ABOUT BYRON KIOURTZOGLOU

Byron is a master software engineer working in the IT and Telecom domains. He is an applications developer in a wide variety of applications/services. He is currently acting as the team leader and technical architect for a proprietary service creation and integration platform for both the IT and Telecom industries in addition to a in-house big data real-time analytics solution. He is always fascinated by SOA, middleware services and mobile development. Byron is co-founder and Executive Editor at Java Code Geeks.

Java Best Practices – String performance and Exact String Matching

Posted by: Byron Kiourtzoglou in Core Java September 4th, 2010 2 Comments 980 Views

Continuing our series of articles concerning proposed practices while working with the Java programming language, we are going to talk about String performance tuning. We will focus on how to handle String creation, String alteration and String matching operations efficiently. Furthermore we will provide our own implementations of the most commonly used algorithms for Exact String Matching. Many of these algorithms can achieve far more superior performance compared to the naive approach for exact String matching available with the Java Development Kit. This article concludes with a performance comparison between the aforementioned Exact String Matching algorithms.

All discussed topics are based on use cases derived from the development of mission critical, ultra high performance production systems for the telecommunication industry.

Prior reading each section of this article it is highly recommended that you consult the relevant Java API documentation for detailed information and code samples.

All tests are performed against a Sony Vaio with the following characteristics :

- System : openSUSE 11.1 (x86_64)
- Processor (CPU) : Intel(R) Core(TM)2 Duo CPU T6670 @ 2.20GHz
- Processor Speed : 1,200.00 MHz
- Total memory (RAM) : 2.8 GB
- Java : OpenJDK 1.6.0_0 64-Bit

The following test configuration is applied :

- Concurrent worker Threads : 1
- Test repeats per worker Thread : 1000
- Overall test runs : 100

String performance tuning

Many people do not have performance in mind when they utilize String objects. Nevertheless misuse of String classes can significantly degrade the performance of an application. The most important things you should keep in mind are :

1. String objects are immutable. Once we create a String object we cannot change it. Every operation that alters a String results in the creation of at least one new object instance. For example concatenating two strings using the concatenation operator (+) results in the creation of two new objects, a temporary StringBuffer object used for the actual concatenation and the new String instance pointing to the concatenated result (the StringBuffer "toString()" operation is utilized to instantiate the resulting String). On the other hand using the String "concat(String ..)" operation to perform String concatenation will provide much better performance results compared to the concatenation operator (+) approach. Behind the scenes the String "concat(String ..)" operation utilizes the native "System.arrayCopy" operation to prepare a character array with the contents of the two concatenated Strings. Finally a new String instance is created that points to the concatenated result
2. String references are pointers to the actual String object instances. Thus using the "==" operator to compare two String instances representing identical literal contents will return "false" in case the actual String objects are different. Furthermore using the String "equals(String ..)" or String "equalsIgnoreCase(String ..)" operations to compare two String instances provides valid results but performs a

character to character comparison in case the two compared Strings are represented by different instances and their literal contents have the same length. As you can imagine the "equals(String ..)" and "equalsIgnoreCase(String ..)" operations are far more costly compared to the "==" operator which compares the strings at instance level. Nevertheless the aforementioned operations perform an instance equality check (this == obj) prior all literal content checks. In order to be able to benefit from the "this == obj" equality check when comparing String instances, you should define String values as literal strings and/or string–valued constant expressions. Doing so, your String instances will automatically be interned by the JVM. An alternate, but not favored, approach is the use of String "intern()" operation so as to intern a String manually. As clearly stated in the Java documentation for the "intern()" method,

"A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned.

Otherwise, this String object is added to the pool and a reference to this String object is returned.

It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.

All literal strings and string-valued constant expressions are interned."

Our proposed best practices when working with String classes are the following :

1. Favor the creation of literal strings and string–valued constant expressions rather than creating new String Objects using one of the String constructor methods
2. Utilizing character arrays to perform String transformation operations yields the best performance results but is the less flexible approach
3. When performing String transformation operations such as removing, inserting, replacing or appending characters, concatenating or splitting Strings use either the StringBuilder or the StringBuffer class. The StringBuilder class is introduced in Java 1.5 and is the non–synchronized counterpart of the StringBuffer class. Thus if only one Thread will be performing the String transformation operations then favor the StringBuilder class because is the best performer

Pattern First Exact String Matching

The Java language lacks fast String searching algorithms. String "indexOf(…)" and "lastIndexOf(…)" operations perform a naive search for the provided pattern against a source text. The naive search is based on the "brute force" pattern first exact string matching algorithm. The "brute force" algorithm consists in checking, at all positions in the text, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right. Nevertheless several other algorithms exist that outperform by far the "brute force" algorithm in both speed and efficiency.

Applications require two kinds of solution depending on which string, the pattern or the text, is given first. In our case, the pattern is provided beforehand, meaning that we always search for a provided pattern against an unknown text. For all applications that need Full Text Search (Text First Exact String Matching) a different set of algorithms is needed that provide indexed scanning. Apache Lucene is one of the most popular text search engine libraries that implement the latter family of algorithms. Nevertheless this article will only investigate algorithms of the first kind.

Thanks to the great work, a book named "Exact String Matching Algorithms", of ***Christian Charras*** and ***Thierry Lecroq*** from ***Laboratoire d'Informatique de Rouen – Université de Rouen***, we were able to implemented in Java the most commonly used algorithms for Exact String Matching, where the pattern is given first. The list below displays the algorithm name as provided on *Christian Charras* and *Thierry Lecroq* book followed by our algorithm implementation "code name" in parenthesis. For further information on each and every algorithm please click on the appropriate link so as to be redirected to the relevant section of the "Exact String Matching Algorithms" book.

- Brute Force algorithm (BF)
- Deterministic Finite Automaton algorithm (DFA)
- Karp-Rabin algorithm (KR)
- Shift Or algorithm (SO)
- Morris-Pratt algorithm (MP)
- Knuth-Morris-Pratt algorithm (KMP)
- Simon algorithm (SMN)
- Colussi algorithm (CLS)
- Galil-Giancarlo algorithm (GG)
- Apostolico-Crochemore algorithm (AC)
- Not So Naive algorithm (NSN)
- Boyer-Moore algorithm (BM)
- Turbo BM algorithm (TBM)
- Apostolico-Giancarlo algorithm (AG)
- Reverse Colussi algorithm (RC)
- Horspool algorithm (HP)
- Quick Search algorithm (QS)
- Tuned Boyer-Moore algorithm (BMT)
- Zhu-Takaoka algorithm (ZT)
- Berry-Ravindran algorithm (BR)
- Smith algorithm (SMT)
- Raita algorithm (RT)

- Reverse Factor algorithm (RF)
- Turbo Reverse Factor algorithm (TRF)
- Forward Dawg Matching algorithm (FDM)
- Backward Nondeterministic Dawg Matching algorithm (BNDM)
- Backward Oracle Matching algorithm (BOM)
- Galil-Seiferas algorithm (GS)
- Two Way algorithm (TW)
- String Matching on Ordered Alphabets algorithm (SMOA)
- Optimal Mismatch algorithm (OM)
- Maximal Shift algorithm (MS)
- Skip Search algorithm (SS)
- KMP Skip Search algorithm (KPMSS)

In the initial version (1.0.0) of the Exact String Search Algorithm suite, for every algorithm we have implemented three utility operations :

- compile(String pattern) – Static operation that performs all necessary preprocessing based on the provided pattern
- findAll(String source) – Returns a list containing all indexes where the search algorithm dictated a valid pattern match
- findAll(String pattern, String source) – This is a helper static operation that encapsulates the functionality of both aforementioned operations

Below is an example use of the Boyer-Moore algorithm (BM) :

Case #1

```
BM bm = BM.compile(pattern);
List<Integer> idx = bm.findAll(source);
List<Integer> idx2 = bm.findAll(source2);
List<Integer> idx3 = bm.findAll(source3);
```

Case #2

```
List<Integer> idx = BM.findAll(pattern, source);
```

In the first case we compile the pattern and perform the search in two distinct steps. This approach is appropriate when we have to search multiple source texts for the same pattern. By compiling the pattern once we can maximize performance results due to the fact that the preprocessing is usually a heavy operation. On the other hand, for once of searches, the second approach provides a more convenient API.

We must pinpoint that our provided implementation is thread safe, and that we do not currently support regular expressions in patterns.

What follows is an example performance comparison between the algorithm implementations of our Exact String Search Algorithms suite. We have searched a 1150000 character text for a 37 character phrase that intentionally did not exist, using the full alphabet size of 65535 characters. Please do not forget that this is a relative performance comparison. Performance results for the vast majority of the provided search algorithms are heavily depended on the provided text, the provided pattern and the alphabet size. Thus you should consider every performance comparison between String search algorithms only as a relative one.

At the beginning of this section we have stated that the Java language lacks fast String searching algorithms. But how slow is the standard Java naive implementation compared to our algorithm suite? To answer the aforementioned question we have implemented two methods so as to retrieve all index values of potential pattern matches using the standard Java API :

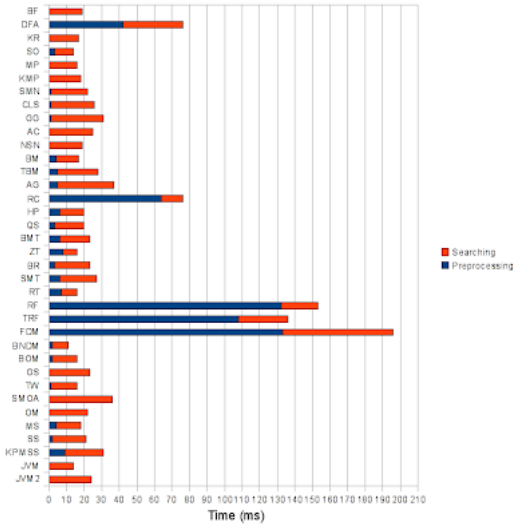
Method #1 – The indexOf() approach

```
public static List<Integer> findAll(String pattern, String source) {
  List<Integer> idx = new ArrayList&ltInteger>();
  int id = -1;
  int shift = pattern.length();
  int scnIdx = -shift;
  while (scnIdx != -1 || id == -1) {
   idx.add(scnIdx);
   id = scnIdx + shift;
   scnIdx = source.indexOf(pattern, id);
  }
  idx.remove(0);

  return idx;
 }
```

Method #2 – The Matcher find() approach

```
public static List<Integer> findAll(String pattern, String source) {
  List<Integer> idx = new ArrayList&ltInteger>();
  Pattern ptrn = Pattern.compile(pattern);
  Matcher mtch = ptrn.matcher(source);
  while(mtch.find())
   idx.add(mtch.start());

  return idx;
 }
```

Below we present the performance comparison chart between the aforementioned search algorithms



The horizontal axis represents the average time in milliseconds every algorithm needed so as to preprocess and parse the provided text. Thus lower values are better. As you can see Java naive implementation (indexOf() approach) outperforms the Java Matcher "find()" approach along with almost all our search algorithm implementations. In other words, when you are dealing with small to medium sized string searches it is preferred to implement something like the code snippet we provided above rather than use a more sophisticated String search algorithm. On the other hand when dealing with large to very large documents one of the fastest algorithms from our suite will surely come in handy!

You can download version 1.0.0 of our Exact String Search Algorithm suite binary distribution here
You can download version 1.0.0 of our Exact String Search Algorithm suite source distribution here

Happy coding

Justin

Related Articles :

- Java Best Practices – DateFormat in a Multithreading Environment
- Java Best Practices – High performance Serialization
- Java Best Practices – Vector vs ArrayList vs HashSet
- Java Best Practices – Queue battle and the Linked ConcurrentHashMap
- Java Best Practices – Char to Byte and Byte to Char conversions

Related Snippets :

- Convert String to byte array UTF encoding
- Convert String to byte array ASCII encoding
- Search String with indexOf method
- StringBuffer append method
- StringTokenizer Count Tokens
- Reverse String with StringTokenizer

Tagged with: EXACT STRING MATCHING | JAVA BEST PRACTICES | STRING

(***0*** *rating,* ***0*** *votes*)

You need to be a registered member to rate this. 2 Comments 980 Views Tweet it!

saqib

Guest

hi , great work.
can you please let me know how to run these programs.
Do i need to install maven.
I am new in this field.
Kindly let me know how can i run these algorithms.
Thanks

0 Reply 3 years ago

Mohamed Mouhoub

Guest

Hello,
Great work, many thanks for that,
I have just a comment about the good results obtained using the default java string matching.
In fact, I have read somewhere (don't have the reference for the moment, but should be easily googled) that the default java string matching/searching algorithm is based on the Knuth-Morris-Pratt algorithm, and as shown in your last figure, the results of both are pretty much the same.
That's all, and thanks again,
Best regards

0 Reply 1 year ago

Java Code Geeks
System Code Geeks
Web Code Geeks

GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial
Java Best Practices – Vector vs ArrayList vs HashSet