Universidade Estadual do Paraná
Campus Apucarana

ORGANIZAÇÃO E ESTRUTURA DE DADOS II: GARBAGE COLLECTION

Rafael Francisco Ferreira

APUCARANA
2016

O **Garbage Collector,** também conhecido como coletor de lixo, é um recurso que surgiu devido a um problema de gerência de memória onde o desenvolvedor precisava fazer o deslocamento de objetos na memória manualmente para que outros objetos fossem alocados.

Em todos os tipos de programas, uma variedade imensa de recursos é utilizada, seja para acessar um banco de dados, espaços de tela, buffers de memória, conexões de rede, etc. Além destes recursos, temos que no ambiente de programação orientada a objetos, cada tipo usado é um recurso potencial disponível para ser usado pelo programa e a utilização desses recursos requer que memória seja alocada para representar o tipo.

Em geral, se pensa que o GC elimina toda a responsabilidade de gerência de memória do desenvolvedor de uma aplicação. Em outros casos ocorre o contrário, isto é, o desenvolvedor quem realiza muito mais trabalho do que é necessário.

Os princípios básicos do coletor de lixo são encontrar objetos de um programa que não serão mais acessados no futuro, esses objetos são tratados como “lixo” e por sua vez ocupam espaço na memória, o que se torna desnecessário e prejudicial ao sistema, visto que aquela parte suja da memória poderia estar sendo usada por outro aplicativo, e desalocar os recursos utilizados por tais objetos. Tornando a desalocação manual de memória desnecessária, e geralmente proibindo tal prática, o coletor de lixo livra o programador de se preocupar com a liberação de recursos já não utilizados, o que pode consumir uma parte significativa do desenvolvimento do software. Também evita que o programador introduza erros no programa devido à má utilização de ponteiros.

Assim o Garbage Collector gerencia a alocação e a liberação de memória para a aplicação. Quando estamos criando um objeto com a instrução new, estamos alocando para ele no *“heap”*, parte da memória utilizada para o armazenamento das estruturas de dados de tamanho e existência não determinada, durante a execução de um programa, espaço na memória física. Assim é alocado na memória até que o *“heap”* não tenha mais espaço. Quando o *Garbage Collector* realiza uma coleção por lixo, ou parte suja da memória, ele verificará se o objeto tem alguma referência na área *“heap”* e se ela está sendo

utilizada, eliminando ela se não estiver.

Vale também ressaltar que o Garbage Collector, eventualmente, trabalha em uma coleção no espaço candidato à liberação para disponibilizar memória. A estrutura *do Garbage Collector* dirá o melhor momento para ele coletar o lixo que está na memória.

Os princípios básicos do coletor de lixo são encontrar objetos de um programa que não serão mais acessados no futuro, esses objetos são tratados como “lixo” e por sua vez ocupam espaço na memória, o que se torna desnecessário e prejudicial ao sistema, visto que aquela parte suja da memória poderia estar sendo usada por outro aplicativo, e desalocar os recursos utilizados por tais objetos. Tornando a desalocação manual de memória desnecessária, e geralmente proibindo tal prática, o coletor de lixo livra o programador de se preocupar com a liberação de recursos já não utilizados, o que pode consumir uma parte significativa do desenvolvimento do software. Também evita que o programador introduza erros no programa devido a má utilização de ponteiros.

Assim o *Garbage Collector* gerencia a alocação e a liberação de memória para a aplicação. Quando estamos criando um objeto com a instrução new, estamos alocando para ele no *“heap”*, parte da memória utilizada para o armazenamento das estruturas de dados de tamanho e existência não determinada, durante a execução de um programa, espaço na memória física. Assim é alocado na memória até que o *“heap”* não tenha mais espaço. Quando o *Garbage Collector* realiza uma coleção por lixo, ou parte suja da memória, ele verificará se o objeto tem alguma referência na área *“heap”* e se ela está sendo utilizada, eliminando ela se não estiver.

Vale também ressaltar que o *Garbage Collector*, eventualmente, trabalha em uma coleção no espaço candidato à liberação para disponibilizar memória. A estrutura *do Garbage Collector* dirá o melhor momento para ele coletar o lixo que está na memória.

NA PRÁTICA:

Diversas linguagens de computador exigem o coletor de lixo, seja como parte da especificação da linguagem (como em Java e C#) ou na implementação (como em linguagens formais tais quais cálculo lambda). Outras linguagens foram desenvolvidas para suportar somente o gerenciamento manual de memória, mas possuem implementações de coletor de lixo disponíveis, como C++.

Para algumas linguagens, há suporte para o gerenciamento manual ou automático de memória ao utilizar diferentes memórias *"heap"* para os objetos manuais ou automáticos, como em Ada. Já D permite o coletor de lixo, mas também a desalocação manual e desabilitar completamente o coletor.

Toda aplicação *.Net* contém um conjunto de *roots*. Cada root corresponde a um local de armazenamento contendo um ponteiro de memória para um objeto de tipo referência. Assim, este ponteiro referência apenas objetos na memória *heap*. Apenas variáveis de tipos referência são consideradas como roots: uma propriedade estática é considerada um *root*, parâmetros de métodos são considerados *roots* e variáveis locais são consideradas como *roots*. Variáveis de tipo valor nunca são consideradas como roots.

Uma propriedade estática é referenciada no *root* durante todo o ciclo de vida de uma *AppDomain*, isto significa que sua referência pode durar para sempre ou até que o *AppDomain* (no qual a propriedade está carregada) seja fechado. Diante disso, é preciso tomar cuidado com os dados que são armazenados em propriedades estáticas, principalmente com a quantidade de itens que são adicionados em coleções referenciadas por propriedades estáticas, pois cada item adicionado é mantido ativo em memória durante toda a existência do *AppDomain*.

Ao contrário das propriedades estáticas, as variáveis e propriedades por referência são marcadas como *"prontas para coleta"* sempre que saem do escopo de execução, isto é, quando se tornam inacessíveis.

Assim, deve-se ter em mente que: tão logo um objeto se torne inacessível, então mais cedo se tornará candidato a ser coletado.

Quando o *garbage collector* começa a trabalhar é mantido como premissa que todos os objetos no *heap* são lixo. Isto é, adota-se que nenhuma variável referência algum outro objeto no *heap*, ou assume-se que nenhum registrador aponta para algum objeto no *heap*, ou que nenhum campo estático referência objetos no *heap*. Assim, após adotada essa premissa, inicia-se a fase de marcação.

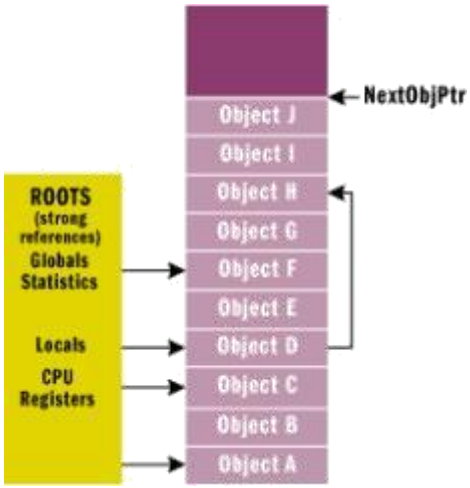
A fase de marcação é responsável por percorrer o *thread stack* verificando todos os *roots*. A cada *root* verifica-se se o root referência no *heap* algum objeto não marcado para coleta, se referenciar então sua referência é marcada como "ainda ativa". Ao fim da fase de marcação pode-se ver um conjunto de referências marcadas e desmarcadas.

Após a fase de marcação inicia-se a fase de compactação, na qual o *garbage collector* compacta o *heap* removendo referências inativas. Naturalmente, a compactação move objetos em memória e invalida variáveis e registradores que contém ponteiros para objetos. Assim, o *garbage collector* é obrigado a revisitar todos os roots e modifica-los para que o *roots* apontem para os endereços corretos.

Quando um processo é iniciado o *CLR* reserva uma região contínua de memória, essa região é chamada de *managed heap*. O *managed heap* contém um ponteiro de memória chamado *NextObjPtr* que indica onde o próximo objeto a ser alocado deve ser posicionado dentro do *managed heap*.

Sempre que executamos o operador *new* forçamos o compilador a emitir uma instrução *newobj* em *IL (Intermediate Language)*, que é responsável por:

- Calcular o número de bytes necessários para a criação do novo tipo;
- Verificar se estes bytes estão disponíveis na memória heap;
- Alocar o espaço na memória heap para o objeto;
- Invocar o construtor;
- Retornar o endereço do novo objeto;
- Avançar o ponteiro NextObjPtr para a próxima posição de memória, na qual o próximo objeto será inserido na memória heap.



Quando uma aplicação executa o operador *new* e não existe espaço suficiente para alocação do objeto dentro do *managed heap* o *garbage collector* entra em ação e começa a coletar lixo.

Vantagens do uso do GC:

1. O coletor de lixo livra o programador de lidar manualmente com o gerenciamento de memória. Como resultado, certas categorias e defeitos de software são eliminadas ou reduzidas. Outro problema é liberar uma região de memória mais de uma vez. Também há certos tipos de vazamento de memória, em que o programa deixa de desalocar memória já não usada de forma a chegar num ponto de esgotamento de memória.
2. Em linguagens que provem alocação dinâmica, o coletor de lixo é essencial para a segurança de memória e está geralmente associado à propriedade de segurança de tipo.
3. Em linguagens orientadas a objetos onde o tempo de carga do objeto é muito alta o coletor pode conter um algoritmo para cache de objetos de forma a devolver objetos que costumam ser utilizados com frequência sem precisar recriá-los.

Desvantagens do uso do GC:

1. Coletores de lixo não conseguem reduzir o risco de vazamentos lógicos, somente físicos.
2. Os GC são processos que consomem recursos computacionais para decidir quais partes da memória podem ser liberadas, enquanto no gerenciamento manual esse consumo é mínimo.

3. No momento em que o objeto é realmente desalocado, ele não é determinístico, o que pode acarretar na variação do tempo de execução de algoritmo em partes aleatórias, o que impensável em sistemas como em tempo real, drivers de dispositivo e processamento de transações.
4. O uso de recursividade atrasa a desalocação automática da memória da pilha de execução até que a última chamada seja completada, aumentando os requisitos de memória do algoritmo.
5. A detecção semântica de objetos a serem desalocados é um problema indefinível para qualquer processo automático, devido ao problema da parada.

FIM