Rethinking Java Strings

Paolo Boldi   Sebastiano Vigna
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano, Italy

Abstract

The Java string classes, `String` and `StringBuffer`, lie at the extremes of a spectrum (immutable, reference-based and mutable, content-based). Motivated by data-intensive text applications, we propose a new string class, `MutableString`, which tries to embody the best of both approaches.[1]

1 Introduction

The Java string classes, `String` and `StringBuffer`, lie at the extremes of a spectrum (immutable, reference-based and mutable, content-based).

However, in several applications this dichotomy results in inefficient object handling, and in particular in the creation of many useless temporary objects. In very large data applications, with millions of alive objects, the cost of this inefficiency may be exceedingly high.

Another problem raises with a typical string usage pattern; in this common scenario, you start with a mutable string object, of a yet-unknown length, and append, delete, insert and substitute characters; at a certain point in time, you end up with a string that will not be changed thereafter, and you would like to "freeze" its state.

To replicate this scenario using the standard Java string classes, you will most probably use a `StringBuffer` in the first phase, and then turn it into a `String`, by using the `toString` method of `StringBuffer`. Unfortunately, the first phase will be severely slowed down by the synchronisation of `StringBuffer` methods, whereas the majority of applications will not need synchronisation at all (or will accommodate their synchronisation needs at a higher level). Moreover, turning the `StringBuffer` into a `String` implies the creation of a new object.

Of course, one might simply decide not to turn the `StringBuffer` into a `String`, but this makes it impossible to use it in the same optimised way as an immutable string; even worse, it is impossible to use a `StringBuffer` in a collection, as it does not override the `equals()` method provided by `Object`.

This dissatisfaction with the behaviour of `String` and `StringBuffer` is well known in the Java community. For instance, the Altavista crawler, Mercator [5], has been written in Java, but the authors admit that one of their first steps was rewriting the standard Java string classes. The authors have also experienced similar troubles when writing their web crawler, UbiCrawler [2], and later when indexing its results.

Sun itself is very aware of the problem: one of the suggestions given to people struggling to improve application performance reads as follows [6]:

> 11.1.3.14 Making immutable objects mutable like using StringBuffers
>
> Immutable objects serve a very good purpose but might not be good for garbage collection since any change to them would destroy the current object and create a new objects i.e., more garbage. Immutables by description, are objects which change very little overtime. But a basic

---

[1] `MutableString` is distributed as free software under the GNU Lesser General Public License within the MG4J project (`http://mg4j.dsi.unimi.it/`).

object like String is immutable, and String is used everywhere. One way to bring down the number of objects created would be to use something like StringBuffers instead of Strings when String manipulation is needed.

For example, consider the following typical situation that exemplifies the scenario discussed above: suppose you want to count the number of occurrences of each word contained in a set of documents. The number of words may get very large (say, millions), so you want to minimise object creation. Thus, while parsing each document you use a `StringBuffer` to accumulate characters and, once you've got your word, you would like to check whether this word is in a dictionary. Since `StringBuffer` does not override `equals`, you have to make it into a `String`. Now, this apparently innocuous action is really causing havoc: the new string will get the buffer backing array, and the buffer will mark itself as "shared". If the word we found must be added to the dictionary, we will insert a string containing a character array potentially much longer than needed (as `StringBuffer`'s backing arrays grow exponentially fast). Even worse, the backing array of our `StringBuffer` cannot be reused, even if we did not really insert the word into the dictionary. In fact, it cannot be reused in any case, as a call to `setLength(0)` will reallocate it to a standard length.

The purpose of this paper is to describe a new string class, `MutableString`, which tries bridge the schism between `String` and `StringBuffer`. The name was chosen so to highlight the fact that we are aimed at replacing `String`, but we want to keep the mutable nature of `StringBuffer`. Several other proposals to replace the standard Java string classes appeared in the last years, for instance [4].

It may be argued that in a lot of other situations `String` and `StringBuffer` are efficient. Nonetheless, in many power application the operations performed by these classes behind the scenes may be very harmful. The following is a simple benchmark counting the number of occurrences of words in a 200 Mbytes text file[2]:

	words/s (Linux)	words/s (Solaris 9)
String	902843	283961
MutableString	2360994	580478

Note that we do not claim that it is not possible to work around the problem and use `String` and `StringBuffer` in a better way: the problem is that to do so you must take into account non-documented behaviours that are clear only to people knowing the API source code in depth.

`MutableString`, on the contrary, has been designed so that its inner workings are extremely clear and well documented, trying to make all trade-offs between time and space explicit, and clarifying the number of new objects generated by a method call. This allows to keep under control the hidden (and potentially very heavy) burden of garbage collection; since the latter runs in time proportional to the number of *alive objects*, incrementing the frequency of garbage collection has a cost that can grow independently of all other parameters. This aspect is usually overlooked in the choice for more efficient algorithms, but it often backfires (as the Java string classes show).

As a final note, one should remark that immutable strings are much safer than mutable strings. Indeed (for obvious reaons) *any* data structure becomes much safer when it is made immutable, especially if immutability is enforced by the language type-checking mechanisms. Nonetheless, we believe that immutable types are inappropriate for data structures that undergo massive manipulation (e.g., strings), as every modification leads to the creation, and eventually to the collection, of objects.

2 Design Goals

String handling is one of the most application-dependent kinds of code, and trying to devise a string class that satisfies everybody may lead to a class satisfying nobody.

---

[2]The benchmarks were produced on a Pentium 2.4 GHz running Linux, and on a Sun Fire V880 based on SPARC 900 MHz processors and runnning Solaris 9.

The design of `MutableString` acknowledges that there are really two kinds of strings: dynamical, fast-growing, write-oriented strings and frozen, static (if not immutable), read-oriented strings. The radical departure from the standard string classes is that the two natures are incorporated in the same Java class.

The other design goal of `MutableString` has been efficiency in space and time. We have in mind applications storing and manipulating dozens of millions of strings; transformations such as upcasing/downcasing, translations, replacements etc. should be made in place, benefit from exponentially-growing backing arrays, and be implemented with efficient algorithms.

As far as memory occupation is concerned, we do not want to waste more than an integer attribute, beside the backing array (notice that a integer attribute is the minimum requirement if you plan to have exponentially-growing backing arrays). On the other hand, we do not want to give up hash code caching (at least for "frozen" strings).

Finally, `MutableString` is a non-final class: it is open to specialisations that add domain-specific features; nonetheless, for maximum efficiency all fundamental methods are final.

3   Compactness and Looseness

A mutable string may be in one of two modes, *compact* and *loose*. When in loose mode, it behaves more or less like a `StringBuffer`: its backing array gets increased exponentially as needed, so that frequent insertion/append operations can be performed efficiently; when in compact mode, the backing array gets increased on request, as before, but no attempt is made to make it larger than it is strictly needed (the rationale being that if a compact string requires modifications they will be very rare: in this case, we prefer minimum space occupancy to maximum time performance). Moreover, when a loose `MutableString` is turned into a compact `MutableString`, the backing array has really the same length as its real content.

The `equals` method for `MutableString` is based on the string content, as in `String`, and the hash code is computed accordingly; the hash code of a compact `MutableString` is cached, although no attempt is made to recompute it upon changes (in that case, it simply becomes invalid).

Note that the mode has only influence on the expected space/time performance, not on the object semantics: a `MutableString` behaves exactly in the same manner, regardless of its mode, although changes are more expensive on compact strings, and the computation of hash code is more expensive on loose strings.

`MutableString` provides two explicit methods to change mode; all remaining methods (except `ensureCapacity`) preserve the string mode, and there are two methods to test whether a `MutableString` is in a given mode or not. A mutable string created by the empty constructor or the constructor specifying a capacity is loose; all other constructors create compact mutable strings. In most cases, you can completely forget whether your mutable strings are loose or compact and get good performance.

4   Implementation Choices

Attributes. A mutable string contains two attributes only: `array`, a backing character array that contains the actual string characters, and `hashLength`, an integer value that embodies information about the mode of the string, and its length or its hash code.

More precisely, if `hashLength` is negative, the string is compact, and corresponds to the entire content of the backing array; moreover, the value of `hashLength` is the hash code of the string (-1 represents an invalid hash code). Otherwise, the string is loose, and `hashLength` represents the actual length of the string, that is, the valid prefix of the backing array. All in all:

1. `hashLength` ≥ 0: the string is loose, and `hashLength` contains its length;

2. `hashLength` $= -1$: the string is compact, and coincides with the content of `array`, but its hash code is unknown;

3. `hashLength` < -1: as above, but `hashLength` contains the hash code.

The hash code of a mutable string is defined to be the hash code of the content-equivalent string with the highest bit set. Thus, we can use the hash code, if computed, to speed up inequality tests with `String`s (albeit we are really using 31 bits only). Note that in this way the empty string has a valid hash code.

Reallocations. Backing array reallocations use a heuristic based on looseness. Whenever a new capacity is required (because of an insert or append operation), compact strings are resized to fit *exactly* the new content. On the contrary, the capacity of a loose string needing new space is computed by maximising the new length with the double of the current capacity.

The effect of this policy is that reused strings or strings created without an initial content will get large buffers quickly, but strings created with other constructors and with few changes will occupy little space and perform very well in data structures using hash codes.

Thus, reused or otherwise heavily manipulated strings may have a rapid growth, if needed, and when their state is not to change anymore you can compact them (of course, compacting a string may require reallocating the backing array).

5 Method Optimisation

One of the objectives of `MutableString` is the efficient implementation of all methods. By "efficient" we mean that it should outperform (at least in most cases) the corresponding methods of `String`.

5.1 Replacement

As an example, we discuss in detail the family of `replace` methods, which allow one to substitute all occurrences of characters from a given array with a corresponding `String` (character, `CharSequence` etc.). These methods turn out to be especially useful for all those situations where one has to quote special characters (e.g., transforming characters into HTML entities, or encoding a query in a string representing a URL).

Consider, for example, the method invocation `s.replace(ca, sa)` where `s` is a `MutableString`, `ca` is a character array and `sa` is an array of `String`s. This method should substitute every occurrence of `ca[i]` with `sa[i]` for all indices `i` less than the length[3] of `ca`.

The `replace` method must scan `s` twice: the first scan is needed to compute the length of the new string, whereas the second scan actually performs the substitutions. In both scans, when analysing a certain character `s[k]`, we should check whether the character should be replaced, in which case we should also determine what is the string to be substituted. This would require a linear scan of `ca`, and most of these scans will probably end up scanning the whole array before knowing that no substitution was really needed (i.e., that there was no `i` such that `s[k]==ca[i]`): in fact, we expect that only a small fraction of characters in the string requires a replacement.

A straightforward solution to this problem is that of creating from `ca` and `sa` a `Map` (from `Character`s to `String`s) and then using the `get` method instead of scanning the original array. From a theoretical viewpoint, this solution might largely reduce the time needed to test whether a character requires a substitution (the `get` method requires logarithmic time, in the case of a `TreeMap`, and constant expected time, in the case of a `HashMap`, under the assumption that the hash codes are evenly spread). Unfortunately, this solution requires the creation of a `Map` object and of as many `Character` objects as the length of `ca`; moreover, the cost of a method call for each test should be taken into account.

As an alternative, one could try to use a home-made tiny hash table to store the set of characters to be substituted, but dimensioning it becomes problematic: hash tables are not well suited for situations, like this, where almost all tests are expected to give a negative answer. Neither is it possible to use a table storing,

---

[3]The arrays `ca` and `sa` must have the same length; moreover, it is assumed that no character appears more than once in the array `ca`.

for each character, the index where it appears in the array, if there is one, because such a table would be exceedingly large for Unicode.

The solution adopted in `MutableString` is to use a single-hash Bloom filter [1]. In other words, we use a hash function mapping all characters to a number between 0 and 31, and keep a 32-bit mask that has a 1 in position i iff there is some substitution character that is hashed to i. Then, every time a character `s[k]` is examined, we first check whether the corresponding bit is set: if not, the character does not need to be substituted. Otherwise, we have to scan the array.

Of course, false positives are possible, but they are quite rare: more precisely, with rough but reasonable probabilistic hypotheses on the inputs and the set of characters to be replaced, and assuming that most of the characters do not need replacement, the gain in speed is easily calculated using the formulae of [1], and turns out to be

$$\frac{m^n}{m^n - (m-1)^n},$$

where m is the number of bits in the filter and n the length of `ca` (omitting, however, the overhead that is necessary to check the content of the filter). In particular, using $m = 32$ we have a speedup of 8 times using 4 characters. The speedup is still more than 2 with 20 characters. Of course, as 64-bit processors become more common, it may be reasonable to use a 64-bit mask. Benchmarks confirmed that this approach is very effective.

In our current implementation, we hash characters using simply their least-significant bits: this solution does not require any method call and can be computed in an extremely optimised way; moreover, because of the way Unicode charts are organised, we expect that natural-language documents contain characters that differ only in their least-significant bits, hence our hash function is expected to work fairly well.

5.2 Searching

A similar lightweight approach is used in the various search methods availble (e.g., `indexOf()`). There are several sophisticated text search algorithm available in the literature, but all of them require a significant setup overhead, which makes them unsuitable for a general-purpose method. For instance, building data structures (and in general using `new`) leads to methods that are unsuitable for everyday usage.

On the other hand, the brute-force approach of `String` (a double loop) is definitely not very efficient. `MutableString` uses a relaxed version of Daniel Sunday's QuickSearch [7], a variant of the Boyer–Moore algorithm. In QuickSearch, a table records for each character in the search alphabet the last occurrence of the character in the search pattern (or the pattern length, if the character does not appear). After checking whether the pattern appears in positions $t, t+1, \ldots, t+\ell-1$ (where ℓ is the length of the search pattern), we look at the character in position $t+\ell$. If corresponding entry in our table is j, we can shift the pattern by $j+1$ positions safely.

The first observation used in `MutableString` is that for short patterns it is often the case that most of the characters appearing in the text to be searched do not appear in the pattern. So a good gain can be already obtained by recording which characters of the search alphabet appear in the pattern. Finally, if the pattern is short a Bloom filter can record this information efficiently: false positives will simply slow down the algorithm, but will not make it incorrect.

It is interesting to note that even implementing fully QuickSearch in an efficient way is a nontrivial problem because of the very large size of the Unicode alphabet. Some techniques to solve this problem (which have been implemented in `TextPattern`, another class of MG4J) have been presented in [3].

6 Benchmarking

Just to give a hint of `MutableString` performance, we consider a simple task: HTML-ising a string. We start from a web page of about 100K, and iteratively replace all occurrences of `&` with `&`.

Type	calls/s (Linux)	calls/s (Solaris 9)
compact MutableString	145.56	141.64
loose MutableString	581.39	199.20
StringBuffer	36.57	12.38
unsync'd StringBuffer	37.16	12.64
String	109.41	74.12

It should be said that the test had to be run on `StringBuffer` using an external loop calling repeatedly `lastIndexOf()` and `replace()`, and that the test on `String` used regular expressions (neither class contains something corresponding to the versatile `replace()` method of `MutableString`). Note also that this test was run without causing garbage collection: the reader should thus consider the result obtained by `String` as a bit optimistic. The unsynchronised buffer case was obtained by recompiling `StringBuffer` after stripping all `synchronize` keywords.

Of course, no class can be both more compact and faster: space and time have their own laws and trade-offs. For instance, the `length()` method has to check whether the string is compact or loose, and act differently. The following benchmark gives an idea of the relative loss:

Type	Mcalls/s (Linux)	Mcalls/s (Solaris 9)
compact MutableString	285	89
loose MutableString	359	45
StringBuffer	64	8
unsync'd StringBuffer	393	223
String	393	177

It should be noted that on such a short method the results are mostly dependent on architectural issues (caches, method inlining, etc.).

7 The Price to Pay:
a String–Freedom Manifesto

In general, Java is a very flexible, well-designed, complete language, with vast and carefully structured APIs; its mind-boggling complexity pays only a small price to space/time efficiency, which makes it more and more appealing for the development of critical, large applications.

APIs are generally designed so that you can simply rewrite a class (or a bunch of classes) if you are not satisfied with its performances; so, a typical programming pattern consists in using the standard, general-purpose APIs in the first steps of development, and then, perhaps after profiling, in substituting only those classes that are critical with other, hand-tailored versions.

Of course, you have to pay a price for this. For example, suppose that `java.net.URL` performs too bad for your needs, and you want to change it with your version `foo.bar.MyURL`. You are free to do so, but what about all classes and methods in the `java.net` package that rely on `URL`? If you use them, you will probably have to rewrite more classes: there is no way out of this, since there is nothing like a URL-interface, and `URL` is `final`.

As you can expect, this price is small if you substitute, so to say, some exotic well-hidden class of the hierarchy, but it becomes high if you substitute some fundamental, pervasive class, like `String`.

So what is the price you have to pay if you want to get rid of `String`, substituting it with something else, say with `MutableString`? Of course, you can forget about string literals: every time you want to initialise a `MutableString` using a literal you have to use a `String` literal and throw it away immediately; however, there is little harm in doing so, except that the Java `String`-literal pool becomes virtually useless.

The use of the concatenation operator + becomes a trap: every time a `MutableString` is concatenated with +, it is first turned into a `String`. Finally, all I/O related methods accepting `String`s require an implicit or explicit call of `toString()`.

`MutableString` tries to lessen this burden by providing built-in methods for common I/O operations: for instance, you can use `s.println(System.out)` to print a `MutableString` to standard output.

A more reasonable solution, however, would be provided by a pervasive use in the Java core APIs of the new `CharSequence` interface. Character sequences are an abstraction of a read-only string, and are used, for instance, by the new regular expression facilities (indeed, you can split a `MutableString` on a regular expression *without creating a `String` object*, since `MutableString` implements `CharSequence`).

Every time there is a method accepting a `String`, there should also be a polymorphic version accepting a character sequence (`String` implements `CharSequence`, so to be true you do not really need two methods: however, calls through an interface are slower). Moreover, the string concatenation operator + should avoid useless calls to `toString` for objects which implement `CharSequence`. These small changes would actually make a customisation of `String` and `StringBuffer` possible. As an additional optimisation, it would be very useful if `CharSequence` required the implementation of a `getChars()` method similar to that of `String`: in this way, bulk copies would be performed much faster.

References

[1] Burton H. Bloom. Space-time trade-offs in hash coding with allowable errors. *Comm. ACM*, 13(7):422–426, 1970.

[2] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. In *Proc. AusWeb02. The Eighth Australian World Wide Web Conference*, 2002.

[3] Paolo Boldi and Sebastiano Vigna. Compact approximation of lattice functions with applications to large–alphabet text search. Technical Report 292-03, Università di Milano, Dipartimento di Scienze dell'Informazione, 2003.

[4] The Apache Software Foundation. `FastStringBuffer`. This is a class of Xalan-J (`http://xml.apache.org/xalan-j/`).

[5] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, pages 219–229, December 1999.

[6] Nagendra Nagarajayya and J. Steven Mayer. Improving java application performance and scalability by reducing garbage collection times and sizing memory using JDK 1.4.1. http://wireless.java.sun.com/midp/articles/garbagecollection2/.

[7] Daniel M. Sunday. A very fast substring search algorithm. *Comm. ACM*, 33(8):132–142, 1990.