SimpleRTJ 자바가상기계에서 클래스 파일의 메모리 상 배치

양희재
경성대학교 컴퓨터공학과
hjyang@star.kyungsung.ac.kr

In-Memory Allocation of Class Files in SimpleRTJ Java Virtual Machine

Heejae Yang
Dept. of Computer Engineering, Kyungsung University

요 약

자바가상기계는 클래스 파일에 있는 클래스, 상수, 필드, 메소드 등의 내부 정보를 읽고 자바 응용 프로그램을 실행한다. 보조기억장치가 없는 내장형 시스템의 경우 클래스 파일은 주기억장치, 즉 메모리에 두어야 하지만 클래스 파일 자체는 크기도 클 뿐 아니라 내부 정보에 접근하는 것도 효율적이지 못하다. 따라서 대개의 경우 클래스 파일을 변형한 형태로 메모리에 배치하는데, 본 논문에서는 특히 simpleRTJ 라고 하는 상용 내장형 자바가상기계에서 적용된 방식에 대해 조사해보았다. 이 플랫폼에서의 분석을 통해 클래스 파일의 크기가 얼마까지 줄어들 수 있고, 내부 정보에 대해서는 얼마나 효율적으로 접근할 수 있는지에 대해 고찰하였으며, 그 결과를 바탕으로 향후 더 개선된 형태로 클래스 파일을 메모리에 배치할 수 있는 방안에 대해 연구하고자 한다.

1 서론

한번 프로그램을 작성하면 어느 플랫폼에서나 그 프로그램을 사용할 수 있다는 (Write Once, Run Anywhere) 자바의 플랫폼 독립성은 다양한 하드웨어 및 운영체제를 갖는 내장형 시스템을 위한 좋은 대안이 될 수 있다는 점에서 최근 더욱 관심을 받고 있다 [1].

자바 프로그램이 실행되기 위해서는 해당 플랫폼에 자바가상기계(JVM)가 설치되어야 한다. 모든 자바 프로그램은 JVM 상에서 실행되어지며 내장형 시스템도 예외가 아니다. 썬마이크로시스템사의 J2ME는 KVM (Kilobyte Virtual Machine) 이라는 JVM을 가정하고 있으며, 그외에도 여러 많은 회사들이 내장형 시스템을 위한 JVM을 다투어 개발하여 출시하고 있다.

본 연구의 관심은 클래스 파일의 메모리 상 배치에 있다. 모든 자바 원천코드는 자바 컴파일러에 의해 클래스 파일로 변환되며, JVM은 이 파일을 읽어서 실제 프로그램 실행을 한다. 내장형 시스템의 경우 보조기억장치가 없는 경우가 많으므로 클래스 파일은 주기억장치, 즉 메모리 상에 놓여져야 하는데, 클래스 파일 자체로는 여러 가지 링크 및 디버깅 정보로 인해 메모리의 낭비가 심하며, 또 클래스 파일 내부에 들어있는 상수나 필드, 메소드 등을 접근하는데 비효율적성이 많다 [2].

따라서 클래스 파일은 다른 새로운 형태로 변형되어 실제 메모리에 적재되는데, 우리의 관심은 어떻게 변형하는 것이 가장 적은 메모리를 차지하면서 동시에 내부의 주요 정보들에 대해 빨리 접근하게 할 수 있을 것인가에 대한 것이다. 지금까지 우리는 상용 제품들에서 사용되어지는 클래스 파일의 변형된 포맷에 대해 조사해왔으며, 본 논문에서는 특히 원천 코드가 공개되어져 있는 simpleRTJ 자바가상기계에서 적용된 포맷에 대해 알아본다 [3].

simpleRTJ의 모델은, 원래의 클래스 파일들은 개발자용 호스트 컴퓨터 상에 있으며, 호스트 컴퓨터는 이들 클래스들에 대한 정적인 안전성 검사, 클래스들에 대한 상호간 연결 등의 작업을 거쳐 하나의 이미지 파일을 생성하고, 이것이 내장형 컴퓨터로 다운로드 되거나 또는 ROM 에 탑재되어 JVM에 의해 읽혀져서 실행되게 하는 것이다 (그림 1).

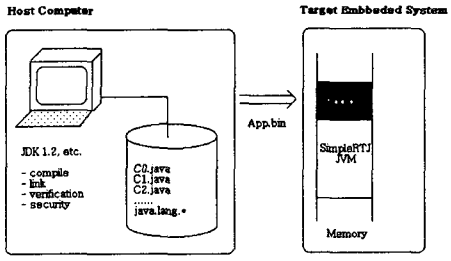

그림 1 simpleRTJ 에서 이미지 파일의 생성 및 배치

이 방법으로 내장형 자바 시스템을 구현하면 JVM은 그대로 둔 채 오직 이미지 파일만 새로 만들어 다운로드 또는 ROM에 탑재하면 되므로 기존 자바 애플리케이션을 수정하거나 새로운 애플리케이션을 적용시키는 것이 쉽게 이루어질 수 있다는 장점이 있다.

본 논문에서는 simpleRTJ 이미지 파일 속에 클래스 파일이 어떻게 변형되어 들어가 있으며, 메모리 사용의 절감 및 클래스 내부 요소에 대한 접근성 등에 대해 알아본다.

이 논문의 구성은 다음과 같다. 2절에서는 simpleRTJ 이미지 파일의 개략적 구조에 대해 알아보고, 3절에서는 이미지 파일 내에서 클래스들이 어떻게 배치되어있는지에 대해 알아본다. 4절에서 이 배치의 효과에 대한 분석과 함께 결론을 맺는다.

2 구조

simpleRTJ에서는 호스트 컴퓨터에 있는 ClassLinker 라는 프로그램이 클래스 파일들을 참조하여 이 클래스들의 변형된 형태의 모임인 이미지 파일, 즉 bin 파일을 만들어 낸다. 본 논문에서는 ClassLinker 1.2.1과 Linear 16MB 메모리 모델을 사

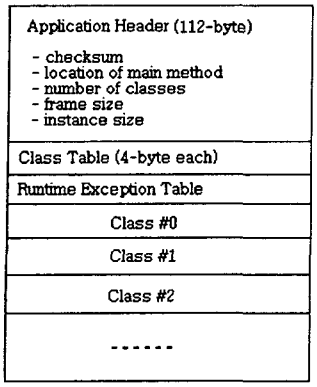

그림 2 simpleRTJ 에서 bin 파일의 구조

용했다.

bin 파일 내에는 다수개의 클래스들이 포함되어 있으며, 그 외에도 클래스의 개수, 프레임 및 인스턴스의 크기, checksum 등의 정보가 들어있는 헤더부분도 들어가 있다 (그림 2).

bin 파일 내에서 가장 핵심적 부분은 각종 클래스들이 있는 곳으로, 일반적인 클래스 파일 내에서의 구조와는 다르다. 다음 절에서 이 구조에 대해 자세히 알아보도록 하자.

3 클래스 파일의 배치

클래스 파일에 포함되어있는 내용 중 실제 실행에 필요한 정보들은 클래스 자체에 대한 정보, 상수에 대한 정보, 필드에 대한 정보, 그리고 메소드에 대한 정보 등 네가지로 나뉜다 [4]. 각 정보들이 신속히 접근될 수 있도록 simpleRTJ에서 어떻게 메모리 상에 배치하였는지에 대해 알아보자.

그림3은 simpleRTJ에서 클래스가 배치된 모습을 나타낸다. 여기서 상수풀 오프셋 테이블(constant pool offset table)은 네 가지 정보를 나타내는데 공통으로 사용된다, 이 테이블은 4바이트 크기를 갖는 포인터값을 엔트리로 갖는 배열구조를 가지며, 각 포인터는 클래스, 상수, 필드, 메소드 구조체 중 어느 하나를 가리키게 된다.

simpleRTJ에서 메모리 절감이 이루어지는 가장 큰 이유 중 하나는 이 테이블에 클래스들간 링크를 위한 각종 심볼명들이

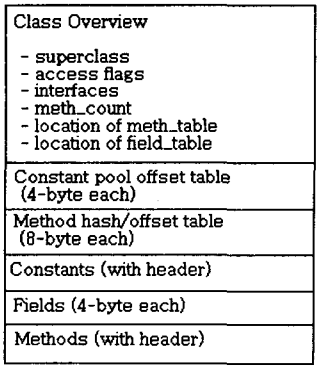

그림 3 클래스의 구조

없다는 것이다. 동적인 클래스 적재를 지원하지 않으므로 필요한 클래스들은 미리 정의되어 호스트 컴퓨터 상에서 모든 링크가 처음부터 이루어질 수 있기 때문이다.

3.1 클래스 정보

클래스 정보는 어떤 클래스의 인스턴스를 만드는 new 명령에 의해 참조된다. 이 명령의 형식은 new #i 와 같으며, i 는 2바이트의 인덱스 값이다. 원래 이 값은 클래스 파일 내의 상수풀을 가리키는 인덱스 값이지만, simpleRTJ 에서는 상수풀 오프셋 테이블의 인덱스값이다.

실제 클래스 정보는 class_T 구조체로 정의된다.

```
struct class_T {
 uint16  flags;   /* class modifier flags */
 uint16  index;
 uint8   *ifaces;
 uint8   *super;  /* pointer to super class */
 uint8   *run;    /* pointer to run() method, if exists */
 uint8   *clinit; /* pointer to <clinit> method */
 uint32  meth_count;
 uint8   *meth_tbl; /* pointer to hash/meth offsets tbl */
 uint8   *field_tbl;      /* pointer to fields table */
};
```

new 명령에 의해 새로운 인스턴스가 만들어질 때는 해당 클래스 정보 뿐 아니라 클래스 계층구조에 따른 모든 상위 클래스의 정보까지 필요하며, super 항목에 의해 상위 클래스를 추적할 수 있게 하고 있다. 상위 클래스 및 인터페이스의 위치는 인덱스 값이 아니라 그것의 실제 주소로서 나타내어지며, 따라서 보다 빠른 접근이 가능하다.

클래스 정보를 얻는 과정은: ① 인덱스값으로 상수풀 오프셋 테이블 내용 읽기; ② 상수풀 오프셋 테이블이 가리키는 번지에서 class_T 구조체를 읽기 등 두 개의 단계를 거친다. 상위 클래스들에 대해서도 동일하게 추적하여 각 단계를 밟는다.

3.2 상수 정보

상수 정보는 어떤 상수값을 스택 상에 적재시키는 ldc 명령에 의해 참조되며, 형식은 ldc #i 와 같다. i는 2바이트의 인덱스 값으로, 원래 이 값은 클래스 파일 내의 상수풀을 가리키는 인덱스 값이지만, simpleRTJ 에서 이 값은 상수풀 오프셋 테이블의 인덱스 값이다.

실제 상수 정보는 const_T 구조체에서 정의된다.

```
struct const_T {
    uint32  type_len;
};
```

이 구조체는 32비트 크기를 갖지만, 실제로는 그 중 하위 16비트만이 이용된다. 이 16비트 중 상위 2비트는 상수의 형식을 나타내며, 나머지 14비트는 길이를 나타낸다. 형식은 세가지가 있으며, int는 1, float는 2, 그리고 문자열은 3이다 (simpleRTJ 에서는 double 및 long 형식은 사용하지 않는다).

이 구조체에 이어 실제값이 놓인다. 예를들어 정수값 50,000 인 경우 상수 정보는 16진수로 00004004 0000C350 와 같이 되며, 문자열 "Hello" 인 경우에는 이 문자열의 16비트 해쉬값인 0C53 이 포함되어 0000C005 0C53 48656C6C6F 가 된다.

하나의 상수를 접근하기 위해, ① 인덱스값으로 상수풀 오프셋 테이블 내용 읽기; ② 상수풀 오프셋 테이블이 가리키는 번지에서 const_T 구조체를 읽기; ③ 형식과 길이를 알고 실제 데이터 읽기 등 세 개의 단계를 거친다.

3.3 필드 정보

필드 정보는 특정 필드값을 접근하여 사용하는 getfield/putfield 등의 명령에 의해 참조되며, 형식은 getfield #i 등과 같다. i는 2바이트의 인덱스 값으로 원래 이 값은 클래스 파일 내의 상수풀을 가리키는 인덱스 값이며, simpleRTJ 에서는 상수풀 오프셋 테이블의 인덱스 값이다.

실제 필드 정보는 field_T 구조체에서 정의된다.

```
struct field_T {
    uint32 type_index;
};
```

이 구조체는 32비트 크기를 갖지만, 실제로는 그 중 하위 16비트만이 이용된다. 이 16비트 중 상위 4비트는 필드의 형식을 나타내는데, 종류는 byte, char, short, int, float, boolean, class, array 가 있고 각각의 값은 0부터 7까지에 해당된다.

나머지 12비트는 그 필드가 인스턴스 변수 배열의 몇번째에 해당하는지를 나타내는 인덱스 값이다. simpleRTJ 에서는 어떤 클래스의 새로운 인스턴스가 만들어지면 그 인스턴스를 위해 인스턴스 변수, 즉 가상 필드들을 저장할 수 있는 공간을 배열 형식으로 만든다. 각 배열 엔트리는 32비트의 크기를 가진다. 이 배열은 그 클래스가 갖는 필드 뿐 아니라 클래스 계층 상에 놓인 모든 상위 클래스들의 필드들을 수용할 수 있는 크기를 가진다. 가장 상위 클래스가 갖는 필드들에게 인덱스 번호 0, 1, 2, ..., n-1 까지가 할당되고, 그 다음 하위 클래스의 필드에게는 n, n+1, n+2, ..., m-1 과 같이 할당된다 ($n \leq m$).

필드 접근에 따르는 시간에 대해 알아보면, ① 먼저 상수풀 오프셋 테이블을 참조하여 특정 필드의 위치를 안다; ② 그 위치에 놓인 필드 정보, 즉 field_T 구조체를 읽어서 그 필드의 형식 및 인스턴스 변수 배열상에서의 인덱스 값을 안다; ③ 끝으로 해당 인스턴스 변수 배열값에 대해 접근하게 된다. 즉 getfield #i 등의 명령이 실행될 때마다 세 번의 메모리 접근이 필요하다.

3.4 메소드 정보

메소드 정보는 어떤 메소드를 호출하는 invokevirtual 등의 명령에 의해 참조되며, 형식은 invokevirtual #i 와 같다. i는 2바이트의 인덱스 값으로 원래 이 값은 클래스 파일 내의 상수풀을 가리키는 인덱스 값이며, simpleRTJ에서는 상수풀 오프셋 테이블의 인덱스 값이다.

실제 메소드 정보는 method_T 구조체에서 정의된다.

```
struct method_T {
 class_t        *class_ptr;
 uint16         flags;   /* method flags */
 uint16         locals;  /* size of locals */
 uint16         hash;
 uint16         nargs;   /* number of arguments */
 uint16         blen_idx;/* bytecodes length */
 uint16         unused;
};
```

이 구조체의 항목들 중 flags, locals, blen_idx 등의 값은 클래스 파일 내에 포함되어있는 내용과 동일하다. 이 메소드가 실행될 때 필요한 스택의 크기는 클래스 파일 내에 포함되어있었지만 이 구조체에서는 생략되어있는데, simpleRTJ 는 모든 메소드들에 대해서 동일한 크기의 스택. 즉 가장 많은 스택 공간을 요구하는 메소드와 동일하게 스택 공간을 할당하기 때문에 별도의 스택 크기 정보는 포함되지 않는다.

이 메소드가 필요로 하는 인자(argument)들에 대한 정보는 클래스 파일 내의 상수풀에 있지만, 여기서는 nargs를 사용하여 몇 개의 인자가 필요한지를 알 수 있게 하였다.

그외에도 class_ptr 과 hash 항목들이 있는데, class_ptr 은 이 메소드가 현재 클래스 내의 메소드인지 또는 상위 클래스 내의 메소드인지를 판단할 수 있게 한다. 바이트코드 정의에 따르면 invokevirtual 명령이 실행되면 새로운 자바 스택 프레임이 만들어지며, 이 프레임의 지역변수 배열의 0번째 항목에는 클래스의 인스턴스를 가리키는 참조값(reference)이 들어간다. 이 참조값은 현재의 인스턴스가 어느 클래스의 것인지를 알려주는 포인터를 가지고 있는데, 그 포인터 값과 class_ptr 이 동일하다면 지금 수행하려는 메소드는 현재 클래스 내의 메소드라는 뜻이며, 그렇지 않다면 상위 클래스에 있는 것이라는 뜻이다.

method_T 구조체에 이어 실제 코드 부분이 따라온다. 메소드 호출에 따른 시간에 대해 알아보면, ① 먼저 상수풀 오프셋 테이블을 참조하여 특정 메소드의 위치를 안다; ② 그 위치에 놓인 메소드 정보, 즉 method_T 구조체를 읽는다. 따라서 invokevirtual #i 등의 명령이 실행될 때마다 두 번의 메모리 접근이 필요한다 (코드부분은 제외하고).

4 분석 및 결론

simpleRTJ 에서는 정적인 클래스 적재만 지원하므로 상수풀 내에서 링크와 관련된 모든 기호명의 사용을 배제할 수 있으며 따라서 메모리 공간면에서 절약의 여지가 매우 많다. 또한 ClassLinker는 클래스 파일에는 포함되어있으나 실제 응용 프로그램에서 사용되지 않는 메소드나 필드는 이미지 파일에 아예 넣지 않으므로 이에 따른 효과도 있다. 일반적인 프로그램에 대해 적용해 본 결과 클래스 파일 크기에 대비한 이미지 파일의 크기는 23% 정도에 불과한 것으로 조사되었다. 향후 보다 실제적인 예제들에 대해 메모리 절감효과를 분석해 보려고 한다. 클래스 내부 정보에 대한 접근의 용이성에 대해서는 3절에서 대략적 분석을 했으며, 향후 보다 정량적인 분석을 시행하고자 한다.

내장형 시스템과 같이 자원의 제약이 많은 환경에서는 메모리를 절감하면서 프로세서로 하여금 고속의 접근이 가능하게 하는 클래스 파일의 포맷의 설정이 매우 중요하며, 본 연구에서는 특히 simpleRTJ에서 적용된 방식에 대해 알아보았다. 보다 면밀한 검토를 통해 메모리 사용면과 접근의 효율면에서 더욱 효과적인 클래스 파일의 메모리 배치에 대해 연구할 계획이다.

참 고 문 헌

[1] S. Helal, "Pervasive Java," *IEEE Pervasive Computing*, pp.82-85, Jan-Mar 2002

[2] J. Engel, *Programming for the Java Virtual Machine*, Addison-Wesley, 1999

[3] RTJ Computing, *simpleRTJ: a small footfrint Java VM for embedded and consumer devices*, http://www.rtjcom.com

[4] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997