Gerenciar memória não é simples

Durante muito tempo, uma das maiores dificuldades na hora de programar era o gerenciamento de memória. Os programadores eram responsáveis pela alocação e liberação dela 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 é responsável pela liberação da memória que não esteja mais sendo utilizada. Quando a aplicação libera as referências para algum objeto, esse objeto passa a ser considerado lixo e poderá ser coletado pelo GC a qualquer momento. Mas não conseguimos determinar o momento em que essa coleta irá ocorrer, isso depende totalmente do algoritmo do garbage collector. E, em geral, o GC não fará coletas para cada objeto liberado: ele vai deixar o lixo acumular um pouco para fazer coletas maiores, de maneira a otimizar o tempo gasto. Isso é inclusive considerado muito melhor em performance e não fragmentação da memória que um programa que faz gerenciamento manual de memória e acaba liberando os espaços assim que ficam disponíveis [66].

Mas como é feita a coleta? Vimos que a coleta não é feita logo que um objeto fica disponível, mas sim de tempos em tempos coletando vários de uma vez. Normalmente, a primeira ideia que aparece ao se pensar em GC, é que ele fica varrendo a memória de tempos em tempos e libera aqueles objetos que estão sem referência. Um pensamento desse tipo é a base de um algoritmo famoso de GC chamado de **Mark-And-Sweep** [48]. Esse algoritmo é constituído de duas fases: na primeira, o GC percorre a memória e marca (*mark*) todos os objetos acessíveis; na segunda, ele varre (*sweep*) novamente a memória coletando os objetos não marcados.

O *mark-and-sweep* foi usado durante muito tempo nas máquinas virtuais e é um dos mais famosos do mundo, junto com o algorítmo de *Reference Counting*. Mas com o tempo foi-se percebendo que ele não era tão eficiente na maioria das aplicações reais rodadas. Estudos extensivos com várias aplicações e seu comportamento em tempo de execução ajudaram a formar premissas essenciais para os algoritmos modernos de GC. A mais importante delas é a **hipótese das gerações**.

Segundo a hipótese das gerações, em média, 95% dos objetos criados durante a execução do programa têm vida curta, isto é, são rapidamente descartados. É o que alguns artigos acadêmicos chamam de *alto índice de mortalidade infantil* entre os objetos. E a hipótese das gerações ainda diz que os objetos sobreviventes em geral têm vida longa. Com base nessas conclusões, chegou-se no que hoje é conhecido como o algoritmo **generational copying** [41, 49] usado na maioria das máquinas virtuais.

É simples de observar o padrão em muitos programas escritos em Java, quando um objeto é criado dentro de um método. Acontece com frequência na maioria das aplicações e assim que o método termina, geralmente todas as suas referências são perdidas e ele se torna elegível para a coleta de lixo. O objeto sobreviveu apenas durante a execução do método e tem **vida curta**:

```
public boolean algumMetodo() {
  ObjetoDeVidaCurta obj = new ObjetoDeVidaCurta();
  // mais um pouco de código...
```

```
  return false;
  // todas as referencias ao objeto são agora descartadas
}
```

No *generational copying* a memória é dividida em gerações, geralmente na *young generation* e na *old generation*. A geração nova é tipicamente bem menor que as demais: na JVM da Sun, normalmente inicia com tamanho de 2.2MB. Todo novo objeto é alocado na geração nova e quando essa geração lota, o garbage collector varre esse espaço em busca de sobreviventes. A sacada está justamente no fato de que o GC varre **apenas a geração jovem**, que é pequena. Dessa forma, a aplicação não fica travada por muito tempo e a coleta é rápida.

Os objetos que sobrevivem à coleta são 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.

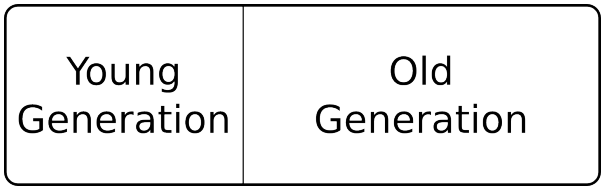

Figura 4.3: Divisão das gerações no heap

Pode-se pensar que o *generational copying* é ruim porque copia objetos na memória ao invés de liberá-la, afinal é muito mais rápido liberar um endereço de memória do que copiar vários bytes entre lugares diferentes. Mas, o grande trunfo do *generational copying* é que ele age nos objetos sobreviventes e não nos objetos descartados como faria um algoritmo tradicional. No descarte, os objetos não são apagados de verdade da memória; o GC apenas marca a memória como disponível. Segundo a hipótese das gerações, portanto, o *generational copying* vai agir em apenas 5% dos objetos e não em 95% deles que têm vida curta. É por isso que esse algoritmo é, na prática, mais eficiente.

Como vimos, em geral o tamanho da geração *young* é menor que o da geração *old*. Novos objetos são alocados na *young* e, assim que ela lota, é efetuado o chamado **minor collect**. É nesse passo que os objetos sobreviventes são copiados para a *old* e todo o espaço da *young* se torna disponível novamente para alocação. Menos frequentemente, são executados os **major collects** que também coletam na geração *old* usando o bom e velho *mark-and-sweep*. Os *major collects* são também chamados de **FullGC** e costumam demorar bem mais, até travando a aplicação nos algoritmos mais tradicionais (não paralelos).

É possível observar o comportamento do GC usando a opção `-verbose:gc` na hora de iniciar a JVM. A análise do 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 muito comum de encontrarmos em aplicações é a geração *young* muito pequena, o que não dá vazão para a maioria dos objetos que morre cedo. Nesse caso, muitos objetos acabam sendo copiados para geração *old* mas logo coletados em uma *major collect* próxima o que acaba prejudicando bastante a performance geral do GC e a eficiência dos *minor collects*.

Outro problema muito comum de se encontrar é o mito de que se criar muitos objetos é ruim pois estressa o GC. Na verdade, segundo a hipótese das gerações, melhor são muitos pequenos objetos que logo somem do que poucos objetos 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* e não usufruem do *generational copying*. A melhor técnica que um programador pode utilizar é encaixar a demanda de memória da sua aplicação na hipótese das gerações e nas boas práticas de orientação a objetos, criando muitos objetos pequenos e encapsulados.

Um efeito colateral bem interessante do algoritmo de cópia de objetos é a **compactação** da memória. Algoritmos de GC costumam causar grande fragmentação na memória porque apenas removem os objetos não mais usados e os objetos ainda vivos acabam ficando espalhados e cercados de grandes áreas vazias. O *generational copying* copia os objetos sobreviventes para outra geração de forma agrupada e a memória da geração anterior é inteira liberada em um grande 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 encapsula e esconde 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. Isso seria muito difícil de se fazer em um ambiente com acesso direto a ponteiros de memória.

Explorando o GC nas JVMs

A área total usada para armazenar os objetos na JVM é chamada de **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 for sendo necessária, a JVM vai alocando 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` o que faz com que a VM aloque memória no sistema operacional apenas no início e nunca mais faça chamadas ao sistema para isso.

A forma como o *heap* é organizado depende totalmente da implementação da JVM. A maioria usa algoritmos baseados em gerações, mas não da mesma forma. A **HotSpot** da Sun, por exemplo, divide a *young generation* em um espaço chamado de *eden* onde são feitas as novas alocações e dois outros espaços chamados de *survivor space*. Na verdade, na HotSpot, antes de um objeto ser copiado para a *old generation* (que, aliás, é chamada de *tenured* pela Sun), ele é copiado do *eden* para os *survivor spaces* algumas

vezes até estar "maduro" o suficiente para ir para o *tenured*. Com parâmetros avançados é possível especificar a proporção entre as duas gerações (*-XX:NewRatio=*) e até entre o *eden* e o *survivor* [38].

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

```
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 da Sun usando o client HotSpot padrão com 64m de `Xms` e `Xmx`, ao habilitar o `-verbose:gc` obtemos uma saída como essa:

```
[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]
....
```

A saída é bem maior, mas praticamente repete os mesmos padrões. Analisando essa saída é possível perceber que os *minor gc* são muito pouco eficientes, liberam pouca memória. Já os *Full GC* parecem liberar muita memória. Isso indica fortemente que estamos indo contra a hipótese das gerações. Os objetos não estão sendo liberados enquanto ainda jovens, sendo copiados para a geração velha de onde, depois de um tempo, serão liberados. Nesse nosso caso, o algoritmo de GC baseado em cópia de objetos está sendo um imenso gargalo de performance.

Propositalmente o código de teste segura referências por um tempo que vá estressar o GC. Em uma aplicação real, o ideal seria alterar o código para que ele se adeque melhor à hipótese das gerações. Mas é importante saber que muitas vezes basta alterar algumas configurações do garbage collector e já teremos algum ganho.

Em nossa máquina de testes, ao aumentar o tamanho da *young generation* usando a opção `-XX:NewRatio=1` (o padrão é 2 na hotspot client), temos um ganho na performance geral da aplicação de mais de 50%. Claro que isso depende muito do programa e da versão da JVM, mas ao olhar para a saída do programa agora, 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 nossos testes, o *Full GC* nem chega a ser executado, os próprios *minor gc* são suficientes para dar conta dos objetos. E, pelo algoritmo do GC ser baseado em cópia, isso implica que poucos objetos são copiados para a *old generation*, influenciando bastante na performance. **Saiba adequar sua aplicação à hipótese das gerações, seja adequando seu código, seja conhecendo as melhores configurações de execução da sua JVM.**

Outro ponto particular da JVM da Sun (e suas derivadas) é a chamada **permanent generation**, ou **PermGen**. 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`, entre outros, 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 com os famosos `OutOfMemoryError` acusando o fim do *PermGen space*. A primeira dificuldade aqui é que, por não fazer parte do `Xms/Xmx`, o *PermGen* acaba confundindo o programador que não entende como a memória pode acabar mesmo ele colocando valores altos naquelas opções. Na JVM da Sun, para especificar o tamanho do *PermGen*, usa-se outra opção, a `-XX:MaxPermSize`.

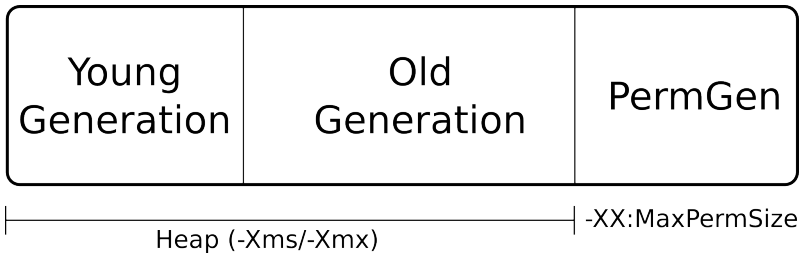

Figura 4.4: Divisão da memória na JVM da Sun

Mas os problemas com estouro do *PermGen* são difíceis de diagnosticar justo porque não se trata de objetos do programa. Na maioria dos casos, está ligado a uma quantidade exagerada de classes que são carregadas na memória estourando o tamanho do *PermGen*. Um exemplo clássico que acontecia algum tempo atrás era usar o Eclipse com muitos plugins (conhecidamente o WTP) nas configurações padrões da JVM da Sun. Por causa da arquitetura de plugins bem organizada e encapsulada do Eclipse, muitas classes eram carregadas e a memória acabava. Um outro problema bem mais corriqueiro são os estouros do *PermGen* ao se fazer muitos *redeploys* nos servidores de aplicação. Por vários motivos possíveis (como veremos durante o tópico de ClassLoaders

), o servidor acaba não conseguindo 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 costumam sofrer bastante com o GC. É fundamental conhecer algumas das principais opções que sua JVM possibilita. No caso da JVM da Sun, diversos algoritmos diferentes estão disponíveis para uso [38, 35]. 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 ele costuma ser um grande desperdiçador de recursos. Isso porque ele é um algoritmo *stop-the-world* o que significa que todo processamento da aplicação deve ser interropido enquanto o GC trabalha, para evitar modificações na memória durante a coleta. Mas por ser serial, isso implica que, se uma máquina tiver muitos processadores, apenas um poderá ser usado pelo GC e todos os demais ficam ociosos pois não podem executar nada do programa enquanto o GC roda. Isso é um desperdício de recursos e um gargalo bem ruim em máquinas com vários processadores e programas que usam muita memória.

É possível então, na HotSpot, habilitar outros algoritmos de GC. 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 várias threads de GC paralelamente. Note que, por causa disso, o algoritmo paralelo acaba demorando um tempo total maior que o serial, pelo custo de sincronizar o acesso paralelo 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 processores e o efeito é que se usa melhor o hardware e o tempo total da coleta diminui [38].

Mas 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 é usar o **Concurrent Mark-and-sweep** (CMS), também chamado de *low pause collector* (ativado com `-XX:+UseConcMarkSweep`). Esse algoritmo consegue fazer a maior parte da análise da memória sem parar o mundo. Ele tem apenas um pequeno pedaço que é *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 e tem custos maiores de controle de concorrência (`synchonized`). Porém, costuma ser uma boa ideia em máquinas com muitos processadores e com bastante memória a ser coletada [38].

A última novidade enquanto escrevemos esse livro em relação a algoritmos GC é o **G1** desenvolvido pela Sun [37]. Ele é um algoritmo concorrente mas 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 serem coletadas (o que acaba dando o sentido lógico daquelas regiões serem a *young gen*). O interessante é que é possível especificar para o G1 o tempo máximo a ser gasto em GC em um determinado período. O G1 então tenta escolher um número de regiões para cada coleta que, baseado em execuções anteriores, ele acha que atingirão a meta pedida. Com isso, espera-se ter um algoritmo de GC mais eficiente e mais previsível.

Esses algoritmos mudam muito de JVM para JVM. Os quatro citados antes são das últimas versões da JVM 6 e da JVM 7 da Sun. Outras máquinas virtuais possuem

outros algoritmos. A JRockit, por exemplo, permite desabilitar o algoritmo geracional e usar um baseado no *mark-and-sweep* usando a opção `-Xgc:singlecon` (há outros também parecidos com os algoritmos *parallel* e *concurrent* do HotSpot) [43].

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. Na conclusão do livro, são abordadas algumas possíveis soluções para enfrentar problemas que necessitam de muita memória, como o uso do *memcached* ou do *terracotta*.