재사용을 고려한 자바 라이브러리 설계 및 구현 방법

최유희[O] 윤석진 양영종
한국전자통신연구원 임베디드 S/W 연구단
{yhchoi[O], sjyoon, yjyang}@etri.re.kr

An Approach to Design and Implementation of Java Library for Reusing

Youhee Choi[O], Seok-Jin Yoon, Young-Jong Yang
ETRI-Embedded S/W Research Division

요 약

현재 임베디드 소프트웨어에 대한 개발 및 연구가 활발히 이루어지고 있다. 이러한 상황에 맞춰 임베디드 시스템을 위한 임베디드 자바 플랫폼의 필요성 또한 증대되고 있다. 그러나 현재 임베디드 시스템에서 일반적으로 사용되는 운영체제인 임베디드 리눅스 상에서 사용될 수 있는 공개 소스 기반의 J2SE 용 자바 플랫폼은 개발되어 있으나 임베디드 시스템의 하드웨어 제약사항 등을 고려한 임베디드 자바 플랫폼인 J2ME용 자바 플랫폼은 개발이 활발히 이루어지고 있지는 않다. 그러나 각 플랫폼이 동일한 가상 머신을 사용할 수 있으므로 J2SE에 J2ME용 라이브러리로만 교체하면 J2ME로 사용될 수 있는 점에 착안하여 본 연구에서는 J2ME용 라이브러리만을 개발하고 기존의 J2SE용 가상 머신과 결합할 수 있는 방법에 대해 제안하고자 한다.

1. 서 론

최근들어 임베디드 소프트웨어의 중요성 및 관심이 증가되고 있다. 임베디드 시스템에 가장 일반적으로 사용되는 운영체제로는 임베디드 리눅스이고 앞으로도 매우 빠른 속도로 보급될 전망이다. 또한 임베디드 시스템의 공통 미들웨어에 대한 요구도 증가되어 이러한 미들웨어로 임베디드 자바 플랫폼의 필요성이 증대되고 있다. 리눅스의 특성상 공개 소스 기반의 프로젝트들이 많이 있으나 현재 J2SE(Java 2 Standard Edition)용 자바 플랫폼은 오픈 소스 등 공개된 자바 플랫폼들이 많이 있으나 공개 소스 기반의 J2ME(Java 2 Micro Edition)[1]용 자바 플랫폼은 Sun사에서 reference implementation으로 제공하는 플랫폼이 유일하다고 할 수 있다. 그러나 J2ME와 J2SE는 하부의 가상 머신(Virtual Machine)과 상부의 라이브러리 계층으로 구성되는 자바 플랫폼의 구조상에서 하부의 가상 머신은 동일한 수준의 가상머신을 사용하고 상부의 라이브러리 계층이 J2ME용은 J2SE용에 비해 라이브러리가 작아지고 각 임베디드 시스템에서 요구하는 라이브러리의 범위에 따라 세 범위의 라이브러리인 FP(Foundation Profile), PBP(Personal Basis Profile), PP(Personal Profile)로 나누어져서 필요한 라이브러리 범위만 사용할 수 있도록 정의되어 있다. 따라서 기존의 J2SE용 가상 머신위에 J2ME용 라이브러리만을 구현하면 기존의 다양한 J2SE용 가상 머신을 사용할 수 있고 J2ME용 자바 플랫폼을 구축하기 위한 개발 시간을 단축할 수 있다. 그러나 현재 대부분의 J2SE용 자바 플랫폼의 경우 가상 머신과 라이브러리의 연결이 자바 네이티브 코드 등으로 인해 여러 클래스들에 흩어져 있는 상태여서 기존의 자바 플랫폼은 이를 분리하기가 어렵다. 또한 일부 자바 라이브러리 클래스들은 가상 머신에 의존적인 자바 클래스를 사용하는 경우도 있어 가상 머신과 라이브러리간의 분리가 어렵다. 따라서 현재의 대부분의 J2SE용 자바 플랫폼의 구조상에서 가상 머신과 라이브러리간의 연결을 최소화하거나 가능하면 연결 부분이 쉽게 파악될 수 있도록 수정되어야 한다. 따라서 본 연구에서는 기존의 다양한 J2SE용 가상 머신에 결합되어 재사용될 수 있도록 하기 위한 자바 라이브러리의 설계 및 구현 방법을 제안하고자 한다.

2. J2ME

J2ME 는 PDA, 팜, 셋탑박스, 핸드폰 등의 모바일 기기와 임베디드 기기 시장을 목표로 하는 자바 플랫폼으로 다양한 임베디드 소프트웨어에 적합한 맞춤형 실행 환경을 제공하기 위한 자바 가상 머신과 다양한 디바이스에 맞추어진 실행환경 API 집합으로 이루어진 Configuration과 Profile로 구성되어 있다. 그림 1은 J2ME 플랫폼을 나타낸다.

J2ME의 주요 요소 중 하나인 Configuration은 비슷한 규모의 소형기기들이 공통으로 가지고 있어야 할 API의 집합을 정의한다. 즉, 각 소형기기의 공통 기능이라고 할 수 있는 것을 정의한 것이다. Configuration을 정의한 목적은 각종 소형기기의 하드웨어(프로세서, 메모리, 네트워킹 등)의 차이와 무관하게 독립적으로 프로그램을 개발할 수 있도록 하기 위해 추상화를 제공하는 것이다.

J2ME의 주요 요소 중 또 다른 하나인 Profile은 특정한 그룹의 소형기기의 특징을 반영한 API의 집합을 말한다.

Configuration이 유사한 규모의 기기 전체에 대한 공통의 기능을 반영하는 것에 비하여 Profile은 특정한 기기 그룹을 지원한다. Profile은 자바의 특징인 이식성을 지원하기 위한 것이다.

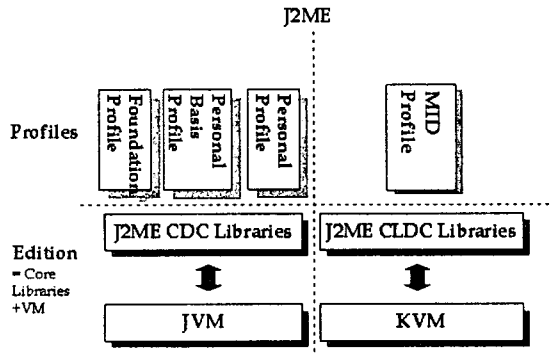

그림 1 J2ME 플랫폼

3. 재사용을 고려한 자바 라이브러리 설계 및 구현 방법

3.1 네이티브 메소드 호출 구조 정의

자바 네이티브 메소드를 호출하는 부분인 가상 머신에 의존적인 부분과 네이티브 메소드를 호출하지 않는 부분인 가상 머신에 독립적인 부분으로 분리하기 위해서 적용할 수 있는 방법은 여러 가지가 있다. 그 중 첫 번째 방법은 네이티브 메소드를 호출하는 부분과 네이티브 메소드를 호출하지 않는 부분을 각각 분리하여 다른 위치에 두는 것으로 네이티브 메소드를 호출하는 부분만 다른 가상 머신에 결합할 때 고려하도록 하는 방법이다. 그러나 이 방법의 경우 자바 네이티브 메소드를 호출하는 부분이 라이브러리 자바 클래스 여러 클래스에 흩어져 있고 이에 따라 네이티브 메소드를 호출하는 부분들만을 모아서 어느 한곳에 두는 방식은 적절하지 않다. 두 번째 방법은 그림 2에서처럼 라이브러리 자바 클래스에서는 네이티브 메소드를 직접적으로 호출하지 않도록 구현하되 네이티브 메소드를 호출하는 대신 임의로 정의한 자바 클래스의 메소드를 호출하게 한다. 그리고 임의의 자바 클래스의 메소드 내부에서 해당 네이티브 메소드를 호출하도록 구현한다. 또한 임의로 정의한 자바 클래스는 각 패키지 및 각 클래스별로 정의하여 구별 가능한 특정한 폴더에 배치하여 가상 머신에 의존적인 부분을 쉽게 파악할 수 있도록 구성한다.

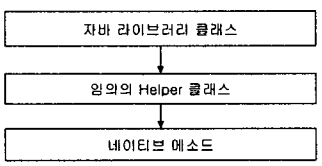

그림 2 네이티브 메소드 호출 구조

3.2 재사용을 고려한 Makefile 정의

임베디드 소프트웨어에서 일반적으로 사용되는 운영체제가 리눅스 환경이므로 make를 통해 컴파일 등의 작업을 하게 된다. 이를 위해 Makefile을 작성해야 될 필요가 있는데 보통의 경우 임의의 변수를 정의하여 이를 configure 과정 및 make 과정을 통해 설정한 뒤 이를 필요로 하는 Makefile에서는 해당 변수를 이용하는 형태로 작성되는 경우가 많다. 그러나 이 경우 해당 변수에 의존관계를 형성하게 되어 해당 Makefile을 해당 변수를 정의하고 있는 configure나 Makefile을 찾아서 파악하여야 하고 이를 사용하기 위해서는 해당 변수의 값을 설정하는 부분이 configure나 Makefile에 포함되어 있어야 한다. 따라서 이러한 의존 관계를 파악하기 쉽게 하기 위해 Makefile을 작성할 때 수정이 필요하다면 해당 Makefile만 수정하면 되도록 해당 Makefile 범위내에서만 사용되는 변수를 사용하거나 이미 정의된 변수만을 사용하여 Makefile이 작성될 수 있도록 한다.

3.3 C 코드간의 의존 관계 파악을 용이하게 하기 위한 구조 정의

임베디드 소프트웨어는 성능 등의 이유로 C 코드를 많이 사용하게 되고 또한 자바 라이브러리는 자바 네이티브 메소드를 호출하는 부분이 있어서 C 코드와 자바 코드가 함께 존재하는 경우에 해당된다. 자바 코드의 경우 일반적으로 패키지 구조로 되어 있어서 각 자바 클래스들간의 의존관계 등을 추적하기 용이하지만 C 코드의 경우 헤더 파일을 포함할 때 헤더 파일의 이름만 포함되고 경로정보는 포함되지 않는다. 또한 자바에서의 클래스 메소드를 호출할 때 해당 메소드가 어떤 클래스의 메소드인지 호출 코드를 보고 쉽게 파악할 수 있는 반면에 C 코드에서의 메소드 호출의 경우 해당 메소드가 어디에 선언 및 구현되어 있는지 메소드 호출 코드만을 봐서는 파악하기가 어렵다. 이러한 C 코드 파악의 어려움이 자바 라이브러리의 재사용을 위해 일부분을 수정해야 될 필요가 있을 때 이를 어렵게 하는 요인 중 하나로 간주될 수 있다. 따라서 이를 방지하기 위해 자바 라이브러리의 구조상에서 C 코드와 관련된 고려가 필요하다. 이를 위해 첫 번째로 헤더나 C 코드에서 메소드를 선언할 때 해당 헤더나 C 파일의 파일명을 메소드명의 앞부분에 포함하도록 정의한다. 이렇게 정의함으로써 다른 C 코드에서 이러한 메소드를 호출할 때 이를 선언 및 구현하고 있는 파일을 쉽게 찾을 수 있도록 할 수 있다. 또한 두 번째로 헤더나 C 파일의 파일명을 알고 있을 때 이를 쉽게 찾을 수 있도록 하기 위해서 특정 패키지에서 사용되는 네이티브 메소드와 관련 있는 C 코드인 경우와 여러 패키지에서 공통적으로 사용되는 C 코드인 경우로 나눈다. 그리고 특정 패키지에서 사용되는 C 코드인 경우, 해당 패키지명으로 구별된 폴더에 두고 해당 C 코드 및 해당 패키지와 관련된 C 코드에서만 사용되는 헤더 파일들을 하위 폴더에 두는 형태로 구조화한다. 그 외의 여러 패키지에서 공통적으로 사용되는 C 코드인 경우, 공통의 하나의 폴더에 두고 해당 C 코드와 관련된 헤더 파일들을 하위 폴더에 두는 형태로 구조화한다. 이를 통해 네이티브 메소드와 관련된 C 코드들을 수

정해야 될 필요가 있을 때 각 코드간의 의존관계를 쉽게 파악할 수 있도록 한다.

4. 적용

본 장에서는 리눅스 상의 J2SE용 자바 라이브러리 공개 소스인 classpath 0.1.7[2] 버전을 예로써 본 연구를 적용한다.

4.1 네이티브 메소드 호출 구조 정의

그림 3은 java.lang 패키지 내의 클래스 중 Math 클래스의 구현 수정을 보여 주기 위한 것이다. 그림 3에서 java.lang.Math 클래스의 구현을 수정하기 전 코드에서는 네이티브 메소드를 직접적으로 선언하고 이를 로컬 메소드에서 호출하는 형태로 구현되어 있다. 이러한 형태로 구성된 경우 네이티브 메소드를 수정하고자 할 경우 네이티브 메소드를 호출하는 부분을 찾아서 모두 수정해 주어야 한다. 따라서 이러한 문제를 해결하기 위해 그림 3과 같이 Math 클래스의 helper 클래스인 NMath 클래스를 정의하고 Math 클래스에서는 네이티브 메소드를 직접적으로 호출하는 대신 NMath 클래스에서 해당 네이티브 메소드를 호출하는 형태로 구현을 변경할 수 있다. 또한 이러한 helper 클래스들은 특정 폴더에 같은 패키지로 구별하여 둠으로써 네이티브 메소드를 변경하고자 할 경우 helper 클래스들만을 찾아서 수정하면 되도록 구성할 수 있다.

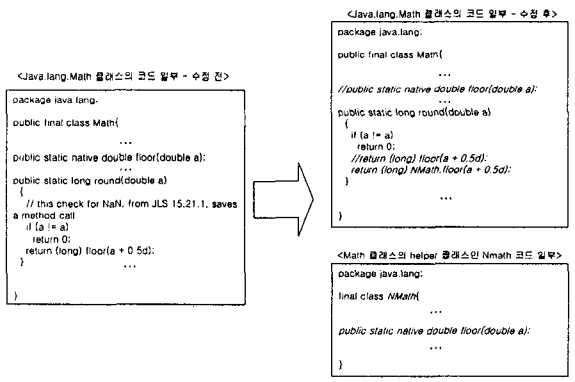

그림 3 java.lang.Math 클래스의 구현 변경 전 및 후

4.2 재사용을 고려한 Makefile 정의

그림 4는 configure나 다른 Makefile 없이 독립적으로 사용될 수 있도록 Makefile을 작성한 예를 나타낸 것이다. 그림 4의 윗부분에서 보듯이 일반적으로 작성되는 Makefile과 configure 파일은 다른 Makefile이나 configure 파일에서 정의된 변수를 사용하는 형태로 작성된다. 이렇게 작성된 경우 해당 변수 설정하는 부분을 찾기 위해 configure나 다른 Makefile을 검색해서 각 변수 정의 부분을 함께 가져 와야 한다. 각 Makefile을 재사용하고자 할 경우 이러한 작업을 여러 번 반복해 주어야 하므로 재사용하기 위해 수정 작업에 많은 시간 및 노력이 필요하다. 따라서 그림 4에서처럼 Makefile을 수정하여 다른 Makefile이나 configure를 확인하지 않고 독립적으로 사용될 수 있도록 구성하는 것이 필요하다.

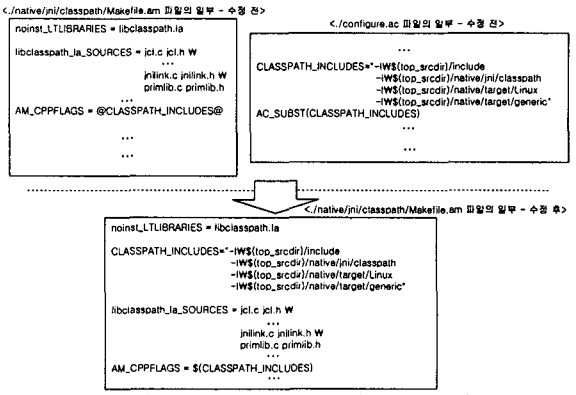

그림 4 Makefile.am 파일의 작성 예

4.3 C 코드간의 의존 관계 파악을 용이하게 하기 위한 구조 정의

기존의 classpath에서는 include 폴더와 native 폴더에 네이티브 코드와 관련된 C 코드 및 헤더 파일이 포함되어 있다. native 폴더에 각 패키지별로 각각의 폴더로 구별되어 네이티브 메소드와 관련된 C 코드 및 일부의 헤더 파일들이 포함되어 있으며 include 폴더아래에 각 패키지에서 사용되는 jni 메소드와 관련된 헤더 파일들이 패키지별로 구별되지 않은 채 포함되어 있다. 또한 native 폴더의 fdlibm 폴더 아래에 각 패키지에 관계 없이 공통으로 사용되는 C파일 및 헤더 파일들이 섞여 있다. 이러한 폴더 구조를 변경하여 fdlibm 폴더에는 헤더 파일을 위한 하위 폴더를 두고 헤더 파일과 C 파일을 구별하여 주고 include 폴더에 있던 각 패키지별 헤더 파일들은 native 폴더 아래의 각 패키지별 폴더 하부에 헤더 파일을 위한 폴더를 두어 관련된 헤더와 C 파일을 쉽게 찾을 수 있도록 구성한다.

5. 결론

본 연구에서는 재사용을 고려한 자바 라이브러리의 설계 및 구현 방법을 제안하였다. 본 연구에서는 여러 다양한 가상 머신에서의 재사용을 고려하여 자바 라이브러리와 가상 머신과의 연결을 쉽게 파악할 수 있도록 구성하고 가상 머신과의 연결 부분의 교체 및 수정을 용이하게 하기 위한 자바 라이브러리의 설계 및 구현 방법을 제안하였다. 향후에는 본 연구를 다양한 가상 머신에 적용하여 재사용성을 검증하는 연구가 필요하다.

6. 참고 문헌

[1] SUN, "Java 2 Platform, Micro Edition (J2ME): JSR 68 overview", http://java.sun.com/j2me/overview.html.
[2] GNU Classpath, http://www.gnu.org/software/classpath.