二

05 String、StringBuffer、StringBuilder有什么区别？-极客时间

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，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 是 Java 1.5 中新增的，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 StringBuilder 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 String concat(String str) {
       return str + “aa” + “bb”;
     }
}
```

先编译再反编译，比如使用不同版本的 JDK：

```
${JAVA_HOME}/bin/javac StringConcat.java
${JAVA_HOME}/bin/javap -v StringConcat.class
```

JDK 8 的输出片段是：

```
         0: new           #2                  // class java/lang/StringBuilder
         3: dup
         4: invokespecial #3                  // Method java/lang/StringBuilder."<i
         7: aload_0
         8: invokevirtual #4                  // Method java/lang/StringBuilder.app
        11: ldc           #5                  // String aa
        13: invokevirtual #4                  // Method java/lang/StringBuilder.app
        16: ldc           #6                  // String bb
        18: invokevirtual #4                  // Method java/lang/StringBuilder.app
        21: invokevirtual #7                  // Method java/lang/StringBuilder.toS
```

而在 JDK 9 中，反编译的结果就会有点特别了，片段是：

```
         // concat method
         1: invokedynamic #2,  0              // InvokeDynamic #0:makeConcatWithCon

         // ...
         // 实际是利用了MethodHandle,统一了入口
         0: #15 REF_invokeStatic java/lang/invoke/StringConcatFactory.makeConcatWit
```

你可以看到，非静态的拼接逻辑在 JDK 8 中会自动被 javac 转换为 StringBuilder 操作；而在 JDK 9 里面，则是体现了思路的变化。Java 9 利用 InvokeDynamic，将字符串拼接的优化与 javac 生成的字节码解耦，假设未来 JVM 增强相关运行时实现，将不需要依赖 javac 的任何修改。

在日常编程中，保证程序的可读性、可维护性，往往比所谓的最优性能更重要，你可以根据实际需求酌情选择具体的编码方式。

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，并且其中约半数是重复的。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) int
```

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。

上一页 下一页