확장배수 조절을 통한 자바 벡터 클래스의 성능향상

김백면(○) 정기원*
숭실대학교 컴퓨터학부
blueist@hanmail.net, chong@computing.ssu.ac.kr

Improving performance of Vector Class in Java by changing growth-rate multiplier in Vector's expansion

Bakmeon Kim(○) Kiwon Chong*
School of Computing, Soongsil Univ.

요 약

자바의 벡터는 크기가 스스로 확장되는 배열 기능을 가지는 클래스이다.[9] 벡터는 그 사용상의 편리함 때문에 자바 프로그램에서 자주 쓰인다.[2] 그러나 벡터는 제공하는 두 가지 크기 확장방식이 성능면에서 각각 지나치게 시간 혹은 공간 소비적이라는 문제점을 가진다. 이를 해결하기 위해 제안하는 새로운 두개의 벡터 생성자는 확장에 사용되는 확장배수를 사용자가 임으로 조절할 수 있게 해준다. 이를 통하여 사용자는 벡터의 크기확장 시 기존에 많이 사용되는 지나친 공간 소비적 방식을 시간, 공간의 병행적인 소비 방식으로 바꿀 수 있으므로 각각 상황에 알맞은 벡터의 확장방식을 선택하여 사용할 수 있다.

1. 서론

최근 널리 쓰이고 있는 자바언어의 여러 기능들 중 벡터는 상용 코드에서 가장 많이 발견할 수 있는 기능 중 하나이다.[2] 벡터는 "동적인 배열"이라 정의할 수 있으며[9] 동적으로 크기가 변한다는 사용상의 편의성 때문에 자바 프로그램에서 많이 쓰이고 있다.[2] 그러나 벡터의 크기 확장방식이 확장방법에 따라 각각 너무 시간 소비적, 혹은 공간 소비적이라는 문제점을 가지고 있다. 본 논문에서는 기존의 공간 소비적이거나 혹은 시간 소비적인 확장방식 대신 사용자가 적절한 확장 변수를 결정할 수 있는 생성자를 제시하여 기존에 많이 쓰이던 공간 소비적인 방식을 개선한 새로운 벡터를 제안한다. 또한 제안한 생성자를 이용한 테스트를 통해 새로운 벡터가 어느 정도의 시간, 공간적인 이득을 얻을 수 있는지 측정하고 기존 생성자의 측정값과 비교한다.

2. 개요

Sun Microsystems Co.에서 제공하는 자바의 Collection interface는 group of objects를 의미하며[9] 이는 element라 불리는 각각의 객체를 담고 있는 배열이다. 단순한 배열보다 기능적성이 좋아 자바 프로그래밍상에서 자주 사용되어지는 이 Collection interface는 각각의 목적에 알맞게 몇 개의 클래스로 재구현 되는데 이런 Collection implements 클래스로는 스택 (Stack), 비트맵 (BitMap), 해쉬 테이블 (HashTable) 그리고 벡터 등이 있다. 이 중 벡터는 객체를 넣고, 꺼내고, 찾고, 추출할 수 있는 기능에 중점을 맞춘 클래스로 일반적인 배열과 달리 그 크기가 실행 시에 자유롭게 확장 가능하다고 하는 특징을 가진다. 그러나 벡터는 크기 확장 시 소비되는 시간 및 공간이 전체 성능에 큰 영향을 준다는 단점을 가진다.

3. 자바의 벡터 클래스

벡터는 '스스로 확장 가능한 배열'로, 추가적인 element 삽입 시 자신의 크기보다 더 많은 수의 element가 삽입되면 벡터 스스로 자신의 크기를 확장한다. [프로그램 1]은 벡터의 크기를 확장하는 과정을 보여준다.

```
int i = Array_Size_in_Present;                    ⓐ
Object oldArray[] = Array_in_Present;             ⓑ
int newArray_Size;
if(capacityIncrement <=0)                         ⓒ
  newArray_Size = i*2;                            ⓓ
else newArray_Size = i + capacityIncrement;       ⓔ
Array_in_Present = new Object[newArray_Size];     ⓕ
  System.arraycopy( - - )                         ⓖ
```

[프로그램 1] 벡터의 확장 알고리즘

벡터의 크기확장은 현재 배열의 크기를 저장하고(ⓐ), 현재배열을 저장하고(ⓑ), 새로운 배열의 크기를 결정한 후(ⓒ,ⓓ,ⓔ) 새로운 배열을 생성하고(ⓕ) 새로 만들어진 배열에 기존 배열의 모든 element를 복사(ⓖ)하는 과정을 통해 이루어진다. 벡터의 확장은 [프로그램 1]에서 나타나듯이 한 개의 배열 생성과 element의 복사를 위한 '시간', 그리고 새로운 배열에 대한 '공간'이 필요하게 된다. 이런 추가적인 시간과 공간은 Synchronized Problem, Casting Cost Problem등과 함께 벡터의 성능상 문제점에 중요한 요인이 된다.

벡터의 확장 시 소요되는 '시간'과 '공간'은 trade-off의 관계를 가진다. 새로운 배열의 크기를 작게 결정하면 추가적인 element가 삽입되어 element의 수가 벡터의 크기를 넘어서는 경우가 자주 발생하며 그때마다 [프로그램 1] 의 알고리즘을 적용한 확장과정이 필요함으로 이를 위한 추가적인 확장의 시간이 필요하게 되고, 새로운 배열의 크기를 크게 결정하면 시간은 절약되지만 공간의낭비가 커진다. 이런 이유로 새로운 배열의 크기를 결정하는 문제는 벡터 전체의 성능과 관련하여 아주 중요한 문제라 할 수 있다.

벡터의 확장 시 새로운 배열의 크기결정은 [프로그램 1]의 capacityIncrement라는 변수의 값에 의해 두 가지 방식으로 나누어진다. capacity-Increment 값을 조사하여(ⓒ) 0 이상의 값으로 정해져 있으면 현재 사용중인 배열의 크기에 capacityIncrement 값을 더한 값을 새로운 배열의 크기로 정한다.(ⓔ) capacityIncrement 값이 0이라면 벡터는 현재 사용중인 배열의 크기에 2를 곱한 값을 새로운 배열의 크기로 정한다.(ⓓ) 이와 같이 벡터는 더하기에 의한 방식과 곱하기에 의한 두 가지 방식으로 자신을 확장한다.

CapacityIncrement 변수의 값은 벡터의 생성자에서 결정한다. 벡터는 [표 1]와 같은 생성자들을 가진다.

[표 1] JDK 1.3 API Specification에서의 벡터의 생성자

Vector ()	ⓐ
Vector (Collection c)	ⓑ
Vector (int initialCapacity)	ⓒ
Vector (int initialCapacity, int capacityIncrement)	ⓓ

4가지 생성자중 벡터의 확장과 관련된 생성자는 ⓐ,ⓒ,ⓓ 이다. initialCapacity는 벡터 생성시 벡터의 초기값을 지정하는 값이고 capacityIncrement는 벡터의 크기 확장 시 확장되는 크기를 지정해주는 값이다. 이 두 가지 값에 의해 벡터는 확장방식을 결정하게 된다.

4. 기존 벡터 클래스의 각 생성자별 성능비교

기존의 벡터클래스의 두 가지 확장 방식에 대한 성능 비교를 위해 각 생성자를 [프로그램 2]의 테스트 프로그램으로 수행하여 각각의 시간과 공간을 비교하였다.

```
import java.util.*;
class Vtest {
  public static void main(String[] arg) {
    Vector testVector = new Vector ();                    ⓐ
    // Vector testVector = new Vector (5000, 10);         ⓑ
    // Vector testVector = new Vector (5000);             ⓒ

    for(int i=0;i<100000;i++)
      testVector.add("String");
    }
}
```

[프로그램 2] 벡터의 성능 측정을 위한 테스트 프로그램

ⓐ 생성자와 ⓑ 생성자와 ⓒ 생성자를 사용하여 수행한 각각의 프로그램의 결과는 [표 2]와 같다.

[표 2] 벡터의 생성자별 성능 비교

	ⓐ	ⓑ	ⓒ
시간 (ms)	110	14500	60
남은 공간 (bytes)	1,275,704	1,756,344	1,451,064

위의 표에서 ⓐ, ⓒ의 결과는 곱하기의 패턴에 의한 확장의 결과이고 ⓑ의 결과는 더하기의 패턴에 의한 결과이다. [표 2] 의하면 시간 성능적인 측면에서는 곱하기의 패턴에 의한 경우가 효율적이지만, 공간적인 측면으로는 더하기 패턴에 의한 경우가 보다 효율적이라는 것을 알 수 있다.

테스트의 결과에 대한 검증을 위해 각 생성자별 확장방식 알고리즘을 분석해 각각의 시간 및 공간별 소모량을 측정하였다.[1] 그 결과는 [표 3]와 같다.

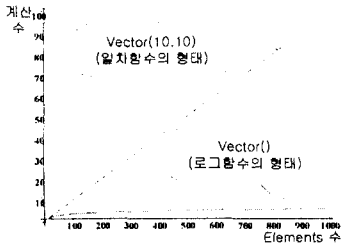
[표 3] 벡터의 생성자별 시간 및 공간 계산[2]

	Vector() ⓐ
계산 수	$\log_2 n - \log_2 10+a$ ($\log_2 10 - \log_2 5 > a >=0$)
사용되는 공간	$10 * 2^c$, $n + a$ ($n > a >=0$)

	Vector(5000,10) ⓑ
계산 수	$\frac{n-m}{I} + a$ ($1 > a >=0$)
사용되는 공간	$m + n*c$, $n + a$ ($n > a >=0$)

	Vector(5000) ⓒ
계산 수	$\log_2 n - \log_2 m + a$ ($\log_2 m - \log_2 m/2 > a >=0$)
사용되는 공간	$m * 2^c$, $n + a$ ($n > a >=0$)

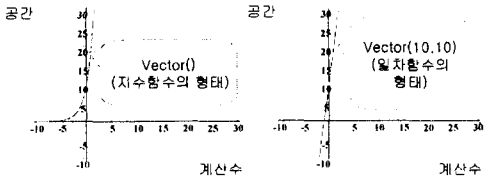
각각의 계산된 공식의 의하여 각 확장 패턴별 시간 성능 비교를 위한 그래프는 [그림 1]과 같다.



[그림 1] 각 확장패턴의 시간 성능비교

시간 성능의 경우 곱하기 패턴으로 수행되는 확장은 로그함수의 형태를 가지는데 비해, 더하기 패턴으로 수행되는 확장은 일차함수의 형태를 가진다. 이로 인하여 Element수가 1000개라면 그 계산 수의 차이는 약 14배까지 생긴다. 이런 이유로 여러 연구에서는 더하기 패턴으로 이루어지는 확장은 특별한 경우 이외에는 쓰지 않기를 권하고 있다.[2]

각각의 계산된 공식의 의하여 각 확장패턴별 공간 성능 비교를 위한 그래프는 [그림 2]와 같다.



[그림 2] 각각의 패턴으로 수행되는 확장의 공간 성능

공간 성능의 경우 곱하기 패턴으로 수행되는 확장은 지수함수의 형태를 가짐에 비해 더하기 패턴으로 수행되는 확장은 일차함수의 형태를 가진다.

위의 테스트 및 알고리즘 그래프에 의해 곱하기 패턴으로 수행되는 확장은 공간 소비량이 많은 공간 소비적인데 반해 더하기 패턴으로 수행되는 확장은 시간 소비량이 많은 시간 소비적이라는 것을 알 수 있다.

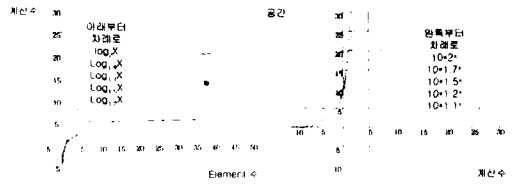
5. 새로운 벡터 클래스

성능 테스트와 계산에 의한 측정에 의하면 시간 소비적인 더하기 패턴의 확장은 공간 성능 측면에서의 장점에도 불구하고 많은 시간 성능적 단점으로 인하여 종합적 성능면에서 많은 성능 손실이 있다. 이런 이유로 많은 연구에서 시간 소비적인 방법은 특별한 경우가 아니라면 쓰지 않기를 권하고 있다. [2]

그러나 공간 소비적인 확장 방법은 공간적인 측면에서 지나치게 많은 성능 손실을 가져오며 현재 제공되는 벡터 클래스로는 이 성능 저하를 만회할 수 있는 방법을 찾기는 어렵다. 현재의 벡터 클래스는 곱하기 패턴의 확장 시 기본적으로 확장 패턴을 x2 로 고정하고 있고, 사용자가 이를 원하는 대로 변경할 수 있는 방법을 제공해주지 않는다. 만약 이 곱해지는 수를 마음대로 조절할 수 있다면 시간과 공간적인 측면에서 어떤 이익이 생기는지를 [표 3]의 식에 값을 대입해 예측해보았다. [그림 3]은 예측 결과를 그래프로 나타낸 것이다.

[1] 시간 성능은 벡터의 확장을 위한 계산의 수를 기준으로 계산하였다

[2] n은 element수, i는 capacity increment, m은 initial capacity, c는 계산 수



[그림 3] 확장배수 조절 시 각 성능에 미치는 영향

[그림 3]의 처음 그래프가 나타내는 시간 성능의 경우 확장 배수가 작아질수록 연산 수는 많아지지만 로그함수의 특성상 element의 수가 많아질수록 그 차이는 그리 크지 않다는 사실을 확인할 수 있다. 반면 두 번째 그래프가 나타내는 공간 성능의 경우 곱해지는 배수가 작을수록 그 기울기가 줄어들며 계산수가 커질수록 더욱 공간의 차이는 확연히 드러난다; 측정의 결과를 분석해보면 공간 소모적인 곱하기 패턴의 확장 시 곱해지는 배수의 조정으로 현재 사용되는 벡터의 성능에 비해 공간적 혹은 시간적 성능의 증가가 가능하다는 사실을 알 수 있다.

이에 따라 새로 제안하는 벡터는 기존의 생성자에 더하여 [표 4] 에서 정의하는 두개의 새로운 생성자를 가진다.

[표 4] 새로 추가되는 벡터의 생성자

Vector (long growRate)
벡터의 확장배수를 주어 기본 크기가 10이고
확장배수가 growRate인 벡터 생성
Vector (int initialCapacity, long growRate)
벡터의 확장배수를 주어 기본 크기가 initialCapacity
이고 확장배수가 growRate인 벡터 생성

각각은 growRate라는 새로운 파라메터를 가진다. 새로 추가된 growRate는 곱하기 패턴으로 이루어지는 확장 시 곱해지는 배수를 가리키며 growRate는 확장 배수이므로 항상 1보다 큰 수이여야만 한다. 이 새로운 확장자를 통해 사용자는 기존 벡터에서는 조절하지 못했던 확장 배수를 조절할 수 있다.

새로운 생성자를 포함한 벡터의 확장은 [프로그램 3]의 알고리즘을 통해 이루어진다.

```
int i = Array_Size_in_Present;
Object oldArray[] = Array_in_Present;
int newArray_Size;
if(capacityIncrement <=0)
  if(growRate <=1)                    ⓐ
    newArray_Size = i*2;              ⓑ
  else newArray_Size = (int)i*growRate;   ⓒ
else newArray_Size = i + capacityIncrement;
Array_in_Present = new Object[newArray_Size];
System.arraycopy(- - );
```

[프로그램 3] 새로운 벡터의 확장 알고리즘

새로운 알고리즘은 먼저 growRate의 설정 여부를 파악하고(ⓐ), growRate가 설정되어있지 않다면 기존의 벡터와 같이 기존의 배열 크기에 2를 곱한 값을 새로운 배열의 크기로 결정한다. (ⓑ) growRate가 설정되어있다면 기존의 배열 크기에 growRate를 곱한 값을 새로운 배열의 크기로 결정한다. (ⓒ)

6. 새로운 벡터 클래스의 성능 평가

새로운 벡터 클래스의 성능평가를 위하여 [프로그램 4]를 작성하여 테스트를 수행하였다.

```
import java.util.*;
class Vtest {
  public static void main(String[] arg) {
    Vector testVector = new Vector (5000, 1.7);

    for(int i=0;i<100000;i++)
      testVector.add("String");
  }
}
```

[프로그램 4] 새로운 벡터의 성능 측정을 위한 테스트 프로그램

새로운 벡터의 initial Capacity는 기존 벡터의 테스트 때와 같이 5000으로 설정하였고 새로 추가된 파라메터인 growRate는 1.7로 설정하여 벡터의 확장 시 기존 벡터의 1.7배를 확장하도록 하였다. 위 프로그램의 실행 결과와 기존 벡터 테스트의 결과와의 비교는 [표 5]과 같다.

[표 5] 새로운 벡터와 기존 벡터와의 성능 비교

	Vector()	Vector (5000,10)	Vector (5000)	Vector (5000,1.7)
시간 (ms)	110	14500	60	67
남은공간(byte)	1275704	1756344	1451064	1521184

새로운 벡터의 테스트와 상황이 가장 유사했던 기존 벡터의 테스트는 ' Vector(5000,10)' 인 경우이다. 이 두 경우를 비교하면 시간 성능의 약간의 손실에 비해 상당히 큰 공간적 성능 이익을 얻은 것을 알 수 있다. 이런 결과로 확장 배수를 사용자가 결정함으로써 벡터의 성능을 향상시킬 수 있다는 것을 알 수 있다.

7. 결론

기존 벡터 클래스는 내부 배열의 확장 시 확장배수를 2로 고정하여 이미 정해진 시간적, 공간적 성능 손실을 피할 수 없었다. 본 연구에서는 이런 측면을 개선하기 위하여 새로운 벡터 클래스를 제안하였다. 이 새로운 벡터 클래스는 내부 배열의 확장 시 확장 배수를 사용자가 임의로 조절 가능하게 하여 사용자로 하여금 상황에 적합한 확장 배수를 선택하여 벡터의 성능을 최대한으로 끌어올릴 수 있도록 하였다. 또한 새로운 벡터 클래스와 기존의 벡터클래스의 교체만으로 같은 프로그램의 실행 결과 성능의 증가를 확인하였다.

벡터는 확장시의 추가적인 시간적, 공간적 성능문제 외에도 Synchronized Problem, Casting Cost Problem등 성능과 관련된 많은 문제점을 가지고 있다. 향후에는 이런 벡터의 성능을 저해하는 문제점을 해결하기 위한 추가적인 연구를 진행할 것이다.

7. 참고문헌

[1] Jack Shirazi, Java Performance Tuning, O'reilly, September, 2000

[2] Dov Bulka, Java Performance and Scalability Volume 1, Addison Wesley, November, 2000

[3] Haggar, Peter, Practical Java Programming Language Guide, Addison-Wesley , February, 2000

[4] Cay S. Horstmann, Core Java 2 Volume 1, Sun Microsystems Press, 2000

[5] Cay S. Horstmann, Core Java 2 Volume 2, Sun Microsystems Press, 2000

[6] Bruce Eckel, Thinking in Java 2nd Edition, www.bruceeckel.com, 2000

[7] 최재영, 최종명, 유재우, 프로그래머를 위한 자바 2, 홍릉과학 출판사, 2000

[8] Deitel & Deitel, Java How to Program 3rd Edition, Prentice Hall, 2000

[9] JavaTM 2 Platform, Standard Edition, v 1.3 API Specification, Sun Microsystems, 2000

[10] Dennis M. Sosnoski, Java Performance Programming, Part 3: Managing Collections,, www.javaworld.com/javaworld/jw-02-2000/jw-02-performance.html, February 2000