JBoss Certified Technology Partner

Israel JBoss User Group

Session 01 / 16.3.2006

JBoss Tuning

By : Zvika Markfeld "Tikal"

Agenda

- JVM
- Web Container
- RMI
- Deployment
- Session Beans
- Connection Pools
- JNDI
- Transaction Manager
- Logging

JVM Tuning – GC Configuration

- We will look at the different garbage collector implementations available in Sun's JDK 1.3+ versions
- Generational and parallel garbage collection algorithms
- New in JDK 5.0: Ergonomics

How Garbage Collectors Work

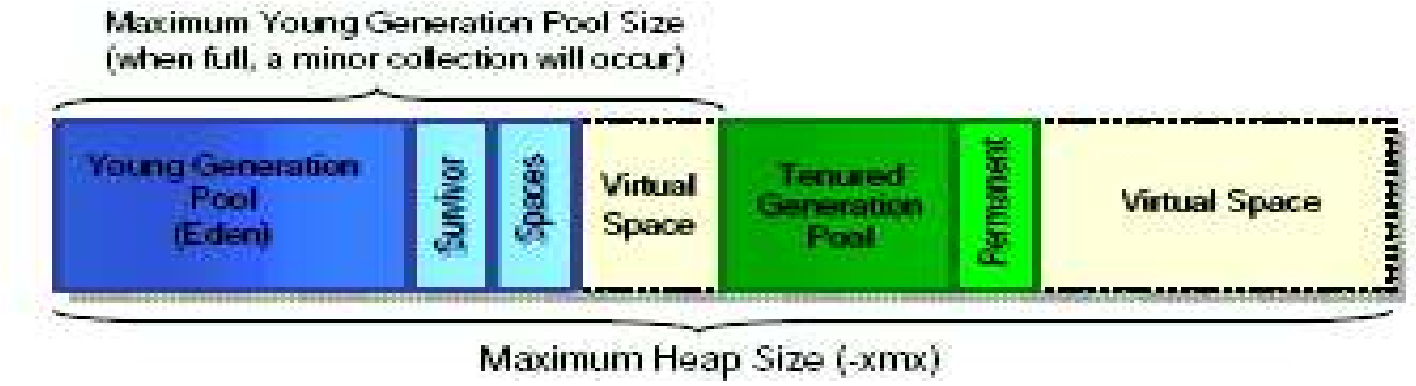
- *Garbage*: An object that cannot be reached from any pointer in a running program(ref-count=0)
- Simplistic garbage collection algorithm:
  - » Iterate over **all** reachable objects, marking them
  - » Any object left over is garbage, reclaim space

How Garbage Collectors Work

- Problems with simplistic garbage collection:
  - » Inefficient with large memory heaps
  - » Increased GC pauses
  - » Memory Fragmentation
- Hey! Aren't we forgetting something?
  - » Statistically, different objects types have different lifetime:
    - Some objects only live a few milliseconds at a time (local variables, iterators…)
    - Other objects may live days and weeks(data sources, EJB pools)

Generational Garbage Collector

- *Generational GC* maintains object pools based on object lifetime
- Uses different garbage collection algorithms for different pools
  - » Tune your GC based on the application profile
  - » Most applications have high “infant mortality”



Tuning Generational GC

- The default GC parameters are optimal for client applications
  - Most server applications are concerned with *Throughput*
    - The total time your JVM spends executing your application and not running the garbage collector
  - Client applications are concerned with *Pause Times*
    - How long is the noticeable pause the full garbage collection algorithms inflict
- For best throughput:
  - Minimize full garbage collections
  - Keep as many of your object instances in Eden

Eden

- Young Generation Pool
- Usually employs a *Minor Garbage Collection*
  - Uses *Fast Copy Collector*
  - Copied live objects to a *Survivor Space*
  - Removes dead objects from Eden
  - After minor collection, Eden is empty
  - Quick
  - Larger memory footprint (copying…)

Tenured Generation Pools

- Objects are moved from Eden to the *Tenured Generation Pool:*
  - » After surviving several minor collections
  - » When Eden fills up
- Employ a *Mark-Compact Collector*
  - » Objects are not copied, no need for extra memory allocation
  - » Compacting is much slower than copying
  - » Used for *Major Garbage Collections*
    - When reclaiming short lived objects does not free enough memory

Configuring Generational Segments

- **`-XX:NewRatio=<n>`**
  - » Sets ratio between Eden and tenured object pool
  - » Young pool is at maximum 1/(n+1) of total heap
- **`-XX:NewSize, -XX:MaxNewSize`**
  - » Sets a fixed size for Eden
  - » Finer granularity

Configuring Generational Segments

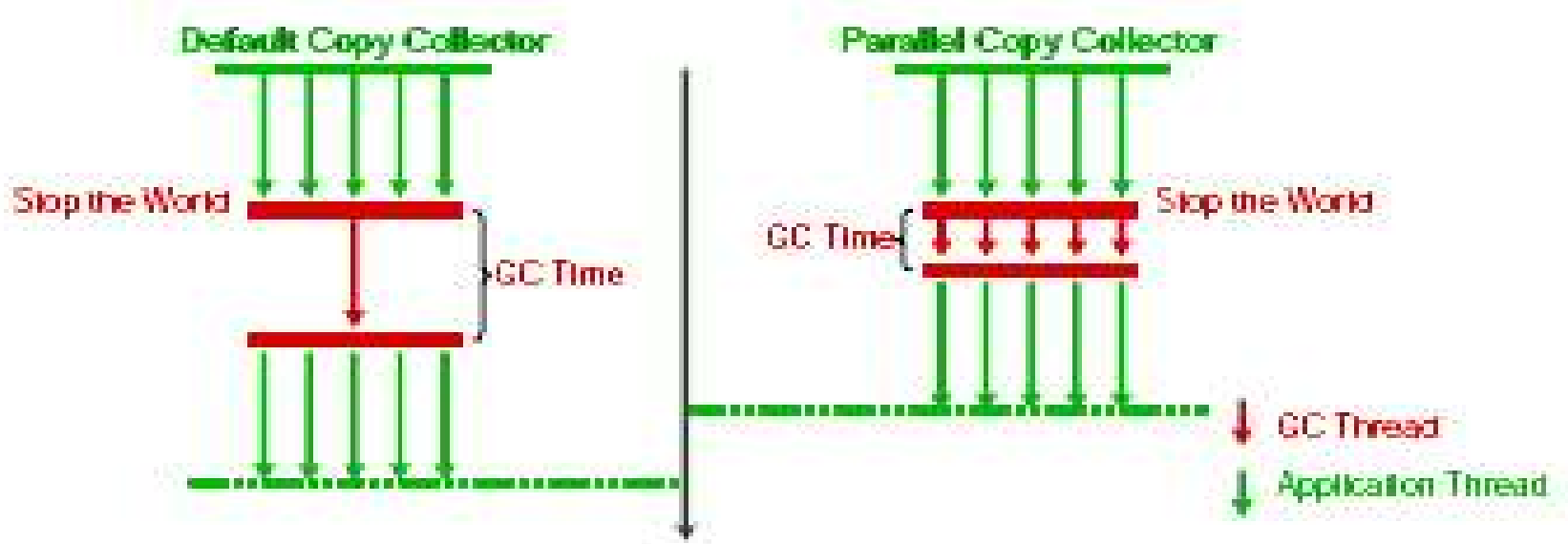
- Note:
  - The worst case scenario is a young generation pool full of live objects!
  - When tenured pool is full, a full GC will occur
- Therefore:
  - Setting Eden size to more than half the maximum heap size is usually counter productive

Parallel Garbage Collection

- Most JVMs implement stop-the-world GC algorithms
  - All running threads are halted
  - Garbage collection is then activated
- On Multi-CPU boxes, it makes sense to use all CPUs in sweeping the world clean
- JDK 1.4.2 introduced two new collectors for that purpose:
  - *Throughput Collector*
  - *Low Pause Collector*
- When running on a multiple CPU box, use JDK 1.4+

Throughput Collector

- Parallel Copy Collector
- Efficient for multi-CPU boxes
- Employs multiple threads to execute GC
- Addresses server-side applications



- ***-XX:+UseParallelGC, -XX:ParallelGCThreads=<n>***
  - » Enable the control of the number of GC threads

Concurrent Low Pause Collector

- *CMS:* Concurrent mark and sweep collector
- Collects tenured generations concurrently with application threads
- Addresses client-side applications

- `-XX:+UseConcMarkSweepGC`

Concurrent Low Pause Collector

- By default, the CMS implementation does not compact the tenured pool
- This might lead to a fragmentation problem, unless using: **`-XX:+UseCMSCompactAtFullCollection`**.
- Combine with parallel GC on young generation for maximum throughput
  - **`-XX:+UseParNewGCh`**

Other GC Considerations

- Calling **`System.gc()`** triggers a major collection, negating any benefits from minor collections
  - » Disable with **`-XX:+DisableExplicitGC`** option
- RMI DGC subsystem forces a major collection once a minute!
  - » Interval can be controlled with system properties:

```
-Dsun.rmi.dgc.client.gcInterval=3600000
-Dsun.rmi.dgc.server.gcInterval=3600000
```

Other GC Considerations

- Default thread stack size on Linux is too large…
  - Each thread allocates ~8 MB memory
  - Use: `Ulimit -s <nnnn>` to limit stack size
- Object pooling forces instances to enter the tenured pool, out of reach of the fast copy collector
  - Negates generational GC
  - Use Non-Shrinking pools only

Ergonomics in the JVM 5.0

- Automatic detection of server class machines
  - More than two CPUs
  - More than two GB of RAM
- Upon positive detection:
  - Throughput garbage collector enabled
  - Initial heap size of 1/64 of physical memory, up to 1GB
  - maximum heap size of ¼ of physical memory, up to 1GB
  - Server runtime compiler enabled

Ergonomics in the JVM 5.0

- *Behaviour Based Tuning*
  - **`-XX MaxGCPauseMillis=<nnn>`**
    Maximum pause time goal: a hint to the GC that pause times of <nnn> milliseconds or less are desired
  - **`-XX:GCTimeRatio=<nnn>`**
    Application throughput goal: the ration between the time spent on GC and the time spent on application
- The JVM guesses the rest of the parameters…

Web Containers – Connectors

- JBossWeb container can be found under: **`deploy/jbossweb-tomcatXXX.sar`**
- The connection configuration is found in **`server.xml`**
- *Connectors* are thread pools that accept inbound requests and process them

```
<!-- A HTTP/1.1 Connector on port 8080 -->
<Connector
        port="8080"
        address="${jboss.bind.address}"
        maxThreads="250"
        minSpareThreads="25"
        maxHttpHeaderSize="8192"
        emptySessionPath="true"
        enableLookups="false"
        redirectPort="8443"
        acceptCount="100"
        connectionTimeout="20000"
        disableUploadTimeout="true"/>
```

Web Containers – Connectors

- **`AcceptCount:`**
  Length of the incoming request queue(when there are no processors available)
- **`ConnectionTimeout`**:
  How long to wait before a URI is received from the stream (the default is 20 seconds)
  - » This avoid problems where a client opens a connection and does not send any data
- **`MaxThreads:`**
  The maximum number of threads in the pool, this is a strict pool.
  - » Rule of thumb: set to 25% more than your maximum expected load (concurrent hits coming in at once)

Web Containers – Connectors

- **`EnableLookups`:**
  Whether to perform a reverse DNS lookups to prevent snoofing
  - » Might cause problems when a DNS is ‘misbehaving’
  - » Turn off when you implicitly trust all clients
- **`MinSpareThreads`:**
  ”On start up, always keep at least this many threads waiting idle”
  - » Rule of thumb: Set to a little more than your normal load
- **`MaxSpareThreads`:**
  “If we ever go above minSpareThreads, always keep this number of threads waiting idle“
  - » Rule of thumb: Set to a little more than your peak load

Web Containers – Valves

- Remove any unnecessary valves and logging
  - For example, if not using JBoss security, remove the security valve

```
<!–
  <Valve className="org.apache.catalina.valves.RequestDumperValve" />

  <Valve
className="org.apache.catalina.valves.FastCommonAccessLogValve"
               prefix="localhost_access_log." suffix=".log"
               pattern="common"
directory="${jboss.server.home.dir}/log"
               resolveHosts="false" />
-->
```

Web Containers – JSP Optimization

- Jasper configuration found in: **`conf/web.xml`**
- Configured by default to **`development mode`**
- This means the jsp page might be checked for modification on *every* access

```
 <servlet>
      <servlet-name>jsp</servlet-name>
      <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-
class>
      <init-param>
         <param-name>fork</param-name>
         <param-value>false</param-value>
      </init-param>
      <init-param>
         <param-name>development</param-name>
         <param-value>false</param-value>
      </init-param>
      <init-param>
         <param-name>checkInterval</param-name>
         <param-value>60</param-value>
      </init-param>
      <load-on-startup>3</load-on-startup>
   </servlet>
```

Web Containers – JSP Optimization

- **`Development=false`:**
  Turns off checking for changes in JSPs
- **`CheckInterval`:**
  While in dev' mode, sets JSP check interval
- JSPs can also be precompiled to servlets to avoid delay on first access

Remote Invocations

- By default, JBoss creates a new thread for every remote invocation(RMI) request
  - Not efficient on a large systems
  - Inadvisable in the case of performance or traffic peeks
- Instead, use pooled invokers:
  - Edit **`conf/standardjboss.xml`**:
  - Change all of the proxy bindings to the pooled invoker by changing every XML fragment reading:

```
<!--
<invoker-
mbean>jboss:service=invoker,type=jrmp</invoker-mbean>
-->

<invoker-
mbean>jboss:service=invoker,type=pooled</invoker-mbean>
```

Deployment Scanner

- The deployment scanner scans for new deployments every 5 seconds
- This eats up cycles, especially on systems with a slow file system
- Edit **`conf/jboss-service.xml`**:

```
<!-- An mbean for hot deployment/undeployment of archives.
-->

<mbean
code="org.jboss.deployment.scanner.URLDeploymentScanner"
name="jboss.deployment:type=DeploymentScanner,flavor=URL">
...

<attribute name="ScanPeriod">5000</attribute>
...
</mbean>
```

Stateless Session Beans

- EJB stateless session beans operate according to a spec-dictated pooling model
- If you find that you need more than the default (10) instances, set minimum pool size:
  - Edit **`conf/standardjboss.xml`:**

```
<container-pool-conf>
  <MaximumSize>100</MaximumSize>
</container-pool-conf>
```

```
<container-pool-conf>
  <MinimumSize>100</MinimumSize>
  <MaximumSize>100</MaximumSize>
  <strictMaximumSize/>
  <strictTimeout>30000</strictTimeout>
</container-pool-conf>
```

- Usually, we wouldn't want these pools growing and shrinking due to memory-fragmentation
- From a performance standpoint, the number should be big enough to serve all your requests with no blocking

Datasource Configuration

- Use XA connections only when necessary
  - » XA connections have performance issues
- Use database specific "ping" support where available for "check-connection"
- Use database-specific driver fail-over support rather than checking connections at all
  - » Remember that not all tuning options may be available in your environment, we're talking optimal here…

JNDI

- It is a common mistake to configure a provider url on the server, as in:
  `java.naming.provider.url=jnp://localhost:1099`
- This forces JNDI to use local sockets for local access
- Inside the server, jndi should be done using internal method calls
  - This occurs when there is no provider url
  - On the client side, however, this forces a multicast discovery of HAJNDI
  - You may disable this behaviour by setting:
    `jnp.disableDiscovery=true`

Transaction Manager

- The transaction manager configuration can be found in: **`conf/jboss-service.xml`**
- The transaction timeout helps breaking blocked threads

```
<mbean code="org.jboss.tm.TransactionManagerService"
       name="jboss:service=TransactionManager"
       xmbean-dd="resource:xmdesc/TransactionManagerService-xmbean.xml">
       <attribute name="TransactionTimeout">300</attribute>
...
</mbean>
```

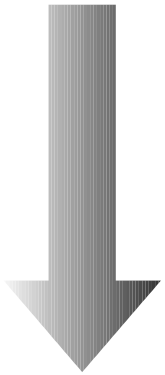
- **`TransactionTimeout:`**
  How long a transaction lasts before the thread gets interrupted and the transaction marked for rollback
  - » The default is 5 minutes

Log4J

- Logging has a profound effect on performance
  - » Setting logging level to TRACE can bring JBoss to a crawl
  - » Changing it to ERROR can improve speed dramatically
- By default, JBoss logs INFO-level messages to the console and to **`server.log`**
  - » Consider not logging to the console
  - » Consider changing the log level to ERROR / WARN
  - » Add a category filter for your Java class hierarchy
- NOTE:
  - » JBoss watches its Log4J configuration file for changes
  - » You can always change configuration at runtime

Log4j

```
<root>
  <appender-ref ref=CONSOLE"/>
  <appender-ref ref="FILE"/>
</root>
```



```
<root>
  <priority value="ERROR" />
  <appender-ref ref="FILE"/>
</root>
```

Q&A

Thank you

zvika@tikalk.com