转载请注明链接:https://blog.csdn.net/feather_wch/article/details/82389184

本文包括如下内容:

> 1. String的特点
> 2. StringBuider的特点
> 3. StringBuffer的特点
> 4. 什么是字符串缓存的intern机制
> 5. 什么是字符串排重
> 6. 什么是intrinsic机制
> 7. Java 9中String的改进

String、SrtingBuilder、StringBuffer详解

版本号：2018/9/5-1(16:16)

问题汇总

1. 【☆】String包含哪些方面的知识?
   > 1. String、StringBuilder、StringBuffer的特点和区别。
   > 2. 字符串缓存的intern机制
   > 3. 字符串排重(JVM)

    4. Intrinsic机制
    5. JAVA9的Compat Strings

2. String、StringBuffer、StringBuilder的区别
3. String的特点
4. String的immutabale特性有哪些优点呢?
5. String的内部原理
6. String比较的equals和==的区别
7. String API的分类（12）
8. String拼接的场景中是否一定要使用StringBuilder或者StringBuffer?
9. String底层实现采用char导致的问题?
10. AbstractStringBuilder是什么（2）
11. AbstractStringBuilder的API分类（7）
12. StringBuffer和StringBuilder的扩容问题(默认容量和性能损耗)
13. StringBuffer的特点（3）
14. StringBuffer的适用场景?
15. StringBuilder的特点（4）
16. StringBuilder的适用场景?
17. 字符串重复的开销问题
18. Java 6开始提供的intern()的作用
19. Java 6中intern的严重缺陷
20. Java 6以后的字符串缓存的优化
21. 字符串缓存大小?如何修改?
22. Java6以后intern()的副作用
23. Oracle JDK 8u20后，推出了字符串排重的新特性
24. JDK 8 的字符串排重的功能如何开启？(GC1)
25. JVM内部的Intrinsic机制是干什么的?
26. 字符串如何利用Intrinsic机制优化的?
27. Java9中StringBuffer和StringBuilder底层的char[]数组都变更为byte[]数组
28. Java9中的字符串引入了Compact Strings进行了哪些方面的修改?
29. String是典型的immutable类，final修饰的类是否就是immutable的类?
30. final的作用？(类、变量、方法)
31. 如何去实现一个immutable类?
32. 【☆】Java中对String的缓存机制?
    intern()、JDK8JVM层的字符串排重
33. getBytes()和new String()采用的什么编码方式?
34. JDK中String的hash值为什么没有采用final修饰，也没有考虑hashcode()在多线程中会重复计算的问题?
35. Java为了避免在系统中产生大量的String对象，引入了字符串常量池。
36. 字符串常量池有什么用?
37. 所有的String都是字符串常量池?

38. 【☆】创建字符串对象的两种方式?
    > 1. 直接赋值： `String str = “bitch”;`
    > 2. new方式创建

39. new方式创建的String对象是否会采用字符串常量池?
40. 为什么StringBuilder、StringBuffer要给定初始值?
41. String采用的不可变模式的优点?
42. String常量池的优化机制?
43. String str = new String("AB")是否还会涉及到常量池？如何验证?
44. String str = new String("AB")会创建几个对象?
45. 【☆】String str = "AB"会创建几个对象?
    > 只会创建一个对象。

46. 如何打印对象的地址?
47. 如何打印出String对象的地址?
48. 如何通过string调用Object的toString
49. intern在JDK1.6和JDK1.7的区别?

String

1、String、StringBuffer、StringBuilder的区别

String	特点	线程安全	性能
String	提供字符串相关功能。内部是final char[]数组。是典型的 `immutable` 类(final的class、final的字段)。	安全	性能低，内不会产生新的String对象。
StringBuffer	用于解决字符串拼接产生的中间对象的问题。继承自 `AbstractStringBuilder` 内部是char[]数组	安全	性能稍低，采用synchronized进行加锁。
StringBuilder	继承自 `AbstractStringBuilder` 内部是char[]数组	线程不安全	性能高

2、String的特点(4)

> 1. String是典型的 `immutable` 类(不可变的)： 修改`String`不会在原有的内存地址修改，而是重新指向一个新对象
> 2. String用final修饰，不可以 继承 ： String本质是 `final`的`char[]`数组 ， 所以 `char[]` 数组的内存地址不会被修改，而且 `String` 没有对外暴露修改 `char[]`数组 的方法。
> 3. String是线程安全的：因为其是 `immutable` 类，实现 字符串常量池 。
> 4. 频繁的 增删操作 不建议使用 `String`

> 5. 操作不当会导致大量临时String对象。

3、String的immutabale特性有哪些优点呢?

> 1. 性能安全
> 2. 拷贝：不需要额外复制数据。

```
public String(String original) {
    this.value = original.value;
    this.hash = original.hash;
}
```

4、String的内部原理

> 1. `String` 内部是一个 `final`修饰 `char[]`数组 ： 名称为 `value`
> 2. `String`的构造 本质上就是对 `value`数组 赋初值的过程。
> 3. `String` 的其他API本质都是对 `value`数组 操作的过程。如： `substring` 是通过 数组的复制 完成的，最终会 返回 新建的 `String` ， 不会影响到原有的数组。

5、String比较的equals和==的区别

> 1. astr.equals(bstr) 比较对象是否相等
> 2. == 仅仅是比较首地址

6、String API的分类（12）

> 1. 构造方法
> 2. 字符串长度：length
> 3. 字符串某一位置：charAt
> 4. 提取子串：substring
> 5. 字符串比较：compareTo、compareToIgnore、equals、equalsIgnoreCase
> 6. 字符串的连接：concat
> 7. 字符串的查找：indexOf、lastIndexOf
> 8. 字符串的替换：replace
> 9. 前后空格的移除：trim
> 10. 起始字符串/终止字符串是否与给定字符串相同：startsWith、endWith
> 11. 是否包含字符串：contains
> 12. 基本类型转换为字符串类型：valueOf

7、String拼接的场景中是否一定要使用StringBuilder或者StringBuffer?

> 不是!
>
> 1. String的可读性更好；StringBuilder的可读性差。
> 2. JDK 8开始， “a”+“b”+“c” 会默认采用 `StringBuilder` 实现(反编译后可以看见)
> 3. JDK 9中，提供了更加统一的字符串操作优化，提供了 `StringConcatFactory` 作为同一入口。

8、String底层实现采用char导致的问题?

> 1. char是两个 `bytes` 大小，拉丁语系语言的字符不需要这么宽的char。
> 2. char的无区别实现，会导致浪费。
> 3. Java9中采用byte[]数组来实现。

常量池

9、Java为了避免在系统中产生大量的String对象，引入了字符串常量池。

> 1. 创建字符串时，会到常量池中检查是否有相同的字符串对象。
> 2. 如果有，就直接返回其引用。
> 3. 如果没有，会创建字符串对象，将其放入常量池，并且返回引用。

10、字符串常量池有什么用?

> Java为了避免在系统中产生大量的String对象

11、所有的String都是字符串常量池?

> 错误!

12、直接赋值的String对象才会放入字符串常量池： `String str = “bitch”;`

13、new方式创建的String对象不妨放入常量池： `String str = new String("Hello");`

> 1. 关键字`new` 创建String对象，不会去检查常量池
> 2. new创建String会直接在 堆区 或者 栈区 创建一个新的对象，也不会放入池中。

14、 `String str = new String("AB");` 是否还会涉及到常量池？如何验证?

15、 `String str = new String("AB");` 会创建几个对象?

> 两个对象!
>
> 1. "AB"会先创建String对象，内容为"AB"。再用该string去创建str。
> 2. 建议使用 `String str = "AB"` ，只会创建一个对象。

16、String的直接赋值得到的对象和new创建的对象是否相同?

```
String s1 = "hello world!";
String s2 = "hello world!";
String s3 = new String(s1);
String s4 = new String("hello world!");

System.out.println("s1 == s2 : " + (s1 == s2));
System.out.println("s1 == s3 : " + (s1 == s3));
System.out.println("s3 == s4 : " + (s3 == s4));
```

1. s1和s2是同一个对象(常量池复用)。
2. s3,s4都是创建的额外的对象。

intern

17、JVM常量池位置的变更

1. JDK1.6,JVM运行时数据区的 方法区 中，有一个常量池。
2. JDK1.6以后，常量池位于 堆空间
3. 常量池的位置会影响intern的效果。

18、intern在JDK1.6和JDK1.7的区别

1. JDK1.6: intern()方法会把首次遇到的字符串实例复制到永久代（可以就当是方法区）中，返回的也是永久代中这个字符串实例的引用。
2. JDK1.7: intern()实现不会复制实例，只是在常量池中记录首次出现的实例的引用。

```
//JDK1.6
String s3 = new String("1") + new String("1");
// 1. 将s3（“11”）的复制品，放到永久带中。
s3.intern();
// 2. s4 = s3的复制品
String s4 = "11";
// 3. 一定不相等，结果为false
System.out.println(s3 == s4);
```

```
//JDK1.7
String s3 = new String("1") + new String("1");
// 1. 将s3（“11”）的引用记录在常量池中。
s3.intern();
// 2. s4 = s3的引用
String s4 = "11";
// 3. 一定相等，结果为true
System.out.println(s3 == s4);
```

19、String的intern方法的实例

1-如下情况有什么结果？(JDK7)

```
// 1. 生成常量池中的Hello、World！生成位于堆空间中“HelloWorld！”的对象
String str1 = new String("Hello")+ new String("World!");
// 2、 str1.intern()，在常量池中记录首次出现的实例的引用。也就是str1的引用，因此str1.intern() = st
System.out.println(str1.intern() == str1);
// 3、“HelloWorld！”会去常量池找，找到了str1的引用
System.out.println(str1 == "HelloWorld!");
```

```
true
true
```

> 2-如下情况有什么结果?(JDK7)

```
// 1. 现在常量池中生成"HelloWorld!"
String str2 = "HelloWorld!";
// 2. 生成常量池中的Hello、World! 生成位于堆空间中“HelloWorld! ”的对象，也就是str1
String str1 = new String("Hello")+ new String("World!");
// 3、 str1.intern()，在常量池中记录首次出现的实例的引用。也就是"HelloWorld!"的引用，所以str1.inte
System.out.println(str1.intern() == str1);
// 4、“HelloWorld! ”会去常量池找，找到了，返回常量池的引用。因此和str1一定不相等
System.out.println(str1 == "HelloWorld!");
```

```
false
false
```

> 3-如下情况有什么结果?(JDK7)

```
        // 1、常量"1"和堆空间对象“1”
        String s = new String("1");
        // 2、在常量池中记录首次出现的实例的引用。也就是常量“1”的引用
        s.intern();
        // 3、从常量池中获取到常量“1”的引用。
        String s2 = "1";
        // 4、s2=常量“1”， s=对象“1”，一定不相等
        System.out.println(s == s2);
```

```
false
```

> 4-如下情况有什么结果?(JDK7)

```
        // 1、生成常量“1”，s3=堆空间对象“11”
        String s3 = new String("1") + new String("1");
        // 2、在常量池中记录首次出现的实例的引用。也就是s3的引用。
        s3.intern();
        // 3、去常量池找“11”，获取到s3的引用
        String s4 = "11";
        // 4、s4和s3肯定相同 = true
        System.out.println(s3 == s4);
```

```
true
```

AbstractStringBuilder

1、AbstractStringBuilder是什么（2）

> 1. StringBuider 和 StringBuffer 都是继承自 AbstractStringBuilder
> 2. 内部是一个char[]数组 ， 没有final 修饰符。

2、AbstractStringBuilder的API分类（7）

> 1. 扩容：ensureCapacity、newCapacity、hugeCapacity--- 扩容方式：以前大小 * 2 + 2
> 2. 追加：append（容量不够就扩容，容量够就追加到 value数组 的最后）
> 3. 插入字符串：insert
> 4. 删除字符串：delete（删除[start, end]之间的字符，通过复制的方式实现）
> 5. 提取子串：substring(new一个新String)
> 6. 字符串的替换：replace
> 7. 字符串某位置的字符：charAt

3、StringBuffer和StringBuilder的扩容问题

> 1. 默认容量是 16
> 2. 如果知道需要的容量需要 手动设置
> 3. 频繁扩容会导致严重的性能损耗，会涉及到创建新数组和 arraycopy 的数据复制，会产生的性能问题。

StringBuffer

4、StringBuffer的特点（3）

> 1. 继承自 AbstractStringBuilder
> 2. 构造 默认是创建 容量为16 的 AbstractStringBuilder
> 3. 线程安全： StringBuffer 的所有方法，都是直接使用父类的方法，并都是用 synchronized 进行加锁保护。
> 4. 性能比StringBuilder低。

5、StringBuffer的适用场景?

> 1. 线程安全 ： 适用于 多线程
> 2. Http参数拼接、xml解析

StringBuilder

6、StringBuilder的特点（4）

> 1. 继承自 AbstractStringBuilder
> 2. Java 1.5中新增
> 3. 初始容量16.

4. 非线程安全： StringBuilder 的所有方法，都是直接使用父类的方法，但是没有使用 synchronized 进行加锁保护。
5. 性能最高，在单线程中推荐使用。

7、StringBuilder的适用场景?

2. 非线程安全： 适用于 单线程
3. SQL语句拼接
4. JSON封装
5. XML解析

字符串缓存

1、字符串重复的开销问题

1. 经过分析，对象中平均25%都是字符串
2. 字符串的50%都是重复的字符串。
3. 如果进行优化，能有效降低内存消耗、对象创建开销。

2、Java 6开始提供的intern()

1. 一种显式的排重机制。
2. string.intern(); 能提示JVM把相应的字符串缓存起来，以备重复使用。
3. 调用该方法时，如果常量池中有String和该String相等，则直接返回常量池中的字符串。(String的equals进行比较)
4. 如果常量池中没有该字符串，则将String对象，添加到常量池中，并且返回引用。

3、Java 6中intern的严重缺陷

1. Java 6中并不推荐使用 intern
2. 将缓存的字符串存储到了 PermGen 里，也就是臭名昭著的 永久代
3. FullGC以外的垃圾回收都不会涉及到 永久代
4. 使用不当，会导致 OOM 问题

4、Java 6以后的字符串缓存的优化

1. 后续采用了 堆 来替代 永久代 来存储 字符串
2. Java 8中采用 MetaSpace(元数据区)

5、字符串缓存大小?

1. 随着发展，已经从最初的 1009 提升到了 60013
2. -XX:+PrintStringTableStatistics 可以查看 具体数值
3. -XX:StringTableSize=XXX 能手动修改， 不建议修改

6、intern()的副作用

> 1. 需要开发者显式调用，使用不方便
> 2. 很难保证效率，因为开发者难以清楚 字符串的重复率 ，最终可能导致代码的污染。

字符串排重

7、Oracle JDK 8u20后，推出了字符串排重的新特性

> 1. G1 GC 下的字符串排重
> 2. 做法：将相同数据的字符串指向同一份数据
> 3. 这种方法是JVM底层的改变，并不需要Java类库做什么改变。

8、JDK 8 的字符串排重的功能默认是关闭的

> 1-需要指定使用G1 GC

```
-XX:+UseG1GC
```

> 2-开启字符串排重功能

```
-XX:+UseStringDeduplication
```

Intrinsic机制

9、JVM内部的Intrinsic机制是干什么的?

> 1. 是一种利用native方法，hard-coded(硬编码)的逻辑，
> 2. 一种特殊的内联(intrinsic-内在的)。
> 3. 很多优化还是需要直接使用特定的CPU指令。

10、字符串如何利用Intrinsic机制优化的?

> 1. 字符串的特殊操作运行的都是特殊优化的 本地代码
> 2. 而不是去运行 Java代码 生成的 字节码

Java9 Compact String

1、Java9中StringBuffer和StringBuilder底层的char[]数组都变更为byte[]数组

2、Java9中的字符串引入了Compact Strings的设计

> 1. String不再使用 char数组
> 2. String采用 byte数组 实现，并且加上标识编码 coder
> 3. 将String相关的操作类都进行了修改。

> 4. String相关的 `intrinsic` 类都进行了重写

知识扩展

1、String是典型的immutable类，final修饰的类是否就是immutable的类?

> 不是

2、final的作用？(类、变量、方法)

3、如何去实现一个immutable类?

4、Java中String的缓存机制。

5、getBytes()和new String()采用的什么编码方式?

> 1. 会先从JVM参数中找有没有指定的file.encoding参数
> 2. 没有找到，会采用操作系统环境的编码方式。
> 3. 建议：getBytes/String相关业务需要指定 编码方式 ，否则因为其不确定性，可能会导致问题。

6、JDK中String的hash值为什么没有采用final修饰，也没有考虑同步问题?

> 1. String的hash值没有采用final修饰，其计算方式是在第一次调用 `hashcode()` 时生成。
> 2. hashcode()方法没有加锁，没有采用 `valatile` 修饰。在多线程中可能会出现 `hash值` 多次计算。
> 3. 虽然运算结果是一致的(同一个对象调用hashcode方法，结果肯定是一致的)，为什么不去优化这种会多次计算的情况。
> 4. 这种优化会导致在通用场景变成 持续的成本 ，volatile有明显开销，但是冲突并不多见。因此不需要这种优化。

```
public int hashCode() {
    int h = hash;
    // 1. h == 0, 决定了在单线程中只会计算一次。
    if (h == 0 && value.length > 0) {
        char val[] = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

7、为什么StringBuilder、StringBuffer要给定初始值?

> 1. 如果采用默认容量16，在字符串很长的情况下，会导致多次扩容。

> 2. 扩容时的创建新数组，arrayCopy的复制都会影响性能。
> 3. 建议在使用时，预估会用到的字符串长度，合理的设定容量。

8、String采用的不可变模式的优点?

> 1. 不可变模式是一种优质的设计模式。
> 2. 能提高多线程程序性能。
> 3. 能降低多线程程序的复杂度。

9、String常量池的优化

> 1. 当两个String对象拥有相同内容是，只会引用常量池中同一个内容。

10、如何打印对象的地址?

> 1-直接打印对象: 会显示其地址(在@后面),十六进制。如果对象重写了toString()就不会打印出内存地址。

```
System.out.println(object); // java.lang.Object@4554617c
```

> 2-对象的toString(): 会显示其地址(在@后面),十六进制。如果对象重写了toString()就不会打印出内存地址。

```
System.out.println(object.toString()); //java.lang.Object@4554617c
```

> 2-对象的hashCode(): 等于内存地址(十进制)。如果对象重写了hashCode()，会导致数值和内存地址不相关的了。

```
System.out.println(object.hashCode()); //1163157884
```

11、如何打印出String对象的地址?

> 1-使用 `System.identityHashCode(s1)` 可以计算出任何对象的hashCode(根据内存地址得到)，就算重写过 `hashCode()` 也不会影响

```
String s1 = "Hello World for Feather!";
String s2 = "Hello World for Feather!";
String s3 = new String(s2);
String s4 = new String("Hello World for Feather!");

System.out.println(System.identityHashCode(s1));
System.out.println(System.identityHashCode(s2));
System.out.println(System.identityHashCode(s3));
System.out.println(System.identityHashCode(s4));
\\输出结果
s1 1163157884
s2 1163157884
s3 1956725890
s4 356573597
```

> 2-用==来比较是否相等，但是无法比较。

12、如何通过string调用Object的toString?

> 不可以

编译和反编译

1、Java代码的编译和反编译

> 1. javac: 编译
> 2. javap: 反编译

```
javac Main.java
javap -v Main.class
```

知识储备

1、JEP 193: Variable Handles 是什么?

> Variable Handles的API主要是用来取代 `java.util.concurrent.atomic`包 以及 `sun.misc.Unsafe`类 的功能。

参考资料

1. Java技术——你真的了解String类的intern()方法吗