Tópicos selecionados de programação em Java

Gerência de Memória em Java

Parte II: Monitoração e configuração da máquina virtual *HotSpot*

Helder da Rocha

Setembro 2005

Otimização da JVM HotSpot

- A HotSpot JVM permite a configuração de vários aspectos relacionados à gerência de memória
  - Escolha de JVMs pré-configuradas (servidor e cliente)
  - Ajustes absolutos e relativos do tamanho total e forma de distribuição do espaço do heap
  - Ajuste de tamanho da pilha (para todos os threads)
  - Escolha de diferentes algoritmos e estratégias de coleta de lixo
  - Metas para auto-ajuste (ergonomics) visando performance (throughput e baixas pausas)
  - Tratamento de referências fracas (soft references)
- Esta apresentação explora esses recursos e aponta estratégias de ajuste visando melhor performance

Assuntos abordados

- Tópicos
  1. HotSpot JVM: arquitetura de memória
  2. Parâmetros de configuração de memória
  3. Escolha do coletor de lixo
  4. Monitoração de aplicações
  5. Ajuste automático (Java 5.0 Ergonomics)
  6. Apêndice: class data sharing

Opções –XX da JVM HotSpot

- Vários parâmetros de configuração mostrados nesta apresentação usam opções **`–XX`** do interpretador Java

  `java -XX:+Opção1 -XX:Opção2=5 ... [+opções] pacote.Classe`

- As opções **`-X`** e **`–XX`** não são padronizados
  - Não fazem parte da especificação da JVM; podem mudar.
  - Diferem entre algumas implementações do HotSpot (ex: IBM)
- Há dois tipos de opções **`–XX`**
  - Valor booleano **`-XX:<+/-><nome>`**
  - Valor inteiro **`-XX:<nome>=<valor>`**
- Nas opções booleanas, o +/- serve para ligar/desligar uma opção

  `-XX:+Opcao` (liga uma opção que estava desligada por default)

  `-XX:-Opcao` (desliga uma opção que estava ligada por default)
- Opções inteiras recebem o valor diretamente

  `-XX:Valor=8`

1. Arquitetura da HotSpot JVM

- A máquina virtual é configurada para as situações mais comuns da plataforma usada
- Há duas opções básicas a escolher
  - Java HotSpot **Client** VM: minimiza tempo de início da aplicação e memória utilizada. Para iniciar a VM com esta opção, use: `java -client`
  - Java HotSpot **Server** VM (opcional): maximiza velocidade de execução da aplicação. Para iniciar a VM com esta opção, use `java -server`
- Tamanhos *default* do heap e gerações permanentes diferem nas duas opções
- O tipo de máquina virtual usada pode ser selecionada automaticamente, de acordo com a plataforma
  - Recurso do Java 5.0: JVM ergonomics: máquinas grandes multiprocessadas usam Server VM e demais usam Client VM

Todas as VM HotSpot possuem

- Compilador adaptativo
  - Aplicações são iniciadas usando o interpretador
  - Ao longo do processo o código é analisado para localizar gargalos de performance; trechos ineficientes são compilados e outras otimizações (ex: inlining) são realizadas
  - Server VM (opção –server) faz otimizações mais agressivas
- Alocação rápida de memória
- Liberação de memória automática
  - Destruição automática de objetos usando algoritmos eficientes de coleta de lixo adaptados ao ambiente usado
  - Default é coletor serial em –client e coletor paralelo de alta eficiência (*throughput*) em –server
- Sincronização de threads escalável

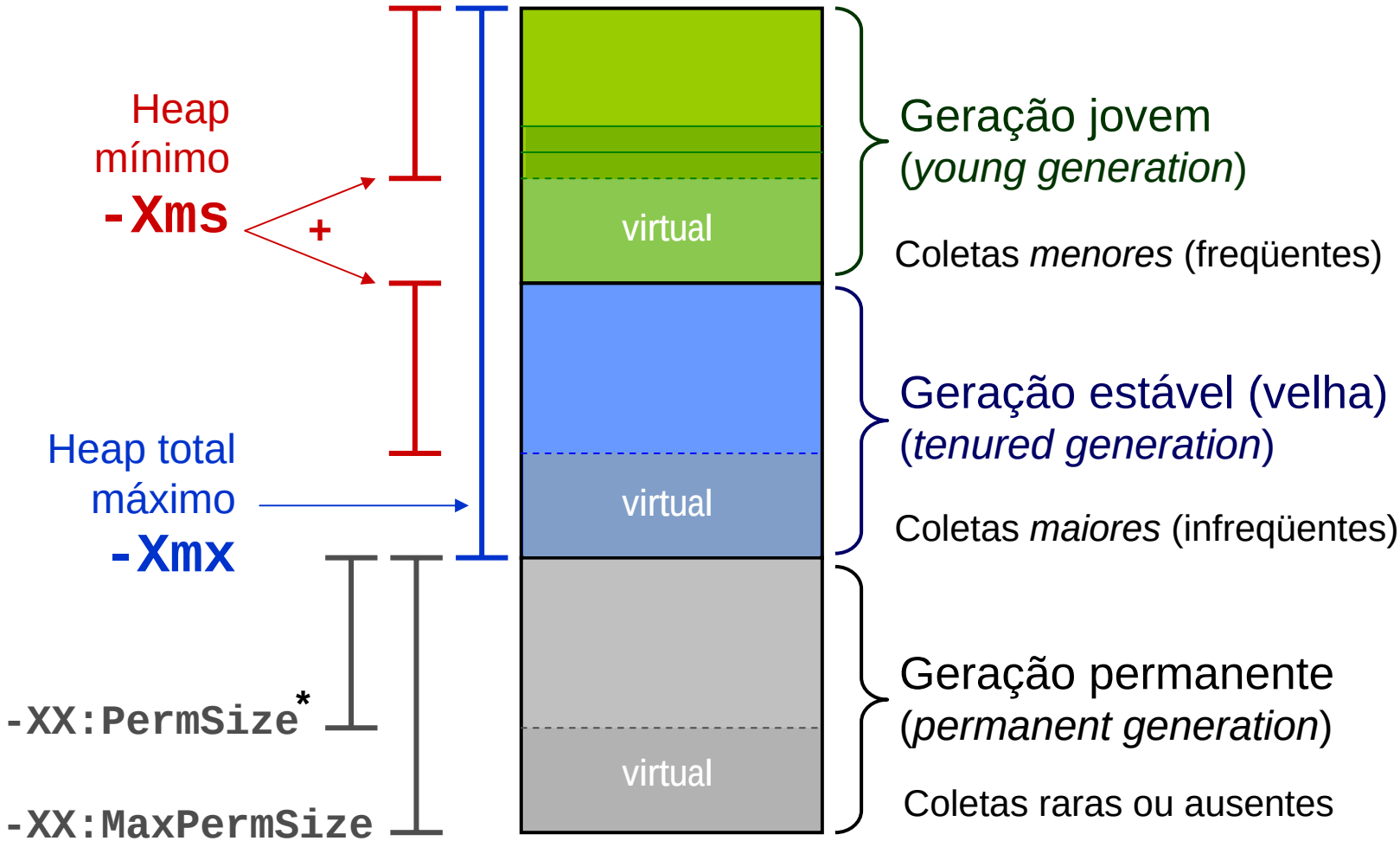
Coleta de lixo em Java: breve história

- Até a versão **1.1**: único coletor mark-and-sweep
  - Causa fragmentação de memória
  - Alto custo de alocação e liberação (o heap *inteiro* precisava ser varrido em cada coleta): pausas longas, *throughput* baixo
- HotSpot (Java **1.2** e posterior): generational collector
  - Geração jovem: algorítmo de cópia (garante que o espaço livre no heap seja sempre contíguo: alocação eficiente)
  - Geração antiga: algoritmo Mark-Compact (sem fragmentação)
- Java **1.3 a 1.5**: várias implementações
  - Diversas diferentes implementações de algoritmos baseados em gerações (serial, throughput, paralelo, concorrente, incremental)
  - **1.5**: auto-ajuste, escolha e otimização baseado em ergonômica
- Java **1.6** (Mustang) e Java **1.7** (Dolphin): ?

Coletor de lixo serial

- *Default* na HotSpot Client VM
- Heap dividido em gerações (generational GC)
  - Heap total contém geração jovem (três áreas) e geração estável (geralmente maior)
  - Geração permanente (sem coleta ou coletada raramente) alocada à parte do heap total
- Algoritmos de coleta de lixo usados
  - Geração jovem: algoritmo de cópia com duas origens e dois destinos (um temporário e um permanente): coletas pequenas e freqüentes: *minor collection*
  - Gerãção estável (velha): algoritmo mark-compact: coletas maiores (totais) e raras: *major collection*
  - Geração permanente: mark-sweep-compact: *coleta muito rara*

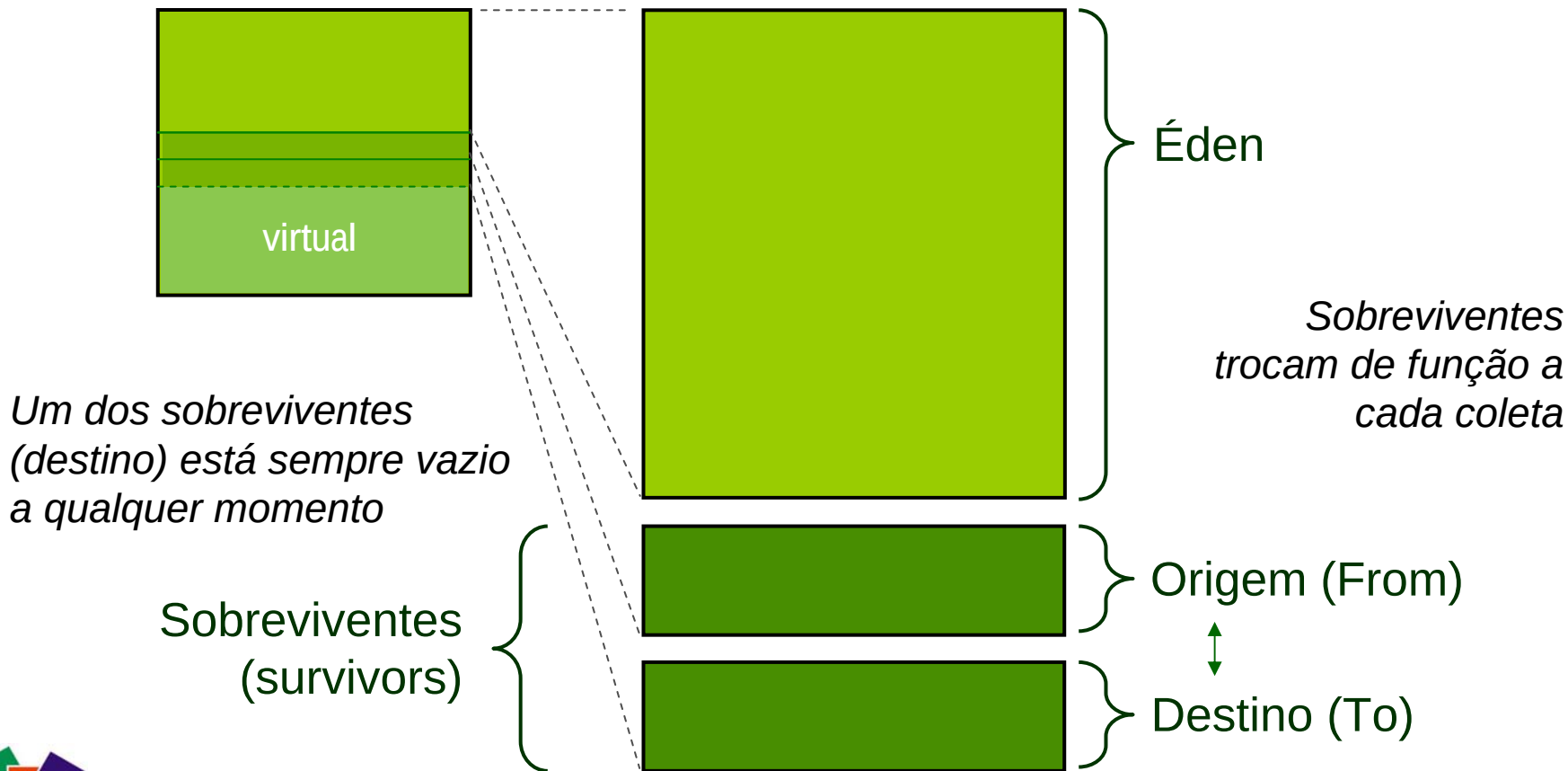
HotSpot: arquitetura do heap



* *opções –X e –XX: da JVM (da Sun) não são padronizadas e podem mudar no futuro*

Geração jovem

- Usa algoritmo de cópia (coleta menor)
  - Objetos são criados no Éden (sempre origem)
  - Sobreviventes são copiados para áreas menores



Coleta menor

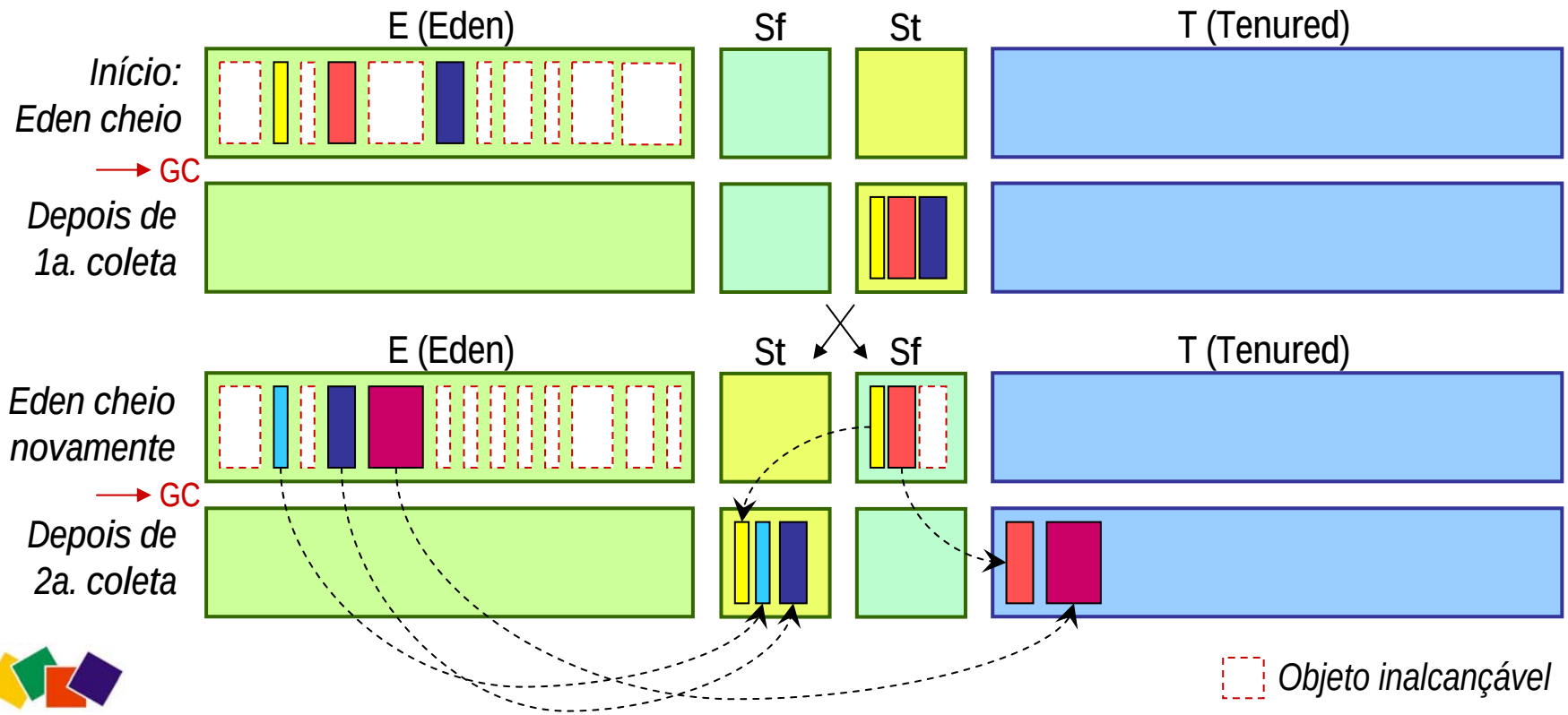
- *Minor collection* (Partial GC)
- Freqüentes e rápidas
  - Acontecem sempre que o Eden enche
- Usam algoritmo de cópia (copying algorithm) com duas áreas de origem (from_space) e duas áreas de destino (to_space)
  - Áreas sobreviventes alternam função de origem e destino (uma sempre está vazia)
  - Área Eden é sempre origem;
  - Geração estável é sempre destino

Coletas menores: algoritmo

- Quando o GC executa uma coleta menor
  - Copia todos os objetos alcançáveis do Éden e sobrevivente origem para a área sobrevivente destino e/ou geração estável
  - Se objeto não couber no sobrevivente, vai para geração estável
  - O GC pode promover um objeto que já foi copiado várias vezes entre as regiões sobreviventes e torná-lo estável
- No final da coleta, Éden e área sobrevivente origem estão vazios
  - Origem muda de função e passa a ser destino
- Coletas seguintes copiam objetos entre sobreviventes ou para a geração estável (quando tiverem idade)
  - Um objeto nunca volta ao Éden
  - Objetos na geração estável nunca voltam à geração jovem

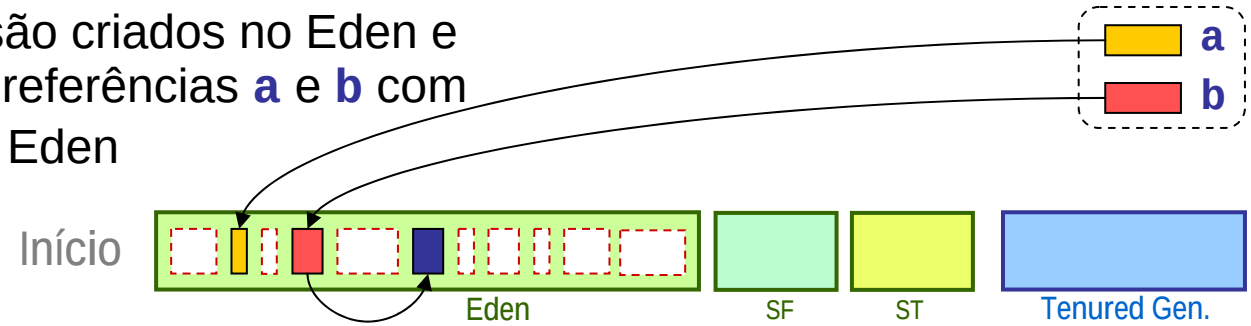
Coletas menores: exemplo

- Quando o Éden enche, GC copia objetos alcançáveis
  - Do **Éden** (**E**) para sobrevivente **To** (**St**) – sempre esvazia o Éden
  - Do sobrevivente **From** (**Sf**) para **St** – sempre esvazia o Sf
  - De **Sf** para a geração estável (**T**) (dependente de algoritmo)
  - Do **Éden** ou **Sf** para **T** (se não cabe em St)

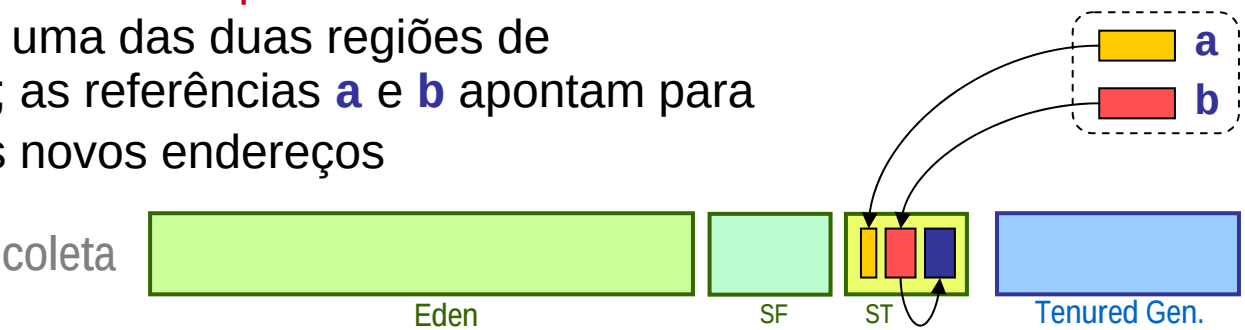


A dança das referências

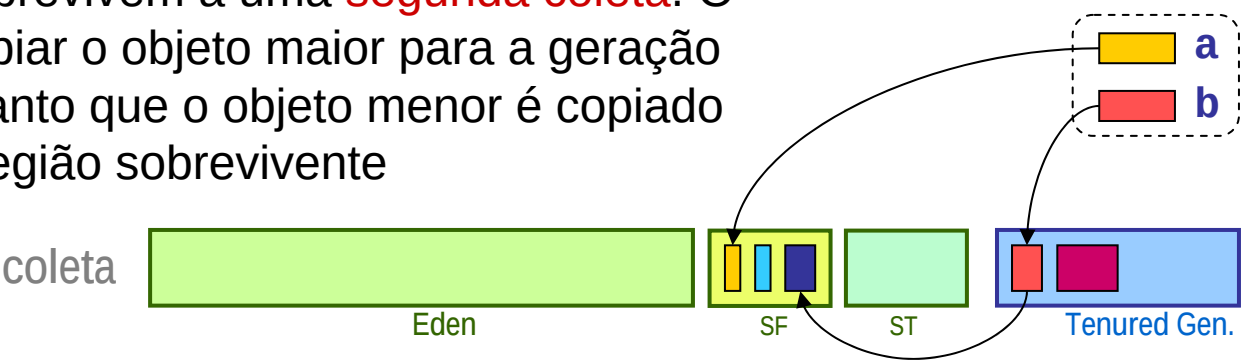
1. Dois objetos são criados no Eden e inicializam as referências **a** e **b** com endereços do Eden



2. Os objetos sobrevivem à primeira coleta e são copiados para uma das duas regiões de sobreviventes; as referências **a** e **b** apontam para os objetos nos novos endereços

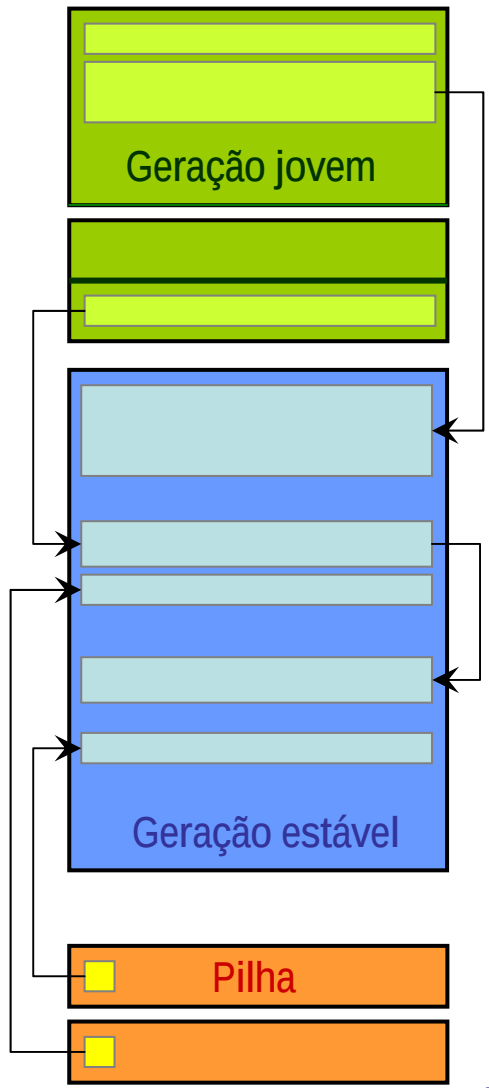


3. Os objetos sobrevivem a uma segunda coleta. O GC decide copiar o objeto maior para a geração estável, enquanto que o objeto menor é copiado para a outra região sobrevivente



Geração velha (estável)

- Consiste principalmente de objetos que sobreviveram a várias coletas menores
  - Objetos copiados várias vezes de um sobrevivente para a outro
  - Algoritmo de GC usado decide quando promover um objeto
  - Objetos jovens que recebem referências de objetos estáveis podem ser emancipados
- Um objeto muito grande que não cabe no na área Éden é criado diretamente na geração estável
- Pode estar sujeita à fragmentação
  - Resultado do algoritmo de GC usado (varia conforme o tipo de coletor de lixo escolhido)



Coleta maior (completa)

- *Major collection* (Full GC)
- Coletas menores gradualmente enchem a região estável
  - Quando a geração estável está cheia, o GC executa uma coleta maior envolvendo todos os objetos de todas as gerações
- A coleta maior (completa) pode acontecer antes, se
  - O algoritmo do coletor escolhido for incremental
  - Uma coleta menor não for possível devido à falta de espaço (sobreviventes estão cheios e há mais objetos ativos no Eden que caberiam na região estável)
- Usa algoritmo mark-sweep (MS) ou mark-compact (MC)
  - Depende do GC escolhido (concorrente, serial, paralelo, etc.)
  - Demora bem mais que uma coleta menor, porém é menos freqüente e pode nunca acontecer

Geração permanente

- Consiste de memória alocada por processos não-relacionados à criação de objetos
  - Carga de classes (ClassLoader)
  - Área de métodos (código compilado)
  - Classes geradas dinamicamente (ex: JSP)
  - Objetos nativos (JNI)
- Coletas de lixo são muito raras
  - Usa algoritmo Mark-Sweep (com compactação quando cheio)
  - Pode-se desligar a coleta de lixo nesta geração: -Xnoclassgc
  - Em MacOS X, existem uma geração “imortal”: parte da geração permanente compartilhada e não afetada por coleta de lixo
- Não faz parte do heap total controlado pelas opções –Xmx e –Xms da máquina virtual
  - Usa opções próprias –XX:MaxPermSize

3. Configuração de memória

- Há várias opções para configurar o tamanho das gerações, do heap total e da gerãção permanente
  - Também é possível determinar o tamanho das pilhas de cada thread
- Os ajustes pode ser realizados
  - De forma absoluta, com valores em bytes
  - De forma relativa, com percentagens ou relações de proporcionalidade 1:n
  - De forma automática (usando ergonômica) baseada em metas de performance

Tamanho absoluto do heap total

- Heap total = geração jovem + geração estável
  - Não inclui geração permanente (configurada à parte)

  -Xmx<número>[k|m|g]
  - Tamanho *máximo* do heap total
  - *Default**: -client: 64MB; -server: ¼ memória física ou 1GB)

  -Xms<número>[k|m|g]
  - Tamanho *mínimo* do heap total
  - *Default**: -client: 4MB; -server: 1/64 memória física ou 1GB)
- Exemplos de uso

  java -Xmx256m -Xms128m ...
  - heap ocupará espaço entre 128 e 256 megabytes.

  java -Xmx256m -Xms256m ...
  - heap ocupará exatamente 256 megabytes (tamanho fixo). Evita que a JVM tenha que calcular se deve ou não aumentar o heap.

** Não dependa dos defaults: costumam variar entre plataformas, versões, fabricantes*

Tamanho da geração permanente

- A geração permanente (onde classes compiladas são guardadas) não faz parte do heap total
  - Pode ser necessário aumentá-la em situações em que há uso de reflexão e geração de classes (ex: aplicações EJB e JSP)

-XX:PermSize=<valor>[k,m,g]

  - Define o tamanho inicial da geração permanente

-XX:MaxPermSize=<valor>[k,m,g]

  - Define o tamanho máximo da geração permanente.
  - Caso o sistema precise de mais espaço que o permitido nesta opção, causará OutOfMemoryError.
- Exemplos

```
java -XXPermSize=32m -XX:MaxPermSize=64m ...
```

  - Aloca inicialmente 32 megabytes para a geração permanente, expansível até 64 megabytes

Geração jovem: tamanho absoluto

- Geração jovem menor causa coletas pequenas mais freqüentes
- Geração jovem maior é mais eficiente pois coletas serão mais raras
  - Mas se ocorrerem irão demorar mais, além de reduzirem o espaço para a geração velha (o que pode causar coletas demoradas freqüentes)
- Para alterar o tamanho da geração jovem, use as opções

  `-XX:NewSize=<valor>[k,m,g]`
  - Define o tamanho mínimo da geração jovem

  `-XX:MaxNewSize=<valor>[k,m,g]`
  - Define o tamanho máximo da geração jovem
- Exemplos de uso

  `java -XX:NewSize=64m -XX:NewSize=64m ...`
  - Define um tamanho fixo de 64 megabytes para a geração jovem

  `java -XX:NewSize=64m -XX:NewSize=64m ...`
  - Define mínimo de 64 MB e máximo de 128MB para a geração jovem

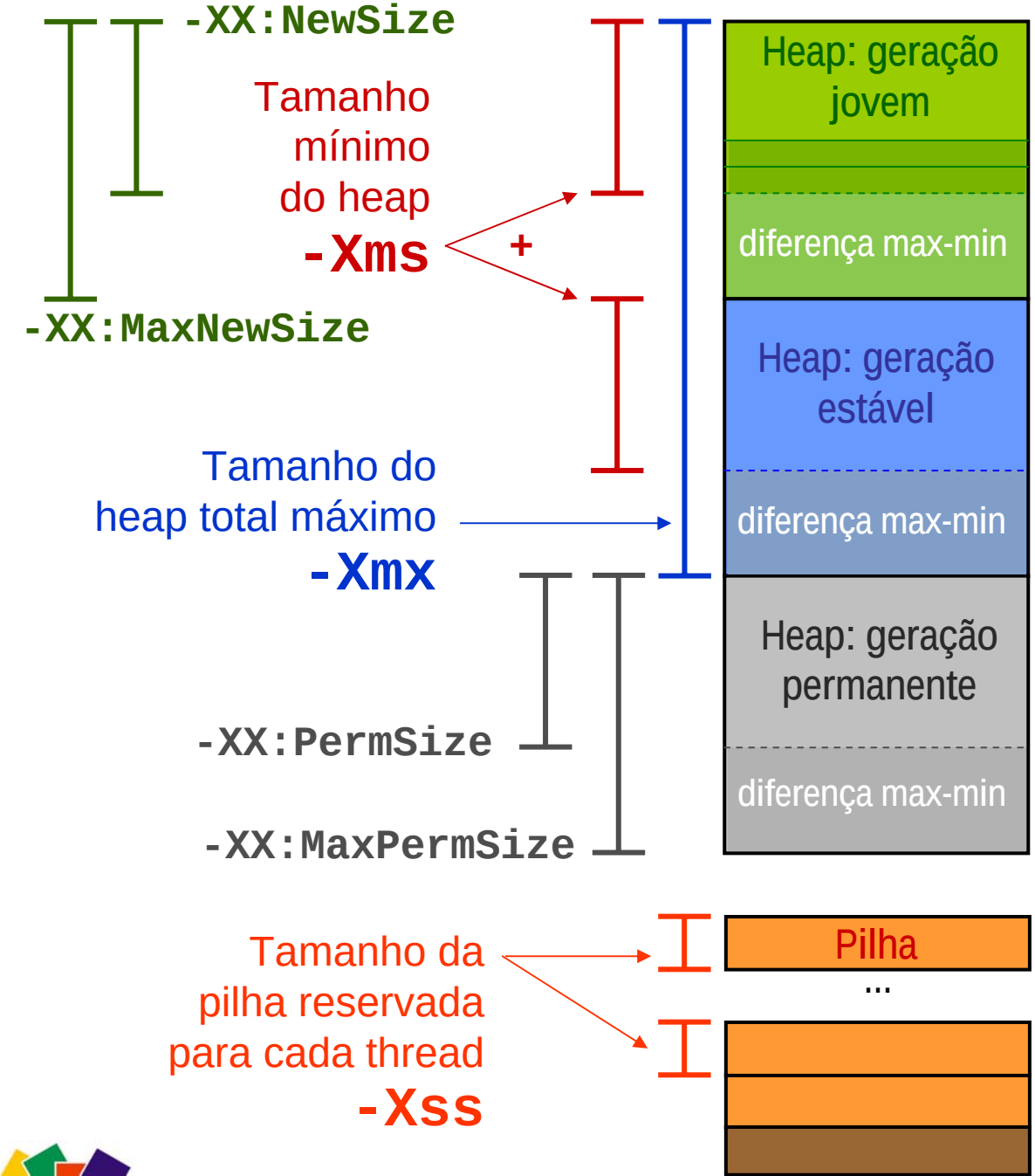
Tamanho da pilha de cada *thread*

- Cada *thread* tem uma pilha
  - A pilha é dividida em quadros (*frames*) para cada método cujos dados não são compartilhados
- Uma pilha grande evita StackOverflowError, porém se a aplicação tiver muitos *threads* (ex: servidores) a memória total pode ser consumida de forma ineficiente levando a OutOfMemoryError.
  - Reduza o tamanho da pilha, havendo muitos *threads*, e teste a ocorrência de erros StackOverflowError.
- O tamanho de cada pilha pode ser definido com a opção

  `-Xss=<valor>[k,m,g]`
  - Define o tamanho da pilha (de cada thread)
- Exemplos de uso

  `java -Xss128k ...`
  - Altera o tamanho da pilha de cada thread para 128 quilobytes

Resumo: ajustes de memória



Valores de ajuste absolutos

Tamanho relativo do heap real

- Esses parâmetros influenciam a JVM a aumentar ou diminuir o heap dentro da faixa -Xms/-Xmx
  - Se –Xms for igual a –Xmx eles não serão considerados

-XX:MinHeapFreeRatio=<percentagemMinima>

  - Define a percentagem mínima do heap que precisa estar disponível após uma coleta. Default* = 40% (Client JVM)
  - Se após uma coleta o heap disponível não corresponder a no mínimo esse valor, a JVM aumentará o espaço do heap proporcionalmente até alcançar a meta ou atingir o limite

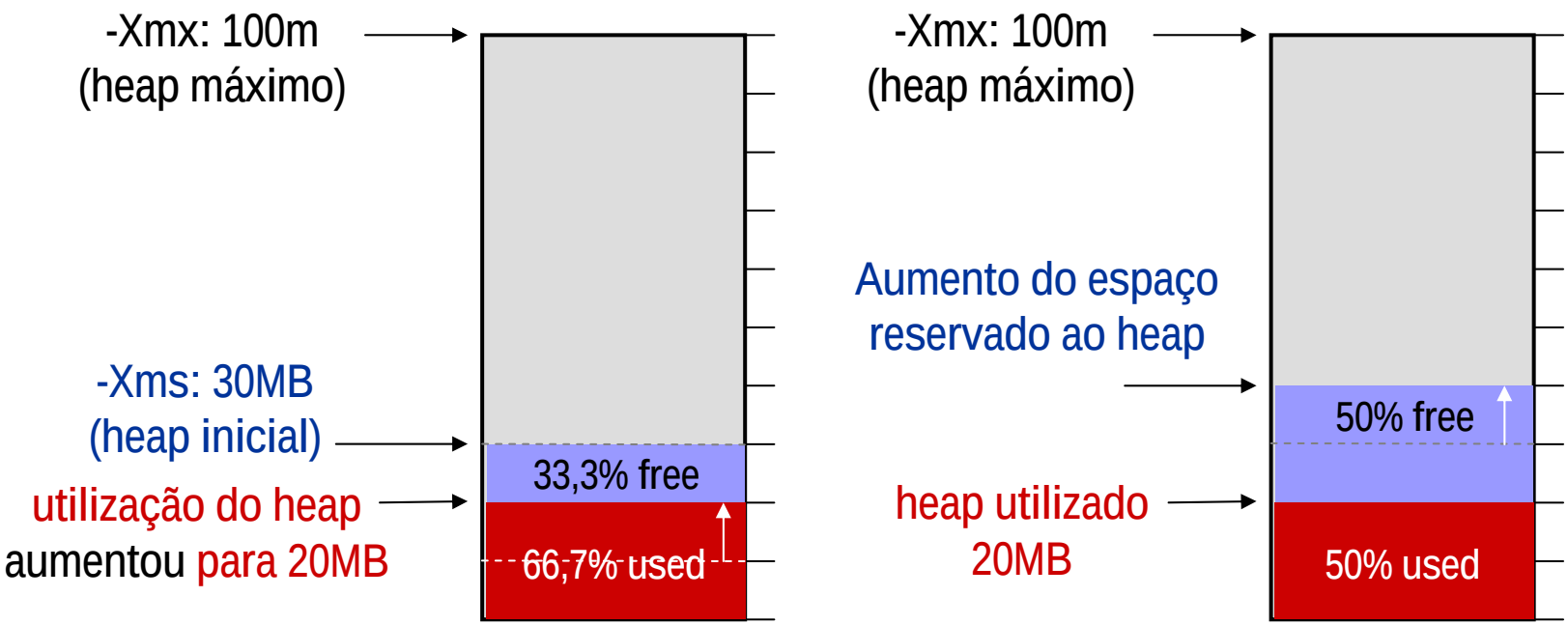
-XX:MaxHeapFreeRatio=<percentagemMaxima>

  - Define a percentagem máxima do heap que pode estar disponível após uma coleta. Default* = 70% (Client JVM)
  - Se após uma coleta o heap disponível for maior que este valor, a JVM irá reduzir o espaço do heap (reduz uso de memória) até alcançar a meta ou atingir o limite mínimo

** valores default variam entre plataformas/JVMs*

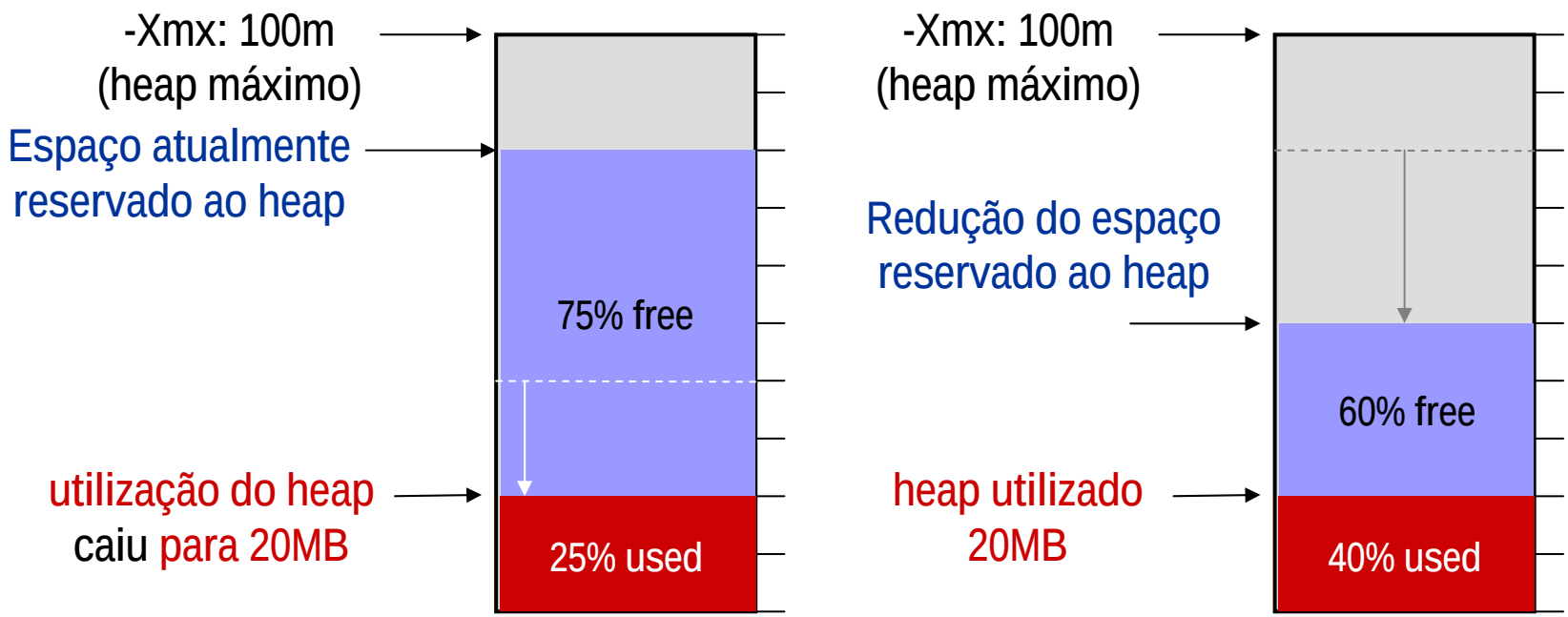
Exemplo: crescimento do heap

```
java -Xms30m -Xmx100m
     -XX:MinHeapFreeRatio=50
     -XX:MaxHeapFreeRatio=60 ...
```



Exemplo: redução do heap

```
java -Xms30m -Xmx100m
     -XX:MinHeapFreeRatio=50
     -XX:MaxHeapFreeRatio=60 ...
```



Conseqüências

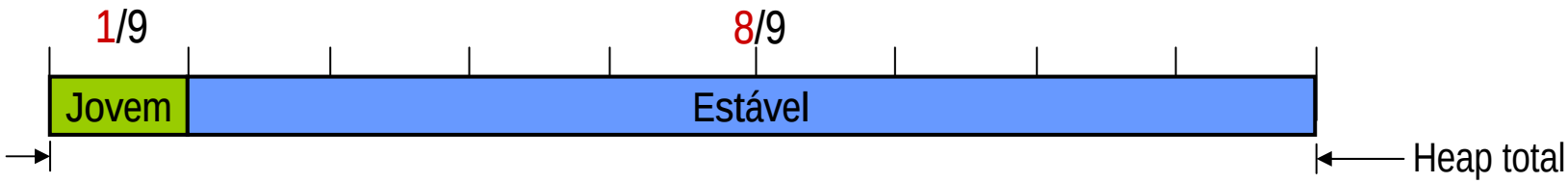
- O ajuste do tamanho do heap é o fator que tem o maior impacto na performance da coleta de lixo geral
  - Os valores atribuídos ao heap inicial e heap máximo são valores limite: a máquina virtual irá procurar utilizar a memória da forma mais eficiente, só ocupando o espaço realmente necessário
  - Aplicações que variam o heap com freqüência poderão melhorar a performance ajustando os parâmetros de redimensionamento para refletir melhor o comportamento da aplicação
- Pode-se definir -Xms and -Xmx para o mesmo valor, evitando o redimensionamento a cada coleta
  - Aumenta a previsibilidade da aplicação
  - Remove uma decisão da máquina virtual: não será preciso recalcular o uso do heap a cada coleta, mas a máquina virtual não poderá compensar se escolha for mal feita.

Proporção geração jovem/velha

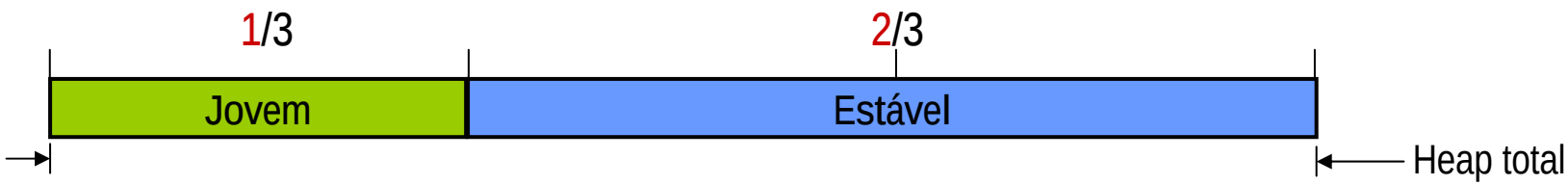
`-XX:NewRatio=`*`n`*

- Define a proporção 1:n entre a geração jovem e a geração velha. A geração jovem ocupará 1/(n+1) do espaço total do heap.

- Valores default dependem do servidor usado
  - No JVM Cliente* (opção -client) a relação é 1:**8** (NewRatio=8)



  - No JVM Servidor* (opção –server) a relação é 1:**2** (NewRatio=2)



- Exemplo de alteração

```
java -XX:NewRatio=3 ...
```

  - n é 3, então a relação é 1:3, ou seja, a geração velha será 4 vezes a geração jovem. A geração jovem ocupará 25% do heap.

** caso típico – valores default variam entre plataformas*

Garantia da geração jovem

- Young Generation Guarantee [Sun 05] (YGG)
  - Reserva prévia de espaço na geração velha (estável)
  - Não é realizada em coletores paralelos
- Para garantir uma coleta menor realizada com sucesso na hipótese em que todos os objetos estejam ativos, é preciso reservar memória livre suficiente na geração estável para acomodar todos os objetos.
  - O espaço poderá nunca ser usado: idealmente, os objetos serão copiados do Éden e sobrevivente origem para o sobrevivente destino
  - Mas não há garantia que todos os objetos caberão no sobrevivente.
- O espaço reservado pressupõe o pior caso: é igual ao tamanho do Éden mais os objetos no sobrevivente origem.
  - Não havendo como reservar o espaço, ocorrerá uma coleta completa.
- Devido à YGG, um Éden maior que metade do espaço do heap comprometido inutiliza as vantagens da Generational GC: apenas coletas maiores iriam ocorrer.

Conseqüências

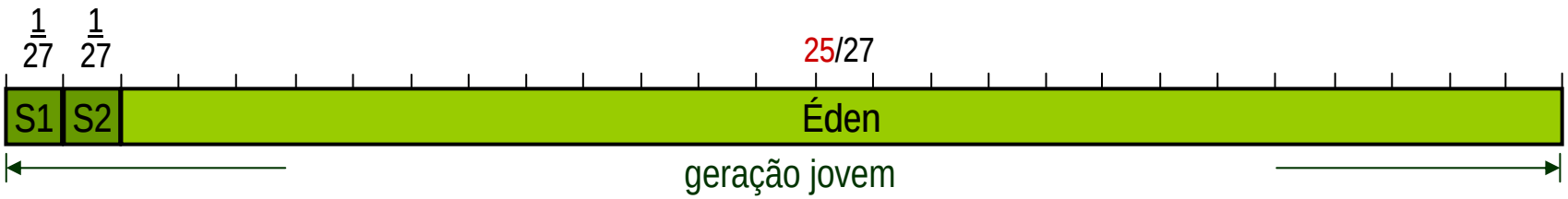
- Geração jovem maior causa
  - Menos coletas menores, porém com pausas maiores
  - Geração velha menor para um heap de tamanho limitado: aumentará a freqüência de coletas maiores (mais lentas – pausas maiores)
- Geração jovem menor causa
  - Maior freqüência de coletas menores, de pausas curtas
  - Geração velha maior, o que pode adiar coletas maiores (que, dependendo da aplicação, podem nunca ocorrer)
- Como escolher?
  - Analise a distribuição dos objetos alocados e estabilizados (tenured) durante a vida da aplicação. Não ajuste se não for necessário.
  - A menos que sejam detectados problemas com pausas longas ou muitas coletas maiores, aloque o máximo de memória à geração jovem
  - Veja se a garantia da geração jovem (YGG) é alcançada (não aumente além da metade do espaço usado do heap)
  - Alocação de objetos pode ocorrer em paralelo, então aumente à medida em que houver mais processadores.

Proporção Éden/sobreviventes

-XX:SurvivorRatio=*n*

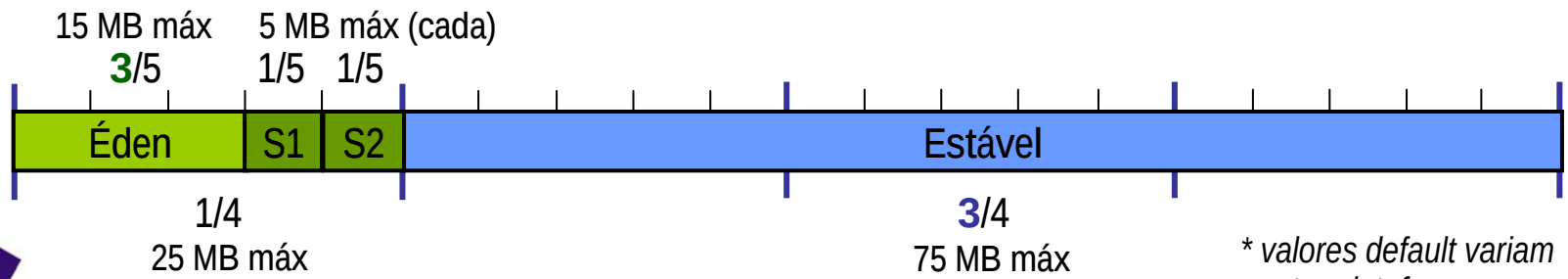
- Proporção entre espaços sobreviventes e o Éden. O número refere-se ao espaço ocupado pelos dois espaços sobreviventes.
- Uma relação 1:n reserva 1/(n+2) do espaço da geração jovem para cada sobrevivente

- *Default** é n=25 (Client JVM; cada sobrevivente ocupa 1/27 do espaço total da geração jovem); n=30 (Server)



- Exemplo

```
java -Xmx=100m -XX:NewRatio=3 -XX:SurvivorRatio=3
```



* *valores default variam entre plataformas*

Conseqüências

- Sobrevivente muito grande
  - Desperdício de espaço (um está sempre vazio)
  - Coletas mais freqüentes no Éden
- Sobrevivente muito pequeno
  - Enche muito rápido ou não cabe objetos, que são copiados diretamente para a geração velha
  - Enche geração velha mais rapidamente: +coletas maiores
- A cada coleta, a JVM define o número de vezes (*threshold*) que um objeto pode ser copiado (entre sobreviventes) antes de ser promovido à geração estável.
  - O objetivo é manter os sobreviventes cheios pela metade. Isto pode ser modificado com -XX:TargetSurvivorRatio (default=50)
  - A opção -XX:+PrintTenuringDistribution mostra esse valor e as idades dos objetos na geração estável (velha).
  - O valor máximo pode ser modificado com -XX:MaxTenuringThreshold (default=31); se for zero objetos são promovidos na primeira coleta

3. Seleção do coletor de lixo

Há quatro coletores pré-configurados no J2SE 5.0

1. Serial collector (*default em* -client, SGC)
   - Ative com a opção **`-XX:+UseSerialGC`**
2. Throughput collector (*default em* -server, TGC)
   - Ative com a opção **`-XX:+UseParallelGC`**
3. Mostly-concurrent low pause collector (CMS)
   - Ative com a opção **`-XX:+UseConcMarkSweepGC`**
4. Incremental (train) low pause collector (Train)
   - Ative com a opção **`-XX:+UseTrainGC`**

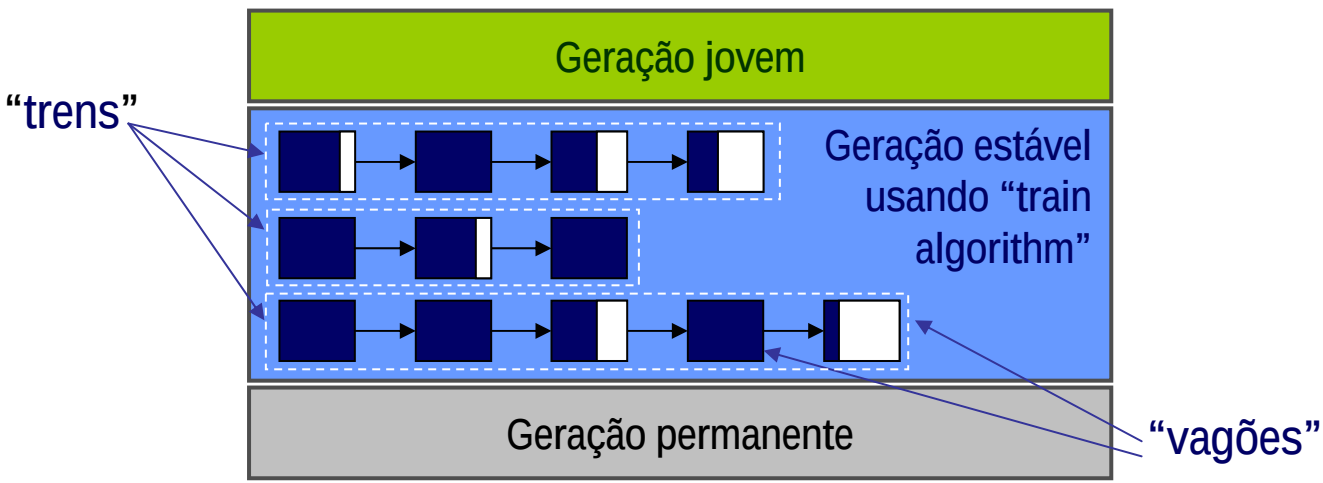
- Cada coletor usa uma combinação de algoritmos disponíveis otimizados para situações distintas
  - É possível configurar e combinar algoritmos diferentes

Algoritmos utilizados

- Algoritmos diferentes são usados para as diferentes gerações e em plataformas multiprocessadas
- Geração jovem
  - **(1)** Coletor serial (copying algorithm) - *default*
  - **(2)** Coletor paralelo (concurrent copying algorithm)
  - **(3)** Coletor paralelo de alta eficiência (scavenge)
- Geração estável
  - **(4)** Coletor mark-compact serial - *default*
  - **(5)** Coletor mark-sweep concorrente
  - **(6)** Coletor train incremental
- Coletores pré-configurados combinam algoritmos e permitem ajustes e alterações na configuração default

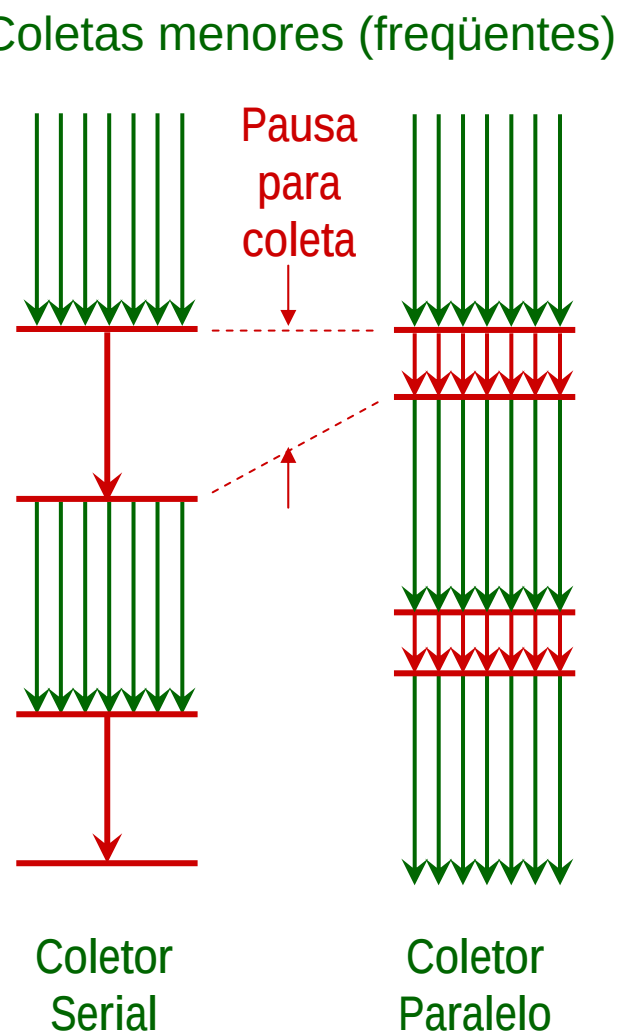
Coleta incremental (Train GC)

- Divide a velha geração em “vagões” (blocos de memória) e “trens” (conjuntos de blocos) e realiza coletas de alguns blocos sempre que possível, evitando parar a aplicação
  - Mas não é um algoritmo de tempo real, pois não é possível determinar um limite máximo de pausas, nem saber quando ocorrem, nem há como impedir que todos os *threads* parem ao mesmo tempo
  - Impõe alto custo sobre a performance: baixa eficiência (*throughput*)
- Para ativar use **`-XX:+UseTrainGC`** ou **`-Xincgc`**
  - Este coletor parou de ser atualizado na versão 1.4.2.



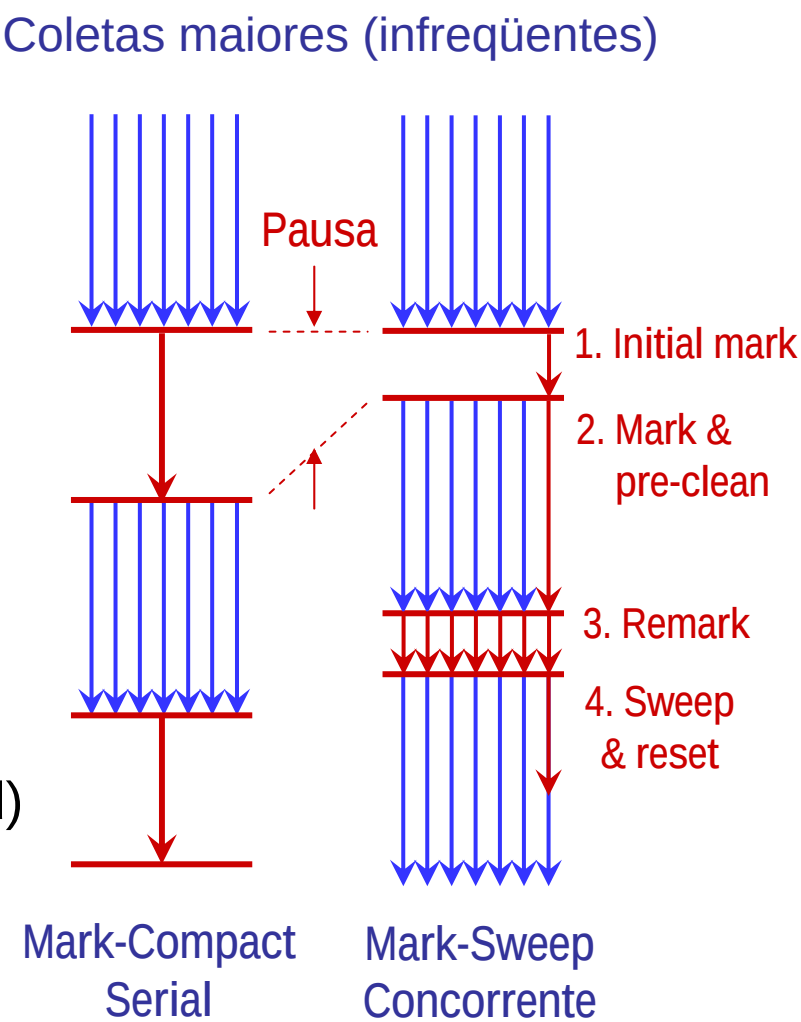
Coleta da geração **jovem**

- Os algoritmos são todos de cópia
- Todos param o mundo
- A principal diferença ocorre em sistemas multithreaded
  - No coletor *default*, todos os threads são interrompidos e um thread executa o algoritmo de cópia serial
  - Nos coletores paralelos, todos os threads são interrompidos e vários threads executam um algoritmo de cópia concorrente
- A pausa provocada em um coletor paralelo diminui com o aumento do número de processadores paralelos



Coleta da geração **estável**

- Fragmentação
  - Coletor serial mark-compact compacta o espaço
  - Coletor concorrente mark-sweep só compacta se espaço acabar (alocação – realizada durante coletas menores – será mais cara)
- Quatro etapas do coletor concorrente
  1. Initial mark (stop-the-world, um thread)
  2. Mark/pre-clean (paralelo, um thread)
  3. Remark (stop-the-world, um ou mais threads)
  4. Sweep/reset (paralelo, um thread)



Coletores pré-configurados

- Serial (-XX:+UseSerialGC*): ideal para sistemas monoprocessados
  - Geração jovem: coletor de cópia **(1)** (*default*)
  - Geração estável: coletor mark-compact **(4)** (*default*)
- Paralelo com eficiência máxima (-XX:+UseParallelGC)
  - Geração jovem: coletor de cópia concorrente de alta eficiência **(3)**
  - Geração estável: *default* **(4)** (mark-compact serial)
- Paralelo com pausas mínimas (-XX:+UseConcMarkSweepGC)
  - Geração jovem: *default* **(1)** (coletor de cópia serial); versão concorrente **(2)** *pode* ser ligada com -XX:+UseParNewGC
  - Geração estável: coletor mark-sweep concorrente **(5)** (com compactação apenas quando memória cheia)
- Incremental (-XX:+UseTrainGC)
  - Geração jovem: default **(1)** (cópia serial) ou -XX:+UseParNewGC **(2)**
  - Geração estável: algoritmo train incremental **(6)**

**Os parâmetros –XX:+Use*GC não devem ser combinados*

Opções de paralelismo

-XX:+UseParNewGC

- Com esta opção a JVM usará um coletor de cópia paralelo **(2)** para a geração jovem.
- Esta opção só pode ser usada com os coletores que não especificam um algoritmo para a geração jovem: –XX+:UseTrainGC ou XX:+UseConcMarkSweepGC

-XX:+CMSParallelRemarkEnabled

- Usada apenas no coletor CMS
- Com esta opção a remarcação (remark) é feita em paralelo, diminuindo as pausas.
- Requer o uso das opções -XX:+UseParNewGC e -XX:+UseConcMarkSweepGC

-XX:ParallelGCThreads=*n* (*Default*: no. threads disponíveis)

- Define o número de threads que serão usados pelo coletor paralelos da geração jovem

Agendamento de coleta

- Uma coleta concorrente deve iniciar e terminar antes que a geração estável fique cheia
  - Difere do coletor serial que inicia quando a geração enche
- Para saber quando iniciar, o coletor mantém estatísticas para estimar o tempo que falta antes da geração estável encher e o tempo necessário para realizar a coleta
  - As suposições são conservadoras
- Uma coleta concorrente também iniciará assim que a ocupação da geração estável passar de um certo limite
  - Este valor pode ser alterado com a opção

    `-XX:CMSInitiatingOccupancyFraction=`*`nn`

    onde nn é a % do espaço ocupado antes da coleta (0-100)
  - O valor inicial é aproximadamente 68% do heap.

Modo incremental: CMS

- Várias opções permitem diminuir pausas quando o CMS for usado, através do modo incremental.
- As principais opções são:

  `-XX:+CMSIncrementalMode` *(default: desabilitado)*
  - Habilita modo incremental

  `-XX:+CMSIncrementalPacing` *(default: desabilitado)*
  - Permite ajuste automático do ciclo com base em estatísticas
- **`-XX:CMSIncrementalDutyCycle=`*`n`*** *(default: 50)*
  - Percentagem de tempo (0-100) entre coletas menores em que o coletor concorrente pode executar.
  - Se *pacing* automático habilitado, este é o valor inicial.

  `-XX:CMSIncrementalDutyCycleMin=`*`n`* *(default: 10)*
  - Percentagem (0-100) limite inferior do ciclo se *pacing* habilitado
- Veja as várias outras opções do CMS na documentação

TGC vs. CMS

- TGC: Parallel Collector, a.k.a Throughput Collector
  - Objetivo: máxima eficiência com eventuais pausas
  - Aplicações que usam esse coletor raramente realizam coletas maiores; quando realizam, não têm problema se o sistema parar
  - Importante que coletas menores sejam rápidas (são sempre realizadas em paralelo) e eficiência a melhor possível
- CMS: Mostly-Concurrent Collector, a.k.a Concurrent Mark-Sweep (CMS) ou Low Latency Collector
  - Objetivo: mínimo de pausas com eventual perda de eficiência
  - Coletas maiores, apesar de pouco freqüentes, podem impor pausas muito longas (principalmente com heaps grandes)
  - Este coletor diminui as pausas da coleta maior, fazendo rode em paralelo com a aplicação principal (que fica mais lenta)
  - Duas pequenas pausas da mesma ordem das coletas menores (normalmente configuradas em paralelo também).

TGC vs. CMS

High throughput
-XX:+UseParallelGC

vs.

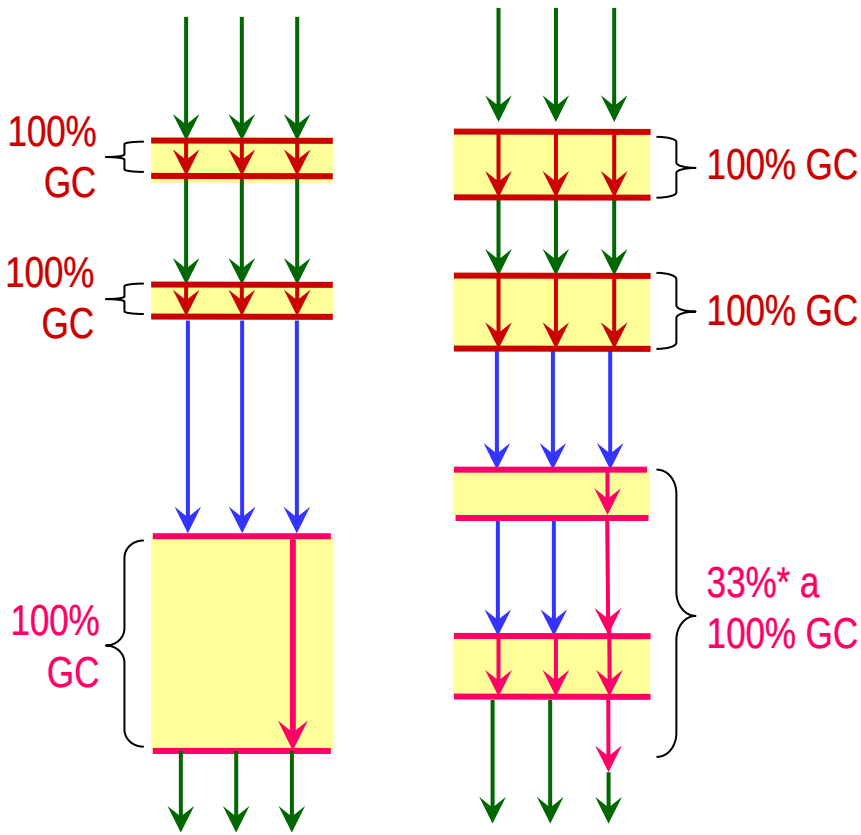
vs.

Low latency
-XX:+UseConcMarkSweepGC
com -XX:+UseParNewGC

Menos tempo total dedicado à coleta de lixo (eficiente)

Ocasional pausa longa (coleta na geração estável é serial e usa um único thread)

Alocação eficiente nas duas gerações sem fragmentação



Mais tempo total dedicado a GC (ineficiente)

Parcialmente incremental; pausas bem curtas

Pausas na nova geração mais longas (alocação na geração estável é mais cara devido à fragmentaçao)

** Neste exemplo, com 3 threads*

Quando a escolha de um coletor de lixo importa para o usuário?

- Para muitas aplicações ele não faz diferença
  - GC realiza pausas de pouca duração e freqüência.
- Mas em sistemas grandes, pode ser significativo
  - Compensa escolher o coletor de lixo correto e ajustá-lo
- Para maior parte das aplicações o SerialGC é adequado
  - Os outros têm *overhead* e são mais complexos
  - Se uma aplicação não precisa do comportamento especial de um coletor alternativo, deve usar o coletor serial
- Em grandes aplicações com muitos *threads*, muita memória e muitos processadores, o coletor serial provavelmente não será a melhor escolha
  - Neste caso, a escolha inicial deve ser o ParallelGC (TGC)

Quando usar

- SGC (SerialGC)
  - Aplicação típica, um processador, menos de 2GB de RAM
- TGC (ParallelGC)
  - Muitos threads alocando objetos (grande geração jovem, adiando ou evitando coleta estável)
  - Performance aumenta proporcionalmente ao número de processadores paralelos
  - Aplicação que tem que ser a mais eficiente possível
- CMS (ConcMarkSweepGC)
  - Pode compartilhar recursos do processador com o coletor de lixo enquanto a aplicação está executando
  - Muitos dados de vida longa (grande geração estável)
  - Requerimento de pausas mínimas (aplicações interativas)
- Train (TrainGC)
  - Requerimento de pausas mínimas; aceita eficiência baixa

4. Monitoração de aplicações

- Para ajustar os parâmetros configuráveis da máquina virtual, é preciso realizar medições
  - Vários parâmetros da máquina virtual HotSpot fornecem informações úteis
  - Ferramentas gráficas mostram o comportamento da máquina virtual e sua alocação/liberação de memória
- É preciso saber
  - O que ajustar e como ajustar
  - O objetivo do ajuste (menos pausas, mais eficiência)
  - As conseqüências do ajuste
- Pode-se também utilizar ajustes automáticos
  - Java 5.0 Ergonomics

Por que ajustar?

- Metas desejáveis: menos pausas e mais eficiência de processamento (*throughput*)
  - Melhorar uma pode piorar a outra
- Eficiência (capacidade de processamento)
  - É a percentagem de tempo total não gasta com coleta de lixo
  - Inclui tempo gasto com alocação
  - Se a *eficiência* for maior que 95%, geralmente não vale a pena fazer ajustes na JVM
- Pausas
  - Tempo em que uma aplicação parece não responder porque está realizando coleta de lixo

Informações sobre as coletas

- Pode-se obter informações sobre quando ocorrem coletas e como isto afeta a memória usando a opção

  `-verbose:gc`

  – Imprime informações básicas sobre as coletas (maiores e menores) de lixo para a saída padrão

  `-Xloggc:<arquivo>`

  – Usado com –verbose:gc grava a informação no arquivo especificado (importante para ferramentas de análise de logs)

- Exemplos de uso

  `java –verbose:gc aplicacao.Main`

  – Imprime informações de coleta na saída padrão

  ```
  java –verbose:gc
       –Xloggc:aplicacao.gc aplicacao.Main
  ```

  – Imprime informações de coleta no arquivo de texto aplicacao.gc

Saída de -verbose:gc

- Exemplo de saída de grande aplicação servidora

```
[GC 325407K->83000K(776768K), 0.2300771 secs]
[GC 325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628K->83769K(776768K), 1.8479984 secs]
```

- A saída mostra duas coletas menores e uma coleta maior
  - Os números antes e depois da seta (325.407K->83.000K) indicam o tamanho total de objetos alcançáveis antes e depois da coleta
  - Depois de pequenas coletas, a contagem inclui objetos que não estão necessariamente alcançáveis mas que não puderam ser coletados
  - O número entre parênteses (776.768K) é o total de espaço disponível (heap total usado menos um dos espaços de sobreviventes e sem contar o espaço da geração permanente)
- As coletas menores levaram em média 0,24 segundos
- A coleta maior levou quase dois segundos

Informações mais detalhadas

`-XX:+PrintGCDetails`

- Esta opção faz com que a VM imprima mais detalhes sobre a coleta de lixo, como variações sobre o tamanho das gerações após uma coleta.
- É útil para obter feedback sobre freqüência das coletas e para ajustar os tamanhos das gerações

- Exemplo de uso e resultado

```
java -XX:+PrintGCDetails
GC [DefNew: 64575K->959K(64576K), 0.0457646 secs]
196016K-133633K (261184K), 0.0459067 secs]]
```

- Não parece muito
  - Há ainda como obter mais detalhes...

Mais detalhes

- Tempo transcorrido e distribuição de objetos durante a aplicação

  -XX:+PrintGCTimeStamps

  – Imprime carimbos de tempo relativos ao início da aplicação.

  -XX:+PrintTenuringDistribution

  – Detalhes da distribuição de objetos transferidos para a área estável.

  – Pode ser usado para estimar as idades dos objetos que sobrevivem à geração jovem e para descrever a vida de uma aplicação.

- Exemplos de uso

```
java -verbose:gc -XX:+PrintGCDetails
  -XX:+PrintGCTimeStamps -XX:+PrintTenuringDistribution
```

- Saída

```
5.350: [GC Desired survivor size 32768 bytes,
        new threshold 1 (max 31)
       - age 1: 57984 bytes, 57984 total
       - age 2: 7552 bytes, 65536 total
       756K->455K(1984K), 0.0097436 secs]
```

Ferramentas: monitoração JMX

- O próprio J2SDK possui uma ferramenta simples que fornece informações gráficas sobre a memória usando JMX (Java Management Extensions)
  - Programável e extensível
- Para habilitar o agente JMX e configurar sua operação, é preciso definir algumas propriedades do sistema ao iniciar a máquina virtual
- As propriedades podem ser passadas em linha de comando da forma

```
> java –Dpropriedade
> java –Dpropriedade=valor
```

- Se um valor não for fornecido, a propriedade utilizará um valor default (se houver e se for aplicável)

Propriedades JMX da JVM

As duas principais propriedades são

- **`com.sun.management.jmxremote`**
  - Habilita o agente remoto JMX
  - Permite monitoração local através de um conector JMX usado pela ferramenta jconsole
  - Os valores podem ser *true* (default) ou *false*.
- **`com.sun.management.jmxremote.port=`***valor*
  - Habilita o agente remoto JMX
  - Permite monitoração remota através de um conector JMX de interface pública disponibilizada através de uma porta
  - O *valor* deve ser o número da porta
  - Esta opção pode requerer outras propriedades (veja tabela 1 em /docs/guide/management/agent.html (documentação J2SE 5.0*)

* [SDK]

Como habilitar o agente JMX para monitoração local

- 1) Execute a classe ou JAR da aplicação via JVM passando a propriedade **jmxremote**

```
> java –Dcom.sun.management.jmxremote pacote.MainClass
> java –Dcom.sun.management.jmxremote –jar Aplicacao.jar
```

- 2) Obtenha o número do processo JVM*

```
> jps
3560 Programa                    (máquina virtual)
3740 pacote.MainClass            (use este número!)
3996 Jps
```

- 3) Inicie o **jconsole** com o número do processo
  - Use o mesmo usuário que iniciou a aplicação

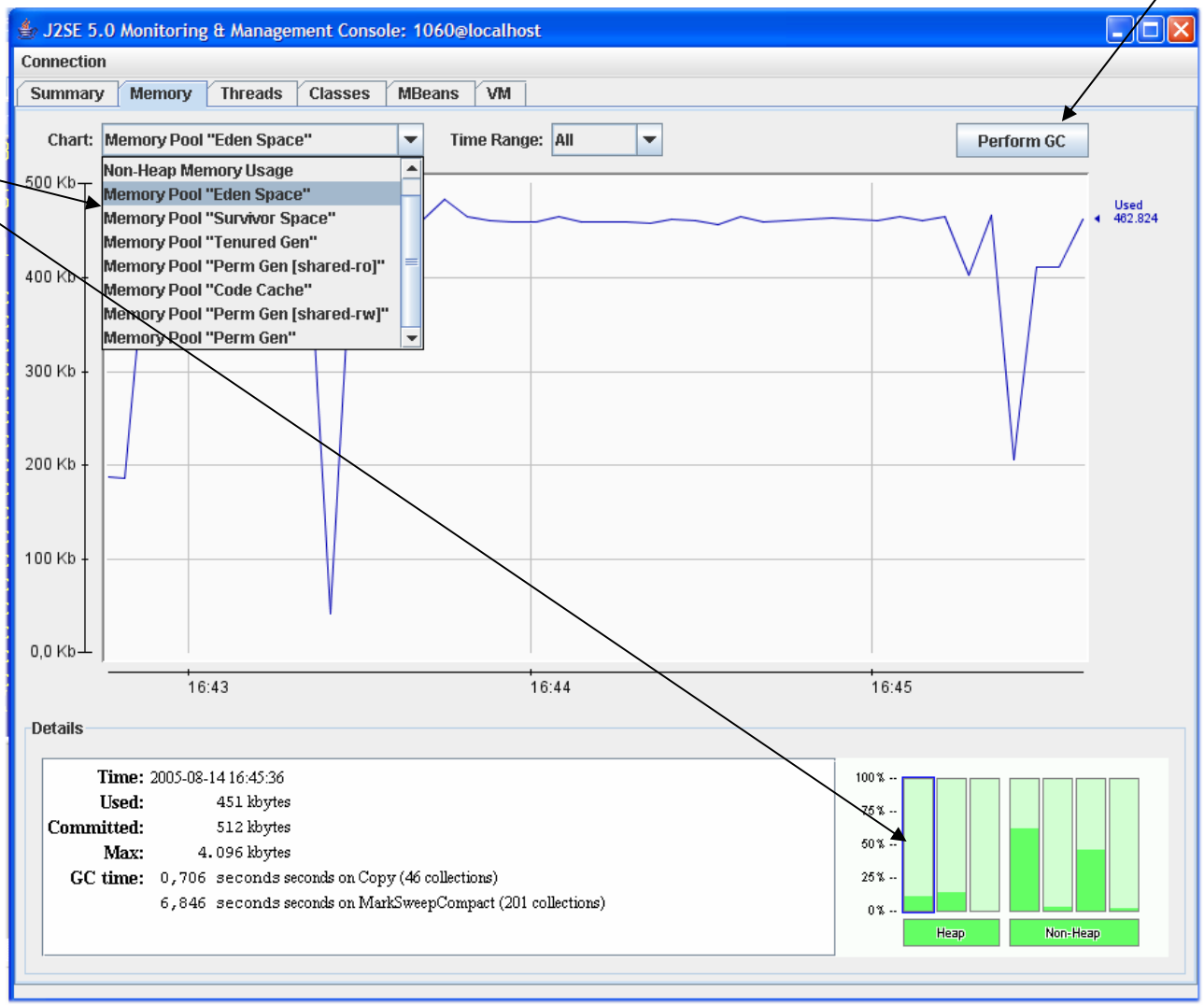
```
> jconsole 3740
```

** jps é uma das ferramentas do pacote experimental jvmstat 3.0, que faz parte do J2SDK 5.0*

jconsole: memória

Execute o Garbage Collector

Selecione a área do heap desejada



Monitoração remota

- Para monitoração em tempo de produção, recomenda-se uso remoto (devido ao *overhead* da aplicação)
- Para configurar, é preciso obter uma porta de rede livre
  - A porta será usada para configurar acesso remoto via RMI
  - O sistema também criará no registro RMI um nome jmxrmi
- Além da porta de acesso, é preciso configurar propriedades de autenticação, dentre outras
  - Para configuração do acesso remoto e outras informações, veja /docs/guide/management/agent.html (documentação J2SE 5.0*)
  - Também pode-se obter acesso programático
- Para executar o jconsole remotamente, informe o nome da máquina e a porta

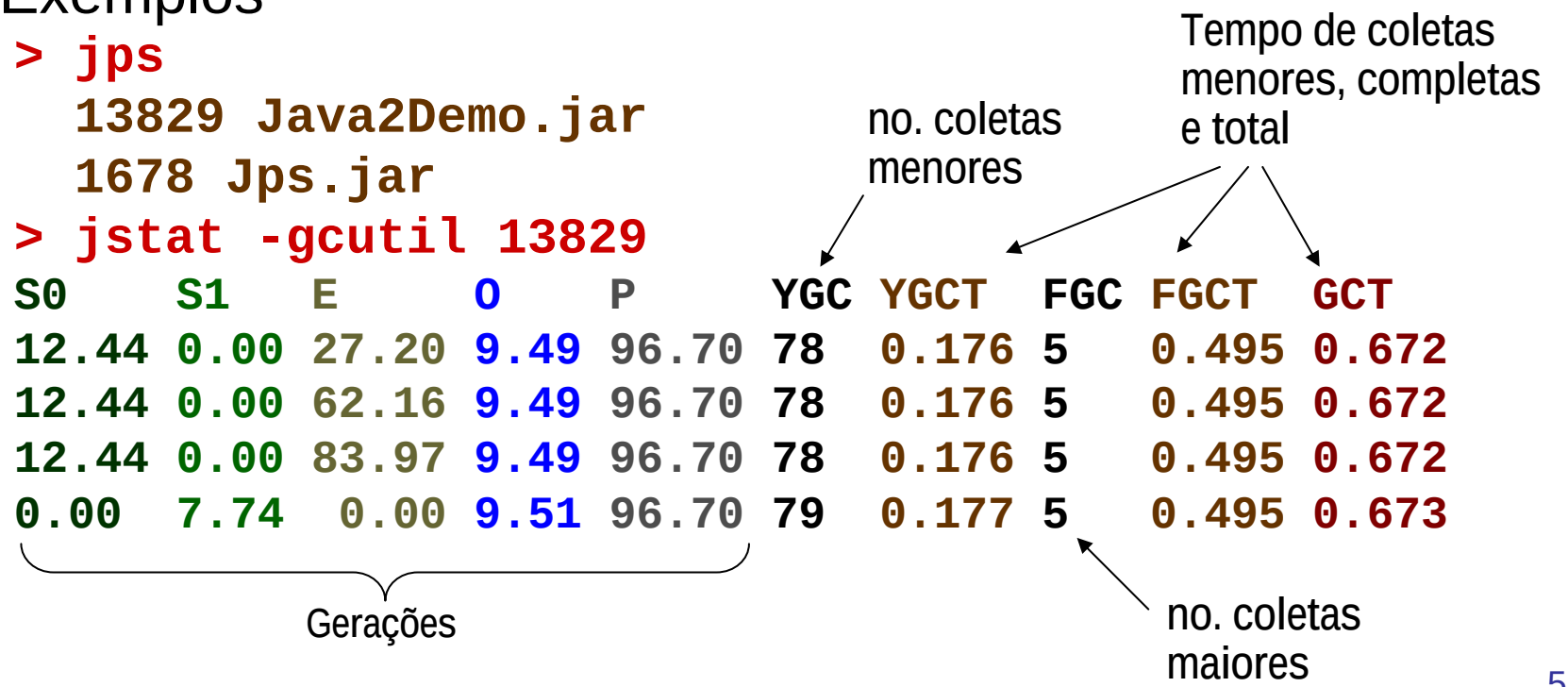
```
> jconsole alphard:3740
```

* [SDK]

Monitoração concisa: jstat

- Jstat é ferramenta do pacote experimental jvmstat
  - Obtém dinamicamente estatísticas de uso das gerações, de compilação, de carga de classes em tempo real
- Para executar, use jstat <opções> jvmid
- Exemplos

```
> jps
  13829 Java2Demo.jar
  1678 Jps.jar
> jstat -gcutil 13829
S0     S1    E      O      P      YGC  YGCT   FGC  FGCT   GCT
12.44  0.00  27.20  9.49   96.70  78   0.176  5    0.495  0.672
12.44  0.00  62.16  9.49   96.70  78   0.176  5    0.495  0.672
12.44  0.00  83.97  9.49   96.70  78   0.176  5    0.495  0.672
0.00   7.74   0.00  9.51   96.70  79   0.177  5    0.495  0.673
```



Visual GC

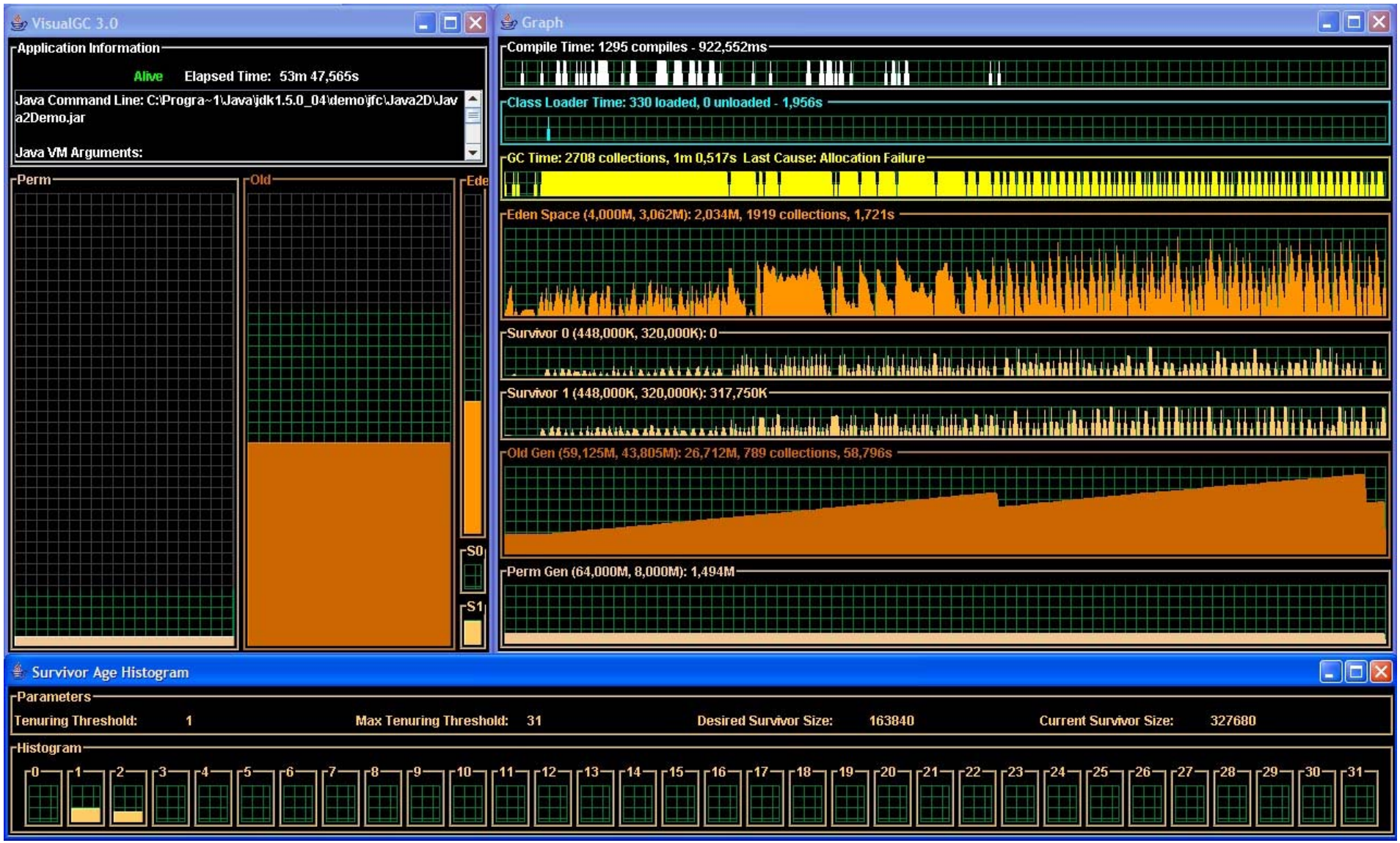
- Visual GC é a ferramenta visual do pacote experimental jvmstat (distribuído com o SDK 5.0)
  - Mostra gerações, coletas, carga de classes, etc. graficamente
  - É preciso baixar o Visual GC separadamente: http://java.sun.com/performance/jvmstat/visualgc.html
- Para rodar, é preciso ter o no. do processo da aplicação a monitorar e o período de coleta de amostras (em ms)
- Exemplo: monitoração local

```
> jps
21891 Java2Demo.jar
1362 Jps.jar
> visualgc 21891 250
```

- Para acesso remoto, é preciso obter o número do processo remoto e acrescentar o nome de domínio

```
> visualgc 21891@remota.com 250
```

Exemplo: Visual GC



Outras ferramentas

- GC Portal (java.sun.com/developer/technicalArticles/Programming/GCPortal)
  - Aplicação J2EE que gera gráficos e estatísticas
  - Requer instalação em um servidor
- GCViewer (www.tagtraum.com)
  - Analisa documentos de texto criados com –Xloggc:arquivo
  - Mostra comportamento das gerações e outras informações
  - Pode executar em tempo real
- Profilers
  - Vários *profilers* comerciais e gratuitos oferecem informações e capacidade de monitoração em tempo real da JVM
  - Comerciais: JProbe, OptimizeIt, JProfiler
  - Gratuitos: NetBeans Profiler, Eclipse Profiler, JRat, EJB, Cougaar, etc.

Exemplo: GC Viewer

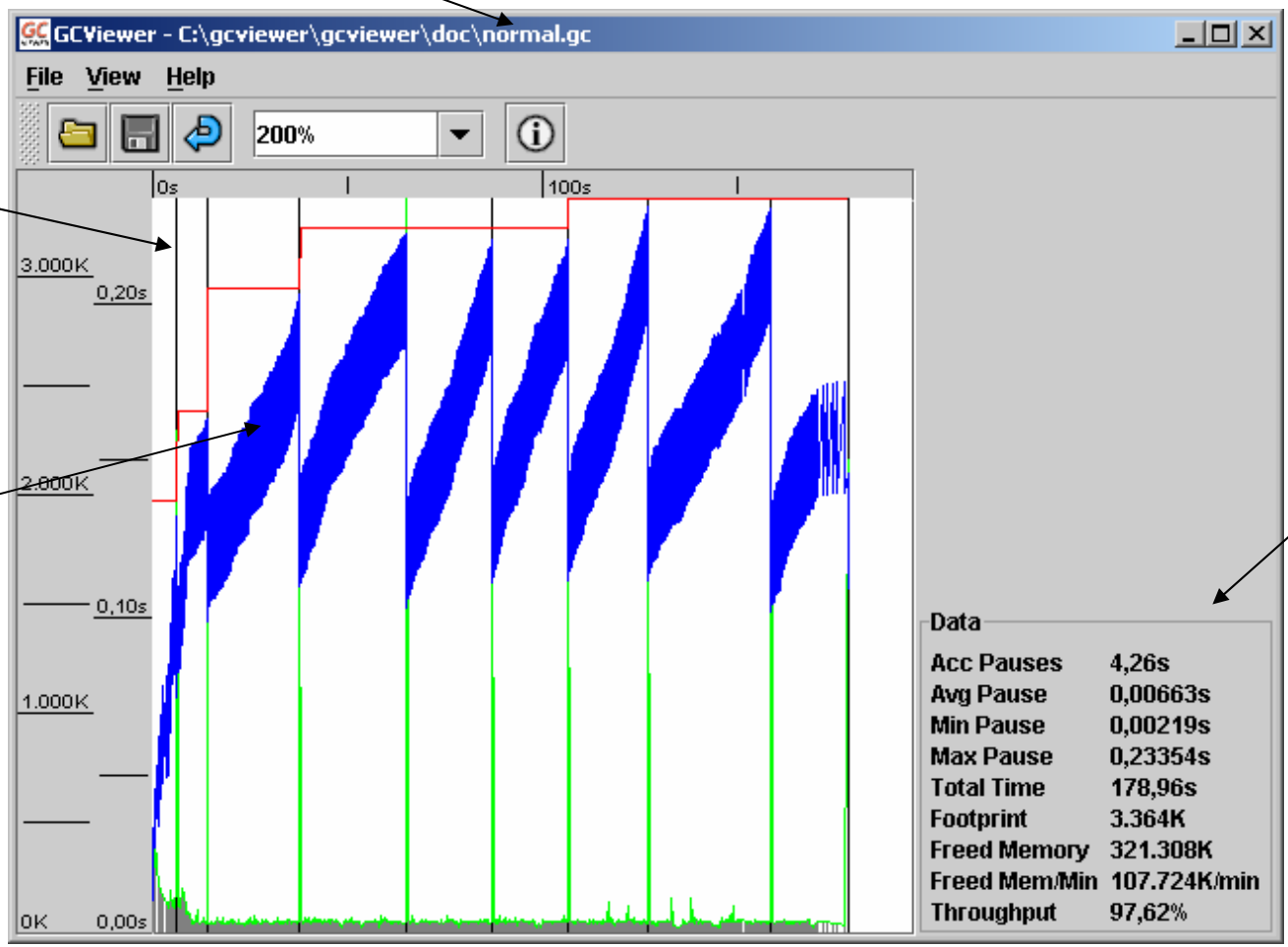
Dados carregados de arquivo gerado com a opção da JVM **`-Xloggc:normal.gc`**

Ferramenta gratuita de http://www.tagtraum.com/

Uma coleta maior

Várias coletas menores

Informações detalhadas (pausas, throughput, memória)



5. Ajuste automático: ergonomics

- O objetivo da ergonômica é obter a melhor performance da JVM com o mínimo de ajustes de linha de comando
  - Mais fácil de ajustar (ajuste manual é difícil)
  - Uso mais eficiente dos recursos
- A ergonômica busca obter, para uma aplicação, as melhores seleções de
  - Tamanho de heap
  - Coleta de lixo
  - Compilador de tempo de execução (JIT)
- Ajustes baseados em metas
  - Metas de pausa máxima
  - Capacidade de processamento desejado
  - Alvo: aplicações rodando em servidores grandes (-server)

Server class machine detection

- Quando uma aplicação inicia, o ambiente de execução tentará descobrir se está rodando em uma máquina "servidor" ou "cliente"
  - Se tiver pelo menos duas CPUs e pelo menos 2GB de memória física, será considerada servidora, caso contrário, é "cliente"
- Se estiver em máquina servidora, inicia a JVM Server
  - Inicia mais lentamente, mas com o tempo roda mais rápido
- Se estiver em máquina cliente, usa a JVM Client
  - Configurada para melhor performance em ambientes cliente
- JVM selecionada será usada e será default
  - Sempre será usada a não ser que seja especificado outra através das opções –server ou –client

Como funciona

- Usuário especifica
  - Meta de pausa máxima
  - Meta de eficiência mínima
  - Uso mínimo/máximo de memória
- Coletor de lixo ajusta automaticamente
  - Aumenta ou diminui tamanho das gerações jovem, espaços sobreviventes, geração estável
  - Altera política de promoção para geração estável
- Não garante que metas serão cumpridas
  - São metas, não garantias
  - Como garantir? Seguir estratégias recomendadas

Coletor paralelo: ergonomics

- As opções abaixo requerem -XX:+UseParallelGC

  -XX:MaxGCPauseMillis=valor em ms

  – JVM tentará manter pausas mais curtas que o valor especificado

  – Tem precedência sobre –XX:CGTimeRatio

  -XX:GCTimeRatio=n

  – Define meta de eficiência (*throughput*). A eficiência é

$$\frac{\textit{tempo de aplicação}}{\textit{tempo de coleta de lixo}} = 1 - \frac{1}{1+n}$$

  – n é um valor normalizado que mede a proporção de tempo dedicado à aplicação da forma 1:n.

  – *Exemplo*: se n for 19, a JVM reservará à aplicação 20 (19 + 1) vezes mais tempo que a coleta de lixo (coleta terá 5% do tempo)

  – Tem menos precedência que -XX:MaxGCPauseMillis.

Coletor paralelo: ergonomics

- **`-XX:+UseAdaptiveSizePolicy`**
  - Com esta opção presente, a JVM coleta dados e se baseia neles para redimensionar as gerações jovem e antiga
  - Esta opção é automaticamente ligada se a opção –XX:+UseParallelGC estiver presente.
- **`-XX:+AggressiveHeap`**
  - Implica no uso das opções –XX:+UseParallelGC e –XX:+UseAdaptiveSizePolicy
  - Não pode ser usada com –XX:+UseConcMarkSweepGC
  - Se esta opção estiver presente, a máquina virtual irá configurar vários parâmetros buscando otimizar tarefas longas com uso intenso de memória. Usará como base informações como quantidade de memória disponível e número de processadores.
  - Esta opção só é recomendada para servidores dedicados.
  - Requer pelo menos 256MB de memória física

Ergonomics: estratégias

1.1 Inicialmente, não escolha um valor máximo para o heap (-Xmx)

1.2 Escolha uma meta de eficiência (*throughput*) que seja suficiente para sua aplicação

- Em uma situação ideal, o sistema aumentará o heap até atingir um valor que permitirá alcançar a meta de eficiência desejada.

2.1 Se o heap alcançar o limite e o *throughput* não tiver sido alcançado

2.2 Então escolha um valor máximo para o heap (menor que a memória física da máquina) e rode a aplicação de novo.

- Se ainda assim a meta de eficiência não for atingida, é alta demais para a memória disponível na plataforma

3. Se a meta de eficiência foi alcançada mas as pausas ainda são excessivas, estabeleça uma meta de tempo máximo para pausas

- Isto pode fazer com que a meta de eficiência não seja mais alcançada
- Escolha valores que garantam um *tradeoff* aceitável

Conclusões

- Máquinas virtuais HotSpot implementam diversos algoritmos clássicos de coleta de lixo
  - Todos baseados em heap dividido em gerações
  - Pré-configurados para situações, plataformas e usos diferentes
  - Permitem ajustes manuais ou automáticos
- O ajuste adequado da JVM em grandes aplicações pode trazer ganhos dramáticos de performance
  - Ajuste pode ser simplesmente selecionar a JVM (server ou cliente) ou definir diversos complexos parâmetros manualmente
- Para ajustar parâmetros (mesmo automáticos) é preciso conhecer funcionamento dos algoritmos
  - Configuração manual (ex: tamanho de heap) impõe conseqüências que têm impactos em outras áreas da performance
  - Metas de ajuste automático afetam outras metas ou parâmetros

6. Apêndice: Class data sharing (CDS)

- Recurso das JVM Client 5.0 para reduzir o tempo de início de pequenas aplicações
- Durante a instalação, é criado um arquivo que será compartilhado pelas JVMs que estiverem executando
  - Durante a execução, múltiplas JVMs compartilharão classes na memória, evitando carregá-las novamente
- Para suportar este recurso, é preciso
  - Usar uma plataforma que não seja Windows 95/98/ME
  - Usar o JVM Client e coletor de lixo serial (*default* em desktops)
- Opções da JVM relacionadas

  `-Xshare:[on|off|auto]` – liga/desliga ou usa CDS se possível

  `-Xshare:dump` – gera novamente o arquivo. É preciso primeiro apagar o arquivo em **`$JAVA_HOME/client/classes.jsa`**

Fontes de referência

[SDK] Documentação do J2SDK 5.0

[Sun 05] Tuning Garbage Collection with the 5.0 Java Virtual Machine, Sun, 2005

[HotSpot] White Paper: The Java HotSpot Virtual Machine, v1.4.1. Sun, 2002.

[Apple 04] Java Development Guide for MacOS X. Apple, 2004

[Gotry 02] K. Gottry. Pick up performance with generational garbage collection. JavaWorld www.javaworld.com. Jan 2002

[Gupta 02] A. Gupta, M. Doyle. Turbo-charging Java HotSpot Virtual Machine, v1.4 to Improve the Performance and Scalability of Application Servers. Sun, 2002. http://java.sun.com/developer/technicalArticles/Programming/turbo

[Nagarajayya 02] N.Nagarajayya, J.S. Mayer. Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1. Sun Microsystems. Nov 2002

[Holling 03] G. Holling. J2SE 1.4.1 boosts garbage collection. JavaWorld. Mar 2003.

[Goetz 03] B. Goetz. Java theory and practice: Garbage collection in the HotSpot JVM. IBM Developerworks. Nov 2003.