된 내용이 자바 소스파일에만 있기 때문에, C/C++ 헤더 파일과 소스 파일에 변경된 내용이 적용되기 위해서는 자바 소스파일을 컴파일하고 C/C++ 헤더 파일을 javah를 통해서 생성하고, C 소스파일은 사용자가 직접 수정을 해야만 한다.

```
<AClassWithNativeMethod.jpp>
public class AClassWithNativeMethod {
    native {
        # include <stdio.h>
    }

    public native void theNativeMethod() {
        printf("Hello JNI World\n") ;
    }

    public void aJavaMethod() {
        theNativeMethod() ;
    }
}
```

(그림 6) JPP 입력 파일 예

이런 기능을 가진 JPP를 사용하기 위해서는 (그림 6)과 같은 JPP 입력 파일을 사용자가 만들어야 한다. 이 파일은 *.jpp 확장자를 가지고 있고, C/C++ 코드를 포함하고 있는 자바 소스 파일이다. JPP는 앞에서 말한 바와 같이 이 *.jpp 파일에서 자바 소스파일과 C/C++ 헤더 파일과 소스파일을 생성하고, 생성된 파일을 사용자가 직접 javac와 C/C++ 컴파일러로 컴파일하여 자바 환경에서 실행할 수 있는 파일을 생성한다.

자바 소스 코드에 네이티브 코드를 같이 기술하도록 하기 위해 JPP에서는 자바의 *native* 키워드의 의미를 확장하였다. 자바 언어명세(Language Specification)에 있는 native 키워드의 의미는 자바 컴파일러에 메소드가 네이티브 메소드이라는 것을 알려주는 역할만을 한다[1]. 그러나 JPP에서는 native 키워드가 C/C++ 코드 영역을 JPP에게 알려주는 역할을 하며, 네이티브 코드 문장 영역을 선언하고 네이티브 코드를 가지는 네이티브 메소드를 정의 할 수도 있도록 한다.

```
public class NativeKeywordExam {
    native {
    /* C/C++ 코드 기술 */
    }

    native NativeMethod() {
    /* C/C++ 코드 기술 */
    }
}
```

(그림 7) 네이티브 문장과 네이티브 메소드 정의 방법

(그림 7)의 예제 코드에서 처음 것은 네이티브 문장영역을 정의한 것이고, 다음 것은 네이티브 메소드를 정의한 것이다. 이를 간략한 EBNF로 표시하면 (그림 8)과 같다[7]. <CStatement>는 C 언어 문법으로 정의된 문장을 나타내는 것이다. 메소드 선언에서 <Block>이 오면 자바의 일반 메소드이고, <CBlock>이 오면 C/C++ 코드를 가지고 있는 네이티브 메소드를 나타낸다.

```
<CBlock> ::= '{' <CStatement> '}'
<NativeStatement> ::= 'native' <CBlock>
<MethodDeclaration> ::= <MethodHeader>(<Block> | <CBlock>)
```

(그림 8) JPP에서 사용하는 자바 언어 EBNF

정리하면 JPP는 네이티브 코드를 가진 자바 소스 파일(*.jpp)에서 자바 소스 코드(*.java)와 네이티브 코드를 가진 C/C++ 소스 파일(*.c)을 생성하고, 네이티브 라이브러리를 만드는데 필요한 C 헤더 파일(*.h)을 생성하는 일을 한다.

3.2 구현 및 실행 환경

JPP의 구현 환경은 다음과 같다.

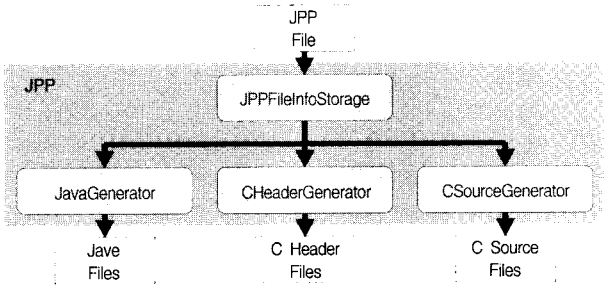
- 사용언어 : 자바
- 컴파일러 : Java2 SDK Standard Edition 1.3
- 운영체제 : Microsoft Windows 2000

그리고 본 논문에서 소개할 JPP의 사용 예제에서 사용하는 C/C++ 소스 파일은 Visual C++ 6.0을 사용하였다.

JPP는 자바의 기본적인 기능만을 사용하기 때문에 어떤 자바 환경에서도 실행 할 수 있지만, JPP가 생성한 파일을 컴파일하고 실행하기 위해서는 JNI가 지원이 되는 자바 환경이 필요하다.

3.3 구조 및 구성 클래스

(그림 9)는 JPP의 구조를 나타낸다. 입력으로 JPP 파일을 받아 JPPFileInfoStorage에 수집된 정보를 저장하고, 저장된 정보는 자바 소스 파일 생성기, C/C++ 헤더파일 생성기, C/C++ 소스파일 생성기에서는 자바 소스파일, C/C++ 헤더파일, C/C++ 소스파일을 생성하는데 사용한다.



(그림 9) JPP 구조도

이 JPP는 다음과 같은 자바 클래스 구성되어 있다. 이 중 InfoStorage는 JPPInfoStorage의 슈퍼클래스, Info는 ClassInfo, FieldInfo, MethodInfo의 슈퍼클래스, Generator는 JavaGenerator, CHeaderGenerator, CSourceGenerator의 슈퍼클래스이다.

- JPP class
- JPPOption class
- InfoStorage classes
  - JPPFileInfoStorage class
- Info classes
  - ClassInfo class
  - FieldInfo class
  - MethodInfo class
  - CodeInfo class
- Generator classes
  - JavaGenerator class
  - CHeaderGenerator class
  - CSourceGenerator class

각 클래스에 대한 설명은 다음절에서 하도록 한다.

3.4 JPP 메인

JPP 메인은 JPP 클래스의 main() 메소드에 주요 내용을 구현하고 있고, JPP의 옵션을 처리해서 JPPOption 객체를 만드는 일과 전체 프로그램 흐름을 제어한다.

JPPOption 클래스는 사용자가 준 옵션 정보를 저장하는데 사용하는 클래스이고, 추후 JPP의 기능 확장을 쉽게 하기 위해서 만든 클래스이다. JPPOption에 저장된 정보는 JPP가 JPP 입력 파일을 처리하는 방식이나 생성되는 자바 소스 파일, C 헤더 파일, C 소스 파일의 내용을 변경하기 위해서 사용될 것이다.

3.5 JPP 파일 정보 저장소

JPP 파일 정보 저장소는 JPP 파일(*.jpp)안의 정보를 읽어 저장하고, 생성기들에서 파일 생성에 필요한 정보를 제공하는 일을 한다. 저장소에는 클래스에 대한 정보와 자바 코드와 C/C++ 코드에 대한 정보가 저장된다. 이 정보를 찾기 위해서 정보 저장소는 클래스 정보를 수집하고 자바 코드와 C/C++ 코드를 분리하는 파서를 사용한다[8, 9].

클래스 정보 수집하는 파서를 살펴보면, 파서에 의해서 수집되는 클래스에 대한 정보는 클래스 자체의 정보, 클래스내의 필드와 메소드에 대한 정보로 나누어진다. 클래스에 대한 정보는 클래스의 이름, 수정자에 대한 정보이고, 필드에 대한 정보는 필드의 이름, 형, 수정자에 대한 정보이다. 메소드에 대한 정보는 메소드의 이름, 복귀형, 인자형과 인자 이름, 수정자에 대한 정보이다. 이 정보를 수집하기 위한 파서는 일반적인 자바 파서가 문장 블록까지 전부 파싱에 하는데 비해, 문장 블록을 무시하고 파싱을 한다. 그리고 파서에 입력으로 들어오는 문자는 자바 코드와 C/C++ 코드를 가지는 자바 소스 파일과 C/C++ 소스파일을 만들기 위해 자바 코드 영역과 C/C++ 코드 영역을 보관하기 위한 곳에 따로 저장을 한다. C/C++ 코드가 네이티브 메소드의 문장 영역에 있는 경우에는 클래스 정보 중에서 관련 메소드 정보를 찾을 수 있는 참조(Reference)를 같이 저장한다. 파서에 들어오는 입력이 자바 코드인지 C/C++ 코드인지를 구분하는 일은 파서가 **native** 키워드를 인식한 이후의 처음으로 나타나는 문장 블록은 C/C++ 코드로 생각하고 C/C++ 코드로 저장을 했다.

이 저장소는 InfoStorage 클래스와 이를 상속받은 JPPFileInfoStorage 클래스로 구현되었다. JPPFileInfoStorage 클래스는 앞에서 설명한 클래스와 코드에 대한 정보만이 아니라, 파일 생성에 필요한 자바 소스 파일 이름과 같은 기타 정보도 같이 가지고 있다. InfoStorage에 저장된 정보는 Info 계열 클래스를 통해서 생성기에 전달이 된다. 이 클래스에는 ClassInfo 클래스와 FieldInfo 클래스, MethodInfo 클래스, CodeInfo 클래스가 있다.

3.6 자바 소스파일 생성기

자바 소스파일 생성기는 JPP 파일 정보 저장소에 저장된 자바 코드 정보를 사용하여 파일을 생성한다. 이 때 생성하는 자바 소스 파일 이름은 JPP 입력 파일 이름과 같은 이름을 사용하고, 이 정보도 JPP 파일 정보 저장소에서 가져온다.

자바 소스파일 생성기와 다음에 설명할 C/C++ 헤더파일 생성기, C/C++ 소스파일 생성기는 Generator 클래스를 상속받아 구현이 되었다. 이 중 자바 소스파일 생성기는 JavaGenerator 클래스로 구현되었다. JavaGenerator는 JPPFileInfoStorage에서 CodeInfo 클래스를 통해 자바 코드 정보를 얻어오고, 자바 코드인지를 나타내는 CodeInfo 클래스의 필드를 사용해서 자바 코드를 가지고 있는 자바 소스파일을 생성한다.

3.7 C/C++ 헤더파일 생성기

C/C++ 헤더파일 생성기는 JPP 파일 정보 저장소에 저장된 클래스와 메소드 정보를 사용하여 파일을 생성한다. 이 때 생성하는 C/C++ 헤더파일 이름은 클래스 이름과 같은 이름을 사용한다. 생성기는 클래스 이름을 사용해서 헤더파일 머리 부분을 만든다. 그리고 클래스에 있는 네이티브 메소드 수만큼 네이티브 메소드의 C/C++ 함수 원형을 만들고, 헤더 파일의 꼬리 부분을 만든다.

이런 C/C++ 헤더파일 생성기는 CHeaderGenerator 클래스로 구현되고, JPPFileInfoStorage에서 ClassInfo 클래스와 MethodInfo 클래스를 통해서 네이티브 메소드에 대한 정보를 가져와 C/C++ 함수 원형을 만들고, 메소드 중 네이티브 메소드인지 구분은 MethodInfo 클래스의 필드를 통해서 이루어진다.

3.8 C/C++ 소스파일 생성기

C/C++ 소스파일 생성기는 JPP 파일 정보 저장소에 저장된 클래스와 메소드 정보, C/C++ 코드 정보를 사용하여 파일을 생성한다. 이 때 생성하는 C/C++ 소스파일 이름은 클래스 이름과 같은 이름을 사용한다. 생성기는 클래스내에 포함된 코드 정보를 사용해서 파일을 생성한다. 이때 코드 정보가 네이티브 메소드의 일부라는 것을 나타내면, 관련된 네이티브 메소드 정보를 사용해서 함수의 원형을 만들고 코드를 함수 안에 포함시킨다.

이런 C/C++ 소스파일 생성기는 CSourceGenerator 클래스로 구현되고, JPPFileInfoStorage에서 ClassInfo 클래스와 CodeInfo 클래스를 통해서 파일을 만든다. 이때 CodeInfo 필드를 사용해서 코드가 메소드에 대한 코드인지 검사하여, 메소드 코드라면 MethodInfo 클래스를 통해서 네이티브 메소드에 대한 C/C++ 함수를 만든다.

4. 실행 결과

본 논문에서는 JPP의 사용 예로 자바 프로그램이 아닌 일반 프로그램을 실행할 수 있는 자바 클래스를 구현하였다. 이 클래스는 실행할 프로그램의 파일 위치와 실행 파일에 대한 인자를 받아 프로그램을 실행을 한다. 클래스의 구조는 (그림 10)와 같다.

```
public class NativeExecute {
    public static native void execute(String path, String[] args) ;
}
```

(그림 10) NativeExecute 클래스

이 클래스의 execute 메소드는 네이티브 메소드이고, 각 플랫폼에서 제공하는 프로그램을 실행시킬 수 있는 함수를 사용해서 구현을 하였다. 사용 예의 구현 및 작동 플랫폼은 윈도이고, 프로그램 실행을 위해 WinExec() 함수를 사용하였다. (그림 11)은 이와 같은 기능을 구현한 JPP 입력 파일이다.

```
<NativeExecute.jpp>
public class NativeExecute {
    static {
      System.loadLibrary("NativeExecute.dll") ;
    }

    native {
      #include <windows.h>
      #include <stdlib.h>
      #include "NativeExecute.h"

      #define MAX_CMD_LEN                    1024
    }

    public static native void execute(String path, String[ ] args) {
      TCHAR cmd[MAX_CMD_LEN] ;
      int i ;

      /* Copy path, args to cmd */

      WinExec(cmd, SW_SHOWNORAML) ;
    }
}
```

(그림 11) JPP 입력 파일

NativeExecute.jpp를 작성한 후, JPP에 입력으로 넣으면 JPP는 NativeExecute.java 자바 소스파일과 NativeExecute.h 헤더파일, NativeExecute.c 파일을 생성한다. 생성된 각 파일은 (그림 12)과 같다.

```
<NativeExecute.java>
public class NativeExecute {
 static {
   System.loadLibrary("NativeExecute.dll") ;
 }
```

```
    pubic static native void execute(String path, String[ ] args) ;
}


<NativeExecute.h>
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeExecute */

#ifndef _Included_NativeExecute
#define _Included_NativeExecute
#ifdef _cplusplus
extern "C" {
#endif
/*
 * Class :      NativeExecute
 * Method :     execute
 * Signature :  (Ljava/lang/String ; [Ljava/lang/String ; )V
 */
JNIEXPORT void JNICALL Java_NativeExecute_execute
  (JNIEnv *, jclass, jstring, jobjectArray) ;

#ifdef __cplusplus
}
#endif
#endif

<NativeExecute.c>
#include <windows.h>
#include <stdlib.h>
#include "NativeExecute.h"

#define MAX_CMD_LEN                    1024

JNIEXPORT void JNICALL Java_NativeExecute_execute
  (JNIEnv * env, jclass jcls, jstring path, jobjectArray args) {
     TCHAR cmd[MAX_CMD_LEN] ;
     int i ;

     /* Copy path, args to cmd */

    WinExec(cmd, SW_SHOWNORAML) ;
  }
}
```

(그림 12) JPP 실행 결과파일

5. 결론 및 향후 연구

JNI는 자바 프로그램과 C/C++를 연결하기 위해서 제공된 자바의 기능이다. JNI를 사용하여 자바는 플랫폼에 의존적인 기능을 사용할 수 있으며 C/C++로 만들어진 많은 라이브러리를 사용하고 기존 시스템과 연결할 수도 있다. 그러나 JNI를 사용하기 위해서는 복잡한 사용 과정과 여러 자바 소스파일과 C/C++ 파일을 관리해야 하는 단점이 있다.

이런 단점을 해결하기 위해 본 논문에서는 자바 코드와 C/C++ 코드를 같이 한 파일에 정의하고, 이 파일에서 JNI를 사용하는데 필요한 자바 소스 파일과 C/C++ 헤더파일, C/C++ 소스 파일을 생성하는 JPP를 설계하고 구현하였다. JPP를 사용하면, 한 파일에 JNI를 사용하는데 필요한 자바 코드와 네이티브 코드를 함께 프로그래밍하기 때문에 자바 소스 파일과 C/C++ 파일을 따로 관리해야 하는 단점을 없애고, 복잡한 사용 과정을 단순화할 수 있다. 또한, JPP를 사용하는 예제로 자바환경에서 플랫폼에 있는 프로그램을 실행 할 수 있는 클래스 라이브러리를 설계하고 구현하였다.

향후 연구 과제는 JNI 사용 과정을 더욱 단순화시키는 것과 네이티브 코드에서 자바 객체에 대한 연산을 쉽게 할 수 있는 방법의 연구, 그리고 C전처리기가 가지고 있는 조건 컴파일과 매크로와 같은 기능을 추가하는 것이다[10]. JNI 사용 과정을 더 단순화에 대한 연구는 한 번의 JPP 실행을 통해 자바 클래스 파일과, 네이티브 라이브러리 파일과 같은 JNI 사용에 필요한 최종 파일을 생성하는 기능이나 JNI에서 사용할 때 자바코드나 공통적으로 나타나는 C/C++코드를 자동으로 추가시키는 기능 등을 생각해 볼 수 있다. 다음으로 JNI에서 정의한 자바 객체에 대한 연산을 쉽게 할 수 있는 방법에 관한 연구는 현재 네이티브 코드에서 JNI에서 정의된 복잡한 자바 객체에 대한 연산 코드를 단순화시키는 것이다. 마지막으로 C/C++ 전처리기의 기능인 조건 컴파일, 매크로와 같은 기능을 JPP에 추가하는 것이다.

참 고 문 헌

[1] The Java Language : An Overview, White Paper, Sun Microsystem, 1996.

[2] Java Native Interface, Javasoft, 1997.

[3] Compione, Walrath, Huml, and Tutorial Team, "Java Tutorial continued," Addison-Wesley, 1998.

[4] Rob Gordon, "Essential JNI," Prentice-Hall, 1998.

[5] 이창환, 오세만, "C/C++에서 자바객체 이용을 위한 인터페이스", 정보처리학회 '99 춘계학술발표논문집, 1999.

[6] 이창환, 오세만, "자바객체를 사용할 수 있는 자바스크립트 해석기의 설계 및 구현", 정보과학회 '99 추계학술발표논문집, 1999.

[7] 오세만, 이양선, 김상훈, 고광만, 자바입문 개정판, 생능출판사, 2000.

[8] Elliot Berk, C. Scott Ananian, JLex : A Lexical Analyzer Generator for Java(TM), http : //www.cs.princeton.edu/~appel/modern/java/JLex/.

[9] Scott E. Hudson, CUP Parser Generator for Java, http : //www.cs.princeton.edu/~appel/modern/java/CUP/.

[10] Nik Shaylor, JPP-A preprocessor for the Java language, http : //www.geocities.com/CapeCanaveral/Hangar/4040/jpp.html, 1996.

이 창 환

e-mail : yich@dongguk.edu

1998년 동국대학교 컴퓨터공학과 졸업 (학사)

2000년 동국대학교 컴퓨터공학과 석사 학위 취득

2000년~현재 동국대학교 대학원 컴퓨터 공학과 박사과정

관심분야 : 프로그래밍 언어, 컴파일러, 자동화 도구, 임베디드 시스템 등

오 세 만

e-mail : smoh@dongguk.edu

1977년 서울대학교 사법대학 수학과 졸업 (학사)

1979년 한국과학기술원 대학원 전산학과 석사학위 취득

1985년 한국과학기술원 대학원 전산학과 박사학위 취득

1993년~1999년 동국대학교 컴퓨터공학과 대학원 학과장

1985년~현재 동국대학교 컴퓨터공학과 교수

관심분야 : 프로그래밍 언어, 컴파일러, XML, 모바일 언어