IBM

Eye on performance: Micro performance benchmarking

Bytecode offers a glimpse into application performance

Level: Intermediate

Jack Shirazi (jack@JavaPerformanceTuning.com), Director, JavaPerformanceTuning.com
Kirk Pepperdine (kirk@JavaPerformanceTuning.com), CTO, JavaPerformanceTuning.com

05 Dec 2003

Java performance enthusiasts Jack Shirazi and Kirk Pepperdine, Director and CTO of JavaPerformanceTuning.com, follow performance discussions all over the Internet to see what's troubling developers. While surfing the Usenet newsgroup `comp.lang.java`, they came across some interesting low-level performance tuning questions. In this installment of *Eye on performance*, they dive into some bytecode analysis to try and answer some of these questions.

Although no Usenet discussion groups are dedicated purely to Java performance, there are many discussions about performance tuning and optimization. A good portion of these discussions rely on results obtained from micro performance benchmarks, so we're also going to talk about some of the benefits and pitfalls of microbenchmarking in this month's column.

Pre or post?

One particular question caught our eye: which operation is faster, `i++` or `++i`? We have seen this question asked in various forms in every forum that we cover. As simple as the question it appears, there seems to be no definitive answer.

First, in case you are unaware of the difference, `++i` uses the *pre-increment* operator, whereas `i++` uses the *post-increment operator*. While both increment the variable `i`, the pre-increment operator returns the value `i` had before the increment operation, and the post-increment operator returns the value `i` has after the increment operation. A simple test program illustrates the difference:

```
public class Test {
  public static void main(String[] args) {
    int pre = 1;
    int post = 1;
    System.out.println("++pre  = " + (++pre));
    System.out.println("post++ = " + (post++));
  }
}
```

Running the `Test` class produces the following output:

```
++pre  = 2
post++ = 1
```

Microbenchmarking

Can't we just try running each operation repeatedly, and see which gives the faster runtime? The quick answer is yes, but the danger is that microbenchmarks don't always measure what you think they are measuring. Quite often, Just-in-time (JIT) Compiler optimizations and variations overwhelm any detectable differences in underlying performance. For example, one such test showed the second `i++` operation to be faster than the first `++i` test. But swapping the order of the tests showed exactly the opposite results! From this we can only conclude that the testing methodology is flawed. Further investigation reveals that the source of this confusing result stems from the HotSpot optimizations that kick in during the first test. These optimizations have the dual effect of adding extra overhead to the first run and removing the cost of interpretation from the second run.

Other variations of the microbenchmark, such as repeating the test after the JIT start-up cost has been incurred, simply give inconclusive results when run repeatedly. This may tell us that there is no difference in speed between the two operators, but we cannot be certain of this.

The iinc bytecode operator

In his newsletter, *The Java Specialists Newsletter*, Dr. Heinz Kabutz asked his readers which was faster: `i++`, `++i` or `i+=1`? In Issue 64 he reported that the question was answered by one of his readers with a simple technique: looking at the compiled bytecode. In fact, he examined four increment statements:

```
++i;
i++;
i -= -1;
i += 1;
```

The compiled bytecode can be easily examined using the disassembler, `javap`, that comes with the Java SDK. For each of the four increment statements, the resulting bytecode was `iinc 1 1`.

The `iinc` operator has two parameters. The first parameter specifies the index of the variable in the JVM's local variable table; the second parameter specifies the value that the variable is to be incremented by.

Well, does that give us our definitive answer? After all, if the different source codes compile to the identical bytecode, then there can be no difference in speed, right?

Operator context

So, if the code fragments all compile to the same bytecode, what is the point of having different operators at all? Surely they aren't just redundant. Well, let's look back to the definitions of pre- and post-increment. The critical aspect is when the variable is accessed. If there is no access of the variable, then there is no difference between the operators. The statements `i++` and `++i` on their own are functionally identical. However, the statements `j=i++` and `j=++i` are *not* functionally identical. We need to examine the bytecode in the context of an additional assignment. Consider these two similar methods:

```
public static int preIncrement() {
  int i = 0, j;
  j = ++i;
  return j;
}
public static int postIncrement() {
  int i = 0, j;
  j = i++;
  return j;
}
```

Disassembling `preIncrement()` and `postIncrement()` yields the following bytcode:

```
Method int preIncrement()
   0 iconst_0
   1 istore_0
   2 iinc 0 1
   5 iload_0
   6 istore_1
   7 iload_1
   8 ireturn

Method int postIncrement()
   0 iconst_0
   1 istore_0
   2 iload_0
   3 iinc 0 1
   6 istore_1
   7 iload_1
   8 ireturn
```

Now we *can* see a difference between the two methods: `preIncrement()` returns `1` whereas `postIncrement()` returns `0`. Let's examine the bytecode to get a better understanding of the difference. First, we'll explain the various bytecode operators that we can see in the disassembled code.

Bytecode operation: i=0

The `iconst_0` operator pushes the integer `0` onto the stack. To fully understand this, remember that the JVM emulates a stack-based CPU. (See the `java.util.Stack` class documentation if you have not come across stacks before.) The JVM pushes things onto the stack when it will need to operate on them later, and pops them off when it is ready to operate on them.

There are a number of different datatypes in the Java language, and there are different bytcode operators for the different datatypes. As a special optimization, the values -1, 0, 1, 2, 3, 4, and 5 have special bytecodes. If we weren't dealing with one of these, the compiler would have generated a `bipush` bytecode operation to push a specific integer onto the stack. (For instance, if the first statement of the method had been `int i = -2`, then the first bytecode would have been `bipush -2`.)

The next statement `istore_0` might look like another of those special bytecodes dealing with integers -1 to 5, but in fact the `_0` this time refers to an index into the local variable table. The JVM maintains a table of variables local to the method, and the `istore` bytecode pops the value at the top of the stack and stores that value in the local variable table. In this case we have `istore_0` so the value is stored at index 0 of the table.

All that long-winded explanation covered the Java bytecode for `"i=0"`, which got converted to the bytecode:

```
0 iconst_0
1 istore_0
```

More bytecode operations

Now that we know about the stack and the local variable table, we can cover the other bytecodes more quickly. The bytecode `iinc 0 1`, as we said earlier, increments the variable at index 0 of the local variable table by the value 1, `iload_0` pushes the value at index 0 of the local variable table onto the stack, and `ireturn` pops the value from the stack and pushes it onto the operand stack of the invoking method. Table 1 below summarizes the bytecodes.

Table 1. Bytecodes

Bytecode	Summary
iconst_0	Push 0 onto the stack
iconst_1	Push 1 onto the stack
istore_0	Pop the value from the stack and store it at index 0 of the local variable table
istore_1	Pop the value from the stack and store it at index 1 of the local variable table
iload_0	Push the value at index 0 of the local variable table onto the stack
iload_1	Push the value at index 1 of the local variable table onto the stack
iadd	Pop the two integers from the operand stack adding them. Push the integer result back onto the stack
iinc 0 1	Increment the variable at index 0 of the local variable table by 1
ireturn	Pop the value from the stack and push it on the operand stack of the invoking method. Exit method

Comparing the methods

Now, let's look at those disassembled bytecodes again. We'll use `lvar` to denote the local variable table as if it were a Java array, and comment the bytecodes:

```
Method int preIncrement()
   0 iconst_0     //push 0 onto the stack
```

```
   1 istore_0     //pop 0 from the stack and store it at lvar[0], i.e. lvar[0]=0
   2 iinc 0 1     //lvar[0] = lvar[0]+1 which means that now lvar[0]=1
   5 iload_0      //push lvar[0] onto the stack, i.e. push 1
   6 istore_1     //pop the stack (value at top is 1) and store at it lvar[1], i.e. lvar[1]=1
   7 iload_1      //push lvar[1] onto the stack, i.e. push 1
   8 ireturn      //pop the stack (value at top is 1) to the invoking method i.e. return 1

Method int postIncrement()
   0 iconst_0     //push 0 onto the stack
   1 istore_0     //pop 0 from the stack and store it at lvar[0], i.e. lvar[0]=0
   2 iload_0      //push lvar[0] onto the stack, i.e. push 0
   3 iinc 0 1     //lvar[0] = lvar[0]+1 which means that now lvar[0]=1
   6 istore_1     //pop the stack (value at top is 0) and store at it lvar[1], i.e. lvar[1]=0
   7 iload_1      //push lvar[1] onto the stack, i.e. push 0
   8 ireturn      //pop the stack (value at top is 0) to the invoking method i.e. return 0
```

Hopefully, it is now a bit clearer what is happening, and what are the functional differences between the methods. The only difference is that the third and fourth bytecodes are swapped over between the two methods. The commented bytecode shows clearly that in the `postIncrement()` method, the `iinc` operation is completely wasted, since no further use is made of `lvar[0]`, the local variable element being updated, from that point on. For this particular method an optimizing JIT Compiler could completely eliminate that bytecode operation. So in this particular situation, the `postIncrement()` method could potentially have one bytecode operation fewer than the `preIncrement()` operation, thus making it more efficient. But in most situations where the post-increment operator is used, the increment cannot be optimized away.

So which is faster?

What have we learned? Well, there is definitely no difference between `++i` and `i++` if they are used as statements on their own. Only the presence of an additional assignment forces a difference in the compiled bytecode.

In the context of an assignment, it is possible that comparing the use of the pre-increment or post-increment operators could result in different runtimes. But that is unlikely where the functional result of using either is the same. Remember, in our examples here, the method actually returned different values depending on whether we used the pre-increment or post-increment operator. In a normal program, one of those varieties would be a bug.

The final word

In the past, we could measure the cost of a set of operations as they were expressed in the language. This is because the translation of these operations to the underlying runtime environment remained static, which is not the case in the Java runtime. The Java runtime's ability to dynamically optimize running code is a extremely powerful feature. Although this feature has not eliminated our ability to conduct a micro-performance benchmark, it has resulted in our needing to take much more care when we

use this technique.

Resources

- Read the *Eye on performance* series by Jack Shirazi and Kirk Pepperdine.
- Greg Travis's "How to lock down your Java code (or open up someone else's)" (*developerWorks*, May 2001) offers information on decompiling a Java class file.
- Learn more about IBM's research into high performance JVMs with The Jikes Research Virtual Machine (*developerWorks*, February 2000).
- The Java HotSpot Virtual Machine, v1.4.1 (*java.sun.com*, September 2002) is the official white paper on JVM HotSpot technology.
- "Micro-Tuning Step-by-Step" by Jack Shirazi (*ONJava*, March 2002) offers practical advice on micro-performance benchmarking.
- Browse for books on these and other technical topics.
- Find hundreds of articles about every aspect of Java programming in the *developerWorks* Java technology zone.

About the authors

Jack Shirazi is the Director of JavaPerformanceTuning.com and author of Java Performance Tuning (O'Reilly).

Kirk Pepperdine is the Chief Technical Officer at Java Performance Tuning.com and has been focused on Object technologies and performance tuning for the last 15 years. Kirk is a co-author of the book ANT Developer's Handbook.