Effective Java Programming

measurement as the basis

Structure

- measurement as the basis
  - benchmarking
    - micro
    - macro
  - profiling
    - why you should do this?
    - profiling tools

Motto

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Donald Knuth

The problem

- Modern software is too complex
- Even the wisest man is not suitable for performance tuning
- need for techniques and tools!
  - benchmarking
  - profiling

What is benchmarking?

- it is a comparison of different solutions on the basis of measurements (time, memory, ...)
- different solutions, but giving the same results
  - different data structures
  - different technologies (ORM, MVC)
  - various containers
- the essence is to compare
  - single result does not say anything
  - results are useful when you can compare them

How to measure?

- stopwatch
  - funny, but often appropriate
  - the most versatile technique
  - no interference with the code
  - does not require complicated tools
  - you do not always need the milliseconds
    - application start time
    - time to open a document
    - time scrolling through large tables
    - DB query execution time

How to measure?

- measurements taken into account in the code
  - calculation based on system time before and after run
  - System.currentTimeMillis ()
  - precision in milliseconds (?)
  - interference with the code - can lead to failure
  - addition to each method is tedious
  - generic stopwatch class

Why build benchmarks?

- ready made ones to compare the JRE
- but they do not evaluate your program
- you need to create a benchmark studying your code
  - comparison of alternative solutions
  - performance profiling in your application
  - track performance and trends in the manufacturing cycle
- benchmark - code, environment, utility
  - allows for comparison of different solutions
  - **repeatable test cases**

Micro-benchmark

- checks for a specific part of the system
- often even a few lines of code, for example:
  - Draw a rectangle 100 000, read 10MB file from disk
  - Sorting an array of 100 000 elements, performance of specific tasks
- good
  - quick execution, focus on the problem
- bad - does not represent the behavior of a real application
  - JVM "warm up"
  - unrealistic interactions in the system

Macro-benchmark

- "true" macro benchmark tests the system in the form the user sees it
- I would have to have the system ready...
- how to simulate?
  - work on real data
  - run on the target platform
  - lean on use cases
- you need to understand how users will work with the system
  - interactions and the correct sequence of events

Benchmark analysis

- results often vary considerably between executions
  - background processes
  - network traffic
  - GC
- perform many times and average the results
- compare results of different solutions
  - min, max, average

What is profiling?

- profiling is to identify components consuming the most resources
  - RAM
  - CPU
  - network
- identify system bottlenecks
- there are special tools
- allows to determine the most inefficient parts of the system

Why profile?

- you know which parts of the system have the greatest impact on performance
- you know what changes will bring the greatest benefits
- you avoid common mistakes
  - premature optimization
  - optimization of the bad parts of the system
  - too thorough optimization

Flat profiles problem

- usually to specify bottlenecks for the first time is quite simple
- after eliminating more difficult to see others
- common mistake is to rely solely on the time spent in the method
- it is also important:
  - how often the method is called
  - how much time is spent calling other methods

Flat profiles problem

Method	Time [ms]
m1	5
m2	2
m3	1
m4	1

Method	Total time [ms]	Time [ms]
m1	30	1
m2	24	2
m3	10	5
m4	3	1

Flat profiles problem

- Having only the execution time you often mistakenly refer to bottlenecks
- what to do?
  - speed up frequently used methods
  - less invokes of slow methods
- real life example - JTable in Swing
  - first optimization - leafs
  - Swing 1.1 - 2 times faster
  - lack of clear bottlenecks
  - JViewport problem when scrolling
    - problem was in JTable update
    - redesigned - 3 times faster

What do the tools give?

- typically allow you to find:
  - what methods are most often called
  - most time consuming methods
  - which methods call most commonly used methods
  - methods consuming most memory
- commercial tools have powerful GUI
  - different views
  - sophisticated statistics

Problems

- Too less memory - 3 kinds of memory:
  - heap
    - for arrays and objects
    - error message: *Java heap space*

when: during class instantiation

  - non-heap
    - stack
    - for addresses, variables, class definitions and strings
    - error message: *PermGen space*
    - when: eg. loading class
  - native
    - managed by OS, used by native methods
    - error message: *request <size> bytes for <reason>. Out of swap space?*
    - when: using native libraries

Problems

- memory leaks
  - GC does not clean up objects that have incidental references pointing to them
  - can lead to memeory leak
  - GC can work more often
- finalization
  - potential memory leak
  - object overriding finalize will not be removed before calling this method
  - it is uncertain when or whether the method will be called
  - in the end similar to memory leaks
- deadlock
  - synchronization is faulty
  - 2 or more threads are simultaneously waiting for monitor release

Problems

- looped threads
  - thread has infinite loop
  - uses more and more CPU time
  - can crash whole application
- too many locks
  - synchronization blocks threads
  - frequent waiting for monitor can starve thread
- application is too slow
  - identify bottlenecks
  - which methods cause most load?

Tools

- *jhat* - Java Heap Analysis Tool
  - analyses heap dump
  - preview in browser
  - has its own query language OQL
- *jconsole* - Java Monitoring and Management Console
  - GUI for monitoring and managing Java applications and JVM
  - works locally and remotely
  - shows memory, threads, classes, pending finalizations

Tools

- *jstat* - JVM Statistical Monitoring Tool
  - shows performance statistics
  - not supported - can be removed in future versions
- *jmap* - Memory Map
  - shows details about the stack
  - not supported
- *jstack* - Stack Trace
  - shows call stack for threads
  - useful when looking for deadlocks
  - not supported
- *java -Xhprof* - Heap/CPU profiling tool
  - since early versions of Java
  - heap and CPU usage by methods

Tools

- Java Visual Virtual Machine
  - bin\jvisualvm.exe
  - since Java 6 update 7
  - much better than *jconsole*
    - GUI
    - track many JVM at once
    - remote JVM tracking
    - profiling options

Profiling on Eclipse

- basic tools are often "raw"
- commercial - expensive
- there are free plugins for Eclipse that provide profilers with a convenient GUI
  - TPTP
  - profiles
    - Eclipse plugins
    - local Java applications
    - complex applications running on multiple machines and different platforms

Conclusions

- what is benchmarking?
- what is a benchmark?
- how differs macro from micro benchmark?
- what is profiling?
- what are the problems with flat profiles?