Was jeder Java-Entwickler über Strings wissen sollte

Bernd Müller

Fakultät Informatik
Ostfalia
Hochschule Braunschweig/Wolfenbüttel

JUG Ostfalen

4.2.2016

Vor vielen, vielen Jahren . . .

Vor vielen, vielen Jahren ...

```
public class HelloWorld {

  public static void main(String[] args) {
    System.out.println("Hello World");
  }

}
```

Vor vielen, vielen Jahren ...

```
public class HelloWorld {

   public static void main(String[] args) {
      System.out.println("Hello World");
   }

}
```

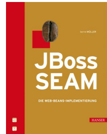
Was ist „String[]“ ?

Was ist „"Hello World"“ ?

Hat das irgend etwas
miteinander zu tun ?

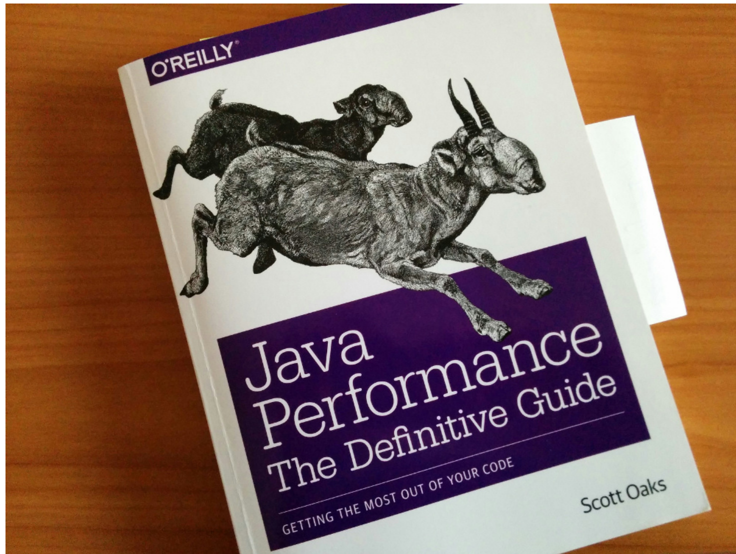
Vorstellung Referent

- Prof. Informatik (Ostfalia, HS Braunschweig/Wolfenbüttel)
- Buchautor (JSF, Seam, JPA, ...)



- Mitglied EGs JSR 344 (JSF 2.2) und JSR 338 (JPA 2.1)
- Geschäftsführer PMST GmbH
- ...

Motivation

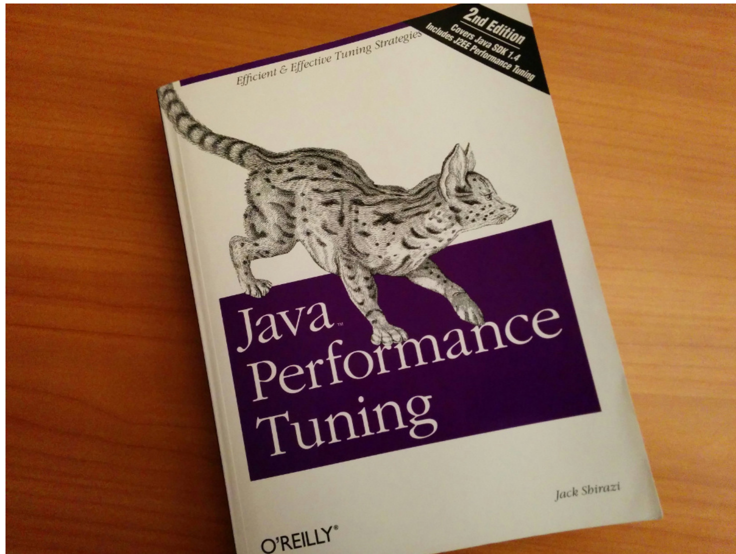

O'REILLY
Java Performance
The Definitive Guide
GETTING THE MOST OUT OF YOUR CODE
Scott Oaks

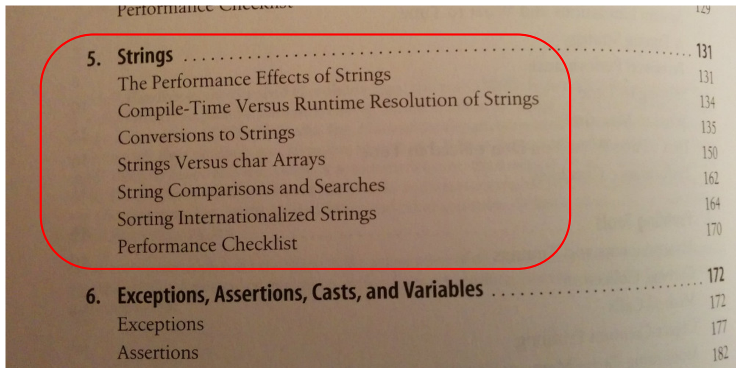
Advanced Tunings 159
Tenuring and Survivor Spaces 159
Allocating Large Objects 163
AggressiveHeap 171
Full Control Over Heap Size 173
Summary 174

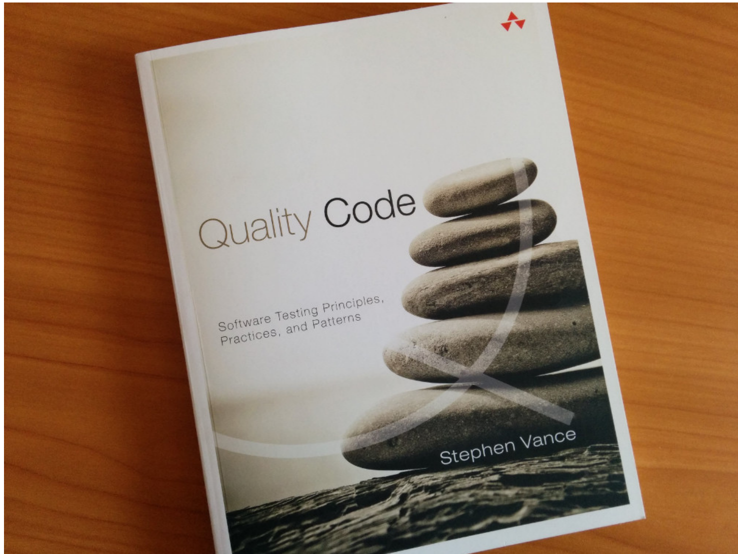
7. Heap Memory Best Practices 177
Heap Analysis 177
Heap Histograms 178
Heap Dumps 179
Out of Memory Errors 184
Using Less Memory 188
Reducing Object Size 188
Lazy Initialization 191
Immutable and Canonical Objects 196
String Interning 198

Auszug aus Abschnitt *String Interning*

„Strings are, far and away, the most common Java object; your application's heap is almost certainly filled with them."


2nd Edition
Covers Java SDK 1.4
Includes J2EE Performance Tuning
Efficient & Effective Tuning Strategies
Java
Performance
Tuning
Jack Shirazi
O'REILLY


5. Strings 131
The Performance Effects of Strings 131
Compile-Time Versus Runtime Resolution of Strings 134
Conversions to Strings 135
Strings Versus char Arrays 150
String Comparisons and Searches 162
Sorting Internationalized Strings 164
Performance Checklist 170
6. Exceptions, Assertions, Casts, and Variables 172
Exceptions 172
Assertions 177
182


Quality Code
Software Testing Principles,
Practices, and Patterns
Stephen Vance

Adjust Visibility 75
Verification by Injection 77

Chapter 7: String Handling 81

Verification by Containment 81
Verification by Pattern 83
Exact Verification by Value 85
Exact Verification with Formatted Results 88

Chapter 8: Encapsulation and Override Variations 91

Data Injection 91

String-Klassen und -Methoden

Klassen mit String-Bezug: die üblichen Verdächtigen

- java.lang.String, seit Java 1.0
  The String class represents character strings.
- java.lang.StringBuffer, seit Java 1.0
  A thread-safe, mutable sequence of characters.
- java.lang.StringBuilder, seit Java 5
  A mutable sequence of characters.
- java.util.StringTokenizer, seit Java 1.0
  The string tokenizer class allows an application to break a string into tokens.
- java.util.StringJoiner, seit Java 8
  StringJoiner is used to construct a sequence of characters separated by a delimiter ...

Klassen mit String-Bezug: die üblichen Verdächtigen

- java.lang.String, seit Java 1.0
  The String class represents character strings.
- java.lang.StringBuffer, seit Java 1.0
  A thread-safe, mutable sequence of characters.
- java.lang.StringBuilder, seit Java 5
  A mutable sequence of characters.
- java.util.StringTokenizer, seit Java 1.0
  The string tokenizer class allows an application to break a string into tokens.
- java.util.StringJoiner, seit Java 8
  StringJoiner is used to construct a sequence of characters separated by a delimiter . . .

Klassen mit String-Bezug: die üblichen Verdächtigen

- java.lang.String, seit Java 1.0
  The String class represents character strings.
- java.lang.StringBuffer, seit Java 1.0
  A thread-safe, mutable sequence of characters.
- java.lang.StringBuilder, seit Java 5
  A mutable sequence of characters.
- java.util.StringTokenizer, seit Java 1.0
  The string tokenizer class allows an application to break a string into tokens.
- java.util.StringJoiner, seit Java 8
  StringJoiner is used to construct a sequence of characters separated by a delimiter ...

Klassen mit String-Bezug: die üblichen Verdächtigen

- java.lang.String, seit Java 1.0
  The String class represents character strings.
- java.lang.StringBuffer, seit Java 1.0
  A thread-safe, mutable sequence of characters.
- java.lang.StringBuilder, seit Java 5
  A mutable sequence of characters.
- java.util.StringTokenizer, seit Java 1.0
  The string tokenizer class allows an application to break a string into tokens.
- java.util.StringJoiner, seit Java 8
  StringJoiner is used to construct a sequence of characters separated by a delimiter ...

Klassen mit String-Bezug: die üblichen Verdächtigen

- java.lang.String, seit Java 1.0
  The String class represents character strings.
- java.lang.StringBuffer, seit Java 1.0
  A thread-safe, mutable sequence of characters.
- java.lang.StringBuilder, seit Java 5
  A mutable sequence of characters.
- java.util.StringTokenizer, seit Java 1.0
  The string tokenizer class allows an application to break a string into tokens.
- java.util.StringJoiner, seit Java 8
  StringJoiner is used to construct a sequence of characters separated by a delimiter ...

Padding

- Padding in der Regel mit Apache Commons-Lang `StringUtils`: `leftPad()`, `rightPad()`

Padding

- Padding in der Regel mit Apache Commons-Lang `StringUtils`: `leftPad()`, `rightPad()`
- Man benötigt aber keine Bibliothek, ist im SDK eingebaut
- Schlüssel: `java.util.Formatter` als `printf`-Nachahmung

Padding

- Padding in der Regel mit Apache Commons-Lang `StringUtils`: `leftPad()`, `rightPad()`
- Man benötigt aber keine Bibliothek, ist im SDK eingebaut
- Schlüssel: `java.util.Formatter` als `printf`-Nachahmung
- Beispiel:

```
String.format("%1$10s", "hello")
String.format("%1$-10s", "hello")
```

Character-Codierungen

Character-Codierungen

- Historisch Java mit 16 Bit codiertem `char`-Datentyp

Character-Codierungen

- Historisch Java mit 16 Bit codiertem `char`-Datentyp
- Mit Java 5 wurde Unicode 4.0 als Character-Codierung eingeführt [CHAR1]

Character-Codierungen

- Historisch Java mit 16 Bit codiertem `char`-Datentyp
- Mit Java 5 wurde Unicode 4.0 als Character-Codierung eingeführt [CHAR1]
- Damit Codierungen mit mehr als 16 Bit möglich, die als sogenannte *Surrogate* repräsentiert werden

Java Tutorial: *Supplementary Characters as Surrogates*

„To support supplementary characters without changing the char primitive data type and causing incompatibility with previous Java programs, supplementary characters are defined by a pair of code point values that are called surrogates. The first code point is from the high surrogates range of U+D800 to U+DFBB, and the second code point is from the low surrogates range of U+DC00 to U+DFFF. For example, the Deseret character LONG I, U+10400, is defined with this pair of surrogate values: U+D801 and U+DC00.“ [CHAR2]

Character-Codierungen (cont'd)

- Beschreibung in Java-Doc java.lang.Character [CHAR4]

Character-Codierungen (cont'd)

- Beschreibung in Java-Doc java.lang.Character [CHAR4]
- Überblick
  - Java 1.4: Unicode 3.0
  - Java 5: Unicode 4.0
  - Java 6: Unicode 4.0
  - Java 7: Unicode 6.0.0
  - Java 8: Unicode 6.2.0

Character-Codierungen (cont'd)

- Beschreibung in Java-Doc java.lang.Character [CHAR4]
- Überblick
  - Java 1.4: Unicode 3.0
  - Java 5: Unicode 4.0
  - Java 6: Unicode 4.0
  - Java 7: Unicode 6.0.0
  - Java 8: Unicode 6.2.0
- Basic Encodings in lib/rt.jar, Extended Encodings in lib/charsets.jar, Dokumentation in [CHAR3]

Performanz

Performanz: `toString()`

Who Cares About toString Performance ?

- Blog von Antonio Goncalves [Gonc]

Who Cares About toString Performance ?

- Blog von Antonio Goncalves [Gonc]
- Use Case: Großer Batch mit Logging und `toString()`, o.ä.

Who Cares About toString Performance ?

- Blog von Antonio Goncalves [Gonc]
- Use Case: Großer Batch mit Logging und `toString()`, o.ä.
- Effective Java [Joshua Bloch]
  Item 10: Always override toString

Who Cares About toString Performance ?

- Blog von Antonio Goncalves [Gonc]
- Use Case: Großer Batch mit Logging und `toString()`, o.ä.
- Effective Java [Joshua Bloch]
  Item 10: Always override toString
- `toString()`-Methode durch IDE generiert oder selbst gemacht mit Alternativen

Who Cares About toString Performance ?

- Blog von Antonio Goncalves [Gonc]
- Use Case: Großer Batch mit Logging und `toString()`, o.ä.
- Effective Java [Joshua Bloch]
  Item 10: Always override toString
- `toString()`-Methode durch IDE generiert oder selbst gemacht mit Alternativen
- Null-Checks nicht vergessen

Who Cares About toString Performance ?

- Blog von Antonio Goncalves [Gonc]
- Use Case: Großer Batch mit Logging und `toString()`, o.ä.
- Effective Java [Joshua Bloch]
  Item 10: Always override toString
- `toString()`-Methode durch IDE generiert oder selbst gemacht mit Alternativen
- Null-Checks nicht vergessen
- Untersucht: JDK, Guava, CommonsLang3

Who Cares About toString Performance ?

- Blog von Antonio Goncalves [Gonc]
- Use Case: Großer Batch mit Logging und `toString()`, o.ä.
- Effective Java [Joshua Bloch]
  Item 10: Always override toString
- `toString()`-Methode durch IDE generiert oder selbst gemacht mit Alternativen
- Null-Checks nicht vergessen
- Untersucht: JDK, Guava, CommonsLang3
- Gemessen: „Average performance with Java Microbenchmarking Harness (ops/s)“

Who Cares About toString Performance ? (cont'd)

Ergebnisse

Technic	Average ops/s
String concat with +	142.075,167
String builder	141.463,438
Objects.toString	140.791,365
Guava	110.111,808
ToStringBuilder (append)	75.165,552
ToStringBuilder (reflectionToString)	34.930,630
ReflectionToStringBuilder	23.204,479

Who Cares About toString Performance? (cont'd)

Zusammenfassung:

„Today with the JVM optimisation, we can safely use the + symbol to concatenate Strings (and use Objects.toString to handle nulls). With the utility class Objects that is built-in the JDK, no need to have external frameworks to deal with null values. So, out of the box, the JDK has better performance than any other technic described in this article." [Antonio Goncalves, Gonc]

Who Cares About toString Performance? (cont'd)

Zusammenfassung:

> *„Today with the JVM optimisation, we can safely use the + symbol to concatenate Strings (and use Objects.toString to handle nulls). With the utility class Objects that is built-in the JDK, no need to have external frameworks to deal with null values. So, out of the box, the JDK has better performance than any other technic described in this article.“ [Antonio Goncalves, Gonc]*

Nachtrag: sowieso nur interessant, wenn Sie `toString()` sehr oft aufrufen

Performanz: String-Konkatenation

JDK String-Konkatenation

```java
@Benchmark
// String +
public static String concat() {
  String result = "";
  for (int i = 0; i < IT; i++) {
    result += i;
  }
  return result;
}
```

```java
@Benchmark
// StringBuffer
public static String concat() {
  StringBuffer builder = new Stri
  for (int i = 0; i < IT; i++) {
    builder.append(i);
  }
  return builder.toString();
}
```

```java
@Benchmark
// Joining Collector
public static String concat() {
  return IntStream.range(0, IT)
    .mapToObj(String::valueOf)
    .collect(Collectors.joining());
}
```

```java
@Benchmark
// StringBuilder
public static String concat() {
  StringBuilder builder = new Str
  for (int i = 0; i < IT; i++) {
    builder.append(i);
  }
  return builder.toString();
}
```

JMH Ergebnisse

Benchmark	Score	Error	Units
SC.concatWithJoiningCollector	2898,767	± 5,378	ops/s
SC.concatWithString	25,434	± 0,059	ops/s
SC.concatWithStringBuffer	5375,578	± 39,959	ops/s
SC.concatWithStringBuilder	6030,804	± 13,088	ops/s

Internte Strings

Interne Strings — Was ist das ?

Interne Strings

Moreover, a string literal always refers to the same instance of class String. This is because string literals - or, more generally, strings that are the values of constant expressions (§15.28) - are "interned" so as to share unique instances, using the method String.intern. [JLS 3.10.5 String Literals]

Interne Strings

Moreover, a string literal always refers to the same instance of class String. This is because string literals - or, more generally, strings that are the values of constant expressions (§15.28) - are "interned" so as to share unique instances, using the method String.intern. [JLS 3.10.5 String Literals]

- String-Memory-Pool mit allen internten Strings

Interne Strings

Moreover, a string literal always refers to the same instance of class String. This is because string literals - or, more generally, strings that are the values of constant expressions (§15.28) - are "interned" so as to share unique instances, using the method String.intern. [JLS 3.10.5 String Literals]

- String-Memory-Pool mit allen internten Strings
- Beim Laden einer Klasse in die VM wird geprüft, ob String-Literal schon im Pool

Interne Strings

Moreover, a string literal always refers to the same instance of class String. This is because string literals - or, more generally, strings that are the values of constant expressions (§15.28) - are "interned" so as to share unique instances, using the method String.intern. [JLS 3.10.5 String Literals]

- String-Memory-Pool mit allen internten Strings
- Beim Laden einer Klasse in die VM wird geprüft, ob String-Literal schon im Pool
- Falls ja, wird es wiederverwendet, falls nein, neu eingetragen

Internte Strings

Moreover, a string literal always refers to the same instance of class String. This is because string literals - or, more generally, strings that are the values of constant expressions (§15.28) - are "interned" so as to share unique instances, using the method String.intern. [JLS 3.10.5 String Literals]

- String-Memory-Pool mit allen internten Strings
- Beim Laden einer Klasse in die VM wird geprüft, ob String-Literal schon im Pool
- Falls ja, wird es wiederverwendet, falls nein, neu eingetragen
- String-Literale gibt es also nur einmal in einer VM

Internte String-Literale — Beispiele

```
String str1 = "Hello, World!";
String str2 = "Hello, World!";
String str3 = new String("Hello, World!"); // sinnlos

Assert.assertSame("Hello, World!", "Hello, World!");
Assert.assertSame(str1, str2);
Assert.assertSame("Hel" + "lo", "Hel" + "lo");

Assert.assertNotSame(str1 + str1, str2 + str2);
Assert.assertEquals(str1 + str1, str2 + str2);

Assert.assertSame((str1 + str1).intern(),
                  (str2 + str2).intern());

Assert.assertNotSame(str1, str3);
```

Amüsantes

Spielereien

Was ist die Ausgabe?

```
System.out.println("Hello, World!");
magic();
System.out.println("Hello, World!");
```

Spielereien

Was ist die Ausgabe?

```
System.out.println("Hello, World!");
magic();
System.out.println("Hello, World!");

private static void magic() throws Exception {
  Field field = String.class
                           .getDeclaredField("value");
  field.setAccessible(true);
  field.set("Hello, World!",
            "tricky intern".toCharArray());
}
```

Zuerst gesehen bei Arno Haase

Spielereien

Was ist die Ausgabe?

```
System.out.println("Hello, World!");
magic();
System.out.println("Hello, World!");

private static void magic() throws Exception {
  Field field = String.class
                    .getDeclaredField("value");
  field.setAccessible(true);
  field.set("Hello, World!",
            "tricky intern".toCharArray());
}
```

Bitte nicht nachmachen

Zuerst gesehen bei Arno Haase

Spielereien (cont'd)

Was ist die Ausgabe?

```
System.out.println("Hello, World!");
magic();
System.out.println("Hello, World!");

public static void magic() throws Exception {
  Field field = String.class
                          .getDeclaredField("value");
  field.setAccessible(true);
  char[] chars =  (char[]) field.get("Hello World");
  System.arraycopy("tricky intern".toCharArray(),
                   0, chars, 0, chars.length);
}
```

Spielereien (cont'd)

Ebenfalls nicht nachmachen

Was ist die Ausgabe?

```
System.out.println("Hello, World!");
magic();
System.out.println("Hello, World!");

public static void magic() throws Exception {
  Field field = String.class
                     .getDeclaredField("value");
  field.setAccessible(true);
  char[] chars = (char[]) field.get("Hello World");
  System.arraycopy("tricky intern".toCharArray(),
                   0, chars, 0, chars.length);
}
```

Nochmal Performanz

Java-Doc String.intern()

Returns a canonical representation for the string object.
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. String literals are defined in section 3.10.5 of the The Java™ Language Specification.
Returns: a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

Java-Doc String.intern()

Returns a canonical representation for the string object.
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.
String literals are defined in section 3.10.5 of the The Java™ Language Specification.
Returns: a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

Java-Doc String.intern()

Returns a canonical representation for the string object.
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.
String literals are defined in section 3.10.5 of the The Java™ Language Specification.
Returns: a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

Java-Doc String.intern()

Returns a canonical representation for the string object.
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.
String literals are defined in section 3.10.5 of the The Java™ Language Specification.
Returns: a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

Java-Doc String.intern()

Returns a canonical representation for the string object.
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. String literals are defined in section 3.10.5 of the The Java™ Language Specification.
Returns: a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

Performanz-Idee: String-Interning und Equals

„On the topic of interning strings, what about using the `intern()` method to make the programm run faster, since interned strings can be compared via the == operator? That is a popular thought, though in most cases it turns out to by a myth. The String.equals() method is pretty fast.

...

Comparing strings via the == operator is undeniably faster, but the cost of interning the string must also be taken into consideration.“
[Java Performance, Scott Oaks]

Performanz-Idee: String-Interning und Equals

„On the topic of interning strings, what about using the `intern()` method to make the programm run faster, since interned strings can be compared via the == operator? That is a popular thought, though in most cases it turns out to by a myth. The String.equals() method is pretty fast.

...

Comparing strings via the == operator is undeniably faster, but the cost of interning the string must also be taken into consideration."
[Java Performance, Scott Oaks]

Performanz-Idee: String-Interning und Equals

„On the topic of interning strings, what about using the `intern()` method to make the programm run faster, since interned strings can be compared via the == operator? That is a popular thought, though in most cases it turns out to by a myth. The String.equals() method is pretty fast.

...

Comparing strings via the == operator is undeniably faster, but the cost of interning the string must also be taken into consideration.“
[Java Performance, Scott Oaks]

Performanz-Idee: String-Interning und Equals

„On the topic of interning strings, what about using the `intern()` method to make the programm run faster, since interned strings can be compared via the == operator? That is a popular thought, though in most cases it turns out to by a myth. The String.equals() method is pretty fast.

...

Comparing strings via the == operator is undeniably faster, but the cost of interning the string must also be taken into consideration.“
[Java Performance, Scott Oaks]

Java Performance, Scott Oaks

„Like most optimizations, interning strings shouldn't be done arbitrarily, but it can be effective if there are lots of duplicate strings occupying a significant portion of the heap.“

Und nochmal Performanz

Auch bitte nicht selbst versuchen: MyString

„In compiler theory, an intrinsic function is a function available for use in a given programming language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically generated instructions for the original function call, similar to an inline function. Unlike an inline function though, the compiler has an intimate knowledge of the intrinsic function and can therefore better integrate it and optimize it for the situation.“[Wikipedia,Intr1]

Auch bitte nicht selbst versuchen: MyString

„In compiler theory, an intrinsic function is a function available for use in a given programming language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically generated instructions for the original function call, similar to an inline function. Unlike an inline function though, the compiler has an intimate knowledge of the intrinsic function and can therefore better integrate it and optimize it for the situation.“[Wikipedia,Intr1]

Auch bitte nicht selbst versuchen: MyString

„In compiler theory, an intrinsic function is a function available for use in a given programming language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically generated instructions for the original function call, similar to an inline function. Unlike an inline function though, the compiler has an intimate knowledge of the intrinsic function and can therefore better integrate it and optimize it for the situation.“[Wikipedia,Intr1]

Auch bitte nicht selbst versuchen: MyString

„In compiler theory, an intrinsic function is a function available for use in a given programming language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically generated instructions for the original function call, similar to an inline function. Unlike an inline function though, the compiler has an intimate knowledge of the intrinsic function and can therefore better integrate it and optimize it for the situation.“[Wikipedia,Intr1]

- In Java Compiler, JIT, evtl. sogar in JVM eingebaut

Auch bitte nicht selbst versuchen: MyString

„In compiler theory, an intrinsic function is a function available for use in a given programming language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically generated instructions for the original function call, similar to an inline function. Unlike an inline function though, the compiler has an intimate knowledge of the intrinsic function and can therefore better integrate it and optimize it for the situation.“[Wikipedia,Intr1]

- In Java Compiler, JIT, evtl. sogar in JVM eingebaut
- `String.equals()` ist intrinsic. `String.indexOf()`, `String.compareTo()` ebenfalls [Krystal Mo,Intr2]
- `String.intern()` ist sogar nativ implementiert

Implementierung

Der String-Pool

- Implementiert als Hashtable fester Größe

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar
- In Java 6 in Permanent Generation, 7 und höher im Heap

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar
- In Java 6 in Permanent Generation, 7 und höher im Heap
  - Daher in Java 6 OOME: PermGen space

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar
- In Java 6 in Permanent Generation, 7 und höher im Heap
  - Daher in Java 6 OOME: PermGen space
  - Java 7 und höher OOME: Java heap space

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar
- In Java 6 in Permanent Generation, 7 und höher im Heap
  - Daher in Java 6 OOME: PermGen space
  - Java 7 und höher OOME: Java heap space
- Größen:

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar
- In Java 6 in Permanent Generation, 7 und höher im Heap
  - Daher in Java 6 OOME: PermGen space
  - Java 7 und höher OOME: Java heap space
- Größen:
  - Vor Java 7u40 1009 Buckets

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar
- In Java 6 in Permanent Generation, 7 und höher im Heap
  - Daher in Java 6 OOME: PermGen space
  - Java 7 und höher OOME: Java heap space
- Größen:
  - Vor Java 7u40 1009 Buckets
  - Java 7u40 und später 60013

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar
- In Java 6 in Permanent Generation, 7 und höher im Heap
  - Daher in Java 6 OOME: PermGen space
  - Java 7 und höher OOME: Java heap space
- Größen:
  - Vor Java 7u40 1009 Buckets
  - Java 7u40 und später 60013
  - VM-Schalter:

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar
- In Java 6 in Permanent Generation, 7 und höher im Heap
  - Daher in Java 6 OOME: PermGen space
  - Java 7 und höher OOME: Java heap space
- Größen:
  - Vor Java 7u40 1009 Buckets
  - Java 7u40 und später 60013
  - VM-Schalter:
    - `-XX:StringTableSize=<value>`

Der String-Pool

- Implementiert als Hashtable fester Größe
- Bucket-Größe ab Java 6 einstellbar
- Nativ in JVM realisiert, Größe nicht veränderbar
- In Java 6 in Permanent Generation, 7 und höher im Heap
  - Daher in Java 6 OOME: PermGen space
  - Java 7 und höher OOME: Java heap space
- Größen:
  - Vor Java 7u40 1009 Buckets
  - Java 7u40 und später 60013
  - VM-Schalter:
    - `-XX:StringTableSize=<value>`
    - `-XX:+PrintStringTableStatistics` (Java 7u6 und größer, 6u32 Backport)

Beispiele -XX:+PrintStringTableStatistics

```
StringTable statistics:
Number of buckets       :    1003
Average bucket size     :      33
Variance of bucket size :      33
Std. dev. of bucket size:       6
Maximum bucket size     :      51
```

Beispiele -XX:+PrintStringTableStatistics

```
StringTable statistics:
Number of buckets         :    1003
Average bucket size       :      33
Variance of bucket size   :      33
Std. dev. of bucket size  :       6
Maximum bucket size       :      51
```

Beispiele -XX:+PrintStringTableStatistics

```
StringTable statistics:
Number of buckets          :    1003
Average bucket size        :      33
Variance of bucket size    :      33
Std. dev. of bucket size   :       6
Maximum bucket size        :      51
```

Schlecht

```
StringTable statistics:
Number of buckets          :   60013
Average bucket size        :       1
Variance of bucket size    :       1
Std. dev. of bucket size   :       1
Maximum bucket size        :      10
```

Beispiele -XX:+PrintStringTableStatistics

```
StringTable statistics:
Number of buckets          :    1003
Average bucket size        :      33
Variance of bucket size    :      33
Std. dev. of bucket size   :       6
Maximum bucket size        :      51
```

Schlecht

```
StringTable statistics:
Number of buckets          :   60013
Average bucket size        :       1
Variance of bucket size    :       1
Std. dev. of bucket size   :       1
Maximum bucket size        :      10
```

Beispiele -XX:+PrintStringTableStatistics

```
StringTable statistics:
Number of buckets          :    1003
Average bucket size        :      33
Variance of bucket size    :      33
Std. dev. of bucket size   :       6
Maximum bucket size        :      51
```

Schlecht

```
StringTable statistics:
Number of buckets          :   60013
Average bucket size        :       1
Variance of bucket size    :       1
Std. dev. of bucket size   :       1
Maximum bucket size        :      10
```

Viel besser

```
StringTable statistics:
Number of buckets          :   60013
Average bucket size        :       0
Variance of bucket size    :       0
Std. dev. of bucket size   :       1
Maximum bucket size        :       5
```

Beispiele -XX:+PrintStringTableStatistics

```
StringTable statistics:
Number of buckets          :    1003
Average bucket size        :      33
Variance of bucket size    :      33
Std. dev. of bucket size   :       6
Maximum bucket size        :      51
```

Schlecht

```
StringTable statistics:
Number of buckets          :   60013
Average bucket size        :       1
Variance of bucket size    :       1
Std. dev. of bucket size   :       1
Maximum bucket size        :      10
```

Viel besser

```
StringTable statistics:
Number of buckets          :   60013
Average bucket size        :       0
Variance of bucket size    :       0
Std. dev. of bucket size   :       1
Maximum bucket size        :       5
```

Beispiele -XX:+PrintStringTableStatistics

```
StringTable statistics:
Number of buckets       :    1003
Average bucket size     :      33
Variance of bucket size :      33
Std. dev. of bucket size:       6
Maximum bucket size     :      51
```

Schlecht

```
StringTable statistics:
Number of buckets       :   60013
Average bucket size     :       1
Variance of bucket size :       1
Std. dev. of bucket size:       1
Maximum bucket size     :      10
```

Viel besser

```
StringTable statistics:
Number of buckets       :   60013
Average bucket size     :       0
Variance of bucket size :       0
Std. dev. of bucket size:       1
Maximum bucket size     :       5
```

Empfehlenswert

In Java 8 ausführlichere Informationen

```
StringTable statistics:
Number of buckets       : 60013 = 480104 bytes, avg   8,000
Number of entries       :   797 =  19128 bytes, avg  24,000
Number of literals      :   797 = 151960 bytes, avg 190,665
Total footprint         :       = 651192 bytes
Average bucket size     : 0,013
Variance of bucket size : 0,013
Std. dev. of bucket size: 0,115
Maximum bucket size     :     2
```

GC1 + JVM-Optionen

JEP 192: String Deduplication in G1 [JEP192]

Summary

„Reduce the Java heap live-data set by enhancing the G1 garbage collector so that duplicate instances of String are automatically and continuously deduplicated. “

JEP 192: String Deduplication in G1 [JEP192]

Summary

„Reduce the Java heap live-data set by enhancing the G1 garbage collector so that duplicate instances of String are automatically and continuously deduplicated. “

Motivation

„. . . Measurements have shown that roughly 25% of the Java heap live data set in these types of applications is consumed by String objects. . . . roughly half of those String objects are duplicates . . . Having duplicate String objects on the heap is, essentially, just a waste of memory“

JEP 192: String Deduplication in G1 [JEP192]

Summary

„Reduce the Java heap live-data set by enhancing the G1 garbage collector so that duplicate instances of String are automatically and continuously deduplicated. “

Motivation

„. . . Measurements have shown that roughly 25% of the Java heap live data set in these types of applications is consumed by String objects. . . . roughly half of those String objects are duplicates . . . Having duplicate String objects on the heap is, essentially, just a waste of memory“

Description

„The value field is implementation-specific and not observable from outside . . . This means that it can safely and transparently be used by multiple instances of String at the same time.

JEP 192: String Deduplication in G1 [JEP192]

Summary

„Reduce the Java heap live-data set by enhancing the G1 garbage collector so that duplicate instances of String are automatically and continuously deduplicated. “

Motivation

„. . . Measurements have shown that roughly 25% of the Java heap live data set in these types of applications is consumed by String objects. . . . roughly half of those String objects are duplicates . . . Having duplicate String objects on the heap is, essentially, just a waste of memory“

Description

„The value field is implementation-specific and not observable from outside . . . This means that it can safely and transparently be used by multiple instances of String at the same time. Deduplicating a String object is conceptually just an re-assignment of the value field, i.e., aString.value = anotherString.value.

-XX:+UseG1GC -XX:+UseStringDeduplication

```
String tmp = "some string";
String string1 = new String(tmp + tmp);
String string2 = new String(tmp + tmp);

Field field = String.class.getDeclaredField("value");
field.setAccessible(true);

Assert.assertEquals(string1, string2);
Assert.assertNotSame(string1, string2);

Assert.assertNotSame(field.get(string1),
                     field.get(string2));

System.gc();
Thread.sleep(1000);

Assert.assertNotSame(string1, string2);
Assert.assertSame(field.get(string1),
                  field.get(string2));
```

Dies und das

Heinz Kabutz, Reflection Madness, JAX London 2014

- Java 1.0 - 1.2
  - String contained char[], offset, count
- Java 1.3 - 1.6
  - Added a cached hash code
  - String became a shared, mutable, but thread-safe class
- Java 1.7
  - Got rid of offset and length and added hash32
- Java 1.8
  - Got rid of hash32 again

Andere String-relevante VM-Optionen

- `-XX:+UseStringCache`
  „Enables caching of commonly allocated strings.“
  Keine weiteren Informationen gefunden. Vorhanden in Java 6 und 7. Entfernt in Java 8

Andere String-relevante VM-Optionen

- `-XX:+UseStringCache`
  „Enables caching of commonly allocated strings.“
  Keine weiteren Informationen gefunden. Vorhanden in Java 6 und 7. Entfernt in Java 8
- `-XX:+UseCompressedStrings`
  „Use a byte[] for Strings which can be represented as pure ASCII.“
  Eingeführt in Java 6u21. Wieder entfernt in Java 7

Andere String-relevante VM-Optionen

- `-XX:+UseStringCache`
  „Enables caching of commonly allocated strings.“
  Keine weiteren Informationen gefunden. Vorhanden in Java 6 und 7. Entfernt in Java 8
- `-XX:+UseCompressedStrings`
  „Use a byte[] for Strings which can be represented as pure ASCII.“
  Eingeführt in Java 6u21. Wieder entfernt in Java 7
- `-XX:+OptimizeStringConcat`
  „Optimize String concatenation operations where possible.“
  Eingeführt in Java 6u20. Optimiert wiederholte StringBuilder append()-Aufrufe

Ausblick — es geht immer weiter . . .

- JEP 254: Compact Strings [JEP254]

Ausblick — es geht immer weiter ...

- JEP 254: Compact Strings [JEP254]
- Bereits 8/2014 definiert

Ausblick — es geht immer weiter ...

- JEP 254: Compact Strings [JEP254]
- Bereits 8/2014 definiert
- „Wiederbelebung“ von `-XX:+UseCompressedStrings`

Ausblick — es geht immer weiter ...

- JEP 254: Compact Strings [JEP254]
- Bereits 8/2014 definiert
- „Wiederbelebung" von `-XX:+UseCompressedStrings`
- kodiert Zeichen als ISO-8859-1/Latin-1 (1 Byte pro Zeichen) oder UTF-16 (2 Bytes pro Zeichen)

Ausblick — es geht immer weiter ...

- JEP 254: Compact Strings [JEP254]
- Bereits 8/2014 definiert
- „Wiederbelebung“ von `-XX:+UseCompressedStrings`
- kodiert Zeichen als ISO-8859-1/Latin-1 (1 Byte pro Zeichen) oder UTF-16 (2 Bytes pro Zeichen)
- JavaOne 2015, CON5483: Compact Strings: A Memory-Efficient Internal Representation for Strings

Ausblick — es geht immer weiter ...

- JEP 254: Compact Strings [JEP254]
- Bereits 8/2014 definiert
- „Wiederbelebung" von `-XX:+UseCompressedStrings`
- kodiert Zeichen als ISO-8859-1/Latin-1 (1 Byte pro Zeichen) oder UTF-16 (2 Bytes pro Zeichen)
- JavaOne 2015, CON5483: Compact Strings: A Memory-Efficient Internal Representation for Strings
- Gemeinsames Projekt von Oracle und Intel, 10+ Entwickler

Ausblick — es geht immer weiter . . .

- JEP 254: Compact Strings [JEP254]
- Bereits 8/2014 definiert
- „Wiederbelebung“ von `-XX:+UseCompressedStrings`
- kodiert Zeichen als ISO-8859-1/Latin-1 (1 Byte pro Zeichen) oder UTF-16 (2 Bytes pro Zeichen)
- JavaOne 2015, CON5483: Compact Strings: A Memory-Efficient Internal Representation for Strings
- Gemeinsames Projekt von Oracle und Intel, 10+ Entwickler
- Kommt in Java 9

Fragen und Anmerkungen

Referenzen

Referenzen

[Gonc] Antonio Goncalves: Who Cares About toString Performance?
[Intr1] Wikipedia: Intrinsic function
[Intr2] Krystal Mo. Intrinsic Methods in HotSpot VM
[Bug1] JDK-6962931: move interned strings out of the perm gen
[Bug2] JDK-6964458: Reimplement class meta-data storage to use native memory
[Bug3] JDK-4513622: (str) keeping a substring of a field prevents GC for object
[JEP192] JEP 192: String Deduplication in G1
[JEP254] JEP 254: Compact Strings

Referenzen (cont'd)

[CHAR1] Supplementary Characters in the Java Platform
[CHAR2] Supplementary Characters as Surrogates
[CHAR3] Unicode Character Representations
[CHAR4] Supported Encodings
[CHAR5] Unicode 4.0 support in J2SE 1.5