

第5讲 | String、StringBuffer、StringBuilder有什么区别？

2018-05-15 杨晓峰



第5讲 | String、StringBuffer、StringBuilder有什么区别？

朗读人：黄洲君 11'59" | 5.49M

今天我会聊聊日常使用的字符串，别看它似乎很简单，但其实字符串几乎在所有编程语言里都是个特殊的存在，因为不管是数量还是体积，字符串都是大多数应用中的重要组成。

今天我要问你的问题是，**理解 Java 的字符串，String、StringBuffer、StringBuilder 有什么区别？**

典型回答

String 是 Java 语言非常基础和重要的类，提供了构造和管理字符串的各种基本逻辑。它是典型的 Immutable 类，被声明成为 final class，所有属性也都是 final 的。也由于它的不可变性，类似拼接、裁剪字符串等动作，都会产生新的 String 对象。由于字符串操作的普遍性，所以相关操作的效率往往对应用性能有明显影响。

StringBuffer 是为了解决上面提到拼接产生太多中间对象的问题而提供的一个类，它是 Java 1.5 中新增的，我们可以用 append 或者 add 方法，把字符串添加到已有序列的末尾或者指定位置。StringBuffer 本质是一个线程安全的可修改字符序列，它保证了线程安全，也随之带来了额外的性能开销，所以除非有线程安全的需要，不然还是推荐使用它的后继者，也就是 StringBuilder。

StringBuilder 在能力上和 StringBuffer 没有本质区别，但是它去掉了线程安全的部分，有效减小了开销，是绝大部分情况下进行字符串拼接的首选。

考点分析

几乎所有的应用开发都离不开操作字符串，理解字符串的设计和实现以及相关工具如拼接类的使用，对写出高质量代码是非常有帮助的。关于这个问题，我前面的回答是一个通常的概要性回答，至少你要知道 String 是 Immutable 的，字符串操作不当可能会产生大量临时字符串，以及线程安全方面的区别。

如果继续深入，面试官可以从各种不同的角度考察，比如可以：

- 通过 String 和相关类，考察基本的线程安全设计与实现，各种基础编程实践。
- 考察 JVM 对象缓存机制的理解以及如何良好地使用。
- 考察 JVM 优化 Java 代码的一些技巧。
- String 相关类的演进，比如 Java 9 中实现的巨大变化。
-

针对上面这几方面，我会在知识扩展部分与你详细聊聊。

知识扩展

1. 字符串设计和实现考量

我在前面介绍过，String 是 Immutable 类的典型实现，原生的保证了基础线程安全，因为你无法对它内部数据进行任何修改，这种便利甚至体现在拷贝构造函数中，由于不可变，Immutable 对象在拷贝时不需要额外复制数据。

我们再来看看 StringBuffer 实现的一些细节，它的线程安全是通过把各种修改数据的方法都加上 synchronized 关键字实现的，非常直白。其实，这种简单粗暴的实现方式，非常适合我们常见的线程安全类实现，不必纠结于 synchronized 性能之类的，有人说“过早优化是万恶之源”，考虑可靠性、正确性和代码可读性才是大多数应用开发最重要的因素。

为了实现修改字符序列的目的，StringBuffer 和 StringBuilder 底层都是利用可修改的（char，JDK 9 以后是 byte）数组，二者都继承了 AbstractStringBuilder，里面包含了基本操作，区别仅在于最终的方法是否加了 synchronized。

另外，这个内部数组应该创建成多大的呢？如果太小，拼接的时候可能要重新创建足够大的数组；如果太大，又会浪费空间。目前的实现是，构建时初始字符串长度加 16（这意味着，如果没有构建对象时输入最初的字符串，那么初始值就是 16）。我们如果确定拼接会发生非常多次，而且大概是可预计的，那么就可以指定合适的大小，避免很多次扩容的开销。扩容会产生多重开销，因为要抛弃原有数组，创建新的（可以简单认为是倍数）数组，还要进行 arraycopy。

前面我讲的这些内容，在具体的代码书写中，应该如何选择呢？

在没有线程安全问题的情况下，全部拼接操作是应该都用 StringBuider 实现吗？毕竟这样书写的代码，还是要多敲很多字的，可读性也不理想，下面的对比非常明显。

```
String strByBuilder = new
StringBuilder().append("aa").append("bb").append("cc").append
    ("dd").toString();

String strByConcat = "aa" + "bb" + "cc" + "dd";
```

其实，在通常情况下，没有必要过于担心，要相信 Java 还是非常智能的。

我们来做个实验，把下面一段代码，利用不同版本的 JDK 编译，然后再反编译，例如：

```
public class StringConcat {
    public static void main(String[] args) {
        String myStr = "aa" + "bb" + "cc" + "dd";
        System.out.println("My String:" + myStr);
    }
}
```

先编译再反编译，比如使用 JDK 9：

```
${JAVA9_HOME}/bin/javac StringConcat.java

${JAVA9_HOME}/bin/javap -v StringConcat.class
```

JDK 8 的输出片段是：

```
6: new          #4          // class java/lang/StringBuilder
9: dup
10: invokespecial #5          // Method java/lang/StringBuilder."<init>":()V
13: ldc           #6          // String My String:
15: invokevirtual #7          // Method java/lang/StringBuilder.append:(Ljava/lang/
18: aload_1
19: invokevirtual #7          // Method java/lang/StringBuilder.append:(Ljava/lang/
22: invokevirtual #8          // Method java/lang/StringBuilder.toString:()Ljava/
```

而在 JDK 9 中，反编译的结果就非常简单了，片段是：

```
7: invokedynamic #4, 0       // InvokeDynamic #0:makeConcatWithConstants:(Ljava/lang/
```

你可以看到，在 JDK 8 中，字符串拼接操作会自动被 javac 转换为 StringBuilder 操作，而在 JDK 9 里面则是因为 Java 9 为了更加统一字符串操作优化，提供了 StringConcatFactory，作为一个统一的入口。javac 自动生成的代码，虽然未必是最优化的，但普通场景也足够了，你可以酌情选择。

2. 字符串缓存

我们粗略统计过，把常见应用进行堆转储（Dump Heap），然后分析对象组成，会发现平均 25% 的对象是字符串，并且其中约半数重复的。如果能避免创建重复字符串，可以有效降低内存消耗和对象创建开销。

String 在 Java 6 以后提供了 intern() 方法，目的是提示 JVM 把相应字符串缓存起来，以备重复使用。在我们创建字符串对象并调用 intern() 方法的时候，如果已经有缓存的字符串，就会返回缓存里的实例，否则将其缓存起来。一般来说，JVM 会将所有的类似“abc”这样的文本字符串，或者字符串常量之类缓存起来。

看起来很不错是吧？但实际情况估计会让你大跌眼镜。一般使用 Java 6 这种历史版本，并不推荐大量使用 intern，为什么呢？魔鬼存在于细节中，被缓存的字符串是存在所谓 PermGen 里的，也就是臭名昭著的“永久代”，这个空间是很有限的，也基本不会被 FullGC 之外的垃圾收集照顾到。所以，如果使用不当，OOM 就会光顾。

在后续版本中，这个缓存被放置在堆中，这样就极大避免了永久代占满的问题，甚至永久代在 JDK 8 中被 MetaSpace（元数据区）替代了。而且，默认缓存大小也在不断地扩大中，从最初的 1009，到 7u40 以后被修改为 60013。你可以使用下面的参数直接打印具体数字，可以拿自己的 JDK 立刻试验一下。

```
-XX:+PrintStringTableStatistics
```

你也可以使用下面的 JVM 参数手动调整大小，但是绝大部分情况下并不需要调整，除非你确定它的大小已经影响了操作效率。

```
-XX:StringTableSize=N
```

Intern 是一种显式地排重机制，但是它也有一定的副作用，因为需要开发者写代码时明确调用，一是不方便，每一个都显式调用是非常麻烦的；另外就是我们很难保证效率，应用开发阶段很难清楚地预计字符串的重复情况，有人认为这是一种污染代码的实践。

幸好在 Oracle JDK 8u20 之后，推出了一个新的特性，也就是 G1 GC 下的字符串排重。它是通过将相同数据的字符串指向同一份数据来做到的，是 JVM 底层的改变，并不需要 Java 类库做什么修改。

注意这个功能目前是默认关闭的，你需要使用下面参数开启，并且记得指定使用 G1 GC：

```
-XX:+UseStringDeduplication
```

前面说到的几个方面，只是 Java 底层对字符串各种优化的一角，在运行时，字符串的一些基础操作会直接利用 JVM 内部的 Intrinsic 机制，往往运行的就是特殊优化的本地代码，而根本就不是 Java 代码生成的字节码。Intrinsic 可以简单理解为，是一种利用 native 方式 hard-coded 的逻辑，算是一种特别的内联，很多优化还是需要直接使用特定的 CPU 指令，具体可以看相关[源码](#)，搜索“string”以查找相关 Intrinsic 定义。当然，你也可以在启动实验应用时，使用下面参数，了解 intrinsic 发生的状态。

```
-XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining  
  
// 样例输出片段  
  
    180    3    3    java.lang.String::charAt (25 bytes)  
          @ 1    java.lang.String::isLatin1 (19 bytes)
```

```
...  
@ 7 java.lang.StringUTF16::getChar (60 bytes) intrinsic
```

可以看出，仅仅是字符串一个实现，就需要 Java 平台工程师和科学家付出如此大且默默无闻的努力，我们得到的很多便利都是来源于此。

我会在专栏后面的 JVM 和性能等主题，详细介绍 JVM 内部优化的一些方法，如果你有兴趣可以再深入学习。即使你不做 JVM 开发或者暂时还没有使用到特别的性能优化，这些知识也能帮助你增加技术深度。

3.String 自身的演化

如果你仔细观察过 Java 的字符串，在历史版本中，它是使用 char 数组来存数据的，这样非常直接。但是 Java 中的 char 是两个 bytes 大小，拉丁语系语言的字符，根本就不需要太宽的 char，这样无区别的实现就造成了一定的浪费。密度是编程语言平台永恒的话题，因为归根结底绝大部分任务是要来操作数据的。

其实在 Java 6 的时候，Oracle JDK 就提供了压缩字符串的特性，但是这个特性的实现并不是开源的，而且在实践中也暴露出了一些问题，所以在最新的 JDK 版本中已经将它移除了。

在 Java 9 中，我们引入了 Compact Strings 的设计，对字符串进行了大刀阔斧的改进。将数据存储方式从 char 数组，改变为一个 byte 数组加上一个标识编码的所谓 coder，并且将相关字符串操作类都进行了修改。另外，所有相关的 Intrinsic 之类也都进行了重写，以保证没有任何性能损失。

虽然底层实现发生了这么大的改变，但是 Java 字符串的行为并没有任何大的变化，所以这个特性对于绝大部分应用来说是透明的，绝大部分情况不需要修改已有代码。

当然，在极端情况下，字符串也出现了一些能力退化，比如最大字符串的大小。你可以思考下，原来 char 数组的实现，字符串的最大长度就是数组本身的长度限制，但是替换成 byte 数组，同样数组长度下，存储能力是退化了一倍的！还好这是存在于理论中的极限，还没有发现现实应用受此影响。

在通用的性能测试和产品实验中，我们能非常明显地看到紧凑字符串带来的优势，即更小的内存占用、更快的操作速度。

今天我从 String、StringBuffer 和 StringBuilder 的主要设计和实现特点开始，分析了字符串缓存的 intern 机制、非代码侵入性的虚拟机层面排重、Java 9 中紧凑字符的改进，并且初步接触了 JVM 的底层优化机制 intrinsic。从实践的角度，不管是 Compact Strings 还是底层 intrinsic 优化，都说明了使用 Java 基础类库的优势，它们往往能够得到最大程度、最高质量的优化，而且只要升级 JDK 版本，就能零成本地享受这些益处。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？限于篇幅有限，还有很多字符相关的问题没有来得及讨论，比如编码相关的问题。可以思考一下，很多字符串操作，比如 `getBytes()`/[String](#) (`byte[] bytes`) 等都是隐含着使用平台默认编码，这是一种好的实践吗？是否有利于避免乱码？

请你在留言区写写你对这个问题的思考，或者分享一下你在操作字符串时掉过的坑，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



版权归极客邦科技所有，未经许可不得转载

精选留言



公众号:代码荣耀

今日String/StringBuffer/StringBuilder心得:

👍 59

1 String

(1) String的创建机理

由于String在Java世界中使用过于频繁，Java为了避免在一个系统中产生大量的String对象，引入了字符串常量池。其运行机制是：创建一个字符串时，首先检查池中是否有值相同的字符串对象，如果有则不需要创建直接从池中刚查找到的对象引用；如果没有则新建字符串对象，返回对象引用，并且将新创建的对象放入池中。但是，通过new方法创建的String对象是不检查字符串池的，而是直接在堆区或栈区创建一个新的对象，也不会把对象放入池中。上

述原则只适用于通过直接量给String对象引用赋值的情况。

举例：String str1 = "123"; //通过直接量赋值方式，放入字符串常量池

String str2 = new String("123");//通过new方式赋值方式，不放入字符串常量池

注意：String提供了inter()方法。调用该方法时，如果常量池中包括了一个等于此String对象的字符串（由equals方法确定），则返回池中的字符串。否则，将此String对象添加到池中，并且返回此池中对象的引用。

(2) String的特性

[A] 不可变。是指String对象一旦生成，则不能再对它进行改变。不可变的主要作用在于当一个对象需要被多线程共享，并且访问频繁时，可以省略同步和锁等待的时间，从而大幅度提高系统性能。不可变模式是一个可以提高多线程程序的性能，降低多线程程序复杂度的设计模式。

[B] 针对常量池的优化。当2个String对象拥有相同的值时，他们只引用常量池中的同一个拷贝。当同一个字符串反复出现时，这个技术可以大幅度节省内存空间。

2 StringBuffer/StringBuilder

StringBuffer和StringBuilder都实现了AbstractStringBuilder抽象类，拥有几乎一致对外提供的调用接口；其底层在内存中的存储方式与String相同，都是以一个有序的字符序列（char类型的数组）进行存储，不同点是StringBuffer/StringBuilder对象的值是可以改变的，并且值改变以后，对象引用不会发生改变；两者对象在构造过程中，首先按照默认大小申请一个字符数组，由于会不断加入新数据，当超过默认大小后，会创建一个更大的数组，并将原先的数组内容复制过来，再丢弃旧的数组。因此，对于较大对象的扩容会涉及大量的内存复制操作，如果能够预先评估大小，可提升性能。

唯一需要注意的是：StringBuffer是线程安全的，但是StringBuilder是线程不安全的。可参看Java标准类库的源代码，StringBuffer类中方法定义前面都会有synchronize关键字。为此，StringBuffer的性能要远低于StringBuilder。

3 应用场景

[A]在字符串内容不经常发生变化的业务场景优先使用String类。例如：常量声明、少量的字符串拼接操作等。如果有大量的字符串内容拼接，避免使用String与String之间的“+”操作，因为这样会产生大量无用的中间对象，耗费空间且执行效率低下（新建对象、回收对象花费大量时间）。

[B]在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在多线程环境下，建议使用StringBuffer，例如XML解析、HTTP参数解析与封装。

[C]在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在单线程环境下，建议使用StringBuilder，例如SQL语句拼装、JSON封装等。

2018-05-15

作者回复

很到位

2018-05-15



Kongk0ng

👍 8

编译器为什么不把

```
String myStr = "aa" + "bb" + "cc" + "dd";
```

默认优化成

```
String myStr = "aabbccdd";
```

这样不是更聪明嘛

2018-05-16



Hidden

👍 7

公司没有技术氛围，项目也只是 功能实现就好，不涉及优化，技术也只是传统技术，想离职，但又怕裸辞后的各种压力

2018-05-16



sea季陪我去看海

👍 5

作者我有个疑问，String myStr = "aa" + "bb" + "cc" + "dd"; 应该编译的时候就确定了，不会用到StringBuilder。理由是：

```
String myStr = "aa" + "bb" + "cc" + "dd";
```

```
String h = aabbccdd
```

Mystr == h 上机实测返回的是true，如果按照你的说法，应该是返回false才对，因为你说拼接用到stringbuilder，那mystr应该是堆地址，h是常量池地址。

2018-05-16



肖一林

👍 5

这篇文章写的不错，由浅入深，把来龙去脉写清楚了

2018-05-15

作者回复

谢谢认可

2018-05-15



薛好运

👍 4

老师，可以讲解这一句话的具体含义吗，谢谢！

你可以思考下，原来 char 数组的实现，字符串的最大长度就是数组本身的长度限制，但是替换成 byte 数组，同样数组长度下，存储能力是退化了一倍的！还好这是存在于理论中的极限，还没有发现现实应用受此影响。

2018-05-15

作者回复

已回复，一个char两个byte，注意下各个类型宽度

2018-05-15



Jerry银银

👍 4

要完全消化一篇文章的所有内容，真得不是一天就能搞定的，可能需要一个月，甚至好几个月。就比如今天的字符串，我觉得这个话题覆盖的面太广：算法角度；易用角度；性能角度；编码传输角度等

但是好在，我获得见识。接下来，花时间慢慢研究呗，连大师们都花了那么多时间研究，我们多花点时间，很正常嘛😊

一点学习心得，和大家分享

2018-05-15



Chris

👍 3

`new string ("ghhh").intern ()`；会从堆到常量池是这个作用吗

2018-05-15



愉悦在花香的日子里

👍 3

`getBytes`和`String`相关的转换时根据业务需要建议指定编码方式，如果不指定则看看JVM参数里有没有指定`file.encoding`参数，如果JVM没有指定，那使用的默认编码就是运行的操作系统环境的编码了，那这个编码就变得不确定了。常见的编码`iso8859-1`是单字节编码，`UTF-8`是变长的编码。

2018-05-15

作者回复

不错，莫依赖于不确定因素

2018-05-15



Bin

👍 2

jdk1.8中，`String`是标准的不可变类，但其`hashCode`没有用`final`修饰，其`hashCode`计算是在第一次调用`hashCode`方法时计算，但方法没有加锁，变量也没用`volatile`关键字修饰就无法保证其可见性。当有多个线程调用的时候，`hashCode`值可能会被计算多次，虽然结果是一样的，但jdk的作者为什么不将其优化一下呢？

2018-05-16

作者回复

这些“优化”在通用场景可能变成持续的成本，`volatile read`是有明显开销的；如果冲突并不多见，`read`才是更普遍的，简单的`cache`是更高效的

2018-05-17



echo

👍 1

`String`是`immutable`，在`security`、`Cache`、`Thread Safe`等方面都有很好的体现。

`Security`: 传参的时候我们很多地方使用`String`参数，可以保证参数不会被改变，比如数据库

连接参数url等，从而保证数据库连接安全。

Cache: 因为创建String前先去Constant Pool里面查看是否已经存在此字符串，如果已经存在，就把该字符串的地址引用赋给字符变量；如果没有，则在Constant Pool创建字符串，并将字符串引用赋给字符串变量。所以存在多个引用指向同一个字符串对象，利用缓存有助于提高内存开销。

Thread Safe: 因为String是immutable，所以它是Automatically thread safe.

问题：我一直不能很好的理解最后一个体现，到底String是如何体现在thread safe，老师能不能用简短的线程代码给讲讲？非常感激。

2018-05-16



debugable

1

java9讲字符串内部使用字节数组保存，取一个中文字符串字符个数是不是就不能用length了？

2018-05-15

作者回复

不用担心，我提到了API行为没变化，包括charAt之类全部都是如此

2018-05-15



曹铮

1

思考题里的平台默认编码，平台指的是JVM所在的操作系统，还是指语言平台本身呢？

2018-05-15

作者回复

环境编码

2018-05-15



Jerry银银

1

特别喜欢这句话：“仅仅是字符串一个实现，就需要Java平台工程师和科学家付出如此大且默默无闻的努力，我们得到的很多便利都是来源于此。”

我想说，同学们，写代码的时候记得感恩哦😊

对于字符串的研究，我觉得能很好的理解计算机的本质和训练计算机思维，提升自己解决问题的能力。

小小的字符串有着诸多巨人的影子

2018-05-15

作者回复

非常感谢

2018-05-15



嘎哈

1

char数组的实现，字符串的最大长度就是数组本身的长度限制，但是替换成byte数组，同样数组长度下，存储能力是退化了一倍的！

怎么理解呢？举个例子呗

2018-05-15

作者回复

已回复，一个char两个byte，注意下各个类型宽度

2018-05-15



jutsu

1

地铁上看起来啦，从来没有string类型的线程问题，受教了

2018-05-15

作者回复

谢谢

2018-05-15



So Leung

0

经过验证new String时，不会再常量池中创建对象。

2018-05-24



©®

0

```
String s1=new String("StringTest");
System.out.println(s1.intern()==s1);
//false ( JDK 8 )
```

```
String s1 = new StringBuilder().append("String").append("Test").toString();
System.out.println(s1.intern() == s1);//true ( JDK 8 )
```

```
String s1 = new StringBuilder("StringTest").toString();
System.out.println(s1.intern() == s1);
//false ( JDK 8 )
```

。。老师，append为什么会造成这个差异

2018-05-24



So Leung

0

那想问下，当new String("a")时，不会在常量池创建对象？

2018-05-24



©®

0

String s2=new String("AB")，，如果，常量池中没有AB,那么会不会去常量池创建，望解答

2018-05-23

作者回复

new只是创建新的；另外，有没有想过怎么通过一段程序证明？这样更有助于理解

2018-05-24

