자바 프로그램에서 메소드 호출 빈도의 분석

양희재
경성대학교 컴퓨터공학과
e-mail:hjyang@star.ks.ac.kr

Analysis of Method Invocation Frequency in Java Program

Heejae Yang
Dept of Computer Engineering, Kyungsung University

요 약

하나의 자바 프로그램은 수많은 클래스들로 이루어져있고, 각 클래스들은 다수의 메소드들을 포함하고 있다. 즉 자바 프로그램은 수많은 메소드들을 사용하고 있는데, 본 연구에서는 프로그램 실행 시 이들 메소드들이 균등하게 호출되는 것이 아니라 특정 메소드들이 집중적으로 호출되는 경향이 있음을 밝혔다. 또한 절반 내외의 메소드들은 전혀 호출되지 않는 것도 발견하였다. 몇 가지 벤치마크 프로그램에 대해 위와 같은 특징들을 실제 실험을 통해 조사하였다.

1 서론

거의 모든 자바 입문서의 처음에 소개되는 가장 간단한 자바 프로그램인 HelloWorld.java [1] (화면에 “Hello, World!” 라는 문자열을 찍는 프로그램)을 실행해보면 HelloWorld 클래스를 비롯하여 java.lang.Object, java.lang.String 등 모두 295개의 클래스가 사용됨을 볼 수 있다 (JDK 1.5 환경 기준). 즉 하나의 자바 프로그램은 다수개의 클래스들로 구성되어지며, 각 클래스들은 여러 개의 메소드들을 포함한다. 각 클래스들은 생성자 메소드를 포함하여 최소 1개 이상의 메소드를 갖는다. 대표적 API 클래스의 한가지인 java.lang.String 은 JDK 1.5 환경에서 13개의 생성자 메소드(deprecated method 2개 포함)와 63개의 일반 메소드를 갖는다 (deprecated method 1개 포함) [2]. 즉 하나의 자바 프로그램은 수많은 메소드들을 포함하는 것을 알 수 있다.

본 논문의 관심은 자바 프로그램 내에서 이들 메소드들의 사용 빈도, 즉 호출 빈도를 분석해보는 것이다. 자바에서는 하나의 메소드가 호출될 때마다 그 메소드의 실행을 위한 별도의 스택 프레임(stack frame)이 할당되고, 모든 연산과 자료이동이 이 스택 프레임에 들어있는 오퍼랜드 스택(operand stack)과 지역변수배열(local variable array)에서 일어난다 [3]. 오퍼랜드 스택과 지역변수배열은 자바가상기계에서 가장 빈번히 접근되는 메모리 영역 중 하나이며, 가장 많은 에너지가 소비되는 곳이기도 하다 [4]. 따라서 시간 지연 면에서나 에너지 소비 면에서 효율적인 자바가상기계의 개발을 위해서는 스택 프레임의 사용에 대한 주목이 필요하며, 스택 프레임은 메소드 호출에 의해 생성되므로 일반적 자바 프로그램에서 메소드 호출 형태를 이해하는 것은 매우 중요하다. 본 연구에서는 특히 메소드 호출 빈도에 대해 관심을 갖고 분석해보았다.

몇 가지 벤치마크 및 응용 프로그램의 실행을 분석해 본 결과 전체 메소드들의 50% 이상이 실제로는 전혀 호출되지 않는다는 것을 발견할 수 있었으며, 20% 이하의 메소드들이 전체 호출 회수의 80% 이상을 차지함을 알 수 있었다. 즉 메소드들은 고르게 호출되는 것이 아니라 특정 메소드들만 집중적으로 사용된다는 것이다. 이런 성질을 이용하여 자주 사용되는 메소드들에 대해 스택 프레임의 사전 할당, 특별 구조의 프레임 메모리 부여 등을 통해 보다 효율적 자바가상기계의 개발이 가능할 것으로 기대된다.

본 논문의 2절에서는 자바 프로그램의 메소드 호출 빈도에 대해 정성적으로 분석해보고, 3절에서 실

† 이 논문은 한국학술진흥재단 지역대학우수과학자 지원에 의해 연구되었음 (R05-2004-000-10967-0)

제 실험을 통해 분석 결과를 확인해본다. 4절에서 본 논문의 결론을 맺는다.

2 자바 메소드의 정성적 분석

객체지향형 언어인 자바에서 특히 구별되는 메소드는 생성자 메소드와 getter/setter 메소드, API 클래스 메소드 등이 있다. 여기서는 일반적 자바 메소드의 사용 빈도에 대해 정성적으로 분석해 본다.

2.1 생성자 메소드

생성자 메소드는 클래스의 인스턴스가 생성될 때 호출된다. 자바 프로그램에서 한 클래스에 대해 여러 개의 인스턴스를 만드는 일은 그리 흔하지 않다. 그러나 자주 사용되는 클래스, 예를 들어 java.lang.String, java.awt.Color 등에 대해서는 생성자 메소드가 빈번히 호출될 확률이 높다.

또한 상속성의 원리에 따라 어떤 클래스의 인스턴스가 생성될 때 자신 뿐 아니라 부모 클래스의 생성자 메소드도 함께 호출된다. 따라서 상속성 트리의 제일 꼭대기에 있는 java.lang.Object 클래스의 생성자는 빈번히 호출될 확률이 매우 높다.

2.2 getter/setter 메소드

자바에서 클래스의 속성(attribute)은 일반적으로 캡슐화(encapsulation)로 감추어져 있으며, 속성에 대한 접근은 getter/setter 메소드에 의해 이루어진다. getter 메소드는 속성을 조회하는 목적으로 사용되며, setter 메소드는 속성을 변경하기 위한 목적으로 사용 된다 [5].

클래스의 속성은 일반적으로 자주 변경될 것으로 기대되지는 않다 [6]. 대개 인스턴스가 생성될 당시 속성 값이 정해지며, 이후에는 주로 속성 값을 참조하는 동작들만이 일어난다. 서론에서 언급한 java.lang.String 클래스의 경우 생성자 메소드를 제외한 63개의 메소드들이 모두 속성 값을 읽기만 할 뿐 바꾸지는 않는다. 또한 java.util.Date 클래스를 사용하는 응용 프로그램의 경우도 대부분 속성값, 즉 날짜를 읽기만 할 뿐 변경하지는 않는다. 즉 거의 모든 응용 프로그램에서 setter 메소드에 비해 getter 메소드가 더욱 빈번히 호출될 것으로 기대된다.

2.3 API 클래스

서론에서 언급한 HelloWorld.java 프로그램의 예에서 알 수 있듯이 하나의 자바 프로그램은 수많은 API 클래스들을 사용하고 있다. 어느 종류의 프로그램이라 하더라도 java.lang.String 클래스를 사용할 확률이 매우 높으며, java.lang.System, java.lang.Thread 등도 널리 사용된다. 그래픽 프로그램이라면 java.awt.Color, javax.swing.JFrame 등도 널리 사용될 것이다. 즉 자바 API 클래스에 속한 메소드들이 호출될 가능성이 매우 높다는 것이다.

또다른 고려점은 동일 API 클래스 내의 메소드라 할지라도 잘 알려진 메소드도 있고 그렇지 않은 것도 있다는 것이다. java.lang.String 클래스의 경우 문자열의 길이를 반환하는 length(), 특정 인덱스 위치의 문자값을 반환하는 charAt(), 문자값이 위치한 인덱스 값을 반환하는 indexOf() 등 메소드는 잘 알려진 메소드로서 자바 프로그래머들에 의해 널리 사용되고 있다. 반면 문자열 내의 내용을 정규표현에 따라 나누는 split(), 문자열을 지정된 형태로 포맷하는 format() 등은 잘 알려져 있지 않으며, 또한 잘 사용되지도 않는다. split() 은 JDK 1.4부터, format() 은 JDK 1.5부터 제공되기 시작한 메소드로서 생소한 반면 length() 등은 JDK 보급 처음부터 제공된 메소드이므로 거의 모든 프로그래머들이 잘 알고 사용하고 있다.

2.4 프로그램 목적에 따른 차이

동일 클래스의 메소드라 하더라도 프로그램 목적에 따라 메소드 호출 빈도가 전혀 달라질 수 있다. 은행 계좌를 의미하는 BankAccount 클래스를 고려해보자. 이 클래스는 입금을 의미하는 deposit() 메소드와 출금을 의미하는 withdraw() 메소드를 가진다. 두 메소드 중 어느 것이 더 자주 호출될지는 프로그램 목적에 따라 다르다. 즉 프로그램 목적이 보통예금 계좌의 구현이라면 deposit() 에 비해 withdraw() 가 더욱 자주 호출될 것이다. 그러나 적금통장 계좌의 구현이라면 deposit() 은 매월마다, withdraw() 는 만기에 한번 호출되는 것이 일반적이다. 또한 프로그램 목적이 정기예금 계좌의 구현이라면 deposit() 이나 withdraw() 모두 한번만 호출될 것으로 예상된다. 즉 같은 메소드라 하더라도 프로그램 목적에 따라 호출빈도는 전혀 달라질 수 있다.

3 실험 및 분석

메소드 호출 빈도를 조사하기 위해 몇 가지 벤치마크 프로그램과 응용 프로그램에 대해 실험을 시행하였다. 실험은 원천코드가 공개되어져 있는

simpleRTJ 자바가상기계 상에서 Embedded CaffeineMark 3.0 벤치마크 프로그램과 두 가지 응용 프로그램 등에 대해 실시했다.

3.1 실험 환경

simpleRTJ 는 RTJ Computing 사에서 개발한 임베디드 및 가전기기를 위한 소규모 자바가상기계이며, 바이트코드 인터프리터 방식에 기반하고 있다 [7]. 기본 API 클래스의 원천코드는 물론 JVM 내부 코드까지 공개되어있으므로 본 실험과 같이 메소드 호출 빈도를 알아보는 목적으로 매우 적합하다.

자바에서 메소드 호출은 invokevirtual, invokespecial, invokestatic, invokeinterface 등 4개의 바이트코드 중 하나에 의해 일어나는데, 본 실험에서는 이들 바이트코드가 실행될 때 호출되는 메소드를 찾아내어 각 메소드별 사용 빈도를 확인하게 했다.

다른 JVM과 달리 simpleRTJ 는 정적 클래스 적재만을 지원한다. 즉 응용 프로그램 실행 전에 필요한 클래스들이 미리 파악되며, 이 클래스들은 사전에 ClassLinker 프로그램에 의해 레졸루션 과정을 거쳐 하나의 이미지 파일 형태로 주어진다. ClassLinker 는 메모리 요구량을 최소화하기 위해 클래스 내에서 실제로 사용될 확률이 없는 메소드는 처음부터 배제하여 이미지 파일에서 제외시킨다. 예를 들어 java.lang.String 클래스의 메소드 중 현재 응용 프로그램 내에서 trim() 메소드가 사용되지 않는다면 이 메소드는 이미지 파일에 포함되지 않는다. 그러나 응용 프로그램이 직접 사용하지 않더라도 간접적으로 사용되는 메소드는 포함된다. 예를 들어 java.lang.String 클래스의 getChars() 메소드는 응용 프로그램이 직접 사용하지는 않지만 응용 프로그램이 사용하는 생성자 메소드 String(String s), String(StringBuffer sb) 등이 이 메소드를 호출하므로 이미지 파일에 포함된다.

ClassLinker 의 사용에 따라 클래스 내에 존재하는 많은 메소드들이 처음부터 고려 대상에서 제외되게 된다. 예를 들어 simpleRTJ에서 java.lang.String 클래스는 6개의 생성자 메소드와 34개의 일반 메소드를 갖지만, 뒤에서 언급할 Embedded CaffeineMark 벤치마크 프로그램에서 실제로 고려 대상이 되는 것은 java.lang.String 클래스에서 4개의 생성자 메소드와 10개의 일반 메소드 뿐이며 나머지 메소드들은 처음부터 이미지 파일에서 제외된다. 이것은 simpleRTJ의 정적 클래스 적재 특징에 따른 것이다.

3.2 실험 프로그램

본 실험에서는 한 개의 벤치마크 프로그램과 두 개의 응용 프로그램을 실험 대상 프로그램으로 삼았다. 벤치마크 프로그램인 Embedded CaffeineMark 3.0은 sieve, loop, logic, string, float, method 등 여섯 개의 각기 다른 벤치마크 프로그램으로 구성되는데 [8], simpleRTJ JVM은 double 형식의 자료형을 지원하지 않으므로 float 벤치마크는 본 실험에서 제외하였다.

일반 응용 프로그램으로서는 simpleRTJ 데모 프로그램으로 제공되는 i386-DemoIO 와 i386-FileCopy 를 사용하였다. 이들은 각각 디렉토리 내의 파일들을 나열하는 기능과 특정 파일을 복사하는 단순 기능을 갖는다.

표 1. 호출 회수별 메소드 분포 (CaffeineMark)

#invoke	%invoke	#method	%method
55275	15.31	2	1.03
35748	9.9	1	0.51
26128	7.24	2	1.03
21005	5.82	1	0.51
20962	5.81	1	0.51
19542	5.41	1	0.51
17874	4.95	3	1.54
17864	4.95	1	0.51
10008	2.77	1	0.51
1673	0.46	2	1.03
1668	0.46	3	1.54
1398	0.39	4	2.05
1383	0.38	1	0.51
1184	0.33	1	0.51
1096	0.3	1	0.51
864	0.24	1	0.51
680	0.19	1	0.51
22	0.01	1	0.51
20	0.01	1	0.51
16	0	3	1.54
15	0	2	1.03
11	0	1	0.51
10	0	7	3.59
7	0	2	1.03
6	0	1	0.51
5	0	17	8.72
4	0	1	0.51
2	0	22	11.28
1	0	25	12.82
0	0	85	43.59
361085	100.00	195	100.00

3.3 결과 및 분석

표 1은 embedded CaffeineMark 3.0 에 대한 실행 결과를 보인 것이다. 분석 결과 이 벤치마크 프로그램 실행을 위해 52개의 클래스가 사용되었으며, 클래스에 포함된 메소드의 개수는 도합 195개였다. 메소드 호출은 총 361,085번 일어났는데, 주목할만한 점은 전체 메소드 개수의 43.6%에 해당되는 85개가 전혀 호출되지 않았다는 것이다. 상위 5개(2.5%)의 메소드가 전체 호출 회수 중 절반을 넘는 55%를 차지했으며, 상위 13개(6.7%)의 메소드가 전체 호출 회수의 94.6%를 차지했다.

표 2는 상위 13개 메소드가 실행된 회수, 메소드명 및 메소드 시그니쳐를 나타낸 것이다. 가장 많이 실행된 메소드는 java.lang.StringBuffer 클래스의 append 였는데, CaffeineMark의 string 벤치마크에서 이 메소드를 반복적으로 호출한 결과로 해석된다. StringBuffer 및 String 클래스의 메소드들이 많이 사용되었으며, 대부분 getter 메소드임을 알 수 있다. 전체적으로 API 클래스의 메소드들이 상위 최다 호출 메소드를 차지하고 있으며, MethodAtom 클래스 등 응용 프로그램 클래스 메소드들도 비교적 많이 호출된 것을 알 수 있다. java.lang.Object 클래스의 생성자인 <init> 메소드도 호출 순위 상위 6위에 들어갔는데, 모든 객체는 Object 클래스의 하위 클래스이므로 2.1절에서 언급한 바와 같이 어느 객체가 생성되든지 결국 Object.<init> 이 실행되기 때문이다.

지면 제한 상 다른 응용 프로그램에 대한 결과는 도표로 나타낼 수 없지만, CaffeineMark와 유사한 결과를 얻을 수 있었다. 즉 i386-DemoIO 프로그램에서는 39개 클래스 156개 메소드가 사용되었으며, 이 중 50%인 78개의 메소드가 전혀 호출되지 않았다.

표 2. 최다 호출 상위 13개 메소드 및 시그니쳐

invoke	class/method	signature
55275	StringBuffer.append	([CII)LStringBuffer;
55275	StringBuffer.append	(LString;)LStringBuffer;
35748	StringBuffer.checkIndex	(I)V
26128	MethodAtom.notInlineableSeries	(I)I
26128	MethodAtom.arithmeticSeries	(I)I
21005	Object.<init>	()V
20962	System.arraycopy	(LObject;ILObject;II)V
15942	StringBuffer.length	()I
17874	String.<init>	(LStringBuffer;)V
17874	StringBuffer.toString	()LString;
17874	StringBuffer.getChars	(II[CI)V
17864	String.indexOf	(LString;I)I
10008	StringBuffer.ensureCapacity	(I)V

총 2,427번의 메소드 호출이 있었고, 상위 15개(9.6%)의 메소드가 전체 호출의 71.2%를 차지했다.

i386-FileCopy 프로그램에서는 39개 클래스 152개 메소드가 사용되었다. 10KB 크기의 파일을 복사할 때 전체 메소드 중 55%인 84개가 전혀 호출되지 않았다. 총 196번의 메소드 호출이 있었고, 상위 38개(25.0%)의 메소드가 전체 호출의 84.7%를 차지했다.

4 결론

본 논문에서는 자바 프로그램이 실행될 때 메소드들이 어떻게 사용되는지 호출 빈도에 대해 분석하였다. 분석 결과 메소드 호출은 모든 메소드에 대해 고르게 일어나는 것이 아니라 특정 메소드로 집중됨을 알 수 있었으며, 전체 메소드의 절반 정도는 전혀 호출되지 않음도 발견하였다. java.lang.Object 의 <init> 메소드가 높은 수준으로 호출되었으며, 자바 API 클래스들이 가진 메소드 사용 빈도가 높았다. 특히 getter 메소드의 사용 빈도가 setter 메소드에 비해 월등히 높다. 이러한 메소드 호출 빈도의 분석 결과는 스택 프레임 메모리의 할당 정책 등에 반영되어 보다 효율적인 자바가상기계의 개발을 가능하게 할 것으로 기대된다.

참고문헌

[1] C. Horstmann and G. Cornell, *Core Java*, Sun Microsystems Press, 1999

[2] J2SE 5.0, http://java.sun.com/j2se/1.5.0

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

[4] 양희재, "에너지 관점에서 임베디드 자바가상기계의 메모리 접근 형태", 한국정보처리학회 논문지, 12-A권 3호, 2005. 6.

[5] P. Winston and S. Narasimhan, *Onto Java 1.2*, Addison Wesley, 1998

[6] 양희재, "전형적 자바 프로그램에서 필드의 사용 형태", 한국정보과학회 추계학술대회, 2005. 11

[7] simpleRTJ, http://www.rtjcom.com

[8] Pendragon Software, *Embedded CaffeineMark 3.0*