Capítulo 2

Java Virtual Machine

Com a JVM no centro da Plataforma Java, conhecer seu funcionamento interno é essencial para qualquer aplicação Java. Muitas vezes, deixamos de lado ajustes importantes de Garbage Collector, não entendemos as implicações do JIT no nosso design ou enfrentamos problemas práticos com ClassLoaders.

Dentre os diversos tópicos associados à JVM, destacamos alguns que julgamos vitais para todo desenvolvedor Java.

2.1. Princípios de garbage collection

Durante muito tempo, uma das maiores dificuldades na hora de programar era o gerenciamento de memória. Os desenvolvedores eram responsáveis pela sua alocação e liberação manualmente, o que levava a muitos erros e *memory leaks*. Hoje, em todas as plataformas modernas, Java inclusive, temos gerenciamento de memória automático através de algoritmos de coleta de lixo.

O **Garbage Collector** (GC) é um dos principais componentes da JVM e responsável pela liberação da memória que não esteja mais sendo utilizada. Quando a

aplicação libera todas as referências para algum objeto, este passa a ser considerado lixo e pode ser coletado pelo GC a qualquer momento. Mas não conseguimos determinar o momento exato em que essa coleta ocorrerá; isto depende totalmente do algoritmo do garbage collector. Em geral, o GC não fará coletas para cada objeto liberado; ele deixará o lixo acumular um pouco para fazer coletas maiores, de maneira a otimizar o tempo gasto. Essa abordagem, muitas vezes, é bem mais eficiente, além de evitar a fragmentação da memória, que poderia aparecer no caso de um programa que aloque e libere a memória de maneira ingênua.[1]

Mas, como é realizada essa coleta exatamente? Vimos que ela não é feita logo que um objeto fica disponível, mas, sim, de tempos em tempos, tentando maximizar a eficiência. Em geral, a primeira ideia que aparece ao se pensar em GC é que ele fica varrendo a memória periodicamente e libera aqueles objetos que estão sem referência. Esta é a base de um conhecido algoritmo de GC chamado **Mark-And-Sweep**,[2] constituído de duas fases: na primeira, o GC percorre a memória e marca (*mark*) todos os objetos acessíveis pelas threads atuais; na segunda, varre (*sweep*) novamente a memória, coletando os objetos que não foram marcados na fase anterior.

Esse algoritmo envelheceu, da mesma forma que o ingênuo *reference counting*. Estudos extensivos com várias aplicações e seus comportamentos em tempo de execução ajudaram a formar premissas essenciais para algoritmos modernos de GC. A mais importante delas é a **hipótese das gerações**.[3]

Segundo esta hipótese, geralmente 95% dos objetos criados durante a execução do programa têm vida extremamente curta, isto é, são rapidamente descartados. É o que artigos acadêmicos chamam de *alto índice de mortalidade infantil* entre os objetos. A hipótese das gerações ainda diz que os objetos sobreviventes costumam ter vida longa. Com base nessas observações, chegou-se ao que hoje é conhecido como o algoritmo **generational copying**, usado como base na maioria das máquinas virtuais.[4,5]

É simples observar esse padrão geracional em muitos programas escritos em Java, quando objetos são criados dentro de um método. Assim que o método termina, alguns objetos que foram criados lá ficam sem referências e se tornam elegíveis à coleta de lixo, isto é, eles sobreviveram apenas durante a execução do método e tiveram vida curta.

Mesmo métodos curtos e simples, como `toString`, acabam gerando objetos intermediários que rapidamente não serão mais referenciados:

```
public String toString() {
   return "[ contatos: " + listaDeContatos + "]";
}
```


ArqDesignSoftware_BOOK.indb 22 12/11/2011 14:48:31

Aqui, durante a concatenação das três partes da `String`, um `StringBuilder` será utilizado, e o mesmo vai ocorrer para a invocação implícita do `toString` da coleção `listaDeContatos`, que gera uma `String` a partir de outro `StringBuilder`. Todos esses objetos podem ser rapidamente descartados, e talvez o próprio retorno do método será referenciado apenas por um breve instante.

No *generational copying*, a memória é dividida em gerações, que geralmente são duas: *young generation* e *old generation* (Figura 2.1). A geração nova é tipicamente menor que a velha; a JVM HotSpot, por exemplo, está programada para ocupar, por padrão, até um terço do espaço total. Todo novo objeto é alocado nesta parte e, quando ela se encher, o GC realizará seu trabalho em busca de sobreviventes. O truque está justamente no fato de que o GC, neste caso, varre apenas a geração jovem, que é bem pequena, sem ter de paralisar a JVM (*stop--the-world*) por muito tempo.

Os objetos que sobrevivem à coleta são, então, copiados para a geração seguinte, e todo o espaço da geração nova é considerado disponível novamente. Esse processo de cópia de objetos sobreviventes é que dá nome ao algoritmo.

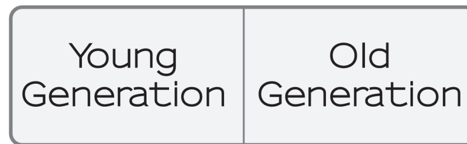

Figura 2.1 – Divisão das gerações no heap.

Pode-se pensar que o *generational copying* não é tão bom, pois, além de liberar objetos da memória, gasta tempo copiando os sobreviventes. Mas seu grande trunfo é que ele age nos objetos sobreviventes, e não nos descartados, como faria um algoritmo tradicional. No descarte, os objetos não são verdadeiramente apagados da memória; o GC apenas marca a memória como disponível. Segundo a hipótese das gerações, portanto, o *generational copying* realizará cópia em apenas 5% dos objetos, os que têm vida mais longa. E, embora uma cópia seja relativamente custosa, copiar apenas os poucos sobreviventes é mais rápido que liberar, um por um, os diversos objetos mortos.

Novos objetos são alocados na *young* e, assim que ela estiver lotada, é efetuado o chamado **minor collect**. É neste passo que os objetos sobreviventes são copiados para a *old*, e todo o espaço da *young* torna-se disponível novamente para alocação. Com menor frequência, são executados os **major collects**, que também coletam na geração *old* usando o *mark-and-sweep*. *Major collects* são também chamados

ArqDesignSoftware_BOOK.indb 23 12/11/2011 14:48:31

FullGC, e costumam demorar bem mais, já que varrem toda a memória, chegando a travar a aplicação nos algoritmos mais tradicionais (não paralelos).

É possível fazer uma observação sucinta do comportamento do GC mesmo sem um profiler, bastando usar a opção `-verbose:gc` ao iniciar a JVM. O log de execução do GC mostra a frequência das chamadas aos *minor* e *major collects*, bem como a eficiência de cada chamada (quanto de memória foi liberado) e o tempo gasto. É importante observar esses valores para perceber se o programa não está gastando muito tempo nos GCs, ou se as coletas estão sendo ineficientes. Um gargalo possível de ser encontrado é a geração *young* muito pequena, fazendo com que muitos objetos sejam copiados para a *old* para, logo em seguida, serem coletados em uma *major collect*, prejudicando bastante a performance geral do GC e a eficiência dos *minor collects*.

Um mito muito comum é o de que estressamos o GC se criarmos muitos objetos. Na verdade, como os algoritmos estão adaptados segundo a hipótese das gerações, o melhor são muitos pequenos objetos que logo se tornam desnecessários, do que poucos que demoram para sair da memória. Em alguns casos, até o tamanho do objeto pode influenciar; na JRockit, por exemplo, objetos grandes são alocados direto na *old generation*, logo não participam da cópia geracional.

A melhor técnica que um desenvolvedor pode utilizar é encaixar a demanda de memória da sua aplicação na hipótese das gerações e nas boas práticas de orientação a objetos, criando objetos pequenos e encapsulados de acordo com sua necessidade. Se o custo de criação do objeto não for grande, segurar suas referências ou fazer caches acaba sendo pior. Obviamente, isso exclui casos em que o custo de criação é grande, como um laço de concatenação de `String` através do operador `+`; nesse caso, é melhor usar `StringBuilders` ou `StringBuffers`.

Um efeito colateral interessante do algoritmo de cópia de objetos é a **compactação** da memória. Algoritmos ingênuos de GC costumam causar grande fragmentação, porque apenas removem os objetos não mais usados, e os sobreviventes acabam espalhados e cercados de áreas vazias. O *generational copying* copia os objetos sobreviventes para outra geração de forma agrupada, e a memória da geração anterior é liberada em um grande e único bloco, sem fragmentação. Fora isso, outras estratégias de compactação de memória ainda podem ser usadas pela JVM, inclusive na *old generation*. É importante notar que isso só é possível por causa do modelo de memória do Java, que abstrai totalmente do programa a forma como os ponteiros e endereços de memória são usados. É possível mudar objetos de lugar a qualquer momento, e a VM precisa apenas atualizar seus ponteiros internos, o que seria muito difícil de realizar em um ambiente com acesso direto a ponteiros de memória.

ArqDesignSoftware_BOOK.indb 24 12/11/2011 14:48:32

Explorando o gerenciamento de memória nas JVMs

A área total usada para armazenar os objetos na JVM é chamada **heap**. O tamanho do *heap* de memória da máquina virtual é controlado pelas opções `-Xms` e `-Xmx`. A primeira especifica o tamanho inicial do heap, e a segunda, o tamanho máximo. Inicialmente, a JVM aloca no sistema operacional a quantidade `Xms` de memória de uma vez, e essa memória nunca é devolvida para o sistema. A alocação de memória para os objetos Java é resolvida dentro da própria JVM, e não no sistema operacional. Conforme mais memória é necessária, a JVM aloca em grandes blocos até o máximo do `Xmx` (se precisar de mais que isso, um `OutOfMemoryError` é lançado). É muito comum rodar a máquina virtual com valores iguais de `Xms` e `Xmx`, fazendo com que a VM aloque memória no sistema operacional apenas no início, deixando de depender do comportamento específico do SO.

A forma como o *heap* é organizado depende totalmente da implementação da JVM. A maioria, atualmente, usa algoritmos baseados em gerações, mas não da mesma forma. A HotSpot da Oracle/Sun, por exemplo, divide a *young generation* em um espaço chamado *eden*, onde são feitas as novas alocações, e dois outros chamados *survivor space*. Na verdade, antes de um objeto ser copiado para a *old generation* (chamada *tenured* pela HotSpot), ele é copiado do *eden* para os *survivor spaces* algumas vezes, até estar "maduro" o suficiente para ir ao *tenured*. Com parâmetros avançados, é possível especificar a proporção entre as duas gerações (`-XX:NewRatio=`), e até entre o *eden* e o *survivor*.[6]

Conhecer essas e outras opções do garbage collector da sua JVM pode impactar bastante na performance de uma aplicação. Considere o código a seguir:

```
for (int i = 0; i < 100; i++) {
   List<Object> lista = new ArrayList<Object>();
   for (int j = 0; j < 300000; j++) {
      lista.add(new Object());
   }
}
```

Rodando um programa com esse código no `main` na JVM 6 HotSpot, usando o `-client` padrão com 64m de `Xms` e `Xmx`, ao habilitar o `-verbose:gc` obtemos uma saída como esta:

```
[GC 25066K->24710K(62848K), 0.0850880 secs]
[GC 28201K(62848K), 0.0079745 secs]
[GC 39883K->31344K(62848K), 0.0949824 secs]
[GC 46580K->37787K(62848K), 0.0950039 secs]
[Full GC 53044K->9816K(62848K), 0.1182727 secs]
....
```


ArqDesignSoftware_BOOK.indb 25 12/11/2011 14:48:32

Os valores mostrados em cada linha correspondem a: memória em uso antes da coleta, memória depois, total de memória da JVM e o tempo gasto na coleta. A saída real é bem maior, mas praticamente repete os mesmos padrões. É possível perceber que os *minor GC* são muito pouco eficientes, liberam pouca memória, ao contrário dos *Full GC* que parecem liberar muita. Isso indica fortemente que estamos indo contra a hipótese das gerações. Os objetos não estão sendo liberados enquanto ainda jovens, nem sendo copiados para a geração velha, de onde, depois de um tempo, serão liberados. Neste nosso caso, o algoritmo de GC baseado em cópia de objetos é um imenso gargalo de performance.

Propositadamente, o código de teste segura referências por um tempo que estressa o GC. Em uma aplicação real, o ideal seria alterar o código para que se adapte melhor à hipótese das gerações. Mas, quando o comportamento da sua aplicação é diferente do habitual, basta alterar algumas configurações do garbage collector para obter algum ganho.

Em nossa máquina de testes, sem alterar o tamanho total do heap, ao aumentar o da *young generation* usando a opção `-XX:NewRatio=1` (o padrão é 2 na HotSpot), temos um ganho de mais de 50% na performance geral da aplicação. Essa opção muda a proporção entre os tamanhos da *young generation* e da *old*. Por padrão, $^{2}/_{3}$ serão *old* e $^{1}/_{3}$ será *young*; ao mudar o valor para 1, teremos $^{1}/_{2}$ para cada área. Claro que esse ganho depende muito do programa e da versão da JVM, mas, ao olhar agora para a saída do programa, é possível observar como uma pequena mudança impacta bastante na eficiência do GC:

```
[GC 49587K->25594K(61440K), 0.0199301 secs]
[GC 43663K->26292K(61440K), 0.0685206 secs]
[GC 47193K->23213K(61440K), 0.0212459 secs]
[GC 39296K->21963K(61440K), 0.0606901 secs]
[GC 45913K->21963K(61440K), 0.0215563 secs]
...
```

Nos novos testes, o *Full GC* nem chega a ser executado, porque os próprios *minor gc* são suficientes para dar conta dos objetos. E, como o algoritmo de GC é baseado em cópias, poucos objetos são copiados para a *old generation*, influenciando bastante na performance do programa. **Saiba adequar sua aplicação à hipótese das gerações, seja adequando seu código, seja conhecendo as melhores configurações de execução da sua JVM.**

Pensando em como usufruir dessas características do GC, o que é melhor para uma aplicação: usar objetos grandes, caches gigantes de objetos e abusar de

ArqDesignSoftware_BOOK.indb 26 12/11/2011 14:48:32

atributos estáticos; ou escrever diversos pequenos objetos criados e liberados a todo momento? Segurar objetos na memória estressa demais o GC baseado em cópias de objetos. A boa prática de Orientação a Objetos já diz que devemos criar pequenos objetos encapsulados, sem muitos dados estáticos, originando instâncias sem nos preocuparmos com caches e outras alegadas otimizações. Se seguirmos a boa prática, o GC também funcionará melhor. A JVM, de um modo geral, foi escrita e adaptada ao longo de sua vida para suportar o desenvolvedor nas boas práticas.

Outro ponto particular da JVM HotSpot da Oracle/Sun (e suas derivadas) é a chamada *permanent generation*, ou **PermGen** (Figura 2.2). É um espaço de memória contado fora do heap (além do `Xms/Xmx` portanto), onde são alocados objetos internos da VM e objetos `Class`, `Method`, além do **pool de strings**. Ao contrário do que o nome parece indicar, esse espaço de memória também é coletado (durante os *FullGC*), mas costuma trazer grandes dores de cabeça, como os conhecidos `OutOfMemoryError`, acusando o fim do *PermGen space*. A primeira dificuldade aqui é que, por não fazer parte do `Xms/Xmx`, o *PermGen* confunde o desenvolvedor, que não entende como o programa pode consumir mais memória do que a definida no `-Xmx`. Para especificar o tamanho do *PermGen*, usa-se outra opção, a `-XX:MaxPermSize`.

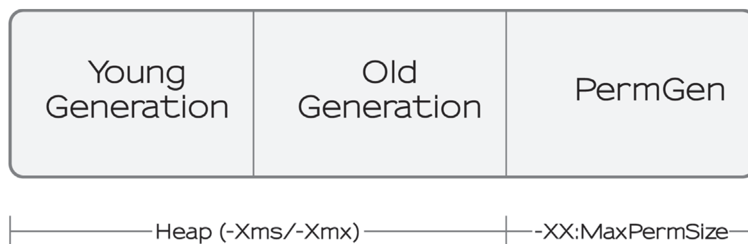

Figura 2.2 – Divisão da memória na JVM da Sun.

Mas os problemas com estouro do *PermGen* são difíceis de diagnosticar, justamente porque não se trata de objetos da aplicação. Na maioria dos casos, está ligado a uma quantidade exagerada de classes que são carregadas na memória, estourando o tamanho do *PermGen*. Um exemplo conhecido, que acontecia antigamente, era usar o Eclipse com muitos plugins (sabidamente o WTP) nas configurações padrões da JVM. Por causa da arquitetura de plugins bem organizada e encapsulada do Eclipse, muitas classes eram carregadas e a memória acabava.

ArqDesignSoftware_BOOK.indb 27 12/11/2011 14:48:32

Um outro problema bem mais corriqueiro são os estouros do *PermGen* ao se fazer muitos *hot deploys* nos servidores de aplicação. Por vários motivos possíveis (como veremos durante o tópico de ClassLoaders), o servidor não consegue liberar as classes do contexto ao destruí- lo. Uma nova versão da aplicação é então carregada, mas as classes antigas continuam na memória. É apenas uma questão de tempo para esse vazamento de memória estourar o espaço do *PermGen*.

Algoritmos de coleta de lixo

Aplicações de grande porte podem sofrer com o GC. É fundamental conhecer algumas das principais opções que sua JVM possibilita. No caso da HotSpot, a JVM mais utilizada, diversos algoritmos diferentes estão disponíveis para uso.[6,7] Por padrão, é usado o **Serial Collector**, que é bastante rápido, mas usa apenas uma thread. Em máquinas com um ou dois processadores, ele costuma ser uma boa escolha. Mas em servidores mais complexos, com vários processadores, pode desperdiçar recursos. Isto porque ele é um algoritmo *stop-the-world*, o que significa que todo processamento da aplicação deve ser interrompido enquanto o GC trabalha, para evitar modificações na memória durante a coleta. Mas, por ser serial, implica que, se uma máquina tiver muitos processadores, apenas um poderá ser usado pelo GC, enquanto todos os demais ficam ociosos, pois não podem executar nenhuma outra tarefa enquanto o GC roda. Isto é um desperdício de recursos, e um gargalo em máquinas com vários processadores.

É possível então, na HotSpot, habilitar diversos outros algoritmos de GC e até usar combinações deles. O **Parallel Collector** (habilitado com `-XX:+UseParallelGC`) consegue rodar *minor collects* em paralelo. Ele também é *stop-the-world*, mas aproveita os vários processadores, podendo executar o GC. Note que, por causa disso, o algoritmo paralelo acaba demorando um tempo total maior que o serial, em razão do custo de sincronizar o acesso à memória sendo coletada (`synchronized`, semáforos, mutexes, ...). Mas, como estamos falando de máquinas com muitos processadores, o que acaba acontecendo é que esse tempo é dividido entre os vários processadores, e o efeito é que se usa melhor o hardware e se diminui o tempo de parada para coleta.[6] É possível ainda habilitar um coletor paralelo para a geração *old* e os *major collects*, usando `-XX:+UseParallelOldGC.`

Quando o requisito principal é diminuir o tempo de resposta, ou seja, diminuir o tempo que a aplicação fica parada esperando o GC, o melhor pode ser usar o **Concurrent Mark-and-sweep** (CMS), também chamado de *low pause collector* (ativado com `-XX:+UseConcMarkSweepGC`). Esse algoritmo age na geração *old*

ArqDesignSoftware_BOOK.indb 28 12/11/2011 14:48:33

e consegue fazer a maior parte da análise da memória sem parar o mundo. Ele tem apenas um pequeno pedaço *stop-the-world*, o que acaba tornando o tempo de resposta bem menor. Mas o preço que se paga é que o algoritmo gasta mais processamento total, e tem custos maiores de controle de concorrência. Pode ser uma boa ideia em máquinas com muitos processadores e com bastante memória a ser coletada.[6] Ao usar o CMS, a geração *young* é coletada com um algoritmo paralelo chamado ParNew e é possível até desabilitá-lo e usar o Serial.

A última novidade em relação a algoritmos GC é o **G1**, desenvolvido pela Sun.[8] É um algoritmo concorrente que tenta trazer um pouco mais de previsibilidade para o GC, uma das grandes críticas de quem prefere gerenciamento manual de memória. Apesar de ser considerado um algoritmo geracional, ele não separa o heap em duas gerações, e sim em muitas pequenas regiões. Dinamicamente, o G1 escolhe algumas regiões para ser coletadas (o que acaba dando o sentido lógico daquelas regiões serem a *young gen*). O interessante é que é possível especificar para o G1 tempo máximo a ser gasto em GC em um determinado intervalo de tempo. O G1, então, tenta escolher um número de regiões para cada coleta que, baseado em execuções anteriores, ele acha que atingirão a meta solicitada. Com isso, espera-se ter um algoritmo de GC mais eficiente e mais previsível, com paradas mais curtas e constantes.

Esses algoritmos mudam muito de uma JVM para outra. Os quatro já citados são das últimas versões da JVM 6 e da JVM 7 da Oracle/Sun. Outras máquinas virtuais possuem outros algoritmos. A JRockit, por exemplo, permite desabilitar o algoritmo geracional e usar um baseado no *mark-and-sweep* com a opção `-Xgc:singlecon` (há outros também parecidos com os algoritmos *parallel* e *concurrent* do HotSpot).[9] A Azul Systems, conhecida por sua JVM focada em alta performance, possui em seus produtos, desde o fim de 2010, um GC sem pausas.[10]

A memória pode ser um gargalo para sua aplicação; portanto, conhecer bem seu funcionamento e seus algoritmos pode ser peça-chave para o sucesso de um projeto.

2.2. Não dependa do gerenciamento de memória

Como vimos, o *Garbage Collector* é um dos serviços mais importantes da JVM e exige pouco esforço para usá-lo. Conhecê-lo, além de seus algoritmos, ajuda no momento de escrever códigos melhores e não ter surpresas. Mas há ainda outras questões relacionadas ao comportamento do GC.

Um dos principais pontos é sua natureza imprevisível. Por ser um serviço da JVM que roda simultaneamente à sua aplicação, não conseguimos controlar