Guide to Java String Pool

Last modified: September 23, 2018

by baeldung (/author/baeldung/)

1. Overview

The *String* object is the most used class in the Java language.

In this quick article, we'll explore the Java String Pool — **the special memory region where *Strings* are stored by the JVM**.

2. String Interning

Thanks to the immutability of *Strings* in Java, the JVM can optimize the amount of memory allocated for them by **storing only one copy of each literal *String* in the pool**. This process is called *interning*.

When we create a *String* variable and assign a value to it, the JVM searches the pool for a *String* of equal value.

If found, the Java compiler will simply return a reference to its memory address, without allocating additional memory.

If not found, it'll be added to the pool (interned) and its reference will be returned.

Let's write a small test to verify this:

```
String constantString1 = "Baeldung";
String constantString2 = "Baeldung";

assertThat(constantString1)
  .isSameAs(constantString2);
```

3. *Strings* Allocated using the Constructor

When we create a *String* via the *new* operator, the Java compiler will create a new object and store it in the heap space reserved for the JVM.

Every *String* created like this will point to a different memory region with its own address.

Let's see how this is different from the previous case:

```
String constantString = "Baeldung";
String newString = new String("Baeldung");

assertThat(constantString).isNotSameAs(newString);
```

4. *String* Literal vs *String Object*

When we create a *String* object using the *new()* operator, it always creates a new object in heap memory. On the other hand, if we create an object using *String* literal syntax e.g. "Baeldung", it may return an existing object from the String pool, if it already exists. Otherwise, it will create a new String object and put in the string pool for future re-use.

At a high level, both are the *String* objects, but the main difference comes from the point that *new()* operator always creates a new *String* object. Also, when we create a *String* using literal – it is interned.

This will be much more clear when we compare two *String* objects created using *String* literal and the *new* operator:

```
String first = "Baeldung";
String second = "Baeldung";
System.out.println(first == second); // True
```

In this example, the *String* objects will have the same reference.

Next, let's create two different objects using *new* and check that they have different references:

```
String third = new String("Baeldung");
String fourth = new String("Baeldung");
System.out.println(third == fourth); // False
```

Similarly, when we compare a *String* literal with a *String* object created using *new()* operator using the == operator, it will return *false:*

```
String fifth = "Baeldung";
String sixth = new String("Baeldung");
System.out.println(fifth == sixth); // False
```

In general, **we should use the *String* literal notation when possible**. It is easier to read and it gives the compiler a chance to optimize our code.

5. Manual Interning

We can manually intern a *String* in the Java String Pool by calling the *intern()* method on the object we want to intern.

Manually interning the *String* will store its reference in the pool, and the JVM will return this reference when needed.

Let's create a test case for this:

```
String constantString = "interned Baeldung";
String newString = new String("interned Baeldung");

assertThat(constantString).isNotSameAs(newString);

String internedString = newString.intern();

assertThat(constantString)
  .isSameAs(internedString);
```

6. Garbage Collection

Before Java 7, the JVM **placed the Java String Pool in the *PermGen* space, which has a fixed size — it can't be expanded at runtime and is not eligible for garbage collection**.

The risk of interning *Strings* in the *PermGen* (instead of the *Heap*) is that **we can get an *OutOfMemory* error** from the JVM if we intern too many *Strings*.

From Java 7 onwards, the Java String Pool is **stored in the *Heap* space, which is garbage collected** by the JVM. The advantage of this approach is the **reduced risk of *OutOfMemory* error** because unreferenced *Strings* will be removed from the pool, thereby releasing memory.

7. Performance and Optimizations

In Java 6, the only optimization we can perform is increasing the *PermGen* space during the program invocation with the *MaxPermSize* JVM option:

```
-XX:MaxPermSize=1G
```

In Java 7, we have more detailed options to examine and expand/reduce the pool size. Let's see the two options for viewing the pool size:

```
-XX:+PrintFlagsFinal
```

```
-XX:+PrintStringTableStatistics
```

The default pool size is 1009. If we want to increase the pool size, we can use the *StringTableSize* JVM option:

```
-XX:StringTableSize=4901
```

Note that increasing the pool size will consume more memory but has the advantage of reducing the time required to insert the *Strings* into the table.

8. A Note About Java 9

Until Java 8, *Strings* were internally represented as an array of characters – *char[]*, encoded in *UTF-16*, so that every character uses two bytes of memory.

With Java 9 a new representation is provided, called *Compact Strings.* This new format will choose the appropriate encoding between *char[]* and *byte[]* depending on the stored content.

Since the new *String* representation will use the *UTF-16* encoding only when necessary, the amount of *heap* memory will be significantly lower, which in turn causes less *Garbage Collector* overhead on the *JVM.*

9. Conclusion

In this guide, we showed how the JVM and the Java compiler optimize memory allocations for *String* objects via the Java String Pool.

All code samples used in the article are available over on Github (https://github.com/eugenp/tutorials/tree/master/java-strings).

Guest

JoeHx (https://hendrixjoseph.github.io/)

Interesting. I never try to worry too much about how Java handles my Strings and instead worry more about the readability of my code, but this is useful nevertheless.

+ 0 −

1 year ago

Guest

salman

So from Java 9 whether it's occupy 1 byte or 2 byte it store in byte [] from not char[] at any type Right ?

+ 0 −

1 year ago

(/author/grzegorz-author/)

Editor

Grzegorz Piwowarek (http://4comprehension.com)

Exactly

+ 0 −

1 year ago

CATEGORIES

SPRING (/CATEGORY/SPRING/)
REST (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SECURITY (/CATEGORY/SECURITY-2/)
PERSISTENCE (/CATEGORY/PERSISTENCE/)
JACKSON (/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (/CATEGORY/HTTP/)
KOTLIN (/CATEGORY/KOTLIN/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)