

>

String表示字符串，Java中所有字符串的字面值都是String类的实例，例如“ABC”。字符串是常量，在定义之后不能被改变，字符串缓冲区支持可变的字符串。因为 String 对象是不可变的，所以可以共享它们。例如：

```
String str = "abc";
```

相当于

```
char data[] = {'a', 'b', 'c'};  
String str = new String(data);
```

这里还有一些其他使用字符串的例子：

```
System.out.println("abc");  
String cde = "cde";  
System.out.println("abc" + cde);  
String c = "abc".substring(2,3);  
String d = cde.substring(1, 2);
```

String类提供了检查字符序列中单个字符的方法，比如有比较字符串，搜索字符串，提取子字符串，创建一个字符串的副本、字符串的大小写转换等。实例映射是基于 Character 类中指定的 Unicode标准的。**Java语言提供了对字符串连接运算符的特别支持（+）**，该符号也可用于将其他类型转换成字符串。字符串的连接实际上是通过 StringBuffer 或者 StringBuilder 的 append() 方法来实现的，字符串的转换通过 toString 方法实现，该方法由 Object 类定义，并可被 Java 中的所有类继承。除非另有说明，传递一个空参数在这类构造函数或方法会导致 NullPointerException 异常被抛出。**String表示一个字符串通过UTF-16(unicode)格式**，补充字符通过代理对（参见Character类的 Unicode Character Representations 获取更多的信息）表示。索引值参考字符编码单元，所以补充字符在String中占两个位置。

定义 属性 构造方法

使用字符数组、字符串构造一个String

使用字节数组构造一个String

使用StringBuffer和StringBuilder构造一个String

一个特殊的保护类型的构造方法

其他方法

getBytes

比较方法

hashCode

substring

replaceFirst、replaceAll、replace区别

copyValueOf 和 valueOf

intern

String对 "+" 的重载

String.valueOf和Integer.toString的区别

一、定义

```
public final class String implements java.io.Serializable, Comparable<String>, CharSequence{}
```

从该类的声明中我们可以看出String是final类型的，表示该类不能被继承，如果允许继承则SubString子类方法可以被重写比如hashCode恒等于0，和定义新的value数组，那么一些函数的参数是String时候，可以传入我们写的SubString，比如map的put方法则所有的SubString子类的实例作为key都放在map的同一个位置，降低了map的存取效率。原始的String类就不可作为map的key输入。

同时该类实现了三个接口： java.io.Serializable 、 Comparable<String> 、 CharSequence

String设计成final class原因

字符串常量池的需要。字符串常量池的诞生是为了提升效率和减少内存分配。字符串重复的可能性很高，并且其不可变性能方便常量池优化。

安全性考虑。正因为使用字符串的场景如此之多，所以设计成不可变可以有效的防止字符串被有意或者无意的篡改。（但是通过反射还是可以入侵的）

作为HashMap、HashTable等hash型数据key的必要。因为不可变的设计，jvm底层很容易在缓存String对象的时候缓存其hashCode，这样在执行效率上会大大提升。如果字符串内部value数组可变，String的hashCode是利用value字符数组计算出来的，那么以String作为key的map则对象放进去之后，可能之后取不出来。造成hashmap的内存泄露

```
private final char value[];
```

这是一个字符数组，并且是final类型，他用于存储字符串内容，从final这个关键字中我们可以看出，String的内容一旦被初始化了是不能被更改的。 虽然有这样的例子：String s = "a" ; s = "b" 但是，这并不是对s的修改，而是重新指向了新的字符串，从这里我们也能知道，**String其实就是用char[]实现的。**

```
private int hash;
```

缓存字符串的hash Code，默认值为 0

```
private static final long serialVersionUID = -6849794470754667710L;  
private static final ObjectOutputStreamField[] serialPersistentFields = new ObjectOutputStreamField[0];
```

因为String实现了Serializable接口，所以支持序列化和反序列化支持。Java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体（类）的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常(InvalidCastException)。

三、构造方法

String类作为一个java.lang包中比较常用的类,自然有很多重载的构造方法.在这里介绍几种典型的构造方法:

1.使用字符数组、字符串构造一个String

我们知道，其实String就是使用字符数组（char[]）实现的。所以我们可以使用一个字符数组来创建一个String，那么这里值得注意的是，**当我们使用字符数组创建String的时候，会用到Arrays.copyOf方法和Arrays.copyOfRange方法。这两个方法是将原有的字符数组中的内容逐一**

的复制到String中的字符数组中。同样，我们也可以用一个String类型的对象来初始化一个

String。这里将直接将源String中的value和hash两个属性直接赋值给目标String。因为String一旦定义之后是不可以改变的，所以也就不用担心改变源String的值会影响到目标String的值。

当然，在使用字符数组来创建一个新的String对象的时候，不仅可以使整个字符数组，也可以使用字符数组的一部分，只要多传入两个参数int offset和int count就可以了。

String类的构造器通过字符数组构造String对象。

复制源字符数组的内容，随后修改源字符数组不会影响新创建的字符串

```
public String(char value[]) { //String构造器源码
    this.value = Arrays.copyOf(value, value.length);
}
public static char[] copyOf(char[] original, int newLength) { //调用native方法对old字符数组拷贝
    char[] copy = new char[newLength];
    System.arraycopy(original, 0, copy, 0,
        Math.min(original.length, newLength));
    return copy;
}
```

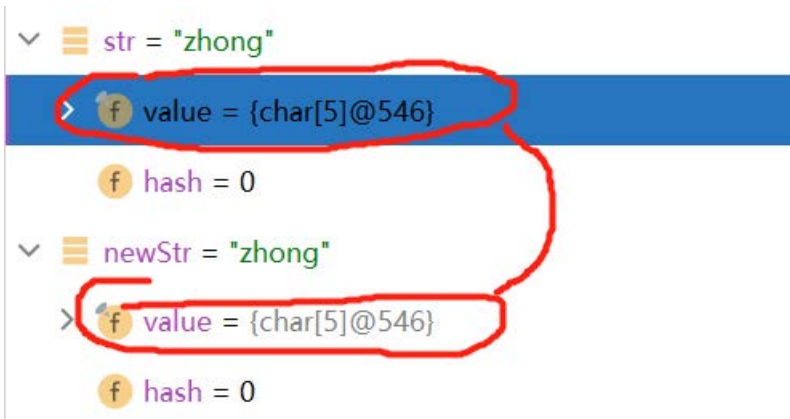
String类的构造器通过原始String构造新的String对象

```
public String(String original) {
    this.value = original.value; 原始字符串的value数组直接被“新字符串数组”value引用    this.hash
    = original.hash;原始字符串的hashCode直接被“新字符串数组”hashCode复制 }    为什么这里私有
    属性original.value可以直接获取?? 我没看明白,因为自己没弄清private访问
    权限
```

String str="zhong";zhong被存放在字符串常量池中，以char[]数组形式保存字面量

String newStr=new String(str);

System.out.println(newStr); newString中内部引用的value字符串指向str中的常量池中char[]数组



通过字符数组和字符串创建字符串的区别

new String[字符数组z h o n g]

new String["zhong"]

第二种方式在JVM中会有常量池zhong的创建以及堆内存中String对象创建，而通过字符数组创建时常量池中没有"zhong"

2.使用字节数组构造一个String

在Java中，String实例中保存有一个 char[] 字符数组，char[] 字符数组是以unicode码来存储的，String 和 char 为内存形式，byte是网络传输或存储的序列化形式。所以在很多传输和存储的过程中需要将byte[]数组和String进行相互转化。所以，String提供了一系列重载的构造方法来将一个字符数组转化成String，提到byte[]和String之间的相互转换就不得不关注编码问题。

String(byte[] bytes, Charset charset) 是指通过charset来解码指定的byte数组，将其解码成unicode的char[]数组，够造成新的String。

这里的bytes字节流是使用charset进行编码的，想要将他转换成unicode的char[]数组，而又保证不出现乱码，那就要指定其解码方式

同样使用字节数组来构造String也有很多种形式，按照是否指定解码方式分的话可以分为两种：

```
String(byte bytes[]) String(byte bytes[], int offset, int length)
```

```
String(byte bytes[], Charset charset)
```

```
String(byte bytes[], String charsetName)
```

```
String(byte bytes[], int offset, int length, Charset charset)
```

```
String(byte bytes[], int offset, int length, String charsetName)
```

如果我们在使用byte[]构造String的时候，使用的是下面这四种构造方法(带有 charsetName 或者 charset 参数)的一种的话，那么就会使用 StringCoding.decode 方法进行解码，使用的解码的字符集就是我们指定的 charsetName 或者 charset。我们在使用byte[]构造String的时候，如果没有指明解码使用的字符集的话，那么 StringCoding 的 decode 方法首先调用系统的默认编码格式，如果没有指定编码格式则默认使用**ISO-8859-1**编码格式进行编码操作。主要体现代码如下：

```
static char[] decode(byte[] ba, int off, int len) {
    String csn = Charset.defaultCharset().name();
    try {
        // use charset name decode() variant which provides caching.
        return decode(csn, ba, off, len);
    } catch (UnsupportedEncodingException x) {
        warnUnsupportedCharset(csn);
    }
    try {
        return decode("ISO-8859-1", ba, off, len);
    } catch (UnsupportedEncodingException x) {
        // If this code is hit during VM initialization, MessageUtils is
        // the only way we will be able to get any kind of error message.
        MessageUtils.err("ISO-8859-1 charset not available: "
            + x.toString());
        // If we can not find ISO-8859-1 (a required encoding) then things
        // are seriously wrong with the installation.
        System.exit(1);
        return null;
    }
}
```

3.使用StringBuffer和StringBuider构造一个String

作为String的两个“兄弟”，StringBuffer和StringBuider也可以被当做构造String的参数。

```
public String(StringBuffer buffer) {
    synchronized(buffer) { 调用native方法对StringBuffer内部的字符数组拷贝
        this.value = Arrays.copyOf(buffer.getValue(), buffer.length());
    }
}

public String(StringBuider builder) {
    this.value = Arrays.copyOf(builder.getValue(), builder.length());
}
```

当然，这两个构造方法是很少用到的，至少我从来没有使用过，因为当我们有了StringBuffer或者StringBuilfer对象之后可以直接使用他们的toString方法来得到String。关于效率问题，Java的官方文档有提到说使用StringBuilder的toString方法会更快一些，原因是StringBuffer的 toString

方法是synchronized的，在牺牲了效率的情况下保证了线程安全。



内存对象

```
public String toString() {  
    // Create a copy, don't share the array  
    return new String(value, 0, count);  
}
```

toString方法构造器是字符数组，因此JVM不会创建常量池对象，只会new字符串堆

```
this.value = Arrays.copyOfRange(value, offset, offset+count);
```

4.一个特殊的保护类型的构造方法

String除了提供了很多公有的供程序员使用的构造方法以外， 还提供了一个保护类型的构造方法（Java 7）， 我们看一下他是怎么样的：

```
String(char[] value, boolean share) {  
    // assert share : "unshared not supported";  
    this.value = value;  
}
```

从代码中我们可以看出，该方法和 String(char[] value) 有两点区别，第一个，该方法多了一个参数： boolean share， 其实这个参数在方法体中根本没被使用，也给了注释，目前不支持使用 false，只使用true。那么可以断定，**加入这个share的只是为了区分于String(char[] value)方法**，不加这个参数就没办法定义这个函数，只有参数不能才能进行重载。那么，第二个区别就是具体的方法实现不同。我们前面提到过，String(char[] value) 方法在创建String的时候会用到 Arrays 的 copyOf 方法将value中的内容逐一复制到String当中，而这个 String(char[] value, boolean share) 方法则是直接将value的引用赋值给String的value。那么也就是说，这个方法构造出来的String和参数传过来的 char[] value 共享同一个数组。那么，为什么Java会提供这样一个方法呢？ 首先，我们分析一下使用该构造函数的好处：

首先，**性能好**，这个很简单，一个是直接给数组赋值（相当于直接将String的value的指针指向char[]数组），一个是逐一拷贝。当然是直接赋值快了。

其次，共享内部数组**节约内存**

但是，该方法之所以设置为protected，是因为一旦该方法设置为公有，在外面可以访问的话，那就破坏了字符串的不可变性。例如如下YY情形：

```
char[] arr = new char[] {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};
```

```
String s = new String(0, arr.length, arr); // "hello world"  
arr[0] = 'a'; // replace the first character with 'a'  
System.out.println(s); // aello world
```

如果构造方法没有对arr进行拷贝，那么其他人就可以在字符串外部修改该数组，由于它们引用的是同一个数组，因此对arr的修改就相当于修改了字符串。

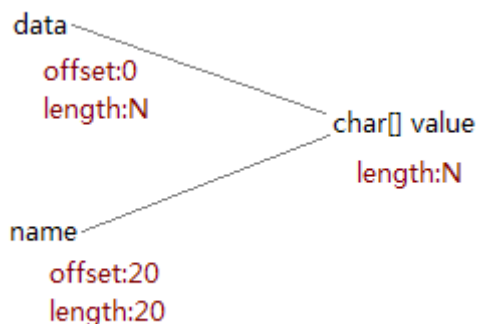
所以，从安全性角度考虑，他也是**安全**的。对于调用他的方法来说，由于无论是原字符串还是新字符串，其value数组本身都是String对象的私有属性，从外部是无法访问的，因此对两个字符串来说都很安全。

在Java 7 之有很多String里面的方法都使用这种“性能好的、节约内存的、安全”的构造函数。比如：substring、replace、concat、valueOf等方法（实际上他们使用的是public String(char[], int, int)方法，原理和本方法相同，已经被本方法取代）。

但是在Java 7中，substring已经不再使用这种“优秀”的方法了，为什么呢？虽然这种方法有很多优点，但是他有一个致命的缺点，对于sun公司的程序员来说是一个零容忍的bug，那就是他很有可能造成**内存泄露**。看一个例子，假设一个方法从某个地方（文件、数据库或网络）取得了一个很长的字符串，然后对其进行解析并提取其中的一小段内容，这种情况经常发生在网页抓取或进行日志分析的时候。下面是示例代码。

```
String aLongString = "...a very long string...";  
String aPart = data.substring(20, 40);  
return aPart;
```

在这里aLongString只是临时的，真正有用的是aPart，其长度只有20个字符，但是它的内部数组却是从aLongString那里共享的，因此虽然aLongString本身可以被回收，但它的内部数组却不能（如下图）。这就导致了内存泄漏。如果一个程序中这种情况经常发生有可能导致严重的后果，如内存溢出，或性能下降。



新的实现虽然损失了性能，而且浪费了一些存储空间，但却保证了字符串的内部数组可以和字符串对象一起被回收，从而防止发生内存泄漏，因此新的substring比原来的更健壮。

额、、、扯了好远，虽然substring方法已经为了其鲁棒性放弃使用这种share数组的方法，但是这种share数组的方法还是有一些其他方法在使用的，这是为什么呢？首先呢，这种方式构造对应有很多好处，其次呢，其他的方法不会将数组长度变短，也就不会有前面说的那种内存泄露的情况（内存泄露是指不用的内存没有办法被释放，比如说concat方法和replace方法，他们不会导致元数组中有大量空间不被使用，因为他们一个是拼接字符串，一个是替换字符串内容，不会将字符数组的长度变得很短！）。

四、其他方法

length() **返回字符串长度**

isEmpty() **返回字符串是否为空**

charAt(int index) **返回字符串中第 (index+1) 个字符**

char[] toCharArray() **转化成字符数组**

trim() **去掉两端空格**

toUpperCase() **转化为大写**

toLowerCase() **转化为小写**

String concat(String str) **//拼接字符串**

String replace(char oldChar, char newChar) **//将字符串中的oldChar字符换成newChar字符**

//以上两个方法都使用了 String(char[] value, boolean share);

boolean matches(String regex) **//判断字符串是否匹配给定的regex正则表达式**

boolean contains(CharSequence s) **//判断字符串是否包含字符序列s**

String[] split(String regex, int limit) **按照字符regex将字符串分成limit份。**

String[] split(String regex)

```
String string = "h,o,l,l,i,s,c,h,u,a,n,g";
String[] splitAll = string.split(",");
String[] splitFive = string.split(",",5);
splitAll = [h, o, l, l, i, s, c, h, u, a, n, g]
splitFive = [h, o, l, l, i,s,c,h,u,a,n,g]
```

getBytes

在创建String的时候，可以使用byte[]数组，将一个字节数组转换成字符串，同样，我们可以将一个字符串转换成字节数组，那么String提供了很多重载的getBytes方法。但是，值得注意的是，在使用这些方法的时候一定要注意编码问题。比如：

```
String s = "你好，世界！";
byte[] bytes = s.getBytes();
```

这段代码在不同的平台上运行得到结果是不一样的。由于我们没有指定编码方式，所以在该方法

对字符串进行编码的时候就会使用系统的默认编码方式，比如在中文操作系统中可能会使用GBK或者GB2312进行编码，在英文操作系统中有可能使用iso-8859-1进行编码。这样写出来的代码就和机器环境有很强的关联性了，所以，为了避免不必要的麻烦，我们要指定编码方式。如使用以下方式：

```
String s = "你好，世界！";  
byte[] bytes = s.getBytes("utf-8");
```

比较方法

```
boolean equals(Object anObject);  
boolean contentEquals(StringBuffer sb);  
boolean contentEquals(CharSequence cs);  
boolean equalsIgnoreCase(String anotherString);  
int compareTo(String anotherString);  
int compareToIgnoreCase(String str);  
boolean regionMatches(int toffset, String other, int ooffset,int len) //局部匹配  
boolean regionMatches(boolean ignoreCase, int toffset,String other, int ooffset, int len) //局部匹配
```

字符串有一系列方法用于比较两个字符串的关系。前四个返回boolean的方法很容易理解，前三个比较就是比较String和要比较的目标对象的字符数组的内容，一样就返回true,不一样就返回false，核心代码如下：

```
int n = value.length;  
while (n-- != 0) {  
    if (v1[i] != v2[i])  
        return false;  
    i++;  
}
```

v1 v2分别代表String的字符数组和目标对象的字符数组。第四个和前三个唯一的区别就是他会将两个字符数组的内容都使用toUpperCase方法转换成大写再进行比较，以此来忽略大小写进行比较。相同则返回true，不想同则返回false

在这里，看到这几个比较的方法代码，有很多编程的技巧我们应该学习。我们看equals方法：

```

public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String) anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i]) 两个字符串是否相等，底层两个字符数组字符一个一个比较
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}

```

该方法首先判断 `this == anObject` ? , 也就是说判断要比较的对象和当前对象是不是同一个对象, 如果是直接返回true, 如不是再继续比较, 然后在判断 `anObject` 是不是 `String` 类型的, 如果不是, 直接返回false, 如果是再继续比较, 到了能终于比较字符数组的时候, 他还是先比较了两个数组的长度, 不一样直接返回false, 一样再逐一比较值。虽然代码写的内容比较多, 但是可以很大程度上提高比较的效率。值得学习~~!!!

`contentEquals` 有两个重载, **StringBuffer 需要考虑线程安全问题**, 再加锁之后调用 `contentEquals((CharSequence) sb)` 方法。 `contentEquals((CharSequence) sb)` 则分两种情况, 一种是 `cs instanceof AbstractStringBuilder`, 另外一种参数是 `String` 类型。具体比较方式几乎和 `equals` 方法类似, **先做“宏观”比较, 在做“微观”比较。**

下面这个是 `equalsIgnoreCase` 代码的实现:

```

public boolean equalsIgnoreCase(String anotherString) {
    return (this == anotherString) ? true
        : (anotherString != null)
        && (anotherString.value.length == value.length)
        && regionMatches(true, 0, anotherString, 0, value.length);
}

```

看到这段代码, 眼前为之一亮。使用一个三目运算符和 `&&` 操作代替了多个 `if` 语句。

hashCode的实现其实就是使用数学公式：

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$

$s[i]$ 是string的第*i*个字符， n 是String的长度。那为什么这里用31，而不是其它数呢？计算机的乘法涉及到移位计算。当一个数乘以2时，就直接拿该数左移一位即可！选择31原因是因为31是一个素数！

所谓素数：

质数又称素数。指在一个大于1的自然数中，除了1和此整数自身外，没法被其他自然数整除的数。

素数在使用的时候有一个作用就是如果我用一个数字来乘以这个素数，那么最终的出来的结果只能被素数本身和被乘数还有1来整除！如：我们选择素数3来做系数，那么 $3*n$ 只能被3和 n 或者1来整除，我们可以很容易的通过 $3n$ 来计算出这个 n 来。这应该也是一个原因！（本段表述有问题，感谢 @沉沦 的提醒）

在存储数据计算hash地址的时候，我们希望尽量减少有同样的hash地址，所谓“冲突”。如果使用相同hash地址的数据过多，那么这些数据所组成的hash链就更长，从而降低了查询效率！所以在选择系数的时候要选尽量长的系数并且让乘法尽量不要溢出的系数，因为如果计算出来的hash地址越大，所谓的“冲突”就越少，查找起来效率也会提高。

31可以由 $i*31 == (i < 5) - 1$ 来表示，现在很多虚拟机里面都有做相关优化，使用31的原因可能是为了更好的分配hash地址，并且31只占用5bits！

在java乘法中如果数字相乘过大会导致溢出的问题，从而导致数据的丢失。

而31则是素数（质数）而且不是很长的数字，最终它被选择为相乘的系数的原因不过与此！

在Java中，整型数是32位的，也就是说最多有 $2^{32} = 4294967296$ 个整数，将任意一个字符串，经过hashCode计算之后，得到的整数应该在这4294967296数之中。那么，最多有4294967297个不同的字符串作hashCode之后，肯定有两个结果是一样的，hashCode可以保证相同的字符串的hash值肯定相同，但是，hash值相同并不一定是value值就相同。

```
public String substring(int beginIndex) {
    if (beginIndex < 0) {
        throw new StringIndexOutOfBoundsException(beginIndex);
    }
    int subLen = value.length - beginIndex;
    if (subLen < 0) {
        throw new StringIndexOutOfBoundsException(subLen);
    }
    return (beginIndex == 0) ? this : new String(value, beginIndex, subLen); 拷贝value字符数组，避免内存泄漏
}
```

前面我们介绍过，java 7 中的substring方法使用String(value, beginIndex, subLen)方法创建一个新的String并返回，这个方法会将原来的char[]中的值逐一复制到新的String中，两个数组并不是共享的，虽然这样做损失一些性能，但是有效地避免了内存泄露。1.6substring通过new String(char[],share)共享字符数组，可能导致内存泄漏。

replaceFirst、replaceAll、replace区别

```
String replaceFirst(String regex, String replacement)
String replaceAll(String regex, String replacement)
String replace(CharSequence target, CharSequence replacement)
```

1)replace的参数是char和CharSequence,即可以支持字符的替换,也支持字符串的替换 2)replaceAll和replaceFirst的参数是regex,即基于规则表达式的替换,比如,可以通过replaceAll(“\\d”, “*”)把一个字符串所有的数字字符都换成星号; **相同点是都是全部替换**,即把源字符串中的某一字符或字符串全部换成指定的字符或字符串, **如果只想替换第一次出现的,可以使用 replaceFirst()**,这个方法也是基于规则表达式的替换,但与replaceAll()不同的是,只替换第一次出现的字符串; **另外,如果replaceAll()和replaceFirst()所用的参数据不是基于规则表达式的,则与replace()替换字符串的效果是一样的,即这两者也支持字符串的操作;**

copyValueOf 和 valueOf

String的底层是由char[]实现的：通过一个char[]类型的value属性！早期的String构造器的实现呢，不会拷贝数组的，直接将参数的char[]数组作为String的value属性。然后 test[0] = 'A'; 将导致字符串的变化。为了避免这个问题，提供了 copyValueOf 方法，每次都拷贝成新的字符数组

来构造新的String对象。但是现在的String对象，在构造器中就通过拷贝新数组实现了，所以这两个方面在本质上已经没区别了。

valueOf()有很多种形式的重载，包括：

```
public static String valueOf(boolean b) {
    return b ? "true" : "false";
}

public static String valueOf(char c) {
    char data[] = {c};
    return new String(data, true);
}

public static String valueOf(int i) {
    return Integer.toString(i);
}

public static String valueOf(long l) {
    return Long.toString(l);
}

public static String valueOf(float f) {
    return Float.toString(f);
}

public static String valueOf(double d) {
    return Double.toString(d);
}
```

可以看到这些方法可以将六种基本数据类型的变量转换成String类型。

intern()方法 内容过多，分jdk版本后面讨论

```
public native String intern();
```

该方法返回一个字符串对象的内部化引用。众所周知：String类维护一个初始为空的字符串的对象池，当intern方法被调用时，如果对象池中已经包含这一个相等的字符串对象则返回对象池中的实例，否则添加字符串到对象池并返回该字符串的引用。

String对“+”的重载

我们知道，Java是不支持重载运算符，String的“+”是java中唯一的一个重载运算符，那么java如何实现这个加号的呢？我们先看一段代码：

```
public static void main(String[] args) {  
    String string="hollis";  
    String string2 = string + "chuang";  
}
```

然后将这段代码反编译：

```
public static void main(String args[]){  
    String string = "hollis";  
    String string2 = (new StringBuilder(String.valueOf(string))).append("chuang").toString();  
}
```

看了反编译之后的代码我们发现，其实String对“+”的支持其实就是使用了StringBuilder以及他的append、toString两个方法。

String.valueOf和Integer.toString的区别

接下来我们看以下这段代码，我们有三种方式将一个int类型的变量变成呢过String类型，那么他们有什么区别？

```
1.int i = 5;  
2.String i1 = "" + i;  
3.String i2 = String.valueOf(i);  
4.String i3 = Integer.toString(i);
```

1、第三行和第四行没有任何区别，因为String.valueOf(i)也是调用Integer.toString(i)来实现的。2、第二行代码其实是String i1 = (new StringBuilder()).append(i).toString();，首先创建一个StringBuilder对象，然后再调用append方法，再调用toString方法。

参考资料：

open <http://greppcode.com/file/repository.greppcode.com/java/root/jdk/openjdk/7u40-b43/java/lang/String.java#String>.jdk

String，是Java中除了基本数据类型以外，最为重要的一个类型了。很多人会认为他比较简单。但是和String有关的面试题有很多，下面我随便找两道面试题，看看你能不能都答对：

Q1： `String s = new String("hollis");` 定义了几个对象。（[直面 Java 第 025 期](https://t.zsxq.com/BUNVzJE) (<https://t.zsxq.com/BUNVzJE>))

Q2： 如何理解 String 的 `intern` 方法？（[直面Java第031期](https://t.zsxq.com/maQzrf2) (<https://t.zsxq.com/maQzrf2>))

上面这两个是面试题和String相关的比较常考的，很多人一般都知道答案。

A1：若常量池中已经存在“ hollis” ，则直接引用，也就是此时只会创建一个对象，如果常量池中不存在“ hollis” ，则先创建后引用，也就是有两个。

A2：当一个String实例str调用`intern()`方法时，Java查找常量池中是否有相同Unicode的字符串常量，如果有，则返回其的引用，如果没有，则在常量池中增加一个Unicode等于str的字符串并返回它的引用；

两个答案看上去没有任何问题，但是，仔细想想好像哪里不对呀。按照上面的两个面试题的回答，就是说 `new String` 也会检查常量池，如果有的话就直接引用，如果不存在就要在常量池创建

一个，那么还要`intern`干啥？难道以下代码是没有意义的吗？

```
String s = new String("Hollis").intern();
```

如果，每当我们使用`new`创建字符串的时候，都会到字符串池检查，然后返回。那么以下代码也应该输出结果都是 `true` ？

```
String s1 = "Hollis";
String s2 = new String("Hollis");
String s3 = new String("Hollis").intern();

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

但是，以上代码输出结果为 (base jdk1.8.0_73)：

```
false
true
```

不知道，聪明的读者看完这段代码之后，是不是有点被搞蒙了，到底是怎么回事儿？

别急，且听我慢慢道来。

字面量和运行时常量池

JVM为了提高性能和减少内存开销，在实例化字符串常量的时候进行了一些优化。为了减少在JVM中创建的字符串的数量，字符串类维护了一个字符串常量池。

在JVM运行时区域的方法区中，有一块区域是运行时常量池，主要用来存储**编译期**生成的各种**字面量**和**符号引用**。

了解Class文件结构或者做过Java代码的**反编译** (<http://www.hollischuang.com/archives/58>)的朋友可能都知道，在java代码被 javac 编译之后，文件结构中是包含一部分 Constant pool 的。比如以下代码：

```
public static void main(String[] args) {
    String s = "Hollis";
}
```

经过编译后，常量池内容如下：

```
Constant pool:
  #1 = Methodref          #4.#20      // java/Lang/Object."<init>":()V
  #2 = String              #21         // Hollis
  #3 = Class                #22        // StringDemo
  #4 = Class                #23        // java/Lang/Object
  ...
  #16 = Utf8                s
  ..
  #21 = Utf8                Hollis
  #22 = Utf8                StringDemo
  #23 = Utf8                java/Lang/Object
```

上面的Class文件中的常量池中，比较重要的几个内容：



```
#16 = Utf8          s
#21 = Utf8          Hollis
#22 = Utf8          StringDemo
```

上面几个常量中，s 就是前面提到的**符号引用**，而 Hollis 就是前面提到的**字面量**。而Class文件中的常量池部分的内容，会在运行期被运行时常量池加载进去。关于字面量，详情参考 [Java SE Specifications \(https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.10.5\)](https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.10.5)。

new String创建了几个对象

下面，我们可以来分析下 `String s = new String("Hollis");` 创建对象情况了。

这段代码中，我们可以知道的是，在编译期，**符号引用** s 和**字面量** Hollis 会被加入到Class文件的常量池中，然后在类加载阶段（具体时间段参考Java 中 `new String(“字面量”)` 中 “字面量”是何时进入字符串常量池的? (<https://www.zhihu.com/question/55994121>))，这两个常量会进入常量池。

但是，这个“进入”阶段，并不会直接把所有类中定义的常量全部都加载进来，而是会做个比较，如果需要加到字符串常量池中的字符串已经存在，那么就不需要再把字符串字面量加载进来了。

所以，当我们说<若常量池中已经存在“hollis”，则直接引用，也就是此时只会创建一个对象>说的就是这个字符串字面量在字符串池中被创建的过程。

说完了编译期的事儿了，该到运行期了，在运行期，`new String("Hollis");` 执行到的时候，是要在Java堆中创建一个字符串对象的，而这个对象所对应的字符串字面量是保存在字符串常量池中的。但是，`String s = new String("Hollis");`，**对象的符号引用 s 是保存在Java虚拟机栈上的，他保存的是堆中刚刚创建出来的的字符串对象的引用。**

所以，你也就知道以下代码输出结果为false的原因了。

```
String s1 = new String("Hollis");
String s2 = new String("Hollis");
System.out.println(s1 == s2);
```

因为，`==` 比较的是 s1 和 s2 在堆中创建的对象地址，当然不同了。但是如果使用 `equals`，那么比较的就是字面量的内容了，那就会得到 true。

在不同版本的JDK中，Java堆和字符串常量池之间的关系也是不同的，这里为了方便表述，就画成两个独立的物理区域了。具体情况请参考Java虚拟机规范。

所以，`String s = new String("Hollis");` 创建几个对象的答案你也就清楚了。

常量池中的“对象”是在编译期就确定好了的，在类被加载的时候创建的，如果类加载时，该字符串常量在常量池中已经有了，那这一步就省略了。堆中的对象是在运行期才确定的，在代码执行到new的时候创建的。

运行时常量池的动态扩展

编译期生成的各种**字面量**和**符号引用**是运行时常量池中比较重要的一部分来源，但是并不是全部。那么还有一种情况，可以在运行期像运行时常量池中增加常量。那就是 String 的 intern 方法。

当一个 String 实例调用 intern() 方法时，Java 查找常量池中是否有相同 Unicode 的字符串常量，如果有，则返回其的引用，如果没有，则在常量池中增加一个 Unicode 等于 str 的字符串并返回它的引用；

intern() 有两个作用，第一个是将字符串字面量放入常量池（如果池没有的话），第二个是返回这个常量的引用。

我们再来看下开头的那个让人产生疑惑的例子：

```
String s1 = "Hollis";
String s2 = new String("Hollis");
String s3 = new String("Hollis").intern();

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

你可以简单的理解为 String s1 = "Hollis"; 和 String s3 = new String("Hollis").intern(); 做的事情是一样的（但实际有些区别，这里暂不展开）。都是定义一个字符串对象，然后将其字符串字面量保存在常量池中，并把这个字面量的引用返回给定义好的对象引用。

对于 `String s3 = new String("Hollis").intern();` , 在不调用 `intern` 情况, `s3`指向的是JVM在堆中创建的那个对象的引用的 (如图中的`s2`) 。但是当执行了 `intern` 方法时, `s3`将指向字符串常量池中的那个字符串常量。

由于`s1`和`s3`都是字符串常量池中的字面量的引用, 所以`s1==s3`。但是, `s2`的引用是堆中的对象, 所以`s2!=s1`。

intern的正确用法

不知道, 你有没有发现, 在 `String s3 = new String("Hollis").intern();` 中, 其实 `intern` 是多余的?

因为就算不用 intern，Hollis作为一个字面量也会被加载到Class文件的常量池，进而加入到运行时常量池中，为啥还要多此一举呢？到底什么场景下才需要使用 intern 呢？

在解释这个之前，我们先来看下以下代码：

```
String s1 = "Hollis";
String s2 = "Chuang";
String s3 = s1 + s2;
String s4 = "Hollis" + "Chuang";
```

在经过反编译后，得到代码如下：

```
String s1 = "Hollis";
String s2 = "Chuang";
String s3 = (new StringBuilder()).append(s1).append(s2).toString();
String s4 = "HollisChuang";
```

可以发现，同样是字符串拼接，s3和s4在经过编译器编译后的实现方式并不一样。s3被转化成StringBuilder及append，而s4被直接拼接成新的字符串。

如果你感兴趣，你还能发现，String s3 = s1 + s2；经过编译之后，常量池中是有两个字符串常量的分别是 Hollis、Chuang（其实 Hollis 和 Chuang 是 String s1 = "Hollis"; 和 String s2 = "Chuang"; 定义出来的），拼接结果 HollisChuang 并不在常量池中。

如果代码只有 String s4 = "Hollis" + "Chuang";，那么常量池中将只有 HollisChuang 而没有“Hollis”和“Chuang”。

究其原因，是因为常量池要保存的是**已确定**的字面量值。也就是说，对于字符串的拼接，纯字面量和字面量的拼接，会把拼接结果作为常量保存到字符串池。

如果在字符串拼接中，有一个参数是非字面量，而是一个变量的话，整个拼接操作会被编译成StringBuilder.append，这种情况编译器是无法知道其确定值的。只有在运行期才能确定。

那么，有了这个特性了，intern 就有用武之地了。那就是很多时候，我们在程序中得到的字符串是只有在运行期才能确定的，在编译期是无法确定的，那么也就没办法在编译期被加入到常量池中。

这时候，对于那种可能经常使用的字符串，使用 intern 进行定义，每次JVM运行到这段代码的时候，就会直接把常量池中该字面值的引用返回，这样就可以减少大量字符串对象的创建了。

如 一 深 入 解 析 String#intern
(https://tech.meituan.com/in_depth_understanding_string_intern.html)文中举的一个例子：

```
static final int MAX = 1000 * 10000;  
static final String[] arr = new String[MAX];  
  
public static void main(String[] args) throws Exception {  
    Integer[] DB_DATA = new Integer[10];  
    Random random = new Random(10 * 10000);  
    for (int i = 0; i < DB_DATA.length; i++) {  
        DB_DATA[i] = random.nextInt();  
    }  
    long t = System.currentTimeMillis();  
    for (int i = 0; i < MAX; i++) {  
        arr[i] = new String(String.valueOf(DB_DATA[i % DB_DATA.length])).intern();  
    }  
  
    System.out.println((System.currentTimeMillis() - t) + "ms");  
    System.gc();  
}
```

在以上代码中，我们明确的知道，会有很多重复的相同的字符串产生，但是这些字符串的值都是只有在运行期才能确定的。所以，只能我们通过 intern 显示的将其加入常量池，这样可以减少很多字符串的重复创建。

总结

我们再回到文章开头那个疑惑：按照上面的两个面试题的回答，就是说 new String 也会检查常量池，如果有的话就直接引用，如果不存在就要在常量池创建一个，那么还要 intern 干啥？难道以下代码是没有意义的吗？

```
String s = new String("Hollis").intern();
```

而intern中说的“如果有的话就直接返回其引用”，指的是会把字面量对象的引用直接返回给定义的对象。这个过程是不会在Java堆中再创建一个String对象的。

的确，以上代码的写法其实是使用intern是没什么意义的。因为字面量Hollis会作为编译期常量被加载到运行时常量池。

之所以能有以上的疑惑，其实是对字符串常量池、字面量等概念没有真正理解导致的。有些问题其实就是这样，单个问题，自己都知道答案，但是多个问题综合到一起就蒙了。归根结底是知识的理解还停留在点上，没有串成面。

一句话概括字符串的不可变性：(内部字符数组不可变)

创建字符串对象时内部包装了字符数组，这个字符数组一旦创建初始化完成之后，存放在内存中就不可以改变。

其余任何基于该old字符串对象的修改动作都是在堆内存上新建一个new字符串，new新字符串内部的字符数组和old字符串内部的字符数组没有任何引用关系。

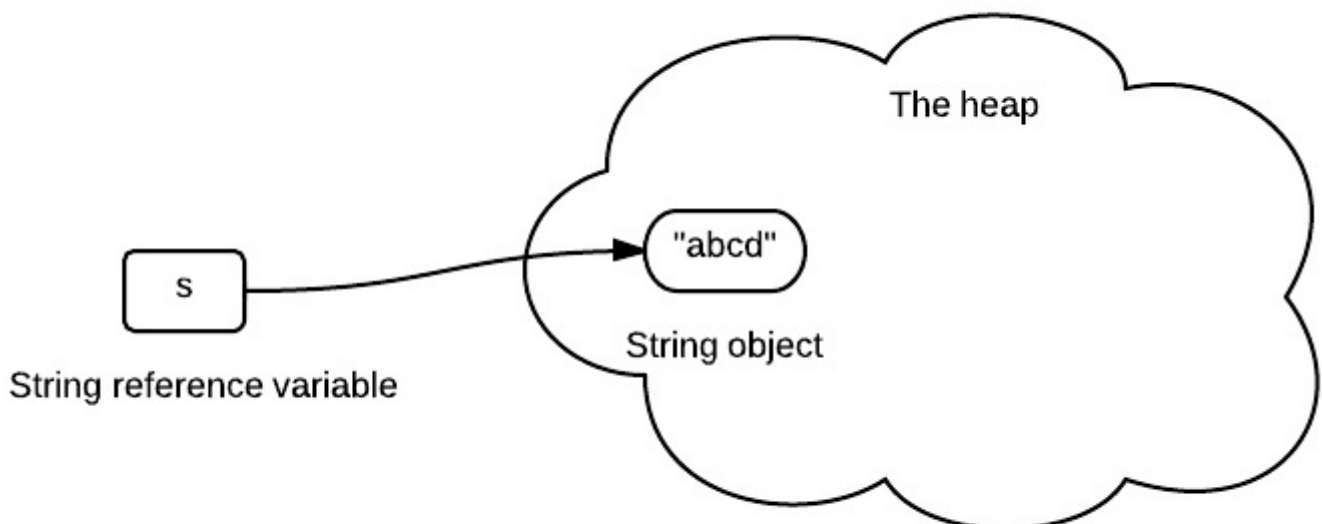
基于反射破坏字符串的不可变性：（就是改变String对象内部包装的value数组）

```
String str=new String("abc");
System.out.println("修改前：" +str);//abc
Field valueField=str.getClass().getDeclaredField("value");
valueField.setAccessible(true);
char[] temp=(char[])valueField.get(str);
temp[0]='1';
temp[1]='2';
temp[2]='3';
System.out.println("修改后：" +str);//123
```

通过反射破坏了字符串内部的private final char字符数组内部的内容。

定义一个字符串

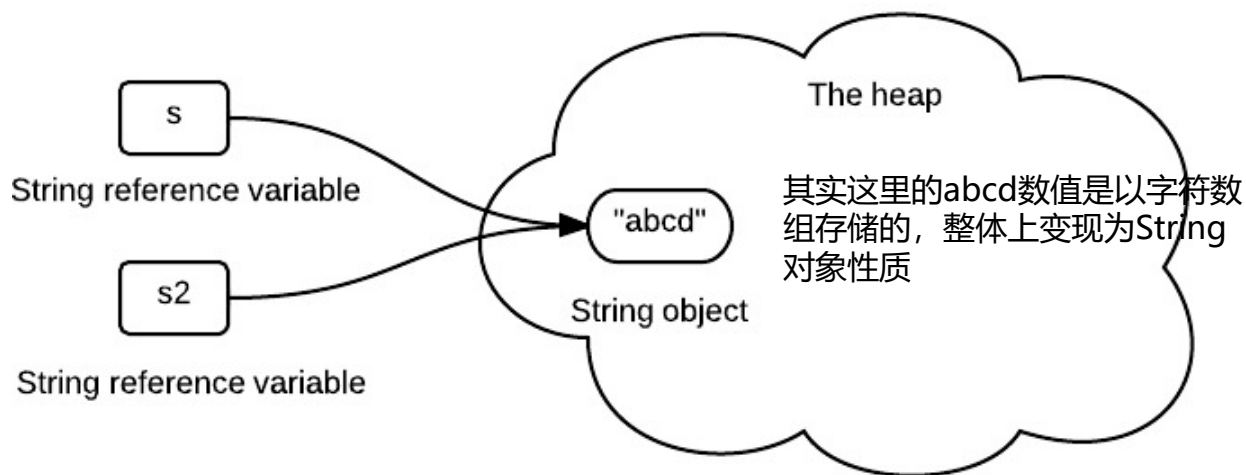
```
String s = "abcd";
```



s 中保存了string对象的引用。下面的箭头可以理解为“存储他的引用”。

使用变量来赋值变量

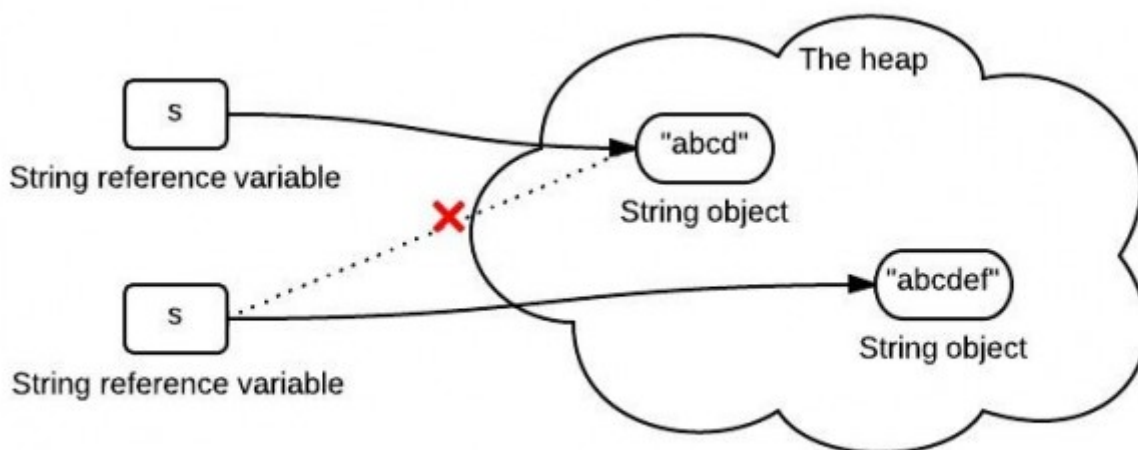
```
String s2 = s;
```



s2保存了相同的引用值，因为他们代表同一个对象。

字符串连接

```
s = s.concat("ef");
```



s 中保存的是一个重新创建出来的string对象的引用。

总结

一旦一个string对象在内存(堆)中被创建出来，他就无法被修改。特别要注意的是，String类的所有方法都没有改变字符串本身的值，都是返回了一个新的对象。

如果你需要一个可修改的字符串，应该使用StringBuffer 或者 StringBuilder。否则会有大量时间浪费在垃圾回收上，因为每次试图修改都有新的string对象被创建出来。

String 是Java中一个不可变的类，所以他一旦被实例化就无法被修改。不可变类的实例一旦创建，其成员变量的值就不能被修改。不可变类有很多优势。本文总结了为什么字符串被设计成不可变的 (<http://www.hollischuang.com/archives/1230>)。将涉及到内存、同步和数据结构相关的知识。

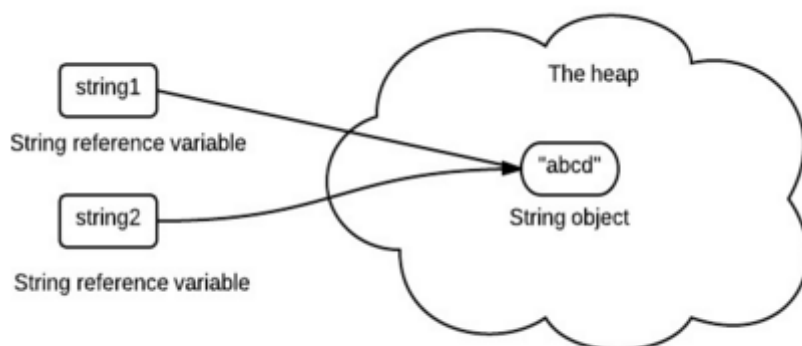
字符串池

字符串池是方法区 (<http://www.programcreek.com/2013/04/jvm-run-time-data-areas/>)中的一部分特殊存储。当一个字符串被创建的时候，首先会去这个字符串池中查找，如果找到，直接返回对该字符串的引用。字符串常量池存在的目的是减少内存占用，避免大量重复字符串存在。因此字符串的不可变性，有利于实现常量池减少重复内存占用功能。

下面的代码只会在堆中创建一个字符串

```
String string1 = "abcd";  
String string2 = "abcd";
```

下面是图示：



如果字符串可变的话，当两个引用指向指向同一个字符串时，对其中一个做修改就会影响另外一个，字符串常量池就可以一直变化，降低效率。（请记住该影响，有助于理解后面的内容）

Java中经常会用到字符串的哈希码（hashCode）。例如，在HashMap中，字符串的不可变性保证其hashCode永远保持一致，这样就可以避免一些不必要的麻烦。这也就意味着每次在使用一个字符串的hashCode的时候不用重新计算一次，这样更加高效。因为不可变性所以String可以作为

map的key，因为String的hashCode只初始化一次后不再改变，equals比较的是内部final char[]数组

在String类中，有以下代码：

```
private int hash; // this is used to cache hash code
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) { // 第一次执行hashCode方法时计算一次hash值，
        char val[] = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h; // 第二次执行hashCode直接返回hash
}
```

以上代码中 hash 变量中就保存了一个String对象的hashCode，因为String类不可变，所以一旦对象被创建，该hash值也无法改变。所以，每次想要使用该对象的hashCode的时候，直接返回即可。

使其他类的使用更加便利

在介绍这个内容之前，先看以下代码：

```
HashSet<String> set = new HashSet<String>();
set.add(new String("a"));
set.add(new String("b"));
set.add(new String("c"));
```

```
for(String a: set)
    a.value = "a";
```

```
Map<String, Object> map = new HashMap();
String key = "abc"; Object value = new Object();
map.put(key, value);
key字符串变化成"bcd", key的hashCode也因此变了
map.get(key) // null, value对象内存泄露
```

如果字符串可变，那么hashCode方法不可以只计算一次，并且如果字符串可变的话，map第一次把字符串作为key放进去的value对象，由于key字符串可变导致再次取get(key)时候为null，原始的value对象内存泄露了。

在上面的例子中，如果字符串可以被改变，那么以上用法将有可能违反Set的设计原则，因为Set要求其中的元素不可以重复。上面的代码只是为了简单说明该问题，其实String类中并没有value这个字段值。

安全性

String被广泛的使用在其他Java类中充当参数。比如网络连接、打开文件等操作。如果字符串可变，那么类似操作可能导致安全问题。因为某个方法在调用连接操作的时候，他认为会连接到某台机器，但是实际上并没有（其他引用同一String对象的值修改会导致该连接中的字符串内容被修改）。可变的字符串也可能导致反射的安全问题，因为他的参数也是字符串。

代码示例：

```
boolean connect(string s){  
  
    if (!isSecure(s)) {  
throw new SecurityException();  
}  
    //如果s在该操作之前被其他的引用所改变，那么就可能导致问题。  
    causeProblem(s);  
}
```



不可变对象天生就是线程安全的

因为不可变对象不能被改变，所以他们可以自由地在多个线程之间共享。不需要任何同步处理。

总之，String 被设计成不可变的主要目的是为了安全和高效。所以，使 String 是一个不可变类是一个很好的设计。

```
public String(String original) { //字符串的构造器  
    this.value = original.value; 写final变量  
    this.hash = original.hash;  
}
```

final变量的写操作jvm级别内存语义

写 final 域的重排序规则禁止把 final 域的写重排序到构造函数之外。

因此多个线程使用同一个ref=new String("abc") ref对象引用时候，jvm保证获取引用之前final修饰的字符数组已经被初始化完毕。

当前位置: HollisChuang's Blog (<https://www.hollischuang.com>) > Java
(<https://www.hollischuang.com/archives/category/java>) > 正文

该如何创建字符串, 使用“ ”还是构造函数? (<https://www.hollischuang.com/archives/1249>)

2016-03-02 来源: Create Java String Using " " or Constructor? (<http://www.programcreek.com/2014/03/create-java-string-by-double-quotes-vs-by-constructor/>) 分类: Java
(<https://www.hollischuang.com/archives/category/java>) 阅读(12841) 评论(1)

[GitHub 19k Star 的Java工程师成神之路, 不来了解一下吗!](#)

<https://github.com/hollischuang/toBeTopJavaer>

在Java中, 有两种方式可以创建字符串:

```
String x = "abc";  
String y = new String("abc");
```

使用双引号和构造函数之间到底有什么区别呢?

双引号 vs 构造函数

这个问题可以使用这两个简单代码实例来回答:

实例一

```
String a = "abcd";  
String b = "abcd";  
System.out.println("a == b : "+(a == b)); // true  
System.out.println("a.equals(b) : "+(a.equals(b))); // true
```

`a == b` 等于 `true` 是因为`x`和`y`指向方法区中同一个字符串常量, 他们的引用是相同的 (`==`比较的是引用) 。

当相同的字符串常量被多次创建时, 只会保存字符串常量的一份副本, 这称为“字符串驻留”。在Java中, 所有编译时字符串常量都是驻留的。

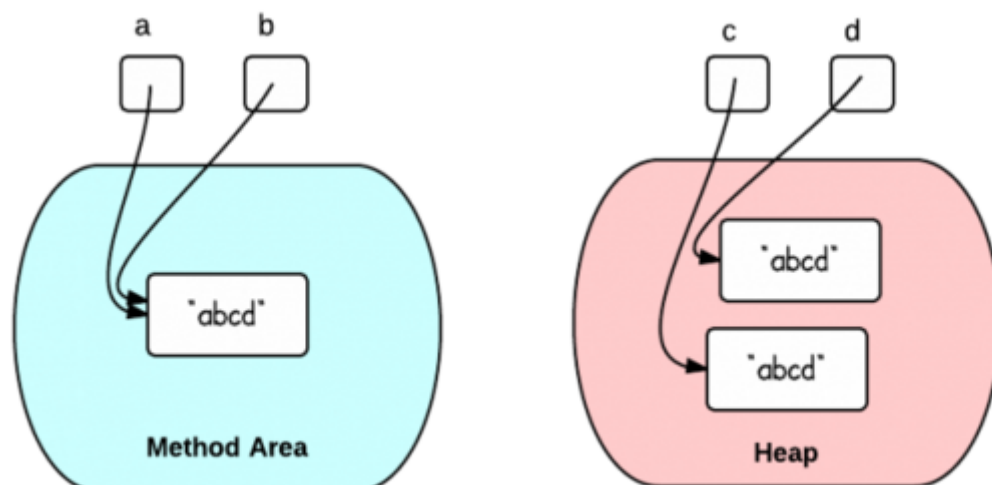
实例二

```
String c = new String("abcd");  
String d = new String("abcd");  
System.out.println("c == d : "+(c == d)); // false  
System.out.println("c.equals(d) : "+(c.equals(d))); // true
```

`c == d` 等于 `false` 是因为 `c` 和 `d` 指向堆中不同的对象，不同的对象拥有不同的内存引用。



下面图论证了以上的结论。



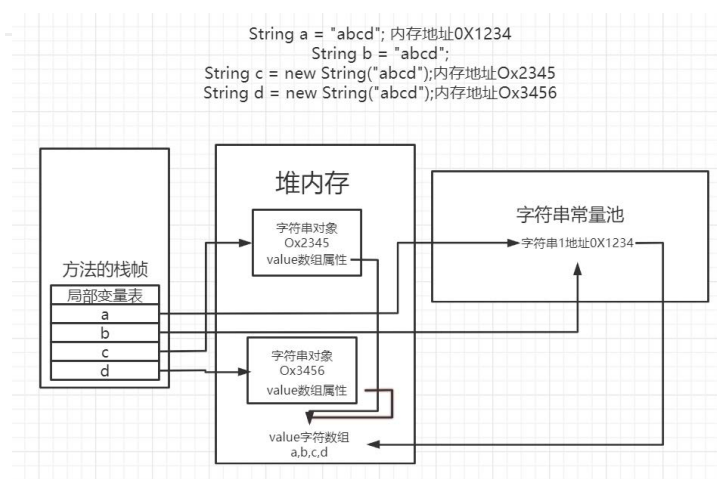
运行时字符串驻留

运行时也会发生字符串驻留，即使两个字符串是由构造函数方法创建的。

```
String c = new String("abcd").intern();
String d = new String("abcd").intern();
System.out.println("c == d : " + (c == d)); // true
System.out.println("c.equals(d) : " + (c.equals(d))); // true    (JDK1.7)
```

因为字面值 “abcd” 已经是字符串类型，那么使用构造函数方式只会创建一个额外没有用处的对象。

因此，如果你只需要创建一个字符串，你可以使用双引号的方式，如果你需要在堆中创建一个新的对象，你可以选择构造函数的方式。



三张图彻底了解JDK 6和JDK 7中substring的原理及区别

String是Java中一个比较基础的类，每一个开发人员都会经常接触到。而且，String也是面试中经常会考的知识点。String有很多方法，有些方法比较常用，有些方法不太常用。今天要介绍的subString就是一个比较常用的方法，而且围绕subString也有很多面试题。

substring(int beginIndex, int endIndex) 方法在不同版本的JDK中的实现是不同的。了解他们的区别可以帮助你更好的使用他。为简单起见，后文中用 substring() 代表 substring(int beginIndex, int endIndex) 方法。

substring() 的作用

substring(int beginIndex, int endIndex) 方法截取字符串并返回其[beginIndex,endIndex-1]范围内的内容。

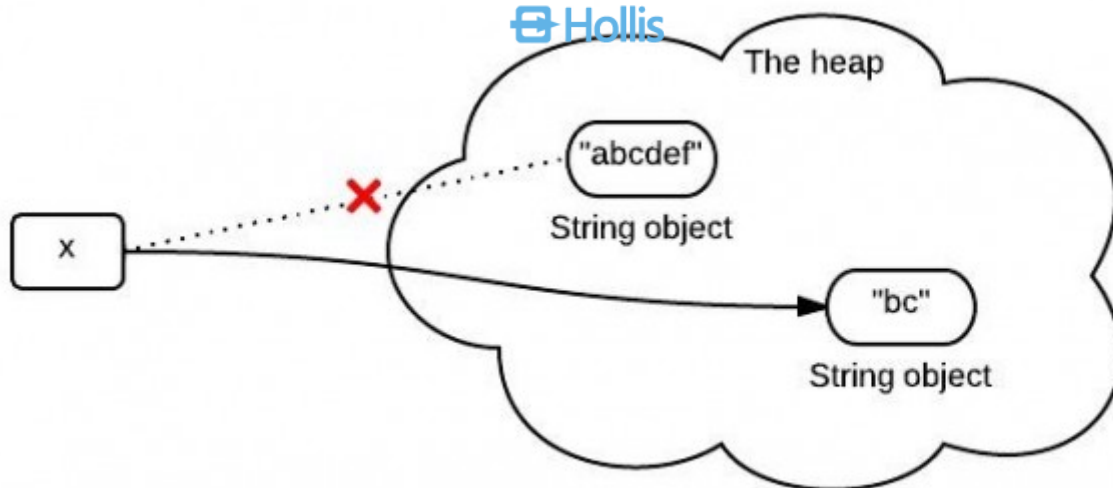
```
String x = "abcdef";  
x = x.substring(1,3);  
System.out.println(x);
```

输出内容：

```
bc
```

调用substring()时发生了什么？

你可能知道，因为x是不可变的，当使用 x.substring(1,3) 对x赋值的时候，它会指向一个全新的字符串：

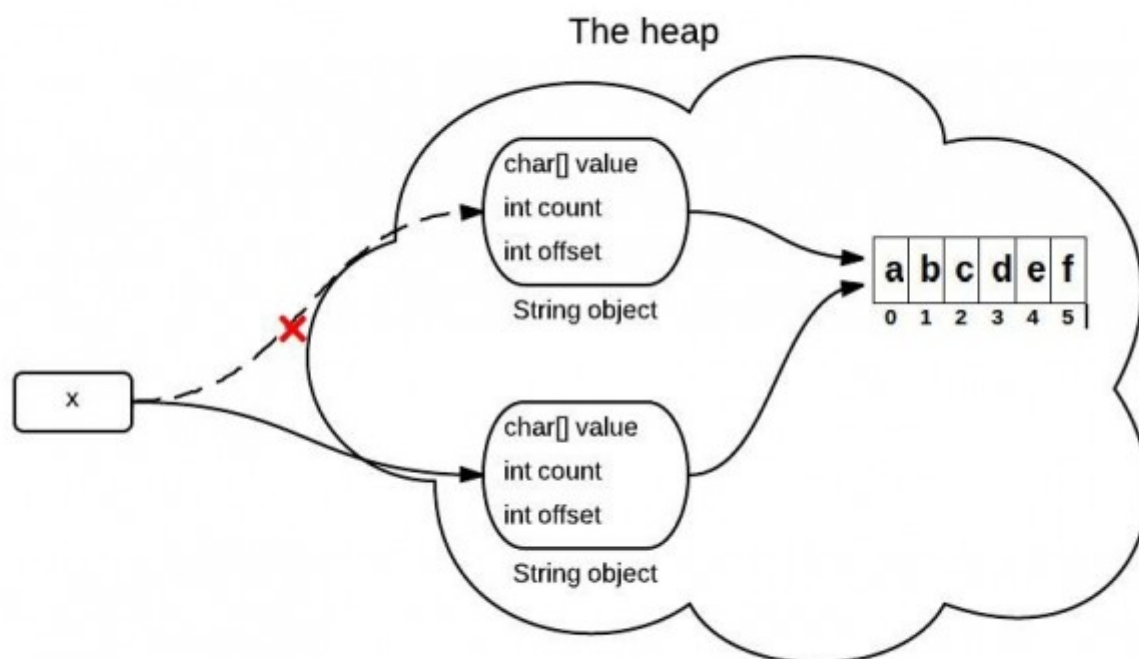


然而，这个图不是完全正确的表示堆中发生的事情。因为在jdk6 和 jdk7中调用substring时发生的事情并不一样。

JDK 6中的substring

String是通过字符数组实现的。在jdk 6 中，String类包含三个成员变量：`char value[]`，`int offset`，`int count`。他们分别用来存储真正的字符数组，数组的第一个位置索引以及字符串中包含的字符个数。

当调用substring方法的时候，会创建一个新的string对象，但是这个string的值仍然指向堆中的同一个字符数组。这两个对象中只有count和offset 的值是不同的。



下面是证明上说观点的Java源码中的关键代码：

```
//JDK 6
String(int offset, int count, char value[]) {
    this.value = value;
    this.offset = offset;
    this.count = count;
}

public String substring(int beginIndex, int endIndex) {
    //check boundary
    return new String(offset + beginIndex, endIndex - beginIndex, value);
}
```

JDK 6中的substring导致的问题

如果你有一个很长很长的字符串，但是当你使用substring进行切割的时候你只需要很短的一段。这可能导致性能问题，因为你需要的只是一小段字符序列，但是你却引用了整个字符串（因为这个非常长的字符数组一直在被引用，所以无法被回收，就可能导致内存泄露）。在JDK 6中，一般用以下方式来解决该问题，原理其实就是生成一个新的字符串并引用他。

```
x = x.substring(x, y) + ""
```

关于JDK 6中subString的使用不当会导致内存系列已经被官方记录在Java Bug Database中：

Type: Bug
Component: core-libs
Sub-Component: java.lang
Affected Version: 5.0

Priority: P4
Status: Closed
Resolution: Duplicate
OS: windows_xp
CPU: x86

Submitted: 2005-07-05
Updated: 2011-02-16
Resolved: 2005-07-26

Related Reports

Duplicate : [JDK-4513622](#) - (str) keeping a substring of a field prevents GC for object

Description

FULL PRODUCT VERSION :

java version "1.5.0_02"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_02-b09)
Java HotSpot(TM) Client VM (build 1.5.0_02-b09, mixed mode, sharing)

ADDITIONAL OS VERSION INFORMATION :

Microsoft Windows XP [Version 5.1.2600]

A DESCRIPTION OF THE PROBLEM :

The bug with ID 4637640 though marked as fixed for JDK 1.5 is still present in the JDK 1.5 releases.

By the way bug 4637640 was never a duplicate of bug 4546734! They're only partially related and maybe 4546734 was fixed but 4637640 is still there in JDK 1.5 release.

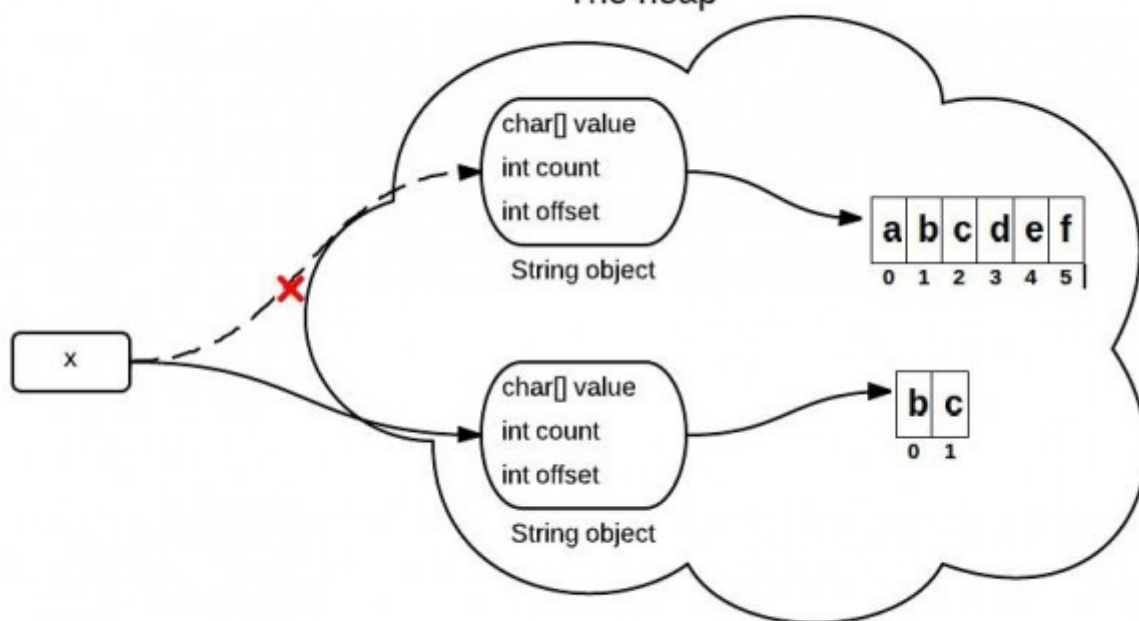
Since this is not a functional bug is not possible to provide a unit test. It's a memory leaking behaviour which can only be observed using a profiler (I used the new Netbeans Profiler) or similar memory monitoring tools.

In my opinion everything necessary is said in bug report 4637640: the

内存泄露：在计算机科学中，内存泄漏指由于疏忽或错误造成程序未能释放已经不再使用的内存。内存泄漏并非指内存存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

JDK 7 中的substring

上面提到的问题，在jdk 7中得到解决。在jdk 7 中，substring方法会在堆内存中创建一个新的数组。



Java源码中关于这部分的主要代码如下：

```
//JDK 7
public String(char value[], int offset, int count) {
    //check boundary
    this.value = Arrays.copyOfRange(value, offset, offset + count);
}

public String substring(int beginIndex, int endIndex) {
    //check boundary
    int subLen = endIndex - beginIndex;
    return new String(value, beginIndex, subLen);
}
```

以上是JDK 7中的`subString`方法，其使用 `new String` 创建了一个新字符串，避免对老字符串的引用。从而解决了内存泄露问题。

所以，如果你的生产环境中使用的JDK版本小于1.7，当你使用String的`subString`方法时一定要注意，避免内存泄露。

在不使用任何带有自动补全功能IDE的情况下，如何获取一个数组的长度？以及，如何获取一个字符串的长度？

这个问题我问过不同水平的程序员，包括初级和中级水平的。他们都不能准确而自信地回答这个问题（如果你能很准确很自信的回答这个问题，那么证明针对这一知识点你比大多数中级程序员掌握的好）。由于现在很多IDE都有代码补全功能，这使得开发人员在很多问题上理解的很肤浅。本文将介绍几个关于Java数组的关键概念。

上面问题的正确回答姿势应该是这样的：

```
int[] arr = new int[3];
System.out.println(arr.length); // 使用length获取数组的长度

String str = "abc";
System.out.println(str.length()); // 使用length()获取字符串的长度
```

那么问题来了，为什么数组有 `length` 属性，而字符串没有？或者，为什么字符串有 `length()` 方法，而数组没有？

为什么数组有length属性？

首先，数组是一个容器对象（Java中的数组是对象吗？ (http://blog.csdn.net/zhangjg_blog/article/details/16116613#t1))，其中包含固定数量的同一类型的值。一旦数组被创建，他的长度就是固定的了。数组的长度可以作为 `final` 实例变量的长度。因此，长度可以被视为一个数组的属性。

有两种创建数组的方法：1、通过数组表达式创建数组。2、通过初始化值创建数组。无论使用哪种方式，一旦数组被创建，其大小就固定了。

使用表达式创建数组方式如下，该方式指明了元素类型、数组的维度、以及至少一个维度的数组的长度。

该声明方式是符合要求的，因为他指定了一个维度的长度（该数组的类型为int，维度为2，第一维度的长度为3）

```
int[][] arr = new int[3][];
```

使用数组初始化的方式创建数组时需要提供所有的初始值。形式是使用 { 和 } 将所有初始值括在一起并用 , 隔开。

```
int[] arr = {1,2,3};
```

注：

这里可能会有一个疑问，既然数组大小是初始化时就规定好的，那么 `int[][] arr = new int[3][];` 定义的数组并没有给出数组的第二维的大小，那么这个 `arr` 的长度到底是如何“规定好”的呢？

其实，`arr` 的长度就是3。其实Java中所有的数组，无论几维，其实都是一维数组。例如`arr`，分配了3个空间，每个空间存放一个一维数组的地址，这样就成了“二维”数组。但是对于`arr`来说，他的长度就是3。

```
int[][] a=new int[3][];  
System.out.println(a.length);//3  
int[][] b=new int[3][5];  
System.out.println(b.length);//3
```

Java中为什么没有定义一个类似String一样Array类

因为数组也是对象，所以下面的代码也是合法的：

```
Object obj = new int[10];
```

数组包含所有从 `Object` 继承下来方法（[Java 中数组的继承关系 \(http://blog.csdn.net/zhangjg_blog/article/details/16116613#t4\)](http://blog.csdn.net/zhangjg_blog/article/details/16116613#t4)），除 `clone()` 之外。为什么没有一个 `array` 类呢？在Java中没有 `Array.java` 文件。一个简单的解释是它被隐藏起来了（注：Java中的数组有点类似于基本数据类型，是一个内建类型，并没有实际的类与他对应）。你可以

思考这样一个问题——如果有一个Array类，那它会像什么样？它会仍然需要一个数组来存放所有的数组元素，对吗？因此，定义出一个Array类不是一个好的主意。（译者注：这里可能有点绕，道理有点类似于：鸡生蛋蛋生鸡问题，可能比喻也不是很恰当，请读者自行理解）

事实上我们可以获得数组的类定义，通过下面的代码：

```
int[] arr = new int[3];
System.out.println(arr.getClass());
```

输出：

```
class [I
```

“class [I” 代表着“ 成员类型是int的“数组” 的class对象运行时类型的签名

为什么String有length()方法？

String背后的数据结构是一个char数组,所以没有必要来定义一个不必要的属性（因为该属性在char数值中已经提供了）。和C不同的是，Java中char的数组并不等于字符串，虽然String的内部机制是char数组实现的。（注：C语言中，并没有String类，定义字符串通常使用 `char string[6] = "hollis";` 的形式）

注：要想把char[]转成字符串有以下方式：

```
char []s = {'a','b','c'};
String string1 = s.toString();
String string2 = new String(s);
String string3 = String.valueOf(s);
```

字符串length()方法

```
public int length() {

    return value.length;

}
```




设为星标 ★



Java之道

📍 扫码关注不迷路

以术识道 · 以道御术 · 传道授业解惑

👍 赞(9)

分享到: 更多 (0)

标签: String (<https://www.hollischuang.com/archives/tag/string>)

数组 (<https://www.hollischuang.com/archives/tag/%e6%95%b0%e7%bb%84>)

上一篇

该如何创建字符串, 使用“ ”还是构造函数?
(<https://www.hollischuang.com/archives/1249>)

下一篇

在Java中如何高效的判断数组中是否包含某个元素
(<https://www.hollischuang.com/archives/1269>)

相关推荐

- 深入解析String中的intern (<https://www.hollischuang.com/archives/1551>)
- 《成神之路-基础篇》Java基础知识——String相关 (<https://www.hollischuang.com/archives/1330>)
- 在Java中如何高效的判断数组中是否包含某个元素 (<https://www.hollischuang.com/archives/1269>)
- 该如何创建字符串, 使用“ ”还是构造函数? (<https://www.hollischuang.com/archives/1249>)
- 为什么Java要把字符串设计成不可变的 (<https://www.hollischuang.com/archives/1246>)
- 三张图彻底了解JDK 6和JDK 7中substring的原理及区别 (<https://www.hollischuang.com/archives/1232>)
- 三张图彻底了解Java中字符串的不变性 (<https://www.hollischuang.com/archives/1230>)
- Java 7 源码学习系列 (一) ——String (<https://www.hollischuang.com/archives/99>)

评论 5

Java中的Switch对整型、字符型、字符串型的具体实现细节

Java 7中，switch的参数可以是String类型了，这对我们来说是一个很方便的改进。到目前为止switch支持这样几种数据类型：byte short int char String。但是，作为一个程序员我们不仅要知道他有多么好用，还要知道它是如何实现的，switch对整型的支持是怎么实现的呢？对字符型是怎么实现的呢？String类型呢？有一点Java开发经验的人这个时候都会猜测switch对String的支持是使用equals()方法和hashCode()方法。那么到底是不是这两个方法呢？接下来我们就看一下，switch到底是如何实现的。

一、switch对整型支持的实现

下面是一段很简单的Java代码，定义一个int型变量a，然后使用switch语句进行判断。执行这段代码输出内容为5，那么我们将下面这段代码反编译，看看他到底是怎么实现的。

```
public class switchDemoInt {
```

tableswitch一般针对case比较密集的情况，比如case 1 3 5或者case a b c，但是如果比较分散使用lookupswitch比如1 3 10。一般来说tableswitch性能更好，比如计算目标case地址，通过已知index-tableswitch.low得到就可以直接得到目标case的字节码指令地址，时间复杂度O(1)，如果是lookupswitch计算目标case地址，通过low < index < heigh二分法找到目标case地址时间复杂度O(log(n))，所以推荐switch case时候，使用连续的case性能更高。

```
    public static void main(String[] args) {
        switch (a) {
            case 1:
                System.out.println(1);
                break;
            case 3:
                System.out.println(3);
                break;
            case 5:
                System.out.println(5);
                break;
            default:
                break;
        }
    }
}
//output 5
```

```
0 iconst_5
1 istore_1
2 iload_1
3 tableswitch 1 to 5 1: 36 (+33)
   2: 66 (+63) 自动添加的case
   3: 46 (+43) 如果数值是3跳转到字节码46位置
   4: 66 (+63) 自动添加的case
   5: 56 (+53) 如果数值是5跳转到字节码56位置
   default: 66 (+63)
36 getstatic #2 <java/lang/System.out>
39 iconst_1
40 invokevirtual #3 <java/io/
   PrintStream.println>
43 goto 66 (+23)
46 getstatic #2 <java/lang/System.out>
49 iconst_3
50 invokevirtual #3 <java/io/
   PrintStream.println>
53 goto 66 (+13)
56 getstatic #2 <java/lang/System.out>
59 iconst_5
60 invokevirtual #3 <java/io/
   PrintStream.println>
63 goto 66 (+3)
66 return
```

反编译后的代码如下：

```

public class switchDemoInt
{
    public switchDemoInt()
    {
    }
    public static void main(String args[])
    {
        int a = 5;
        switch(a)
        {
            case 1: // '\001'
                System.out.println(1);
                break;

            case 5: // '\005'
                System.out.println(5);
                break;
        }
    }
}

```

我们发现，反编译后的代码和之前的代码比较除了多了两行注释以外没有任何区别，那么我们就知道，**switch对int的判断是直接比较整数的值。**

二、switch对字符型支持的实现

直接上代码：

```

public class switchDemoInt {
    public static void main(String[] args) {
        char a = 'b';
        switch (a) {
            case 'a':
                System.out.println('a');
                break;
            case 'b':
                System.out.println('b');
                break;
            default:
                break;
        }
    }
}

```

编译后的代码如下：`public class switchDemoChar`

```
public class switchDemoChar
```

```
{
```

```
    public switchDemoChar()
```

```
{
}
```

```
    public static void main(String args[])
```

```
{
```

```
        char a = 'b';
```

```
        switch(a)
```

```
{
```

```
    case 97: // 'a'
```

```
        System.out.println('a');
```

```
        break;
```

```
    case 98: // 'b'
```

```
        System.out.println('b');
```

```
        break;
```

```
}
```

```
}
```

```
}
```

lookupswitch字节码指令

通过键匹配和跳转访问跳转表。lookupswitch指令的表match-offset对必须按match以递增的数字顺序排序。该键必须是int类型，并从操作数堆栈中弹出。将key键与匹配值进行比较。如果等于其中之一，则通过将相应的偏移量添加到此lookupswitch指令的操作码的地址来计算目标地址（字节码指令地址）。如果密钥key与任何匹配值都不匹配，则通过添加默认值来计算目标地址到此lookupswitch指令的操作码的地址。然后执行在目标地址处继续。lookupswitch中的case按照递增排序，寻找case时时间复杂度O(log(n)),类似于一个map不过key是有序递增的

0 bipush 98 整数98push到操作数栈中

2 istore_1 98写入到局部变量表中1slot位置

3 iload_1 加载局部变量表1 slot位置数据到操作数栈

4 lookupswitch 2 操作数栈中pop出数据进行比较

97: 32 (+28) 如果数值是97则跳转到32字节码位置

98: 43 (+39) 如果数值是98则跳转到39字节码位置

default: 51 (+47)

32 getstatic #2 <java/lang/System.out>

35 bipush 97 操作数栈压入97

37 invokevirtual #3 <java/io/PrintStream.println> 打印输出97

40 goto 51 (+11) goto return返回

43 getstatic #2 <java/lang/System.out>

46 bipush 98 操作数栈压入98

48 invokevirtual #3 <java/io/PrintStream.println>弹出操作数栈98并打印

51 return

通过以上的代码作比较我们发现：对char类型进行比较的时候，实际上比较的是ascii码，编译器会把char型变量转换成对应的int型变量

三、switch对字符串支持的实现

还是先上代码：

```
public class switchDemoString {
```

```
    public static void main(String[] args) {
```

```
        String str = "world";
```

```
        switch (str) {
```

```
    case "hello":
```

```
        System.out.println("hello");
```

```
        break;
```

```
    case "world":
```

```
        System.out.println("world");
```

```
        break;
```

```
    default:
```

```
        break;
```

```
}
```

```
}
```

```
}
```

如果两个字符串hashCode相同switch怎么处理？

String str01 = "Aa"; IDEA反编译后的代码

```
byte var3 = -1;
```

```
switch(str01.hashCode()) {
```

case 2112://字符串Aa, BB都是2112, 此时除了equals之后还要加上if判断属于哪个equals

```
if (str01.equals("BB")) {
```

```
    var3 = 1;
```

```
} else if (str01.equals("Aa")) {
```

```
    var3 = 0;
```

```
}
```

```
break;
```

```
case 3168:
```

```
if (str01.equals("cc")) {
```

```
    var3 = 2;
```

```
}
```

```
}
```

String str01 = "Aa";原始代码Aa,与BB的hashCode相同

```
switch (str01) {
```

```
    case "Aa":
```

```
        System.out.println("A");
```

```
        break;
```

```
    case "BB":
```

```
        System.out.println("B");
```

```
        break;
```

```
}
```

对代码进行反编译：

```
public class switchDemoString
```

```
{
```

```
    public switchDemoString()
```

```
    {
```

```
    }
```

```
    public static void main(String args[])
```

```
    {
```

```
        String str = "world";
```

```
        String s;
```

```
        switch((s = str).hashCode())
```

```
        {
```

```
        default:
```

```
            break;
```

```
        case 99162322:
```

```
            if(s.equals("hello"))
```

```
                System.out.println("hello");
```

```
            break;
```

```
        case 113318802:
```

```
            if(s.equals("world"))
```

```
                System.out.println("world");
```

```
            break;
```

```
        }
```

```
    }
```

```
}
```

所以字符串先使用hashCode进行比较，后比较equals方法，由于hashCode比较大随机，所以底层使用lookupswitch字节码比较指令

看到这个代码，你知道原来字符串的switch是通过 equals() 和 hashCode() 方法来实现的。**记住，switch中只能使用整型**，比如 byte 。 short ， char (ackii码是整型)以及 int 。还好 hashCode() 方法返回的是 int ，而不是 long 。通过这个很容易记住 hashCode 返回的是 int 这个事实。仔细看下可以发现，进行 switch 的实际是哈希值，然后通过使用equals方法比较进行安全检查，这个检查是必要的，因为哈希可能会发生碰撞。因此它的性能是不如使用枚举进行switch或者使用纯整数常量，但这也不是很差。因为Java编译器只增加了一个 equals 方法，如果你比较的是字符串字面量的话会非常快，比如 " abc " == " abc " 。如果你把 hashCode() 方法的调用也考虑进来了，那么还会再多一次的调用开销，因为字符串一旦创建了，它就会把哈希值缓存起来。因此如果这个 siwtch 语句是用在一个循环里的，比如逐项处理某个值，或者游戏引擎循环地渲染屏幕，这里 hashCode() 方法的调用开销其实不会很大。

好，以上就是关于switch对整型、字符型、和字符串型的支持的实现方式，总结一下我们可以发现，**其实swich只支持一种数据类型，那就是整型，其他数据类型都是转换成整型之后在使用switch的。**

由于byte,shore底层字节码指令都是复用int的指令，因此情况和对整形的支持一样

深入解析String中的intern

原文地址：[深入解析 String#intern \(http://tech.meituan.com/in_depth_understanding_string_intern.html\)](http://tech.meituan.com/in_depth_understanding_string_intern.html)

引言

在 JAVA 语言中有8中基本类型和一种比较特殊的类型 `String`。这些类型为了使他们在运行过程中速度更快，更节省内存，都提供了一种常量池的概念。常量池就类似一个JAVA系统级别提供的缓存。

8种基本类型的常量池都是系统协调的，`String`类型的常量池比较特殊。它的主要使用方法有两种：

直接使用双引号声明出来的 `String` 对象会直接存储在常量池中。


如果不是用双引号声明的 `String` 对象，可以使用 `String` 提供的 `intern` 方法。`intern` 方法会从字符串常量池中查询当前字符串是否存在，若不存在就会将当前字符串放入常量池中

接下来我们主要来谈一下 `String#intern` 方法。

一, `intern` 的实现原理

首先深入看一下它的实现原理。

1, JAVA 代码



```

/**
 * Returns a canonical representation for the string object.
 *
 * <p>
 * A pool of strings, initially empty, is maintained privately by the
 * class String.
 *
 * <p>
 * When the intern method is invoked, if the pool already contains a
 * string equal to this String object as determined by
 * the {@link #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.
 *
 * <p>
 * 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.
 *
 * <p>
 * All literal strings and string-valued constant expressions are
 * interned. String literals are defined in section 3.10.5 of the
 * <cite>The Java® Language Specification</cite>.
 *
 * @return a string that has the same contents as this string, but is
 *         guaranteed to be from a pool of unique strings.
 */
public native String intern();

```

String#intern 方法中看到，这个方法是一个 native 的方法，但注释写的非常明了。“如果常量池中存在当前字符串，就会直接返回当前字符串。如果常量池中不存在此字符串，会将此字符串放入常量池中后，再返回”。

2, native 代码

在 jdk7 后，oracle 接管了 JAVA 的源码后就不对外开放了，根据 jdk 的主要开发人员声明 openJdk7 和 jdk7 使用的是同一分主代码，只是分支代码会有些许的变动。所以可以直接跟踪 openJdk7 的源码来探究 intern 的实现。

native 实现代码：

\openjdk7\jdk\src\share\native\java\lang\String.c

```

Java_java_lang_String_intern(JNIEnv *env, jobject this)
{
    return JVM_InternString(env, this);
}

```

\openjdk7\hotspot\src\share\vm\prims\jvm.h

```

/*
 * java.lang.String
 */
JNIEXPORT jstring JNICALL
JVM_InternString(JNIEnv *env, jstring str);

```



\openjdk7\hotspot\src\share\vm\prims\jvm.cpp

```

// String support //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
JVM_ENTRY(jstring, JVM_InternString(JNIEnv *env, jstring str))
    JVMWrapper("JVM_InternString");
    JvmtiVMObjectAllocEventCollector oam;
    if (str == NULL) return NULL;
    oop string = JNIHandles::resolve_non_null(str);
    oop result = StringTable::intern(string, CHECK_NULL);
    return (jstring) JNIHandles::make_local(env, result);
JVM_END

```

\openjdk7\hotspot\src\share\vm\classfile\symbolTable.cpp

```

oop StringTable::intern(Handle string_or_null, jchar* name,
                        int len, TRAPS) {
    unsigned int hashValue = java_lang_String::hash_string(name, len);
    int index = the_table()->hash_to_index(hashValue);
    oop string = the_table()->lookup(index, name, len, hashValue);
    // Found
    if (string != NULL) return string;
    // Otherwise, add to symbol to table
    return the_table()->basic_add(index, string_or_null, name, len,
                                hashValue, CHECK_NULL);
}

```

\openjdk7\hotspot\src\share\vm\classfile\symbolTable.cpp

```

oop StringTable::lookup(int index, jchar* name,
                        int len, unsigned int hash) {
    for (HashtableEntry<oop>* l = bucket(index); l != NULL; l = l->next()) {
        if (l->hash() == hash) {
            if (java_lang_String::equals(l->literal(), name, len)) {
                return l->literal();
            }
        }
    }
    return NULL;
}

```

它的大体实现结构就是:

JAVA 使用 jni 调用c++实现的 `StringTable` 的 `intern` 方法, `StringTable` 的 `intern` 方法跟Java中



的 `HashMap` 的实现是差不多的, 只是不能自动扩容。默认大小是1009。

要注意的是, `String`的`String Pool`是一个固定大小的 `Hashtable` , 默认值大小长度是1009, 如果放进`String Pool`的`String`非常多, 就会造成 `Hash` 冲突严重, 从而导致链表会很长, 而链表长了后直接会造成的影响就是当调用 `String.intern` 时性能会大幅下降 (因为要一个一个找) 。

在 `jdk6`中 `StringTable` 是固定的, 就是1009的长度, 所以如果常量池中的字符串过多就会导致效率下降很快。在`jdk7`中, `StringTable`的长度可以通过一个参数指定:

```
-XX:StringTableSize=99991
```

二, `jdk6` 和 `jdk7` 下 `intern` 的区别 可以参考jvm中String内存分配文章后, 很好理解

相信很多 JAVA 程序员都做做类似 `String s = new String("abc")` 这个语句创建了几个对象的题目。这种题目主要就是为了考察程序员对字符串对象的常量池掌握与否。上述的语句中是创建了2个对象, 第一个对象是" abc" 字符串存储在常量池中, 第二个对象在JAVA Heap中的 `String` 对象。

来看一段代码:

```
public static void main(String[] args) {  
    String s = new String("1");  
    s.intern();  
    String s2 = "1";  
    System.out.println(s == s2);  
  
    String s3 = new String("1") + new String("1");  
    s3.intern();  
    String s4 = "11";  
    System.out.println(s3 == s4);  
}
```

打印结果是

```
jdk6 下false false  
jdk7 下false true
```

具体为什么稍后再解释, 然后将 `s3.intern();` 语句下调一行, 放到 `String s4 = "11";` 后面。将 `s.intern();` 放到 `String s2 = "1";` 后面。是什么结果呢

```

public static void main(String[] args) {
    String s = new String("1");
    String s2 = "1";
    s.intern();
    System.out.println(s == s2);

    String s3 = new String("1") + new String("1");
    String s4 = "11";
    s3.intern();
    System.out.println(s3 == s4);
}

```

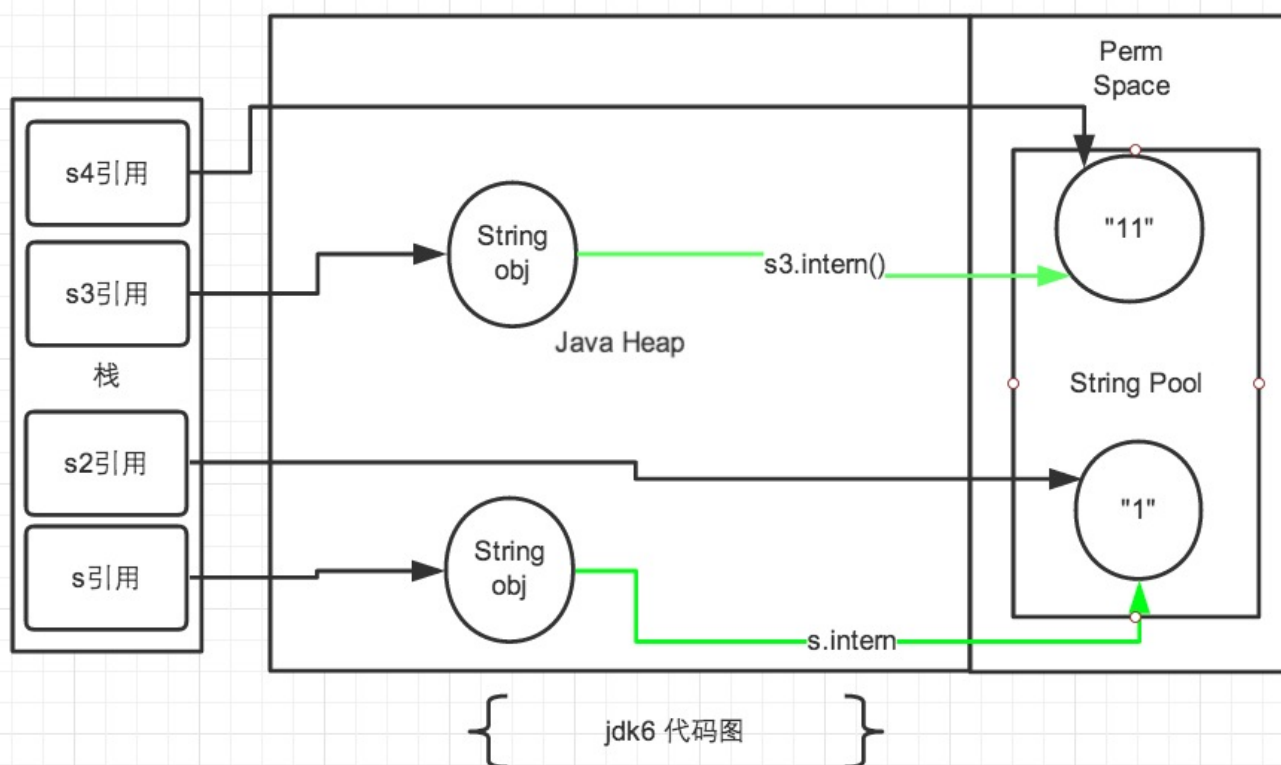
打印结果为：

```

jdk6 下 false false
jdk7 下 false false

```

1, jdk6中的解释



注：图中绿色线条代表 string 对象的内容指向。黑色线条代表地址指向。

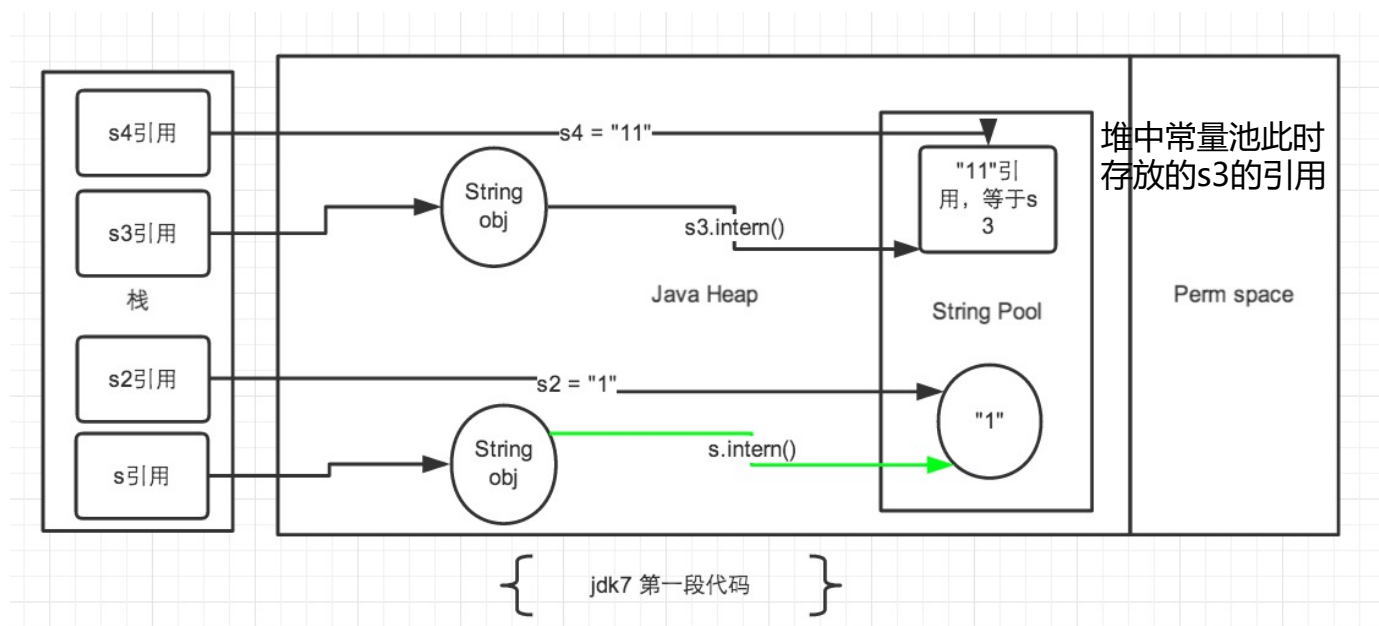
如上图所示。首先说一下 jdk6 中的情况，在 jdk6 中上述的所有打印都是 `false` 的，因为 jdk6 中的常量池是放在 Perm 区中的，Perm 区和正常的 JAVA Heap 区域是完全分开的。上面说过如果是使用引号声明的字符串都是会直接在字符串常量池中生成，而 new 出来的 String 对象是放在 JAVA Heap 区域。所以拿一个 JAVA Heap 区域的对象地址和字符串常量池的对象地址进行比较肯定是不相同的，即使调用 `String.intern` 方法也是没有任何关系的。

2, jdk7中的解释



再说说 jdk7 中的情况。这里要明确一点的是，在 Jdk6 以及以前的版本中，字符串的常量池是放在堆的 Perm 区的，Perm 区是一个类静态的区域，主要存储一些加载类的信息，常量池，方法片段等内容，默认大小只有 4m，一旦常量池中大量使用 intern 是会直接产生 `java.lang.OutOfMemoryError: PermGen space` 错误的。所以在 jdk7 的版本中，字符串常量池已经从 Perm 区移到正常的 Java Heap 区域了。为什么要移动，Perm 区域太小是一个主要原因，当然据消息称 jdk8 已经直接取消了 Perm 区域，而新建立了一个元区域。应该是 jdk 开发者认为 Perm 区域已经不适合现在 JAVA 的发展了。

正式因为字符串常量池移动到 JAVA Heap 区域后，再来解释为什么会有上述的打印结果。



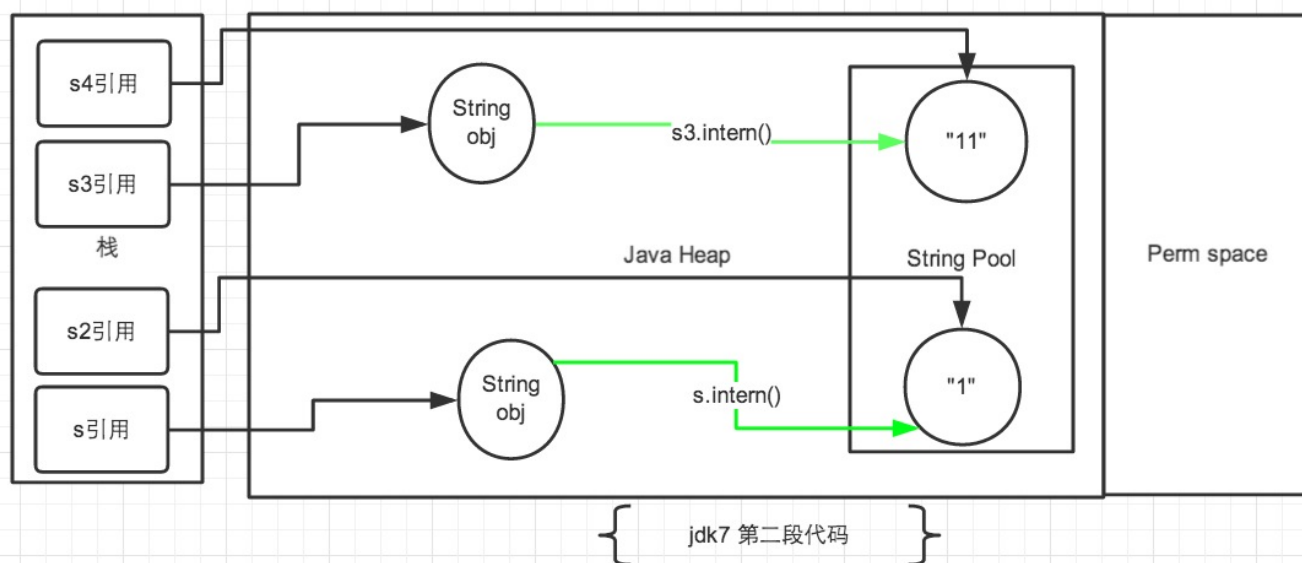
在第一段代码中，先看 s3和s4字符串。`String s3 = new String("1") + new String("1");`，这句代码中现在生成了2最终个对象，是字符串常量池中的“1”和 JAVA Heap 中的 s3引用指向的对象。中间还有2个匿名的 `new String("1")` 我们不去讨论它们。此时s3引用对象内容是“11”，但此时常量池中是没有“11”对象的。

接下来 `s3.intern();` 这一句代码，是将 s3中的“11”字符串放入 String 常量池中，因为此时常量池中不存在“11”字符串，因此常规做法是跟 jdk6 图中表示的那样，在常量池中生成一个“11”的对象，关键点是 jdk7 中常量池不在 Perm 区域了，这块做了调整。常量池中不需要再存储一份对象了，可以直接存储堆中的引用。这份引用指向 s3 引用的对象。也就是说引用地址是相同的。

最后 `String s4 = "11";` 这句代码中“11”是显示声明的，因此会直接去常量池中创建，创建的时候发现已经有这个对象了，此时也就是指向 s3 引用对象的一个引用。所以 s4 引用就指向和 s3 一样了。因此最后的比较 `s3 == s4` 是 true。

再看 s 和 s2 对象。 `String s = new String("1");` 第一句代码，生成了2个对象。常量池中的“1”和 JAVA Heap 中的字符串对象。 `s.intern();` 这一句是 s 对象去常量池中寻找后发现“1”已经在常量池里了。

接下来 `String s2 = "1";` 这句代码是生成一个 s2的引用指向常量池中的“1”对象。结果就是 s 和 s2 的引用地址明显不同。图中画的很清晰。



来看第二段代码，从上边第二幅图中观察。第一段代码和第二段代码的改变就是 `s3.intern();` 的顺序是放在 `String s4 = "11";` 后了。这样，首先执行 `String s4 = "11";` 声明 `s4` 的时候常量池中是不存在“11”对象的，执行完毕后，“11”对象是 `s4` 声明产生的新对象。然后再执行 `s3.intern();` 时，常量池中“11”对象已经存在了，因此 `s3` 和 `s4` 的引用是不同的。

第二段代码中的 `s` 和 `s2` 代码中，`s.intern();`，这一句往后放也不会有什么影响了，因为对象池中在执行第一句代码 `String s = new String("1");` 的时候已经生成“1”对象了。下边的 `s2` 声明都是直接从常量池中取地址引用的。`s` 和 `s2` 的引用地址是不会相等的。

小结

从上述的例子代码可以看出 jdk7 版本对 `intern` 操作和常量池都做了一定的修改。主要包括2点：

将String常量池 从 Perm 区移动到了 Java Heap区

`String#intern` 方法时，如果存在堆中的对象，会直接保存对象的引用，而不会重新创建对象。

三，使用 intern

1,intern 正确使用例子

接下来我们来看一下一个比较常见的使用 `String#intern` 方法的例子。

代码如下：

```

static final int MAX = 1000 * 10000;
static final String[] arr = new String[MAX];

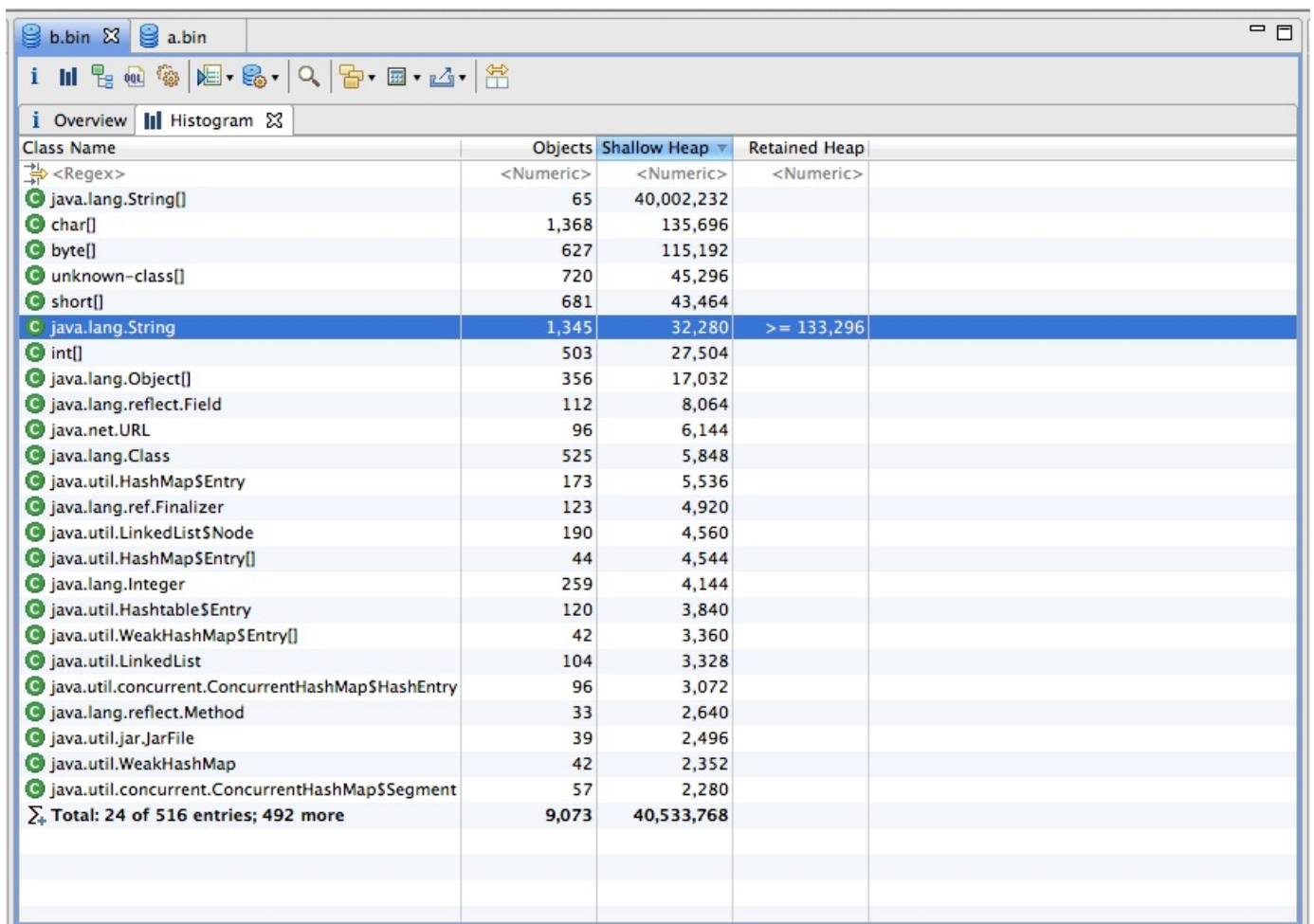
public static void main(String[] args) throws Exception {
    Integer[] DB_DATA = new Integer[10];
    Random random = new Random(10 * 10000);
    for (int i = 0; i < DB_DATA.length; i++) {
        DB_DATA[i] = random.nextInt();
    }
    long t = System.currentTimeMillis();
    for (int i = 0; i < MAX; i++) {
        //arr[i] = new String(String.valueOf(DB_DATA[i % DB_DATA.length]));
        arr[i] = new String(String.valueOf(DB_DATA[i % DB_DATA.length])).intern();
    }

    System.out.println((System.currentTimeMillis() - t) + "ms");
    System.gc();
}

```

运行的参数是：-Xmx2g -Xms2g -Xmn1500M 上述代码是一个演示代码，其中有两条语句不一样，一条是使用 intern，一条是未使用 intern。结果如下图

2160ms



Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.lang.String[]	65	40,002,232	
char[]	1,368	135,696	
byte[]	627	115,192	
unknown-class[]	720	45,296	
short[]	681	43,464	
java.lang.String	1,345	32,280	>= 133,296
int[]	503	27,504	
java.lang.Object[]	356	17,032	
java.lang.reflect.Field	112	8,064	
java.net.URL	96	6,144	
java.lang.Class	525	5,848	
java.util.HashMap\$Entry	173	5,536	
java.lang.ref.Finalizer	123	4,920	
java.util.LinkedList\$Node	190	4,560	
java.util.HashMap\$Entry[]	44	4,544	
java.lang.Integer	259	4,144	
java.util.Hashtable\$Entry	120	3,840	
java.util.WeakHashMap\$Entry[]	42	3,360	
java.util.LinkedList	104	3,328	
java.util.concurrent.ConcurrentHashMap\$HashEntry	96	3,072	
java.lang.reflect.Method	33	2,640	
java.util.jar.JarFile	39	2,496	
java.util.WeakHashMap	42	2,352	
java.util.concurrent.ConcurrentHashMap\$Segment	57	2,280	
Total: 24 of 516 entries; 492 more	9,073	40,533,768	

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	10,001,358	400,135,296	
java.lang.String	10,001,335	240,032,040	>= 640,132,656
java.lang.String[]	65	40,002,232	
byte[]	627	115,192	
unknown-class[]	720	45,296	
short[]	681	43,464	
int[]	503	27,504	
java.lang.Object[]	356	17,032	
java.lang.reflect.Field	112	8,064	
java.net.URL	96	6,144	
java.lang.Class	525	5,848	
java.util.HashMap\$Entry	173	5,536	
java.lang.ref.Finalizer	123	4,920	
java.util.LinkedList\$Node	190	4,560	
java.util.HashMap\$Entry[]	44	4,544	
java.lang.Integer	259	4,144	
java.util.Hashtable\$Entry	120	3,840	
java.util.WeakHashMap\$Entry[]	42	3,360	
java.util.LinkedList	104	3,328	
java.util.concurrent.ConcurrentHashMap\$HashEntry	96	3,072	
java.lang.reflect.Method	33	2,640	
java.util.jar.JarFile	39	2,496	
java.util.WeakHashMap	42	2,352	
java.util.concurrent.ConcurrentHashMap\$Segment	57	2,280	
Total: 24 of 516 entries; 492 more	20,009,053	680,533,128	

通过上述结果，我们发现不使用 intern 的代码生成了1000w 个字符串，占用了大约640m 空间。使用了 intern 的代码生成了1345个字符串，占用总空间 133k 左右。其实通过观察程序中只是用到了10个字符串，所以准确计算后应该是正好相差100w 倍。虽然例子有些极端，但确实能准确反应出 intern 使用后产生的巨大空间节省。

细心的同学会发现使用了 intern 方法后时间上有了一些增长。这是因为程序中每次都是用了 new String 后，然后又进行 intern 操作的耗时时间，这一点如果在内存空间充足的情况下确实是无法避免的，但我们平时使用时，内存空间肯定不是无限大的，不使用 intern 占用空间导致 jvm 垃圾回收的时间是要远远大于这点时间的。毕竟这里使用了1000w次intern 才多出来1秒钟多的时间。

2, intern 不当使用

看过了 intern 的使用和 intern 的原理等，我们来看一个不当使用 intern 操作导致的问题。

在使用 fastjson 进行接口读取的时候，我们发现在读取了近70w条数据后，我们的日志打印变的非常缓慢，每打印一次日志用时30ms左右，如果在一个请求中打印2到3条日志以上会发现请求有一倍以上的耗时。在重新启动 jvm 后问题消失。继续读取接口后，问题又重现。接下来我们看一

1, 根据 log4j 打印日志查找问题原因




在使用log4j#info打印日志的时候时间非常长。所以使用 housemd 软件跟踪 info 方法的耗时堆栈。

```
trace SLF4JLogger.
trace AbstractLoggerWrapper:
trace AsyncLogger
```

```
org/apache/logging/log4j/core/async/AsyncLogger.actualAsyncLog(RingBufferLogEvent) sun.m
isc.Launcher$AppClassLoader@109aca82 1 1ms org.apache.logging.log4j.core.async
.AsyncLogger@19de86bb
org/apache/logging/log4j/core/async/AsyncLogger.location(String) sun.m
isc.Launcher$AppClassLoader@109aca82 1 30ms org.apache.logging.log4j.core.async
.AsyncLogger@19de86bb
org/apache/logging/log4j/core/async/AsyncLogger.log(Marker, String, Level, Message, Throwable)
```

代码出在 AsyncLogger.location 这个方法上。里边主要是调用了 return Log4jLogEvent.calcLocation(fqcnOfLogger); 和 Log4jLogEvent.calcLocation()

Log4jLogEvent.calcLocation()的代码如下:




```
public static StackTraceElement calcLocation(final String fqcnOfLogger) {
    if (fqcnOfLogger == null) {
        return null;
    }
    final StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();
    boolean next = false;
    for (final StackTraceElement element : stackTrace) {
        final String className = element.getClassName();
        if (next) {
            if (fqcnOfLogger.equals(className)) {
                continue;
            }
            return element;
        }
        if (fqcnOfLogger.equals(className)) {
            next = true;
        } else if (NOT_AVAIL.equals(className)) {
            break;
        }
    }
    return null;
}
```

经过跟踪发现是 Thread.currentThread().getStackTrace(); 的问题。

2, 跟踪Thread.currentThread().getStackTrace()的 native 代码, 验证String#intern

Thread.currentThread().getStackTrace();native的方法:



```

public StackTraceElement[] getStackTrace() {
    if (this != Thread.currentThread()) {
        // check for getStackTrace permission
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkPermission(
                SecurityConstants.GET_STACK_TRACE_PERMISSION);
        }
        // optimization so we do not call into the vm for threads that
        // have not yet started or have terminated
        if (!isAlive()) {
            return EMPTY_STACK_TRACE;
        }
        StackTraceElement[][] stackTraceArray = dumpThreads(new Thread[] {this});
        StackTraceElement[] stackTrace = stackTraceArray[0];
        // a thread that was alive during the previous isAlive call may have
        // since terminated, therefore not having a stacktrace.
        if (stackTrace == null) {
            stackTrace = EMPTY_STACK_TRACE;
        }
        return stackTrace;
    } else {
        // Don't need JVM help for current thread
        return (new Exception()).getStackTrace();
    }
}

private native static StackTraceElement[][] dumpThreads(Thread[] threads);

```

下载 openJdk7的源码查询 jdk 的 native 实现代码，列表如下【这里因为篇幅问题，不详细罗列涉及到的代码，有兴趣的可以根据文件名称和行号查找相关代码】：

\openjdk7\jdk\src\share\native\java\lang\Thread.c

\openjdk7\hotspot\src\share\vm\prims\jvm.h line:294:

\openjdk7\hotspot\src\share\vm\prims\jvm.cpp line:4382-4414:

\openjdk7\hotspot\src\share\vm\services\threadService.cpp line:235-267:

\openjdk7\hotspot\src\share\vm\services\threadService.cpp line:566-577:

\openjdk7\hotspot\src\share\vm\classfile\javaClasses.cpp line:1635-[1651,1654,1658]:

完成跟踪了底层的 jvm 源码后发现，是下边的三条代码引发了整个程序的变慢问题。

```

oop classname = StringTable::intern((char*) str, CHECK_0);
oop methodname = StringTable::intern(method->name(), CHECK_0);
oop filename = StringTable::intern(source, CHECK_0);

```

这三段代码是获取类名、方法名、和文件名。因为类名、方法名、文件名都是存储在字符串常量池中的，所以每次获取它们都是通过String#intern方法。但没有考虑到的是默认的StringPool的长度是1009且不可变的。因此一旦常量池中的字符串达到的一定的规模后，性能会急剧下降。

3.fastjson 不当使用 String#intern

导致这个 intern 变慢的原因是因为 fastjson 对String#intern方法的使用不当造成的。跟踪 fastjson 中的实现代码发现，

com.alibaba.fastjson.parser.JSONScanner#scanFieldSymbol()

```
if (ch == '\\') {
    bp = index;
    this.ch = ch = buf[bp];
    strVal = symbolTable.addSymbol(buf, start, index - start - 1, hash);
    break;
}
```

com.alibaba.fastjson.parser.SymbolTable#addSymbol():

```
/**
 * Constructs a new entry from the specified symbol information and next entry reference.
 */
public Entry(char[] ch, int offset, int length, int hash, Entry next){
    characters = new char[length];
    System.arraycopy(ch, offset, characters, 0, length);
    symbol = new String(characters).intern();
    this.next = next;
    this.hashCode = hash;
    this.bytes = null;
}
```

fastjson 中对所有的 json 的 key 使用了 intern 方法，缓存到了字符串常量池中，这样每次读取的时候就会非常快，大大减少时间和空间。而且 json 的 key 通常都是不变的。这个地方没有考虑到大量的 json key 如果是变化的，那就会给字符串常量池带来很大的负担。

这个问题 fastjson 在1.1.24版本中已经将这个漏洞修复了。程序加入了一个最大的缓存大小，超过这个大小后就不会再往字符串常量池中放了。

[1.1.24版本的com.alibaba.fastjson.parser.SymbolTable#addSymbol() Line:113]代码

```
public static final int MAX_SIZE = 1024;

if (size >= MAX_SIZE) {
    return new String(buffer, offset, len);
}
```

这个问题是70w 数据量时候的引发的，如果是几百万的数据量的话可能就不只是30ms 的问题了。因此在使用系统级提供的String#intern方式一定要慎重！

五，总结

本文大体的描述了 String#intern和字符串常量池的日常使用，jdk 版本的变化和String#intern方法的区别，以及不恰当使用导致的危险等内容，让大家对系统级别的 String#intern有一个比较深入的认识。让我们在使用和接触它的时候能避免出现一些 bug，增强系统的健壮性。

引用：

以下是几个比较关键的几篇博文。感谢！

Save Memory by Using String Intern in Java
(<https://blog.codecentric.de/en/2012/03/save-memory-by-using-string-intern-in-java/>).

Java String array: is there a size of method?
(<http://stackoverflow.com/questions/921384/java-string-array-is-there-a-size-of-method>).

Understanding String Table Size in HotSpot
(<http://xmlandmore.blogspot.com/2013/05/understanding-string-table-size-in.html>).

How is Java's String#intern() method implemented?
(<http://stackoverflow.com/questions/3323608/how-is-javas-stringintern-method-implemented>).

JDK7里的String.intern的变化

(全文完)

扫描二维码，关注作者微信公众号