Správa paměti na platformě Java

Ladislav Thon

Obsah

1. závěr
2. JVM a paměť
3. sledování a ovlivňování
4. problémy a jejich řešení

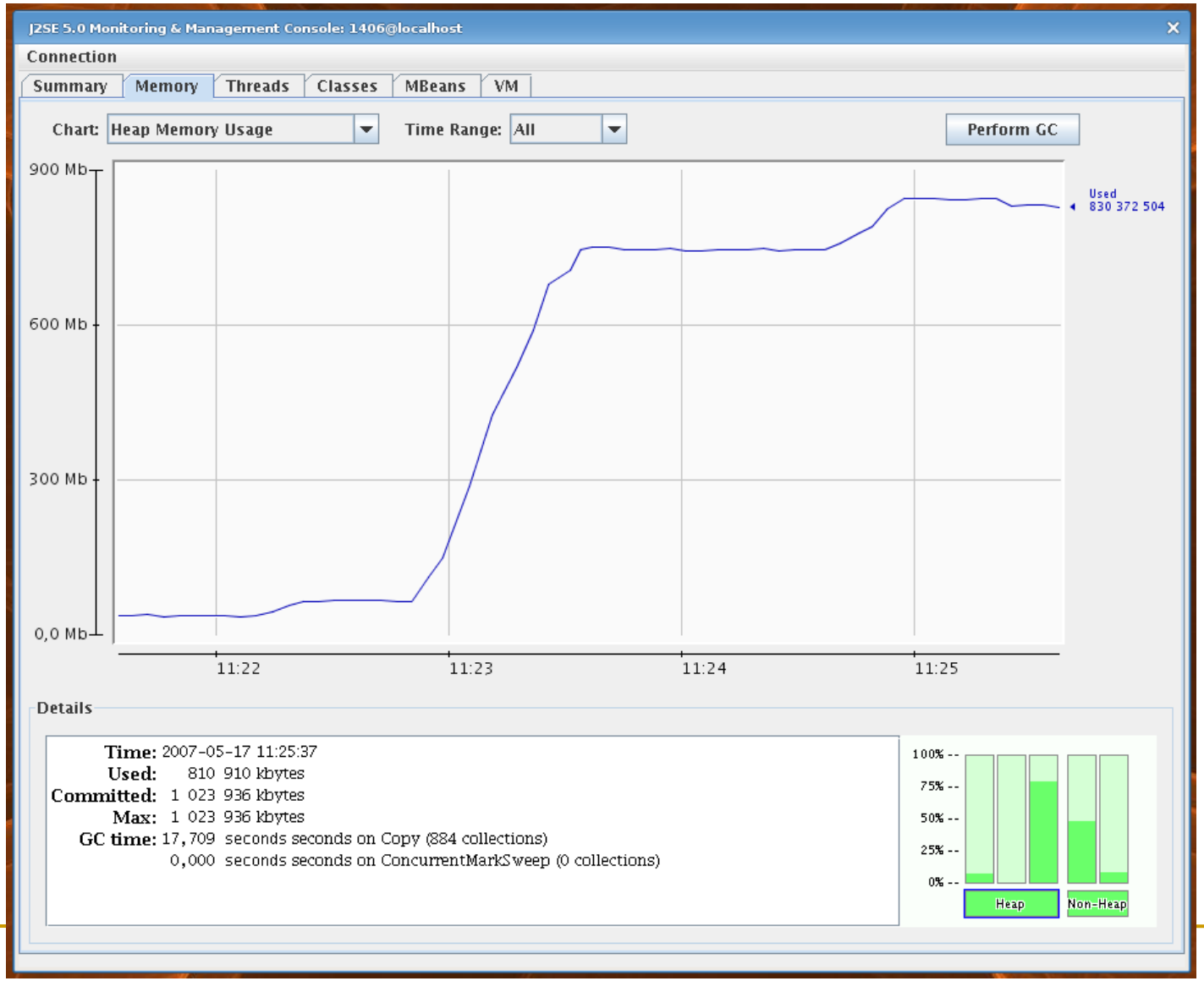
Upozornění: Sun JVM ≥ 5.0 only!

Ostatní implementace (IBM, Bea) se liší.

Není pravda, že v Javě se nemusíme starat o paměť!

Lépe: nemusíme se starat o paměť,
ale musíme se starat o objekty!

Jinak…


J2SE 5.0 Monitoring & Management Console: 1406@localhost
Connection
Summary
Memory
Threads
Classes
MBeans
VM
Chart: Heap Memory Usage
Time Range: All
Perform GC
900 Mb
600 Mb
300 Mb
0,0 Mb
Used
830 372 504
11:22
11:23
11:24
11:25
Details
Time: 2007-05-17 11:25:37
Used: 810 910 kbytes
Committed: 1 023 936 kbytes
Max: 1 023 936 kbytes
GC time: 17,709 seconds seconds on Copy (884 collections)
0,000 seconds seconds on ConcurrentMarkSweep (0 collections)
100%
75%
50%
25%
0%
Heap
Non-Heap

Struktura paměti I.

- JVM je *zásobníkový* (stack) stroj
  - každé vlákno jeden zásobník
  - každé volání metody jeden *aktivační záznam* (frame)
  - každý frame:
    - parametry metody
    - lokální proměnné
    - atd.

Struktura paměti II.

- pole a objekty se alokují na *heapu*
  - omezená velikost!
- referenční typy $\Rightarrow$ automatická správa paměti
  - alokace (konstruktor)
  - finalizace (žádné destruktory)
  - GC

Struktura paměti III.

- *Perm Space*
  - vždy platné objekty (permanent)
  - objekty dynamického typového systému
    - `java.lang.Class`
    - `java.lang.reflect.Method`
  - řetězcové konstanty
    - `java.lang.String.intern()`
- interní paměť JVM:
  - cache JIT překladače, …

Garbage collection I.

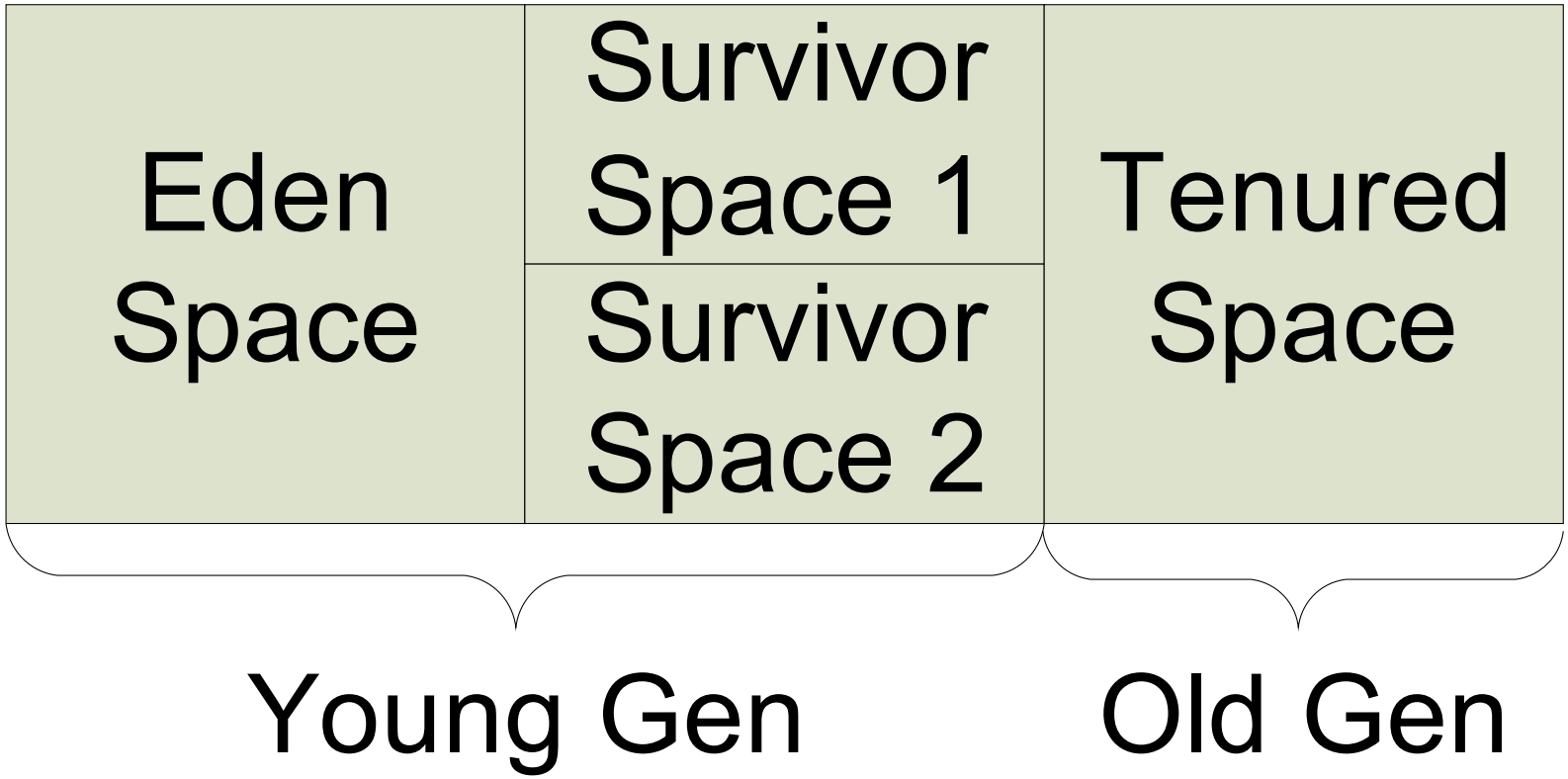
- odstraňování nepotřebných objektů z heapu
- GC hledá *dosažitelné* objekty
  - prohledávání z *kořenových* objektů
  - kořenové objekty – vždy dosažitelné:
    - reference na zásobníku
    - reference ve statických proměnných
- všechny ostatní může uvolnit
  - lokalita
  - fragmentace

Garbage collection II.

- řada různých přístupů:
  - (počítání referencí)
  - mark and sweep GC
  - kopírovací GC
  - generační princip
  - atd.
- JVM je kombinuje

GC v Javě I.

- Struktura heapu:



GC v Javě II.

- generační princip (2 generace)
  - většina objektů má krátkou životnost
  - objekty, které přežijí jistý počet GC, přesunuty do Tenured Space
- minor collection
  - pouze Young Gen
  - kopírovací algoritmus
- major (full) collection
  - pro Old Gen mark and sweep algoritmus

GC v Javě III.

- sériový GC (default):
  - všechna vlákna pozastavena, GC vykonáván jedním vláknem na jednom procesoru
- paralelní GC (default na serverech):
  - všechna vlákna pozastavena, GC Young Gen vykonáván více vlákny na více procesorech
  - pro Old Gen sériový GC
- lze paralelní GC i pro Old Gen

GC v Javě IV.

- konkurenční GC
  - pro Young Gen paralelní GC
  - pro Old Gen GC, který běží paralelně s běžnými vlákny aplikace
    - pozastavena pouze ve dvou místech algoritmu
    - musí začít dřív, než dojde paměť!
  - vlákno GC zabere jeden procesor jen pro sebe
- inkrementální GC (obsolete)
  - konkurenční GC, který se občas vzdá procesorového času ve prospěch vláken aplikace

Sledování správy paměti I.

- offline sledování – logy
  - `-verbose:gc` (`-Xloggc:<file>`)
  - `-XX:+PrintGCTimeStamps`
  - `-XX:+PrintGCDetails`
  - + další
  - `jinfo` – nastavení logování za běhu
    - od Javy 6
  - vizualizace z logů
    - http://www.tagtraum.com/gcviewer.html

Sledování správy paměti II.

- on-line sledování
  - konzolové utility
    - `jps` – seznam javovských procesů
    - `jstack` – seznam vláken + jejich stacktrace
    - `jmap` – informace o heapu (blokující!)
    - `jstat` – různé statistiky, zejm. o paměti a GC
  - grafické nástroje
    - `jconsole` – JMX konzole, mj. i sledování paměti
    - profilery
      - NetBeans Profiler, VisualVM, …
      - overhead

Ovlivňování správy paměti I.

- můžeme potřebovat:
  - změnit velikost dostupné paměti
    - vysoké nároky (algoritmy? cache?)
    - problémy (aplikace? knihovny?)
  - změnit chování GC
    - běžný GC ⇒ aplikace „zamrzne“
      - obvykle max. stovky milisekund
      - ale i 10 minut! (velký heap, swap)

Ovlivňování správy paměti II.

- základní parametry:
  - `-Xmx1000M` – maximální velikost heapu
  - `-Xms500M` – velikost heapu po spuštění
- dále:
  - `-Xmn100M` – velikost Young Gen
  - `-XX:MaxPermSize=200M` – velikost Perm Gen
- lze nastavit velikost všech oblastí paměti
  - i zásobníku

Ovlivňování správy paměti III.

- výběr GC:
  - `-XX:+UseParallelGC` – paralelní
  - `-XX:+UseParallelOldGC`
  - `-XX:+UseConcMarkSweepGC` – konkurenční
    - `-XX:+CMSPermGenSweepingEnabled`
    - `-XX:+CMSClassUnloadingEnabled`
    - `-XX:CMSInitiatingOccupancyFraction=80`
- detailní možnosti ladění chování GC

Paměťové problémy I.

```
class MyServlet extends HttpServlet {
  static List<HttpServletRequest> processedRequests
    = new ArrayList<HttpServletRequest>();

  protected void doGet(HttpServletRequest request,
      HttpServletResponse response) {
    ...
    processedRequests.add(request);
  }
}
```

To se stává i v lepších rodinách…

Paměťové problémy II.

```
Connection conn = ...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(...);
while (rs.next()) {
   ...
}
```

Rod Johnson: „Using JDBC directly should be a sackable offence.“

Paměťové problémy III.

- i v Javě existují memory leaky!
  - memory leak = alokovaná, ale již zbytečná paměť, která nikdy neuvolněna
  - v Javě spíše object leak
    - dosažitelný objekt, který již zbytečný
- obvyklé projevy:
  - `java.lang.OutOfMemoryError`
  - :-(
  - přesčas

Paměťové problémy IV.

- `OutOfMemoryError` – chybí stacktrace!
  - až od Javy 6
- heapdump
  - snímek paměti (heapu) v určitém okamžiku
  - při OOM:
    - `-XX:+HeapDumpOnOutOfMemoryError`
    - `-XX:HeapDumpPath=/home/ladicek/dumps`

Paměťové problémy V.

- heapdump
  - na vyžádání:
    - `-XX:+HeapDumpOnCtrlBreak`
      - podle zvěstí nejnovější revize Javy 1.4.2, 5 a 6
      - ale ještě jsem takovou neviděl :-?
    - `jmap -dump`, JMX (`jconsole`)
      - od Javy 6
    - lze získat i z coredumpu
      - `gcore`

Paměťové problémy VI.

- analýza heapdumpu
  - jhat
  - VisualVM
  - NetBeans Profiler
  - SAP Memory Analyzer
  - komerční profilery (?)

OutOfMemoryError: PermGen space

- class/classloader leak
  - typicky po undeploynutí aplikace v kontejneru
  - JDBC
    - registrace JDBC driveru ⇒ reference ze statické mapy v systémovém ClassLoaderu na třídu JDBC driveru v aplikačním ClassLoaderu
  - cache introspekce
    - `java.beans.Introspector.flushCaches()`
  - různé knihovny

Resumé I.

- pozor na statické kolekce!
  - platí i pro singletony
- pečlivě uvolňovat externí zdroje
  - nespoléhat na finalizaci
  - čím dřív, tím líp
- občas se hodí „slabé“ reference
  - `WeakHashMap` nebo plnohodnotná cache

Resumé II.

- povinně logovat činnost GC
- povinně generovat heapdump při OOM
- rozumný analyzátor heapdumpu při ruce
- při řešení problémů nastávají jiné problémy

Out Of Memory

dotázky?