Java 処理系の使用メモリ削減の一手法

A Method to Reduce the Memory Footprint of Java VM

河内谷 清久仁 † 緒方 一則 † 小野寺 民也 †

Kiyokuni KAWACHIYA Kazunori OGATA Tamiya ONODERA

† 日本アイ・ビー・エム（株）東京基礎研究所

IBM Research, Tokyo Research Laboratory

kawatiya@jp.ibm.com

本論文では,「文字列メモリの無駄」を減らすことで Java 処理系の使用メモリを削減する手法について述べる．最近の Java アプリケーションでは，ヒープ領域の多くの部分が文字列データで占められており，その割合は 30%にものぼる．実際のアプリケーションにおいて文字列データの使用状況を解析したところ，同じ内容の `String` オブジェクトが多数存在している，内容を保持するための `char` 配列に不使用部分が存在する，という 2 種類の無駄があることが判明した．これらの無駄はライブオブジェクトの形で存在しているため，従来のガーベージコレクション（GC）では取り除くことができない．そこで我々は，通常の GC と協調して動作し上記の無駄を取り除く「文字列ガーベージコレクション（StringGC）」を提案する．IBM の商用 Java 処理系に StringGC のプロトタイプを実装し評価を行ったところ，文字列メモリの無駄の 90%以上を取り除くことができ，ライブヒープのサイズを最大 15%減らすことができた．

1 はじめに

Java 言語 [7] の問題の一つに，C などで書かれたプログラムに比べて使用メモリサイズが大きくなることがあげられる．その原因の一つは，Java がオブジェクト指向言語であることで，アプリケーション，ミドルウェア，および Java 処理系自身によって多数のオブジェクトが生成されるため，ヒープ領域が大きくなってしまう．ヒープ領域はコードやクラス用データなどと違い他の Java プロセスと共有する [5, 15] ことが難しいため，そのサイズを減らすことは Java のメモリ使用効率向上のために非常に重要である．

大規模 Java アプリケーションを実行し，そのヒープ領域をオブジェクトのクラス別に分析すると，`char` の配列クラス（`char[]`）のオブジェクトがライブヒープの約 30%を占めていることがわかった．`char` 配列オブジェクトの多くは `String` クラスのあらわす文字列を保持するために用いられているが，その詳細を分析すると以下のような無駄があることが判明した．

A. 同じ内容の `String` が多数存在している．

B. `char` 配列内に使用されていない部分がある．

これらの無駄はヒープの 17%にもなるが，ライブなオブジェクトとして（もしくはライブオブジェクト内部に）存在しているため，デッドオブジェクトを対象とする従来のガーベージコレクションでは取り除くことができなかった．

本論文では，上であげた `String` 用文字データの 2 つの無駄を取り除くことでメモリ使用効率を上げる手法について提案する．この「文字列ガーベージコレクション（StringGC）」により，同一内容の `String` がまとめられ，`char` 配列の不使用部分が取り除かれる．プロトタイプを IBM の商用 Java 処理系に実装し評価を行ったところ，文字列メモリの無駄の 90%以上を取り除くことができ，ライブヒープのサイズを最大 15%減らすことができた．

本論文の主な貢献としては以下の点があげられる．

- Java 処理系における「文字列メモリの無駄」の発見と実態調査．
- 文字列メモリの無駄を取り除くための「StringGC」の提案．
- StringGC の効率よいプロトタイプ実装と，実アプリケーションでの有効性の確認．

本論文の残りの部分は以下のように構成されている．まず 2 章では，Java における `String` の典型的な実装と文字列メモリの無駄について示す．次に 3 章で，この無駄を取り除くための StringGC について提案し，いくつかの実装バリエーションについても議論する．4 章では，StringGC プロトタイプを用いた性能評価について示す．5 章で関連する研究について述べ，6 章でまとめと今後の課題を示す．

```
1 public final class String ... { // フィールドはすべて private
2   private char[] value; // 文字データを保持する char 配列
3   private int offset;   // 文字列の char 配列内での開始位置
4   private int count;    // 文字列の長さ
5         :
6 }
```

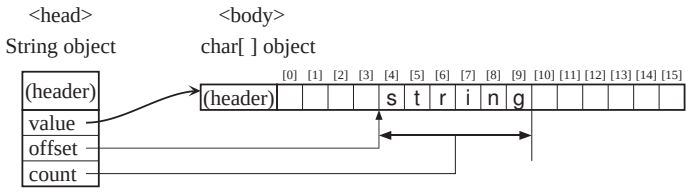


図 1: 典型的な実装における String オブジェクトの構造

2 Java における文字列メモリの無駄

本章では，Java における文字列処理の仕様と典型的な実装について述べ，その中にあるメモリ使用上の無駄について示す.

2.1 Java における文字列処理

Java で文字列を扱うためのクラス「String」は処理系から提供される基本クラスの一つである [7]. その大きな特徴として，ユーザープログラムは一旦作った String オブジェクトの内容を変更できない（immutable）ことがあげられる．内容を変更可能な文字列構造としては別途，StringBuffer などのクラスが用意されている．これらのクラスのオブジェクトは，javac コンパイラによって暗黙のうちに生成・変換されることがある.

String クラスをどのように実装するかは Java 仕様では規定されていないが，その典型的な実装は図 1 のようになる．これは Apache Harmony プロジェクト [1] による実装の簡略版であるが，筆者らの知るほとんどの Java 処理系において同様の実装が用いられている．各 String オブジェクトは 3 つのフィールド value，offset，count を含んでいる．value フィールドは，文字データの実体を保持する char 配列オブジェクトを指している．ただし，この char 配列全体がそのまま String オブジェクトのあらわす文字列というわけではなく，その部分を示すために offset と count の 2 つの int フィールドが用いられる．たとえば，図 1 下の String オブジェクトは，"string" という文字列をあらわしている[1]．なお，これらのフィールドはすべて private 宣言されており，ユーザープログラムから直接アクセスすることはできない.

このような構造になっているのは，StringBuffer.toString() や String.substring() などで新しく String オブジェクトを作る際に，いちいち char 配列の内容をコピーせず共用するためである[2]．しかしその代償として，文字データの前後に使用されていない部分ができてしまう可能性がある．たとえば，図 2 上の Java プログラムを考えてみる．Java 処理系の実装にもよるが，9 行目の時点で String jar2 は図 2 下のような構造になる．この例では，34 文字分のサイズを持った char 配列のうち，26 文字分が不使用領域となっている．なお，使用されている文字列全体より char 配列が大きいのは，4 行目の処理で内部的に StringBuffer が使用され，大き目の char 配列がとられるためである．これは説明のために用意したプログラムであるが，Java 処理系内部では同様の文字列操作が頻繁に行われている.

Java の文字列処理に関するもう一つの無駄として，同じ内容の文字列データ（String および char 配列）が複数存在しているという問題がある．図 2 の例でも，jar2 は usrjar と同じ内容"user.jar"であるが，これらの 2 つの String オブジェクトおよび内容を保持する char 配列オブジェクトはヒープ内に個別に存在している．本節冒頭で述べたように，Java の String オブジェクトは内容を変更不能であり，これらはプログラムの動作に影響を与えずにまとめられる可能性がある.

[1] 図 1 の各オブジェクト中の「header」は，オブジェクトのクラス情報などを管理するために Java 処理系が使用するエリアである.

[2] StringBuffer の文字列はその insert()，delete() などのメソッドを通じて修正可能であるが，String オブジェクトと char 配列を共用している StringBuffer に対して修正操作が行われる場合は，修正前にまず char 配列のコピーが行われる．なお，StringBuffer は文字列操作のために一時的にしか使用されないことが多いため，実際にこのコピー処理が行われることは少ないと考えられる.

```
1 class StringSample {
2   public static void main(String[] args) {
3     String sysjar = "system.jar", usrjar = "user.jar";
4     String tmpstr = sysjar + ":" + usrjar + ":" + ".";
5     int colon1 = tmpstr.indexOf(":"),
6     int colon2 = tmpstr.indexOf(":", colon1+1);
7     String jar2 = tmpstr.substring(colon1+1, colon2);
8     tmpstr = null;
9
10    System.out.println(jar2); // "user.jar"が出力される
11  }
12 }
```

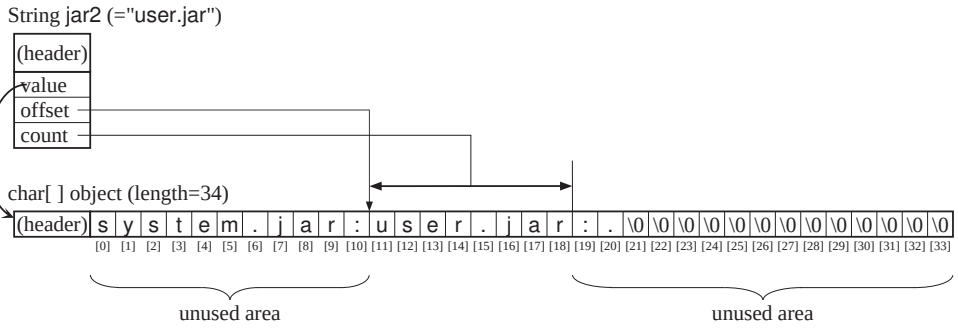


図 2: 文字列操作のサンプルプログラムと，それによって作られる内部データ構造

Java ヒープ内の，どこからも参照されていないオブジェクト（デッドオブジェクト）は，ガーベージコレクション（GC）[11] により回収することができる．しかし，ここであげた 2 つの無駄はいずれも，ライブな `String` オブジェクトや `char` 配列に存在するものであり，通常の GC ではこれらの無駄を回収することはできない．文字列メモリの無駄を解消しメモリ使用量を削減するには，Java 処理系に新しい仕組みを導入する必要がある．

2.2 文字列メモリの無駄の調査

最近の Java アプリケーションでは，XML 処理やデータベースアクセスなどのために文字列データを扱う機会が増えており，上であげた 2 つの文字列メモリの無駄はメモリ管理上無視できないものになってきている可能性がある．そこで，実際の Java アプリケーションの実行状況において，それぞれの無駄がどれくらい存在しているのかの調査を行った．

調査には IBM の商用 Java 処理系である J9 Java VM [4, 8] 1.5.0 for Linux を用い，指定された「調査ポイント」でシステム GC を実行して，生き残ったライブなオブジェクトをオフラインで解析した．調査に用いた Java アプリケーションと調査ポイントは以下のとおりである．

Trade6:
IBM WebSphere Application Server (WAS) [10] Version 6.1 に，Trade6 ベンチマークアプリケーション [9] をインストールし，リクエストを 3 分間処理した時点で調査．

Tuscany:
Service Component Architecture (SCA) [13] の処理フレームワークを提供するオープンソースミドルウェアである Tuscany [3] Incubating-M1 を，Apache Tomcat [2] にインストールし，サンプル SCA アプリケーション BigBank に対するリクエストを 3 分間処理した時点で調査．

調査結果をまとめたものを表 1 に示す．「全ライブオブジェクト」は，調査ポイントでのライブなオブジェクトの数とサイズで，GC 後に残っていた全オブジェクトを集計したものである．なお，測定に用いた J9 Java VM は，type-accurate GC [11] を装備しており [4]，どこからも参照されていないオブジェクトは完全に回収されている．

「`String` オブジェクト」は，ヒープ内の `String` オブジェクトの数とサイズを示している．なお，この数値には `value` フィールドから指される `char` 配列オブジェクトの分は含まれていない．次の「重複しているもの」の行はこのうち，文字列としての内容が同じであるため削除可能な `String` オブジェクト

測定項目	Trade6 数	Trade6 サイズ	Tuscany 数	Tuscany サイズ
全ライブオブジェクト	621,463	33,938 KB	281,973	15,332 KB
String オブジェクト	106,957	2,995 KB	57,922	1,622 KB
- 重複しているもの	42,163	1,181 KB	33,939	950 KB
char 配列	97,306	9,641 KB	49,150	4,470 KB
- String のみが使用	95,639	8,810 KB	48,118	3,628 KB
- 重複しているもの	32,839	2,107 KB	25,578	1,596 KB
- 重複以外の不使用	5,923	139 KB	2,019	46 KB
削減可能な量	75,002	3,427 KB	59,517	2,593 KB
- ヒープに対する割合	12.1%	10.1%	21.1%	16.9%

表 1: 実際の Java アプリケーションにおける文字列メモリの無駄

の数とサイズを示している[3]．たとえば Trade6 の場合，106,957 個の String オブジェクトのうち 42,163 個（39.4%）は重複しており削除できるということである．

「char 配列」は，ヒープ内の char 配列オブジェクトの総数とサイズである．次の「String のみが使用」は，このうち String の内容としてのみ使用されているもので，StringBuffer のデータとして共用されているものや char 配列自体がプログラムから使用されているものを除いた値である．ほとんど（個数で 98%）の char 配列が，String の内容を保持するために使われていることがわかる．「重複しているもの」はこのうち，同じ内容の String をまとめる作業によって使用されなくなり，削除可能となる char 配列オブジェクトの数とサイズを示している．「重複以外の不使用」の行は，この重複した char 配列を除いた後の状況において，図 2 で説明した不使用領域が存在している char 配列の数と，不使用領域の総バイト数を示している．

「削減可能な量」は，以上の文字列メモリの無駄を合計したもので，削減可能なオブジェクトの総数と，削減されるバイト数を示している[4]．最後の「ヒープに対する割合」は，それらのライブヒープに対する割合である．

以上の調査結果をまとめると，次のことが言える．

- 調査した Java アプリケーションでは，String データ保持のために 5〜12MB が消費されている．これは，ライブヒープサイズの約 35%にあたる量である．
- このうち 30〜50%は，重複や不使用による無駄である．これを取り除くことができれば，必要ヒープエリアを 10〜17%も減らすことができる．

この結果は，実 Java アプリケーションにおいて文字列メモリの無駄を取り除くことが非常に重要であることを示している．次章では，そのための具体的方法について検討する．

3 「StringGC」の提案

前章の調査で，String に関連する以下の 2 つの無駄をなくすことで，Java のヒープ領域を最大 17%削減できることがわかった．

A. 同じ内容の String オブジェクトが多数存在している．
B. 内容を保持するための char 配列に不使用部分が存在する，

本章では，これらの無駄を取り除き Java 処理系のメモリ使用効率を改善するための機構として，「文字列ガーベージコレクション（StringGC）」を提案する．

3.1 アルゴリズム

StringGC は，Java 処理系にもともと備わっているガーベージコレクションに以下の処理ステップを追加することで実現できる．なお，ここでは，stop-the-world 型の GC を例として説明を行うが，GC において開発された様々な並行処理手法 [11] が StringGC にも適用可能である．一部については 3.2 節で議論する．

[3]同じ内容の String オブジェクトが 100 個あった場合，「重複しているもの」の数は「99」となる．
[4]不使用領域のある char 配列は，削減可能なオブジェクトには数えていないが，その不使用領域は削減可能なバイト数には含んでいる．

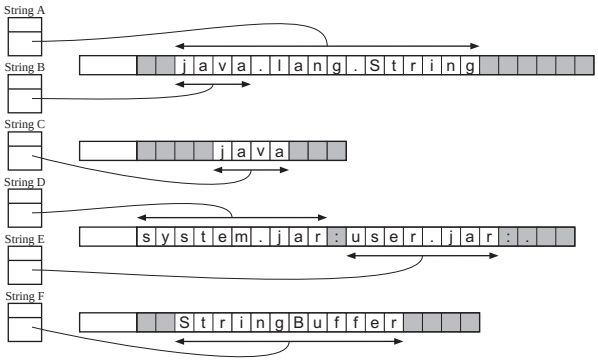
ステップ 1: String と char 配列の一覧表を作成.
ヒープをスキャンし，ライブな String オブジェクトと char 配列の一覧表を作る．この際，各オブジェクトがどこから参照されているかについても記録する．char 配列が String 以外からも参照されている場合は変形不能なので表から取り除く．以上の処理は通常の GC のマーク処理の際に行うことができる.

ステップ 2: 同じ内容の String をまとめる.
一覧表の String オブジェクトを調べ，同じ内容のものがあった場合は一つにまとめてしまう．この際，元の String オブジェクトの参照元は参照を付け替える．これは通常 GC におけるオブジェクト移動と同様の手法で行える．この処理により上述の無駄 A が解決される．なお，String オブジェクト自体をまとめることの是非については 3.3 節について議論している.

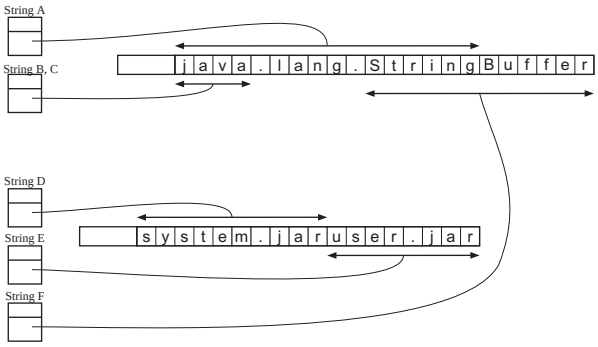
ステップ 3: String 用 char 配列の不使用部分除去.
次に，一覧表の char 配列を順に調べる．参照元の String（複数あってもよい）のどれからも使用されていない部分があった場合，その部分を切り捨てて char 配列オブジェクトを作りなおす．同時に，参照元 String オブジェクトの value および offset フィールドを適切に書き換える．これにより上述の無駄 B が解決される.

ステップ 4: String 用 char 配列をまとめる.
一覧表の char 配列を調べ，他の char 配列と文字データ（の一部）を共用可能な場合，それを一つの char 配列にまとめる．具体的には，char 配列の内容が他の char 配列の一部になっている場合などがあげられる．変更があった場合は上と同様に，元の char 配列を指していた String オブジェクトを適切に修正する.

図 3 は，StringGC の適用イメージを示したものである．(a) は適用前の状態で，灰色部分が char 配列内の無駄を示している（無駄 B）．また，(部分的に) 重複した String がいくつか存在している（無駄 A）．これに上記の各ステップを完全に適用すると (b) のようになる．同じ内容"java"を持つ String オブジェクト B と C が一つにまとめられ（ステップ 2），各 char 配列内の，String の内容として使用されていなかった部分（灰色部分）はすべて取り除かれている（ステップ 3）．また，一部分が同じである String



(a) StringGC 適用前



(b) StringGC 適用後

図 3: 文字列メモリの無駄と，StringGC による削減

オブジェクト A と F が同じ char 配列を共用するようになっている（ステップ 4）.

3.2 バリエーション

上記の StringGC の各ステップには，組み合わせる GC や必要な効果と処理時間などに応じたいくつかのバリエーションが考えられる.

まず，ステップ 1 の一覧表作成処理は各 String や char 配列ごとに独立して行えるので，一回の StringGC ですべてについての表を作らず部分的に行うことも可能である．一覧表に使用できるメモリ量が少ない場合は，最初に出てきた n 個のオブジェクトについてだけ行うなどが考えられる．また，複数プロセッサ（スレッド）で分担して並列に作成を行うこともできる.

ステップ 1 で作る一覧表と参照元の情報は，必ずしも新たにメモリ領域を確保して構築しなくてもよい．GC の過程においては通常，各オブジェクトの参照元を知ることが出来る構造が作られる．このような構造が GC によって作られていれば，ライブなオブジェクトを随時スキャンし String や char 配列

だった場合に処理を行うことで，追加のメモリオーバーヘッドなしにステップ 2〜4 の処理を行うことができる．

ステップ 3 で char 配列内の不使用部分を調査する際に，各文字ごとに記録するのではなく，String から使用されている文字データの「最初と最後だけ」を記録する簡略化が考えられる．この方式では，図 3(a) の 3 つ目の char 配列のように途中に不使用文字があった場合に取り除くことができないが，そのような状況はあまりないと考えられる．また，char 配列の先頭部分が使われなくなるのは，String.substring() などで部分文字列を取り出した場合だけなので，使用されている文字列の「最後の位置だけ」を記録するのでも十分かもしれない．

ステップ 4 の char 配列をまとめる処理は，多くのバリエーションが考えられる部分である．最も単純な方式は，全く同じ内容のものがあった場合にそれらを一つの char 配列にまとめるというものである．なお，実際の実装では，char 配列の再構成によるコピーを減らすため，ステップ 4 はステップ 3 とまとめて行うことがのぞましいだろう．

ステップ 4 の発展実装として，1 つの char 配列が別の char 配列の部分文字列となっている場合，前者を後者にマージしてしまうことが考えられる．図 3 の"java"という文字データはこれでまとめられる．さらに進んだ実装として，char 配列の後半と別の char 配列の前半に共通部分がある場合も，一つの char 配列にまとめることが可能である．図 3 の例では，"String"という文字データがこの手法によりまとめられている．

どの実装バリエーションを用いるかは，必要な効果とかけられる処理時間によって異なってくる．しかし，表 1 の結果によれば，単純に「同じ内容の場合をまとめる」だけでも，String 用 char 配列の量を 30〜50%削減できることがわかる．組み合わせる GC にいくつかの処理レベルがある場合，フル GC の場合だけよりアグレッシブな手法を用いるという方法も考えられる．

3.3 議論

StringGC の処理ステップ 2 では，String オブジェクトが同じ内容である場合，オブジェクト自体をまとめるという処理を行っている．たとえば図 3 では，"java"という内容を持つ String オブジェクト B と C が一つにまとめられている．この方法には，String オブジェクトの「(非) 同一性」に基づいた処理を行っている Java プログラムがあった場合，挙動が変わってしまうという危険性がある．そのような処理には，「==」によるオブジェクト比較，モニタの獲得・解放，System.identityHashCode() の使用などがあげられる．

```
 1 class BadManner {
 2  public static void main(String[] args) {
 3   String s1 = new String("java");
 4   String s2 = new String("java");
 5   if (s1 == s2) // should use s1.equals(s2)
 6    System.out.println("Same Strings");
 7   else
 8    System.out.println("Different Strings");
 9  }
10 }
```

図 4: String をまとめると挙動が変わるプログラム

図 4 は挙動が変わってしまうプログラムの例で，String オブジェクト s1 と s2 がまとめられてしまうと，本来「Different Strings」となるべき出力が「Same Strings」となってしまう．しかし，Java において文字列比較に「==」を用いることは本来避けるべきで，String.equals() を使用するべきである．我々は，「行儀よく」書かれた Java アプリケーションでは String の同一性に依存した処理は行われていないと考えている．

もし，String オブジェクトをまとめることに抵抗があるのであれば，ステップ 2 において，同じ内容でも String 自体はまとめず char 配列の部分だけをまとめるという方法も可能である．この場合，削減できるメモリ量は減ってしまうが，表 1 の結果によれば，それでも文字列メモリの無駄の 65%程度は取り除くことができる．

一方，同じ内容の String オブジェクトをまとめることには，String.equals() の処理が速くなるという副次効果がある．このメソッドは 2 つのオブジェクトをまず「==」で比較するため，String オブジェクトがまとめられていれば文字ごとの逐次比較が不要になるからである．

StringGC では，処理中に String オブジェクト内のフィールド，具体的には value や offset が書き換えられることがある．そのため，Java プログラムがこれらのフィールドにアクセスしている途中で StringGC が起こらないように注意が必要である．幸い，String クラスは Java 処理系が提供する final クラスであり，内部フィールドにアクセスできるのは String ク

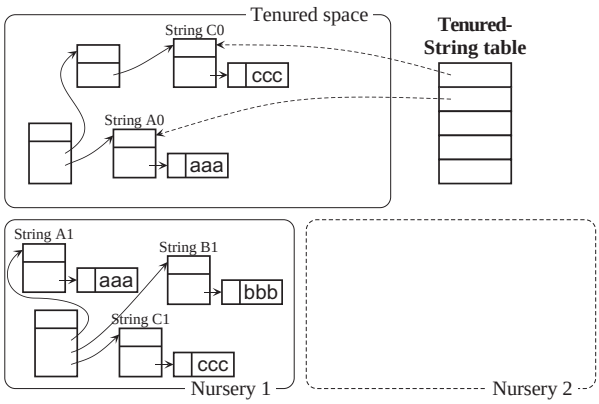
ラスのメソッドか処理系自身に限定されるため，対処は比較的容易である．たとえば，String クラス内のメソッドではフィールドアクセスをなるべくまとめ，その途中では GC セーフポイントを設けないようにして StringGC が起きないようにする方法が考えられる．他には，offset フィールドの値が変わる場合は String オブジェクトや char 配列をまとめないという方法でも対処できる．

3.4 現実的プロトタイプ

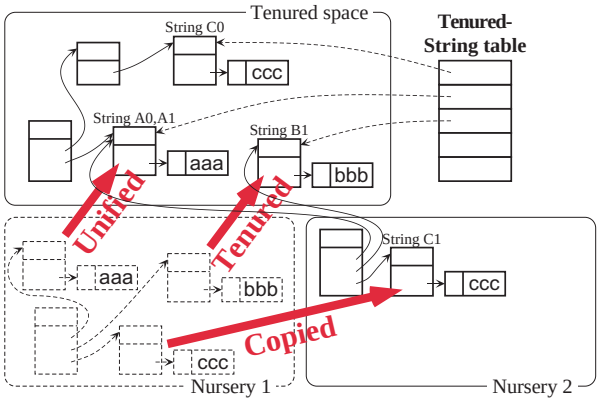
3.1 節で述べた StringGC の各ステップを完全に適用すれば，2 章で示した文字列メモリの無駄をほぼ完全に取り除くことができる．しかし，表 1 からわかるように，大規模 Java アプリケーションではヒープ中の String オブジェクトの数が 10 万個に及ぶこともあり，これらをすべて一覧表で管理しチェックすると，処理が非常に遅くなることが懸念される．そこで，プロトタイプ実装にあたり，処理系の性能を落とさずに，それなりの効果が得られる「現実的」な StringGC 手法を検討する．

表 1 の結果によれば，文字列メモリの無駄の 90%以上は文字列の重複によるものである．そこで，同じ内容の String オブジェクトをまとめることにフォーカスする．もっともナイーブな実装は，String クラスのコンストラクタを修正し，同じ内容の String オブジェクトがすでにあった場合にはそれを返すようにすることである．しかし，同じ内容の String オブジェクトの検索にはコストがかかるため，一時的にしか使用されない String オブジェクトにも適用してしまうと，プログラムの実行自体が遅くなってしまう可能性が高い．性能を落とさず StringGC の効果を得るためには，ある程度「長生き」な String オブジェクトについてのみこの処理が行われるようにしたい．

そこで我々は，世代別 GC [11] と組み合わせ，十分長生きした String オブジェクトが「殿堂入り（tenuring）」する時点で，同じ内容のものをまとめてしまう方式を考案した．性能評価のため，この現実的プロトタイプを 2.2 節の評価に用いた Java VM に実装した．実装したプロトタイプでは，殿堂（tenure）にある String オブジェクトはすべて，「殿堂入り String 一覧表」に登録されている．世代別 GC が長生きした String オブジェクトを殿堂に移動する際にこの表を検索し，同じ内容のものがすでにあった場合はその String オブジェクトにまとめてしまう．なかった場合は通常通り殿堂に移動し，表に登録する．なお，一覧表に登録されている String が他のどこからも参照されなくなった場合，殿堂を GC する際に回収され登録も削除される．



(a) GC 前



(b) GC 後

図 5: StringGC プロトタイプの動作

図 5 はこのプロトタイプの動作例で，若いオブジェクト用の領域（Nursery 1）がいっぱいになり GC が行われた状況を示している．長生きした String A1 が，殿堂入りの際に同じ内容の A0 にまとめられている．一方，B1 は同じ内容のものがないので，殿堂入りの際に表に新規登録されている．C1 は殿堂内に同じ内容のものがあるが，十分長生きしていないため新領域（Nursery 2）にそのまま移動されている．移動もしくはまとめられたオブジェクトへの参照は，適切に修正されている．

4 性能評価

3.4 節で説明したプロトタイプ実装を用いて，いくつかの Java アプリケーションについて StringGC の効果を測定した．測定はすべて，2 個の 3.06 GHz

```
 1 class MicroBench {
 2  static String[] doCreate(int dupRatio) {
 3   String[] strs = new String[1000000];
 4   int n = 0, dupCount = 1000000 * dupRatio/100
 5   for (int i = 0; i < dupCount; i++)
 6    strs[i] = "STR_"+n;     //"STR_0"
 7   for (int i = dupCount; i < 1000000; i++)
 8    strs[i] = "STR_"+(++n);//"STR_1","STR_2",..
 9   return strs;
10  }
11  static int doCompare(String[] strs) {
12   String str0 = "STR_0";
13   int dupCount = 0;
14   for (int i = 0; i < 1000000; i++)
15    if (strs[i].equals(str0)) dupCount++;
16   return dupCount*100 / 1000000;//==dupRatio
17  }
18       :
19 }
```

図 6: StringGC 用マイクロベンチマーク

Xeon プロセッサと 4 GB のメモリを搭載し Red Hat Enterprise Linux 3 AS オペレーティングシステムが動作している PC 上で行ったものである．

4.1 マイクロベンチマーク

まず，StringGC の特性を確認するためにマイクロベンチマークを行った．用いたプログラムは図 6 に示したものである．このプログラムの `doCreate()` メソッドは，`StringBuffer` を経由して 100 万個の `String` オブジェクトを生成する．このうち，引数 `dupRatio` パーセント分は同じ内容（`"STR_0"`）になっている．

このプログラムを，StringGC あり/なしの Java VM 上で様々な `dupRatio` について実行し，`doCreate()` メソッド終了直後のヒープ状況を調査したのが図 7 のグラフである．上はライブなオブジェクト数，下はその使用ヒープサイズを示している．StringGC なしの場合，`dupRatio` にかかわらずほぼ一定量のヒープが消費されているが，StringGC ありの場合，`dupRatio` が高くなるにつれてヒープ内のオブジェクト数，使用バイト数とも減っていることがわかる．削減される率はほぼ `dupRatio` と一致しており，StringGC の同一内容 `String` をまとめる処理がうまく機能していることがわかる．なお，`dupRatio` が 70%のあたりから削減率が下がっているのは，「若い」`String` オブジェクトが殿堂入りしておらずまとめられていないためと考えられる．

図 8 の上のグラフは，`doCreate()` メソッドの実行

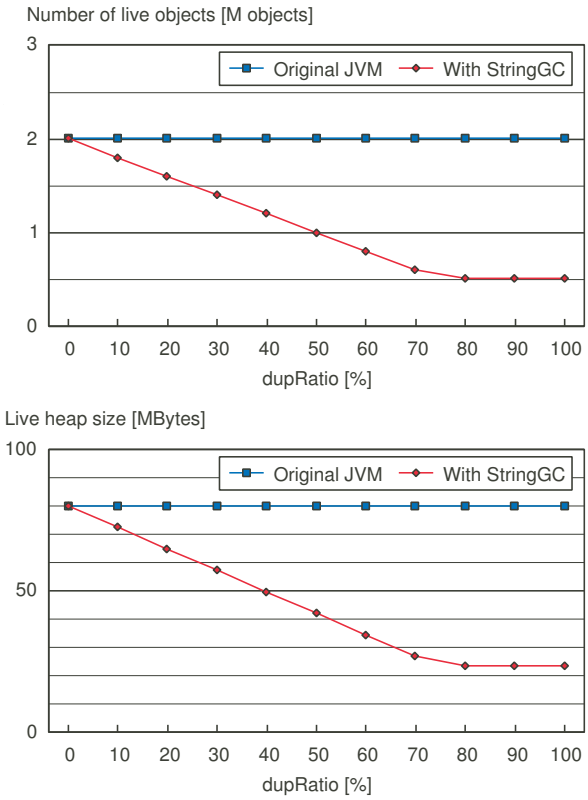


図 7: `doCreate()` 後のヒープの状況

に要した時間の測定結果である．StringGC なしの場合，実行時間は `dupRatio` にかかわらず一定である．一方 StringGC ありの場合，`dupRatio` が小さいと実行時間が遅くなってしまっている．これは，殿堂入り `String` 一覧表に 100 万個近い `String` を登録し管理するためのコストと考えられる．`dupRatio` が大きくなるにつれ表のサイズが小さくなるため，性能の改善がみられる．

さて，3.3 節でも議論したように，同一内容の `String` がまとめられると `String` の比較が高速になるという副次効果が得られる．この効果についても，マイクロベンチマークで測定を行った．測定には，図 6 の `doCompare()` メソッドを用いた．このメソッドは，`doCreate()` メソッドで生成した 100 万個の `String` を，`String.equals()` により `"STR_0"` という内容の `String` と比較するもので，`dupRatio` パーセントの比較が成功することになる．図 8 の下のグラフは，このメソッドの実行に要した時間を示したものである．StringGC なしの場合は，`dupRatio` が上がると性能が低下している．これは，`String` が同じ内容である場合は最終的に一文字ずつ比較する必要があるためである．一方，StringGC ありの場合

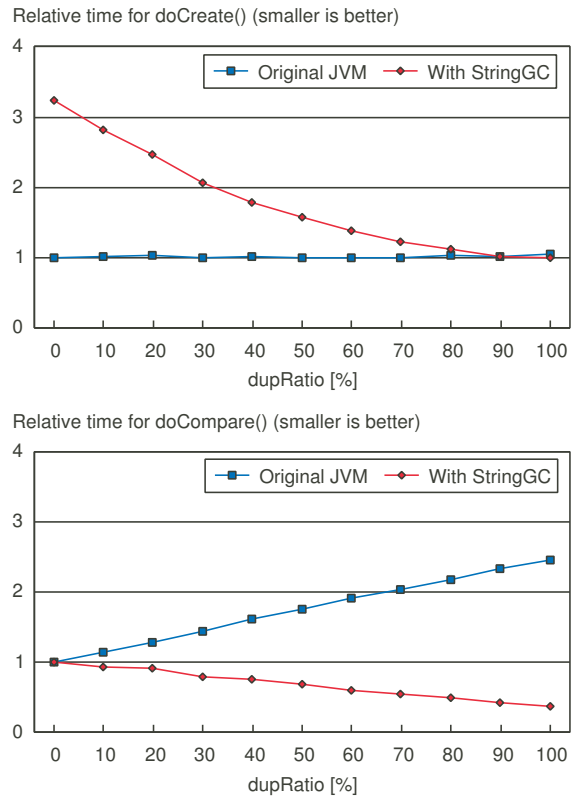


図 8: doCreate() および doCompare() の実行時間

は，同じ内容の String はまとめられてしまうため，dupRatio が高い方が処理が高速になっている．

なお，以上いずれの測定においても doCreate() および doCompare() メソッドは JIT コンパイルされたことを確認している．

4.2 マクロベンチマーク

次に，StringGC による実際の Java プログラムの性能への影響を調べるため，SPECjvm98 ベンチマーク [14] による性能測定を行った．測定結果としては，各プログラムをアプリケーションモードで個別に 5 回実行し，実行時間の最高と最低を除いた 3 つの平均を用いている．

図 9 は，各プログラムについて，StringGC なしの実行時間を 1 とした場合の StringGC ありの実行時間を示している．驚いたことに，StringGC ありの場合 _209_db の実行時間が 30%近く短くなっている．これは，図 8 下のグラフで示したように，同一内容の String をまとめる処理によって，このベンチマーク内で多く行われる文字列の比較が高速化されたためだと考えられる．測定した環境では，このベンチマークの 1 回分の実行で約 8 万個の String が殿堂

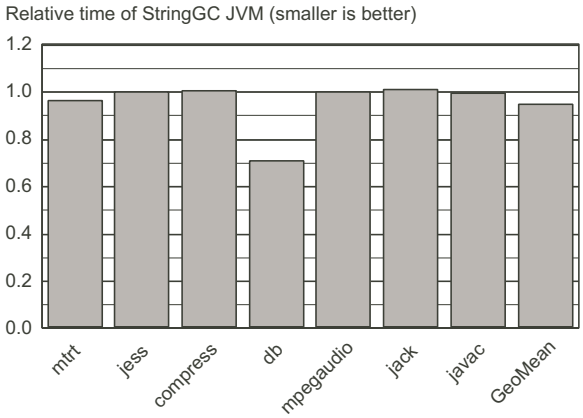


図 9: SPECjvm98 ベンチマークの相対実行時間

入りし，そのうち約 5 万個は他の String にまとめられていた．また，String.equals() は 146 万回ほど行われていた．ほかには，_227_mtrt でも 3.6%性能が向上している．いずれにせよ，StringGC ありの場合でも前節のマイクロベンチマークのような性能低下は観測されなかった．

最後に，StringGC の本来の目的であるヒープメモリの削減効果について，大規模 Java アプリケーションを用いて測定を行った．2.2 節と同様に「Trade6」と「Tuscany」について，調査ポイントでのライブオブジェクトを解析した結果が表 2 である．Trade6 では，StringGC により String オブジェクトの 38.7%，char 配列の 32.0%が削減できた．これにより，ライブヒープのサイズは 9.2%減少した．Tuscany では効果がより顕著で，String オブジェクトの 52.9%，char 配列の 46.3%を削減し，ライブヒープを 15.0%減らすことができた．この結果を表 1 と比較すると，実装した StringGC プロトタイプでは，文字列メモリの無駄の約 90%を取り除けたことになる．

5 関連研究

本章では，StringGC に関連する手法や研究をあげ，比較を行う．

Java において文字列メモリの無駄を減らす手段としては，String.intern() により文字列を「インターン」する方法がある．このメソッドは，対象 String オブジェクトと同じ内容の String オブジェクトを生成し，Java VM の管理する「String テーブル」に登録する．もし同じ内容の String オブジェクトがすでに登録されていた場合は，それが使用される．返り値はテーブルに登録された String オブジェ

測定項目	Trade6 数	Trade6 サイズ	Tuscany 数	Tuscany サイズ
元の Java VM				
全ライブオブジェクト	621,463	33,938 KB	281,973	15,332 KB
- `String` オブジェクト	106,957	2,995 KB	57,922	1,622 KB
- `char` 配列	97,306	9,641 KB	49,150	4,470 KB
StringGC あり				
全ライブオブジェクト	547,999	30,824 KB	228,379	13,026 KB
- `String` オブジェクト	65,584	1,836 KB	27,306	765 KB
- `char` 配列	66,149	7,662 KB	26,370	3,032 KB
削減できた量	72,530	3,137 KB	53,396	2,295 KB
- 元のヒープに対する割合	11.7%	9.2%	18.9%	15.0%
- 削減可能な量に対する割合	89.6%	91.5%	86.8%	88.5%

表 2: StringGC による，実際の Java アプリケーションのメモリ削減効果

クトになるので，同じ内容の場合は必ず同じ `String` オブジェクトになり，重複による無駄を取り除くことができる．しかし，インターン処理はアプリケーションにおいて明示的に行わなければならない．Java プログラミングでは，インターン処理は，`String` を「`==`」で比較可能にするために行うのが一般的であり，メモリ使用効率を上げる目的にはあまり使用されていない．

`String` オブジェクトの生成時に，システムが強制的にインターン処理を行う方式も考えられるが，一時的にしか使用されない `String` オブジェクトまで比較・複製が行われることになり，実行速度が低下してしまう．3.4 節で述べた我々の StringGC プロトタイプは，長生きした `String` オブジェクトに対してだけインターン処理を強制的に行うことで，この問題を解決したものと考えることもできる．

プログラム中の文字列リテラル（定数）の重複を取り除くことは，いろいろなシステムで行われている．Java では `String` リテラルは生成時にインターン処理されると規定されている [7] し，C 用にはプログラム内の文字列を抽出してまとめるための `xstr(1)` というツールなどがある．しかし，これらの機構はプログラムが実行中に動的に生成する文字列については働かない．Java の場合，文字列リテラルをまとめても多数の重複文字列が残っていることは，表 1 に示したとおりである．

Java プログラムがどのようにヒープを利用しているかについては，Dieckmann と Hölzle が SPECjvm98 ベンチマークについて広範な調査を行っている [6]．しかし，本論文で指摘した文字列メモリの無駄は調査対象になっていない．Marinov と O'Callahan は，プロファイリングにより同一内容の Java オブジェクトを発見する「Object Equality Profiling（OEP）」を提案している [12]．OEP により，実際の Java プログラム中には `String` に限らず同一内容のオブジェクトが多数存在することが発見されているが，それをまとめるためにはプログラムの修正が必要である．

6 まとめと今後の課題

本論文は，Java 処理系における文字列メモリの無駄について調査し，それを減らすための「文字列ガーベージコレクション（StringGC）」の提案・実装・評価を行った．

最近の Java アプリケーションでは，XML 処理やデータベースアクセスのために大量の文字列データが使われるため，文字列メモリの無駄が無視できないものになってきている．StringGC はこの無駄を取り除くための機構で，実装したプロトタイプでは，世代別 GC の殿堂入り処理の際に同一内容の `String` をまとめることでこれを実現している．

IBM の商用 Java 処理系である J9 Java VM を用いて StringGC プロトタイプの効果を測定したところ，実アプリケーションにおいて文字列メモリの無駄の 90%以上を取り除くことができ，ライブヒープのサイズを最大 15%減らすことができた．

今後の課題としては，StringGC の実行オーバーヘッドの減少，3.2 節で述べたいくつかの拡張を含

む，より完全な StringGC の実装，Java アプリケーションの実行中の動的なヒープ状態の調査などを考えている．

謝辞

普段より有用な意見をいただいている，IBM 東京基礎研究所・システムズグループの皆様に感謝します．

参考文献

[1] The Apache Software Foundation. Apache Harmony.
http://harmony.apache.org/

[2] The Apache Software Foundation. Apache Tomcat.
http://tomcat.apache.org/

[3] The Apache Software Foundation. Apache Tuscany.
http://incubator.apache.org/tuscany/

[4] C. Bailey. Java Technology, IBM Style: Introduction to the IBM Developer Kit: An overview of the new functions and features in the IBM implementation of Java 5.0, 2006.
http://www.ibm.com/developerworks/java/library/j-ibmjava1.html

[5] B. Corrie. Java Technology, IBM Style: Class Sharing: The Shared Classes feature helps reduces memory footprint and improves startup performance, 2006.
http://www.ibm.com/developerworks/java/library/j-ibmjava4/

[6] S. Dieckmann and U. Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmark. *Proc. ECOOP '99*, 92–115, 1999.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*, Addison Wesley, 2005.

[8] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java Just-In-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. *Proc. USENIX VM '04*, 151–162, 2004.

[9] IBM Corporation. IBM Trade Performance Benchmark.
https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6

[10] IBM Corporation. WebSphere Application Server: Product Overview.
http://www.ibm.com/software/webservers/appserv/was/

[11] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley, 1996.

[12] D. Marinov and R. O'Callahan. Object Equality Profiling. *Proc. OOPSLA '03*, 313–325, 2003.

[13] Open SOA. Service Component Architecture Home.
http://osoa.org/display/Main/Service+Component+Architecture+Home

[14] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks.
http://www.spec.org/osg/jvm98/

[15] Sun Microsystems. Class Data Sharing, 2004.
http://java.sun.com/j2se/1.5.0/docs/guide/vm/class-data-sharing.html