Rubyにおけるメモリ管理改善手法の提案

東京工業大学
大学院情報理工学研究科
数理・計算科学専攻

太田 眞敬
(08M37090)

平成21年度 修士論文

指導教官 佐々 政孝 教授

平成22年1月29日

東京工業大学
大学院情報理工学研究科
数理・計算科学専攻
佐々研究室 太田眞敬

Ruby におけるメモリ管理改善手法の提案

Ruby(ルビー) はまつもとゆきひろ氏によって開発された，オブジェクト指向スクリプト言語である．クラス定義，ガーベージコレクション (GC)，正規表現，マルチスレッド，イテレータ，例外処理などの機能がサポートされている．当初は同じスクリプト言語の Perl と同じように，正規表現を用いた文字列処理など，簡単なテキスト処理を，より容易に行うことを目的とした言語，という位置づけであった．しかし海外で Ruby を用いた Web アプリケーションフレームワーク Ruby on Rails が普及したことによって，Web アプリケーション開発言語として利用されることが多くなってきた．

本稿では Ruby におけるメモリ管理に焦点を当て，その改善手法について述べる．Ruby ではメモリ管理を行う上で RVALUE と呼ばれる固定長（5 Word）の大きさの構造を用いることでメモリ管理を容易にし，かつメモリの断片化を防いでいる．ただし固定長であるために，この大きさを超えるオブジェクトを生成する際には malloc を用いてさらにメモリ領域 (外部領域) を確保する必要がある．

GC を考える上で最も重要となってくるのが RVALUE である．RVALUE の情報を用いて，参照を辿り，マークなどを行うのだが，１つ問題点が存在する．それは Ruby が C 言語によるライブラリ (拡張ライブラリ) 作成を許している点である．拡張ライブラリ内では自由に Ruby オブジェクトを生成することが可能であり，それらのオブジェクトは Ruby 側に伝達しなくてもよいことになっている．これにより容易なライブラリ作成が可能となるが，GC を難しくしている．それに伴い，GC に関する様々な研究がされているが，それらは RVALUE に関するものがほとんどである．

そこで本稿では外部領域に注目し，特定の型に関してリニアアロケーションを用いる．リニアアロケーションとは，予めリニアアロケーション用の広大な領域を確保しておき，オブジェクト作成の際に先頭から順にアロケーションを行う手法である．これによりフリーリストによる管理の必要がなくなり，高速なアロケーションが実現される．さらに外部領域に関してコンパクションを行う．コンパクションとはメモリ上で空き領域を埋めてオブジェクトを再配置する手法である．これにより断片化の防止およびリニアアロケーション用領域の増大が見込める．

実験では動作確認用のテストプログラム，Ruby on Rails を用いた WEB アプリケーション，RDoc を用いた．

その結果，テストプログラムでは実行時間の改善が見られたが，WEB アプリケーションなどでは逆に遅くなる結果になった．この実行時間増加の原因はコンパクションによるコストの増加によるものだと考えられるため，それを改善する方法について述べる．

目 次

第 1 章 はじめに **4**
- 1.1 背景 4
- 1.2 本研究の概要 5

第 2 章 メモリ管理 **6**
- 2.1 メモリ領域の構成 6
- 2.2 ヒープ領域管理 8
  - 2.2.1 メモリ管理ルーチン 8
- 2.3 ガーベージコレクション (GC) 9
  - 2.3.1 GC の基礎 9
  - 2.3.2 参照カウント GC 13
  - 2.3.3 トレース方式 GC 14
  - 2.3.4 コンパクション 19
  - 2.3.5 不安全な言語に対する保守的な GC 25
  - 2.3.6 その他の GC 25

第 3 章 Ruby **28**
- 3.1 言語 28
  - 3.1.1 エントリーポイント 28
  - 3.1.2 オブジェクト 28
  - 3.1.3 変数 28
  - 3.1.4 クラス・メソッド 29
- 3.2 実装 30
  - 3.2.1 オブジェクト 30
  - 3.2.2 VALUE 埋め込みオブジェクト 33
  - 3.2.3 オブジェクト管理 34
- 3.3 Ruby における GC 39
  - 3.3.1 マーク 39
  - 3.3.2 ルート集合 42
  - 3.3.3 スイープ 42

第 4 章 予備調査 **44**
- 4.1 Rails アプリケーションにおけるオブジェクトの割合 45
- 4.2 RDoc におけるオブジェクトの割合 45
- 4.3 Rails アプリケーションにおける文字列及び配列オブジェクトの詳細 46
- 4.4 RDoc における文字列及び配列オブジェクトの詳細 46

第5章 提案手法 49
5.1 リニアアロケーション 49
5.2 コンパクション 50
5.3 Ruby における改善手法 51
5.3.1 外部領域に対するリニアアロケーションアルゴリズム 51
5.3.2 外部領域に対するコンパクションアルゴリズム 53

第6章 実験・評価 54
6.1 ソースコード 54
6.2 実行時間・GC 時間・GC 回数 55
6.3 文字列・配列オブジェクトのメモリ使用量 55

第7章 考察 60
7.1 実行時間・GC 時間・GC 回数 60
7.2 文字列・配列オブジェクトのメモリ使用量 61
7.3 今後の方針 62

第8章 関連研究 63
8.1 Mostly-Copying GC 63
8.2 ヒープ領域の拡張 63
8.3 スナップショット GC 63
8.4 ビットマップマーキング方式 GC 63

第9章 まとめ 65

第1章　はじめに

1.1　背景

Ruby(ルビー)[11][12] はまつもとゆきひろ氏によって開発された，オブジェクト指向スクリプト言語である．機能として，クラス定義，ガーベージコレクション (GC)，正規表現，マルチスレッド，イテレータなどがサポートされている．

当初は同じスクリプト言語の Perl[16] と同じように，正規表現を用いた文字列処理など，簡単なテキスト処理を，より容易に書くことを目的とした言語，という位置づけであった [10]．しかし海外で Ruby を用いた Web アプリケーションフレームワーク Ruby on Rails[7] が普及したことによって，Web アプリケーション開発言語として利用されることが多くなってきた．

Ruby は多くのプログラミング言語と同様に，GC による自動メモリ管理を採用している．そのためプログラマは煩雑なメモリ管理を行わなくともよく，メモリを意識せずにプログラムを書くことが可能となる．

現在 Ruby の代表的な実装には以下のものがある．

- MRI(Matz' Ruby Implementation)
  公式に配布されている，ヴァージョン 1.8.x 以前の実装．松本行弘氏によって開発されはじめた C 言語による実装であり、最も広く使われている。後述 JRuby に対して CRuby と呼ばれることもある。

- YARV(Yet Another Ruby VM)[18]
  公式に配布されているヴァージョン 1.9.0 以降の実装．笹田耕一氏によって開発されはじめ，MRI を拡張した公式な処理系．これはソースコードをジャスト・イン・タイム (JIT) 方式でバイトコードへコンパイルした後，バイトコードを仮想機械上で実行するインタプリタである．

- JRuby[15]
  Java ベースの実装．純粋な Java で実装が行われているため，プラットフォーム非依存の利用が可能．ほとんどの Ruby クラスが組み込みで提供されている．JIT 方式に加え，(アヘッド・オブ・タイム)AOT 方式の Java バイトコードへのコンパイラも用意されている (AOT 方式でコンパイルした Java バイトコードは JRuby が無くても他の Java プラットフォーム上で動作させることが可能となる)．

- IronRuby[4]
  .NET Framework 上で Ruby を動作させる実装であり，.NET Framework のライブラリと連携させることができる．JIT 方式のバイトコードインタプリタ．共通言語基盤に準拠した実装で動作するため、プラットフォーム非依存の利用も可能．

- RubyCocoa，MacRuby
  いずれも Mac OS X 上で動作する Ruby 実装．Cocoa を含む様々なフレームワークと

の連携が可能．RubyCocoa は Mac OS X Leopard 以降に標準でインストールされており，MacRuby は RubyCocoa の問題点を解決するために開発されている．

本稿ではこれらの実装の中から，YARV を対象とすることとする．

Ruby では GC において保守的マークアンドスイープアルゴリズムを採用している．保守的な GC を採用している理由の１つには，Ruby が C 言語によりライブラリ (拡張ライブラリ) を作成することが可能だからである．拡張ライブラリはライブラリ作成者が自由に Ruby オブジェクトを作成することが可能であり，それらのオブジェクトを Ruby 側に伝達しなくてもいいことになっている．このような仕様となっている理由は，JNI[8] での Java 仮想マシンの拡張を行う際に必要とされる冗長なオブジェクト処理を記述しなくとも，容易にライブラリを作成することを可能とするためである．その結果，GC にコピーアルゴリズムなどを単純に適用することができなくなり，マークアンドスイープアルゴリズムを採用している．

このように Ruby は，他の GC を採用している言語と比べると，効率が悪い．ただし Ruby は当初，簡単なテキスト処理などで用いられることが多かったため，GC が問題となることは少なかった．しかし現代では前述のように Web アプリケーションなど長時間稼動するアプリケーショも Ruby で開発される機会が多くなり，それに伴ない GC に関して，速度や効率などが問題となってくることが多くなってきた．

このような背景から Ruby におけるメモリ管理のより効率的な手法を提案する．

1.2 本研究の概要

Ruby ではメモリ管理を行う上で RVALUE と呼ばれる固定長（5 Word）の大きさの構造を用いることでメモリ管理を容易にし，かつメモリの断片化を防いでいる．ただし固定長であるために，この大きさを超えるオブジェクトを生成する際には malloc を用いてメモリ領域をさらに確保する必要がある．RVALUE に関して論じている論文は数多く存在するが，malloc 領域のメモリ管理に関して論じている論文はそれほど存在しない．そこで本論文ではこの malloc 領域に注目し，特定の型に関してリニアアロケーションを用いることで，高速な malloc を実現し，実行速度を改善する．さらに malloc 領域に関してコンパクションを行うことで，メモリ使用領域に関して改善を行う．

第2章　メモリ管理

2.1　メモリ領域の構成

この節ではまず C 言語のようにプログラムをコンパイルし目的プログラムを生成した後，目的プログラムを実行した際のメモリ構造について考える．目的プログラムとは，コンパイラによって作成されるファイルのことである．Ruby におけるメモリ構造に関しての説明はその後行う．

プログラムは実行される際，独自の論理アドレス空間で実行される．プログラムで扱う値は，それぞれ確保された記憶場所に格納される．論理アドレス空間の管理はコンパイラ，オペレーティングシステム，目的プログラムの 3 者が関与している．オペレーティングシステムは論理アドレス空間から物理アドレス空間への変換を行う．物理アドレス空間はメモリ上に点在している場合が多い．しかし物理アドレス空間の情報はオペレーティングシステムによって隠蔽されるため，プログラム作成者は点在を気にせずプログラムを作成することが可能となる．

論理アドレス空間における実行時の目的プログラムは図 2.1 に示すようにコード領域とデータ領域によって表現される．

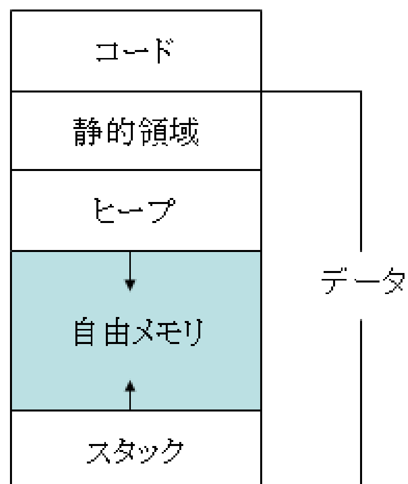

図 2.1: 目的プログラムのメモリ領域の構成

生成される目的コードの大きさは，コンパイル時に決定できるため，コード領域におくことができる．通常この領域は低アドレスメモリ番地を用いる．同様に大域変数やコンパイラが使用する情報などがコンパイル時に決まるため，それらもコード領域と同じように，静的に配置することが可能である．このようにコンパイル時に静的に配置する領域のことを静的領域と呼び，通常コード領域とは別である．メモリ上に静的に配置することで，目的コード

中に配置したデータのアドレスを直接埋め込むことが可能となるため，実行が高速になる．

ここまではコンパイル時に決定できる情報のための領域だが，実行しなければ把握できない情報も存在する．そういった情報を配置する領域がスタック領域とヒープ領域の 2 つである．これら 2 つの領域は動的であり，領域の大きさはプログラム実行につれて変化する．そしてお互いの領域は，相手の方へ向かって伸びるように配置することが多い．

スタック領域は関数，メソッドなどが呼び出されるごとに伸び，それらの呼び出しが終わることで縮む．このようにスタック領域には，1 回の関数呼び出しで必要がなくなる，比較的生存期間が短い情報が配置される．それに対してヒープ領域は比較的生存期間が長いオブジェクトを管理するための領域である．C 言語では malloc で確保が行え，free で開放が行える．Java の場合は new で確保を行い，GC によって開放が行われる．

Ruby の場合

Ruby について考える．Ruby はインタプリタであると同時に，C 言語で作られたプログラムでもある．そのため Ruby 実行時には，Ruby を実行するために必要なコードや静的領域がメモリ上に配置される．さらにその上で Ruby で書かれたプログラムを読み込み，それらを解析し必要な情報を適宜，スタックやヒープに配置していく．これを表した図が図 2.2 である．

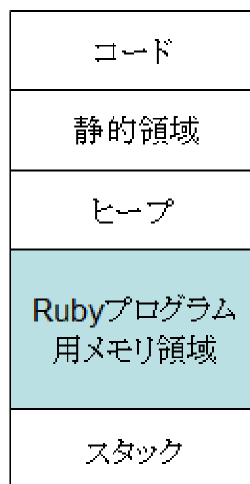

図 2.2: Ruby プログラム実行時のメモリ領域の構成

具体的にどのような情報が Ruby プログラム用メモリ領域に配置されていくかを考える．基本的に Ruby プログラムを読み込み解析された情報はすべてヒープ上に配置される．Ruby プログラムが関数呼び出しを実行した際には，Ruby 自身によってヒープ上に作られたスタックにそれらの情報が置かれる．また Ruby は RVALUE という機構を用いて，オブジェクト管理をしているが，これらの情報もヒープに置かれる．ほぼすべての情報がヒープに置かれるためマシン上のスタックは Ruby 自身が使っている以上に伸びることはない．ただし拡張ライブラリを用いて関数呼び出しなどを行った際には Ruby によって作られたスタックでは

なく，マシン上のスタックに情報が置かれる．これらをまとめた図が図 2.3 である．

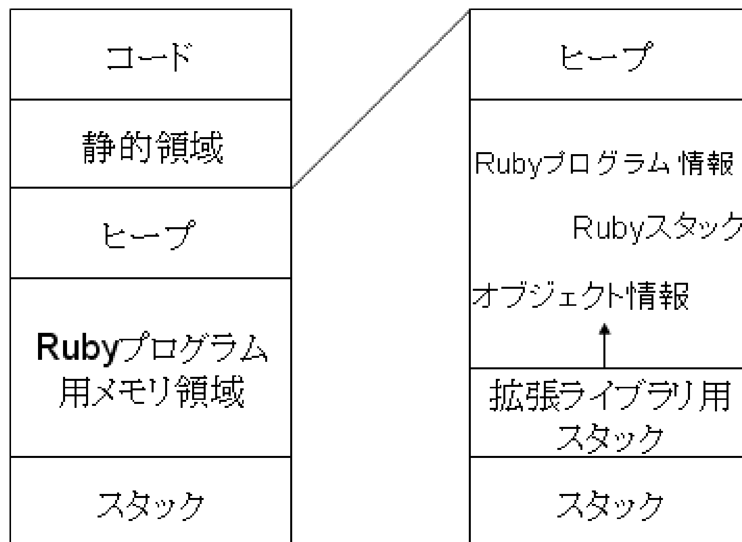

図 2.3: Ruby プログラム実行時のメモリ領域の構成 (詳細)

2.2 ヒープ領域管理

一般にヒープ領域はプログラムで明示的に削除されない限りいつまでも生存できるデータのために使用する記憶域である．局所変数は，関数が終了するとアクセスできなくなるが，ヒープ上に配置されたデータは関数とは無関係に生存することができる．例えば C 言語の malloc で確保された領域や Java の new で確保されたオブジェクトがそれにあたる．これらの領域は，その領域を示すポインタを用いることで関数から関数へと引き渡すことができる．したがって，その領域は，それを作成した手続きに関係なく，生存できる．

2.2.1 メモリ管理ルーチン

メモリ管理ルーチンはヒープ領域におけるすべての使用可能な領域を管理しており，その基本的な機能は次の 2 つがある．

- 割付け
  プログラムからメモリ領域の使用要求を受けると，メモリ管理ルーチンは要求された連続するヒープメモリを確保する．このとき出来る限りヒープ中の空き領域を用いて，確保を行うが，ヒープ中に空き領域が存在しない場合はオペレーティングシステムが管理する仮想メモリから領域を獲得しヒープ空間を拡大する．

- 解放
  メモリ管理ルーチンは，メモリ空間が解放されると，それを自由空間のプールに戻す．戻された空間は，他の割付け要求に応えるために再利用される．プール領域が増大し，

プログラムが使用しているヒープ使用量が減少したとしても，メモリ管理ルーチンがそれらの領域をオペレーティングシステムに返却することは，基本的にしない．

次の条件が満たされると，メモリ管理は非常に容易になる．

- どの割付け要求も同じ大きさのメモリしか必要としない
- 領域の解放が予測できる

Ruby においてこの条件を満たすために，RVALUE と呼ばれる 5word のオブジェクトを用意し管理している．だが，RVALUE を設けても，通常これら 2 つの条件を満たすことは考えられない．オブジェクトの大きさは異なるのが普通であり，オブジェクトの生存期間を予想するのは非常に難しい．Ruby の RVALUE も，5word と書いてはいるが，それ以上の大きさのオブジェクトが必要となった際には，別領域に必要な大きさの領域を確保している．

したがって，メモリ管理ルーチンは任意の大きさの領域を任意のタイミングで割付け及び解放できるようにする必要がある．

メモリ管理ルーチンに望まれる諸性質は次の通りである．

- 空間効率
  メモリ管理ルーチンは，プログラムの実行に必要なヒープ空間全体が最小になるようにしなければならない．これは割付け，解放を繰り返し行うことによって発生する断片化を最小にすることで達成される．
- プログラム実行効率
  プログラムの実行時間は，オブジェクトがメモリ上のどこに配置されているかで大きく変化する．データには局所性と呼ばれる性質が見られるため，この性質を活かしたメモリ配置をすることが望ましい．局所性とは，メモリアクセスを観察したときに，メモリがある一定の範囲で非ランダムに集中してアクセスされる性質をいう．
- 低オーバーヘッド
  メモリの割付けと解放はプログラム中頻繁に行われるため，それらの操作を出来る限り効率良く実行するこが重要となる．そのため割付け，解放にかかるオーバーヘッドは最小にすることが望まれる．

2.3 ガーベージコレクション (GC)

2.3.1 GC の基礎

参照できないデータは一般にゴミとして知られている．手動によるメモリ管理の負担を無くすために，高水準プログラミング言語の多くは到達不能なデータをプログラマに代わって解放してくれる GC と呼ばれる機構を用意している．この節では参照できないデータをプログラム上で判断する手法を説明する．

設計方針

GC はプログラムでアクセスできなくなったオブジェクトを対象として，それが使用していた領域を再利用することである．オブジェクトは型を持ち，実行時にその型がわかるもの

と仮定する．その型情報からオブジェクトがどれくらいの大きさなのか，そこに含まれるどの要素が他のオブジェクトを参照するかがわかるものとする．またオブジェクトへの参照には，そのオブジェクトの先頭アドレスを使用し，オブジェクトの途中の場所を指すポインタはないものとする．したがって，1 つのオブジェクトへの参照はどれも同じ値を持ち，簡単に識別ができるものとする．ただし，Ruby では拡張ライブラリを用いて作成されたオブジェクトについては，参照を完全に知ることが不可能であるため，上記の条件を満たさないが，ここではまず一般的な GC の説明をするため，拡張ライブラリについては考えないものとする．

ヒープ中に存在するオブジェクト集団に対して書き換えを行うユーザープログラムをミューテータと呼ぶ．ミューテータはメモリ管理ルーチンからメモリ領域を獲得してきて，オブジェクトを生成したり，変更を加えたり，削除させるものとする．これらの操作をミューテーションと呼び，ミューテータから到達できないオブジェクトがゴミとなる．

型の安全性について考える．世の中に存在するすべての言語が GC の対象となるわけではない．GC のルーチンが正しく働くためには，与えられたデータの要素がオブジェクトなのかポインタなのかがわかる必要がある．どんなデータに関しても型がわかることをその言語は型安全であるという．型安全な言語には静的なもので ML，動的なものでは Java がある．不安全な言語には C や C++，Ruby などがある．これらの言語は GC を機械的にすべて行うことは不可能である．なぜなら不安全な言語では任意のポインタに対して算術演算を行って，新しいポインタを生み出したり，任意の整数をポインタとすることができるためである．このような状況下において，ある値が整数なのか，ポインタなのかを判断することができないため，不安全と言える．

ただし，実際には使用することができない，適当なポインタをプログラムで生成することはないため，不安全な言語だったとしても，正しく動く GC が開発されている．[5]

性能評価尺度

GC はコストが高くつく場合が多いため，何十年も前に提案され，メモリリークを完全に回避できるにも関わらず多くの主要なプログラミング言語で採用が見送られてきた．長年研究されているが，最良のアルゴリズムは未だに開発されていない．ここでは GC にかかるコストを分析し，評価尺度について説明する．

- 全実行時間
  GC はプログラムの本質的な部分ではないため，全体の実行時間を増加しないことが重要である．しかし GC はほぼすべてのオブジェクトを調べる必要があるため，かなりのコストがかかってしまう場合がある．

- 空間使用効率
  GC によって，メモリの断片化をどの程度防げるかも重要な評価尺度である．これにより使用できるメモリ領域が増加し，またメモリ使用量が少なくできる．

- 中断時間
  単純な GC の場合は，プログラム実行中に突如 GC が起動する．しかも長時間プログラムを停止させてしまうため，評判が悪い．したがってプログラム全体での GC 実行時間を短くするだけではなく，GC 一回にかかる時間も短くする必要がある．実行しているプログラムの性質にもよるが，場合によっては GC をあえて実行しないということも考えられる．

- プログラムの局所性
  GC はゴミを解放するだけではなく，再配置もすることがある．これにより局所性が増し，プログラムの実行時間に影響を与える場合がある．そのため局所性の改善も性能尺度の一つとなる．

これらの評価尺度は互いに相反する場合もあるため，プログラムがどんな振る舞いをするかを考え，それらの兼ね合いを判断しなければならない．また様々な特徴を持つオブジェクトがある場合，GC ルーチンにはそれぞれの特徴に応じた扱いが要求される．

例えば小さいオブジェクトを大量に生成するようなプログラムの場合，割付けにかかるオーバーヘッドを小さくする必要がある．オブジェクトの再配置をするときも，小さいオブジェクトの再配置はそれほどオーバーヘッドかからないが，大きな場合にはオブジェクトの大きさに比例したオーバーヘッドかかる．

その他にも後述するトレース式 GC の場合は GC を実行する間隔をあけることで，回収できるゴミの割合が増えるという性質がある．その理由はオブジェクトには短命の傾向が見受けられるためである．その結果長時間間隔を開けたあとで GC を実行することで，ゴミの回収コストが平均的に小さくすむということになる．しかしそうした場合データの局所性が低下し，中断時間も長くなる．

これに対して参照カウント方式の GC は，ゴミの発生と同時に回収を行うため，中断時間が長くなることはない．しかしミューテータ側での多くの操作を行うため，オーバーヘッドがかさみ，プログラム全体の実行時間を大幅に遅らせる事がある．

到達可能性

プログラムが直接アクセスできるデータのことをルート集合と呼ぶ．ポインタを通して参照されるデータはこの集合から除く．例えば C 言語の場合は，大域変数とスタックに積まれた変数がそれにあたる．Java においては静的なメンバ変数とスタックに積まれた変数がそれにあたる．すなわち，ルート集合の要素はプログラム実行中常に到達可能ということになる．その上で到達可能性は次のように再帰的に定義される．

到達可能性定義

到達可能なオブジェクトのフィールドメンバもしくは，配列の中に格納されている参照によって参照の対象となるオブジェクト自身も到達可能である．

到達可能性はコンパイラによって最適化される場合には少し複雑になる．まず考えられるのが，参照変数がレジスタに保持される場合である．レジスタ上の値もルート集合の一部として考える必要がある．次にプログラムのメモリアドレスについてである．型安全な言語を使った場合，ユーザーはポインタを使わずにコードを書くこととなるが，言語によっては最適化によって実行時はポインタ演算を用いて，レジスタが配列やオブジェクトの途中を指していたりすることがある．このような場合，正しいルート集合や参照を見つけるために，GC はコンパイラから以下の協力が必要となる．

- コンパイラはコード中で隠れ参照が存在しないプログラム点だけに限って，GC を呼び出すように制限できる．

- GC ですべての参照を解決できるようにコンパイラは，オブジェクトに対する，必要な参照情報をすべて伝える必要がある．

- GC が呼び出されたときに，すべての参照可能オブジェクトについて，その基底アドレスを必ず参照できるように，コンパイラは保証することができる．

到達可能オブジェクトの集合は，プログラムの実行とともに変化する．その集合はオブジェクトが生成されると成長し，到達不能になると収縮する．オブジェクトは一度到達不能になると，再度到達可能になることは決してない．ミューテータが到達可能オブジェクトの集合に変更を加える基本的な操作には次の 4 種類がある．

- オブジェクトの割付け
  メモリ管理ルーチンによって行われ，新しく割付けられたメモリへの参照が到達可能オブジェクトの集合に加わる．

- 引数の引渡しと値の返却
  オブジェクトへの参照が実引数によって，対応する仮引数へ渡される操作と，呼び出し側へ結果が返される操作である．それらのオブジェクトによって指されるオブジェクトもまた到達可能となる．

- 参照代入
  u と v を参照として $u = v$ という形の代入を考えると以下の 2 つの効果が考えられる．まず u は v が参照していたオブジェクトへの参照となる．u が参照可能ならば u が参照するオブジェクトもまた参照可能である．次に，u が参照していた参照は消失する．ただし u 以外にも参照が存在する可能性があるため，参照されていたオブジェクトが必ずゴミになるというわけではない．

- 手続きからの戻り
  手続きからでるとき，局所変数を保持していたスタックフレームは消去される．このときオブジェクトへの参照がスタックフレームにしか存在しなかったものはゴミとなる．さらにそのゴミが参照を持っている場合は，その参照先もゴミとなり，以下同様である．

これらを踏まえると，到達可能なオブジェクトの発見方法には 2 種類の基本的な方法があると考えられる．

1 つ目は上記の動作が行われ，参照を失った際に，ゴミかどうかを判断する方法である．この方法は参照カウント法と呼ばれている．参照カウンタと呼ばれる変数をすべてのオブジェクトに保持させ，参照の追加があったら，カウンタを 1 増加させ，参照の消失があった場合にはカウンタを 1 減少させる．このカウンタが 0 となった場合はゴミと判断する方法である．

2 つ目めの方法は定期的にすべての到達可能なオブジェクトを突き止め，それ以外のオブジェクトをすべて到達不可能なオブジェクトとする方法である．この方法はトレース方式とよばれている．トレース方式のゴミ集めはルート集合に含まれるすべての到達可能なオブジェクトに対して，印をつける．その後そのオブジェクトから参照されているオブジェクトに対しても参照をつけるという動作を繰りかえす．この方式は全ての到達可能なオブジェクトを発見するまで到達不可能なオブジェクトを決定することができないが，到達可能な集合を決定した時点で，到達不可能なオブジェクトをすべて発見することができる．その結果同時に多数の領域が解放される．この方式には様々なアルゴリズムが存在する．

2.3.2 参照カウント GC

前節で軽く触れた参照カウント法と呼ばれる GC について説明する．このアルゴリズムは非常に単純であるが，回収できないゴミが発生してしまうなどの理由により，実用的ではない．

基本的な考え方は，オブジェクトが到達可能から到達不可能へ切り替わったときを捉えて，それをゴミと判断する．オブジェクトの参照カウントがゼロになれば，そのオブジェクトは解放されるのである．参照カウント法を用いる場合，すべてのオブジェクトに参照カウント用フィールド (参照カウンタ) を別途，設ける必要がある．参照カウンタが変更されるのは以下の 5 つの場合である．

- オブジェクトの割付け
  新しく生成されたオブジェクトの参照カウンタを 1 に設定する．
- 引数の引渡し
  手続きに渡される各オブジェクトの参照カウンタを 1 増加させる．
- 参照代入
  $u = v$ という式が存在した場合，v の参照カウンタを 1 増加させる．同時に u が参照していたオブジェクトの参照カウンタを 1 減少させる．
- 関数からの戻り
  関数を終了するとき，その関数内の局所変数が参照しているオブジェクトの参照カウンタを減少させる必要がある．いくつかの局所変数が同じオブジェクトへの参照を保持しているときは，それぞれの参照ごとにそのオブジェクトの参照カウントを 1 ずつ減らす必要がある．
- 到達可能性の推移的消失
  オブジェクトの参照カウントがゼロとなったとき，そのオブジェクトの中から参照しているオブジェクトの参照カウンタも減少させる必要がある．

これらを表現したのが図 2.4 である．

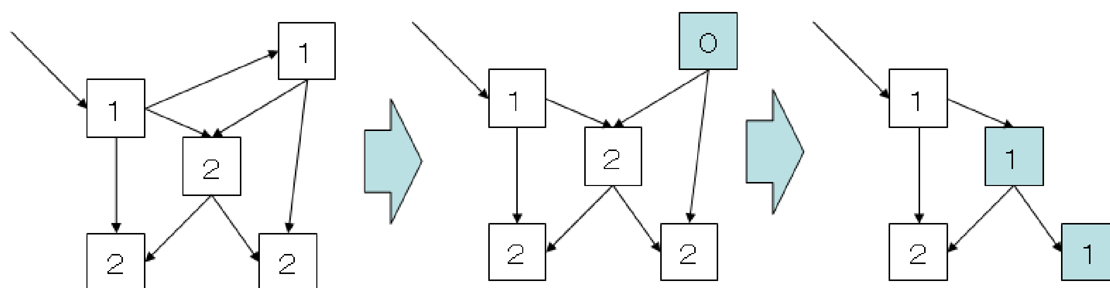

図 2.4: 参照カウント法の例

ただしこの方法には 2 つの欠点がある．1 つは到達不能な循環データ構造は回収できないという事と，コストが高く付くことである．循環データ構造は，親のデータを指したり，相互リストのように一般的に使用される．図 2.5 は 3 つの相互参照を示した図である．

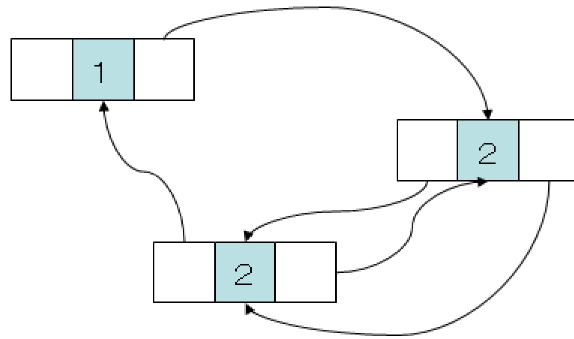

図 2.5: 参照カウント法における回収不可能なゴミの例

これらはルート集合から到達不可能にもかかわらず，参照カウント法では回収されない．このため GC のために参照カウント法を用いると，ゴミが解放されずに，メモリリークとなる．メモリリークとはプログラムが確保したメモリが解放されずに，確保したままになってしまうことを言う．

参照カウント法では，オブジェクトの生成，参照の代入，手続きの開始と終了で毎回余分なオーバーヘッドがかかる．これらのコストは 1 回のコストは少ないとしても，頻繁に起こるため，全体としてのコストは大きくなる．さらに次節で説明するトレース方式の GC がオブジェクトの個数に比例してコストがかかるのに対して，参照カウント方式はオブジェクトの個数と計算コストに比例してコストがかかる．なぜなら，計算を行った場合何かしら参照に変更が加えられることが多いため，計算コストに比例すると考えられる．さらにルート集合の参照の変更が起きた際には，参照の変更がかなりの回数連鎖する可能性があるため，注意が必要である．ただしこの連鎖を防ぐために据え置きカウントという方式も提案されている．これはルート集合からの参照は，参照カウントに含めず，ゴミが発生した際に，ルートに参照がないことを確認して，解放するという方式である．

ここまで参照カウントの欠点を上げたが，利点も存在する．それは GC をインクリメンタルに行えることである．インクリメンタルとは少しずつ進めることができるという意味である．これは GC によってプログラムが長時間停止しないことを意味しており，長時間停止に耐えられない会話型のアプリケーションなどでは重宝される．その他には，ゴミを直ちに解放できるため，メモリ使用率を低く抑えることができるという利点が存在する．

2.3.3 トレース方式 GC

トレース方式の GC は，定期的に到達不能なオブジェクトを見つけ，メモリ空間の再利用を図る．そのため普通は使用可能なヒープ領域を使い果たしてしまったり，ある使用可能なヒープ領域をある一定以上使用したときに実行される．トレース方式の中で最も単純な方法はマークアンドスイープと呼ばれており，これは Ruby でも採用されている方式である．トレース方式にはその他にも様々な方式が存在するため，それらの一部をここで説明する．

マークアンドスイープ

マークアンドスイープのアルゴリズムは非常に単純である．プログラムを一旦すべて停止させ，到達不可能なオブジェクトを見つけ出し，それらを解放するアルゴリズムである．計

算を一旦すべて停止させることをストップザワールドと呼ぶこともある．アルゴリズムの詳細と擬似コードを示す．

マークアンドスイープアルゴリズム

- 入力
  ルート集合の各要素オブジェクト，ヒープ
- 出力
  すべてのゴミを解放した後に得られるフリーリスト
- データ構造
  各オブジェクトに到達/未到達を示すフラグ
  未走査リスト：すでに到達が可能とわかっているが，その中に含まれているオブジェクトの走査していないオブジェクトのリスト
  フリーリスト：到達が不可能なオブジェクトのリスト
- アルゴリズム
  1. ルート集合のオブジェクトを未走査リストに追加
  2. 未走査リストからオブジェクトを１つ取り出す
  3. 取り出したオブジェクトに未到達フラグがセットされていたら，到達フラグをセットし，そのオブジェクトから参照されているオブジェクトを未走査リストに追加
  4. 未走査リストが空になるまで，2. 3. を繰り返す
  5. すべてのオブジェクトに対して 6. を行う
  6. 取り出したオブジェクトに到達フラグがセットされていたら，未到達フラグに変更．未到達フラグがセットされていたらフリーリストに追加

マークアンドスイープアルゴリズム擬似コード

```
/* マークフェイズ */
ルート集合の要素を未走査リストに追加
while 未走査リスト ≠ 0 do
  未走査リストからオブジェクト o を削除
  o に到達フラグをセット
  for all o が参照しているオブジェクト ó do
    if ó が未到達，すなわち到達フラグがセットされていない then
      ó を未走査リストに追加
    end if
  end for
end while

/* スイープフェイズ */
for all プログラム中すべてのオブジェクト o do
  if o が未到達，すなわち到達フラグがセットされていない then
    o をフリーリストに追加
  else
    o の到達フラグをリセット (状態をリセット)
  end if
end for
```

トレース方式アルゴリズムの抽象化

マークアンドスイープアルゴリズムではオブジェクトに注目して，アルゴリズムを説明したが，ここではさらに抽象化を行い，オブジェクトの保存領域 (記憶ブロック) について考える．記憶ブロックとはオブジェクトを保存している記憶装置（レジスタ，メモリなど）の使用領域のことである．トレース方式のアルゴリズムにはさまざまな方式があるが，記憶ブロックは必ず 4 つの状態に分けて考えることが可能である．

- Free
  記憶ブロックが割付けできる状態である．したがって Free の状態にある記憶ブロックに到達可能なオブジェクトが含まれていてはならない．

- Unreached
  到達可能なことが，トレースによってまだ確認されていない状態．到達可能性が認められていない記憶ブロックは GC の間この状態であり続ける．メモリ管理ルーチンによって記憶ブロックが割り付けられるとこの状態となる．1 回の GC が終了した後，到達可能オブジェクトは次回に備えてこの状態に設定し直される．

- Unscanned
  到達可能であるとわかった記憶ブロックは Unscanned か Scanned かのいずれかの状態である．到達可能であっても，そのオブジェクトが持つポインタをまだ調べ終わっていなければ Unscanned の状態である．

- Scanned
  Unscanned 状態の記憶ブロックは，どんな状態でも最終的には Scanned 状態になる．さらに Scanned 状態が参照している記憶ブロックは必ず Unscanned か Scanned となり，Unreeached となることはない．

Unscanned が無くなった時点で到達可能性の計算は終了する．その時点で Scanned となっているブロックが到達可能ブロックである．その他の状態は GC によって，メモリ空間を再利用されることになる．以下に状態の遷移を表した図 2.6 を示す．

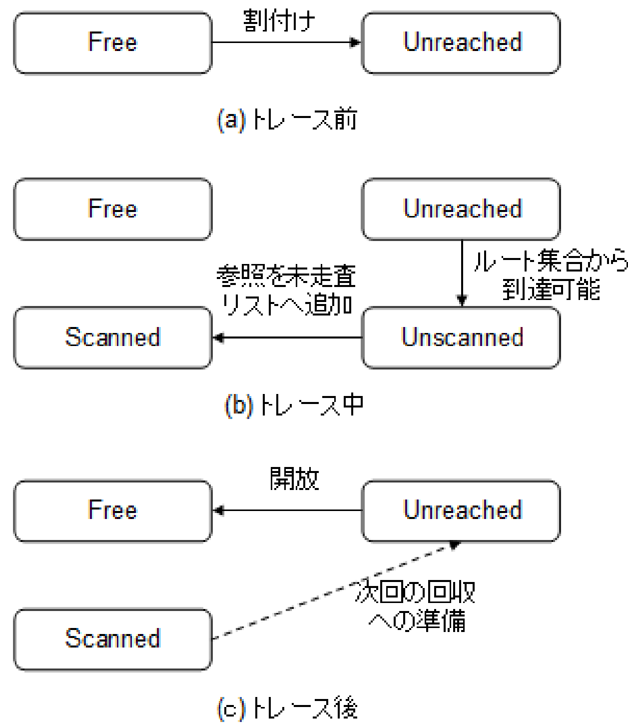

図 2.6: トレース方式における状態遷移

4 つの状態がどのような関係にあるかを考えると，表 2.1 となる．

状態名	Free 変数	Unscanned 変数	到達フラグ
Free	Yes	No	未到達
Unreached	No	No	未到達
Unscanned	No	Yes	到達
Scanned	No	No	到達

表 2.1: トレース方式の状態表現

ここから各オブジェクトに対して 3 つの変数を持つことで実現が可能となることがわかる．

マークアンドスイープの改良

マークアンドスイープアルゴリズムの最終ステップでは，到達不可能なオブジェクトを簡単に見つける方法はなく，ヒープ全体を調べる必要があった．そのため全オブジェクト数に応じたコストがかかってしまう．そのため Baker による改良アルゴリズ [2] では，割付けオブジェクトと到達可能オブジェクトをリストに記憶しておく．するとフリーリストに繋ぐオブジェクトの集合は割付け済みのオブジェクトと到達可能オブジェクトとの差集合で求めることができる．

マークアンドスイープ改良アルゴリズム

- 入力
  ルート集合の各要素オブジェクト，ヒープ
- 出力
  すべてのゴミを解放した後に獲られるフリーリスト
- データ構造
  各オブジェクトに到達/未到達を示すフラグ
  走査リスト：Scanned 状態を持つオブジェクトのリスト
  未走査リスト：Unscanned 状態をもつオブジェクトのリスト
  フリーリスト：Free 状態をもつオブジェクトのリスト
- アルゴリズム
  1. ルート集合のオブジェクトを未走査リストに追加
  2. 未走査リストからオブジェクトを１つ取り出す
  3. 取り出したオブジェクトに未到達フラグがセットされていたら，到達フラグをセットして，走査リストへ追加．また，そのオブジェクトから参照されているオブジェクトを未走査リストに追加．
  4. 未走査リストが空になるまで，2.3. を繰り返す
  5. 全オブジェクトから走査リストの差を求めることで，フリーリストが求められる．

マークアンドスイープ改良アルゴリズム擬似コード

```
/* マークフェイズ */
全体集合 = 全てのオブジェクト
走査リスト = ∅
未走査リスト = ルート集合の要素
while 未走査リスト ≠ ∅ do
  オブジェクト o を Unscanned から Scanned へ変更
  o に到達フラグをセット
  for all o が参照しているオブジェクト ó do
    if ó が未到達，すなわち到達フラグがセットされていない then
      ó を未走査リストに追加
    end if
  end for
end while

/* スイープフェイズ */
フリーリスト = 全体集合 − 走査リスト
```

2.3.4 コンパクション

マークアンドスイープアルゴリズム，マークアンドスイープ改良アルゴリズムともに，記憶ブロックの再配置に関しては考えていない．そのためここでは記憶ブロックの再配置について考える．

再配置 GC は，メモリの断片化を避けるためにヒープ中で到達可能なオブジェクトの移動を行う．一般に到達可能オブジェクトが占めるメモリ空間は，フリーリストが作る空間よりもはるかに小さい．したがって，すべてのオブジェクトの状態を識別して，それらを別個に解放するよりも，到達可能なオブジェクトをヒープのどこかに再配置し，残りをフリーリストのための空間とする方が望ましい．GC アルゴリズムはすべてのオブジェクトの参照を解析するので，その参照を新しい記憶場所へのポインタで置き換えても，それほど手間は掛からない．

到達可能なオブジェクトを連続域へと再配置することで

- メモリ空間の断片化が減少する
- 大きいオブジェクトを置くことが可能となる
- 新しく作成するオブジェクトは，それとほぼ同時に作られるオブジェクトの近くに作成される可能性が高まるため，局所性の改善になる
- 接近した記憶ブロックのオブジェクトどうしが一緒に使用されるときには，プリフェッチによる効果も大きくなる
- 自由空間の管理も簡単になり，大きなブロックで管理が可能となる

以上のような利点が生まれる．

再配置には，順に詰めながら行うものと，再配置用に別領域を確保しておくものが存在する．

- マーク・圧縮方式
  オブジェクトを隙間なく詰めて配置する．そのため，メモリ使用を低く抑えることができる．

- コピー方式
  メモリをあらかじめ 2 つの領域に分割しておく．その上で，オブジェクトを片方の領域からもう片方の領域に移動させる．再配置のために余分な空間を確保しておき，到達可能オブジェクトが見つかるたびに，そこへ移動を行う．

マーク・圧縮方式

マーク・圧縮方式は以下の 3 フェーズからなる．

1. マークフェイズである．前述のマークアンドスイープ方式のマークアルゴリズムと同様.

2. ヒープ中の割付け済み部分を走査し，そこに含まれる各到達可能オブジェクトについて，新しいアドレスを計算する．そのアドレスは，ヒープの低アドレスから割り当てていくため，オブジェクトの間に穴が開く事はない．新しいアドレスは NewLocation と呼ばれるデータ構造の中に記録しておく．

3. オブジェクト中の参照を新しい記憶場所へのポインタに書き換えながら，新しいオブジェクトにコピーする．この処理に必要なアドレスは NewLocation から得られる

マーク・圧縮方式アルゴリズム

- 入力
  オブジェクトのルート集合，ヒープ
- 出力
  新しい free ポインタの値
- データ構造
  未走査リスト，走査リスト
  $free_{ptr}$：割付け可能領域の先頭ポインタ $NewLocation(o)$：$NewLocation$ は以下の機能が満たされていれば，どんなデータ構造でも良い．
  (a) オブジェクト o に新しいアドレスが設定されていなければ，$NewLocation(o)$ はオブジェクト o のために新しいアドレスを設定する
  (b) オブジェクト o に新しいアドレスが設定されていれば，与えられたオブジェクト o に対して，$NewLocation(o)$ の値を得る
- アルゴリズム
  1. マークアンドスイープ方式と同じように走査リストを計算する
  2. $free_{ptr}$ をヒープの先頭で初期化
  3. ヒープ中の記憶ブロックの低アドレスに属するオブジェクト o から順に 4. を実行
  4. $NewLocation(o)$ を実行し，$free_{ptr}$ のアドレスに設定し，$free_{ptr}$ をオブジェクト o の大きさ分進める
  5. ヒープ中の記憶ブロックの低アドレスに属するオブジェクト o から順に 6. を実行
  6. オブジェクト o を再配置し，o に含まれる参照のアドレスを $NewLocation$ によって書き換える
  7. ルート集合はヒープ領域に存在しないため，中の参照が書き換えられていない．そのためそれらの参照を $NewLocation$ によって書き換える

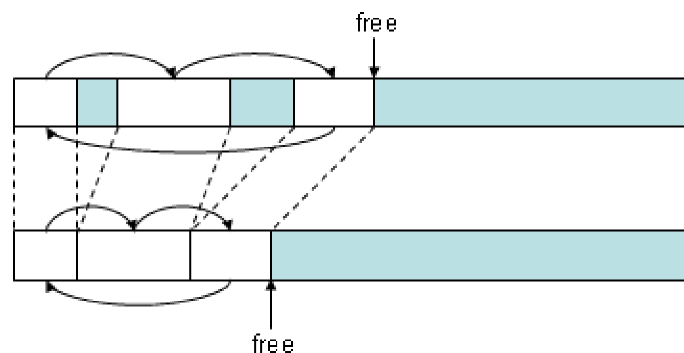

図 2.7: マーク・圧縮方式の概念図

マーク・圧縮方式アルゴリズム擬似コード

```
/* マークフェイズ */
走査リスト = ∅
未走査リスト = ルート集合の要素
while 未走査リスト ≠ ∅ do
  オブジェクト o を未走査リストから走査リストへ移動
  o に到達フラグをセット
  for all o が参照しているオブジェクト ó do
    if ó が未到達，すなわち到達フラグがセットされていない then
      ó を未走査リストに追加
    end if
  end for
end while

/* 新しい記憶場所の記憶 */
free = ヒープ中の先頭アドレス
for all ヒープ中の低アドレスな o から順に do
  if o は到達済み then
    NewLocation(o) = free_ptr
    free_ptr = free_ptr + sizeof(o)
  end if
end for

/* 参照の付け替えとオブジェクトの移動 */
for all ヒープ中の低アドレスな o から順に do
  if o は到達済み then
    for all o.r ∈ o からの参照 do
      o.r = NewLocation(o.r)
    end for
    o を NewLocation(o) の位置にコピー
  end if
end for
for all r ∈ ルート集合の参照 do
  r = NewLocation(r)
end for
```

コピー方式

コピー方式によるアルゴリズムは予めヒープを 2 つに分割しておいた状態で，再配置を行う．この 2 つの空間を From 空間と To 空間と呼ぶこととする．ミューテータは From 空間を使ってオブジェクトの割付けを行う．そして From 空間を使い切ったら，To 空間へ到達可能オブジェクトの再配置を行いながら，コピーする．コピーがすべて終わったら From 空間と To 空間を入れ替える．これが，C.J.Cheney によって考えられたアルゴリズムである．[6]

コピーアルゴリズム

- 入力
  オブジェクトのルート集合，From 空間，To 空間
- 出力
  To 空間における割付け可能ポインタの先頭 $free$
- データ構造
  $free_{ptr}$：割付け可能領域の先頭ポインタ
  $unscanned_{ptr}$：到達可能だが，走査していないオブジェクトの先頭ポインタ
  $NewLocation(o)$：$NewLocation$ は以下の機能が満たされていれば，どんなデータ構造でも良い．
  (a) オブジェクト o に新しいアドレスが設定されていなければ，$NewLocation(o)$ はオブジェクト o のために新しいアドレスを設定する
  (b) オブジェクト o に新しいアドレスが設定されていれば，与えられたオブジェクト o に対して，$NewLocation(o)$ の値を得る
- アルゴリズム
  1. $free_{ptr}$, $unscanned_{ptr}$ を To 空間の先頭ポインタにセットする
  2. ルート集合の各オブジェクト o に対して $NewLocation(o)$ を実行する．実行した結果，To 空間での新しいアドレスが割り当てられる．オブジェクトを To 空間へコピーし，その大きさの分 $free_{ptr}$ を進める
  3. To 空間の $unscanned_{ptr}$ が指す，オブジェクト o の大きさ分 $unscanned_{ptr}$ を進める．o が参照しているオブジェクト $o.r$ が To 空間にコピーされていなければ，Root 集合のオブジェクトと同じように，$NewLocation(o.r)$ を実行し，新しいアドレスを取得する．To 空間へコピーし，その大きさだけ $free_{ptr}$ を進める．
  4. To 空間のポインタが $unscanned_{ptr} = free_{ptr}$ となるまで，3. を繰り返す

コピー方式アルゴリズム擬似コード

```
/* 初期化 */
for all From 空間のすべてのオブジェクト o do
  NewLocation(o) = null
end for
unscanned_ptr = free_ptr =To 空間の先頭アドレス
/* すべてのオブジェクトを To へ移動 */
while unscanned_ptr ≠ free_ptr do
  o = unscanned_ptr が指すオブジェクト
  for all o の中の各参照 o.r do
    o.r = LockupNewLocation(o.r)
  end for
  unscanned_ptr = unscanned_ptr + sizeof(o)
end while

/* オブジェクトがコピー済みならば，コピー先を返却 */
/* そうでなければ，オブジェクトを To 空間にコピーし，free_ptr を進める */
LockupNewLocation(o){
if NewLocation(o) == null then
  NewLocation(o) = free_ptr
  free_ptr = free_ptr + sizeof(o)
  o を To 空間にコピー
end if
return NewLocation(o)
}
```

$NewLocation(o)$ は以下の機能が満たされていれば，どんなデータ構造でも良い．最も現実的なデータ構造はハッシュ法だと考えられる．
(a) オブジェクト o に新しいアドレスが設定されていなければ，$NewLocation(o)$ はオブジェクト o のために新しいアドレスを設定する
(b) オブジェクト o に新しいアドレスが設定されていれば，与えられたオブジェクト o に対して，$NewLocation(o)$ の値を得る．

コストの比較

Cheney のコピー方式アルゴリズムは，到達不可能なオブジェクトには一切触れないという特徴を持つ．一方，到達可能なオブジェクトに対しては，すべて走査し，移動させなければならない．大きなオブジェクトや，何回も GC をくぐり抜けて，生き続けている長寿命オブジェクトがある場合，この処理コストは高くなる．本章で説明してきた 4 種類のアルゴリズムの実行時間は以下のとおりである．ただしルート集合の処理に要するコストは除いてある．

- 基本的なマークアンドスイープアルゴリズム
  ヒープ中の全てのオブジェクトの個数に比例する

- Baker のマークアンドスイープアルゴリズム
  到達可能オブジェクトの個数に比例する
- マーク・圧縮アルゴリズム
  ヒープ中全てのオブジェクトの個数と，到達可能オブジェクト全体の大きさに比例する
- Cheney のコピーアルゴリズム
  到達可能オブジェクト全体の大きさに比例する

2.3.5 不安全な言語に対する保守的な GC

C，C++，Ruby のような不安全な言語で書かれたプログラムにおいて，完全な動作をする GC を作成することは不可能である．ここでいう完全な動作とは，すべてのゴミを回収できるという意味である．ただし，完全な動作をしないため，GC を作成することが不可能というわけではない．完全ではないが，ゴミを回収する GC を作成することはできる．この節では不安全な言語において，どのように GC を実現するかを説明する．

不安全な言語において完全な GC を作成することが不可能な理由は，ポインタに対して，整数値と同じように算術演算を行うことができることに起因している．これによりコンパイラ側では，ポインタと整数値の区別ができない場合がでてくる．区別ができない場合，オブジェクトへの参照かどうかを判断できないため，完全な GC を作成することが不可能になるのである．

この問題に対して，疑わしい値に対しては操作を行わない，という方針をとるのが保守的な GC である．疑わしい値を発見した際に，それが整数値なのにも関わらず，再配置を行い，値を変更すると，計算結果が変わる．これはプログラムとしての機能が失われるため，保守的な GC では操作を行わない．ただし走査を行わない方針をとった場合，メモリリークの原因にはなるかもしれないが，プログラムとしての機能は失われない．このような考えから，不安全な言語では保守的な GC を採用する場合が多く見られる．

2.3.6 その他の GC

この章では基本的な GC に関しての説明をした．GC はその他にも様々なアルゴリズムが存在する．この節ではそれらに関して簡単な説明をする．またここで説明したもの以外にも GC には多様なアルゴリズムが存在する．

インクリメンタル GC

前節で説明している GC は，GC を行っている間，一切の計算を停止状態にしている．GC 時間が長時間にわたると，ユーザープログラムの停止時間も長時間にわたることになる．リアルタイム性が重要となるユーザープログラム (GUI など) では，こういった GC は適していない．その欠点を補うために生まれたのが，インクリメンタル GC である．

インクリメンタル 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，割付け処理を行なうときに，少しづつ GC を進めるというものである．

ただし GC を少しずつ進める場合，ライトバリアとよばれる処理が必要になる場合があ

る．これは GC とユーザープログラムを交互に動かすため，それまで進めた GC の結果がユーザープログラムによって変更される可能性があるため，それを感知する方法である．ライトバリアを行うことで，余計なコストがかかるが，長時間停止はしないため，インクリメンタル GC はリアルタイム性を求めるプログラムでは有益な方法と言える．

世代別 GC

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．この傾向を利用した GC が世代別 GC 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良いため，古いオブジェクトは GC の対象とせず (無条件に生き残るとみなす)，新しいオブジェクトを重点的に探索することによって，GC の効率を上げる方法である．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 (図 2.8)．

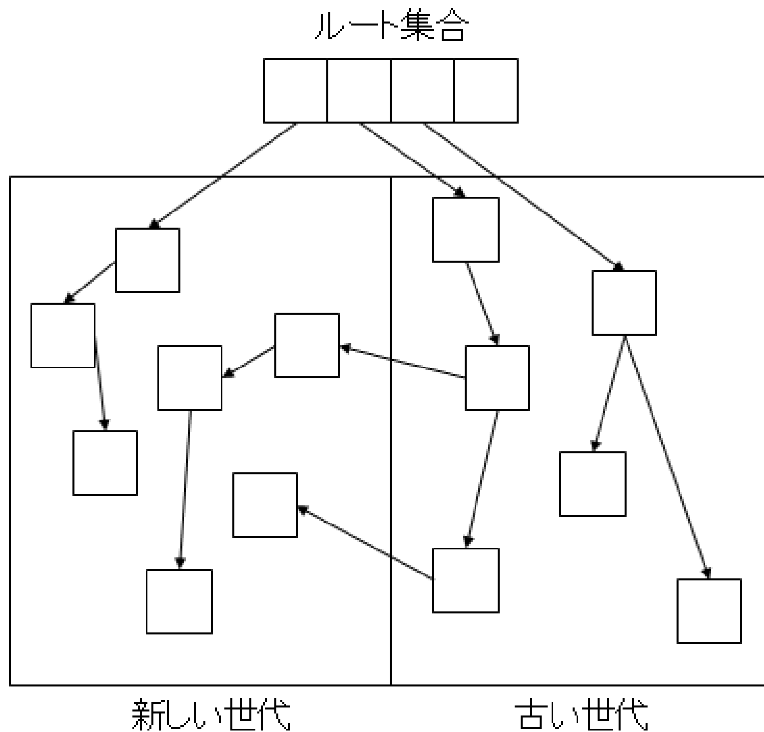

図 2.8: 世代別 GC

トレイン GC

世代別 GC の欠点である古い世代への GC によるプログラム停止時間の短縮化を図った手法である．この手法では，古い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，1 つ以上の車両を連結したものを列車と呼ぶ．1 度の古い世代への GC で 1 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，とい

う方法である．世代別 GC ではまれに全てのオブジェクトに対して GC を行う必要があったが，この GC の場合はそれを防ぐことができる．

第3章　Ruby

3.1　言語

この章では言語としての Ruby に関して簡単な説明を行う．

3.1.1　エントリーポイント

C 言語や Java では main 関数を定義することでプログラムのエントリーポイントを定義したが，Ruby ではそのような関数を定義する必要はない．プログラムのネストレベルが一番低い場所に式を書くと，それが実行される．

print.rb

```
p "first"
p "second".upcase()
```

このようなプログラムを考えた場合，実行すると以下のように表示される．

print.rb の実行結果

```
% ruby hoge.rb
first
SECOND
```

「p」はコンソール上に変数の内容を表示するためのメソッドである．また文の終端にセミコロンが存在しないが，これは Ruby が基本的に改行を文の終端と考えるためである．

3.1.2　オブジェクト

Ruby はオブジェクト指向スクリプト言語であり，動的型付け言語である．そのため C 言語のように int などの型宣言を行わずに変数を使うことが可能である．また Java には int や long のような基本型が存在するが，Ruby は操作できるものは全てオブジェクトであるため，基本型は存在しない．

3.1.3　変数

Ruby では変数や定数は全てオブジェクトへの参照を保持している．そのため別の変数に代入しただけで深いコピー[1]が実行されることはない．言いかえると，変数への代入はポインタの代入ということになる．ただし C 言語のようにそのポインタ自体の値を変更することはできない．

[1]全く同じ内容のオブジェクトをコピー元とはメモリ上で別の場所に作成すること．そのため，コピー先の内容に変更を加えたとしても，コピー元には影響は及ばない．

参照の例

a = ”foo”
b = a
c = b

このようなプログラムを書いたとき変数は図 3.1 のように，”foo”オブジェクトへの参照を保持することとなる．

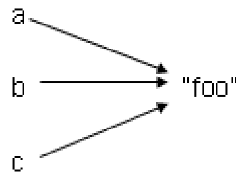

図 3.1: 参照の例

3.1.4 クラス・メソッド

Ruby では，オブジェクトとメソッドを結びつける仕組みとして伝統的なクラスの概念を採用している．すなわち全てのオブジェクトはただ一つのクラスに所属しており，呼び出せるメソッドをクラスで決定する．そのため”second”.upcase() を実行した際の挙動を考えると，”second”が String クラスのインスタンスであるため，Ruby は String クラスから upcase() というメソッドを探索し，実行する．という流れになる．もし対応するメソッドが見つからなかった場合，継承しているクラスのメソッドを探索することとなる．最終的に該当するメソッドが見つからなかった場合は「NoMethodError」というエラーが発生する．

イテレータとブロック

Ruby ではイテレータと呼ばれる特殊なメソッドを利用することで，様々な繰り返し処理を実装できる．イテーレータはブロックと呼ばれる中括弧で囲まれたコードに繰り返し処理の本体を記述し，それをイテレータメソッドに付属することで実現する．

iter.rb

```
a = [1, 2, 3, 4]
b = a.map{|x| x+x}
c = a.select{|x| x % 2 == 0}
```

このコードの場合 `b = [2, 4, 6, 8]` となり，`c = [2, 4]` となる．この他にも each や inject など様々なイテレータメソッドが実装されており，容易に繰り返し処理を実装することができるようになっている．

3.2 実装

今回対象とする Ruby 処理系は公式に配布されている Ruby(MRI，YARV) である．それらの処理系は C 言語で実装が行われているため，この章ではそれに準じた説明を行う．

3.2.1 オブジェクト

Ruby ではオブジェクトを C 言語の構造体で実装し，扱う際には常に VALUE 型とよばれる値を経由して扱う．クラスによって構造体の構成は変わるが，VALUE 型はどのクラスでも同じものを使う．VALUE 型の定義は

VALUE 型

```
typedef unsigned long VALUE;
```

となっている．これを各構造体へのポインタにキャストして使う．そのためポインタのサイズ > unsigned long のサイズが成り立つと Ruby は動かない．

クラスをあらわす構造体にはさまざまな種類が存在し，オブジェクトのクラスによって使い分ける．

struct RClass	クラスオブジェクト
struct RFloat	小数
struct RString	文字列
struct RArray	配列
struct RRegexp	正規表現
struct RHash	ハッシュテーブル
struct RFile	IO、File、Socket など
struct RData	C レベルで定義されたクラス全て
struct RStruct	Ruby の構造体 Struct クラス
struct RBignum	大きな整数
struct RNode	構文木を構成するためのクラス
struct RMatch	正規表現の検索結果
struct RRational	分数
struct RComplex	複素数
struct RObject	上記にあてはまらないもの全て

表 3.1: 構造体の種類

”str”という文字列を表現するオブジェクトは図 3.2 のようになる．

図 3.2: 文字列オブジェクト

本稿では特に RString 構造体と RArray 構造体に関して提案を行うので，それらに関して詳しく見る．

RString の定義

```
struct RString {
  struct RBasic basic;
  union {
    struct {
      long len;
      char *ptr;
      union {
        long capa;
        VALUE shared;
      } aux;
    } heap;
    char ary[RSTRING_EMBED_LEN_MAX + 1];
  } as;
};
```

RArray の定義

```
struct RArray {
  struct RBasic basic;
  union {
    struct {
      long len;
      union {
        long capa;
        VALUE shared;
      } aux;
      VALUE *ptr;
    } heap;
    VALUE ary[RARRAY_EMBED_LEN_MAX + 1];
  } as;
};
```

Ruby は上記 2 つの構造体に限らず，すべてのオブジェクトを表す構造体において，RBasic 構造体の変数 basic をメンバとして持つ（図 3.3）．この結果，VALUE がどのような構造体へのポインタだったとしても，RBasic*でキャストすれば basic メンバへアクセスできる．

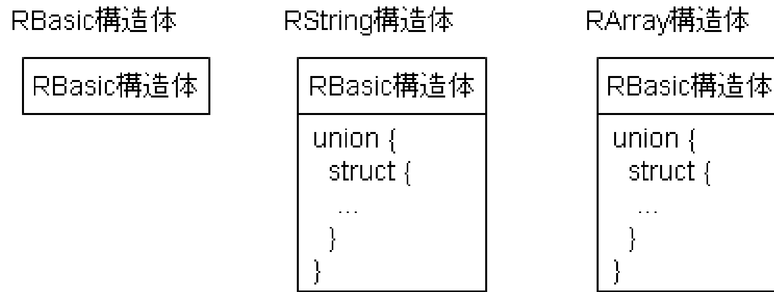

図 3.3: 構造体レイアウト

また RBasic 構造体は

RBasic の定義

```
struct RBasic {
  unsigned long flags;
  VALUE klass;
};
```

このような構造になっている．flags は読んで字のごとく，さまざまなフラグが格納されている．例を挙げると GC のマークフラグなどである．klass はオブジェクトが属するクラス (RClass 構造体) へのポインタを格納している．これによりオブジェクトがどのクラスに属しており，さらにそのクラスがどのようなメソッドを持っているかがわかるようになるため，メソッドを実行することができるようになる．

RString 構造体のメンバについて詳細を述べる．Rstring 構造体は大きく分けて 2 つのメンバを持つ．heap 構造体と ary 配列である．基本的には heap 構造体が使われることになるが，文字列の長さが RSTRING_EMBED_LEN_MAX 以下の場合 ary 配列が使われることになる．

heap 構造体は以下の要素を持つ．

- `long len`
- `char *ptr`
- `long capa`
- `VALUE shared`

`long len` は文字列の長さを保存しており，実際の保存場所は `char *ptr` で指された場所となる．`long capa` は `char *ptr` で指された場所の大きさである．これは文字列オブジェクトの内容が変更された際に活用される．変更された際，`long capa` 以下ならば `char *ptr` の内容を書き換えることになる．Java の場合文字列オブジェクトはイミュータブルであるため，内容が書き換わるたびにオブジェクトを新しく作っていた．しかし Ruby の場合は違い，オブジェクトを再利用する．また同じ内容のオブジェクトを何回も作成するとメモリ領域が無駄になるため，それを防ぐために `VALUE shared` が存在する．同じ内容だった場合は,`VALUE shared` に同じ文字列を保存している RString 構造体へのポインタを保存し，メモ

リ領域の削減を行う．

最後に ary 配列であるが，これは文字列の内容を RString 構造体に埋め込むことで，参照を減らし処理が高速になるようにしたものである．

RArray 構造体もほぼ同様の内容である．ただし文字列の場合は保存するものが文字と決まっていたが，配列の場合は中にさまざまなオブジェクトが入る可能性があるため，VALUE 型のポインタを保持している．

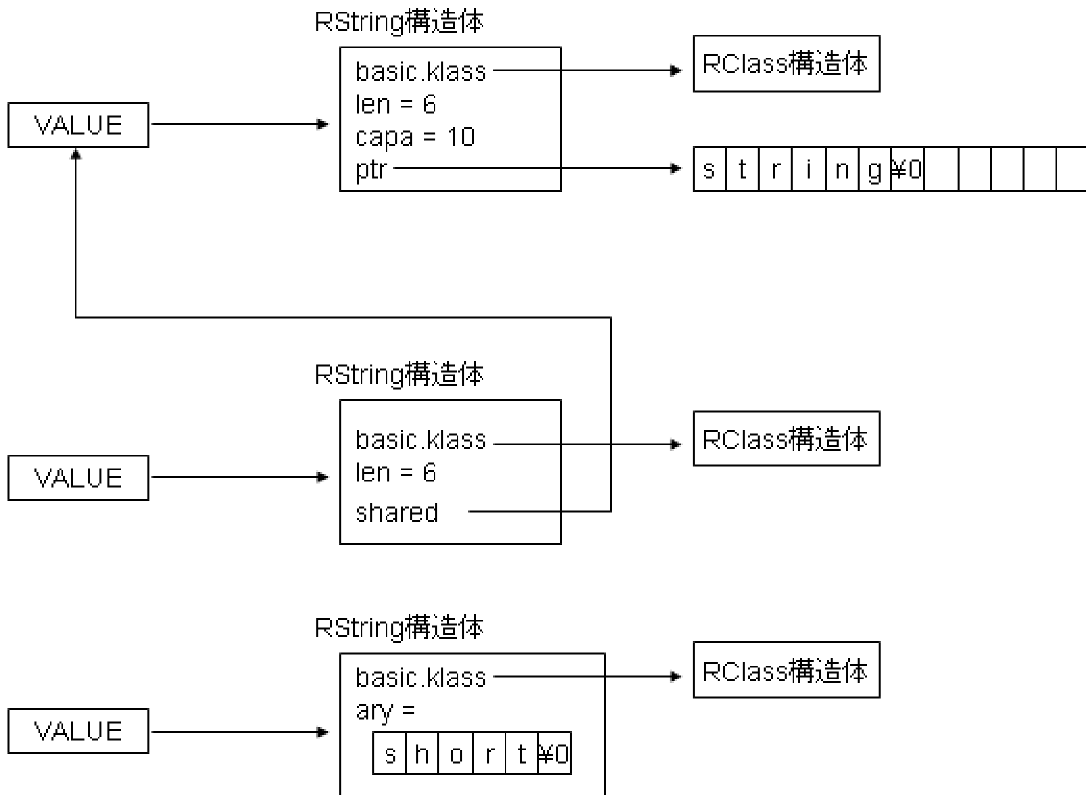

図 3.4: 文字列オブジェクトの例

3.2.2 VALUE 埋め込みオブジェクト

VALUE はポインタ的役割のほかに複数の役割を持っている．そういった理由から void* ではなく，unsigned long で定義されている．ポインタ以外の役割には以下の 6 つがある．

- 小さな整数
- シンボル
- true
- false
- nil
- Qundef

小さい整数

Ruby はすべてのデータがオブジェクトで表現されるため，整数も例外なくオブジェクトとなる．しかし整数は for 文など様々な箇所で繰り返し用いられることが多いため，毎回オブジェクトを生成していては，実行速度が遅くなることがある．そのため小さな整数に関してはオブジェクトを生成せずに直接扱うことにしている．小さな整数とは sizeof(VALUE)*8-1 ビットに収まる符号付き整数を指す．これにより for 文などで繰り返し何かを実行する際に，無駄なオブジェクト生成をせずにすむ事になる．

シンボル

シンボルとは文字列と一対一に対応する整数のことである．プログラミングでは様々な文字列を処理する必要がある，それらをすべて文字列で扱っていては速度が遅くなる．そのためそれらをシンボルという整数に変換することで，高速化している．同時にメモリ管理も行わずにすむようになる．一般的にはハッシュテーブルのキーとしてシンボルが使われることが多い．

true，false

true，false はそれぞれ C 言語レベルで

true，false

```
#define Qfalse 0
#define Qtrue  2
```

と定義されている．true，false はともに Ruby 上ではオブジェクトとして扱われているが，C 言語上では数値として扱われていることを意味する．そのため構造体は使われずに，VALUE で表現している．

nil

nil はオブジェクトが存在しないことを示すオブジェクトである．これは C 言語レベルで

nil

```
#define Qnil 4
```

と定義されている．

Qundef

Qundef はインタプリタ内部で，値が未定義であることを表すために使われる．そのため実際に Ruby でプログラミングを行う際には使うことはできない．

3.2.3 オブジェクト管理

GC を行う上で，オブジェクトの管理は非常に重要である．そのためこの章では Ruby がどのようにオブジェクトを管理しているかを説明する．

RVALUE 構造体

オブジェクトの実体は構造体である．そのためオブジェクトを管理することは，構造体を管理することと等しい．本来ならば，VALUE 埋め込みオブジェクトも考える必要があるが，本質ではないため，ここでは割愛する．構造体の大きさは型によって違うため，管理が面倒である．そのため Ruby では RVALUE という構造体を共用体を介して作成し，管理を楽にしている．RVALUE の定義は以下のようになっている．

RVALUE 構造体

```
typedef struct RVALUE {
  union {
    struct {
      unsigned long flags;
      struct RVALUE *next;
    } free;
    struct RBasic    basic;
    struct RObject   object;
    struct RClass    klass;
    struct RFloat    flonum;
    struct RString   string;
    struct RArray    array;
    struct RRegexp   regexp;
    struct RHash     hash;
    struct RData     data;
    struct RStruct   rstruct;
    struct RBignum   bignum;
    struct RFile     file;
    struct RNode     node;
    struct RMatch    match;
    struct RRational rational;
    struct RComplex  complex;
  } as;
} RVALUE;
```

共用体の最初の要素である free 構造体について説明する．これは RVALUE 構造体が使われてないときに使われる構造体である．そしてその時 `as.free.flags` は 0 となる．RVALUE 構造体が使われているとき，RVALUE には free 構造体以外の何かの構造体を持つこととなる．このとき全ての構造体は必ず RBasic 構造体を持つことが約束されている．そして RBasic 構造体は `flags` を要素に持つため，`as.free.flags` は `as.basic.flags` と同値ということになる．`as.basic.flags` には型情報が保存されているため，free 構造体を共用体が持っていた場合，`as.basic.flags` が 0 になることはない．そのため `as.basic.flags` が 0 のときは必ずその RVALUE 構造体が使われていないということになる．

RVALUE 構造体の管理

複数のオブジェクトを管理するために heaps_slot 構造体が用意されている．この構造体は複数のオブジェクトを管理するうえで基本となる構造体である．

heaps_slot 構造体

```
struct heaps_slot {
  void *membase;
  RVALUE *slot;
  int limit;
};
```

membase には slot の先頭のアドレスが代入される．このアドレスを用いることで GC を行う際に，heaps_slot 構造体によって管理されているオブジェクトかどうかの判定を行う．RVALUE の保持は slot 部分で行う．このとき slot の大きさは固定長（デフォルトでは 16KB）となっている．また limit は 1 つのヒープスロットあたりに入る RVALUE の個数である (図 3.5).

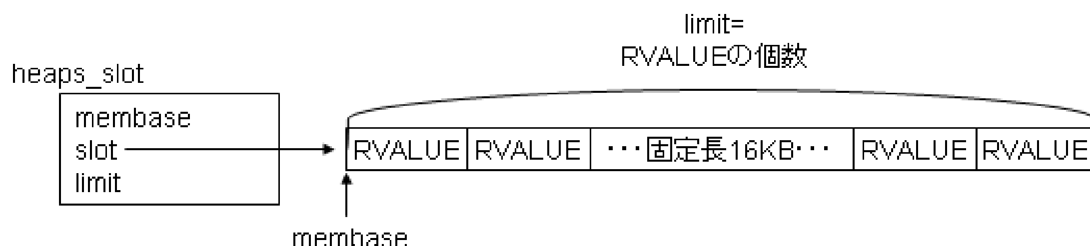

図 3.5: heaps_slot 構造体

slot は固定長であるため，heap_slot では限られた数しかオブジェクトを管理することができない．そのため heaps_slot 構造体をさらにリスト構造にすることで，オブジェクトを管理することになる．これを行うのが，rb_objspace 構造体である．rb_objspace 構造体はオブジェクトの管理だけでなく，GC 情報の管理なども行う．そのためここではオブジェクトの管理を行う部分のみ説明する．

rb_objspace 構造体

```
typedef struct rb_objspace {
  struct {
    size_t limit;
    size_t increase;
  } malloc_params;
  struct {                    ... (A)
    size_t increment;
    struct heap_slot *ptr;
    size_t length;
    size_t used;
    RVALUE *freelist;
    RVALUE *range[2];
    RVALUE *freed;
  } heaps;
  struct {
    int dont_gc;
    int during_gc;
  } flags;
  struct {
    st_table *table;
    RVALUE *deferred;
  } final;
  struct {
    VALUE buffer[MARK_STACK_MAX];
    VALUE *ptr;
    int overflow;
  } markstack;
  struct {
    int run;
    gc_profile_record *record;
    size_t count;
    size_t size;
    double invoke_time;
  } profile;
  struct gc_list *global_list;
  unsigned int count;
  int gc_stress;
} rb_objspace_t;
```

オブジェクトの管理を行っているのは (A) の heap 構造体部分であり，その概念図が，図 3.6 である．heaps とは (A) の heap 構造体のメンバである ptr ポインタ部分である．生成可能なオブジェクト数は，heaps のエントリ数に図 3.5 の limit をかけ合わせたものになる．それ以上に生成を行いたい場合は，heap_slot を新たに生成し，heaps エントリにそれを追加する

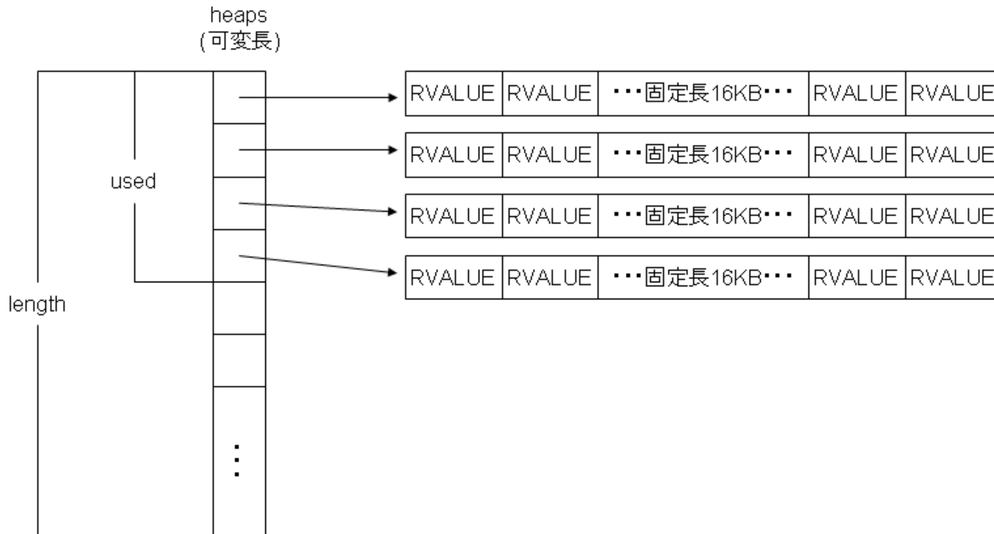

図 3.6: rb_objspace 構造体概念図

ことで生成を行う．

外部領域

RVALUE 構造体は型の違いによる差異を吸収している．さらに共用体 as を用いることで無駄なメモリ領域を使わずに，固定長の大きさを保つことができている．これらの特徴により，Ruby ではオブジェクト管理を heaps や heap_slot という単純な構造で実現できている．しかし固定長とした場合，それ以上の大きさのオブジェクトを扱うことが難しくなるという欠点が存在する．Ruby ではその欠点を外部に別領域を保持することで解決している．3.2.1 章の RString 型や RArray 型などの定義で ptr となっている部分がそれである．これを考慮した図が図 3.7 である．外部領域は RVALUE とは違い内容によって大きさが変わる．

C 言語拡張ライブラリ

Ruby では C 言語を用いてライブラリを書き，それを Ruby 上で使用するということが可能である．さらに C 言語で書かれたライブラリの記述を容易にするために，ライブラリ内で自由に Ruby オブジェクトの作成・変更・削除をできるようにしている．ただしこれらの行為は Ruby 側から把握することができない．そのため拡張ライブラリ内で作られたオブジェクトは RVALUE による管理を受けないことになる．これは Ruby 処理系が全てのオブジェクトを管理できない事を意味する．そのため Ruby 処理系はメモリ中で数値を発見した際，それが数値なのかポインタなのかを判断する必要がでてくる．この判断を間違えると，2.3.5 節で説明したように，プログラムとして不適切な動作となってしまう．そのため Ruby では保守的な GC を採用している．

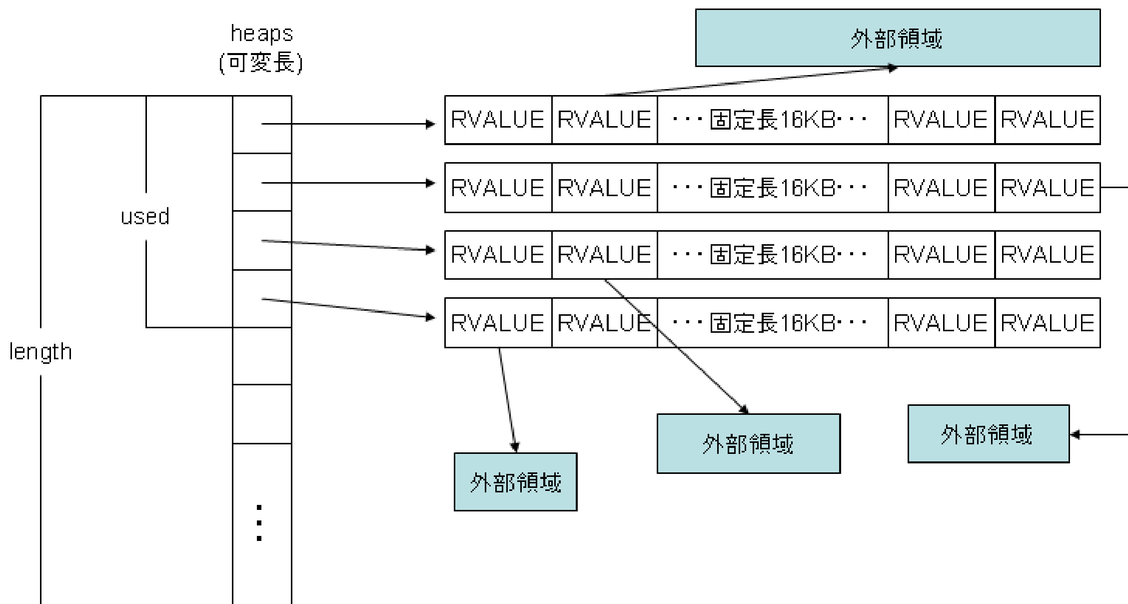

図 3.7: 外部領域を考慮した rb_objspace 構造体概念図

3.3 Ruby における GC

Ruby では C 言語を利用して拡張ライブラリを作成することができる．このため保守的なマークアンドスイープ GC を採用している．この章ではそれを踏まえてどのように GC が実装されているかを説明する．

3.3.1 マーク

到達可能なオブジェクトにマークをする際に，Ruby では RBasic 構造体の要素である flags を利用する．この中に FL_MARK フラグと呼ばれるフラグが存在するため，それにチェックを入れる．そして，到達可能なオブジェクトから参照可能なオブジェクトも到達可能であるため，そこからそれらのオブジェクトの VALUE 全てに対してマークを再帰的に実行すればよい．

gc_mark

```
static void
gc_mark(rb_objspace_t *objspace, VALUE ptr, int lev) {
  register RVALUE *obj;

  obj = RANY(ptr);
  if (rb_special_const_p(ptr)) return;
  if (obj->as.basic.flags == 0) return;
  if (obj->as.basic.flags & FL_MARK) return;  /* already marked */
  obj->as.basic.flags |= FL_MARK;

  if (lev > GC_LEVEL_MAX || (lev == 0 && stack_check())) {
    if (!mark_stack_overflow) {
      if (mark_stack_ptr - mark_stack < MARK_STACK_MAX) {
        *mark_stack_ptr = ptr;
        mark_stack_ptr++;
      }
      else {
        mark_stack_overflow = 1;
      }
    }
    return;
  }
  gc_mark_children(objspace, ptr, lev+1);
}
```

gc_mark はオブジェクトに対してマークを行う関数である．最初の 3 つの if 文でマークが必要かどうかを判断している．ここでマークの必要がないと判断されれば，再帰を用いたマークは行われない．そうでなかった場合は `obj->as.basic.flags |= FL_MARK;` によってオブジェクトにマークされる．次にマークされたオブジェクトから参照されているオブジェクトもマークする必要があるため，gc_mark_children 関数によってマークを行う．gc_mark_children 関数の説明は次に行う．

`if (lev > GC_LEVEL_MAX || (lev == 0 && stack_check())) {` から始まる部分が存在するが，これは再帰によってスタックオーバーフローが起こらないようにしている部分である．この説明はマーク作業の本質ではないため割愛する．

gc_mark_children

```
static void gc_mark_children(*objspace, ptr, lev)
  rb_objspace_t *objspace; VALUE ptr; int lev;
  {
  register RVALUE *obj = RANY(ptr);
  goto marking;
  ...
again:
  obj = RANY(ptr);
  if (rb_special_const_p(ptr)) return;
  if (obj->as.basic.flags == 0) return;  /* free cell */
  if (obj->as.basic.flags & FL_MARK) return;  /* already marked */
  obj->as.basic.flags |= FL_MARK;
  ...
  switch (BUILTIN_TYPE(obj)) {
  ....
marking:
  case RSTRING:
    if (FL_TEST(obj, RSTRING_NOEMBED)             ··· (A)
        && FL_ANY(obj, ELTS_SHARED|STR_ASSOC)) {
      ptr = obj->as.string.as.heap.aux.shared;
      goto again;
    }
    break;
  case RARRAY:
    if (FL_TEST(obj, ELTS_SHARED)) {
      ptr = obj->as.array.as.heap.aux.shared;
      goto again;
    }
    else {
      long i, len = RARRAY_LEN(obj);
      VALUE *ptr = RARRAY_PTR(obj);
      for (i=0; i < len; i++) {
        gc_mark(objspace, *ptr++, lev);
      }
    }
    break;
  case RDATA:
    if (obj->as.data.dmark) (*obj->as.data.dmark)(DATA_PTR(obj));
    break;
  ...
}
```

このコードは説明のために，実際のソースコードに一部変更を加えてある．

`gc_mark_children` 関数は goto 文を使うことで，一部再帰を簡略化している．そのため，関数に入った際にはまず marking 部分から実行されるようになっている．marking 部分では，`switch(BUILTIN_TYPE(obj)` でオブジェクトの型を判断し，それに応じた処理を行っている．ここでは本稿において重要な 3 つの型に関して説明を行う．

RString 構造体は 3.2.1 節で説明したように，大きく分けて 2 つの構造を持つことになる．1 つ目は文字列本体が RVALUE とは違う外部領域に保存されている場合である．これは `(A)` 部分が `false` の時である．この場合，このオブジェクトは他のオブジェクトへ参照を持たないため，再帰は中止される．2 つ目は shared メンバによって，他の RString 構造体とオブジェクトを共有している場合である．これは `(A)` 部分が `true` の時である．この場合，共有は参照とみなすことができるため，実装では `ptr` をマーク対象オブジェクトとし，goto 文を使って，再度マークを行っている．

次に RArray 構造体の場合である．この構造は RString 構造体と似ているため，処理の考え方は同じになる．しかし配列の場合，配列の要素にオブジェクトが格納されているため，それらもマークの対象とする必要がある．それが for 文によって行われている部分である．

最後に RDATA 構造体の場合である．この構造体は拡張ライブラリで作成したクラスを表す構造体である．拡張ライブラリを用いた場合，内部でどのような処理を行っているかは関与できないため，ここでは `dmark` という関数ポインタを使って処理を行っている．`dmark` はライブラリ作成者が Ruby にマークすべきオブジェクトを教えるための関数ポインタである．そのため拡張ライブラリにおいてマーク作業はライブラリ作成者が実装することになる．

3.3.2 ルート集合

Ruby におけるルート集合は以下のとおりである．

- CPU レジスタ
- マシンスタック
- レジスタウィンドウ
- Ruby スタック
- グローバル変数
- その他

その他には，Ruby の END 文などで登録したプログラムの終了時に実行される手続きオブジェクトなどが含まれる．マーク作業はこれらルート集合の要素から行われる．ただしその詳しい手順に関しては省略する．

3.3.3 スイープ

スイープはマークされていないオブジェクトを回収すれば終了する．しかし 2 点注意する必要がある．1 つは構文木を構成するノードの存在，もう 1 つはファイナライザーである．これらについて簡単に説明を行う．

構文木ノード

Ruby は yacc というソフトを使って構文解析を行っている．そのためソースコードの構文解析中は yacc が用意したスタックで解析が行われる．このスタックは環境によってはマシンスタック上に置かれない．マシンスタック上にあるならば，ルート集合とみなされるため，解析情報はすべてマークされる．しかしそうでない場合，マークされない可能性がでてきてしまう．マークされなかった場合，解析が正しく行われない可能性があるため，スイープ時にはそれを考慮し，解析中に限り，解析情報を全てマークし，その後スイープを行う．

ファイナライザー

Ruby ではオブジェクトが回収される際に，フックをかけることができる．このフックをファイナライザーと呼ぶ．ファイナライザーが定義されている場合，それを呼び出す必要があるため，回収の際にはファイナライザーが定義されているかをチェックし，回収を行う．ただしファイナライザーが定義されていたとしても，ファイナライザーを定義したオブジェクトは既に死んでいるため，そのオブジェクトを使うことはできない．そのため死んだオブジェクトに依存しない処理を書くことになる (例：死んだオブジェクトの個数をカウントする)．ファイナライザーが定義されているかどうかは，`obj->as.basic.flags` の `FL_FINALIZE` フラグを使って判定している．

第4章　予備調査

詳しい手法は5章で説明するが，本稿では文字列オブジェクトと配列オブジェクトを対象とし，リニアアロケーション及びコンパクションを行うことでメモリ管理の改善を図る．この方針が有効かどうかを調べるために，以下の予備調査を行った．

- Rails アプリケーションにおけるオブジェクトの割合
- RDoc におけるオブジェクトの割合
- Rails アプリケーションにおける文字列及び配列オブジェクトの詳細
- RDoc における文字列及び配列オブジェクトの詳細

1つ目，2つ目の実験で全体に対してどの程度文字列及び配列オブジェクトが存在するかを確認する．3つ目，4つ目の実験で，具体的にどの程度改善対象とすることができるかを把握する．
実験環境は以下の通りである．

CPU	Intel Corei7 920
OS	Ubuntu9.10(Linux 2.6.31)
メモリ	3GB
Ruby	1.9.1 p243

グラフ上に出てくる用語は以下の意味で用いている．

- NODE：構文木を構成するクラス
- STRING：文字列クラス
- ARRAY：配列クラス
- DATA：C 言語で作られた拡張ライブラリ用クラス
- ETC：その他すべてのクラス
- live：生きているオブジェクト
- dead：死んでいるオブジェクト
- small：固定長部分のみのオブジェクト
- large：固定長＋可変長構造をしているオブジェクト

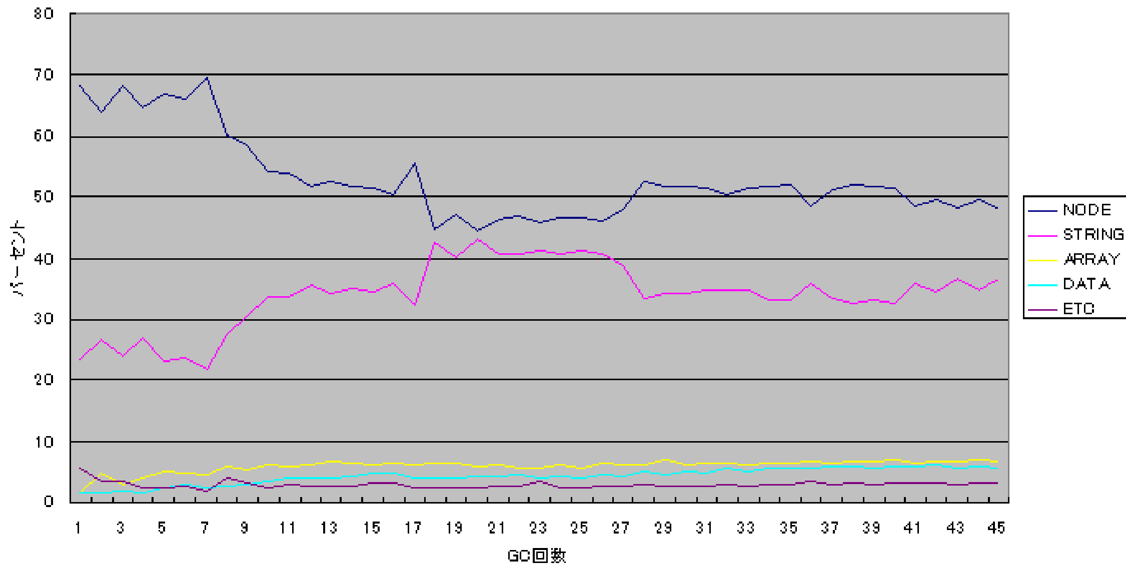

図 4.1: Rails アプリケーション実行中におけるオブジェクトの割合

4.1 Rails アプリケーションにおけるオブジェクトの割合

GC が発生した際に，オブジェクトがどのクラスに属するかを調査した．実験では Ruby が普及した要因の 1 つでもある Ruby on Rails を用いた．具体的には Ruby on Rails を用いて，簡単な掲示板システムを構築し，ランダムな文字列を 15 回書き込むことで調査を行った．結果を図 4.1 に示す．

GC 回数が 40 回目まではサーバー起動時における GC であり，それ以降がリクエスト応答時の GC である．横軸は GC 回数であり，時間ではないことに注意擦る必要がある．この実験の場合では，サーバー起動に約 4 秒，書き込みは 1 回の書き込みを 2 分程度の間隔で行っているため，30 分ほど使用している．本稿で対象とする文字列及び配列クラスは全体の約 40%であることが確認できる．

4.2 RDoc におけるオブジェクトの割合

RDoc とは，Ruby で書かれたソースコードを解析し，それに基づきドキュメントを作成するツールである．Javadoc の Ruby 版である．今回の実験では，Ruby に標準で備わっているライブラリのソースコードに対して RDoc を実行し，メモリ状況を調査した．結果を図 4.2 に示す．Rails は「サーバーの起動」と「クライアントへの応答」と処理を明確に分けることができたが，RDoc は全ての処理で 1 つのアプリケーションとして成り立っているため分割できない．そのため RDoc では処理を分割して調査を行っていない．

この場合 Rails アプリケーションとは違い，その他のオブジェクトが大量に生成されていることがわかる．しかしこれらのオブジェクトは本稿で提案する手法を適用することができないオブジェクトであるため考えないこととする．文字列と配列オブジェクトに関しては全体の約 80%であることが確認できる．

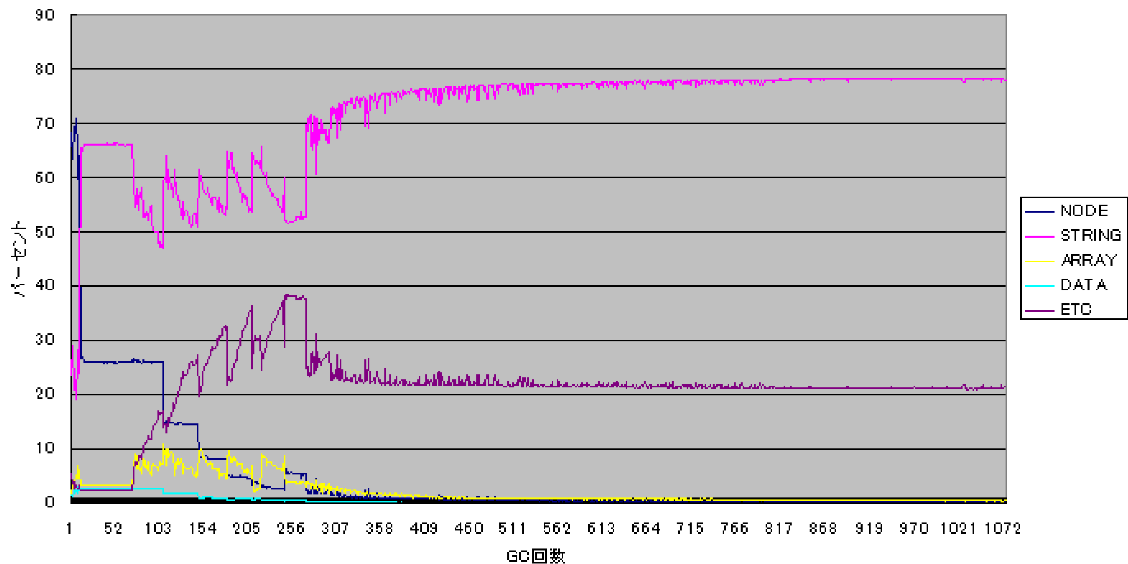

図 4.2: RDoc 実行中におけるオブジェクトの割合

4.3 Rails アプリケーションにおける文字列及び配列オブジェクトの詳細

3.2.1 節で説明したように，文字列オブジェクトと配列オブジェクトは固定長 (RVALUE) ＋可変長という構造をしている．本稿の改善手法は可変長部分に着目して行う．そのため，ここでは文字列及び配列オブジェクトがどの程度，可変長部分を持っているかの調査を行うことでメモリ管理の改善を図る．結果を図 4.3 と図 4.4 に示す．

図 4.3 では文字列オブジェクトの個数全体を 100%としており，図 4.4 では配列オブジェクトの個数全体を 100%としている．

これも 4.1 章と同じように，GC 回数が 40 回目まではサーバー起動時における GC であり，それ以降がリクエスト応答時の GC である．

リクエスト応答時の場合，1 回のリクエストで完結するオブジェクトが多くなるため，回収されるオブジェクトが多くなっている．また可変長部分を使っているオブジェクトは使っていないオブジェクトより少ないということもわかる．

4.4 RDoc における文字列及び配列オブジェクトの詳細

結果を図 4.5 と図 4.6 に示す．

文字列においては，生きているオブジェクトは増加傾向にあり，回収されるオブジェクトは少ないことがわかる．配列に関しては増減が激しく，安定していないが，全体的に固定長部分のみを使った配列が多いことがわかる．それに比べ可変長部分を使用している配列は少ないことがわかる．この結果から，可変長部分に対する改良を行う本稿では，サーバー起動後は起動時より効果が出ないことが予想される．　これらの結果を踏まえ次章でメモリ改善手法を提案する．

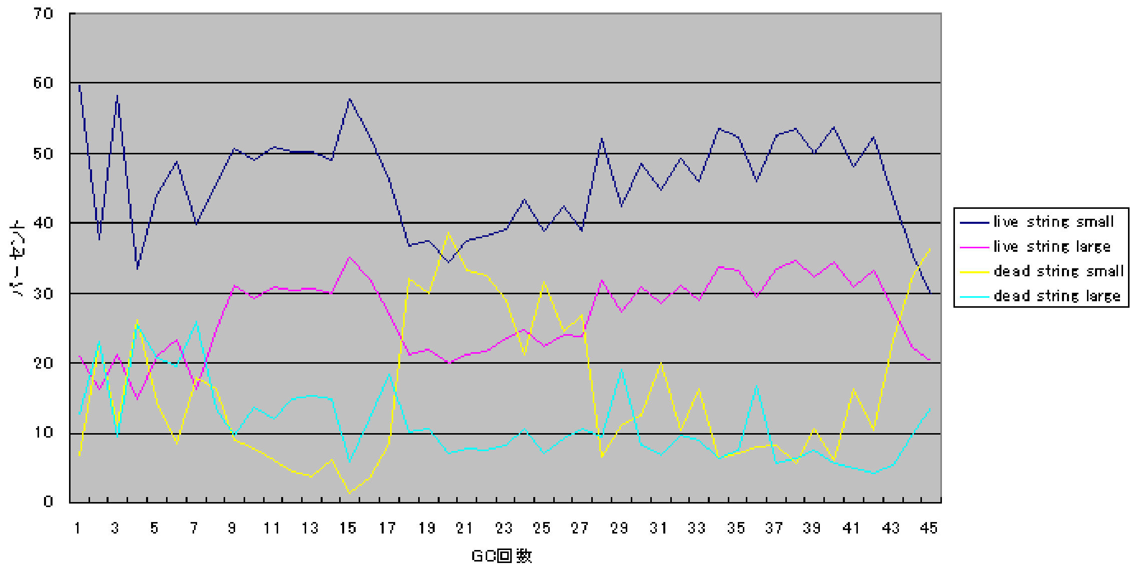

図 4.3: Rails アプリケーションにおける文字列オブジェクトの内訳

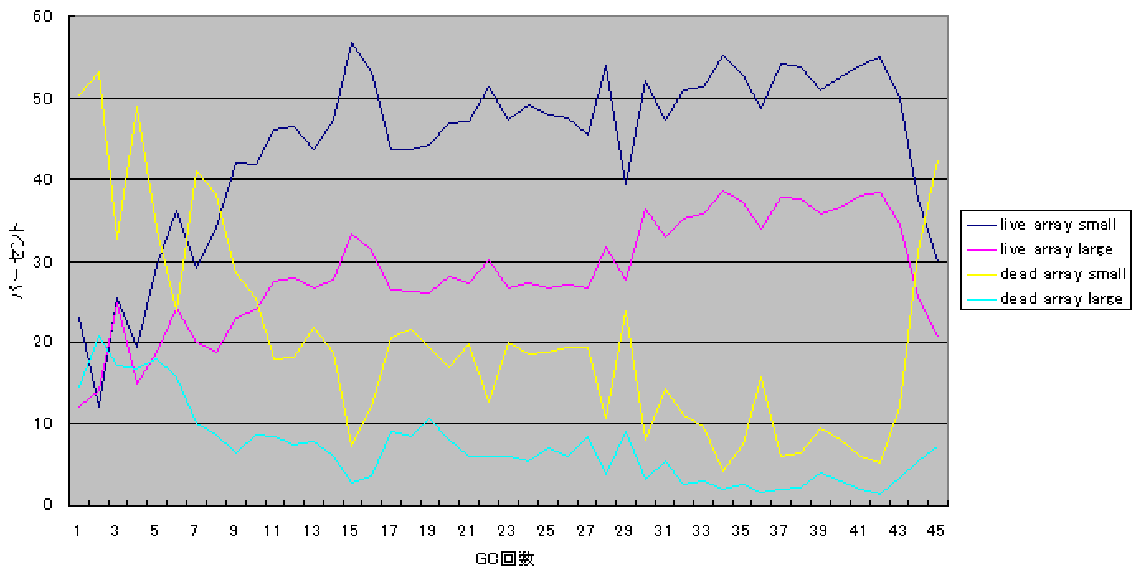

図 4.4: Rails アプリケーションにおける配列オブジェクトの内訳

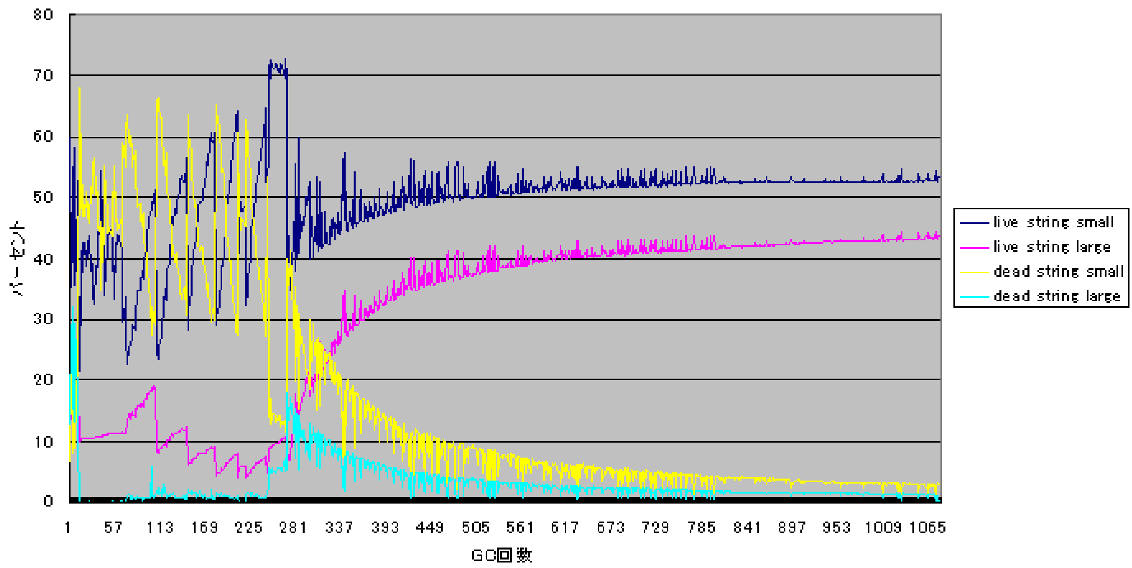

図 4.5: RDoc における文字列オブジェクトの内訳

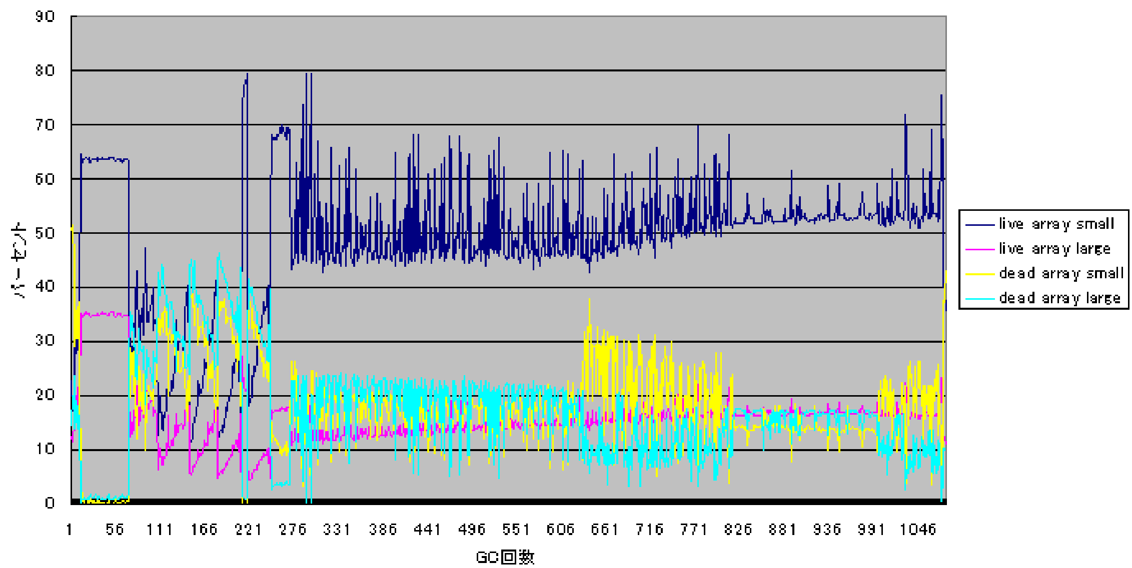

図 4.6: RDoc における配列オブジェクトの内訳

第5章　提案手法

Rubyのメモリ管理をする上で以下のような特徴がある．

- 拡張ライブラリ内で生成されたオブジェクトに関して処理系が管理することはできない
- 拡張ライブラリは処理系が管理しているRVALUEに変更を加えることができる

この特徴によりRubyは容易な拡張が可能となっている．しかしメモリ管理を行う観点から見ると，メモリ管理を複雑なものにしてしまっている．そのためより効率のよいメモリ管理を目指し様々な手法が提案されている[19][17][13]．これらの多くの論文ではRVALUEに主眼をおいているため，外部領域に関して論じていない．そういった現状を踏まえ本稿では外部領域に関する以下の2つの改善手法を提案する．

- リニアアロケーション
- コンパクション

5.1　リニアアロケーション

リニアアロケーションとは広大な空き領域の端から順にアロケーションする方式である．この方式は通常のアロケーションとは異なり，フリーリストによる管理を必要としないため，高速なアロケーションが可能となる．以下が擬似コードである．

リニアアロケーション擬似コード

```
void *free_ptr, *limit_ptr;

void *linear_allocation(size_t size) {
  void *prev_ptr;

  prev_ptr = free_ptr + size;

  if (prev_ptr < limit_ptr) {
    void *result_ptr = free_ptr;
    free_ptr = prev_ptr;
    return result_ptr;
  }
  else {
    … (A)
  }
}
```

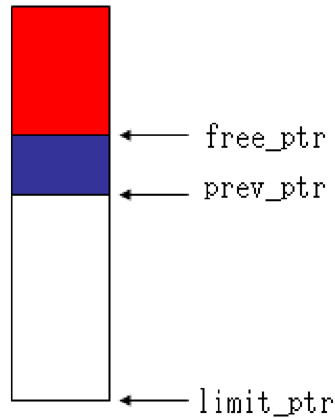

図 5.1: リニアアロケーション概要図

free_ptr は空き領域の先頭ポインタ，limit_ptr は空き領域の末尾のポインタである．アロケーション時には，まず size によって必要な大きさが渡されるため，prev_ptr を用いて，割り当てられるかを決定する．prev_ptr が limit_ptr(空き領域の末尾) を超えていなければ，空き領域の先頭ポインタを返す．このとき次回割り当て時に，その領域が使われないように，free_ptr を進めておく．(A) の部分には空き領域が無いときの処理を書く．通常，GC を起動する，空き領域をさらに確保するなどの処理が考えられる．またアロケーション用ポインタである free_ptr と limit_ptr をグローバルレジスタに保存しておくことで，さらなる高速化が可能となる．

この様子を図で表したのが図 5.1 である．先頭 (最上部) から free_ptr までが割り当て済み領域，free_ptr から prev_ptr までが現在割り当てを行おうとしている領域．prev_ptr から limit_ptr までが空き領域である．この図の場合，必要な領域が空き領域を超えないため，割り当てが可能となる．

既存の多くの言語ではリニアアロケーションを行うために，コピー方式の GC を採用している．コピー方式の GC を行うためにはヒープ領域を 2 分割する必要があり，使用する領域を交互に入れ替える．この入れ替え時にオブジェクトをコピーするのだが，通常の malloc によるアロケーションより高速にコピーを行うために，リニアアロケーションが行われる．同時に次の章で説明するコンパクションも行われるため，入れ替え以降のオブジェクト割り当て時にもリニアアロケーションを行うことができる．

しかし Ruby の場合拡張ライブラリによってオブジェクトの移動に制限がかかる．そのため全てのオブジェクトに対してコピー方式を採用することはできない．これに関しては 5.3 節で詳しく説明する．

5.2 コンパクション

コンパクションとは 2.3.4 節で説明した，圧縮方式の事である．リニアアロケーションがオブジェクトの割り当て時に行われる作業であるのにに対して，コンパクションは GC 時に行われる．コンパクションは以下の手順で行われる．

1. ルート集合から到達可能なオブジェクト全てをマークする

マークする際に，オブジェクトへのポインタを保存しておく

2. マークされなかったオブジェクトを開放する
3. マークの際に保存したポインタのソートを行う
4. ソートされた順に空き領域を埋めつつ，コンパクションを行う

マークアンドスイープを行った後にコンパクションを行う形になる．コンパクションを行うことで，以下の利点が生まれる．

- 連続した空き領域が増える
- 断片化を防ぐことができる

ただし 5.1 節で述べたように，Ruby では移動できないオブジェクトが存在するため，このアルゴリズムを全てのオブジェクトに適用することは不可能である．これに関しては 5.3 節で詳しく説明する．

5.3 Ruby における改善手法

Ruby のメモリ管理をする上で注意すべきことのひとつに「RVALUE の一部を移動させることができない.」という制約がある．これはオブジェクトに対して，拡張ライブラリから `rb_obj_id` 関数や，Ruby 上から `ObjectSpace._id2ref` 関数を実行することで，オブジェクトへのポインタ値をハッシュとする値を取得することができるためである．取得された値はポインタから一意に決定できる値であるため，GC がこの値を書き換えると，プログラムが正しく動作しなくなる可能性がある．この制約により Ruby では RVALUE に対して単純なコピー方式の GC を採用することは不可能であり，同時にコンパクションを行うことが難しくなる．

そこで本稿では RVALUE に対してリニアアロケーション及びコンパクションを適用させるのではなく，RVALUE から伸びている外部領域に対して適用することとする．

Ruby のクラスの中で外部領域が存在するクラスには次の 3 つがある．

- 文字列クラス
- 配列クラス
- 拡張ライブラリクラス

この中で「拡張ライブラリクラス」はオブジェクトの構造がライブラリ作成者に依存するため，改善の対象とすることは難しい．そのため文字列クラスと配列クラスを改善対象とする．これらのクラスは 4 章で述べたように，全体の中でも多くの割合を占めていることが確認できる．提案手法の概念図を図 5.2 に示した．

5.3.1 外部領域に対するリニアアロケーションアルゴリズム

外部領域に対するリニアアロケーションは以下の手順で行われる．まずは文字列に対してのアルゴリズムを示す

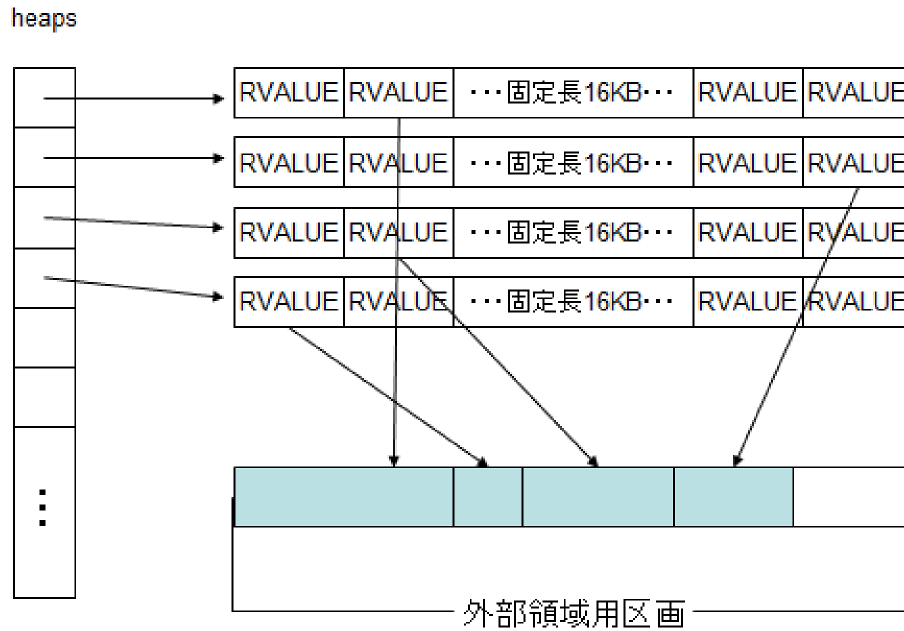

図 5.2: 提案手法概要図

文字列に対するリニアアロケーション擬似コード

```
void *free_ptr, *limit_ptr;
size_t LINEAR_ALLOCATION_AREA_SIZE;

char *string_linear_allocation(size_t size) {
  void *prev_ptr;

  /* (A) */
  if (size > LINEAR_ALLOCATION_AREA_SIZE)
    return (char *)malloc(sizeof(char) * size);

  /* (B) */
  prev_ptr = free_ptr + size;

  if (prev_ptr < limit_ptr) {
    void *result_ptr = free_ptr;
    free_ptr = prev_ptr;
    return (char *)result_ptr;
  }
  else {
    /* (C) */
  }
}
```

リニアアロケーション用領域より大きい領域を必要とした場合，リニアアロケーションが不可能である．そのため通常と同じように malloc によるアロケーションを行う．その部分が (A) である．

(B) からのコードがリニアアロケーションの本質部分になるが，基本的には 5.1 章と同じである．(C) では GC を起動し，外部領域に対してコンパクションを行うことで空き領域を作成する．空き領域が生まれたのであれば，その領域に対して再度リニアアロケーションを行うことで外部領域を確保する．作成できなかった場合は，新たに LINEAR_ALLOCATION_AREA_SIZE の大きさをもつリニアアロケーション用領域を確保し，その領域に対してリニアアロケーションを行う．

配列も上記のアルゴリズムと同じように行えるため，ここでは省略する．

5.3.2　外部領域に対するコンパクションアルゴリズム

基本的なアルゴリズムは 5.2 章で説明したとおりだが，これはリニアアロケーション用領域が 1 つしかない場合である．今回の場合その領域を複数持つため，アルゴリズムに変更を加える必要がある．まずコンパクション用領域に以下のような構造を持たせる．

リニアアロケーション用領域の管理構造

```
struct linear_slot {
  void *start_ptr;
  void *free_ptr;
  void *limit_ptr;
};

struct linear_slot *linears;
```

linear_slot にリニアアロケーション用領域の情報を保存しておく．start_ptr は領域の先頭ポインタ，free_ptr は空き領域の先頭ポインタ，limit_ptr は領域の末尾ポインタである．先頭ポインタは，外部領域がどのリニアアロケーション用領域に属するかを判断するために用いる．linear_slot 構造体は 1 つのリニアアロケーション用領域に対応するため，これを複数管理するためにリスト構造 (linears) を用いる．

このデータ構造の上で以下のアルゴリズムを実行する．

1. ルート集合から到達可能なオブジェクト全てをマークする
   マークする際に，外部領域を使っているオブジェクトが存在するならば，対応するリニアアロケーション用領域ごとにポインタを保存しておく
2. マークされなかったオブジェクトを開放する
3. 1. のマークの際に保存したポインタをリニアアロケーション用領域ごとにソートを行う
4. 各領域でソートされた順に空き領域を埋めつつ，コンパクションする

第6章　実験・評価

実験環境は予備調査と同じ以下の環境で行った．

CPU	Intel Corei7 920
OS	Ubuntu9.10(Linux 2.6.31)
メモリ	3GB
Ruby	1.9.1 p243

実験では外部領域のリニアアロケーション用領域を 1MByte とした．実験に関しては以下の 5 つの実験を行った．

1. Rails で scaffold を使った掲示板システム
2. Ruby 標準ライブラリのソースコードに対して RDoc を実行
3. 外部領域を大量に使うテストプログラム (生存オブジェクトが多い)
4. 外部領域を大量に使うテストプログラム (生存オブジェクトが少ない)
5. 長寿命オブジェクト数の方が短寿命オブジェクト数より多くなるテストプログラム

1.，2. は実践的な状況で，どの程度効果がでるかを確認している．3.，4. の実験を行うことで，提案手法により改善されているかどうかの確認を行う．5. は今回の提案手法では効果が薄いと考えられる状況であるため，実験を行う．

実験の評価は以下の尺度で行う．

- プログラム全体の実行時間
- GC の実行時間
- 文字列・配列オブジェクトのメモリ使用量

6.1　ソースコード

テストプログラムで用いたソースコードは以下のとおりである．

外部領域を大量に使うテストプログラム (生存オブジェクトが多い)

```
LOOP_COUNT = 30000000
ary = Array.new

LOOP_COUNT.times{|i|
  ary[i] = "ptr pointer used malloc#{i}"
}
```

外部領域を大量に使うテストプログラム (生存オブジェクトが少ない)

```
LOOP_COUNT = 30000000
ary = ""

LOOP_COUNT.times{|i|
  ary = "ptr pointer used malloc#{i}"
}
```

長寿命オブジェクト数の方が短寿命オブジェクト数より多くなるテストプログラム

```
LOOP_COUNT1 = 30000000
LOOP_COUNT2 = 10
LOOP_COUNT3 = (LOOP_COUNT1 * 0.1).to_i

ary = Array.new

LOOP_COUNT1.times {|i|
  ary[i] = "ptr pointer used malloc#{i}"
}

LOOP_COUNT2.times{|i|
  LOOP_COUNT3.times {|j|
    ary[i] = "ptr pointer used malloc#{i}#{j}"
  }
}
```

「Rails で scaffold を使った掲示板システム」と「Ruby 標準ライブラリのソースコードに対して RDoc を実行」に関してのソースコードは既存のコードを用いたため本稿には掲載しない．

6.2 実行時間・GC 時間・GC 回数

結果を表 6.1 に示す．ただし Rails プログラムはサーバープログラムであるため，実行時間の測定は行わなかった．

表中の時間に関する単位は全て「秒」である．GC 時間とは平均 GC 時間に GC 回数を掛けたもの，アプリ時間とは全体実行時間から GC 時間を引いたものである．今回の実験では GC 時間がコンパクションの影響を受け，アプリ時間がリニアアロケーションの影響を受けている．

6.3 文字列・配列オブジェクトのメモリ使用量

提案手法において，従来手法と，どの程度メモリ使用量に差があるかを図 6.4 から図 6.8 に示す．この図は「提案手法のメモリ使用量 – 従来手法のメモリ使用量」を図にしたものである．従来手法のメモリ使用量は外部領域の大きさの合計，提案手法のメモリ使用量はリニアアロケーション用領域の合計とする．

Rails					
	平均 GC 時間	GC 回数	GC 時間	アプリ時間	全体実行時間
従来手法	0.007788	45	0.35046		
提案手法	0.008243	45	0.370935		
提案 / 従来	1.058423215	1	1.058423215		

RDoc					
	平均 GC 時間	GC 回数	GC 時間	アプリ時間	全体実行時間
従来手法	1.042456	1069	1114.385464	630.944536	1745.33
提案手法	1.183348	1069	1264.999012	610.480988	1875.48
提案 / 従来	1.135153906	1	1.135153906	0.967566804	1.074570425

テストプログラム（生存多）					
	平均 GC 時間	GC 回数	GC 時間	アプリ時間	全体実行時間
従来手法	0.271633	65	17.656145	40.068855	57.725
提案手法	0.3166601	65	20.5829065	38.0653935	58.6483
提案 / 従来	1.165764469	1	1.165764469	0.949999532	1.015994803

テストプログラム（生存少）					
	平均 GC 時間	GC 回数	GC 時間	アプリ時間	全体実行時間
従来手法	0.000713	6583	4.693679	33.907321	38.601
提案手法	0.000725	6583	4.772675	31.719325	36.492
提案 / 従来	1.016830295	1	1.016830295	0.935471281	0.94536411

テストプログラム（長寿命多）					
	平均 GC 時間	GC 回数	GC 時間	アプリ時間	全体実行時間
従来手法	0.459952	76	34.956352	111.093648	146.05
提案手法	0.623391	76	47.377716	108.832284	156.21
提案 / 従来	1.355339253	1	1.355339253	0.979644525	1.069565217

表 6.1: 実行時間・GC 時間

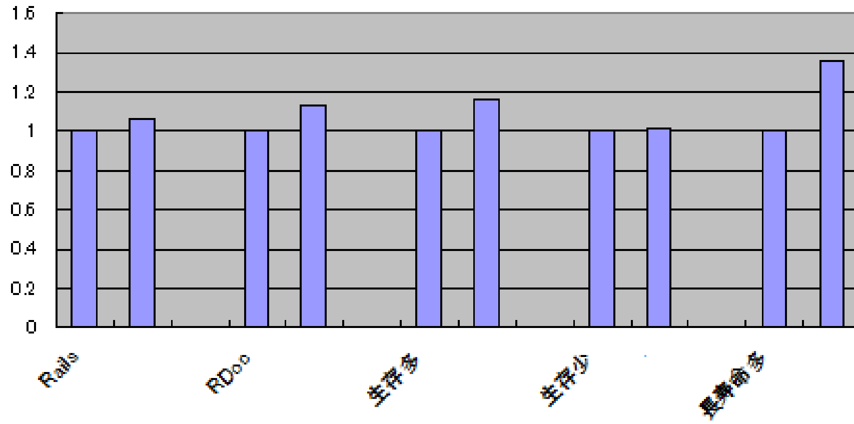

図 6.1: GC 時間比較図

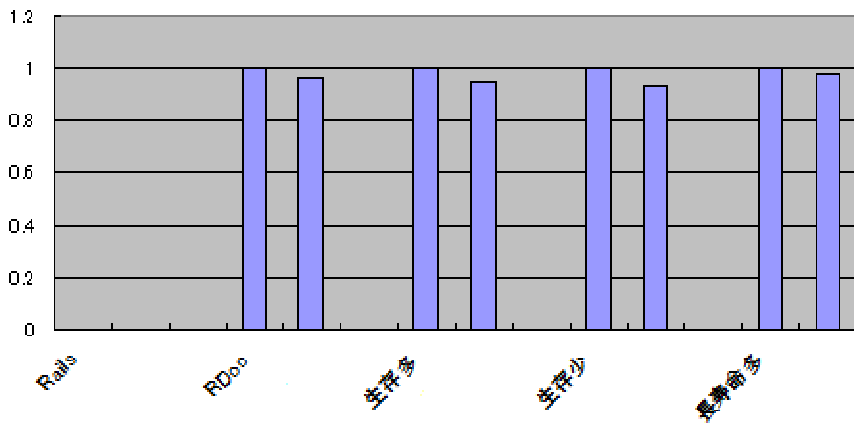

図 6.2: アプリ時間比較図

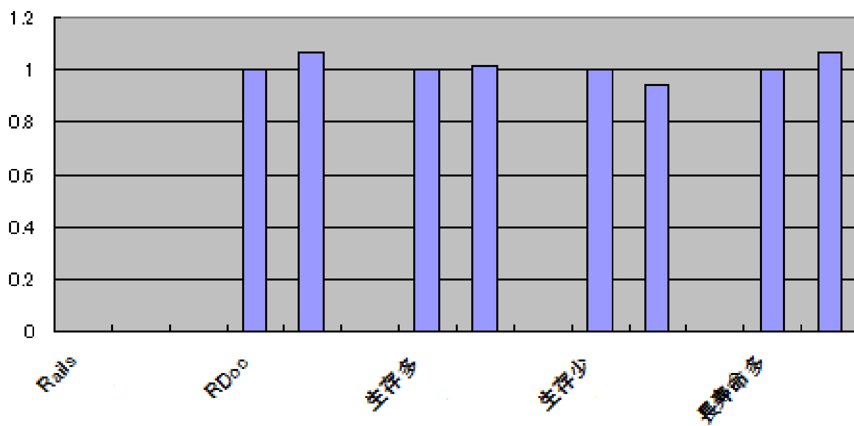

図 6.3: 実行時間比較図

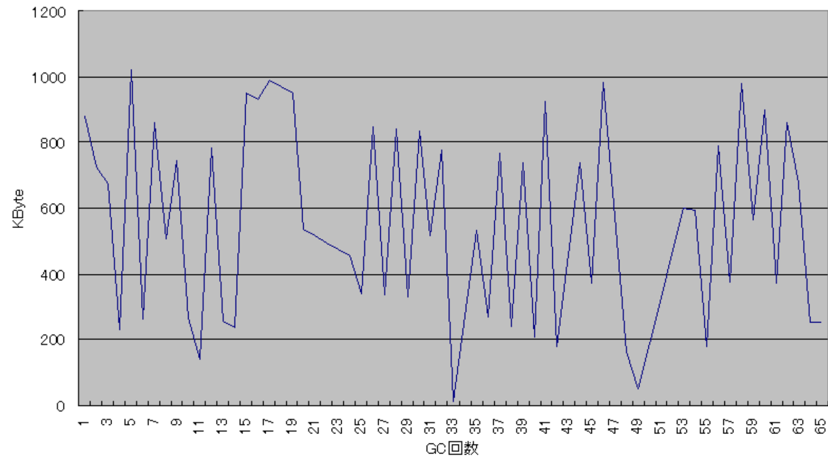

図 6.4: メモリ使用量比較図 (生存オブジェクトが多い)

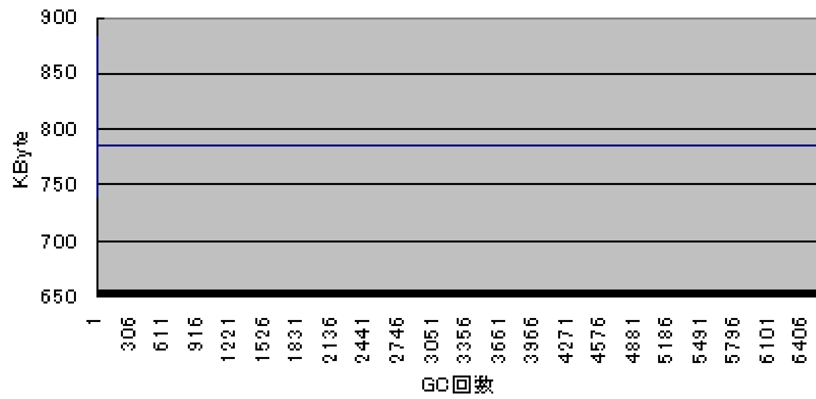

図 6.5: メモリ使用量比較図 (生存オブジェクトが少ない)

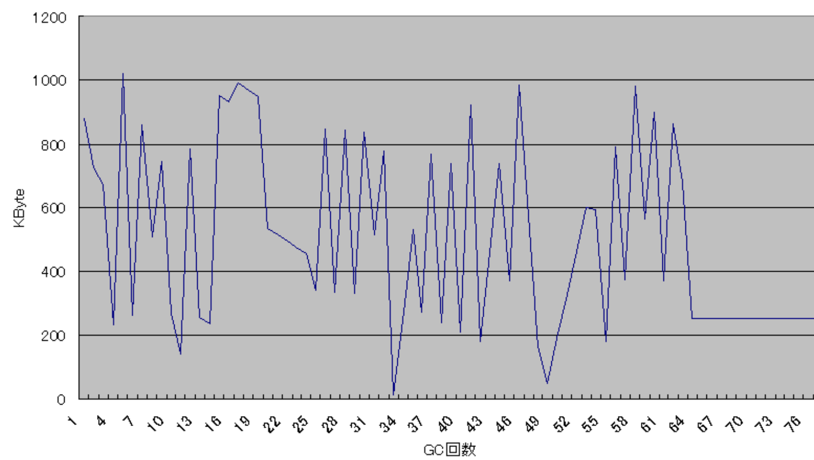

図 6.6: メモリ使用量比較図 (長寿命オブジェクトが多い)

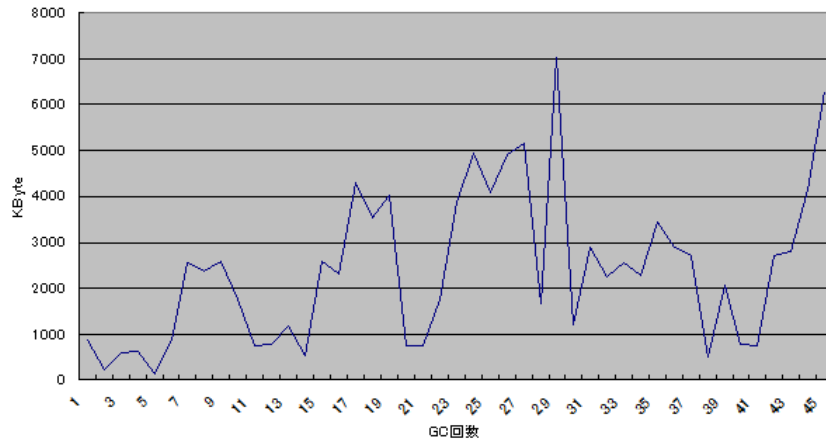

図 6.7: メモリ使用量比較図 (Rails)

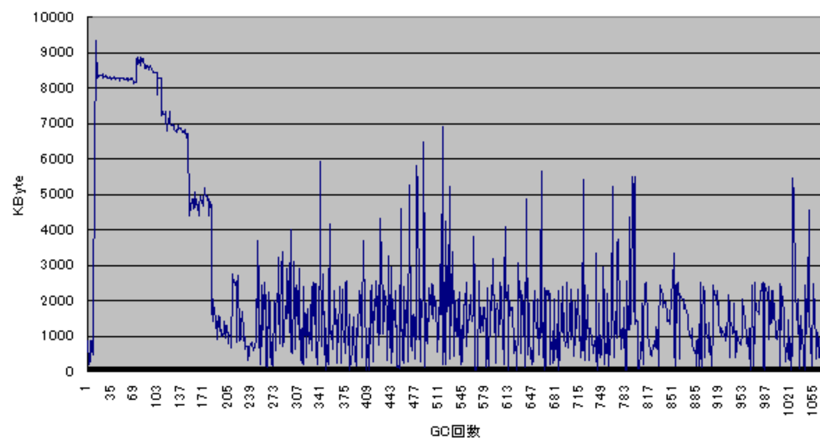

図 6.8: メモリ使用量比較図 (RDoc)

第7章　考察

7.1　実行時間・GC時間・GC回数

GC時間はコンパクションのコストがGCに加わるため，従来手法より時間がかかる結果となっている．またアプリ時間においてはリニアアロケーションを適用することにより，アロケーションのコストが少なくなるため，時間は短くなっている．しかし，リニアアロケーションのコストより，コンパクションのコストの方が大きいため，全体実行時間は遅くなっているものが多い．ただしテストプログラム「生存少」ではコンパクションによるコストがほとんどかかっていないため，全体実行時間は速くなっている．

GC回数について述べる．Rubyでは以下の3つのタイミングでGCが起動する．

1. Rubyプログラム内で`GC.start`を実行したとき
2. RVALUEのフリーリストが空になったとき
3. ファイルディスクリプタが無く，ファイルが開けなかった時

1.はプログラマが能動的に起動を行った場合である．2.はRVALUEの空き領域が無く，オブジェクトを割り当てられなかったときに，死んでいるオブジェクトを破棄することで，オブジェクトが割り当てられることを期待してGCを起動している．3.はファイルオブジェクトが死んでおり，かつ閉じられていない場合，ファイルを閉じた後オブジェクトを破棄する．それによりファイルディスクリプタが空くかもしれない，という考えからGCを起動している．

テストプログラム「生存少」のGC回数はテストプログラム「生存多」のGC回数と比べ100倍近く多くなっている．これは2.の理由によるものである．生存オブジェクトが少ない場合，GCを行うことでほとんどのオブジェクトは破棄される．そのためRVALUEにも空きができる．GC終了後，この空いた部分に新たに生成されたオブジェクトを入れるのだが，空き領域が少なかった場合，GC終了後すぐに再度GCを起動することとなってしまう．そのためテストプログラム「生存少」ではGC回数が多くなっているものと考えられる．

具体的な数を出して説明する．全体で100個のRVALUEが存在するものとする．この内90個は処理系が内部的に使用するオブジェクトによって，使用されているとする．そのためRubyプログラムで使えるRVALUEは10個となる．プログラムが進むことで，10個のオブジェクトが生成され，GCが起動される．今回の条件では生存オブジェクトが少ないため10個すべてが破棄される(本来は全てではないが，簡単のため全てとする)．この10個の部分に新たにオブジェクトを割り当てるのだが，プログラムでかなりの数のオブジェクトが生成される場合，すぐにその領域が埋まってしまい再度GCが起動させる．その後は10個を破棄，10個生成を繰り返すことになる．この例のように，RVALUEの空き個数がプログラム全体で生成されるオブジェクト数に比べかなり少ないと，GCの回数が増加することになる．このため，テストプログラム「生存少」ではGC回数が多くなったと考えられる．

それに対してテストプログラム「生存多」ではGCによってオブジェクトが破棄されない

ため，空きができない．この場合，heap_slot を追加し，RVALUE の管理領域を広げるため，新たに使用できる RVALUE の個数が多くなる．その結果，次の GC まで間隔が開くことになり，結果的にテストプログラム「生存少」より GC の回数が少なくなっているものと考えられる．

RDoc に関しては, 今回の実験では多くのクラスを解析させているため，生成するオブジェクトが多くなり，他の実験結果より GC 回数が増えていると考えられる．さらに RDoc では一度全てのソースコードを読み取り，解析を行った後でドキュメントファイルを作成している．そのため，ドキュメントを生成するまでは全ての解析情報を残しておく必要がある．その結果，生きているオブジェクトが多くなり，死ぬオブジェクトが少なくなる．この状況はテストプログラム「生存多」と似ている．こういった理由も GC 回数が他の実験結果より多くなる原因として考えられる．

7.2 文字列・配列オブジェクトのメモリ使用量

図 6.4 から図 6.8 において，従来手法のメモリ使用量とは外部領域の大きさの合計，提案手法のメモリ使用量とはリニアアロケーション用領域の合計としている．従来手法のメモリ使用量はオブジェクトが確保している領域，提案手法のメモリ使用量はオブジェクトが確保している領域 + 今後確保させるかもしれない領域となる．そのため「提案手法のメモリ使用量 − 従来手法のメモリ使用量」は今後確保させるかもしれない領域ということになる．この領域は，確保される可能性があるだけで，実際には使っていない領域なので，ある意味無駄な領域だが，これはリニアアロケーションを行う上で，必要な無駄である．今回の実験ではこの無駄がどの程度存在するかを確認した．

図 6.4 では，メモリ使用量の差が増加と減少を繰り返しているのがわかる．これは図 6.4 のプログラムではオブジェクトの数は単調増加になるため，提案手法によるリニアアロケーション用領域を新たに確保した際に，差が大きくなりそれ以降はその領域に外部領域を割り当てるため，差が小さくなっているものだと考えられる．ただし配列への代入処理や内部での処理があるため一部周期的ではない部分も見受けられる．

図 6.5 について考えるが，この図は一定値を保ち続けていることがわかる．そこで一定値になる理由を考える．

前章で説明したように，GC はフリーリストの空きがなくなったときに起動される．GC 後オブジェクトが解放されず，フリーリストが空のままだったならば，さらに RVALUE 用の領域を確保する．この大きさは，デフォルトの設定では 16KByte となっている．RVALUE が 5word ということを考えると 32bit 環境では，16KByte 中には RVALUE を 819 個 (16KByte / 5word) 入れることができる．全てのオブジェクトが均等な大きさの外部領域を持つと仮定したとき，リニアアロケーション用領域が 1 つ，1MByte であることを考えると，1 つのオブジェクトに対して 1280Byte(1Mbyte / 819) の領域が与えられることになる．これは 1 バイト文字ならば 1280 文字，2 バイト文字なら 640 文字に相当する．

ここでテストプログラムのソースコードを見ると，プログラム中の文字列はどれも 1280Byte の大きさに達しない．すなわちリニアアロケーション用の領域を使い切る前に，RVALUE 用の領域を使い切り，GC が起動されることになる．GC が起動されると生存オブジェクトが少ない図 6.5 の実験では，生成されたオブジェクトはほぼすべて破棄される．破棄された後，今回のメモリの使用量を計測しているため，その後何回 GC を行ったとしても，同じ挙動を繰り返すことになり，結果的にグラフが一定値を示すことになる．こういった理由により，図 6.5 は一定値を示しているものと考えられる．

図 6.6 について考える．このプログラムの前半部分はテストプログラム「生存多」と同じ処理をしている．そのため GC 回数が 65 回目までは図 6.4 と同じ形になっており，それ以降は一定値を保っている．この一定値は図 6.5 と同じ理由によるものだと考えられる．

テストプログラム「長寿命多」のソースコードの後半ループ部分では，`LOOP_COUNT3` によって書き換えられた値が `LOOP_COUNT2` によって再度書き換えられる．これは書き換えられた部分を短寿命オブジェクトとするためだが，短寿命オブジェクトであるため，すぐに死ぬという性質も持っている．すぐに死ぬオブジェクトが存在する際のグラフは図 6.5 である．そのため図 6.6 の後半部分は一定値になっているものだと考えられる．

図 6.7 や図 6.8 では，1024Kbyte(1MByte) を超えて差が生じている部分が存在する．これはリニアアロケーション用領域 1 つ分以上の領域を無駄にしているということになる．こうした場合外部領域を別のリニアアロケーション用領域に移動させることで無駄を削減することができると考えられるが，今回の実装では行っていない．これに関しては次章の「メモリの断片化対策」で述べる．

7.3 今後の方針

従来手法との動的切り替え

今回の手法は寿命が短いオブジェクトが多い場合に有効であることが確認できた．しかしそうでない場合はコスト的に従来手法より高くなるため，好ましくない．これを改善する手法として，プログラム実行中に動的に切り替える手法が考えられる．動的に切り替える方法としてコンパクションのコストを定義し，その値を元にコンパクション実行の有無を決定する，という方法が考えられる．コンパクションのコストは

- $n =$ 生存オブジェクトの個数
- $s =$ 生存オブジェクトの大きさ
- $c =$ 単位サイズあたりのコピーのコスト

から見積もることができる．単純化のためにキャッシュの影響は考えないものとする．これら 3 つの変数を元に $cost = f(n, s, c)$ となる関数を定義し，その値からコンパクションの有無を決めることで，これは実現できると思われる．

メモリの断片化対策

今回のアルゴリズムではリニアアロケーション用領域をまたがってのコンパクションは行わなかった．これにより，無駄な空き領域が増え，メモリ使用量が大きくなってしまうことになった．これはオブジェクトを移動させることにより，移動先を計算するコスト，移動するためのコストの 2 つがかかる．という理由と，移動先を考える際に，適当に移動させると，移動は行われたが，メモリ使用量は変わらない．ということが考えられたためである．そのため確実にメモリ使用量が減らせるアルゴリズムを実装できるのであれば，それを導入することで，断片化を防ぐことができると考えられる．

第8章　関連研究

8.1　Mostly-Copying GC

鵜川 [19] は Ruby において Mostly-Copying GC の研究を行っている．Mostly-Copying GC[3] とは，Bartlett が Scheme 言語から C 言語へのコンパイラのために考案した保守的なコピー GC である．通常のコピー方式の GC ではヒープ領域を 2 分割にして行うが，Mostly-Copying GC の場合はある一定の大きさでヒープ領域を分割する．Ruby は従来から RVALUE を管理するために 16KByte に分割して管理を行っていたため，Ruby における Mostly-Coying GC はヒープ領域をその大きさに分割して GC を行う．これにより RVALUE の断片化抑制に成功しているが，プログラムによっては RVALUE のコピーがほとんど行えないため，そういった場合の効果は薄くなっている．

これに関連して永原ら [13] はさらに Mostly-Copying の世代別 GC の実現を目指している．ただしこれに関しては実装する前の調査段階にあるため，実現はしていない．

8.2　ヒープ領域の拡張

笹田 [17] は Ruby1.9 系から導入された仮想マシン yarv の開発者であるが，彼は GC の時間を短縮するために，ヒープ領域を大きくすることで時間短縮を図っている．ヒープ領域を予め大きめに確保しておくことで，数多くのオブジェクトを生成することができるようになるため，GC の回数が減り，GC 全体の時間が短縮される．この方針は，既存の Ruby 処理系に大きな変更を加えずにできるために，組み込み易いのが特徴である．

8.3　スナップショット GC

相川ら [1] はスナップショット GC の実装を行っている．スナップショット GC とは GC 開始時のオブジェクトの状態をスナップショットのように写真に撮り，その状態でのゴミを回収する手法である．このアルゴリズムは並列化，並行化において一番性能の良いアルゴリズムと考えられている．しかしこれを行うためには，プログラムのソースコードに変更を加える必要があるため，変更箇所を比較的容易に発見する方法も提案されている．

8.4　ビットマップマーキング方式 GC

中村ら [14] は Ruby が昨今 WEB アプリケーションとして使われていることから，そこで効率的に動く GC を考えている．Apache を用いてサーバー側処理を実装すると，サブプロセスを大量に作ることになる．サブプロセスを生成した際，通常サブプロセスのために親プロセスのメモリのコピーを作成する．しかしサブプロセスを生成する際に，その作業を毎回行っていてはコピーでかなりの時間をとられてしまう．それを軽減するために，Linux では

サブプロセスが親プロセスと共有している部分を書き換えた場合のみ，その書き換え部分のコピーを行うことにしている．これにより無駄なコピーをすることがなくなるため，効率が良いのだが，残念ながらこの方法はマークアンドスイープ方式とは相性が悪いという問題がある．この問題を解消するために中村らはマークする場所をオブジェクトではなく，オブジェクトとは別の場所に集めることで，コピーの量を少なくする提案をしている．その結果，メモリ使用量が軽減されている．

第9章　まとめ

本研究ではRubyにおけるメモリ管理の改善を目指し，外部領域に対して，リニアアロケーションによる高速なアロケーションの実現とコンパクションによるメモリ領域の削減の研究を行った．これによりテストプログラムの一部では実行時間に改善が見られたが，Ruby on Railsを用いたプログラムなどでは実行時間の増加が見られた．この実行時間増加の原因はコンパクションによるコストの増加によるものだと考えられる．そのため今後，コンパクションの有無を決定する手法を確立することで，ボトルネックが解消しさらなる改善が進むものと思われる．

謝辞

本研究を進めるにあたり多大なるご指導ご鞭撻をいただいた，東京工業大学 大学院情報理工学研究科 数理・計算科学専攻教授の佐々政孝先生に深く感謝の意を表します.

また，GC に関する様々な知識を教えていただきました東京工業大学 大学院情報理工学研究科 数理・計算科学専攻の遠藤敏夫氏，Ruby に関する仕様などを教えていただきました東京大学 大学院情報理工学系研究科 創造情報学専攻 笹田耕一氏，さまざまな面で助力をいただきました佐々研究室の皆様にあらためまして，ここに深くお礼申し上げます.

参考文献

[1] 相川光・笹田耕一・本位田真一. Ruby 処理系へのスナップショット GC の実装. 情報処理学会プログラミング研究会, Vol. 3, No.3, 2008.

[2] Baker, H. G. Jr. The treadmill: real-time garbage collection without motion sickness. ACM SIGPLAN Notices, Vol. 27, No. 3, (Mar., 1992), pp. 66-70.

[3] Bartlett, J.F. Compacting garbage collection with ambiguous roots. ACM SIGPLAN Lisp Pointer, Vol. 1, No. 6, (1988), pp. 3-12.

[4] Wilco Bauwer, Microsoft inc. IronRuby.net. http://ironruby.net/

[5] Boehm, H. Dynamic Memory Allocation and Garbage Collection. Computers in Physics, Vol. 9, No. 3, May/June 1995, pp. 297-303.

[6] Cheney,C.J. A nonrecursive list compacting algorithm. Comm. ACM, Vol. 13, No. 11, (Nov., 1970), pp. 677-678.

[7] David Heinemeier Hansson. Ruby on Rails: Web development that doesn't hurt. http://www.rubyonRails.org

[8] Rob Gordon, 林秀幸 (訳). JNI:Java Native Interface プログラミング. ソフトバンククリエイティブ, 1998.

[9] 小林義徳・遠藤敏夫・田浦健次郎・米澤明憲，マークスイープとコピーの混合による効率的なゴミ集め，日本ソフトウェア科学会プログラミングおよびプログラミング言語ワークショップ 2003 (PPL 2003), Mar., 2003.

[10] 松本行弘．Ruby の真実．情報処理．Vol. 44，No. 5，pp. 515-521，2003.

[11] まつもとゆきひろ，石塚圭樹．オブジェクト指向言語 Ruby．株式会社アスキー 1999.

[12] まつもとゆきひろ他．オブジェクト指向スクリプト言語 ruby．http://www.ruby-lang.org/ja/

[13] 永原治・鵜川始陽・岩崎 英哉. 世代別 Mostly-Copying GC の Ruby VM への実装に向けて. 情報処理学会プログラミング研究会, 2009-3-(2).

[14] 中村成洋・松本行弘. 効率的なビットマップマーキングを用いた Ruby 用ごみ集め実装. 情報処理学会プログラミング研究会, 2008-3-(1).

[15] Charles Nutter, Thomas Enebo, Ola Bini, Nick Sieger. JRuby.org. http://www.jruby.org/

[16] O'Reilly Media Inc. Perl.com: The source for perl - perl development, perl conference. http://www.perl.com/

[17] 笹田耕一. Ruby のメモリ管理の改善. 情報処理学会プログラミング研究会, 2009-3-(2).

[18] 笹田耕一, 松本行弘, 前田敦司, 並木美太郎. Ruby 用仮想マシン yarv の実装と評価. 情報処理学会論文誌:プログラミング, Vol. 47, No. SIG 2(PRO28), pp. 55-73, 2006.

[19] 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, Vol. 2, No.2, pp. 1-12, 2009.