JavaLife

William Rossi

CS 1699 - DELIVERABLE 4: Performance Testing Conway's Game of Life

Profiling Conway's Game of Life turned out to be a much more problematic task than I had originally thought it would be. I performed a good deal of troubleshooting to figure out how to run the program with the default packages included (curse you, com.laboon!) and then it took even further tinkering before I figured out how to get VisualVM to profile my CPU at the method level. What I learned is that I had to tell the profiler what package to look in for the methods that it would be profiling (CURSE YOU, COM.LABOON!).

Once I finished the initial setup, however, the profiling process was actually remarkably easy. I set the game to run through 30,000 iterations and profiled the CPU usage of the game's methods while it was running, taking a snapshot after an arbitrary (but long enough) amount of time had passed. The snapshots below show what methods were being called, how many times they were called, and how much time the CPU spent executing these methods (relative to the other methods). The results were pretty striking, and it was immediately apparent to me that the toString() method in the World class was the most inefficient method. As shown in the snapshot before I refactored the method, the CPU was spending just about all of its time (100%, although I'm fairly certain that's not an exact figure) executing the toString() method, even though other methods--like the getNumNeighbors() method--were being invoked much more often.

Upon examination of the World class, I realized that its inefficiency stemmed from its use of string concatenation: in order to create the visual aspect of the game, the toString() method was creating an initial string and then continuously concatenating it with new strings. From my prior experience with the Java language I knew that string objects were immutable, so what this method was really doing was creating a new string object every time the "+=" operator was used to concatenate the old string with a new one. This concatenation happened many times within each invocation of the toString() method, and the toString() method was being called (in total) 30,000 times! Not only did this constant creation of new objects waste time, but it also wasted space!

To refactor the method, I decided to replace the string object within the method with a StringBuffer. Then, instead of concatenating two strings together repeatedly, I could simply append the necessary strings onto the StringBuffer. Finally, I would return the String representation of that StringBuffer (by calling its built-in toString() method) so that the type of variable returned would not need to be changed--and this would also ensure that my unit tests were passing both before and after the refactor as is also shown in the screenshots below. This not only saved space because each call to World.toString() would only create ONE StringBuilder object rather than MANY String objects of increasing size, but it also saved time because appending to a StringBuilder is much faster than creating a new String by concatenating two strings. As shown in the "after" snapshot of the CPU profile, the toString() method decreased in total execution time by about 80%, and note that the method itself had been invoked MANY more times than in the snapshot taken before the refactor!

To unit test the method, I tried to think of various boundary cases that might apply to the method. My first thought was to test the toString() output of a world of size zero, and this test was easy to write because I could very quickly determine what the output should be and how long it should be. I wanted to test a very large world, but manually determining the size and value of that toString() output was too challenging. I worked around this challenge by testing that two identical worlds generated from the same arbitrary arguments both produced the same output and output length when their toString() methods were invoked; the worlds ranged in size from 0 to 199. Once that was done, I wanted to ensure that two worlds produced using different sizes would NOT produce strings of the same length when their toString() methods were invoked (as this would mean that the program was broken), so I tested that as well. Finally, I decided to test that a world created from arbitrary (but correct in terms of the program's usage) values would not return a null string from its toString() method, because this would also mean that the program was broken. Since the pre- and post-refactor toString() method both returned String objects (and these strings were functionally equivalent), the tests passed before and after I refactored the method.

Aside from the challenges I faced initially (described in the first paragraph), there was one minor annoyance that I faced. I realized that, even though I knew from experience how String concatenation worked in terms of space usage, it would be more convincing if I profiled the memory usage of the program both before and after my refactor to demonstrate that I had, in fact, decreased the amount of space used by the program. Unfortunately, for some reason I could not make my machine run the original version of the World.toString() method; I think maybe it had cached the new version and continued to run that. I tried clearing my cache, and still the same problem was occurring. For whatever reason, every time I tried to memory profile the old version of the program, it kept creating StringBuilder objects, and this was a tipoff to me that it was not calling the old toString() method because that did not contain any StringBuilders. I eventually became so frustrated with the problem that I gave up. I figured it would be pointless to put in a profile of the memory usage of the newly refactored method since there was nothing to compare it to, AND I am also confident that everything I've already mentioned about the space savings of StringBuffers versus Strings when concatenating is true.

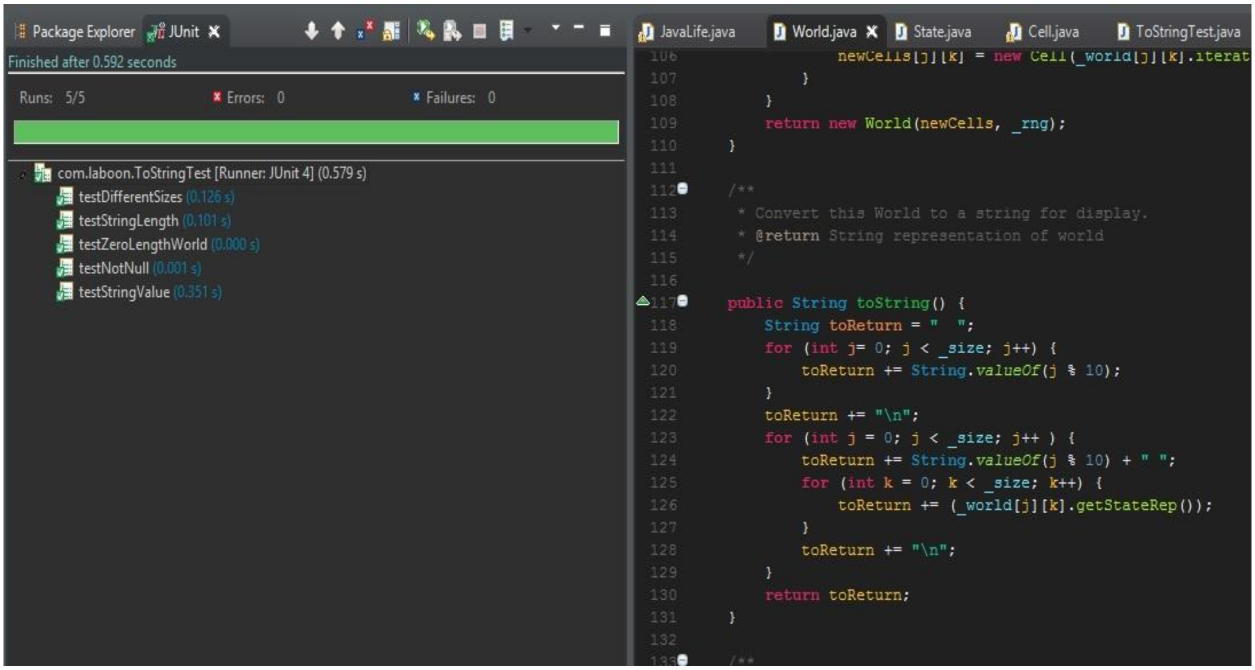
Before refactor:

Call Tree - Method	Time [%]	Time	Invocations
main		25540 ms (100%)	1
com.laboon.World.**toString** ()		26201 ms (100%)	8275
Self time		26272 ms (100%)	8275
com.laboon.Cell.**getStateRep** ()		0.000 ms (0%)	20686185
com.laboon.Cell.**getStateRep** ()		0.094 ms (0%)	100
com.laboon.World.**iterate** ()		0.000 ms (0%)	8275
Self time		510 ms (2%)	8275
com.laboon.World.**<init>** (com.laboon.Cel		1.12 ms (0%)	8275
com.laboon.World.**getNumNeighbors** (cc		0.000 ms (0%)	20687500
Self time		1001 ms (3.9%)	20687500
com.laboon.Cell.**isAlive** ()		0.000 ms (0%)	165500000
com.laboon.Cell.**iterate** (int)		0.000 ms (0%)	20687500
com.laboon.Cell.**<init>** (com.laboon.State		0.000 ms (0%)	20687500

After refactor:

Call Tree - Method	Time [%]	Time	Invocations
main		3806 ms (100%)	1
com.laboon.World.**iterate** ()		3022 ms (79.4%)	17491
Self time		1725 ms (45.3%)	17491
com.laboon.World.**getNumNeighbors** (cc		1460 ms (38.4%)	43727009
Self time		4076 ms (100%)	43727009
com.laboon.Cell.**isAlive** ()		0.000 ms (0%)	349816067
com.laboon.World.**<init>** (com.laboon.Cel		3.78 ms (0.1%)	17490
com.laboon.Cell.**iterate** (int)		0.000 ms (0%)	43727008
com.laboon.Cell.**<init>** (com.laboon.State		0.000 ms (0%)	43727008
com.laboon.World.**toString** ()		785 ms (20.6%)	17490
Self time		931 ms (24.5%)	17490
com.laboon.Cell.**getStateRep** ()		0.000 ms (0%)	43725000

Unit Tests before refactor:



Unit Tests after refactor: