Políticas de Coleta de Lixo Usadas pela JVM da Sun

Rodrigo Vieira Couto
MAC5765 – 2o semestre 2003

Agenda

- Arquitetura da JVM HotSpot™ da Sun
- Políticas de Coleta de Lixo
- Problemas e possíveis soluções
- Demo - *jvmstat*
- Referências

Arquitetura da JVM HotSpot™ da Sun

Detalhes gerais

- Segue a especificação *Java Virtual Machine Specification* [Lindholm & Yellin 1999]
- Atualmente na versão 1.4.2_02 (beta do 1.5 já está no ar)
- Alta performance devido:
  - Rápida sincronização de *threads*
  - Compilação adaptativa
  - Coleta de lixo que leva em conta a “idade” dos objetos (*Generational garbage collection*)

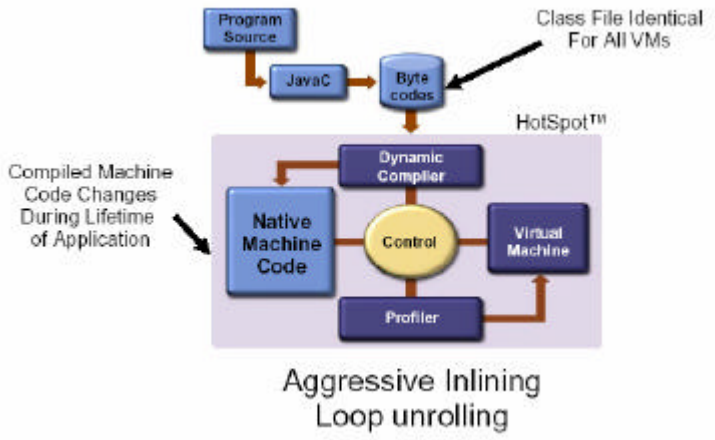
Características comuns

- Interpretação de *bytecode*
- Alocação de memória
- Coleta de lixo
- Sincronização de *threads*

Compilação adaptativa

- Compila *bytecode* para código nativo da máquina
- Ao contrário dos compiladores *Just In Time* (JIT), não compila todo código na inicialização
- Dois “sabores”:
  - client
  - server

Compilação adaptativa



Compilador -server

- *Aggressive Inlining*
  - Semelhante a usar modificador `inline`
  - Analisa *hot spots* do código em tempo de execução para tentar embutir código nativo de métodos a serem chamados
- Outras otimizações, comum de compiladores: detecção de código inutilizado, detecção de invariantes de loops, etc.

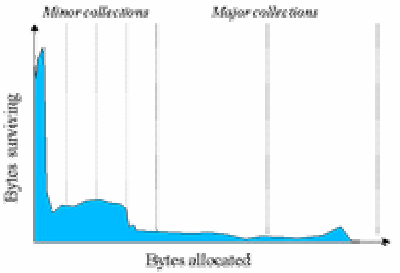
Algoritmos de Coleta de Lixo (CL)

Responsabilidades da CL

- Detecção de lixo
  - Distinguir objetos ativos do lixo
  - Contabilidade de referência entre objetos
  - Resolver problemas como referência cíclica
- Liberação de lixo
  - Tornar espaço livre para reutilização do programa em execução

Ciclo de vida de objetos

- Maioria dos objetos "vivem" muito pouco
  - 80-98% de todos os objetos recém alocados morrem em menos de alguns milhões de instruções (a 2.0GHz, ms)
- Grande impacto na escolha de algoritmos para CL

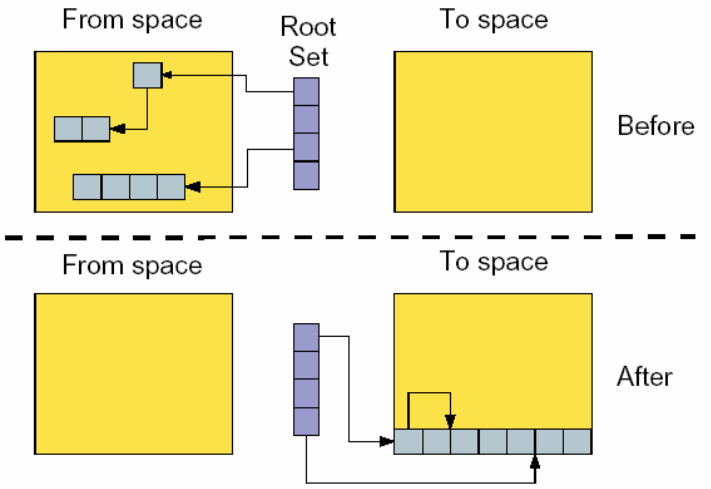


Solução: lidar com "gerações" de objetos

Algoritmos de CL

- Cópia
- Marcar e Varrer
- Marcar e Compactar
- Incremental
- *Generational*
- Cópia Paralela
- Concorrente
- Colheta Paralela

Cópia



Cópia (cont.)

- Pausa a execução do programa
- Muito eficiente
  - Varre a lista de objetos e copia de um semi-espaço a outro em uma única rodada
  - Detecção e liberação de lixo simultâneo
- Pausa diretamente proporcional ao número de objetos ativos
  - Semi-espaços maiores melhora eficiência
  - Menos freqüente, mais lixo acumulado

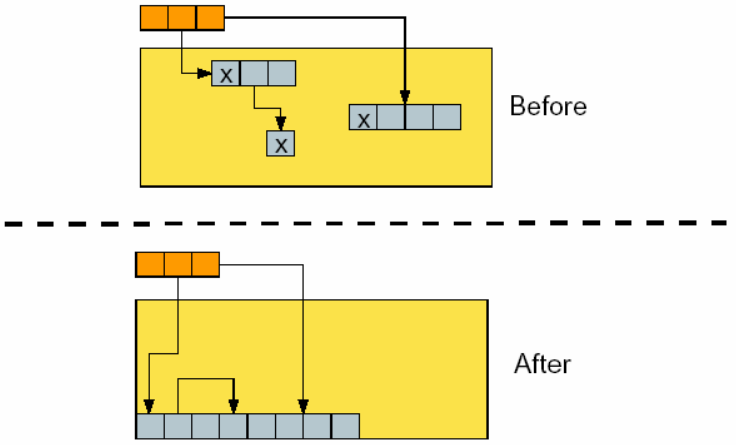
Marcar e Varrer

- Pausa a execução do programa
- Detecção
  - Varre o grafo de referências e marca objetos que podem ser atingidos
- Liberação
  - Varre o *heap* e limpa quem não estiver marcado
  - Cria lista de espaço livre

Marcar e Varrer (cont.)

- Problemas
  - Objetos de tamanhos diferentes causam fragmentação (cria-se listas de espaço livre para blocos de tamanhos distintos)
  - Custo da coleta proporcional ao tamanho do *heap*
  - Princípio da localidade furada: objetos velhos misturam-se com objetos novos

Marcar e Compactar



Marcar e Compactar (cont.)

- Elimina problemas de fragmentação
- Princípio da localidade garantida (ordenação dos objetos é mantida)
- Problema: requer múltiplas passadas
  - Marcar
  - Computar nova localização
  - Atualização dos ponteiros

Incremental

- Resolve o problema da pausa da execução (*heaps* grandes são problemáticos)
- Troca grandes pausas por muitas pequenas pausas
- Agrupa *clusters* de objetos que se referenciam mutualmente e coleta esses *clusters* individualmente
- Prejudica o throughput

`-Xincgc`

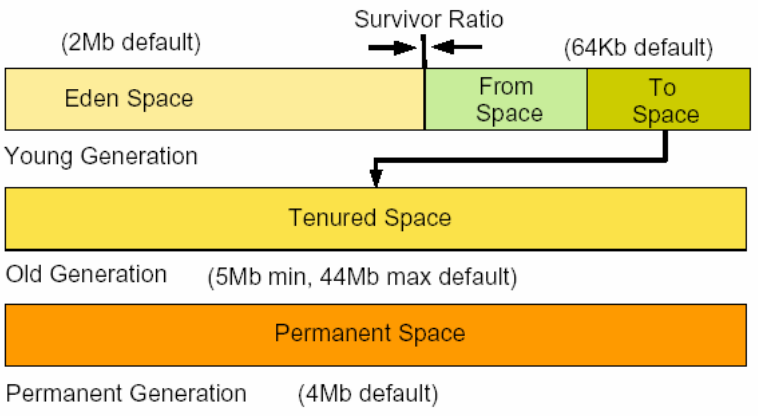
Generational - default

- Objetos antigos tendem a viver por muito tempo
  - CL pode ficar copiando e analisando mesmo objetos várias vezes
- CL *Generetional* divide o heap em múltiplas áreas (gerações)
  - Objetos são segregados por idade
  - Novos objetos morrem rapidamente, rode CL mais vezes
  - Gerações mais antigas são coletados menos vezes
  - Gerações diferentes usam diferentes algoritmos

Generational (cont.)

- Objetos são alocados no “eden”, como numa pilha, sem espaço entre eles
  - Alocação é rápida, contas simples de ponteiros
- Overflow!
  - Nesse ponto, maioria dos objetos já estão mortos
  - Coleta objetos vivos com CL Cópia e passa para outro semi-espaço
- Depois de `-XX:MaxTenuringThreshold` cópias entre semi-espaços de sobreviventes, passa para geração velha
- Objetos permanentes são coletados usando Marcar e Compactar, pois não há espaço adicional

Layout do Heap do HotSpot™



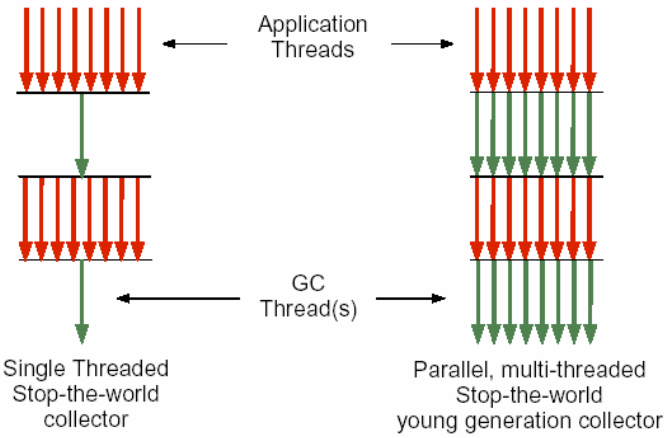
Desvendando mitos...

Tam. Total Máx. do *Heap*	**-Xmx**
Tam. Inicial do *Heap*	**-Xms**
Tam. Máx. do *Heap* da Ger. Nova	**-XX:MaxNewSize**
Tam. Inicial do *Heap* da Ger. Nova	**-XX:NewSize**
Tam. Ger. Nova / Tam. Ger. Velha	**-XX:NewRatio**
Sobreviventes / Eden	**-XX:SurvivorRatio**
Tam. Max. Ger. Permanente	**-XX:MaxPermSize**
Tam Inicial da Ger. Permanente	**-XX:PermSize**

CL para os “grandinhos”

- Cópia Parelela
- CL Concorrente
- Colheta Paralela

Cópia Paralela

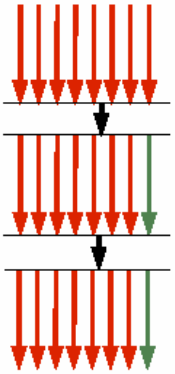


Cópia Paralela (cont.)

- Semelhante ao Cópia
  - Pausa a execução da aplicação
- Aloca o maior número de CPUs o possível
  - Otimizado para minimizar a pausa

```
-XX:+UseParNewGC
-XX:ParallelGCThreads=<num>
```

CL Concorrente



ApplicationThreads

Stop-the-world initial mark phase

Concurrent mark phase

Stop-the-world re-mark phase

Concurrent sweep phase

```
-XX:+UseConcMarkSweepGC
```

Colheta Paralela

- Pausa a execução: semelhante ao Cópia Paralela
- Otimizado para grandes Ger. Novas (12-80Gb)
- Escalável no núm. De CPUs
- Política de tamanhos adaptável

```
-XX:+UseParallelGC
-XX:ParallelGCThreads=<num>
-XX:+UseAdaptiveSizePolicy
```

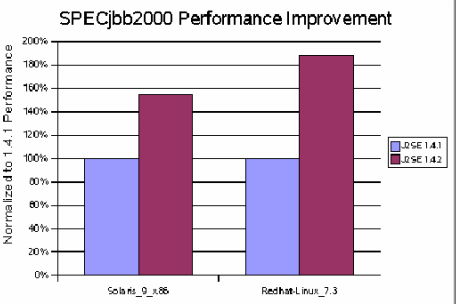
Problemas e possíveis soluções

Dicas iniciais

- *"Profile, profile, profile"* – Simon Ritter
- Observe:
  - Taxa de criação de novos objetos
  - Tempo de vida de objetos
  - Tipos de objetos: tamanhos e complexidade
  - Referências entre gerações

Dicas iniciais (cont.)

- *Rules of thumb*
  - sempre atualize sua máquina virtual
  - geração nova grande, geração velha pequena
  - evite usar swap



Changed implementation of **AggressiveHeap** option

Pausas muito longas?

- Tente usar da coleta incremental
- Tente aumentar tamanho do eden
  - Torna CL mais freqüentes e mais certeiras
  - Mais objetos mortos
  - Colocar o tamanho da geração nova para ? do total parece funcionar (segundo o FAQ)

Objetos vivendo demais?

- Fuce no código
  - Remova referências quando não mais necessárias
  - Pode-se fazer isso explicitamente por `object=null`
- Evite criar objetos em excesso

Ajudaria se...

- Criasse *pools* de objetos...
- Chamar `System.gc()` periodicamente...
- Preparar *loops* fáceis de serem desenrolados...

Não!

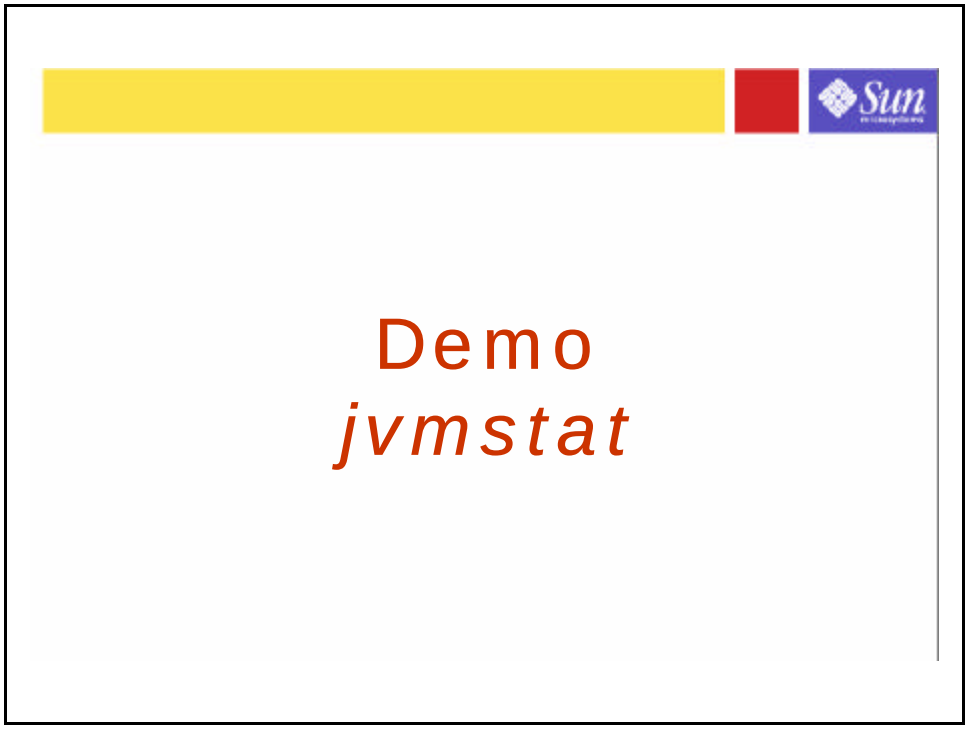
- ... *pools* contribuem para uma vida longa de objetos. *Pools* somente para recursos custosos (conexões, etc.)
- ... chamadas à CL apenas confundem os algoritmos do HotSpot™
- ... na boa, alguém se preocupa com isso ainda?

Configurações escondidas

- Configuração de defaults em `${JAVA_HOME}/jre/lib/i386/jvm.cfg`
- Visitar página java.sun.com/docs/hotspot/VMOptions.html
- Versão 1.4.1 possuía 224 opções de linha de comando

Só para quem pode...

- -XX:+UseAgressiveHeap
  - Mínimo de 256Mb
  - Tamanho total do *heap* ~ 3850Mb
  - Área de alocação de Thread: 256Mb
  - CL é adiada o máximo possível
  - Não compatível com –Xms e –Xmx
  - Não é recomendado para servidores não-dedicados


Demo
jvmstat


Referências

Referências

- **Java Plataform Performance – Strategies and Tatics** [Wilson & Kesselman 2001] - http://java.sun.com/docs/books/performance/1st_edition/html/JPTitle.fm.html - Dicas desde configurações da VM a dicas de programação para Swing
- **Faster Java Applications : How to Tune The HotSpot Virtual Machine** [Ritter – apresentação no Sun Tech Days 2003] - http://developers.sun.com/events/techdays/presentations/brazil.html
- **FAQ about the Java HotSpot Virtual Machine** - http://java.sun.com/docs/hotspot/PerformanceFAQ.html
- **Tuning Garbage Collection with the 1.3.1 Java Virtual Machine** - http://java.sun.com/docs/hotspot/gc/index.html
- **The Java HotSpot Virtual Machine , v1.4.1 – Technical Whitepaper** [2002] - http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_2.html
- **SPEC JVM98 Benchmarks** - http://www.spec.org/jvm98/ - Comparação entre as mais famosas JVM do mercado