GCアルゴリズム詳細解説

日本語に資料がすくないGCアルゴリズムについて詳細に解説します

メニュー

トップページ
最近更新したページ
ページ一覧
タグ一覧

Wiki内検索

検索

最近更新したページ

2008-06-25
GC
2008-01-10
GCChild
トップページ
2007-03-02
MenuBar1
MenuBar2

最新コメント

メニューバーA

ここは自由に編集できるエリアです。

タグ

GC

このWikiが目指す所

ずばり
「日本語によるGCアルゴリズムの詳細解説」
です。

GCについて勉強したい！
と思ったのは今から半年前くらいです。
それから、ずーっとWeb上を探し回ったんですが、
GCというマニアックな領域のせいか、日本語による詳細な解説が
乗っているWebページがありませんでした。
また、これによって日本人のGCへの関心の低さに驚いたんです。
(Webを使ってる人はみんな恩恵を貰ってるのにね。)

「誰も書かないんだったら、私が書くか。」ということで
一念発起してこのWikiを書くに至っております。
このWikiには最新の技法から自分のまとめたものをちまちま書いていくつもりです。
また、他の人の編集も可能にしていますので、よかったら更新してあげてください。

GCとは？

どこからも参照されなくなった(つまり不要)メモリ領域を自動で掃除してくれる便利なやつ。
本当はmalloc/freeと手動でメモリ管理しなければならない所(C言語など)を
あるタイミング(使う側には指定できない事が多い)でいらない領域をfreeしてくれる。
本Wikiは概要についてはあんまり説明しないのでガベージコレクション(wikipedia)を参照

GCは二つの略語をさします。
(garbage collection) ガベージコレクション -- プログラム上で不要となったメモリを回収する動作。
(garbage collector) ガベージコレクタ -- 上記を実現するメカニズム。

本Wikiでは、統括して「GC」と呼ぶ事にします。

GCを学ぶ前に知っておく事

専門用語を知らないと中々辛いので、まとめています。
用語集

実行時メモリ構造

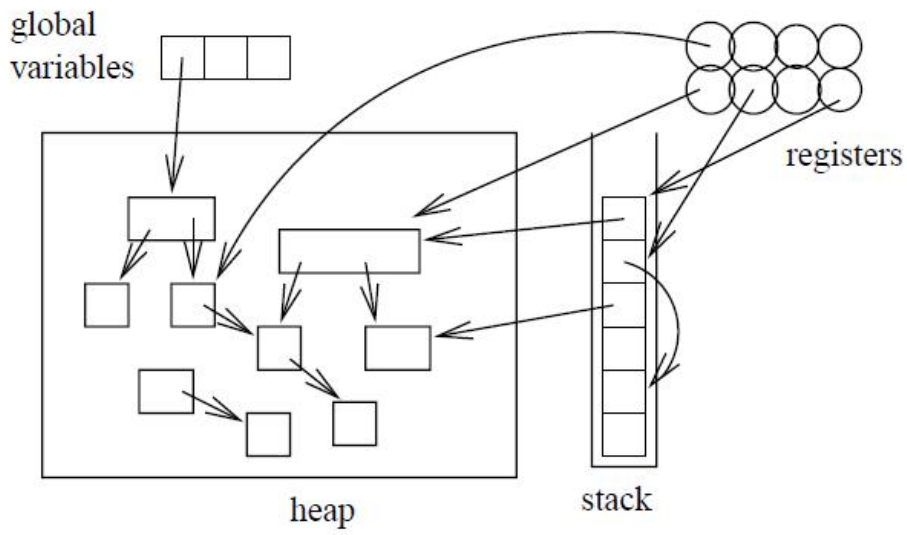


Figure 2: プログラム動作中のメモリ構造の例

これからの物事を理解するために、プログラムが動いている時のメモリ領域の構造を図 に示す。
大まかには、Java, ML, C 言語も含めほとんどの言語は似たような構造になる。
ヒープ内の小さい四角はオブジェクトを表す。
また、あるオブジェクトA の中に含まれるポインタが、別のオブジェクトB を指していることがある。
(record の中にrecord がある場合や、オブジェクトのフィールドに別のオブジェクトを含む場合など)
図中では参照関係を、A からB への矢印で表す。

スタックには、局所変数や関数呼び出しの履歴などが格納される。(話をごく簡単にすると) 関数呼び出しが起こると伸びて、return が起こると縮む。

大域変数を格納する領域が用意される。定数が格納される領域も独立に用意される場合が多い。

レジスタは、関数の引数、計算の途中結果など様々な値をとる。オブジェクトを指すポインタの可能性もある。

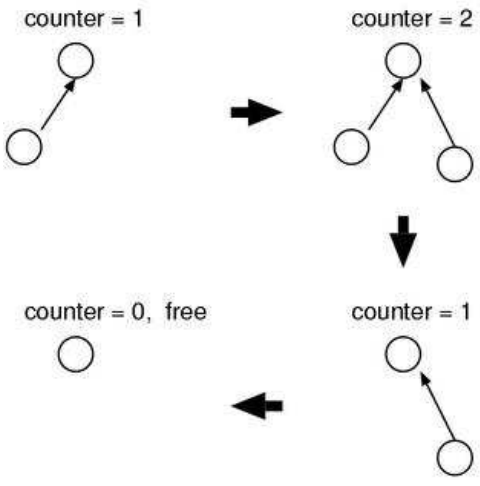
ヒープは自由な順番でメモリ領域を確保/解放できる領域である。
オブジェクト作成(C ではmalloc)を行なうと、ヒープの中に必要な大きさの領域が確保される。

図には示していないが、プログラムのコード領域もどこかにある。

基本アルゴリズム編

Reference Counter

和訳:参照カウント(GC)



アルゴリズム

・自身のオブジェクトが参照されている数のカウンタをもっている。
・参照される度に+1、参照が切れる度に-1される。
・カウンタが0になった時点でどこからも参照されない(つまり不要)となり解放される。

利点

1.負荷が分散される
2.停止時間が短い
3.ルートを必要としない

もっとも優れている点はカウンタを-1する際に一緒に解放処理(利点1)ができると言うこと。
後のアルゴリズムでは全オブジェクトを走査する必要があるが、この方法だと
その場(参照が切れたタイミング)での解放処理が可能。これによって停止時間が短い。(利点2)

欠点

1.参照がサイクルしてる場合(相互に参照している場合)の解放ができない。(致命的な問題)
　→解決方法1:何かのタイミングでMark&Sweepなどの相互参照を問題にしないGCアルゴリズムを使用する。
　→解決方法2:ゴミ参照(相互参照の場合)を検知する為にヒープ領域を走査する。
2.カウンタ処理が分散されてうざい。
3.カウンタ処理が結構重い。

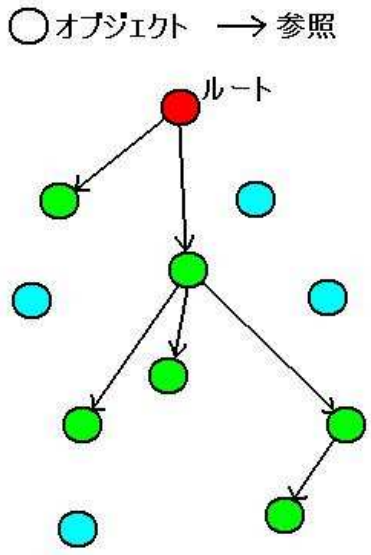
GCでは処理の早さがかなり重視される。(言語処理系の処理時間を結構左右するため)
カウンタ処理って軽いじゃん。という事でさえ神経質にならなくてはならない。
欠点2に関してはしょうがないが、欠点3についてはWriteBarrier(用語集)なども有るため
重いのでこのアルゴリズムは避けよう、という事には一概にはならない。

実コードを見たい場合

・Perl5処理系
(欠点1を補うためにスレッド終了時にMark&Sweepを実施している)

Mark&Sweep

和訳:印づけ、掃除



アルゴリズム

ルート(用語集)を全走査し、参照されているオブジェクトを見つけ出す。
見つけたオブジェクトに対して1bitマークをし、続けてオブジェクト内で参照しているオブジェクト(枝のオブジェクト)
に対して、印をつけていく。
ルートの走査が終わると、印があるもの(図:緑)とないもの(図:水色)に別れ、
印がないものはゴミ(参照なし)と判断され解放される。

この印付けをMark、ゴミの解放をSweepという。
保守的GCによく使われる。

オブジェクトの格納領域(ヒープ領域)を作成する際に、新規にallocateする処理の為に、
フリーリストで利用可能な領域を繋ぐ(空のオブジェクトを繋ぐと考えるといいかも)

そして、ユーザーがオブジェクトを作成する際にフリーリストから一つオブジェクトを貸し出す。
Sweepフェイズではヒープ領域を全走査し、Markされていないものを解放後、フリーリストにつなぎ直す。

利点

1. GCの実装以外の場所ではGCのことを(あまり)考えなくていい
2. サイクルも解放できる(サイクルについてはリファレンスカウントの項を参照)

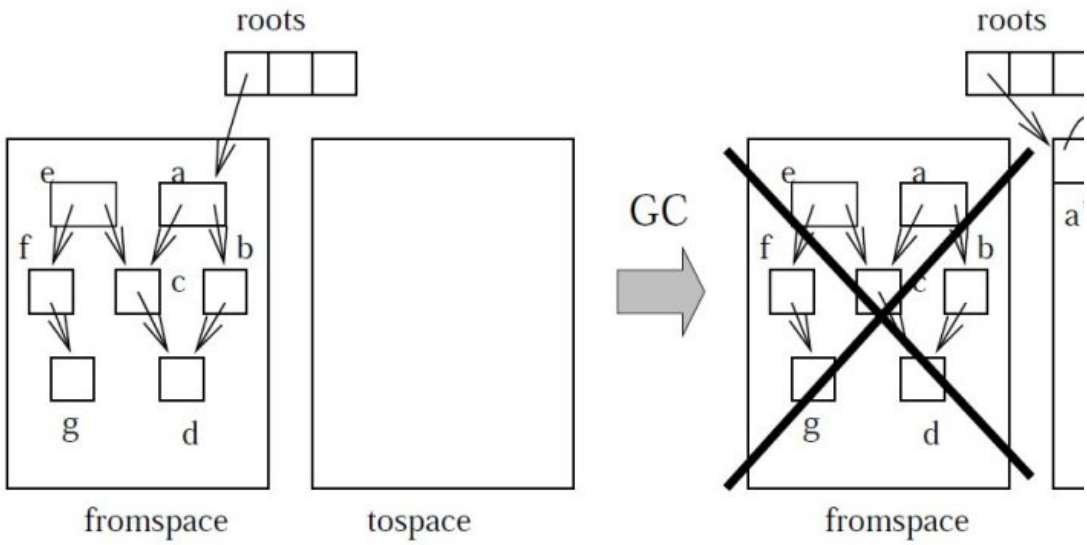
欠点

1. スイープのため最低一度はオブジェクトを全部なめる必要がある
2. GCの負荷が一点に集中する(停止時間が長い)

実コードを見たい場合

・Ruby処理系
→読みやすいのでおすすめ
・BoehmGC

Copying



アルゴリズム

Copying GC アルゴリズムは、生きたオブジェクトをすきまをつめながら移動するというものである。
これにより、mark sweep やreference counting で問題となるfragmentation が全く起らないという利点がある。
また、多くのSchemeやML処理系で採用されている。

ここでは最も単純な、ヒープを2 等分する方式を紹介する。
この方式では、一度に使えるヒープは2等分のうち片方のみである。使用中の片方が埋まった時点でGC を起動する。
そして到達可能なオブジェクトたちを、そっくりもう片方のヒープに隙間をつめながらコピーする。
コピー元のヒープをfrom-space、コピー先のヒープをto-space と呼ぶ。
コピーが終った時点ではすでにfrom-space には生きたオブジェクトの残骸とゴミしか残っていないので用無しとなる。
GC 終了後、ユーザプログラムはto-space を用いて動作を続ける。このように、GC の度に2 つのヒープの役割を交替しながらシステムは動作する。

この方式を実装しようとするとき、いくつか注意が必要である。
例えばコピー後のグラフは必ずto-space内で完結し、from-space にポインタがのびていてはいけない。
また、複数箇所から同オブジェクトが参照されている場合にも注意する。
図ではb とc がd を参照しているので、to space においてもb'とc'は同オブジェクトを参照しなければならない。
アルゴリズムを3 色モデルを用いて考えると以下のようになる。

白・・・まだコピーされていない(from-space にしかない) オブジェクト
灰色・・・それ自身はコピーされたが、from-space のオブジェクト(白オブジェクトや他のオブジェクトのコピー前) を指している可能性がある。
黒・・・それ自身はコピーされたし、from-space を指している可能性もない。

上のような条件を満たしていれば、深さ優先でも幅優先でも良い。ここでは、copying GC の代名詞に
なっている、Cheney の幅優先アルゴリズムを紹介する。これは再帰呼び出しもマークスタックのような領域も使わない。
ヒープ以外のメモリ使用量はO(1)!
GC 開始時に、ルートオブジェクトから直接参照されるオブジェクト(図ではa のみ) をto space にコピーする。この時点でa は灰色と考えられる。
このアルゴリズムは再帰探索のために2 つのポインタを用いる。

1. scanned・・・初期値はto-space の先頭。scanned より手前は全て黒オブジェクトである。
2. unscanned・・・初期値は、図の場合(to-space の先頭) +sizeof(a)。scanned とunscanned の間は全て灰色オブジェクトであり、unscanned 以降は空き領域。

GC は以下の処理を繰り返す。scanned が指す先頭オブジェクトo を取り出し、o の子オブジェクト達のうち、
未コピーであるものをアドレスunscanned にコピーする(o の子オブジェクト達を灰色にする)。その度にunscanned を進める。
同時に、o 中のポインタがコピー先をさすように書き換え、scanned を進める(o を黒にする)。
scanned がunscanned に追いついたら、GC を終了する。

さて、同じオブジェクトへの複数参照に対応するためには、from-space 中のd を見ただけで、
「これはすでにコピー済みであり、コピー先はd' である」
ということが分かる必要がある。
そのために、灰色化の時点でd にforwarding tag (コピー済みであることを表す)、
forwarding pointer(コピー先を教えてくれる)を書き込んでおく。

利点

1. メモリ回収と同時にcompactionできる。
2. 参照関係を考慮したコンパクションにより、キャッシュとの相性がよい(=プログラムの実行速度が上がる)。
3. Mark&Sweepなどと比べてallocate処理が早い(FreeListなど使わなくていい為)
4. また解放処理も早いよ。

欠点

1. オブジェクトの移動により、ポインタの書き換えが起こり、保守的GCは作りにくい。
2. ヒープ領域を分割するため、メモリ領域を何倍も取る。

実コードを見たい場合

Scehme処理系の何か

応用アルゴリズム編

ここでは基本アルゴリズムを応用し、欠点を解決したアルゴリズムの紹介をする。

IncrementalGC

ちょっとづつGC

アルゴリズム

基本アルゴリズムのCopyingとMark&Sweepでは一気にオブジェクトの走査、解放を行うため、
GCによる停止時間が非常に長かった。
この欠点を補うのがこのアルゴリズムである。

Incremental GC はGC の探索処理を細切れにして、ユーザプログラムと交互に動かす。
これにより一回あたりの停止時間を短くするアプローチであり、リアルタイム性の必要なプログラムに適している。
交互で動かす最もメジャーな方法は、allocate 処理を行なうときに、こっそり少しづつGCを進めるというものである。
以下では、mark sweep GC をincremental 化する場合について述べる。
Incremental GC を実装するには、いくつか問題点をクリアしなければならない。

GC 処理のスケジューリング。これまでのように、ヒープが満杯になったらGC を始める・・のでは間に合わない。
GC の再帰探索が一通り終るより先に、メモリが尽きるという状況は極力避けなければならない。
ちなみに、GC の仕事を少し行なったからといって、メモリをすぐに解放してくれるわけではない(=省メモリ性は相変わらず悪い)
ことに注意

writebarrierが必要になる。

なぜ、writebarrierが必要になるかというと、GCの細切れで動かす本アルゴリズムはMarkの途中で
ユーザープログラムの処理へ戻ることになる。
その際に、Mark済みのオブジェクトへの新規参照が追加され、またその新規参照されたオブジェクトが

Mark済みオブジェクトからしか参照されていなかった場合、Sweepフェイズでそのオブジェクトは Markが付いていないのでゴミと判断され解放されてしまう。
それを防ぐ為にwritebarrierを用いて、変更があったオブジェクトに印をつけ、もう一度Markし直す処理が必要になる。

基本的には、incremental GC は停止時間を短くするのが目的であって、GC の仕事の量を減らそうとするものではない。
このため、非incremental GC と比べてトータル実行時間は短くならない。
逆に仕事の切替えオーバヘッドやwrite barrier の分、遅くなるだろう。
Incremental を使うべきかどうかはプログラムの性質に応じて決める必要がある。

最近のスクリプト言語にはこのアルゴリズムが使用されていることが多い。

利点

1. 停止時間が短い

欠点

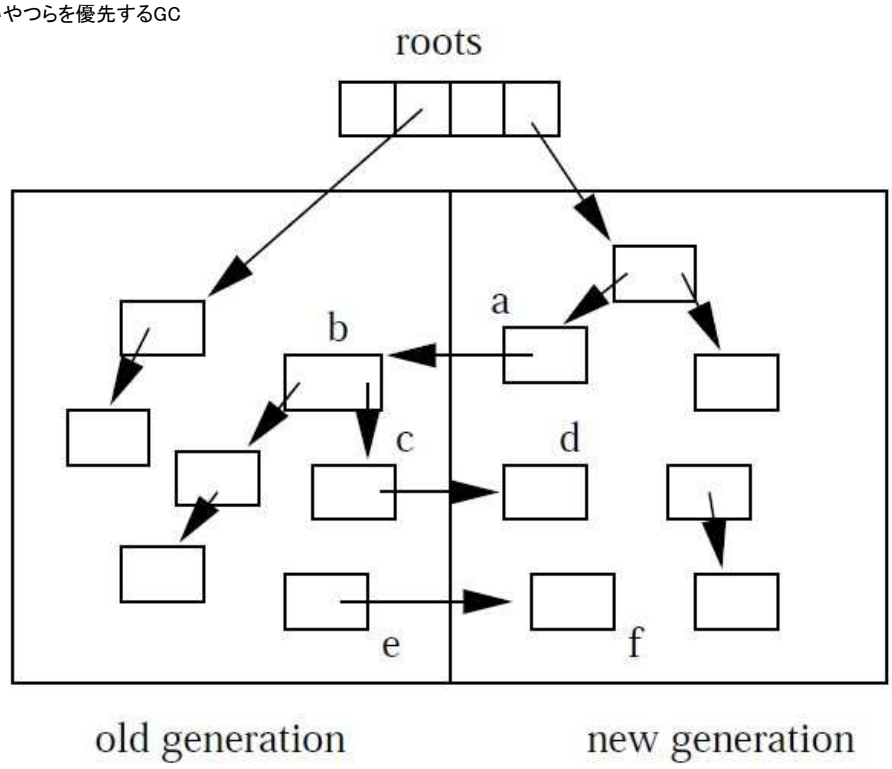
1. スケジューリングが結構大変
2. writebarrierの分、ミュテータに一律負荷が増加する

実コードを見たい場合

・Io処理系
・Lua処理系
・BoehmGC

世代別GC

若いやつらを優先するGC



アルゴリズム

オブジェクトの寿命について、多くのプログラムで観測されるある傾向が知られている。
それは「古くから生き残っているオブジェクトはそのまま生き残りやすく、新しく確保されたオブジェクトほどすぐにゴミになりやすい」(*)
という傾向である。

GC の実行効率は生き残るオブジェクトが少ないほどよいのだから、
古いオブジェクトはほうっておいて(無条件に生き残るとみなして) 新しいオブジェクトを重点的に探索することによって利益が得られそうである。
Generational GC (世代型GC) は、新しめのオブジェクトのためのヒープと、古めのオブジェクトのためのヒープを用いる(上図)。

Copying GC を基にするのであれば、図には示されていないが、新世代のヒープと旧世代のヒープのそれぞれがfrom/to に2 等分される。
オブジェクトはまず新世代ヒープに確保される。新世代ヒープが一杯になったら、ヒープ全体のGCを行なう代わりに、新世代ヒープ

だけを対象としたGC(minor collection) を行なう。
Minor collection を繰り返した結果、ある程度以上長寿命なオブジェクトが見つかったなら
(例えばGC がn 回起こる間生き残ったら)、そのオブジェクトは旧世代ヒープに移される(殿堂入り、またはpromote と呼ぶ)。
さて、minor collection においては旧世代ヒープ中のオブジェクトをスキャンする必要がない(旧世代オブジェクトは無条件に「生き」とみなされる)。
図ではオブジェクトb 以降の探索は行なわない。このため、minor collection はヒープ全体を探索するよりも速く終了する。

入れ替わりの激しい新世代ヒープと対照的に、旧世代ヒープは殿堂入りオブジェクトによってゆっくりと満たされていく。
やがて旧世代ヒープさえ一杯になったら、そこではじめて、ヒープ全体を探索するGC(major collection) を試みる。

Generational GC を実装する際には、以下のような落し穴に気をつける必要がある。
Minor collectionを行なう際に、旧→新のポインタに注意が必要である。
たとえばc → d のポインタに気付かないと、到達可能なオブジェクトd を間違って解放してしまう。
上図の場合、GC は新世代へのポインタを持っているcとe を全て覚えておき、
minor collection はそれらもルートと見なす。この目的のために(incremental GCの節で説明したような)
write barrier を用いることによって、普段から旧→新のポインタができたかどうかを全て覚えておかねばならない。

Generational GC におけるwrite barrierには、remembered list、remembered set、card marking、page markingがある。

remembered list

旧→新の参照が発生した場合に、参照先の世代のremembered list が参照元のオブジェクトを(配列などの形で)記憶しておく。GCの際にはこのremembered list をルートとしてスキャンすることで旧→新のポインタを発見できる。

remembered set

remembered list と似ているが、オブジェクトが登録されていることをオブジェクト自身が記憶している点が異なる。このため、オブジェクト内部に1ビット余計に取る必要がある。これにより、同じオブジェクトが2回以上登録されるのを防ぐことができる。

card marking

ヒープを2のk乗サイズに分割し、それぞれをカードとする。各オブジェクトは1つ以上のカードに属している(複数のカードにまたがってもよい)。カード内のオブジェクトが書き換えられたかどうかを記録する配列を用意する。要素数はkである。旧→新の参照が発生したとき、参照元のオブジェクトが属するカードがマークされる。GC実行時にはこの配列をスキャンすればよい。

page marking

card marking のカードのサイズをページと同サイズにしたものである。ページのサイズはOSにより異なるが、通常は4KBである。

利点

1. Minor collection の実行時間はヒープ全体のGC に比べ短いので、プログラムの停止時間を改善できる。
2. 上述の(*) という傾向が成り立つようなプログラムにおいて、トータル実行時間を改善できる。

欠点

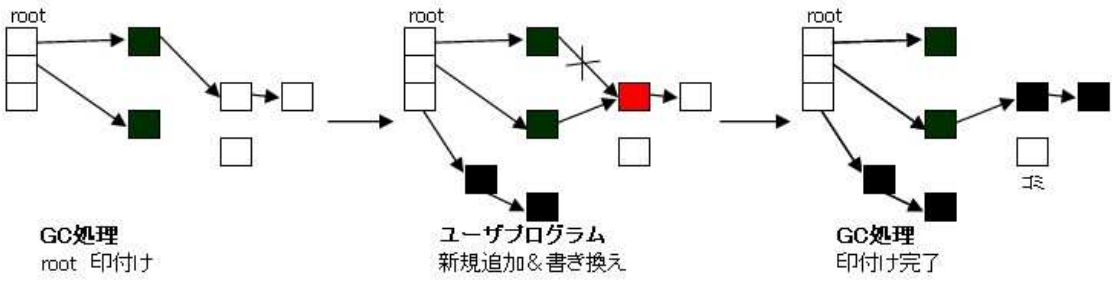
1. major collection が起こってしまえば長く停止してしまう。
2. 上述(*) の逆の傾向にある(古いオブジェクトの方が先に死にやすい) プログラムでは逆効果となりうる。

実コードを見たい場合

JVM処理系(複雑なアルゴリズムの為、詳細は後述する)
Smalltalk処理系の何か

スナップショット型GC(湯浅式GC)

開始時の写真を元にゴミ掃除



アルゴリズム

このアルゴリズムはMark&Sweepを基本に考えられる。
GC開始時にルートから参照できるセルに印をつける。

実はここまではIncrementalGCの主な実装と同じで、
ミュテータのセル操作によるものは一律印付けしてしまう。(上図参照)
想定されるは以下の2ケース

1. マーク中に新規割り当てされたセル
2. マーク中に書き換えられた黒→白のセル

2に関しては書き換えられたセルの枝にあるセルも全て印付けを行う。
1に関しては新規割り当てのセルの為、枝のセルは存在し得ない。

この単純な操作により、GC開始時にゴミであったセルが削除される。
GCの開始時の写真を撮って、その時のゴミを回収する事から
「スナップショット型」「スナップショット方式」と呼ばれる。

このアルゴリズムはGCのIncrementalGC、ParallelGC、ConcurrentGCを行う為のものであり、
現在のGCの並列化、並行化において一番性能のいいアルゴリズムと思われる。

発案者はCommonLisp入門などの著者　湯淺太一教授

利点

1. 単純なアルゴリズムの為、実装が簡単。
2. 漸次化、並列化、並行化を可能にする。

欠点

1. GC開始時のゴミしか回収しない為、ゴミの取りこぼしが多くなり、飢餓状態に陥りやすい。
2. ルートマークだけは処理分割(並列化など)できない。

2はスタックを処理分割する「リターンバリア」という手法が検討されている。

実コードを見たい場合

・もしかしたらJVM

LazySweep

遅延Sweep

アルゴリズム

このアルゴリズムはMark&Sweepを基本に考えられる。
前述しているIncrementalGCのSweep部分のみを考えもらうと分かりやすい。
最初GCではMark処理のみを行い、Sweep処理はObjectをAllocateする際にすこしずつ行う。
そして、すべてSweepし終わったら、また同じように繰り返す。

Mark処理を漸次化する場合、ミュテータの書き換えに対応するため、WriteBarrierなどの対応が必要になる。
が、Sweep処理のみなら漸次的に処理することが可能。(ゴミははっきりと分かるため)

利点

1. Sweepにかかる停止時間を分散させることができる。
2. つまり最大停止時間が短くなり、よりリアルタイムに処理ができる。

欠点

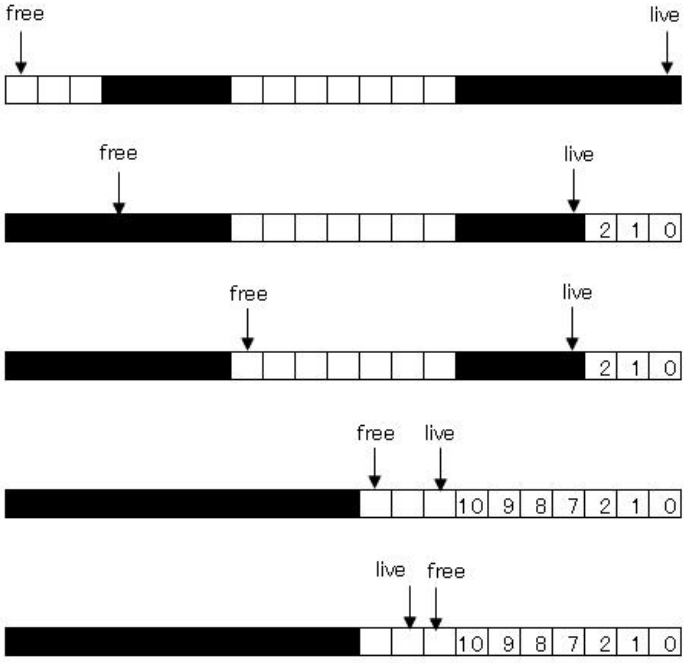
1. スループットが増加する

実コードを見たい場合

・RubyのパッチRubyLazySweep for patch

TwoFinger -Mark&Compact-

空いてる所につっこむ



アルゴリズム

先にMark＆Compactについて説明しておく。
簡単に言えば、Markを行った後で、フラグメント化したHeap領域を再配置し、
Compactionを行う手法である。

CopyingGCでは分割されたHeap領域にてCompactionを行っていたが、
このアルゴリズムではHeap領域は分割せず単一のHeap領域内で行う事が可能。
もちろん、Compactionを行うことでメモリの局所性がましキャッシュなど色々な恩恵を受けることができる。
また、オブジェクトの割り当てもポインタをずらすだけなので早い。
しかし、その分、再配置にかかる時間がプラスされ停止時間は増大する。
また、Compaction時にポインタを更新する為、原則としてConsevertiveなGCには使えない。

ここで紹介するのはMark&Compactの手法の一つであるTwoFingerアルゴリズムである。
図を見てもらうと分かるとおり「free」「live」の二つのポインタを使う。(これが名前の由来だと思う)

まず、Markフェイズで生きているObjectを数えておく。
その後、上図のように両端からscanしてゆき、freeとliveが出会ったらCompactionは終了。

その後、CopyingGCを同じ様にrootでさしていたポインタ、オブジェクト内で指しているポインタの
書き換えを行う。
その際、ポインタがliveを超えていれば移動された事になるので

```
if (pointer > live) pointer = Heap[pointer]
```

といった形でポインタを更新する。
(式は雰囲気のみ味わって欲しい)

利点

1. アルゴリズムが単純で中々高速
2. 無駄な領域を使用しない

欠点

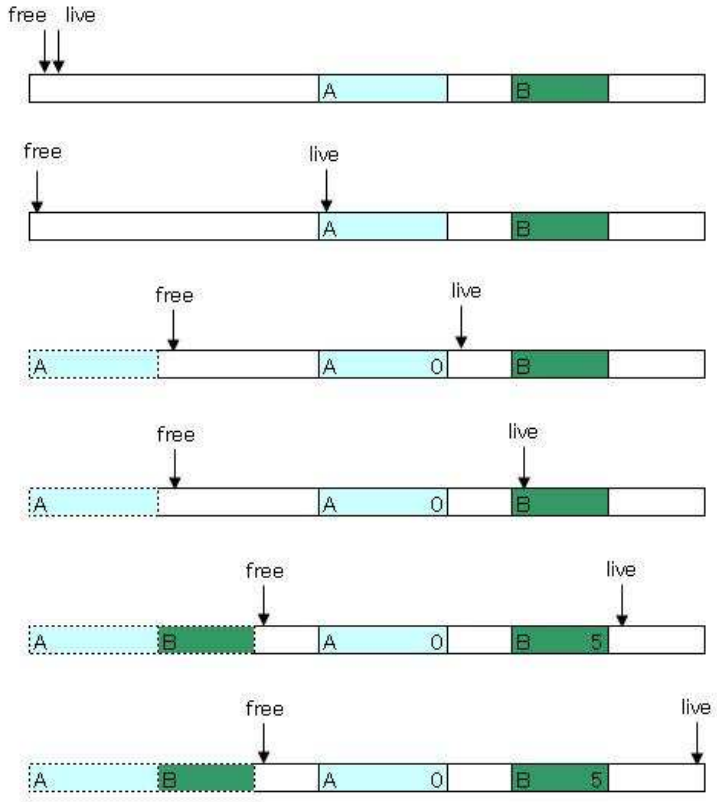
1. オブジェクトのサイズが同じでなければならない(サイズの違い毎にHeapを作る事で回避可能)
2. 順序がバラバラになり局所性が低下する

実コードを見たい場合

・不明

Lisp2 -Mark&Compact-

先頭にずらす



アルゴリズム

ヒープの先頭から詰めていく、もっとも素朴なアルゴリズム

1.マーク
rootを走査し、生きているオブジェクトに印を付ける。

2.移動先の計算
上図のようにliveポインタでオブジェクトを見つけ、オブジェクト内に持っているforwarding pointerに移動先を書き込んでおく。
その後、freeポインタはオブジェクトのサイズの分だけ移動する。
liveポインタがHeapの最後に到達したら終了。

3.ポインタ更新
オブジェクトを参照しているすべてのポインタをオブジェクト内にもっているforwarding pointerで書き換る。

4.移動
実際にforwarding pointerの通りに移動して、Compactionを終了とする。

なぜ後で移動するのか？最初に移動すればいいのにと考えたが、
オブジェクトを参照しているポインタとforward_pointerを結びつけるものが、
その実際のオブジェクトしかない為、移動する事ができない。

利点

1. 色々なサイズのオブジェクトを利用できる
2. オブジェクトの順序も保たれる

欠点

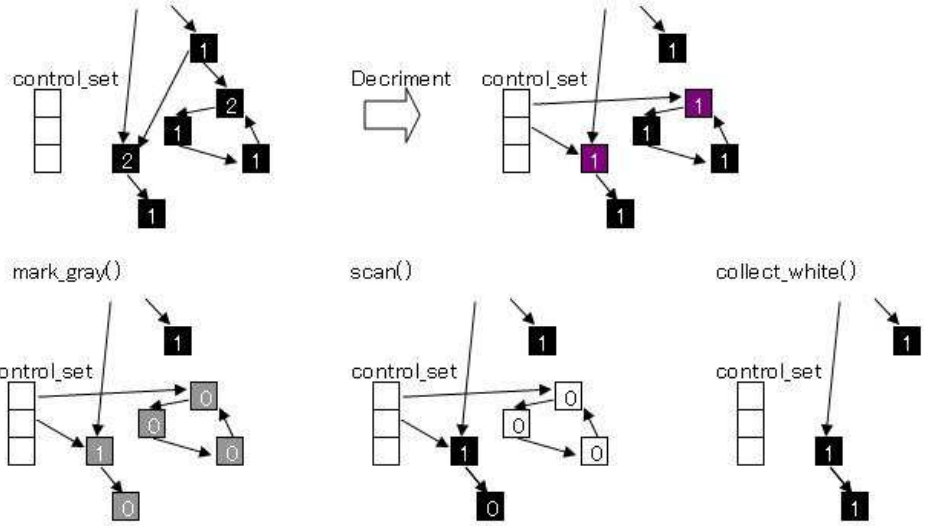
1. forwarding pointerの余分な領域がオブジェクトに必要になる
2. TwoFingerは「マーク、移動、ポインタ更新」の3パスだが、このアルゴリズムでは「マーク移動先の計算、ポインタ更新、移動」の4パスが必要であり、遅い

実コードを見たい場合

・不明

Partial Mark and Sweep -Cycle Collection-

部分的なMarkandSweep



アルゴリズム

まず、CycleCollectionとは
referenceCoountingGC(RC)で発生する循環参照を改善する為のGCである。
つまりRCの改良版。

1.インクリメント
インクリメントでは+1すると共にObjectのColorをblackにする

2.デクリメント
図のようにデクリメントでは-1すると共に、control_setにポインタをプッシュする。
その際、ObjectColorをpurpleにし、すでにpurpleであるものはcontrol_setには追加しない。

3.Mark Gray
control_setがいっぱいになったタイミングで起動し、図の様にcontrol_setにあるObjectとその子供をGrayにする。
それに伴って子供Objectのみ-1を行い、循環参照だった場合は全て0にすることが出来る。
以下擬似コード

```
MarkGray(S)
  if (color(S) != gray)
    color(S) = gray
    for C in S.child
      count(C) = count(C) - 1
      MarkGray(C)
```

4.scan
control_set内でカウントが0のもののみwhite、その他はblackに塗り分け、
カウントを減らしていたgrayオブジェクトに+1をし、カウントを元に戻す。
また、control_setからみえるObjectが1以上であった場合、そのObjectと子供については
whiteにする事はしない。
(まだ参照されている為)

5.collect_white
whiteのObjectを改修して、循環参照が解消される。
その後、control_setからpopされる。

利点

1. 循環参照が解決される
2. MarkandSweepを部分的に行え、時間の短縮になる。

欠点

1. 不明

実コードを見たい場合

・Python

・Firefox3

Mostly Parallel GC

Hans J. Boehmらによって考案された、Mark&Sweepを基本とするアルゴリズムである。
ヒープ領域をページ単位で管理し、ハードウェアのメモリ管理機能を利用する。

アルゴリズム

1. GC開始時にルートから直接参照できるセルに印をつける。
2. 1が終わると、ヒープ全体を書込み禁止にしておく。
3. マーク操作をミューテータの実行と交互に少しずつ進めていく。ミューテータにより、セルへの書き込みが行われた場合にはページ保護違反が起きる。これをキャッチし、書き込みが起こったページを記録しておく。また、そのページには書き込みができるようにしておく。
4. マーク操作が終わると、書き込みがあったページのみをスキャンし、再度マーク操作を行う(これは、ミューテータの操作とは並行できない)。
5. スイープ操作を行う。ミューテータの操作と交互に行うため、一度にn(定数)ページずつスイープする。

注意

オブジェクトの内部にマークビットを持たせると、GCのマーク操作により、ページ保護違反が起きる。
そのため、ヒープ領域の外にマーク用のビットマップを作るのがよい。

利点

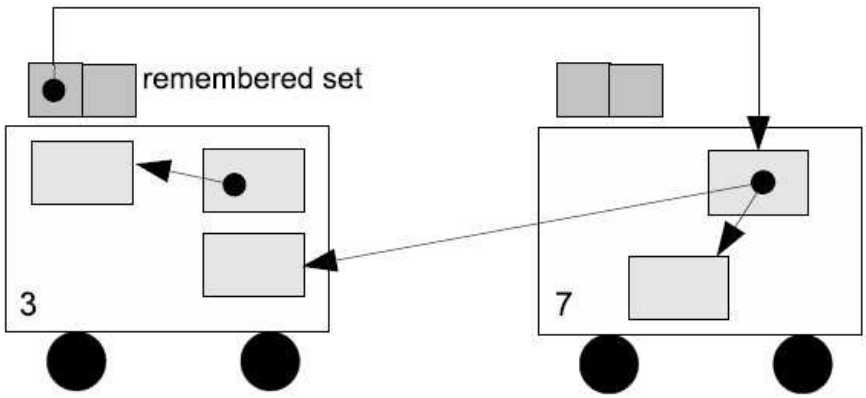
1. 一度書き込みのあったページは次回以降バリアなしで書き込みができる(書き込みの局所性を利用)。

欠点

1. ページを保護する操作は、ハードウェアに依存する。
2. ミューテータの実行と平行できない部分がある。つまり、完全なリアルタイム性は保証されない。

train gc(トレインごみ集め)

Old世代を車両に分割してインクリメンタルにGC



アルゴリズム

世代別GCの欠点であるメジャーGCの停止時間の短縮化を図った手法である。
この手法では、旧世代を固定サイズのブロックに分割し、そのブロックを車両と呼び、
一つ以上の車両を連結したものを列車と呼ぶ。
一度のメジャーGC で一つの車両のみをGCの対象とする。
各車両には通し番号をつけ、この順にGCを進めていく。

新世代と各車両にはremembered set を用意する。
新世代のremembered setには旧世代のオブジェクトからの参照を記録する。
各車両のremembered set には車両間の参照のみを記録する。
新世代から旧世代への参照を記録しておかなくてよいのは、
メジャーGC がマイナーGC の直後に行われるため、メジャーGC の際にそのような参照はないからである。
また、車両間の参照のうち番号の小さい車両のオブジェクトからから番号の大きい車両のオブジェクトへの参照も記録しなくてよ

い。
これは、GCを車両の番号順に進めていくためである。
つまりその車両がGCされる際には常に先頭にその車両があるため、参照の記録は後続のみで事足りる。
各車両のremembered set を上図 に示す。

メジャーGC では、まずルートスキャンによりごみ集め対象の車両にあるオブジェクトを適当な列車の最後尾の車両へコピーする。
ルートスキャン後はコピーされたオブジェクトをスキャンし、GC対象となっている車両のオブジェクトのみをルートスキャンと同様にコピーしていく。
次にGC対象の車両のremembered set をスキャンする。
remembered setに登録されているオブジェクトを辿り、参照元のオブジェクトの列車の最後尾の車両にコピーする。
つまり、GC対象の車両のオブジェクトと、それを参照しているオブジェクトを根こそぎ最後尾にコピーしてしまう。

コピーする際に車両がいっぱいであれば空の車両を連結し、そこにコピーする。
ごみ集め対象の車両には生きているオブジェクトは残っていないのでフリーリストに連結し、あとで再利用できるようにしておく。

remembered set に参照を記録するかをチェックするタイミングはポインタを書き換える時である。
車両番号を比較し、コピーされた車両より、参照先の車両の方が若い番号だった場合のみremembered setに記録する。
具体的なタイミングは以下となる。

1. 新世代から旧世代へコピーされる時
2. 古い車両をGCし最後尾にコピーする時
3. ミュテータによって参照が変更された時

この手法では参照関係のあるオブジェクト同士が同じ列車に配置されるようになっており、
大きな循環構造を持ったオブジェクト群もやがて列車ごと回収することができるのが特長である。

利点

1. 最大停止時間を軽減できる
2. 大きな循環構造を取り除くことができる

欠点

1. スループット性能が悪くなる

実コードを見たい場合

・不明

補足

conservativeGC(保守的GC)とexactGC(絶対的GC)

大抵のGCでは最初にルートの走査を行う。
具体的には参照先のアドレス(ポインタ)がヒープ領域内にあるかどうか？という
チェックをルートすべてに行うのだが、
この際に、もし、ヒープ領域内を指す数値が見つかった場合、
それがプログラム中に使用された数値(intなど)なのか、オブジェクトへの参照(ポインタ)なのか
判断する事ができない。
この問題にどう対処するかによって、conservativeGCとexactGCの2種類にGCは分けられる。

conservativeGC(保守的GC)

conservativeGCはこの場合、オブジェクトへの参照では無く、只の数値だったとしても
そのオブジェクトの解放は行わない。
つまり、数値であった場合にポインタの値を変更(CopyGCなど)した時はプログラムとして致命的である為、
何も触らない方向に倒すという事だ。
まさに保守的なGCだ。

利点

無駄な細工をせずに済むので処理が軽い
実装が楽

欠点

回収されないゴミが残る可能性がある
CopyingGCなどオブジェクトを移動する事ができない。

→解決方法1:オブジェクトとポインタの間に一枚ハンドルを噛まして抽象化する。
実際にハンドルの数値だけが変わり元のポインタに影響はない。
(間接参照になる為、通常の処理が遅くなる欠点)

exactGC(絶対的GC)

conservativeGCではなんでもオブジェクトとみなして、ゴミが残る可能性があったが
exactGCではポインタと数値をはっきり区別し、ゴミを残さず回収する事をポリシーとする。
ポインタと数値を区別する具体的な方法は、

整数値の範囲を1 bit減らして、その1bitをタグビットとして利用する。
stack map方式
Cの関数スタックにオブジェクトのポインタを漏らさない

がある。

利点

ゴミを全て回収するのでメモリ効率がいい
オブジェクトの移動が楽にできる。(コンパクション)

欠点

色々と制限がつく
実装が面倒

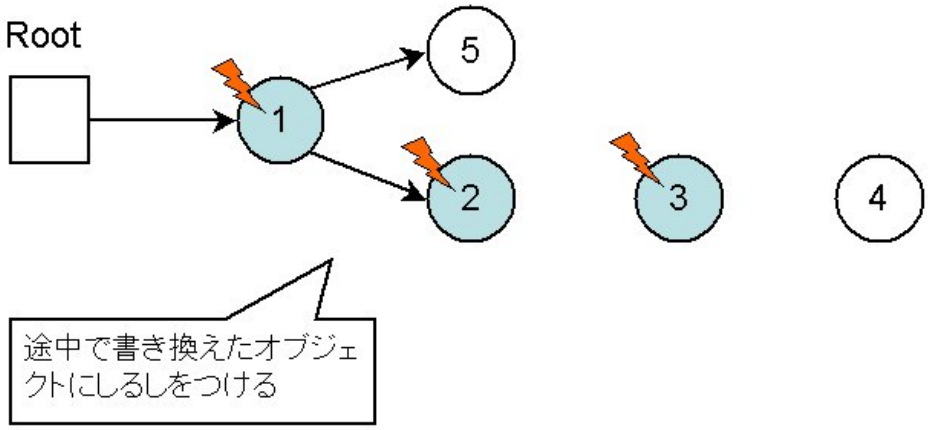
ちなみに「絶対的GC」という訳はオリジナル。べたに訳すと「確実なGC」になる。

writebarrier

Mark&Sweepアルゴリズムでは「Root から到達可能かどうか」をチェックするためにオブジェクトにマークを行っていたが、
これとは別に「書き換えられたオブジェクト」をチェックするために別種類の印をつける。
例えば、以下のような Java プログラムを実行しているスレッドが、object1 のフィールドになんらかのオブジェクトの参照を書き込む場合、
object1 に「書き込まれた」という印を付けるのである。
この処理を ライトバリア (write barrier) と呼ぶ。
object2 が null でない場合は、どんなオブジェクトでもチェックを行う。

```
void method( Class1 object, Class2 object2 ){ object1.field = object2; }
```

ライトバリアされた後の、メモリ中のイメージとしては下の図のようになる。



finalize

用語集

個人的にわからなかった単語たち

tenured

和訳:保有権のある、終身在職権のある、身分保障のある
世代別GCの古い世代を置く領域の事

ミュテータ

オブジェクトの状態を書き換える者、すなわちユーザープログラムのこと

scavenge

和訳:清掃をする

sweep

和訳:〔ほうきやブラシなどによる〕掃除、清掃

root

和訳:根、根源
GCにおいて、「確実に必要なオブジェクト」。
スタック領域、レジスタ、グローバル変数、定数などから参照されているヒープ領域を指す。
このオブジェクトを起点に必要なオブジェクトが探索され、探索されなかったオブジェクトがGCの対象になる。

chunk

和訳:大きな塊
GCの対象になる塊、場合によってはオブジェクトだったりデータだったり

compaction

Mark&Sweepを何度も繰り返すとメモリの断片化が起こるが、
その状態を整頓するする事を指す。
Windowsで言う所のデフラグの様なもの。

allocate

和訳:割り当てる
メモリ割り当てる行為。GC関連の資料では管理対象のオブジェクトを作成する事を指すことが多い。

promote

和訳:昇進させる、昇格させる
新世代から旧世代に移動させること、殿堂入りとも言う。

starvation

和訳:飢餓状態
要求に応じられるだけのフリーなデータがなくなった状態

ルート挿入

ルート集合から直接さされているセル(データ)に一括して印をつけ、GC用スタックに挿入すること

セル

GCではよくLispなどの処理系で説明が行われる。オブジェクトと同義だと思ってよい。

参考文献

一般教養としてのGarbageCollection(pdf)
微酔半壊 Copying Garbage Collector
Gaku's Space Wiki (Garbage Collection)
RHG ガベージコレクション
VisualWorks の GC の戦略発表 OHP
GC on GC
Mostly-Concurrent Mark & Sweep GC のアルゴリズム
Garbage Collection Jones&Lines著(GCの教科書。。。英語だけどね)

Garbage Collection: Algorithms for Automatic Dynamic Memory Management

Wiki作者

id:authorNari

コメント(0) | トラックバック(0) |
カテゴリ:パソコン > ガベージコレクション(GC)
「GC」をウェブ検索する

2008年06月25日(水) 00:53:32 Modified by 118.152.118.204

添付ファイル一覧(全15件)

0a901ad7.jpg (15.21KB)
Uploaded by 60.44.167.43 2008年05月23日(金) 03:07:32

46f56857.png (11.05KB)
Uploaded by 60.44.167.43 2008年05月23日(金) 03:03:49

6cdb0d76.JPG (24.32KB)
Uploaded by author_nari 2008年04月09日(水) 16:19:56

912c9e82.JPG (25.82KB)
Uploaded by author_nari 2008年03月25日(火) 18:12:05

5d0f3e56.JPG (20.62KB)
Uploaded by author_nari 2008年03月25日(火) 17:12:35

0e3a1848.jpg (24.81KB)
Uploaded by author_nari 2008年01月23日(水) 00:47:35

f689ddc7.jpg (25.53KB)
Uploaded by author_nari 2008年01月14日(月) 16:20:09

6ae949ea.jpeg (21.06KB)
Uploaded by author_nari 2008年01月14日(月) 15:28:53

73bae2c1.png (5.99KB)
Uploaded by author_nari 2008年01月14日(月) 15:27:12

bee64d39.jpg (34.26KB)
Uploaded by author_nari 2008年01月13日(日) 23:11:32

7a4d3aed.jpg (47.96KB)
Uploaded by author_nari 2008年01月13日(日) 22:38:41

991f361b.gif (11.81KB)
Uploaded by author_nari 2008年01月13日(日) 18:27:57

f15ea572.jpg (8.96KB)
Uploaded by author_nari 2008年01月11日(金) 16:59:00

9f44f30e.JPG (12.37KB)
Uploaded by author_nari 2008年01月11日(金) 16:31:19

f503b6e0.JPG (28.54KB)
Uploaded by author_nari 2008年01月10日(木) 17:00:29