一种针对 Java 中字符串的内存管理方案

彭 文
(中国科学技术大学 计算机科学与技术学院,安徽 合肥 230027)

摘 要：提出一种针对 Java 中字符串的内存管理方案。该方案分析了字符串操作接口的行为特征,利用编译时指令插桩技术在字符串操作接口调用点插入回收指令,以主动回收无用字符串占用的堆空间,可以有效提高活跃堆空间的利用率,同时减轻垃圾收集的负担,从而达到改善 Java 虚拟机性能的效果。

关键词：Java 字符串;操作接口;指令插桩;内存管理
中图分类号：TP312 **文献标识码**：A **文章编号**：1674-7720(2011)17-0013-03

Memory management for Java strings

Peng Wen
(School of Computer Science & Technology, University of Science & Technology of China, Hefei 230027, China)

Abstract: This paper presents a kind of memory management method for Java strings. In particular, it analyzes the behaviors of string operation methods, and inserts reclaim instructions after the call site methods with the help of inst instrument algorithm, then reclaims the useless strings in the live heap. This method can effectively make great use of heap space and improve the performance of Java virtual machine.

Key words: Java strings; operation interface; inst instrument; memory management

Java[1]语言为字符串操作提供了丰富的支持,它将字符串封装在三个类中并提供多种字符串操作接口。在 Java 应用程序中,由于对字符串的使用量比较高,从而使得其需要消耗较大的堆空间。例如在 J2EE 应用服务器运行过程中,约 40%的活跃堆空间被用来保存字符串数据[2]。

通过对 Java 中字符串操作接口的分析可以发现,随着这些操作的运行会产生较多的无用字符串,它们不再被 Java 类封装并且也不被任何变量引用。这些无用字符串数据将一直停留在活跃堆中,直到 Java 虚拟机启动垃圾收集将其回收。而由于字符串数据具有单个对象占用空间较小但总体数量很大的特征,大量的无用字符串数据不仅会影响堆空间的利用率,并且对 Java 虚拟机垃圾收集的性能有较大影响。

当前对 Java 中字符串的内存管理优化方案主要关注于字符串的使用效率上,如消除常量重复、延迟分配等技术[2],通过修改 Java 虚拟机对字符串分配回收的支持来提高堆中字符串的使用效率。然而这些方案无法处理堆中已经成为无用字符串的数据,只能等待垃圾收集来处理。

近期编译时的独立对象回收策略[3]则专注于在编译阶段对应用程序做分析并插入回收指令以回收无用对象空间,但是该方案对 Java 库函数只做保守分析从而无法回收这些无用字符串。为此,本文从对字符串操作接口的分析出发,识别各类操作对字符串的改变情况以利用独立对象回收策略中的指令插桩技术来主动回收无用字符串对象,以提高堆空间的利用率、减低垃圾回收的负担、改善 Java 虚拟机的性能。

1 Java 中字符串的支持与分析

1.1 Java 中字符串的支持

Java 语言将字符串的表示和操作都封装在 StringBuilder、StringBuffer 和 String 三个类中。其中前两个类指向的字符串是可变的,String 类指向的字符串是不变的。这三个类的内部结构基本上一致,以 StringBuilder 为例,StringBuilder 在 Java 中的结构如图 1 所示。

从图 1 可以看出,字符串数据由 StringBuilder 对象

```
Public final class StringBuilder{
  private char[]value;              //内容 char 数组
  private int count;                //value 数组的长度
  private int hashCode;
}
```

图 1 StringBuilder 的内部结构

指向的 value 域保存，在内存空间上反映为两个对象：StringBuilder 对象通过 value 域指向字符串对象。由于该类提供常用的可变字符串操作接口且相对另一个类 StringBuffer 具有较高的执行效率，对字符串数据的操作在 Java 虚拟机中一般会将其转换为 StringBuilder 对象再做处理。下面以一个语句示例来说明这一点：

String s=new String('aa'+'bb'+'cc');

该语句的语义是将三个字符串连接在一起并生成一个 String 对象，在 Java 语言的源程序级别上不会出现 StringBuilder 对象，但是经过编译器优化之后，这条语句实际被翻译为下面的字节码形式(为简化描述，本文以源语言来表示字节码的操作)：

```
StringBuilder t=new StringBuilder('aa');
t.append('bb');
t.append('cc');
String s=t.toString();
```

即 Java 编译器会首先创建一个 StringBuilder 对象，完成字符串的连接工作之后再将其转变为 String 对象。由于类似于这种情况的字符串操作较多地出现在输出方法和字符串创建方法中，所以可推断出 StringBuilder 有着较大的使用频率，故将以其为代表分析其提供的接口对字符串的影响。

1.2 无用字符串的产生

在上节的示例中，StringBuilder 类提供的 append()接口将会改变 value 域所指向的字符串，其做法是：新建长度为连接后字符串长度之和的字符数组，分段复制之后使其成为 value 域指向的新数组，而 value 域指向的原数组将被丢弃成为无用字符串。

图 2 为示例语句中 append()接口引起的 value 域指向字符串变化图。

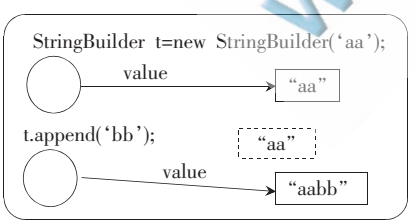

图 2 append()对字符数组操作示例

从图 2 可以看出，在 append()接口执行过后，对象的字符数组将指向新建的字符串'aabb'，原有字符串'aa'将不被任何变量指向而成为无用字符串。

由于 StringBuilder 类的 value 域指向的字符串是可变的，在其提供的接口中存在大量类似 append()可能对 value 域做出改变的接口，如 insert()、replace()等。而在 Java 虚拟机对这些接口的调用频率较多，表 1 是基准测试程序 Jolden[4]的 4 个子程序中字符串操作接口调用次数以及可能对 value 域做出改变的接口调用次数对比。

表 1 字符串操作接口调用次数统计

测试程序	字符串操作接口调用次数	可能对 value 域做出改变的接口调用次数	比例/%
BH	706	319	45.2
TSP	301	68	22.6
Power	473	216	45.7
Health	379	136	35.9

由表 1 可以看出，可能对 value 域做出改变的调用次数占字符串操作接口调用次数的 22.6%~45.7%，占有不可忽视的比例。下面将深入分析这些可能改变 value 域的操作接口的具体实现。

1.3 字符串操作接口分析

可能对 value 域做出改变的操作接口有一个共同点，即 this 对象不会发生变化，只是其 value 域指向一个新建的字符串。对字符串的操作接口做深入分析后可知，在 append()等可能改变 value 域指向的操作接口的实现中，存在两条改变分支：例如在接口 append(s)中，如果 s 为 null 或者 s 的 value 域指向一个空字符串，则该接口不会改变 this 对象的 value 域指向；否则才会新建一个字符串以被 this 对象的 value 域指向。

可以将这两条改变分支表现为下面的形式：

分支 1：不做任何改变。

分支 2：新建字符串，使其被 this 对象的 value 域指向，原有字符串成为无用字符串。

下面将给出根据本节的分析给出的无用字符串回收方案。

2 字符串的回收方案

对无用字符串的回收存在两个难点：(1)不可深入改变 Java 的库函数实现。因为回收方案需要具有较强的通用性和灵活性；(2) 由于操作接口具体实现中对 value 域的改变存在分支，并且只能在应用程序的运行阶段判断究竟执行的是哪个分支。

本文采用独立对象显式回收策略中的指令插桩技术来解决上述两个难题：在可能发生改变的字符串操作接口调用点处插入判定语句来对操作接口执行的分支做判断，然后根据结果来实施字符串的回收方案。由于这些语句都插桩在用户程序中，不会改变 Java 库函数的实现，而且这些语句会随着字符串操作接口的执行而执行，所收集的信息属于运行时信息，故可以很好地判断运行时分支的情况。

由于两条分支的不同之处表现为操作接口执行完

毕之后，this 对象的 value 域指向是否发生了变化，故可以采取接口调用前后 value 域比较的方式来判断具体执行的分支。本文使用指令插桩技术，在 Java 虚拟机重编译 Java 字节码时对其做指令插桩工作，其处理流程为：

(1) 在 Java 虚拟机处理应用程序指令时判断其是否为可能引起字符串变化的操作接口调用指令。

(2)如果是则实施步骤(3)~(5)的指令插桩工作。

(3)在调用指令之前插入 this 对象的 value 域引用保存指令。

(4)在调用指令之后插入 this 对象的 value 域引用保存指令。

(5)安插两个引用的对比指令，如果不同，则插入回收指令以回收调用之前保存的引用；否则将不做处理。

本方案用到了独立对象回收技术中的回收指令，需要在 Java 虚拟机的内存管理模块支持这个回收指令。由于对回收指令的支持对原有的分配和回收方案影响很小，故其实现较简单并且具有一定的通用性。

以图 3 为例来说明本文的字符串回收方案。由于该方案处理的为 Java 字节码，为了方便理解，将以实际代码的形式体现：由图 3 可看出，在调用点前后加入了 value 域引用保存指令记录了调用点执行前后的 value 域的引用信息，然后将两者做对比处理来判断调用点是否对 value 域的引用做出了改变，如果引用信息有了变化，则之前的 value 域引用成为了无用字符串，可以插入回收指令将其回收。

```
1. StringBuilder t=new StringBuilder('aa');
2. vp=load(s.value);
3. t.append('bb');
4. vo=load(t.value);
5. if(vp! =vo)
6. free(up)
7.……
8. t.append('cc')
9. String s=t.to.String();
```

图 3 无用字符串回收示例

可以将该无用字符串回收方案应用到其他可能对对象内部 value 域指向的字符串做出改变的接口调用点，即可以在运行时回收由这些接口运行而出现的无用字符串数据。

3 实验结果及分析

在 Apache 的开源 Java SE (Standard Edition) 平台 Harmony[5]上实现了针对字符串的回收方案，并做了相关的实验测试。测试平台的操作系统是 Windows 7 Ultimate，CPU 为 Intel Pentium Dual E2200，主频为 2.2 GHz，内存为 2 GB。测试用例为 Jolden 中的 4 个基准程序。

在实现了无用字符串的显式回收之后，可以在运行中主动回收一些无用字符串以提高活跃堆空间的利用率和降低 Java 虚拟机垃圾回收的开销。表 2 给出了主动回收的无用字符串大小和总分配大小的比较情况。

表 2 无用字符串主动回收统计

测试程序	无用字符串主动回收大小/MB	程序总分配空间/MB	比例/%
BH	10	70	14
TSP	2.5	53	5
Power	5	27	18
Health	5.5	71	8

由表 2 可以看出，本文的回收方案主动回收的无用字符串占应用程序总分配空间的 5%~18%，对堆空间的利用率有较大的提升。对无用字符串的主动回收也带来了 Java 虚拟机的性能提升，因为可以减轻垃圾收集的负担。表 3 给出了实现无用字符串回收方案前后测试程序在 Java 虚拟机中的执行时间对比。

表 3 无用字符串主动回收统计

测试程序	采用了主动回收的执行时间/ms	未采用主动回收的执行时间/ms	比例/%
BH	1 254	1 321	95
TSP	4 235	4 278	99
Power	1 711	1 821	94
Health	3 383	3 417	96

由表 3 可以看出，经过对无用字符串的主动回收处理，Java 虚拟机对应用程序的执行效率也有了改善，分别减少了 1%~5%。这说明无用字符串的回收可以提升 Java 虚拟机的性能。

本文提出一种对 Java 应用程序中无用字符串进行显式回收的方案，以指令插桩的形式收集应用程序运行时信息并主动回收堆中无用字符串，以提高堆空间的利用率。实验结果表明，该方案可以有效提高 Java 虚拟机的性能和 Java 应用程序的执行效率。

参考文献

[1] The Java language specification. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html. 2011.

[2] KAWACHIYA K, OGATA K, ONODERA T. 2008 Analysis and reduction of memory inefficiencies in Java strings[C]. In Proceedings of the 23rd ACM SIGPLAN Conference on Object –oriented Programming Systems Languages and Applications, Nashville, TN, USA, 2008:385–402.

[3] SAMUEL Z, GUYER F M: A static analysis for automatic individual object reclamation [C]. Proceedings of the 2006 ACM SIGPLAN Conference on programming language design and implementation, 2006: 264–375.

[4] CAHOON B, MCKINLEY K S. Jolden Benchmarks [CP/DK]. ftp://ftp.cs.umass.edu/pub/osl/benchmarks/jolden.tar.gz.2002.

[5] Apache software foundation. Harmony–Open Source Java SE implementation.http://harmony.apache.org/index.html.2003–2009.

(收稿日期：2011-05-12)

作者简介：

彭文，男，1983 年生，硕士研究生，主要研究方向：程序分析技术。