究極の GC に向けて

養 安 元 気[†] 中 田 晋 平[†]
菅谷 みどり[†,††] 倉 光 君 郎[†,††]

近年，Garbage Collection (GC) は様々なスクリプト言語に実装されており，その機能は Reference Count GC (RCGC) や，Mark and Sweep GC (MSGC) など，比較的単純なものが多く，高速な GC アルゴリズムを実装したスクリプト言語は少ない．しかし，現在のスクリプト言語の要求は高く，応用範囲も広がっており，スクリプト言語の高速化が求められている．本稿ではスクリプト言語 KonohaScript における GC の経緯と，現在行なっている，世代別 GC の実装について述べたのち，これからの高速化における発展に関して報告を行う．

A View of Ultimate GC

Motoki Yoan,[†] Shinpei Nakata,[†] Midori Sugaya[†,††]
and Kimio Kuramitsu [†,††]

Recently, many scripting language implements Garbage Collection (GC). Almost their GC Algorithm is Reference Count GC (RCGC) and Mark and Sweep GC (MSGC). These algorithms are simple, and few scripting language implements complex GC algorithm. However, scripting languages are requeired in various fields and they needs to be faster. In this presentation, we disscuss detail of GC implementation, generational GC implements and view of improvement in it on our implemented programming language, KonohaScript.

1. はじめに

スクリプト言語はその書きやすさから，様々な分野での応用が期待されている．多くのスクリプト言語は開発効率を重視して，メモリ管理を動的に行う Garbage Collection(GC) を実装している．しかし，GC はランタイムにメモリ管理を行うため，静的にメモリを管理するプログラムより実行時間が長くなる．近年，スクリプト言語において速度性能が求められており，GC の速度が向上するよう開発が行われている．

我々はスクリプト言語，KonohaScript を開発している．KonohaScript は Mark-and-Sweep GC (MSGC) と Reference Counting GC (RCGC) を実装しており，KonohaScript も速度性能を向上させるために GC の高速化を行なってきた．しかし，MSGC と RCGC のそれぞれにボトルネックがある．本稿では，KonohaScript の GC を更に高速にするために，ボトルネックを明確にし，2 つの GC におけるボトルネック解消に向けた指針を示した．ただし，それだけではボトルネックの本質的な解決とは言えないため，KonohaScript のボトルネックを解消できる未実装の GC アルゴリズムの検討した．具体的に，Java, Ruby, Lua, KonohaScript(MSGC, RCGC) の GC 時間を計測して，評価を行なった上で，GC アルゴリズムの性能を定性的に検証した．それらの結果から，我々は世代別 GC を次の GC アルゴリズムとして採用した．

本稿では，世代別 GC を選択する際に KonohaScript での世代別 GC が効果があるのか，オブジェクトの生存時間について統計をとり，その評価と，世代別 GC の設計までを論じる．

本稿では，第 2 章で現在の KonohaScript におけるメモリ構造と GC の実装を述べ，第 3 章にて現在の実装における問題点を挙げ，第 4 章にて，その解決に向けて新しい GC アルゴリズムの考察を述べる．第 5 章ではオブジェクトの生存時間を測定し，KonohaScript における世代別 GC の効果に対して評価を行う．第 6 章にて今後の発展について述べ，第 7 章にて関連研究を示す．第 8 章にて本稿をまとめる．最後に第 9 章に

† 横浜国立大学大学院
Graduate School of Yokohama National University

†† 日本科学技術振興機構 CREST
Japan Science and Technology Agency/CREST

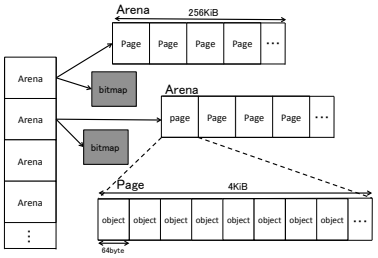

図 1 KonohaScirpt オブジェクト管理構造

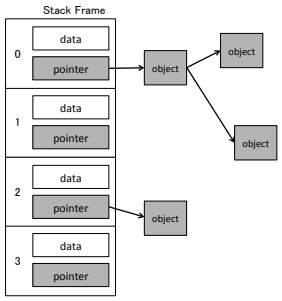

図 2 KonohaScript スタック構造

て今回の夏のプログラムシンポジウム 2011 の発表で行われた質疑応答をまとめる．

2. 現在の KonohaScript における GC の実装

現在，KonohaScript の GC は，3 層のメモリ構造を持っており，MSGC と RCGC の 2 種類が実装されている．これらの GC は，コンパイル時のオプションによって，静的に切り替えることができる．本章では，KonohaScript の GC の問題点を述べる前に，KonohaScript におけるメモリ構造と GC の実装を説明する．

2.1 オブジェクト管理機構

KonohaScript は，静的な型付けをもったスクリプト言語である．スクリプトはバイトコンパイルされ，C 言語で実装された専用の仮想マシン型インタプリタで実行される．GC を用いた自動的なメモリ管理を提供している．

まず，MSGC と RCGC の共通部分となるオブジェクト管理機構について述べる．KonohaScript のメモリ構造は図 **1** のようになっている．オブジェクトの 1 階層上には，オブジェクトを最小単位とした，ページと呼ばれる複数のオブジェクトの集合体がある．また，その上はアリーナと呼ばれる複数のページをまとめた集合体で構成されている．まず，留意すべきそれぞれ 3 つの階層と，オブジェクトの管理に必要な bitmap についての設計を以下に述べる．オブジェクトには KonohaScript のオブジェクトヘッダを定義する構造体とオブジェクトの情報を格納する body 部に別れる．

```
1  typedef struct knh_hObject_t {
2          knh_uintptr_t magicflag;
3          const struct knh_ClassTBL_t *cTBL;
4          knh_uintptr_t gcinfo;
5          void *meta;
6  } knh_hObject_t ;
7
8  typedef struct knh_Object_t {
9          knh_hObject_t h;
10         void *ref; // Object body
11 } knh_Object_t ;
```

64bit アーキテクチャでは KonohaScript の各オブジェクトはすべて KonohaScript のオブジェクトが 64byte でアラインメントされており，オブジェクトのヘッダにはクラス情報やオブジェクトの状態を表すフラグと共に各 GC が利用することができる—gcinfo—のフィールドが用意されている．

ページは 4Kbyte であり，OS が扱うページと同じ長さである．これによって，オブジェクトが参照されたときに，関連性が高いと考えられる周辺のオブジェクトも共にキャッシュに乗せられ，キャッシュヒット率が向上出すると考えている．

KonohaScript は起動と同時にオブジェクトのためのメモリ領域を確保し，アリーナとして管理する．理由は後述するが，ページやオブジェクトを全てアライメントできるようにアリーナはページアライメントされる．アリーナが用意した未使用オブジェクトを使い切った場合は，アリーナは拡張（新しいアリーナの追加) される．アリーナ拡張時は，アリーナのサイズも調整されるが，初期サイズや調整サイズは GC アルゴリズムによってポリシーが異なる．

KonohaScript は，アリーナの確保と同時に bitmap を独立したページ上に確保する．この bitmap は，オブジェクトのトレース時に対応する bit をマークする，GC 情報を管理する領域である．bitmap はオブジェクトと 1 対 1 対応するようになっており，各ページの先頭は，そのページに含まれるオブジェクトと対応する bitmap が格納されている．ページがアライメントされているため，オブジェクトのアドレスの (2 進数で) 下 12 桁を 0 でマスクするだけで，ページの先頭の bitmap を参照することができる．また，オブジェクトのアドレスからページの先頭アドレスを引き，オブジェクトサイズで割れば，オブジェクトが先頭から何番目か把握でき，オブジェクトと 1 対 1 対応する

bit を，オブジェクトのアドレスから計算量 O(0) で計算することが出来る．KonohaScript のオブジェクト管理において，もうひとつ留意するところは，静的な型情報を活かした自動的な BOX/UNBOX である．整数値 (論理値) と浮動小数点数は，必要がない限り，UNBOX された状態，オブジェクトではなく数値のまま扱われる．また，スタックにポインタや整数値が区別なく積まれた状態では，はそのデータが整数値なのか，オブジェクトのアドレスなのか判断することができない．しかし，スタックも数値とオブジェクト参照 (ポインタ) を明確に区別可能なワイドスタック (2) を採用している．そのため，オブジェクト参照に対してのみ，正確に GC 操作を行うことができる．

ただし，正確に GC 操作が可能なのは，KonohaScript 独自のワイドスタックであり，C 言語のスタックに積まれたオブジェクトは，保守的な GC 操作となってしまう．ポインタと整数を誤認する可能性を減らすために，

- オブジェクトのヘッダにあるオブジェクト情報
- オブジェクトの位置 (アリーナの中にあるか)
- 64byte にアラインメントされているか

という条件でオブジェクトかどうか判断をしている．

KonohaScript のオブジェクトアロケータは，新しいアリーナを確保したら，未使用オブジェクトの freelist を作成し，KonohaScript のコンテキストポインタに登録する．新しいオブジェクトを生成するとき，コンテキストポインタから freelist を pop することでオブジェクトを生成することができる．

2.2 RCGC の実装

RCGC は被参照数をオブジェクトがカウントし，その数が 0 になったらオブジェクトを解放するという単純なアルゴリズムである．KonohaScript を開発する際，当初 Reference Counting GC (RCGC) を採用していた．RCGC ではオブジェクトのヘッダに参照数をカウントするためのフィールドがある．オブジェクトへの参照をすべて初期化するときは，オブジェクト全体を Null で初期化する．そして変数の代入操作が発生したときに，右辺値のオブジェクトの参照数を増やし，左辺値のオブジェクトの参照数を減らす．変数の代入 $a = b$ の操作は次のような参照数の操作をともなう．

```
1
2  // a = b
3  b->h.refc++;
4  a->h.refc--;
5  if(a->h.refc == 0) {
6   //オブジェクトごとに異なる解放処理を行う
7   a->h.cTBL->free(a);
8  }
9  a = b;
```

KonohaScript は非常にシンプルな RCGC のコレクタポリシーを採用している．また，参照数が 0 となったオブジェクトは即座に解放処理を行う．解放処理とは前節で述べた freelist への push 操作である．また解放されるオブジェクトがさらに内部参照を持っている場合は深さ優先探索を用いて順次参照数を減らす処理を行う．この時，参照数が 0 となれば再帰的にオブジェクトの解放処理を繰り返す．

RCGC では変数の代入処理の他にメソッド呼出などワイドスタックにオブジェクトを 配置する際にも参照数操作を行う．そのため，GC 処理がボトルネックとなることがわかった．それ以降，参照数操作のボトルネックを取り除くため，トレーシング方式の GC の検討を行った．

2.3 MSGC の実装

KonohaScript ではトレーシング方式の GC として BitMap を用いた Mark-sweepGC を採用した．MSGC の実行の流れは以下のとおりである．

まず，KonohaScript はスクリプトをコンパイルする際に，MSGC が安全に GC を実行できる場所に GC セーフポイントをバイトコード命令として挿入する．また，C の拡張ライブラリを記述する際にも同様に，マシンスタックや CPU レジスタが参照しているオブジェクトがルートセットからたどれる箇所に GC セーフポイントを置いている．GC セーフポイントは仮想マシン上では以下の箇所に自動的に挿入される．

- NEW 命令 (オブジェクト生成のための命令) を実行する直前
- BOX 命令 (数値を boxing するための命令) を実行する直前
- for 文，while 文などループ文の先頭

GC セーフポイントでは，メモリ利用状況をチェックし，メモリ利用率がスレッシュホールド以下であれば MSGC を起動する．

MSGC の実行は以下の手順で行う．

(1) アリーナのすべての bitmap を 0 でクリアする．
(2) 深さ優先探索アルゴリズムに従いルートセットから参照関係を辿り，到達可能なオブジェクトが対応する bitmap に記録する.(mark フェーズ)
(3) アリーナのすべての bitmap を探索し，オブジェクトがごみである場合 (ビットが 0 である場合) にはオブジェクトの回収を行う.(sweep フェーズ)

なお，MSGC では mark フェーズにおいて，collec-

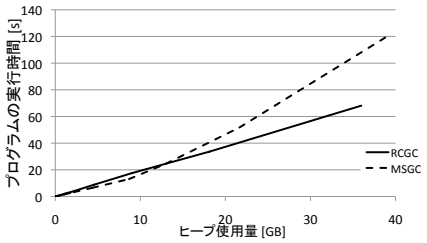

図 3　ヒープサイズと実行時間の比較

tor は mutator の実行を完全に停止させる STOP THE WORLD 方式を採用している．また sweep フェーズにおいて，オブジェクトの回収操作は RCGC の freelist への push と同じ操作であるが，オブジェクトの再帰的な回収は行わない．また，sweep フェーズが終わった段階で，オブジェクトの回収率を確認し，回収率が低い場合にはアリーナを拡張し，collector の起動する頻度を下げる工夫を行なっている．

3. RCGC，MSGC の実装における問題

本章では，RCGC，MSGC の実装における問題点を述べる．

3.1 RCGC と MSGC の問題点

RCGC は実装自体はシンプルであり，最大停止時間が短いが，代入操作の度に参照カウントが発生してしまうため，GC 時間が増加する．代入処理を行う際に，必ず参照カウントのインクリメントとデクリメントによるキャッシュミスが発生する．さらに，参照カウントが 0 になった場合の条件分岐でパイプラインハザードが発生する可能性があり，速度向上の障害となっている．数値演算の場合の代入は数値をオブジェクトに boxing しないことで参照カウントの更新を行わなくて済むが，KonohaScript のワイドスタックに StackTrace の情報を載せる必要があり，どうしても参照カウントの更新が入ってしまう．

MSGC はトレース方式の GC のため，mutator がオブジェクトをしきい値まで作成するなどの，GC 発動条件を満たさない限り，GC は発生しない．そのため，代入毎に参照数操作が発生する RCGC よりスループットが高く，実行時間は短くなるが，その分，GC1 回の最大停止時間が伸びてしまう．

3.2 大規模メモリ環境下での問題点

また，以前の研究内容として大規模メモリ環境下での GC 性能の測定を行なった．

図 3 は，大規模メモリ環境を想定して 96Gbyte のメモリを積んだマシンを実験環境として，実行時間を RCGC と MSGC で比較したものである．測定には二分木の深さを変更することでヒープサイズを変えられるようなベンチマークを用いた．図 3 から，KonohaScript の MSGC は使用するヒープが約 13GByte を超えると，RCGC よりも実行時間がかかるということがわかった．これは，メモリを大量に消費するようなアプリケーションの場合，オブジェクトの探索範囲が拡大するため，マークにかかる時間，スイープにかかる時間が増加し，MSGC の最大停止時間が上がったためだと思われる．

4. 問題への解決に向けた考察

本章では前章で述べた問題点に関して，RCGC，MSGC 両方の対応を今後の発展として述べた後，その他の GC アルゴリズムについて考察を行う．

4.1 RCGC，MSGC の発展

現在，KonohaScript で実装されている RCGC と MSGC それぞれの問題点を改善する方法を考察する．

RCGC では，Zero Count Table (ZCT) を用いた遅延参照カウント GC(Defferd Reference Count GC) による高速化が考えられている．これは RCGC の被参照数が 0 になったオブジェクトでもすぐには解放せず，管理テーブルを用意し，そこに集めておく．そして，管理テーブルがしきい値に達した時，まとめて解放を行う．KonohaScript ではオブジェクトのマークフェーズ使われる bitmap を ZCT として利用することで，遅延参照カウント GC が実装可能であると考えている．

MSGC ではスイープフェーズを行わない代わりに，mutator から要求がある度にオブジェクトを要求分だけ解放する遅延スイープ GC(Lazy Sweep GC) を実装することが考えられる．スイープフェーズでヒープ全体を走査する代わりに，mutator がすぐに使うオブジェクトを解放するため，mutator のキャッシュを汚すことがなく，高速化が期待できる．

4.2 他の言語との比較，評価

GC が実装されている様々なプログラミング言語でも 3 と同様のベンチマークで GC 時間の計測を行なった．比較に用いた言語は以下のとおりである．

- Java(Java 1.6.0, HotSpot 17.1), 世代別 Copy & Concurrent Mark-sweep GC: 新世代は Copying GC, 旧世代は Concurrent Mark-sweep GC を用いる．世代別 GC とは，トレーシング方式の一種で，新しく作られたオブジェクトは新世代オブジェクトと呼ばれる．GC から生き延びた回数を各オブジェクトが記録しておき，一定回数の GC

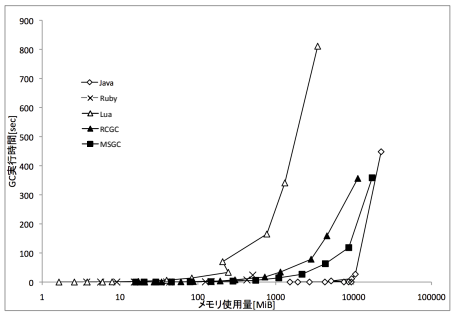

図 4 binarytrees 言語別 GC 時間

を経ても回収されなかったオブジェクトは旧世代オブジェクトに昇格する．旧世代オブジェクトはプログラム中で長期間生き残る可能性が高い[2] として，minor GC のトレース対象から外すことができるため，探索時間を削減できるアルゴリズムである．

- Ruby(1.9.0), Mark-sweep GC: 保守的な Mark-sweep を使用．
- Lua(5.1.4), インクリメンタル Mark-sweep GC : インクリメンタル Mark-sweep アルゴリズムはトレーシング方式の一種で，メモリ領域の一部を部分的に探索，回収を行いながらプログラムを動作させる．使用メモリ領域の全体を探索する Mark-sweep と異なり，探索時間がプログラム実行時間全体に分散されるため mutator の停止時間が短い．

図 4 の結果より，今回比較したプログラミング言語では Java の GC が最も性能が良いということが分かった．

4.3 他の GC アルゴリズムの考察

MSGC の問題点は最大停止時間が増加することである．これを防ぐためには，探索範囲を狭めることや，探索を複数に分けることが挙げられる．そこで，言語間の GC 比較を元に，Copying GC, Major GC と Minor GC が共に MSGC である世代別 GC (本稿では世代別 MSGC と呼ぶ), Incremental GC を新しく実装する GC アルゴリズムの候補にあげ，GC の各フェーズの探索範囲と，ヒープサイズ， スループット，停止時間について MSGC を比較対象とし，評価を行なった．

Copying GC の場合，スイープが一切必要ないことが，探索範囲の削減に有効とであり，To 空間へオブジェクトを移動する際のコンパクションでフラグメンテーションがなくなるメリットもある．しかし，スキャンフェーズが有ることや，ヒープサイズが 2 倍になってしまうことがデメリットとして挙げられる．また，Zorn[4] らの研究で MSGC と Copying GC では，MSGC の方が性能が劣るが，メモリ使用効率が良いと結論づけている．世代別 MSGC のマーク範囲は MinorGC の場合，Tenure なオブジェクトをトレースしないため，範囲が狭いと考えられる．スイープ範囲は，遅延スイープの場合，ヒープを一括してスイープしないため，より最大停止時間が短くなると予想される．Minor GC はトレース時に Tenure なオブジェクトをトレースしないため，ヒープサイズが増加しても速度性能が MSG 程下がらないことが期待できる．しかし，Major GC ばかりが発生するデータベースのようなアプリケーションの場合，ライトバリアのオーバーヘッドがある分，世代別 GC の効率は MSGC よりも悪くなる．Incremental GC は 1 回の MSGC を分割して GC を行うような GC アルゴリズムのため，最大停止時間が抑えられることが期待できる．しかし，ライトバリアがあるためスループットは低下すると考えられる．以上の考察より MSGC の問題点であった，最大停止時間を最も改善できると予想される GC アルゴリズムは世代別 GC であるとした．そこで本稿では，オブジェクトの生存時間の情報を取り，KonohaScript でも世代別 GC の効果が期待できるかを検証する予備実験を行なった．

5. 実　　験

本章では，KonohaScript における世代別 GC の効果を検証するために，オブジェクトの生存時間を測定した．実験環境として表 1 の環境を用いた．

OS	MacOSX 10.6.8
CPU	Intel Core i7
CPU Clock	2.7GHz
Core	2cores
L3 (LLC)	4MByite
主記憶	DDR3 8GB

表 1 実験環境

- ao-benchmark: レイトレーシングによるレンダリングアプリケーション
- binarytrees: 二分木を作り大量にメモリを消費するアプリケーション
- large script: 10,000 個の関数を作成し，実行するアプリケーション
- log parser: 本実験に用いた，オブジェクトの生存時間を集計するログパーサ

オブジェクトの生存時間の分布を図 5 から図 8 に示

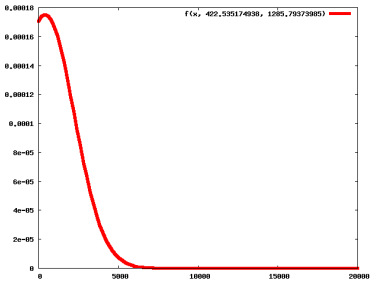

図 5 オブジェクト生存時間分布 (ao-benchmark)[ms]

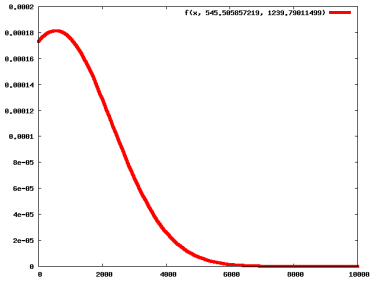

図 6 オブジェクト生存時間分布 (binarytrees)[ms]

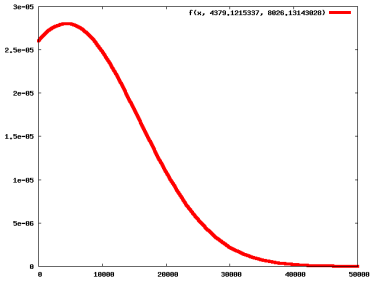

図 7 オブジェクト生存時間分布 (large script)[ms]

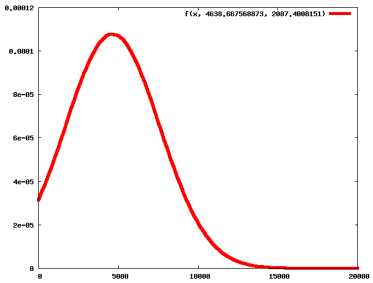

図 8 オブジェクト生存時間分布 (log parser)[ms]

す．なお，横軸が生存時間 [ms]，縦軸がオブジェクトの個数である．それぞれの実行時間，オブジェクトの平均生存時間，その標準偏差は，表 2 のようになった．

今回実験で使用したアプリケーションの全てのオブジェクトの標準偏差は，ao-benchmark などのプログラムで，多くのオブジェクトがすぐに死ぬことが確認できた．binarytrees や large script など，アプリケーションの実行時間が短いものは，標準偏差が高くなっているものと思われる．log parser に関しては，log の結果をパースし，プログラムが終了し，ファイルに出力するまで死なないオブジェクトが多く残るため，標準偏差が高くなってしまったものだと考えられる．

benchmark	実行時間	平均生存時間	標準偏差
ao-bench	19260 [ms]	422.535 [ms]	1285.793
binarytrees	9282 [ms]	545.505 [ms]	1239.790
large script	7201 [ms]	4638.667 [ms]	2087.408
log parser	36672 [ms]	4379.121 [ms]	8026.131

表 2 実験結果

以上の結果から，KonohaScript でも多くのオブジェクトがすぐに死ぬことが確認でき，世代別 GC を実装した際の速度向上が期待できることがわかった．

6. 今後の発展

本章では，今後の発展として，KonohaScript での世代別 GC の設計と実装について述べた後，マルチスレッド GC について論じる．

6.1 世代別 MSGC の設計

KonohaScript のオブジェクトには一般的な世代別 GC のオブジェクトの昇格に不可欠な，オブジェクトの移動ができないものがある．そこでこの章では，世代別 GC の設計について論じた後，移動ができないオブジェクトがどのようなものかを論じ，移動ができないオブジェクトをどのように対処するかを論じる．

まず，世代別に分けるため，Surviver 領域と Eden 領域を別領域として作成する．

KonohaScript では移動が不可能なオブジェクトがあるため，オブジェクトの移動を行わない MSGC を Major GC, Minor GC 共に採用した．

6.1.1 オブジェクトの構造

世代別 GC の KonohaScript のオブジェクトは図 10 のようなオブジェクト構造を持っており，オブジェクトのヘッダに含まれている RCGC の被参照数を管理していた gcinfo を利用して，オブジェクトの昇格先のポインタ，Tenure フラグ，RememberSet フラグ，オブジェクトの年齢のフラグを管理する．KonohaScript のオブジェクトが 64byte でアラインメントされていることから，2 進数における下 7 桁は必ず 0 であると決まっているため，それぞれのフラグを挿入することが出来る．また，昇格先のポインタが必要な場合は，下 7 桁を 0 にマスクすることで得ることが出来る．

現在，オブジェクトの年齢に関しては決め打ちで 4

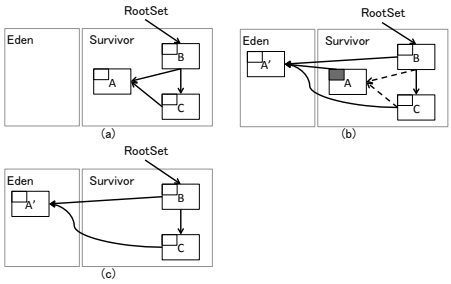

図 9 オブジェクトの昇格

回 GC を生き延びたら昇格されるようにしている．年齢のカウントは gcinfo における下 2 桁を用いて管理されていて，マークフェーズにおいてインクリメントされる．

6.1.2 オブジェクトの昇格

トレースの処理中に，年齢がしきい値を超えたオブジェクトにたどり着いた場合，オブジェクトの昇格が行われる．昇格までの手順は以下のとおりである．

(1) 昇格先のオブジェクトに昇格元のオブジェクトの情報をコピーする
(2) 昇格先のポインタを gcinfo に格納する
(3) 昇格元のオブジェクトを参照しているオブジェクトのポインタを変更する
(4) 昇格元のオブジェクトを解放する

上記の手順の (1) から (3) はトレースの処理中に行われる．トレースの初期状態が図 9 の (a) のようであるとする．各オブジェクトの左上にある四角は，gcinfo を表す．オブジェクトが昇格すると分かると，Eden 領域から昇格先となるオブジェクトを pop して，昇格元のオブジェクトの情報をコピーする．このとき，オブジェクトは 64bytes で統一されているため，それより大きなデータを持つオブジェクト (例えばオブジェクトの配列) は，その内容もあわせてコピーする必要がある.(図 9 の (b) のオブジェクト A') 昇格先にオブジェクトをコピーした後，昇格元の gcinfo に昇格先のポインタを格納する.(図 9 の (b) のオブジェクト A の左上) これ以降，昇格元をトレースしてきたオブジェクトは，gcinfo に昇格先のポインタが格納されているか判断をして，格納されていれば，昇格先にポインタをつけかえる．参照元のオブジェクトはマークされていないため，スイープフェーズにて解放される.(図 9 の (c))

6.1.3 移動が不可能なオブジェクト

2.1 で述べたように，KonohaScript では C のスタックに乗っているものが整数なのか，オブジェクトへのポインタなのか確実に判定できない．仮に C のスタックに載っているものをポインタと仮定し昇格してしまうと，それが整数値であった場合に，その値を破壊してしまう．したがって，C のスタックに載っているオブジェクトは昇格させるべきではない．

しかし，C のスタックに載っているオブジェクトは，将来的にどこかから参照されるため，今回の実装では，C のスタックをトレースする際は，マーク処理をしても，年齢 (何度 GC を生き延びたら昇格されるか) をカウントしないことで，対処をした．

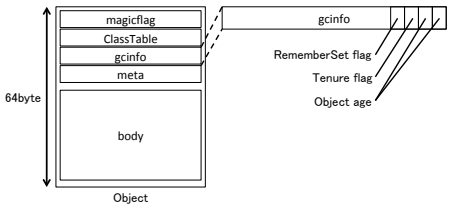

図 10 KonohaScript オブジェクト構造

6.2 マルチスレッド GC

KonohaScript は現在，マルチスレッドの mutator による実装を行なっており，スイープフェーズのマルチスレッド化を考えている．マークフェーズは Stop The World で行うことで，スレッドローカルなオブジェクトとグローバルなオブジェクトを mutator に分けさせる手間を省かせることができる．その代わり，ロックフリーなアルゴリズムを調査していきたいと考えている．

現在は，一つのスレッドが Collector を起動する前に他のスレッドにロックをかけて，安全にマークフェーズを行うように設計をしている．その後，各スレッドがそれぞれのヒープをスイープするようにしている．

7. 関 連 研 究

関連研究として，鵜川らが Ruby において Mostly-Copying GC の実装を行なっている[5]． 鵜川らの Mostly-Copying GC ではヒープを複数のオブジェクトが入るブロックとして分割する．曖昧なポインタから参照されたオブジェクトは Immovable マークをつけ，一つでも Immovable マークのついたオブジェクトの格納されたブロックは移動させない (昇格する) ようにしている．鵜川らの研究では，使用した RubyVM, YARV[3] の拡張モジュールの作り方によっては曖昧なルートからのポインタが多く含むことになってしまう．オブジェクト一つ一つを To 空間に入れては，曖昧なルートから参照されたポインタを From 空間に戻す Bartlett のアルゴリズム[1] よりも，ブロックとしてまとめることで効率が上がる．本稿の目指す世代別 GC

でのオブジェクトの移動でも C のスタックなど，曖昧なルートから参照されるポインタは移動させないことが共通点となるが，Copying GC を主目的としている点と，曖昧なルートの量という，GC を実装する言語の環境が異なる．

8. ま と め

本稿では，KonohaScript における現在の GC の実装とその問題点について述べ，その問題点を解決するための GC アルゴリズムを考察し，候補であった世代別 GC が KonohaScript に対しても有効であるか，実験を行った．今後の方針としては，世代別 GC の実装を行い，移動ができないオブジェクトへの対応を明確にして行きたい．

9. 質 疑 応 答

Q ポインタの移動が難しいという説明があったが，そこについて詳しく知りたい．(九州工業大学 小出様)

A ライブラリをバインドする際に，KonohaScript はラッパー関数を用いて，ライブラリ内でアロケートされた構造体をラッピングしています．この構造体がさらに参照しているポインタまでは KonohaScript は認識できないため，移動させることができません．また，KonohaScript 独自のスタックは正確な GC が可能ですが，C のスタックにあるポインタは可能性が低いとはいえ，整数値とオブジェクトのポインタを誤認することが考えられます．

Q ページのダーティビット，など OS 依存が強くなると思う．それでもプラットフォーム非依存を貫くのか？ それとも，OS の援助を受けてでも速い GC を作るのか？ (東京大学 田中様)

A GCC 4 以降でサポートしているコンパイラの機能は利用しますが，それ以外は使わずに，マルチプラットフォーム化を行う予定です．

Q RCGC はもう少し早くなるような気がするがどうか？

A zero count table いれたら早くなると皆さまから言われますが，KonohaScript では MSGC の方が 40%程性能が向上したため，現在はこちらを利用しています．

Q Konoha は OOP，世代別 GC に向いたオブジェクトのアロケートをしてくれると思うが，それも測ってくれるのか？

A 世代別 GC は現在製作中です．完成次第，計測を行いたいと思います．

Q 次の GC を作るポイントで，ロックフリー，並列化があるが，それは GC だけなのか? mutator も並列化してくれるのか? (九州工業大学 小出様)

A 両方共やっていきたいと考えています．しかし，現在は GC の方がネックになっています．スレッドをただ入れるのは簡単ですが，GC との整合性をとるが難しいです．

Q 現在の GC はシングルスレッドモデルだが，GC と Actor モデルが絡むと難しいのか?

A そんなことはないです．GC 作成時に，シングルスレッドモデルを採用したのはシンプルに実装しようと考えた為です．

謝辞 本研究は，JST/CREST「実用化を目指した組込みシステム用ディペンダブル・オペレーティングシステム」領域の研究助成を受けて行われた．

参 考 文 献

1) Joel F. Bartlett. Compacting garbage collection with ambiguous roots. *SIGPLAN Lisp Pointers*, 1:3–12, April 1988.
2) Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26:419–429, June 1983.
3) Koichi Sasada. Yarv: yet another rubyvm: innovating the ruby interpreter. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 158–159, New York, NY, USA, 2005. ACM.
4) Benjamin Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 87–98, New York, NY, USA, 1990. ACM.
5) 鵜川 始陽. Ruby における mostly-copying gc の実装. 情報処理学会論文誌. プログラミング, 2(2):1–12, 2009-03-23.