Funcionamento do coletor de lixo

gleydson@jspbrasil.com.br

Em Java, diferente de outras linguagens como C e C++, o programador não precisa desalocar memória, usando funções como `free e delete`. Java possui um coletor de lixo, um thread de baixa prioridade que roda junto com a Java Virtual Machine e procura por áreas de memória que não estejam mais em uso para realizar a liberação da mesma.

A execução do coletor de lixo é transparente para o programador, porém, podemos em algumas situações querer usar alguns artifícios para otimizar o uso da memória usada pela JVM. O coletor de lixo deixa o programador livre da tarefa de gerenciamento de memória, porém isso não significa que o programador não deve se preocupar com o uso da memória em suas aplicações.

O coletor de lixo libera a memória mantida por um objeto se não houver mais nenhuma referência para ele. Dependendo do algoritmo da coleta de lixo usado pela máquina virtual Java no qual o código está sendo executado, nem todo objeto desreferenciado será desalocado pelo coletor de lixo. Um objeto mais antigo, de longa duração, é menos provável de ser desreferenciado do que um objeto novo, de modo que um algoritmo comum para coleta de lixo é analisar os objetos mais antigos com menos freqüência do que os objetos mais novos.

Podemos ter um métrica da eficiência da implementação do coletor de lixo utilizado pela sua JVM com o código abaixo:

```
Runtime rt = Runtime.getRuntime();
long mem = rt.freeMemory();
System.out.printn(“Memória Livre: “ + mem);


// ... algum código


System.gc();

// ...

mem = rt.freeMemory();
System.out.println(“Memória Livre: “ + mem);
```

O código acima obtém o ambiente de execução com o método estático `getRuntime()` da classe Runtime. O método `freeMemory()` retorna a memória disponível no ambiente runtime. A linha `System.gc()` faz a chamada explícita do coletor de lixo.

A grande maioria das implementações da JVM resulta na execução do coletor de lixo como resultado da chamada do método `System.gc()`. Além disso, a maior parte das máquinas virtuais fazem a chamada do coletor implicitamente quando a memória está baixa ou quando a CPU está inativa por um certo período de tempo.

Problemas de memória podem surgir quando os objetos contém variáveis de instância que são iniciadas no construtor, ocupam um grande quantidade de memória e não são mais utililzadas no restante da aplicação. Isso pode degradar o desempenho do seu código.

Desreferenciado variáveis atribuindo-as null

Uma maneira de tornar o conteúdo da variável um candidato a ser limpo pelo coletor de lixo, é atribuindo null a variável. Com isso, o dado torna-se desreferenciado e pode ser limpo pelo coletor de lixo. O programa abaixo ilustra isso:

```
import java.util.*;


class GarbageExample {

        private static Vector vetor;

        public static void main(String args[]) {

                vetor = new Vector();
                for (int a=0; a < 500; a++)
                        vetor.addElement(new
StringBuffer("teste"));

                Runtime rt = Runtime.getRuntime();

                System.out.println("Memória Livre: " +
rt.freeMemory());

                vetor = null;

                System.gc();

                System.out.println("Memória Livre: " +
rt.freeMemory());

        }
}
```

O código acima aloca 500 objetos StringBuffer e os coloca dentro de um Vector. O objeto da classe Vector é referenciado pela variável `vetor`. Quando atribui-se o valor `null` para a variável vetor o conteúdo do objeto Vector (500 StringBufferes) não possui mais referência na aplicação. A chamada do coletor de lixo faz a limpeza desse objeto e imprime a memória antes e depois da limpeza.

Essas técnicas podem solucionar o problema de manter grandes quantidades de memória desnecessárias. Para usar o mínimo de memória possível, os objetos que existem pela duração do programa devem devem ser os menores possíveis. Além disso, grandes objetos deverão existir pelo menor tempo possível.

A limitação da quantidade de memória utilizada por um programa em Java pode ajudar no

desempenho apenas do código em Java rodando na mesma instância da JVM, e não necessariamente de outros programas rodando no mesmo sistema. Isso porque muitos algoritmos de gerenciamento de heap alocam antecipadamente uma parte da memória para ser usada pelo seu código. Ou seja, você libera a memória da máquina virtual Java, o que não afetará outras aplicações não-Java, pelo fato da JVM pre-alocar o heap que ela precisa.

A qualquer momento você poderá solicitar a execução do coletor de lixo, chamando o método `System.gc()`. Porém, é necessário analisar o impacto de desempenho na sua aplicação. Muitos algoritmos de coleta de lixo suspendem todos os outros threads antes de entrar em execução. Isso garante que, quando o coletor de lixo for executado, ele terá acesso completo à memória no heap e poderá realizar suas tarefas com segurança, sem que seja interrompido por outro thread. Quando o coletor de lixo termina seu trabalho, todos os threads que ele suspendeu são retomados.

O coletor de lixo de Java é executado com um grande freqüência, sua chamada explícita dificilmente é necessária. Porém, em alguns casos pode tornar-se conveniente. Procure evitar a chamada do método `System.gc()` e nunca o use dentro de loops.

O método finalize()

O programador pode definir o método finalize que será chamado sempre antes do objeto ser coletado pelo coletor de lixo. A aplicação abaixo ilustra isso:

```
import java.util.*;

class GarbageExample {

        private static MeuVetor vetor;
```

```
class MeuVetor extends Vector {
       public void finalize() {
                System.out.println("Vou ser coletado!!");
       }
}
```

O método finalize da classe MeuVetor é executado antes do objeto ser coletado da memória, a saída deste programa é:

```
Memória Livre: 459808
Vou ser coletado!!
Memória Livre: 600664
```

Neste artigo procurei dar uma idéia da coleta de lixo e dicas para otimizar o uso da memória em suas aplicações Java. A chamada do `System.gc()` deve ser evitada, porém há situações em que ela pode ser usada.