链滴

深入理解 Java String

作者：wuhongxu

原文链接：https://ld246.com/article/1533720931387

来源网站：链滴

以下代码或分析来源于jdk8

简介

String 类由final标记，实现了java.io.Serializable,Comparable<String>,CharSequence(字符序列)个接口。

String/StringBuilder/StringBuffer本质上都是通过字符数组实现的。

String是字符串__常量__,对String值的操作是产生新的String对象。

StringBuilder/StringBuffer都是字符串__变量__,即可变的字符序列，都继承自AbstractStringBuilde，实现CharSequence接口。对StringBuffer/StringBuilder类操作是对字符数组进行扩容。

StringBuffer是线程安全的，StringBuilder是非线程安全的。

源码浅析

String类包含char[] value属性，StringBuffer/StringBuilder的父类AbstractStringBuilder包含char[] value属性。同时注意此value在String中的声明为private final char value[],对于字符串的处理都是处理value属性。

String类做如此多的限定是一种Immutable设计模式的典型应用，String变量一旦初始化后就不能更，禁止改变对象的状态，从而增加共享对象的坚固性、减少对象访问的错误，同时还避免了在多线程享时进行同步的需要。

另外一点对与String类如此设计的猜测，设计到java对于String类型本身的设计，字符串都存储在常池中用于共享从而提高性能的，但是如果String值是可变的，那么就可能引起冲突，所以需要设计成可变的，以此来避免字符串共享所引起的数据冲突。

常用功能实现

hashcode方法实现：

```
public int hashCode() {
    int h = hash;//hash默认是0
    if (h == 0 && value.length > 0) { //只算一次
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

indexOf方法：

```
public int indexOf(int ch, int fromIndex) {
    final int max = value.length;
    if (fromIndex < 0) {
```

```
        fromIndex = 0;
    } else if (fromIndex >= max) {
        // Note: fromIndex might be near -1>>>1.
        return -1;
    }
    //Character.MIN_SUPPLEMENTARY_CODE_POINT = 65536 也就是两个字节所能存储的最大值
    if (ch < Character.MIN_SUPPLEMENTARY_CODE_POINT) {
        // handle most cases here (ch is a BMP code point or a
        // negative value (invalid code point))
        final char[] value = this.value;
        for (int i = fromIndex; i < max; i++) {
            if (value[i] == ch) {
                return i;
            }
        }
        return -1;
    } else {
        return indexOfSupplementary(ch, fromIndex);
    }
}

/**
 * Handles (rare) calls of indexOf with a supplementary character.
 */
//java中特意对超过两个字节的字符进行了处理
private int indexOfSupplementary(int ch, int fromIndex) {
    if (Character.isValidCodePoint(ch)) {
        final char[] value = this.value;
        final char hi = Character.highSurrogate(ch);
        final char lo = Character.lowSurrogate(ch);
        final int max = value.length - 1;
        for (int i = fromIndex; i < max; i++) {
            if (value[i] == hi && value[i + 1] == lo) {
                return i;
            }
        }
    }
    return -1;
}
```

java特意对多余两个字节的字符例如一些emoji做了特殊的处理。

在lastIndexOf方法中，有这样一段对于多重循环的处理值得注意以下：

```
static int lastIndexOf(char[] source, int sourceOffset, int sourceCount,
            char[] target, int targetOffset, int targetCount,
            int fromIndex) {
    ...
    startSearchForLastChar:
    while (true) {
        while (i >= min && source[i] != strLastChar) {
            i--;
        }
        if (i < min) {
```

```
            return -1;
        }
        int j = i - 1;
        int start = j - (targetCount - 1);
        int k = strLastIndex - 1;

        while (j > start) {
            if (source[j--] != target[k--]) {
                i--;
                continue startSearchForLastChar;
            }
        }
        return start - sourceOffset + 1;
    }
```

使用了continue label的方式来处理多重循环下的跳出，这是java中存在但是很少被使用的方式，类似语言中的goto语句，java可以在循环体前设置跳转标志，然后配合continue/break关键字更好的控多重循环。

重点讲解以下split方法：

```
public String[] split(String regex, int limit) {
    char ch = 0;
    if (((regex.value.length == 1 &&
          ".$|()[{^?*+\\".indexOf(ch = regex.charAt(0)) == -1) ||
          (regex.length() == 2 &&
          regex.charAt(0) == '\\' &&
          (((ch = regex.charAt(1))-'0')|('9'-ch)) < 0 &&
          ((ch-'a')|('z'-ch)) < 0 &&
          ((ch-'A')|('Z'-ch)) < 0)) &&
        (ch < Character.MIN_HIGH_SURROGATE ||
         ch > Character.MAX_LOW_SURROGATE))
    {
        int off = 0;
        int next = 0;
        boolean limited = limit > 0;
        ArrayList<String> list = new ArrayList<>();
        while ((next = indexOf(ch, off)) != -1) {
            if (!limited || list.size() < limit - 1) {
                list.add(substring(off, next));
                off = next + 1;
            } else {    // last one
                //assert (list.size() == limit - 1);
                list.add(substring(off, value.length));
                off = value.length;
                break;
            }
        }
        // If no match was found, return this
        if (off == 0)
            return new String[]{this};

        // Add remaining segment
```

```
        if (!limited || list.size() < limit)
            list.add(substring(off, value.length));

        // Construct result
        int resultSize = list.size();
        if (limit == 0) {
            while (resultSize > 0 && list.get(resultSize - 1).length() == 0) {
                resultSize--;
            }
        }
        String[] result = new String[resultSize];
        return list.subList(0, resultSize).toArray(result);
    }
    return Pattern.compile(regex).split(this, limit);
}
```

其中的第一个if有一段非常复杂的判断，且体现了jdk源码中极致的优化。考虑这样的情况，如果传入仅仅只有一个字符，并且这个字符不是任何特殊的正则表达式，明显我们不需要经过正则表达式编译次。同样，如果是两个字符，并且这两个字符是以\开头，并且不是字母或者数字的时候，例如'|,为什么会有这种情况呢?

用实际代码试一试：

```
String s = "a|b|c";
String[] split = s.split("|");
for (String s1 : split) {
    System.out.println(s1);
}
---------输出---------
a
|
b
|
c
```

很奇怪吧？这是出于第一个判断".$|()[{^?*+\\".indexOf(ch = regex.charAt(0)) == -1)，判断的是是是特殊的正则表达式，而|符号正好位列其中，|在正则中表示__或__，而"|"就等同于""。所以就相当于字符串用空格分开。于是就有了这一个考虑，通过转移，让'|'字符生效,里面又包含了一个有意思的代片段：(((ch = regex.charAt(1))-'0')|('9'-ch)) < 0 &&((ch-'a')|('z'-ch)) < 0 && ((ch-'A')|('Z'-ch)) <))。判断字符是否是数字或者字母，使用或运算减少代码量提高效率。之后的第三个判断是用于判断否是两个字节的unicode字符。

还有一条很优雅的代码片段值得注意：

```
boolean limited = limit > 0;
ArrayList<String> list = new ArrayList<>();
while ((next = indexOf(ch, off)) != -1) {
    if (!limited || list.size() < limit - 1) {
        list.add(substring(off, next));
        off = next + 1;
    } else {    // last one
        //assert (list.size() == limit - 1);
        list.add(substring(off, value.length));
        off = value.length;
        break;
```

```
    }
}
...
// 将最后剩余的一段连接上
if (!limited || list.size() < limit)
    list.add(substring(off, value.length));
// 构造结果集合
int resultSize = list.size();
if (limit == 0) {
    while (resultSize > 0 && list.get(resultSize - 1).length() == 0) {
        resultSize--;
    }
}
String[] result = new String[resultSize];
return list.subList(0, resultSize).toArray(result);
```

将判断limit和分配剩余字符串两者巧妙的结合。

Immutable设计模式

String类有两个构造方法值得注意：

```
public String(char value[]) {
    this.value = Arrays.copyOf(value, value.length);
}
public String(char value[], int offset, int count) {
    ...
    if (count <= 0) {
        ...
        if (offset <= value.length) {
            this.value = "".value;
            return;
        }
    }
    ...
    this.value = Arrays.copyOfRange(value, offset, offset+count);
}
```

这两个构造方法都调用了Arrays.copy方法，经过查看源码，都是复制数组，也就是说传入的char[]数在外部的变化并不会影响String本身的值，体现不可变的原则，其他类似的传入数组的构造器都是类的行为。也因此，String所有能够返回值的方法也同样是采用类似的复制数组的方式，例如：

```
public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin) {
    ...
    System.arraycopy(value, srcBegin, dst, dstBegin, srcEnd - srcBegin);
}
```

在源码中到处充斥着这样的代码，3000多行代码（包含注释），都在严谨的遵循着这个原则。

String的本质

String类中所有修改值的方法都是返回新的对象，例如String.substring方法：

```
//截取字符串
public String substring(int beginIndex, int endIndex) {
  ...
    int subLen = endIndex - beginIndex;
  ...
    return ((beginIndex == 0) && (endIndex == value.length)) ? this
    : new String(value, beginIndex, subLen);
}
```

处理value返回新的字符串对象，new String(value,beginIndex,subLen)构造函数会截取数组并复制新String对象的value。

但是我们是否能够通过反射来修改呢？做一个尝试：

```
String s1 = "abcd";
String s2 = "abcd";
String s3 = new String("abcd");
String s4 = new String(s1);
System.out.println("s1==s2:" + (s1 == s2));
System.out.println("s1==s3:" + (s1 == s3));
System.out.println("s3==s4:" + (s3 == s4));
Field field = String.class.getDeclaredField("value");
field.setAccessible(true);
char[] value = (char[]) field.get(s1);
value[1] = 'p';
System.out.println("s1 = " + s1);
System.out.println("s2 = " + s2);
System.out.println("s3 = " + s3);
System.out.println("s4 = " + s4);

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

输出如下：

```
s1==s2:true
s1==s3:false
s3==s4:false
s1 = apcd
s2 = apcd
s3 = apcd
s4 = apcd
s1==s2:true
s1==s3:false
s3==s4:false
```

由此可以看出，反射能够修改String的值，但是另外出现了一个有趣的点，在修改了s1的值之后，s2/3/s4的值都经过了修改。一个一个的看，先看最简单的s4。

通过s1来new一个新的String对象，翻看构造器源码：

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

```
  this.hash = original.hash;
}
```

很简单，是通过传递引用value来构造的新的String对象，也就是说s1和s4享有同一个char数组。所无论我们修改s1还是s4的值，都是修改的这个char数组。所以s4随着s1变化，可以理解。

接下来看s2/s3为什么变化：

表面上看，s1与s2的赋值虽然都是“相等”的字符串，但是无论声明还是使用看起来都确实是分开的这里从jdk源码上我们无从得知具体原因。这牵扯出了我们对于String类型的最大疑问，字符串的本质底是怎样的，jvm是如何对待一特殊的类型的。探究出s2自然而然就能了解s3变化的始末。

其实从前面的源码查看中，我们已经能够看出来，字符串的本质就是__字符数组__。但是按照不同的式来初始化出来的字符串却并不是被一视同仁的。

String的定义方法归纳出来大致分为三种方式：

- 双引号直接赋值：`String str = "str";`
- 使用new关键字：`String str = new String("str");`
- 连接字符串：`String str = "s" + "tr";`

在此，需要先了解一下jvm中的堆栈、常量池的概念。

常量池

常量池指的是在编译期就被确定，并且保存在已编译的.class文件中的一些数据，包含代码中所定义各种基本类型（如int、long等等）和对象型（如String及数组）的常量值(final)还包含一些以文本形出现的符号引用。比如：

- 类和接口的全限定名
- 字段的名称和描述符
- 方法的名称和描述符

堆

一个运行时的数据区域，类的对象就是在这里分配空间的。这些对象通过new、newarray、anewarray和multianewarray等指令建立，它们不需要程序代码来显式的释放。堆是由垃圾回收来负责的，堆优势是可以动态地分配内存 大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的Java的垃圾收集器会自动收走这些不再使用的数据,也因此大小和声明周期并不确定。但缺点是，由于在运行时动态 分配内存，存取速度较慢。堆中的对象不可以共享（注意，这里的共享不是指我们在jav代码中的赋值给另外一个引用，而是指:`Object a = new Object();Object b = new Object()`则`a == b`情况）

栈

存取速度比堆要快，仅次于寄存器，栈数据可以共享(`int a = 1;int b = 1`则`a == b`)，存放基本类型的量数据和对象的引用，但对象本身不存放在栈中，而是存放在堆（new 出来的对象）

ps:网上有说对象不一定存储在堆中，这句话是对的，但是举出来的反例是字符串对象存储在常量池中这个反例是错误的，大胆举证一个最简单的例子，`String a = new String("a");String b = new String "a")`，如果按照这种说法，那么`a == b`，因为常量池中的数据是共享的，然后明显不对，至于更深层

的原因下面讲到。正确的反例应当是《深入理解java虚拟机》一书中所说到的 “随着JIT编译期的发与逃逸分析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所有的对象都分到堆上也渐渐变得不那么“绝对”了”。通熟点就是java底层优化越来越变态导致的。

解析String的定义

在编译期就被确定的（以双引号定义的）字符串就被存储在常量池中，如果运行期（new）才能确定就存储在堆中。

这里了解以下String.intern方法。String的intern()方法会查找在常量池中是否存在一份equal相等的符串,如果有则返回该字符串的引用,如果没有则添加自己的字符串进入常量池。

运行时常量池相对于CLass文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一只有编译期才能产生，也就是并非预置入CLass文件中常量池的内容才能进入方法区运行时常量池，行期间也可能将新的常量放入池中，这种特性被开发人员利用比较多的就是String类的intern()方法。

- 第一种定义方法执行过程(String str = "str")：在程序编译期间，编译程序先去字符串常量池检查是否由"str"存在，如果不存在，则在常量池中开辟一个内存空间存放"str"，如果存在，则不开辟。接，会在堆中创建一个String对象，注意这里和网上有些说法不一样，下面我们通过代码证明这里的正性，之后在栈中开辟一块空间，命名为str，存放对象的__地址__。

```
String s = new String("str");
        String s2 = s.intern();
        System.out.println(s == s2);

-------输出---------------
false
```

明显看出s2并不是指向的s，而是指向的"str"字符串所指向的对象。

- 第二种定义方法执行过程(String str = new String("str")):在程序编译期间，编译程序先在字符串量池检查，是否存在"str"，如果不存在，则在常量池中开辟一个内存空间存放"str"。如果不存在，则辟一个内存空间，否则不开辟。然后在内存堆中开辟一个空间，存放new出来的String实例，之后在中开辟一个空间，命名为str,存放堆中String实例的内存 **地址** ，这个过程就是将引用str指向new出来Sring实例。
- 第三种定义方法执行过程:对于第三个方法的过程会稍显复杂，在这个举例中jvm会进行小技巧优化我们将定义的形式变化一下，绕过这个优化，然后根据字节码来判断，当我们进行字符串拼接初始化符串时会发生什么。

原代码如下：

```
String s = "1"+"2"+"3";
System.out.println(s);
String s2 = "2";
for (int i = 0; i < 10; i++){
  s2 += s2;
}
System.out.println(s2);
```

字节码输出如下：

```
Code:
0: ldc           #2                  // String 123
```

```
2: astore_1
3: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
6: aload_1
7: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
10: ldc          #5                  // String 2
12: astore_2
13: iconst_0
14: istore_3
15: iload_3
16: bipush       10
18: if_icmpge    46
21: new          #6                  // class java/lang/StringBuilder
24: dup
25: invokespecial #7                 // Method java/lang/StringBuilder."<init>":()V
28: aload_2
29: invokevirtual #8                 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)L
ava/lang/StringBuilder;
32: aload_2
33: invokevirtual #8                 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)L
ava/lang/StringBuilder;
36: invokevirtual #9                 // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
39: astore_2
40: iinc         3, 1
43: goto         15
46: getstatic    #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
49: aload_2
50: invokevirtual #4                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
53: return
```

简单描述一下关键：

- 0: ldc #2 // String 123 代表引用常量池里的值，这里代码经过编译器优化之后，"1"+"2"+"3"被化成了"123"
- astore_2 代表将ldc的值“2”存到__局部变量表__(用于存放方法参数和方法内部定义的部变量)中的第2个槽中
- 13-43行是一个循环。也就是我们的0-10的循环，可以看到在这个循环中在不停的new StringBuilder（21行）。这种没有经过编译器优化的拼接是采用StringBuidler实现的。
- 最后会调用StringBuilder的toString方法，然后jvm会拿到字符序列，并按照String对象的处理方进行处理。

简单对比我们主动采用StringBuider类拼接字符串。

源码：

```
StringBuilder s2 = new StringBuilder("2");
for (int i = 0; i < 10; i++){
  s2.append("2");
}
System.out.println(s2);
```

字节码：

```
0: new           #2                  // class java/lang/StringBuilder
```

```
3: dup
4: ldc           #3                  // String 2
6: invokespecial #4                  // Method java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
9: astore_1
10: iconst_0
11: istore_2
12: iload_2
13: bipush        10
15: if_icmpge     31
18: aload_1
19: ldc           #3                  // String 2
21: invokevirtual #5                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)L
ava/lang/StringBuilder;
24: pop
25: iinc          2, 1
28: goto          12
```

明显看到在0处new 了一次StringBuilder之后，就一直在重用此对象。显然提高了很多的效率。

回到定义方式继续探讨，三种定义方式都抛出了一个非常重要的问题，new出来的String对象实例和常量池中的字符串是什么关系？有什么区别？为什么直接定义的字符串同样可以调用String对象的各方法呢？

以”String s1 = new String("somestring")“为例，在字符串常量池中存储"somestring"字符序列在堆中开辟String对象空间，在栈中存储这个地址命名为s。首先就需要弄清楚，怎么通过对象找到常池中的字符串？

在解析阶段，虚拟机发现字符串常量somestring，它会在一个内部字符串常量列表【各种虚拟机实现式可能不一样，例如hotspot使用hashtable实现】中查找，如果没有找到，那么会在堆中创建一个含字符序列[somestring]的string对象_s_，然后把这个字符序列和对应的String对象做为名值对（[soestring],s）保存到内部字符串常量列表中。

弄懂了这些原理，我们就能够清楚的知道前面s1/s2/s3/s4发生变化的原因了，很简单，因为都是操的同样的一个字符序列，这也是java废了很大力气去优化的一点，让所有的想同字符串字面值只存储次，以实现性能、空间利用的最大提升。

完成以上问题之后，就可以完美解决面试经常遇到的以下问题：

- 得出以下输出

```
String s1 = "str";
String s2 = "str";
String s3 = new String("str");
String s4 = new String(s1);

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

- 每一行代码（(每行代码分开运行，不考虑前置因素)），各自产生了多少个对象，字符串常量池里有哪几个值。

```
String s2 = "str";
```

```
String s3 = new String("str");
String s5 = new String("str")+"str";
```

StringBuilder/StringBuffer源码解析

弄清楚了String，那么StringBuilder/StringBuffer其实已经相当清楚了，本质上讲StringBuilder/StringBuffer是对字符数组进行扩容的对象，都继承字AbstractStringBuilder。查看AbstractStringBuilde的源码，发现与String一样，包含char[] value和int count。但是与String不同的是，它们没有final饰符。因此得出结论：String、StringBuffer和StringBuilder在本质上都是字符数组，不同的是，在行连接操作时，String每次返回一个新的String实例，而StringBuffer和StringBuilder的append方法接返回this，所以这就是为什么在进行大量字符串连接运算时，不推荐使用String，而推荐StringBuffe和StringBuilder。

StringBuilder和StringBuffer的源码

查看两个类的append和toString方法：

```
//StringBuilder
@Override
public StringBuilder append(String str) {
    super.append(str);
    return this;
}
@Override
public String toString() {
    // Create a copy, don't share the array
    return new String(value, 0, count);
}
//StringBuffer
@Override
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
    return this;
}
@Override
public synchronized String toString() {
    if (toStringCache == null) {
        toStringCache = Arrays.copyOfRange(value, 0, count);
    }
    return new String(toStringCache, true);
}
```

可以看出第一个区别，StringBuffer采用synchronized修饰，是线程安全的，而StringBuilder不是。

另外一点，在StringBuffer中有这样一个属性private transient char[] toStringCache;。它缓存了字数组，StringBuffer下所有的修改方法都会清空掉这个缓存，只有在执行toString时才会赋值，而在StrngBuilder中完全没有这种机制。这是为什么呢？首先要指明的是，其实这个缓存意义不算很大，因两次toString而没有改变的情况是真的少见。但是从设计上来说，这种缓存对于真正遇到这种情况的能是有提升的，而StringBuffer因为线程安全，所以可以设置这样一个缓存，但是StringBuilder类并是线程安全的，如果这样设计会导致可能产生不一致的情况，对照我们平时写的代码来看，缓存应该同步安全的条件下才被设置以用来提升性能，否则这将导致产生toString结果与预期不一致。

所以在使用选择上来说，如果在多线程环境可以使用StringBuffer进行字符串连接操作，单线程环境用StringBuilder，它的效率更高。

答案

true,false,false,false

对象：1,2,3。常量池："str","str",["str","strstr"]

参考链接

- java内存分配和String类型的深度解析
- java--String常量池问题的几个例子
- java+内存分配及变量存储位置的区别