I. Garbage Collection

Zoltán Majó / Christoph Denzler

In vielen modernen Programmiersprachen werden Objekte explizit angelegt, aber nicht mehr explizit freigegeben. In komplexen (verteilten oder zumindest verteilt entwickelten) Systemen kann oft nicht mehr genau festgelegt werden, ab wann ein Objekt nicht mehr verwendet wird und damit freigegeben werden kann. Diese Arbeit übernimmt der Garbage Collector (dt.: Müllsammler), kurz GC.

1. Grundlagen (einfacher Mark & Sweep Garbage Collector)

Wie können alle Objekte gefunden werden, die nicht mehr referenziert werden? Indem man alle Objekte markiert, die noch referenziert werden! Alle nicht markierten Objekte werden demnach auch nicht mehr referenziert und sind somit Müll (engl. Garbage) der eingesammelt (engl. to collect) werden kann. Ein Mark & Sweep Garbage Collector arbeitet in zwei Phasen (dabei werden alle anderen Threads angehalten):

1. In einer ersten Phase (*Mark Phase*) werden alle erreichbaren Objekte markiert.
2. In der zweiten Phase (*Sweep Phase*) werden die nicht markierten Objekte eingesammelt und die Markierungen auf den anderen Objekten gelöscht.

Simpel! Aber im Detail stellen sich einige Fragen:

- Wo beginnt die Mark Phase mit der Suche nach erreichbaren Objekten? (1.1)
- Wie muss der Speicher verwaltet werden, damit in der Sweep Phase nicht markierte Objekte gefunden werden? (1.2)
- Was wird beim „einsammeln“ von nicht markierten Objekten genau gemacht? (0)
- Wie werden eigentlich neue Objekte im Speicher allokiert? (0)

1.1 Root Set

Die Mark Phase ist nichts anderes als eine Traversierung eines Graphen. In diesem Graph sind die Objekte Knoten und Referenzen sind Kanten. Ein Objekt wird markiert sobald es zum ersten Mal erreicht wird. Das bedeutet, dass jedes Objekt neben den – für die Programmiererin sichtbaren – Nutzdaten auch noch Verwaltungsdaten enthalten muss. Zum Beispiel ein Flag für den GC. Die Startreferenzen für die Speichertraversierung werden auch Wurzeln (engl. root) genannt. Die Menge aller Startreferenzen wird als *root set* bezeichnet. Generell sind dies: lokale Variablen auf den Call-Stacks, globale Variablen und Prozessorregister. In Java:

- Aktive Threads: der *Call-Stack enthält lokale Variablen*. Diese können auf Objekte zeigen.
- Statische Variablen: *Variablen die static deklariert wurden* sind globale Variablen.
- Sehr *kurzfristig angelegte Objekte* werden nur über Register referenziert. Beispiel das folgende Integer-Objekt unmittelbar vor der Rückgabe: `return new Integer(5);`
- Java erlaubt über das Java Native Interface (JNI) auch nativen Code (z.B. C++) auszuführen. Dieser Code kann Java-Objekte anlegen und darauf zugreifen. Man nennt diese Referenzen aus dem nativen Code auch JNI-Referenzen. Da ein Objekt evtl. nur noch über *JNI-Referenzen* gehalten wird, gelten diese als GC Roots.

1.2 Speicherverwaltung

Jedes Objekt muss in seinen Verwaltungsdaten seine eigene Bruttogrösse (also Nutz- plus Verwaltungsdaten) speichern. Somit kann der gesamte Speicher linear traversiert werden. Zu den Verwaltungsdaten können auch noch folgende Informationen gehören: Pointer auf Klassenobjekt, GC-Flag (erreichbar), etc.

1.3 Free List

Freie Objekte werden in einer eigenen Liste verwaltet, die sog. *free list*. Die free list kann optimiert werden, indem unmittelbar benachbarte freie Speicherbereiche zu einem grösseren Bereich verschmolzen werden.

Wird Speicher für ein neues Objekt benötigt, so muss die free list nach einem Speicherblock durchsucht werden, der das neue Objekt aufnehmen kann. Der freie Block muss dann um die Brutto-Objektgrösse verkleinert werden.

2. Lebensdauer von Objekten

Ein simpler Mark & Sweep GC ist wegen der aufwendigen free list Verwaltung und der drohenden Fragmentierung wenig geeignet für die JVM. Denn wie sich gezeigt hat, leben sehr viele Objekte nur sehr kurz. Die Verteilung der Lebensdauer von Objekten in einer typischen Java-Applikation sieht wie folgt aus:

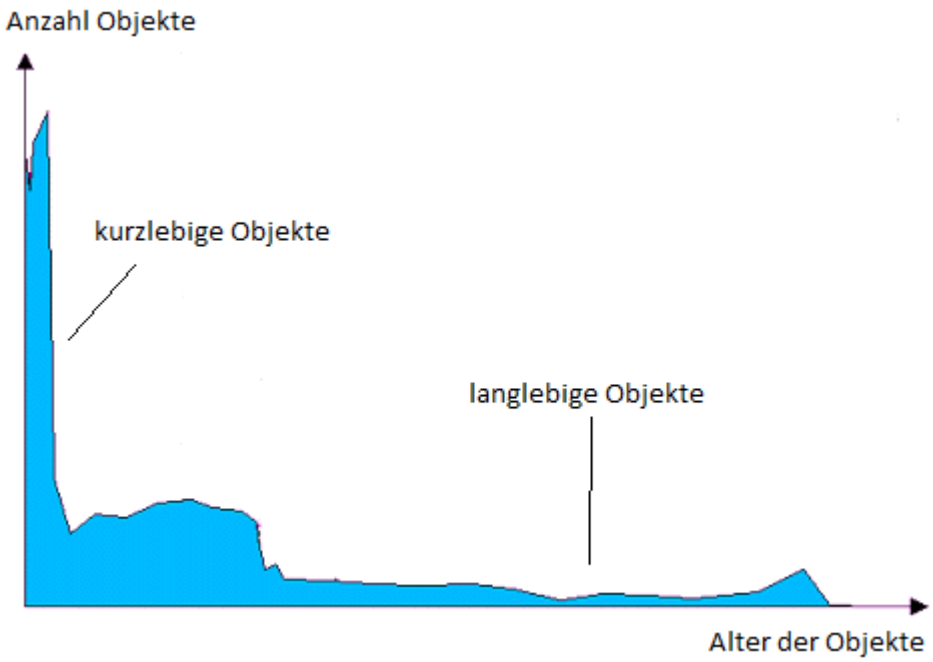


Abbildung 1: Objektlebensdauer

2.1 Kurzlebige Objekte

- Die meisten Objekte können kurz nach ihrer Erzeugung wieder abgeräumt werden.
- Kurzlebige Objekte werden oft lokal in einem Block angelegt

Beispiele: Generierte Objekte wie z.B. Iteratoren für for-Schleife oder Wrapper für Autoboxing.

Die vielen kurzlebigen Objekte belasten den GC und müssen deshalb möglichst effizient abgeräumt werden können. Aus diesem Grund können andere Programmiersprachen auch Objekte auf dem Stack allokieren: z.B. in C# die Structs. Diese Objekte müssen dann vom GC überhaupt nicht mehr berücksichtigt werden, da sie mit dem Abbau des Stacks automatisch „entsorgt“ werden.

2.2 Mittellang lebende Objekte

- Ein signifikanter Teil der Objekte hat eine mittlere Lebensdauer.
- Objekte die nicht auf dem Stack abgelegt werden können, da sie Methodenaufrufe überleben müssen.

Beispiele: Session State, Statistische Daten, gecachte Daten

2.3 Langlebige Objekte

- Wenige Objekte leben sehr lange, manchmal so lange wie die JVM läuft.
- Dies sind Objekte die beim Programmstart angelegt werden und bis zu dessen Ende vorhanden sein müssen.

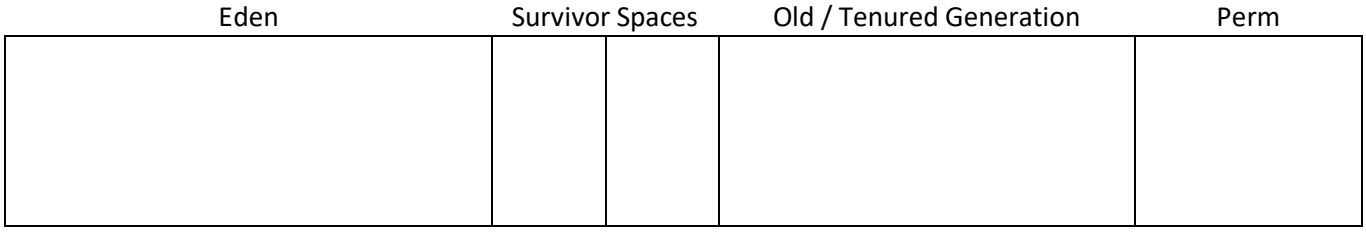
Beispiele: Thread Pools, Singletons, alle Framework-Klassen in welchen eigene Klassen eingebunden werden also z.B. Servletcontainer

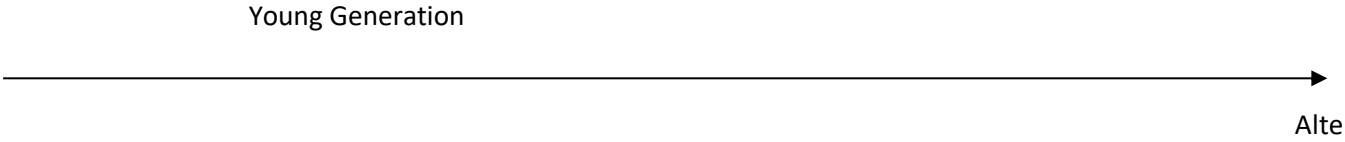
3. Generational Garbage Collection

Kernidee:

- Diese typische demographische Struktur für den Garbage Collector nutzen.
- Statisch den Heap in Bereiche aufteilen für verschiedene Altersgruppen.
- Dynamisch: in den verschiedenen Bereichen unterschiedliche GC-Algorithmen anwenden.

3.1 Statische Heapstruktur





Legende:

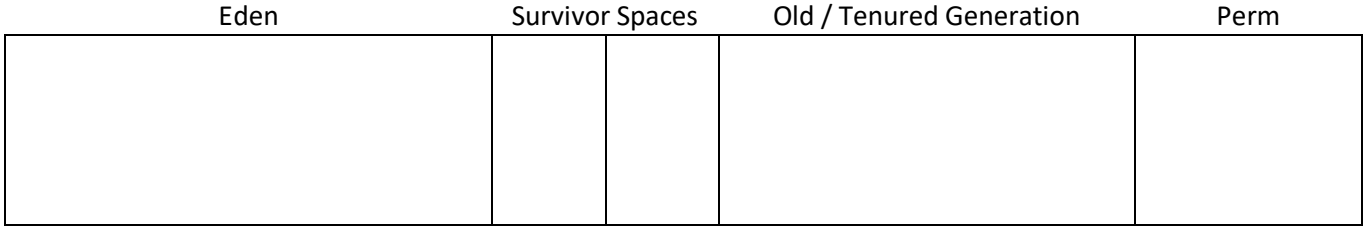
Perm: Die Permanent Generation hält alle Class-Objekte der geladenen Klassen sowie Objekte, welche die JVM für ihre Arbeit benötigt.

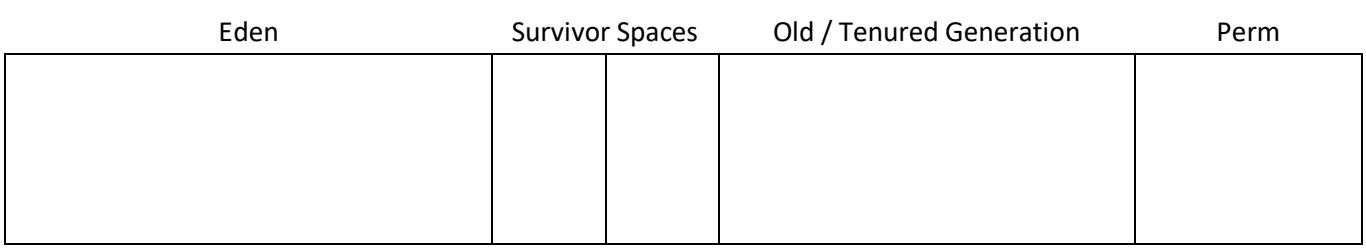
Eden: Nur hier werden neue Objekte allokiert.

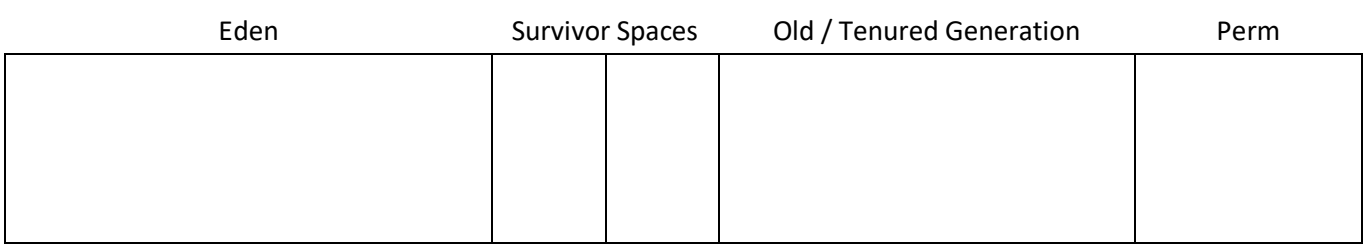
freespace, Beginn des freien Speichers, Adresse des nächsten Objektes.

Referenz, die zum Root Set gehört.

3.2 Allokation







Neue Objekte werden immer im Eden Space angelegt. Allokation ist billig, da Speicher fortlaufend vergeben werden kann.

3.3 Mark & Copy Algorithmus

Eden	Survivor Spaces		Old / Tenured Generation	Perm

Eden	Survivor Spaces		Old / Tenured Generation	Perm

Wenn Eden voll wird:

- Alle Objekte die noch am Leben (= erreichbar) sind werden in einen Survivor Space kopiert. Referenzen müssen angepasst werden.
- Danach gilt der Eden Space wieder als freier Speicher, keine weitere Arbeit nötig!
- Ein kopierender GC wird auch als *Scavenger* bezeichnet (to scavenge = spülen, reinigen)

Vorteile

- Die meisten Objekte sind tot. Nur wenige Objekte müssen kopiert werden.
- Kontinuierlicher freier Speicher, schnelle Allokation, keine Fragmentierung

Nachteile

- Grösserer Memory-Footprint da immer ein Survivor Space leer (= ungenutzt) ist.

3.4 Aging

Eden	Survivor Spaces		Old / Tenured Generation	Perm

Eden	Survivor Spaces		Old / Tenured Generation	Perm

- Einer der Survivor Spaces ist immer leer. GC läuft auf Eden und im gefüllten Survivor Space.
- Danach sind alle „überlebenden“ Objekte im vorher leeren Survivor Space. Eden und der alte Survivor Space gelten als freier Speicher.
- Überlebende Objekte werden einige Male zwischen den Survivor Spaces hin und her kopiert (Aging). Weil in der Old Generation viel Aufwand für die GC betrieben werden muss, will man so verhindern, dass junge und mittelalte Objekte zu schnell in die Old Generation kommen.
- Wie oft Objekte zwischen den Survivor Spaces kopiert werden, wird bestimmt durch
    - Grösse des Survivor Space (beide sind genau gleich gross)
    - Anzahl Objekte in Eden und im alten Survivor Space
    - Age Threshold (konfigurierbar)

3.5 Promotion

Eden	Survivor Spaces		Old / Tenured Generation	Perm

Eden	Survivor Spaces		Old / Tenured Generation	Perm

- Irgendwann kann ein Survivor Space nicht mehr alle Young Generation Objekte aufnehmen. Es kommt zur Promotion in die Old Generation.
- Die ältesten Objekte werden in die Old Generation kopiert.

3.6 Minor Garbage Collection

- Die bisher beschriebenen Schritte (3.2- 3.5) werden als *Minor Garbage Collection* bezeichnet:
  - Eden → Survivor Space
  - Survivor Space → Survivor Space
  - Survivor Space → Old Generation
- Eine *Full* oder *Major Garbage Collection* wird immer durch eine Minor Garbage Collection ausgelöst.
- Minor Collections kommen häufiger vor als eine Major Garbage Collection
  - das genaue Verhältnis hängt vom Verhalten der Applikation ab und
  - von der Grösse der verschiedenen Heap-Bereiche (Tuning-Potential!)
- Minor Collections benötigen weniger Zeit als Major Collections

3.7 Mark & Compact Algorithmus

Eden	Survivor Spaces		Old / Tenured Generation	Perm

Eden	Survivor Spaces		Old / Tenured Generation	Perm

Major Collections führen einen Mark & Compact GC auf der Old Generation durch:

- lebende Objekte werden markiert und
- innerhalb der Old Generation kopiert um eine Fragmentierung zu vermeiden.

Vorteile

- Memory Footprint moderat, kein unbenutzter Survivor Space

Nachteile

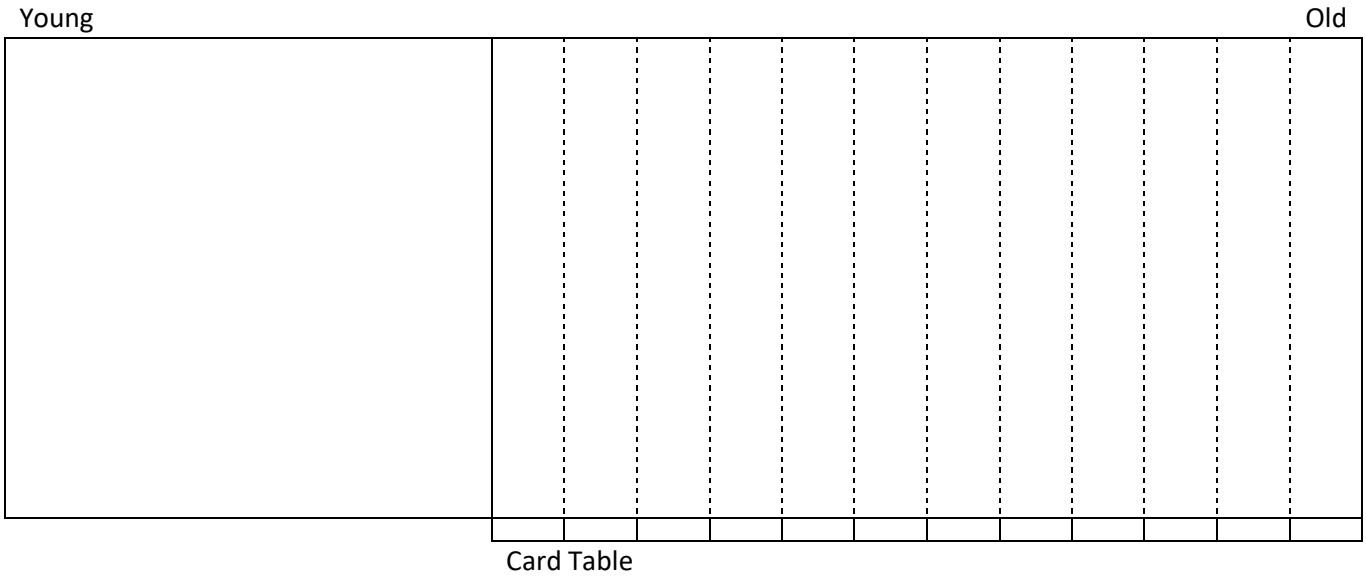
- langsam, da das Kompaktieren aufwändig ist
- Dies bedeutet relativ lange Pausen für die Applikation

Alternative**: Inkrementeller GC**

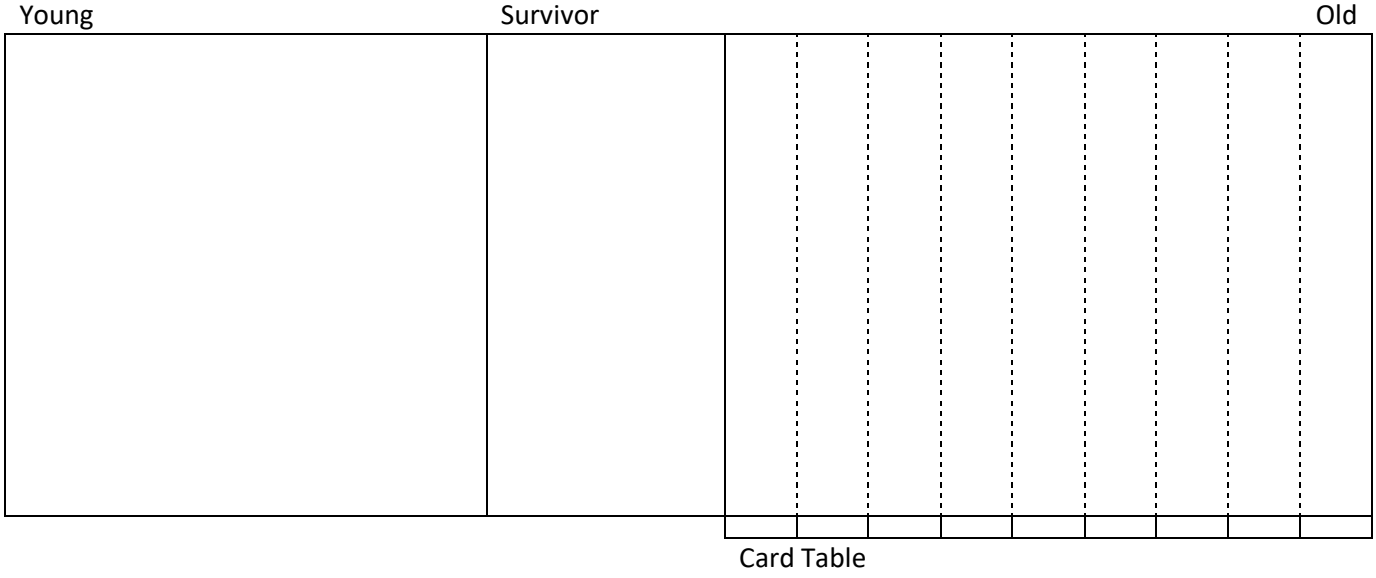
- Räumt inkrementell nur einen Teil der Old Generation auf.
- Aufwändig, da zusätzlicher Verwaltungsaufwand betrieben werden muss.
- Kürzere Pausen für die Applikation, aber merklich geringerer Durchsatz! (GC läuft häufiger)

3.8 Old-to-Young Referenzen

Ein Problem stellen Referenzen von Objekten aus der Old-Generation auf Objekte aus der Young-Generation dar. In der Mark-Phase eines Minor GC müssen auch Objekte der Old-Generation mit berücksichtigt werden:



- Old Generation Heap wird in Chunks aufgeteilt, sogenannte *Cards*.
- Wird eine Referenz in einem Objekt der Old Generation verändert, so wird die Card in welcher sich das Objekt befindet als *dirty* markiert. Die Markierung wird auch vorgenommen wenn ein Objekt aus dem Survivor Space in die Old Generation kopiert wird, und dieses Objekt noch Referenzen auf junge Objekte hält.
- Während einer Minor Collection, werden in der Mark-Phase nur die dirty Bereiche der Old Generation gescannt um Root-Referenzen zu finden.
- In der Copy-Phase einer Minor Collection ist es wichtig alle Old-to-Young-Referenzen nachzuführen. Auch hierbei hilft die Card Table, sie enthält ja alle Bereiche die eine Old-to-Young Referenz enthalten:



3.9 Schlussfolgerungen

Generational GC setzt verschiedene Algorithmen für verschiedene Generationen von Objekten ein. Die Algorithmen können so ihre jeweiligen Stärken ausspielen.

- Mark & Copy GC auf der Young Generation:
  - Schnell, benötigt aber viel Speicher
  - Wird häufig ausgeführt
- Mark & Compact GC auf der Old Generation:
  - Langsam, aufwändig, kein zusätzlicher Speicherbedarf
  - Wird selten ausgeführt

Wann lohnt sich ein Generational Garbage Collector?

4. Parallele und Nebenläufige Garbage Collection

Ein unangenehmer Umstand ist bei allen besprochenen Algorithmen, dass sie sogenannte Stop-the-World (STW) Phasen benötigen. Das heisst, die gesamte VM steht still und macht nur noch Garbage Collection. Dies kann zu unangenehmen Effekten führen (z.B. die Darstellung ruckelt). Eines vorneweg: in den gängigen VMs arbeiten *alle* GCs mit STW-Phasen. Es ist aber möglich diese Phasen möglichst kurz und möglichst selten werden zu lassen.

4.1 Parallele und nebenläufige GCs – Unterschiede

Parallel heisst, das der Garbage Collector selbst in mehreren Threads läuft: dabei werden die CPU-Cores optimal ausgenutzt, kein Prozessor/Core steht still während GC.

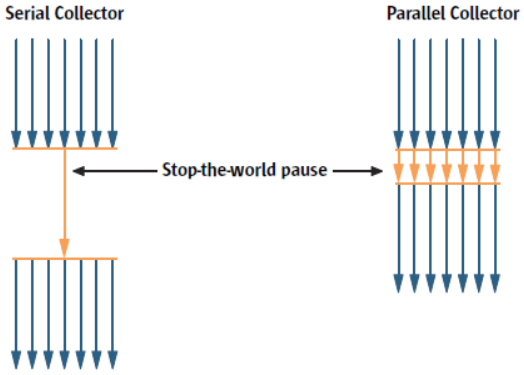


Abbildung 2: Serielle vs parallele Collection

Nebenläufig heisst, dass es weniger STW-Phase braucht, da der GC parallel zur Applikation läuft. Dies ist nicht mit allen GC-Phasen möglich!

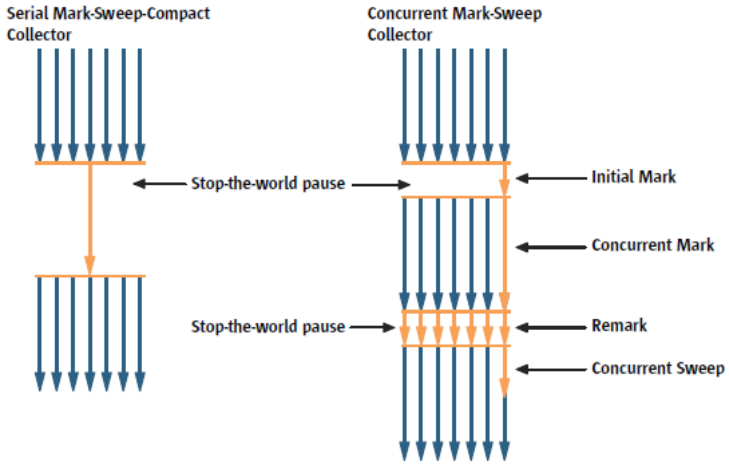


Abbildung 3: Serielle vs nebenläufige Collection

4.2 Parallel Young Generation Garbage Collection

Die Herausforderung liegt hier vor allem in der Copy-Phase:

- Mehrere Threads markieren gleichzeitig erreichbare Objekte im from-Space (=Eden und alter Survivor Space). Sie hinterlassen dabei Forwarding-Referenzen auf den to-Space (= neuer Survivor Space)
- Mehrere Threads allokieren dabei gleichzeitig Objekte im to-Space, es wird eine Synchronisation unter den GC-Threads notwendig. Damit wird die Objekt-Allokation zum Flaschenhals.

Lösung: Für jeden Thread einen grösseren Bereich bereitstellen in welchen nur *ein* Thread kopieren darf. Diese werden *thread local allocation buffers (TLAB)* genannt.

4.3 Parallel Old Generation Garbage Collection

Seit der JVM 5.0 Update 6 gibt es einen parallelen Mark-Sweep-Compact Algorithmus. Er ist meistens parallel, und hat immer noch STW-Phasen. Er arbeitet in drei Phasen (siehe auch Abbildung 4):

1. **Marking Phase** (parallel)
   Jede Referenz im Rootset wird durch einen eigenen Thread bearbeitet. Die Implementation achtet darauf, dass das Markieren in einer atomaren Operation vorgenommen wird. Dadurch müssen sich die verschiedenen Threads bei Ihrer Arbeit nicht synchronisieren. Der Old-Generation Heap wird zudem logisch in gleich grosse Regionen aufgeteilt. Pro Region wird Buch geführt über die Grösse und Adresse der lebenden Objekte.
2. **Summary Phase** (seriell)
   Jetzt wird jede Region bezüglich ihrer Dichte untersucht, d.h. das Verhältnis lebender Bytes zur Grösse der Region. Dabei wird von folgender Annahme ausgegangen: Der Old-Generation Heap wird von links nach rechts (konkret aufsteigend nach Addressen) gefüllt. Zudem haben vorangegangene Kompaktierungen den Speicher nach links kompaktiert. Es ist also vernünftig anzunehmen, dass der linke Teil des Heaps dichter gefüllt ist als der rechte Teil. Und mit dem Wissen um die demographische Verteilung der Objekte (siehe Abbildung 1) gilt: je weiter links das Objekt, desto älter, und desto wahrscheinlicher ist es noch am Leben. Dieses Wissen wird nun für eine Performance-Optimierung genutzt: Von links her wird die erste Region bestimmt, bei der sich eine Kompaktierung überhaupt lohnt. Der Bereich links davon wird *dense prefix* genannt. Darin wird keine weitere Arbeit mehr verrichtet.
   Zudem kann der Collector in dieser Phase bestimmen wie viel Speicher in jeder Region nach dem Kompaktieren verbleibt, und damit kann er auch bestimmen welche Regionen wohin kopiert werden müssen. Es kann also für jede Ziel-Region angegeben werden von welcher Quell-Region sie gefüllt wird. Und für jede Quell-Region kann bestimmt werden, in welche Ziel-Region die Objekte kopiert werden.
3. **Compaction Phase** (parallel)
   Regionen die ohne Synchronisation mit anderen Threads kompaktiert werden können, werden nun auf die GC-Threads aufgeteilt. Dies sind Regionen, die nicht andere Regionen füllen. Entweder weil sie in sich selbst, oder weil sie in komplett leere Regionen kompaktiert werden. Nachdem ein Thread eine Region gesäubert hat startet er sofort damit, sie mit Objekten aus seiner Quell-Region zu füllen – die wurde ja in der Summary-Phase schon ermittelt. Da bekannt ist, dass kein anderer Thread von der Quell-Region liest oder in die Ziel-Region schreibt kann der Thread seine Arbeit ohne Synchronisation verrichten.

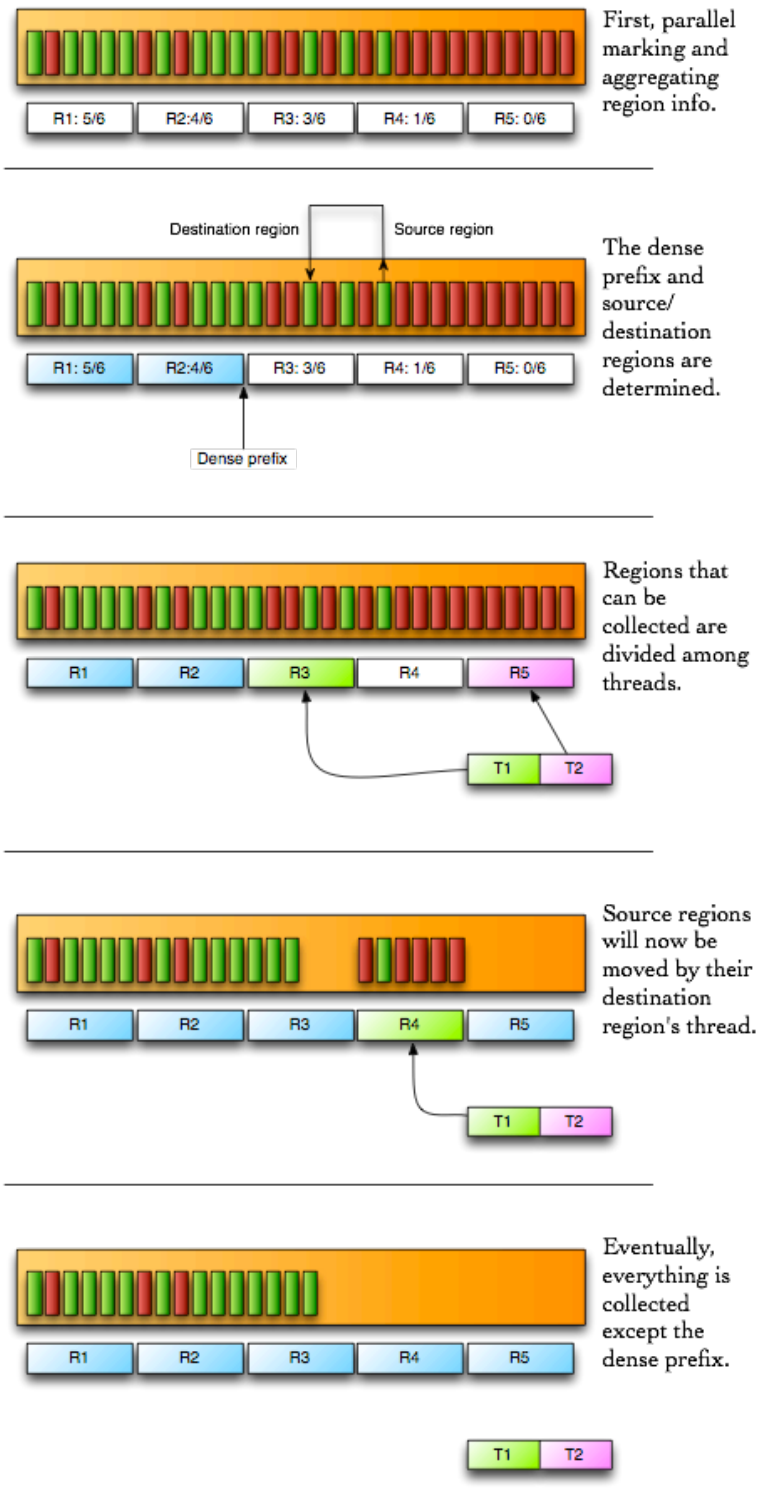


Abbildung 4: Parallel Old Generation GC

Der parallele Algorithmus hilft den Durchsatz hoch zu halten. Zudem nützt er sämtliche zur Verfügung stehenden CPUs aus, d.h. er weist auf einer Multi-Core- oder Multi-CPU-Maschine kürzere STW-Zeiten auf, was wiederum den Durchsatz der Applikation erhöht. Allerdings können die STW-Zeiten immer noch spürbar sein. Auf einer Single-CPU oder Single-Core-Maschine bringt dieser Algorithmus nichts!

4.4 Concurrent Old Generation Garbage Collection

Das Prinzip dieser Collection ist in Abbildung 3 dargestellt. Dieser Garbage Collector wird auch Concurrent Mark and Sweep oder kurz CMS-Collector genannt. Seine Phasen sind:

1. Initial Marking (seriell)
   Hält die Welt an um eine initiale Menge an Objekten zu markieren, die *direkt* von den Root-Referenzen her erreichbar sind.
2. Marking (concurrent)
   Nun werden in einem nebenläufigen Thread alle Objekte markiert, die transitiv von der initialen Objektmenge erreichbar sind. Da die Applikation während dieser Zeit weiterläuft und Referenzen verändert, ist nicht garantiert, dass am Ende dieser Phase alle lebenden Objekte auch markiert sind.
3. Remarking (parallel)
   Das Markieren wird nun abgeschlossen indem alle Objekte, die während der vorherigen Phase verändert wurden nochmals besucht werden. Da die Remark-Phase aufwändiger ist als das Initial Marking, wird diese Arbeit mit mehreren Threads parallel ausgeführt. Dazu wird die Welt nochmals angehalten.
4. Sweep (concurrent)
   Jetzt da alle lebenden Objekte markiert sind, können die toten abgeräumt werden. Dies geschieht nebenläufig und deshalb ist es auch nicht möglich den Speicher zu kompaktieren.

Der CMS-Collector versucht in erster Linie die STW-Phasen zu vermeiden bzw. soweit zu verkürzen, dass sie möglichst nicht mehr auffallen. Die Nebenläufigkeit hat allerdings ihren Preis:

- Geringerer Durchsatz, da der Verwaltungsaufwand grösser ist.
- Fragmentierung, weil keine Kompaktierung stattfinden kann. Zudem muss aufwändig eine *freelist* unterhalten werden, in der man sich die freien Speicherbereiche merkt. Aneinanderstossende freie Bereiche müssen erkannt und verschmolzen werden.
- Teurere Allokation in der Old Generation: um ein neues Objekt anlegen zu können, muss die freelist nach einem passenden Block durchsucht werden. Zur Optimierung kann eine VM eine Statistik über die populärsten Objektgrössen führen und dann Speicherregionen mit gleichgrossen Objekten verwalten.
- Eine Major-Collection sollte möglichst nicht erst bei Bedarf, sondern schon bei einem bestimmten Füllgrad durchgeführt werden.
- Falls alles nichts nützt, ist der Fallback trotzdem eine normale serielle Garbage Collection!

5. Ziele des Garbage Collection Tunings

Welches sind eigentlich die Ziele, die man verfolgen kann mit GC Tuning? Der Garbage Collector ist nur eine Komponente im komplexen Umfeld des Performance Tunings. Verschiedene Applikationen erfordern auch verschiedenes GC-Verhalten. Es gibt keine Standardlösung!

Betrachten wir die Garbage Collection aus zwei verschiedenen Gesichtspunkten: dem Benutzer und dem Software-Ingenieur:

5.1 Benutzeraspekte

Durchsatz *(engl. Throughput):* Der Prozentsatz der Zeit, die die Applikation nicht mit GC verbringt. Also die Zeit in der die Applikation selbst läuft in Bezug zur verstrichenen Zeit.
Der Durchsatz ist wichtig in Applikationen mit asynchronem Input (Batch, Servers, Enterprise Applications).

Pausen: Die Zeit in welcher die Applikation „stehen bleibt" weil gerade eine GC läuft.
Die Pausen sind relevant in Applikationen mit synchronem Input (z.B. interaktive Applikationen) oder bei (Beinahe-) Echtzeitsystemen (z.B. Telekommunikation, Musik, Video).

5.2 Software-Engineering-Aspekte

Footprint: Der von der Applikation benötigte (virtuelle) Speicher.
Relevant in Applikationen mit limitiertem Speicherumgebungen (PDAs, Smartphones…)

Scalability: Die Fähigkeit Performanceverbesserungen zu erreichen, indem mehr Ressourcen zur Verfügung gestellt werden.
Relevant in grossen Applikationen mit viel Speicher und CPU-Leistung

Promptness: Zeit die vergeht zwischen dem Moment in welchem ein Objekt unerreichbar wird, bis zum Moment wo dessen Speicher wieder verwendet werden kann.
Vermeiden von *„floating garbage"*

5.3 Vorbeugen statt Tunen

Folgende Programmiertechniken helfen die Arbeit des GC zu beschleunigen:

- Anzahl und Grösse der Objekte reduzieren. (use statics, flatten objects)
- Die Lebensdauer von Objekten verkürzen (prevent memory leaks)
- Referenzen zwischen den Generationen vermeiden (follow: age ignores youth)
- Zusätzliche Aufwände vermeiden (Soft-, Weak-References, Finalizers)

Beachte: dies sind Ratschläge aus der Sicht des GC-Performance-Optimierers. Diese Ratschläge können aber Programming Guidelines oder Best Practices entgegenlaufen! Manchmal widersprechen sie sich sogar. Beispiel: Soft- und Weak-References helfen gerade dabei die Lebensdauer von Objekten zu verkürzen!

Eine Applikation ist für die Benutzer geschrieben und nicht für die Entwickler. Also werden wir im Folgenden die beiden Aspekte Durchsatz und Pausen zu optimieren versuchen. Beachte: dies sind in der Regel gegensätzliche Ziele und können nicht beide gleichzeitig optimal eingestellt werden.

6. Speicher optimieren

Generell bedeutet mehr Speicher auch eine bessere GC-Performance. Die erste Regel beim Durchsatz optimieren lautet daher:

1
So viel Speicher wie möglich zur Verfügung stellen

Da der GC seltener läuft erreicht man dadurch mehr Durchsatz. Allerdings muss der GC auch einen grösseren Heap bearbeiten, d.h. längere Pausen.

Der Heap kann sich automatisch dem aktuellen Speicherverbrauch anpassen. Diese Dynamik kostet aber immer auch Durchsatz und bedeutet auch längere Wartezeiten. Um dies zu vermeiden ist es ratsam bei Messungen die minimale und maximale Heapgrösse beim Start der JVM gleich gross zu setzen. Bsp: -Xms64m –Xmx64m.

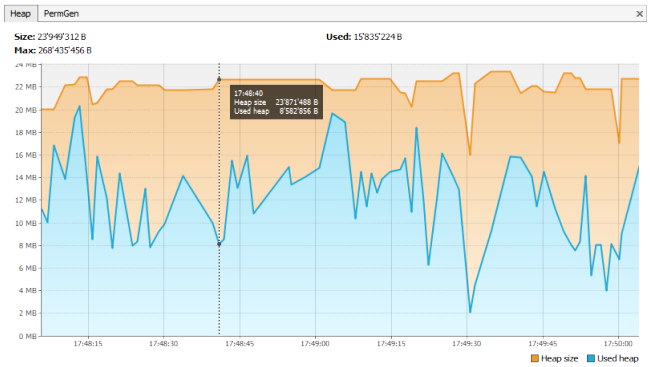


Abbildung 5: dynamischer Heap

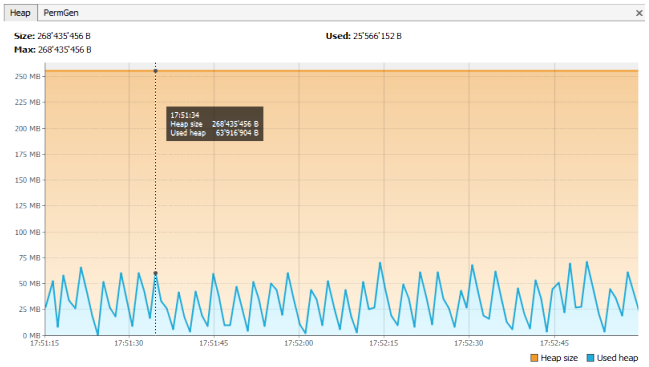


Abbildung 6: statischer Heap

In einem produktiven System sollte aber der JVM eine gewisse Flexibilität eingeräumt werden. Warum?

7. Durchsatz optimieren

Es können sehr viele Hebel angesetzt werden um den Durchsatz zu erhöhen. Einige davon werden hier erwähnt.

7.1 Grösse der Generationen einstellen

Die optimale Grösse der Generationen hängt von der demographischen Verteilung der Objekte ab. Die zweite Regel beim Durchsatz optimieren lautet

2
Vergrössere die Young Generation

Warum?

Welchen Effekt hat eine grössere Young Generation?

Eine zu kleine Old Generation hat aber zur Folge, dass häufiger in der Old Generation eine GC nötig wird
→ im schlimmsten Fall triggert jede Minor GC eine Full GC!

7.2 Das Altern kontrollieren

Eine Minor-GC ist billiger als eine Full-GC. Es lohnt sich daher die Objekte in den Survivor-Spaces ein paarmal hin und her kopieren zu lassen. Eine Promotion in die Old Generation ist teuer, da mit deren Memory-Management interagiert werden muss (Platz allokieren, Card Table, TLAB …).

Wenn also ein grosser Eden-Space und gleichzeitig grosse Survivor-Spaces vorhanden sind, so können die meisten Objekte in der Young-Generation sterben ohne eine Promotion zu durchlaufen. Zudem ist auch weniger häufig eine Minor-GC notwendig.

Positiv: mittelalte Objekte können in den Survivor-Spaces sterben, keine Promotion

Negativ: alte Objekte werden unnötig häufig hin und her kopiert.

Für den Entwickler ergeben sich noch folgende Aspekte:

- Grosse Survivor-Spaces verschwenden Speicher → grosser Footprint
- Minor GC ist nicht nebenläufig → behindert Scalability

Die dritte Regel der Durchsatzoptimierung lautet also:

3
Lass Objekte wenn immer möglich in der Young-Generation sterben

Frage: Über Optionen kann eingestellt werden, dass Objekte immer oder gar nie eine Promotion durchlaufen. Warum ist letzteres keine gute Idee?

7.3 Parallel (Young) GC

-XX+UseParallelGC schaltet den parallelen Minor-GC ein. Er kann *nicht* mit dem CMS-Collector kombiniert werden. Zudem kann mit –XX:ParallelGCThreads=<nn> die Anzahl Threads angegeben werden, die der GC verwenden darf. Default ist die Anzahl CPUs. Falls nur eine CPU vorhanden ist, wird automatisch der serielle GC verwendet. Parallele GC macht nur Sinn auf Multi-CPU-Rechnern, daher die vierte Regel der Durchsatzoptimierung:

4
Verwende parallele Garbage Collection auf Multi-CPU Maschinen

7.4 Old Generation GC einstellen

Hier ist die Regel ganz einfach: Je weniger algorithmischer Overhead betrieben werden muss, desto schneller ist der GC. Dies bedeutet, dass der herkömmliche serielle Mark-and-Sweep-Collector der schnellste ist. Allerdings muss man dabei längere Pausen in Kauf nehmen.

8. Pausenzeiten reduzieren

Die Pausenzeiten zu reduzieren ist schwieriger als den Durchsatz zu optimieren. Eine grosse Young-Generation verringert zwar die Anzahl Minor-GCs aber auf Kosten deren Dauer, da mehr Speicher aufgeräumt werden muss.

1
Erhöhe den Speicher der Young-Generation soweit wie es die Pausenzeiten zulassen

8.1 Der Concurrent Mark and Sweep Collector (CMS)

Der CMS-Collector wurde entwickelt mit dem Ziel die Pausenzeiten möglichst tief zu halten. Er ist allerdings sinnlos auf einer Single-CPU-Maschine und bringt praktisch nichts auf einer Dual-CPU-Maschine. Je mehr CPUs zur Verfügung stehen, desto mehr Sinn macht der CMS-Collector. In der Old-Generation ist auf folgende Regel zu achten:

2
Passe den Belegungsschwellenwert (occupancy fraction) der Grösse der Old-Generation an. Bei einer kleinen Old-Generation sollte er verringert werden.

Warum?

8.2 Threads für parallele und nebenläufige Ausführung

Die JVM erlaubt über Optionen die Einstellung der Anzahl paralleler Threads in der STW-Phase sowie die Anzahl nebenläufiger (concurrent) Threads in der Mark-Phase. Parallele und nebenläufige Algorithmen lohnen sich, sobald mehrere CPUs zur Verfügung stehen.

3 *Verwende parallele Garbage Collection auf Multi-CPU Maschinen* *Verwende nebenläufige Garbage Collection auf Multi-CPU Maschinen*

Falls mehr als 4 CPUs zur Verfügung stehen, lohnt es sich auch, die GC inkrementell erledigen zu lassen. Dies kostet zwar Durchsatz verringert aber die Pausenzeiten.

4 *Verwende den inkrementellen Modus im nebenläufigen GC*

Dasselbe gilt für auch für den parallelen kompaktierenden GC. Falls der CMS nicht schnell genug ist, oder zu einer inakzeptablen Fragmentierung führt, dann:

5 *Verwende den parallelen kompaktierenden GC*

9. Ausblick

Oracle hat mit den Jahren immer ausgefeiltere GC-Algorithmen entwickelt. Doch ein Ziel blieb stets unerreicht: Ein Garbage Collector, der garantieren kann, dass eine Pause nicht länger als eine bestimmte Zeit dauert. Oracle hat deshalb einen völlig neuen GC entwickelt, den sogenannten G1. Dies steht für Garbage First. Der G1 Collector beruht auch auf dem Generationenprinzip, arbeitet aber auf einem Heap, der in viele gleichgrosse Bereiche aufgeteilt ist. Jeder dieser Bereiche ist entweder eine Young-Region oder eine Old-Region und wird entsprechend behandelt. Die Algorithmen sind um einiges komplexer als in den bisherigen GCs, und der Einsatz des G1 lohnt sich erst ab 4 CPUs.

Trotz den hohen Zielen, auch der G1 kann keine maximalen Pausenzeiten garantieren…

Hinweis:

Alle hier angesprochenen Algorithmen sind in der JVM von Oracle implementiert. Es gibt aber noch weitere namhafte JVMs, z.B. die IBM JVM. Deren Garbage Collection Algorithmen sind ähnlich, unterscheiden sich aber in den Tuningmöglichkeiten.