

Garbage Collectors

Rémi Forax

Plan

Histoire

Garbage Collection

- Compteur de références vs Tracing
- Allocation / Desallocation
- Compacter vs Copier

Garbage Collectors de Java

Les références faibles

Histoire

Bref histoire des *Garbage Collectors*

- 1970
 - LISP (John McCarty 1969)
 - Cheney algorithm (semispace)
- 1980
 - BASIC, PERL
- 2000
 - Sun Concurrent Mark And Sweep: CMS
- 2010
 - Garbage First: G1
 - Azul Continuous Concurrent Compacting Collector: C4

Langage

Langages qui utilisent un GC

- Languages typés dynamiquement
 - SmallTalk, PERL, Python, VB, PHP, JavaScript, etc
- Langages typés statiquement
 - Objective-C, Java, D, C#, Go, etc

Lexique

Concurrent

- Un GC concurrent est un GC qui s'exécute en même temps que l'application

Parallele

- Un GC parallele est un GC qui effectue une des tâches du GC en utilisant plusieurs threads

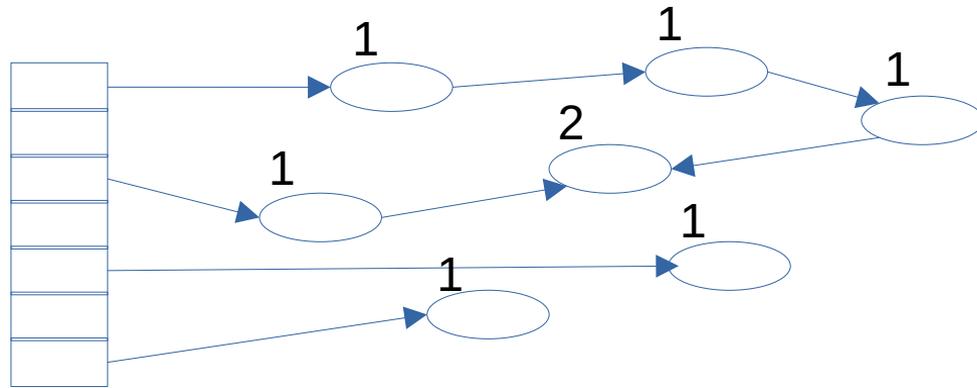
Incrémentale

- Un GC incrémentale est un GC qui effectue une partie de la tâche au lieu d'effectuer toute la tâche

Compteur de références

Compteurs de référence

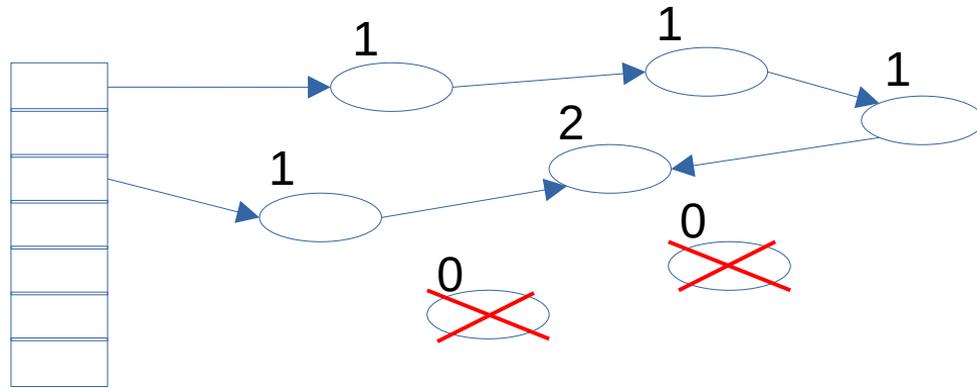
Chaque objet a un compteur de références



À chaque fois que l'on crée une référence sur un objet,
on incrémente le compteur

Compteurs de référence

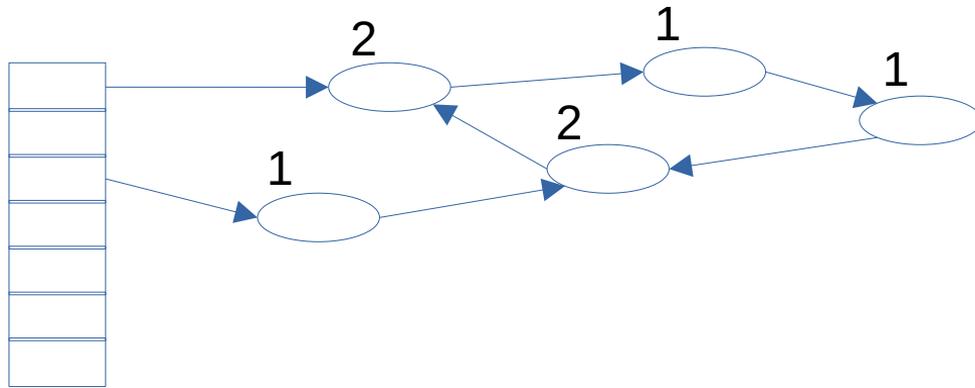
Chaque objet à un compteur de références



Si le compteur tombe à zéro, on supprime l'objet

Compteurs de référence

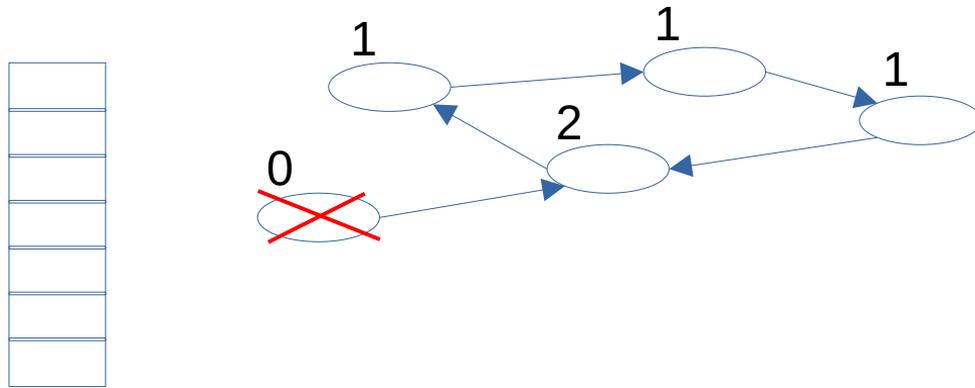
Chaque objet à un compteur de références



Problème avec les cycles !

Compteurs de référence

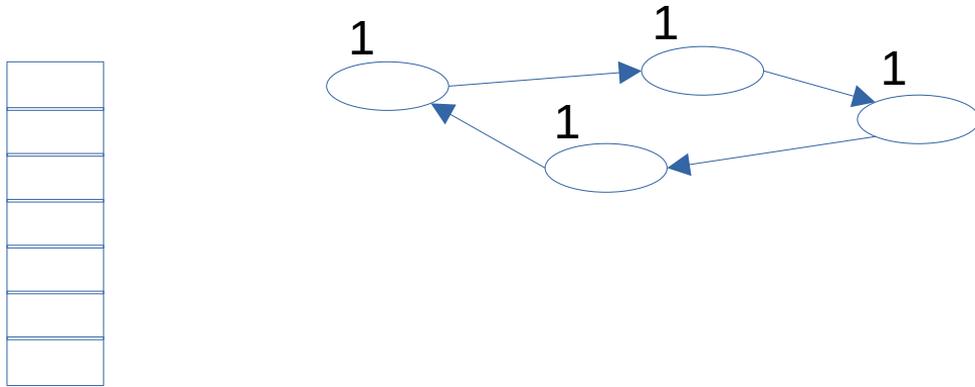
Chaque objet à un compteur de références



Problème avec les cycles !

Compteurs de référence

Chaque objet à un compteur de références



Oups ! Python détecte les cycles, Objective-C non !

Compteur de référence

Avantage: la libération est prédictible

- Cela revient à faire les free() automatiquement

Deux problèmes

- Les cycles
- Le multi-threads, il faut mettre à jour les compteurs de façon atomique
 - Les opérations atomiques coutent cher (aller/retour en RAM)

Tracing

Tracing *roots*

Ce sont les endroits où l'on va trouver les objets initiaux qui référencent les autres objets

- La pile de chaque thread
- Les globales (static en Java)
- Les globales de JNI/VM (pour les objets Java stockés dans le code C)

Tracing

On part des *roots*, on marque tous les objets que l'on rencontre récursivement

– 2 problèmes

- Comment on sait si on a une référence et pas un grand entier sur la pile ?
- Comment on fait pour que l'algo ne soit pas récursif ?

Precise Tracing

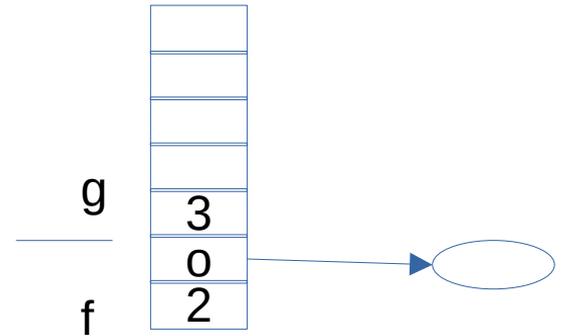
Le garbage collecteur ne se déclenche qu'à des endroits (*safepoints*) précis du programme

- Lors d'une allocation (new, etc)
- En début (V8) ou en fin de fonction (OpenJDK)
- Avant le goto d'une boucle dont on ne connaît pas le temps d'exécution

Le bytecode est typé donc on peut calculer par interprétation abstraite les positions des pointeurs sur la pile

Exemple

```
- void f(Object o) {  
    return 2 + g(o);  
}  
  
- void g(Object o) {  
    return 3;  
    // Le GC se déclenche ici !  
}
```



Conservative Tracing

Boehm GC

- Marche sur une pile C
 - Pas d'information sur le type des valeurs sur la pile
 - On sait que le tas est entre deux adresses
 - Toutes les valeurs entre les deux adresses sont des pointeurs
- => Cela veut dire que l'on garde des objets que l'on ne devrait pas !

Tracing

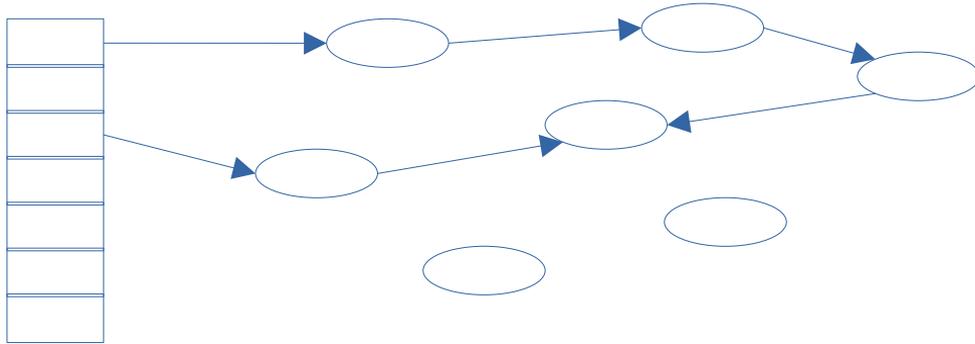
L'algorithme de "3-color marking" n'est pas récursif

- Au lieu d'avoir 2 couleurs white/black (marqué/pas marqué)
 - On a une couleur (grey) qui indique qu'il faut parcourir les fils

3-color Marking

3 ensembles White / Grey / Black

- White tous les objets à marquer
- Grey: objets dont les fils ne sont pas encore marqués
- Black: objets dont les fils sont marqués

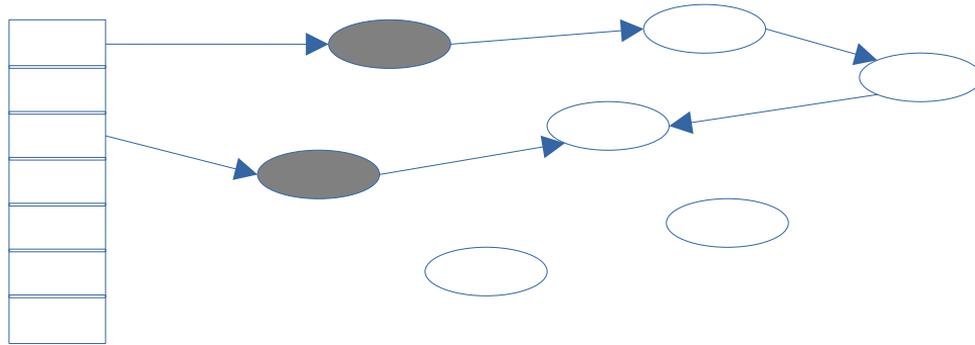


Au début tous les objets sont blancs

3-color Marking

Grey: objets dont les fils ne sont pas encore marqués

Black: objets dont les fils sont marqués

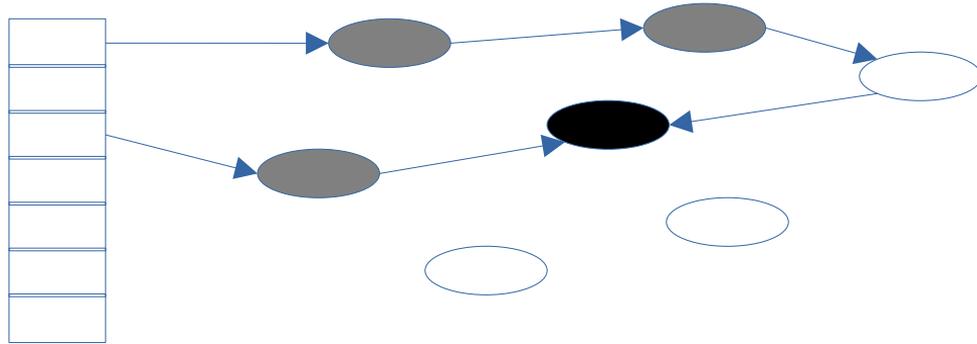


On marque les objets accessibles à partir de l'ensemble gris

3-color Marking

Grey: objets dont les fils ne sont pas encore marqués

Black: objets dont les fils sont marqués

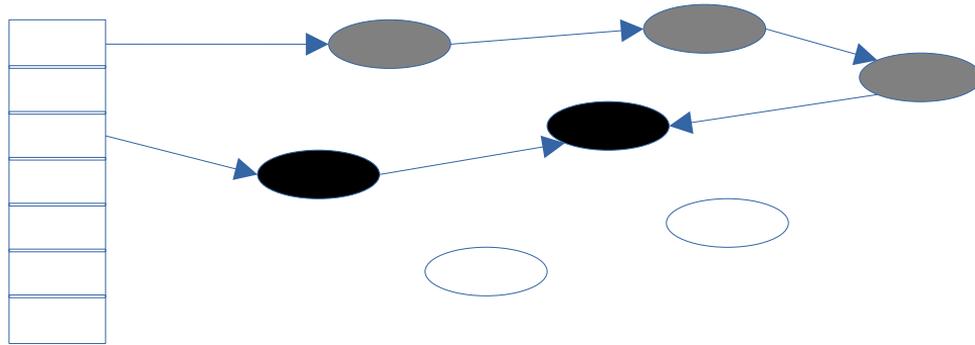


On marque les objets accessibles à partir de l'ensemble gris /
les objets dont les fils sont parcourus sont noirs

3-color Marking

Grey: objets dont les fils ne sont pas encore marqués

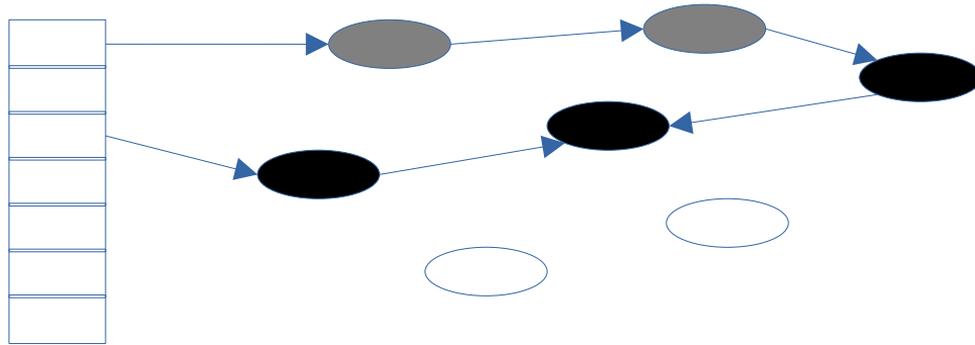
Black: objets dont les fils sont marqués



On marque les objets accessibles à partir de l'ensemble gris /
les objets dont les fils sont parcourus sont noirs

3-color Marking

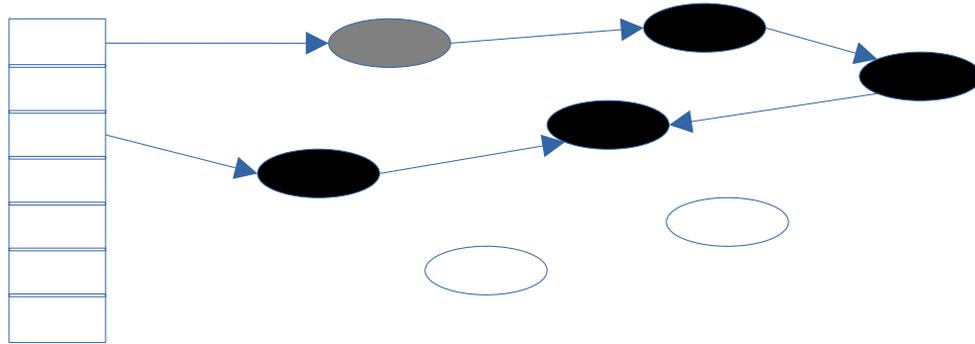
Black: objets dont les fils sont marqués



On marque les objets accessibles à partir de l'ensemble gris /
les objets dont les fils sont parcourus sont noirs

3-color Marking

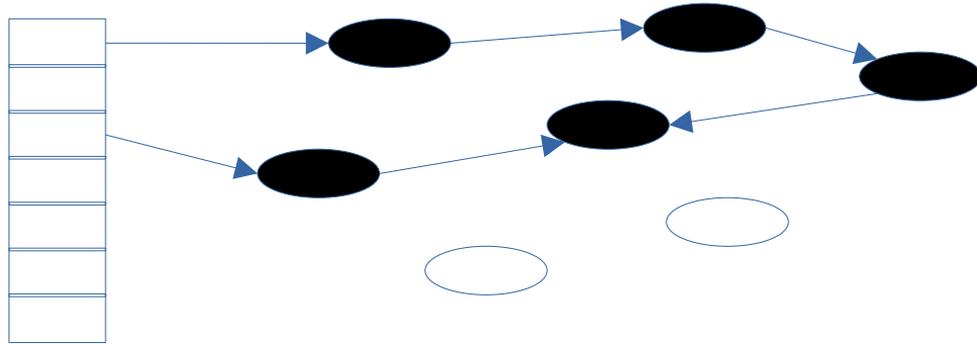
Black: objets dont les fils sont marqués



On marque les objets accessibles à partir de la l'ensemble gris /
les objets dont les fils sont parcourus sont noirs

Tricolor Marking

Black: objets dont les fils sont marqués

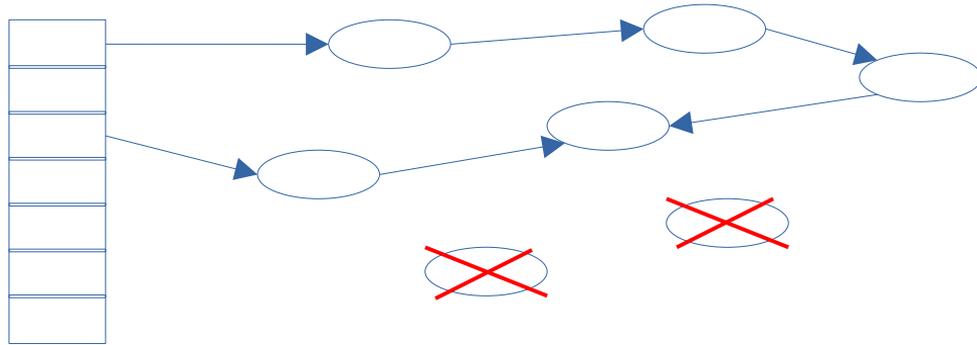


On marque les objets accessibles à partir de l'ensemble gris /
les objets dont les fils sont parcourus sont noirs

Tricolor Marking

Une fois que l'on a fini

- On "libère" les objets white
- black devient le nouveau white



Concurrent Tracing

Il existe une version concurrente du 3-color tracing

- Tous les nouveaux objets créés pendant que l'algo s'exécute sont noirs
- Si on mute un champ qui contient une référence, on passe l'objet en gris (cf plus tard)

Concurrent \Leftrightarrow s'exécute en même temps que le programme

Marquer les objets

Historiquement, on marque les objets en gardant 2 bits dans le header des objets

- Mais plein d'écritures en mémoire (lent !)

Les implantations incrémentales très récentes gardent dans des hashsets les adresses des objets marqués

Allocation / désallocation

Stratégie Sweep : Malloc / Free

Si on veut être compatible avec le C

On appelle malloc() lors de l'allocation

On appelle free() sur les objets pas marqués (white)

BoehmGC ou le GC de Python/Go utilise malloc/free

Mais

- Malloc est lent, il merge les morceaux de mémoire libérés par free()
- Problèmes : concurrence => locks et fragmentation en mémoire

Stratégie: Allocation rapide

Pointer Bump: on alloue les objets les uns derrière les autres

- Pas de liste chaînée de suivants comme malloc

Thread Local Buffer: pour le multi-thread, chaque thread à sa propre zone d'allocation

- Opérations atomiques que si une zone est pleine

Compacter vs Copier

Deux techniques

- Compacter : On bouge les objets vers le début du tas (LISP)
- Copier : on copie les objets vivants dans une nouvelle zone, l'ancienne zone est considérée comme vide

Cela demande de mettre à jour les pointeurs dans les globales, sur la pile, et dans les champs

GC Lisp-2 (1970)

Marking: On parcourt la pile et on marque tous les objets vivants récursivement

Planning: On parcourt le tas et calcul pour chaque objet vivant son nouvel emplacement (*forward address*). Il faut un espace dans l'entête de chaque objet (*Brooks pointer*)

Fixing: on parcourt la pile et on met à jour les pointeurs,
on parcourt le tas, on met à jour les champs

Moving: on parcourt le tas et copie les objets à leur nouvelle place

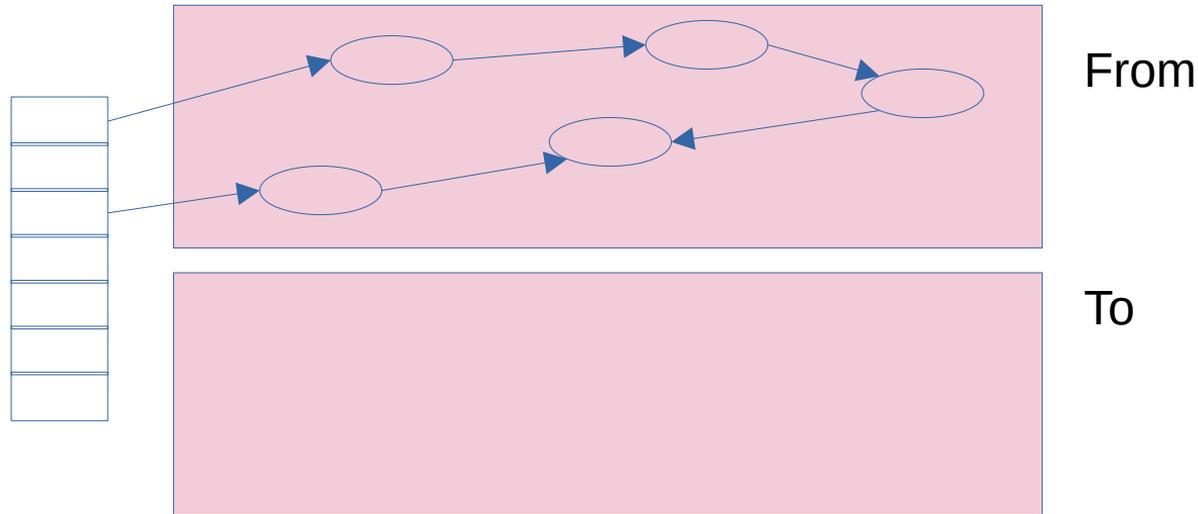
GC Semispace – Cheney (1970)

On sépare la pile en 2 zones (“from”, “to”)

- On alloue dans la zone “from”, une fois celle-ci pleine
- **Marking**: On marque les objets vivants
- **Copy**: Pour chaque référence sur la pile, on copie l’objets de “from” vers “to”. On écrit dans l’objet dans “from” la *forward address* vers “to”. Pour chaque champs, on fait la même chose s’il n’y a pas de *forward address*.
- On swap from et to.

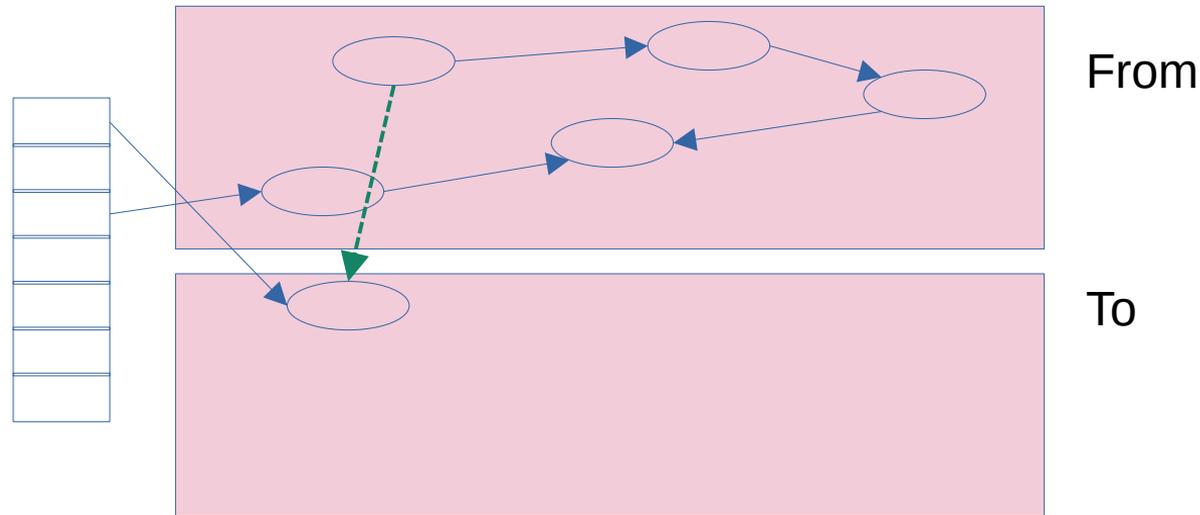
Algorithme de Cheney

On s'intéresse aux objets marqués



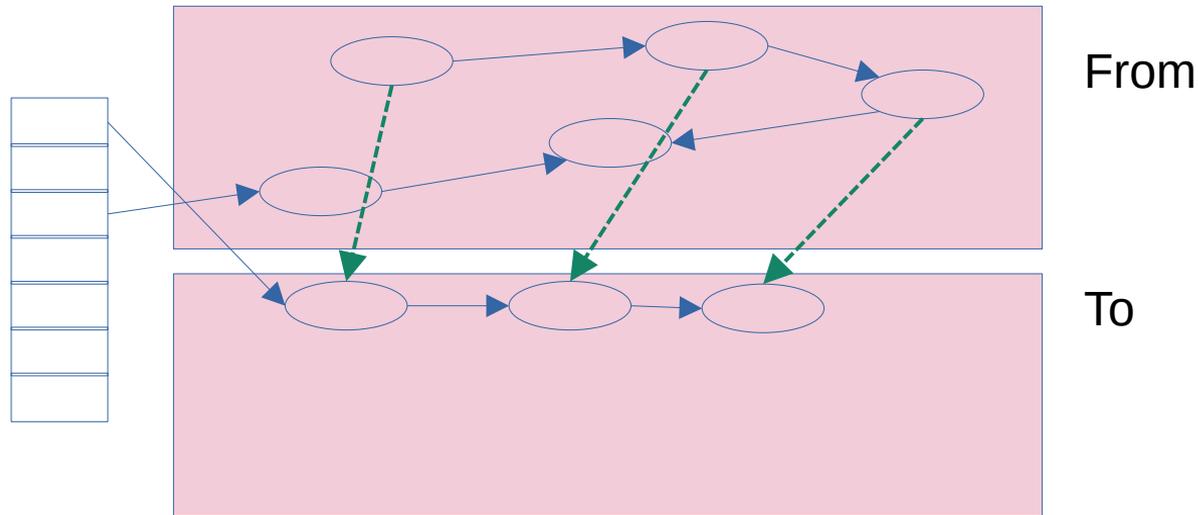
Algorithme de Cheney

On parcourt la pile, copie l'objet + forward address



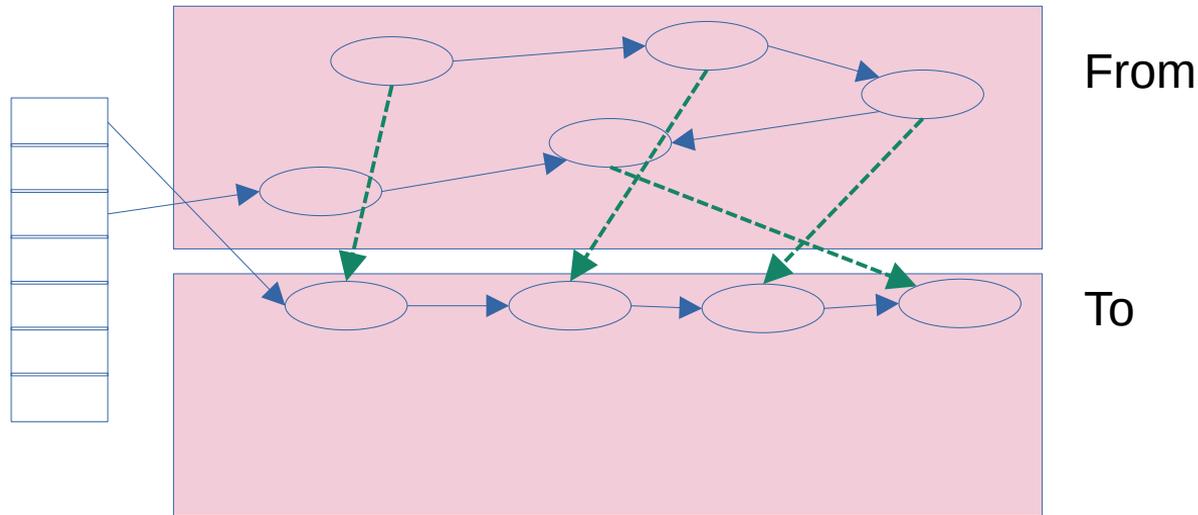
Algorithme de Cheney

On parcourt le champ, copie l'objet + forward address



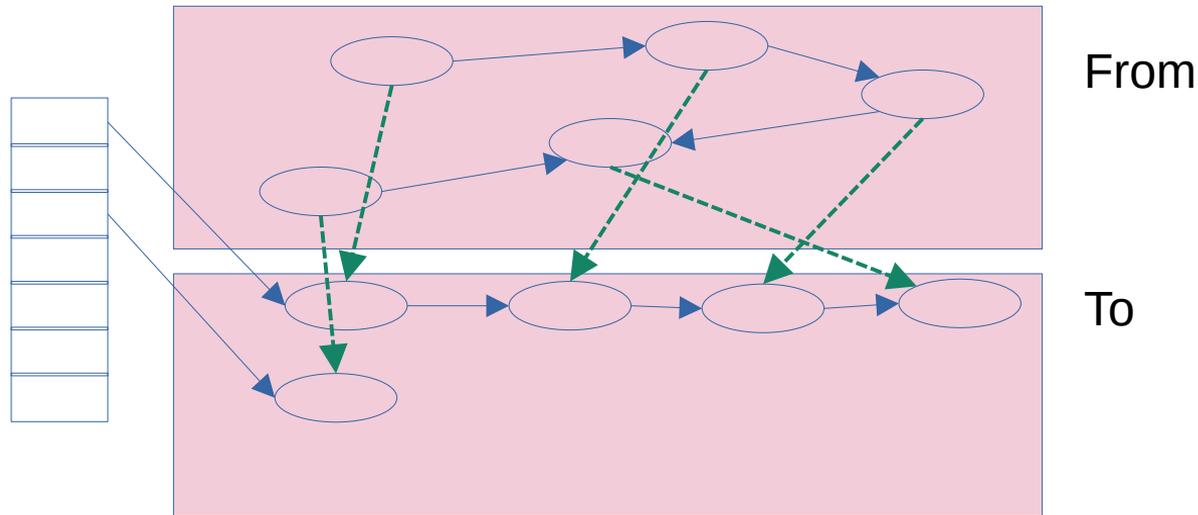
Algorithme de Cheney

On parcourt le champ, copie l'objet + forward address



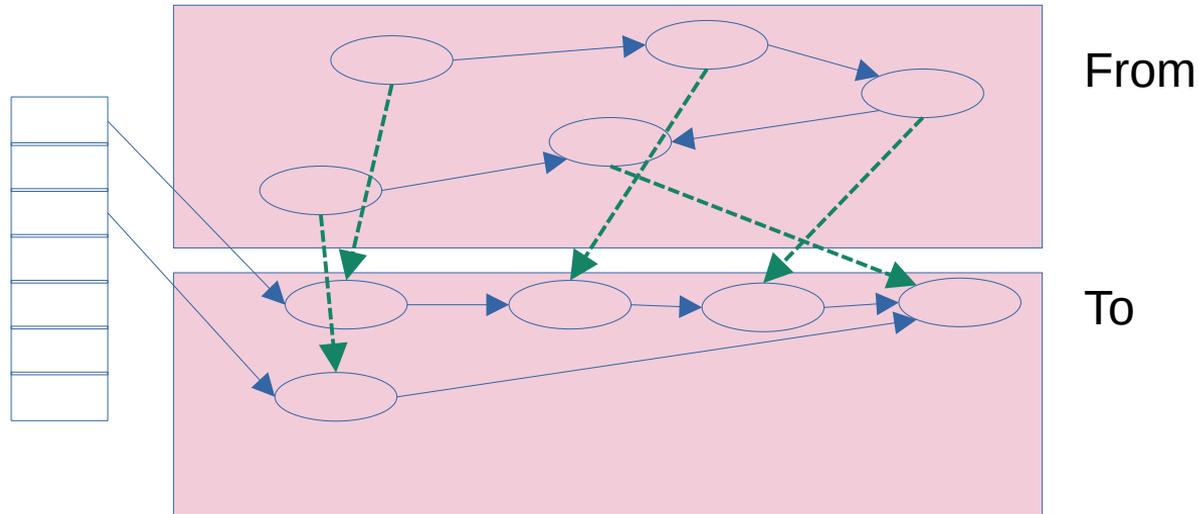
Algorithme de Cheney

On parcourt la pile, copie l'objet + forward address



Algorithme de Cheney

On parcourt le champ qui a déjà une forward address



Et, "to" devient le nouveau "from"

Compacter vs Copier

Les algorithmes historiques

- Compacter est lent
 - La complexité dépend de la taille du tas
 - Copier demande plus de mémoire que nécessaire
 - La complexité dépend uniquement des objets vivants
- => Découper l'espace en différentes zones,
on marque/compacte/copie pas tout !

Hypothèse Générationnelle

Hypothèse générationnelle

La plupart des objets meurt jeunes

- Donc on doit concentrer les efforts du GC dans les zones où l'on vient d'allouer des objets
 - Ou dans les zones où l'on vient de copier les objets qui viennent d'être allouer
- On attribut à chaque zone en entier de génération
 - Le GC privilégie les zones avec un entier petit

Incremental Marking

L'algo de marking nécessite de parcourir tous les objets

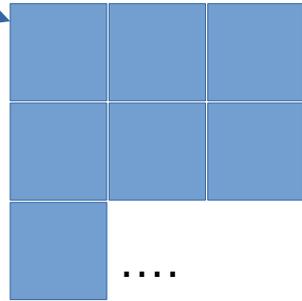
- Donc pas incrémentale

Mais si chaque zone maintient si elle possède des pointeurs sur une autre zone

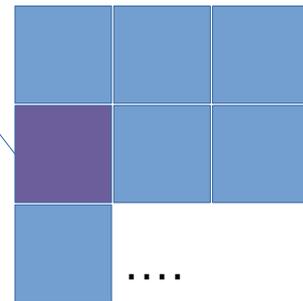
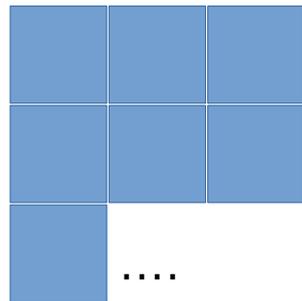
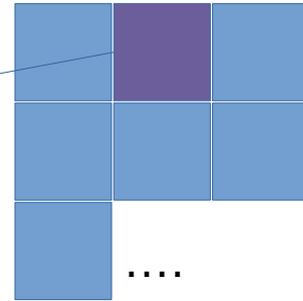
- On peut utiliser 1 bit mais trop gros grain
 - On utilise 1 bits pour une partie de la zone (une *card*) ~ 256 bytes
 - La structure de données est appelée CardMark

Card Marking

une *card*



z1: 0100...
z2: 0000...
z3: 0001...



Une fois que la zone0 est évacuée, on reset les bits

Incremental Card Marking

Comment mettre à jour la structure de CardMark ?

- À chaque fois que l'on fait une écriture
 $a.b = c$
- Dans la zone qui contient 'c', la *card* qui contient 'c' indique quelle est référencée à partir de la zone de 'a'
 - On appelle le code qui fait la mise à jour une *GC WriteBarrier* ou barrière en écriture pour le GC

Incremental Card Marking (2)

Marking incrémentale des zones jeunes

On ne marque pas les objets en dehors des zones jeunes

On marque à partir des objets racines (roots)

Et on parcourt les objets contenus dans les *cards* qui pointent vers les zones

Collection Mineur / Majeur

Collection mineur

- on récupère uniquement les objets qui sont jeunes

Collection majeur

- on récupère les objets des zones pleines

Collection full (*allocation failure* = mode panic)

- on récupère tous les objets

Evacuation/Copie concorrente

Copie concurrente

On peut faire la copie des objets en même temps que le programme s'exécute

- Il faut planifier où les objets doivent être copiés (calculer la *forward address*)
- Si le GC n'a pas encore déplacé un objet, le programme doit le faire :)
 - Il faut une *GC Read Barrier* (une barrière en lecture pour le GC)
 - Si le GC est en phase de copie des objets, avec a.b il faut regarder si l'objet "a" appartient à une zone copiée, faire la copie vers la *forward address* puis modifier la variable 'a'.

Les *garbages collectors* de Java

Garbage Collectors de l'OpenJDK

L'OpenJDK possède 5 *garbage collectors* différents

- Peu de mémoire / core: Serial, Parallel, G1
- Beaucoup de mémoire / core: G1, Shenandoah, ZGC

L'allocation utilise des TLB (NUMA aware) + *bump pointer*. (Presque) tous les GCs sont générationnels

Serial et Parallel

Serial et Parallel sont GC generationnels, stop the world

- Minor: mark & copy, Major: mark & compact
- Serial GC utilise 1 thread, memoire < 100 M
 - Ligne de commande, serverless lambda, ATM, IOT
- Parallel GC utilise toutes les threads, memoire < 128 G
 - GC le plus rapide (throughput) mais avec des temps de pause (latence) qui peuvent être horribles
 - Il faut bien connaitre la *workload* de l'application
 - Full GC en parallèle à besoin d'un flag !

Garbage First (aka G1)

est un GC generationnel, c'est le GC par défaut

- Minor/Major: mark & copy
- memoire < 128 G
- On indique le temps de pause maximal (> 100ms)
- Le marking est concurrent, la copie incrémental
 - Rapide mais peut louper des échéances

Shenandoah

est un GC développé principalement par RedHat

- Minor/Major: mark & copy
- Préfère la latence au débit (throughput)
 - Mémoire < 1T
 - Temps de pause < 1~3 ms
- La version générationnelle est en bêta
- Le marking et la copie sont concurrents
 - Le parcours des piles est concurrent aussi

ZGC

est un GC développé principalement par Oracle

- Minor, Major : mark & copy
- Préfère la latence au débit (throughput)
 - Mémoire > 32G (pointeur sur 64 bits), < 16T
 - Temps de pause < 1 ms
- La version générationnelle est en production depuis 21
 - Il faut un flag supplémentaire
- Le marking et la copie sont concurrents
 - Le parcours des piles est concurrent aussi

Les Références faibles (Weak Reference)

Références Faibles

Une référence faible est une référence qui n'est pas compté/parcouru par le GC

Le package `java.lang.ref` contient plusieurs classes (qui hérite de `Reference`) et qui permet de définir différentes références faibles

Comment créer une référence ?

Les références faibles possèdent un constructeur qui prend en paramètre une référence (forte)

```
var myObject = new MyObject();  
var ref = new WeakRef<>(myObject);  
ref.refersTo(myObject); // true  
  
myObject = null;
```

– Plus tard

- ref.get(); // renvoie null si myObject a été GC
- ref.refersTo(null); // true si myObject a été GC

ReferenceQueue

FIFO qui contient les références des objets qui ont été GC

- **var** myObject = **new** MyObject();
var queue = **new** ReferenceQueue<MyObject>();
var ref = **new** WeakRef<>(myObject, queue);

myObject = **null**;
- Lorsque myObject est GC
 - queue.poll() // renvoie ref ou null
 - queue.remove() // block et renvoie ref
 - queue.remove(timeout); // block et renvoie ref, ou null si timeout

Types de références faibles

Il existe 3 types de références faibles

- PhantomReference & WeakReference
 - Le référent devient null lorsqu'il est GC
- SoftReference
 - Si le GC a besoin de mémoire, la référence devient faible
 - Le référent devient null lorsqu'il est GC

Finalization (Legacy)

Finalization (deprecated)

java.lang.Object possède une méthode

- protected void finalize()

Appelée par un thread (exécuté par le GC)
lorsqu'une instance devrait mourir

- Rend l'objet pas thread safe (ahhh)
- Peut faire revivre l'objet une fois

Finalization (deprecated)

Historiquement, utiliser pour appeler close()

```
class MyObject implements java.io.Closeable {  
    private final InputStream input; // doit être final ou volatile !  
    ...  
  
    protected void finalize() {  
        input.close();  
    }  
  
    public void close() {  
        input.close();  
    }  
}
```

Finalization (deprecated)

En réalité

- finalize() est appelée que quand on manque de mémoire
 - close() est jamais fait si on manque pas de mémoire (ahh)
- Pas efficace, on fait revivre l'objet pour pouvoir appeler close()
 - Le GC doit lancer une thread (ou plusieurs !)

Finalization (removal)

Les classes du JDK implémentent plus finalize()

- Marche encore pour les autres classes

Flag pour éviter la finalization (--finalization=disabled)

Bientôt

- Flag pour que la finalization marche (--finalization=enabled)
- Puis la finalization marchera plus !
 - finalize() va être supprimé de java.lang.Object

PhantomReference vs WeakReference

Historiquement, le référent

- d'une phantom ref devenait null avant la finalization
- d'une weak ref devenait null après la finalization

Maintenant, plus de différence !