Garbage Collection em Ruby e Critérios de Linguagens de Programação

Andrei P. Koenich[1], **Henrique P. Ribeiro**[1], **Izaias S. L. Neto** [1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{apkoenich,hpribeiro,islneto}@inf.ufrgs.br

Abstract. *There is a loss of efficiency in programming languages when garbage collection exists. Despite this, it is a widely used feature. This article studies garbage collection in Ruby, explaining the basis for the discussion, comparing different configurations of garbage collection and analyzes garbage collection in terms of programming languages criteria.*

Resumo. *Há uma perda de eficiência em linguagens de programação que usam garbage collection. Apesar disso, esse recurso é amplamente utilizado. Este artigo faz um estudo sobre garbage collection em Ruby, explicando a base para uma discussão mais ampla, comparando diferentes configurações de garbage collection e realiza uma análise desse mecanismo em termos de critérios de linguagens de programação.*

1. Introdução

Garbage collection (GC) é um mecanismo automático de desalocação de memória, útil para facilitar a administração de memória dinamicamente alocada em um programa. Apesar da perda de eficiência gerada pela utilização desse mecanismo adicional, os ganhos em utilização da memória e em facilidade de desenvolvimento são relevantes, levando muitos programadores a optar pela utilização de GC. Para lidar com essa perda de eficiência, muitos algoritmos de GC foram criados.

Nesse artigo, a seção 2 exibe uma visão geral de Ruby, enquanto a seção 3 explora a alocação de memória para basear a discussão da seção 4. A seção 4 estuda GC em Ruby, realiza uma comparação entre diferentes configurações do GC e apresenta uma análise em relação aos critérios de linguagens de programação. A seção 5 possui as conclusões finais do artigo.

2. Ruby

Ruby é uma linguagem de programação multiplataforma e *open source*, com uma gramática considerada complexa e, simultaneamente, expressiva. Embora seja considerada principalmente uma linguagem orientada a objetos, Ruby também pode ser utilizada para a criação de programas por meio do uso dos paradigmas procedural ou funcional, sendo, portanto, uma linguagem multiparadigma [Flanagan 2008].

Conforme discutido em [Flanagan 2008], a linguagem foi criada em 1995 por Yukihiro Matsumoto. Com algumas inspirações obtidas a partir das linguagens Python, Lisp, Smalltalk e Perl, a linguagem Ruby possui tipagem dinâmica e forte, além de uma

sintaxe considerada de fácil compreensão, principalmente por parte dos programadores que já possuem alguma experiência com C/C++ ou Java. Também é mencionado que Ruby possui um sistema de *threading* capaz de atuar de forma independente do sistema operacional utilizado pelo programador e, assim como na linguagem Python, todos os elementos são tratados como objetos.

3. Alocação de Memória

Para operar sobre dados, um programa deve alocá-los em memória. Toda e qualquer informação em um programa pode ser representada por uma estrutura de dados. É comum que a memória disponível seja dividida em partes, a fim de facilitar a administração das estruturas. Nesse contexto, existem dois tipos de alocação de memória em linguagens de programação: alocação estática e dinâmica [Knuth 1997].

Em alocação estática de memória, assume-se que todos os tipos de dados alocados possuem um tamanho previamente definido. Dessa forma, o espaço de memória necessário para o armazenamento desses dados é verificado em tempo de compilação. Em alguns casos, esse método de alocação de memória pode gerar desperdício, em razão de uma alocação de espaço superdimensionada, visto que nem sempre é possível determinar, de forma prévia, a quantidade de memória necessária para uma determinada operação [Casavella 2020].

Já em alocação dinâmica de memória, os tamanhos das porções de memória a serem alocadas são determinadas em tempo de execução. Linguagens de programação que permitem ao programador realizar esse tipo de alocação oferecem, portanto, instruções específicas para garantir a alocação e a desalocação de uma certa quantidade de memória. Assim, a tarefa sobre como gerenciar a memória pode ser realizada pelo programador [Casavella 2020].

4. Garbage Collection em Ruby

Garbage collection é um mecanismo automático de desalocação de regiões da memória, anteriormente alocadas de forma dinâmica e referenciadas por meio do uso de ponteiros. Assume-se que uma determinada região de memória contém ”lixo”nos casos em que o conteúdo dessa região não é mais referenciado por nenhum ponteiro, tornando tal conteúdo inacessível. Dessa forma, o GC garante que essa região ”lixo”será desalocada, tornando possível seu reuso posterior. Esse mecanismo permite, portanto, uma otimização da ocupação de memória em tempo de execução, tornando a gerência de memória com alocação dinâmica mais simples e conveniente para os programadores [Wilson 2005].

4.1. Algoritmo de Garbage Collection

Existem vários algoritmos para a implementação de GC. Em Ruby, utiliza-se o algoritmo *tri-color mark-and-sweep* de forma incremental e geracional. O algoritmo 1 mostra um pseudo-código do seu funcionamento.

Inicialmente, todos os objetos são marcados como brancos, indicando falta de informação. Após isso, todos os objetos raízes são marcados com cinza, pois são sempre alcançáveis. Para cada objeto marcado com cinza, marca-se de cinza os objetos brancos que ele referencia. Após isso, o objeto marcado com cinza escolhido inicialmente é marcado com preto, ou seja, está vivo. Esse processo continua iterativamente, até que

Algorithm 1 Tri-Color Mark-and-Sweep

```
1: marque todos os rvalues de BRANCO
2: marque todos os rvalues raíz de CINZA
3: while CINZA rvalues do
4:     obj = escolha um rvalue CINZA
5:     marque todos os rvalues que obj referencia de CINZA
6:     marque obj de PRETO
7: end while
```

não existam mais objetos cinzas. Os objetos restantes serão brancos ou pretos. Objetos brancos são, então, descartados [Karsh 2023].

Este algoritmo atua sobre a *heap* do Ruby. A heap é composta por *pages*, cujo tamanho padrão é 16KB. Cada *page* comporta até 409 *slots*. Cada *slot*, também conhecidos como *rvalues*, tem 40 *bytes* e é responsável por guardar um único objeto.

Para que o GC seja executado, é necessário que não haja mais *slots* vazios presentes em *heap*. Com isso, um ciclo (execução) *minor* do GC acontece. Nesse tipo de ciclo, somente os objetos jovens são analisados. Os objetos jovens são aqueles que sobreviveram a, no máximo, três ciclos de GC e, ao sobreviverem a mais ciclos, tornam-se objetos velhos. Se forem liberados poucos *slots*, é necessário fazer um ciclo *major* (que atua sobre todos os objetos) do GC. Essa ”idade”dos *slots* é que dá o caráter geracional do GC.

Além disso, se um objeto tem mais de 40 *bytes*, ele é guardado na *heap* do sistema operacional (SO). Se houver mais de 16MB de objetos na *heap* do SO, um ciclo *major* é feito. Ademais, se a quantidade de objetos velhos exceder 23.556, então um ciclo *major* é executado [Berkopec 2017].

O GC do Ruby é incremental, ou seja, seu processo é dividido em partes, com o objetivo de suspender o programa em execução a menor quantidade de vezes possível [Karsh 2023].

4.2. Módulo de Garbage Collection

Com o módulo de GC em Ruby, é possível configurar algumas das suas características de operação, efetivamente mudando o comportamento do GC. É disponibilizado um método para executar o GC manualmente, e métodos para desabilitar e habilitar o GC [Flanagan 2008]. Além disso, existe um método para habilitar a compactação nos ciclos *major*, que diminui a fragmentação de memória e reduz a utilização de memória do programa, liberando memória para uso do SO e da *heap* do Ruby.

Neste exemplo (Figura 1), configurou-se o GC para que a compactação fosse sempre executada nos ciclos *major* (linha tracejada). A tarefa executada foi a construção de uma lista encadeada, com os tamanhos indicados no eixo horizontal. Percebe-se que listas com tamanhos maiores demandam um maior tempo na compactação da *heap*.

4.3. Análise em Critérios de Linguagens de Programação

A versatilidade disponibilizada pela configuração do GC é útil para aumentar o desempenho, se utilizada em conjunto com ferramentas de *profiling* [Dymo 2015]. Existe a possi-

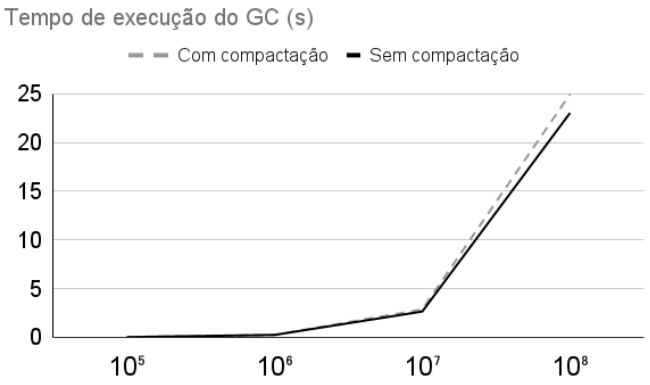

Figura 1. Tempo de execução do GC com e sem compactação

bilidade de perda da legibilidade do código, em função de chamadas dispersas a métodos de GC. Apesar disso, a sua utilização cuidadosa pode levar a um código mais fácil de entender, em relação às linguagens que não possuem GC, e mais eficiente em termos de memória em relação à não utilização de GC. Aplicações com demanda de baixa latência não críticas, que requerem GC diluído no tempo, poderiam usufruir dessas características específicas de GC em Ruby.

5. Conclusão

Esse artigo fornece detalhamento suficiente para o entendimento do GC e da sua utilidade em uma linguagem de programação. Com essa base, foi possível investigar aspectos do GC em Ruby e examinar como a configuração do GC afeta seu funcionamento. Fundamentando-se nos conceitos anteriores, foi possível observar algumas vantagens na utilização de um GC configurável, em aplicações de baixa latência não críticas. Essas vantagens incluem um código mais fácil de entender, em relação às linguagens que não possuem GC, e mais eficiente em termos de memória, em relação à não utilização de GC.

Referências

Berkopec, N. (2017). Practical Garbage Collection Tuning in Ruby. `https://www.speedshop.co/2017/03/09/a-guide-to-gc-stat.html`.

Casavella, E. (2020). Alocação Dinâmica em C. `https://linguagemc.com.br/alocacao-dinamica-de-memoria-em-c/`.

Dymo, A. (2015). *Ruby performance optimization*. Pragmatic Bookshelf.

Flanagan, David e Matsumoto, Y. (2008). *The Ruby programming language*. O'Reilly Media, Sebastopol, CA.

Karsh, P. (2023). Deep Dive into Garbage Collection in Ruby 3: Incremental Garbage Collection. `https://t.ly/QsSye`.

Knuth, D. E. (1997). *The art of computer programming*. Addison Wesley, Boston, MA, 3 edition.

Wilson, P. R. (2005). Uniprocessor garbage collection techniques.