我们学习了Java开发基础,知道Java有8中基本类型数据,其中char用来描述UTF-16编码中的一个代码单元,那么如果想要表示多个字符呢?即一个字符串我们该怎么表示呢?

在Java语言中将字符串作为对象来处理,可以通过java.lang包中String类来处理字符串.下面我们来认真讲解一下String,注意:String是Java中很关键的类,是基础中的基础.

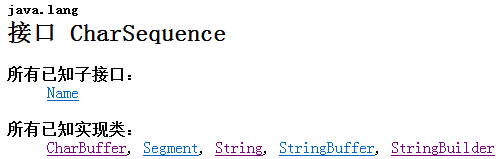
在认识String之前,我们先来看一下String实现的接口,从接口开始学习,这也是面向接口编程的意义所在.

# 7.0 CharSequence

查看API我们知道String、StringBuffer、StringBuilder都实现了CharSequence接口，查看API对接口的解释如下：

public interface CharSequence

CharSequence 是 char 值的一个可读序列。此接口对许多不同种类的 char 序列提供统一的**只读访问**。char 值表示 Basic Multilingual Plane (BMP 基本的多语言级别) 或代理项中的一个字符。有关详细信息，请参阅 [Unicode 字符表示形式](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/Character.html" \l "unicode)。 此接口不修改 [equals](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "equals(java.lang.Object)) 和 [hashCode](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \l "hashCode()) 方法的常规协定。因此，通常未定义比较实现 CharSequence 的两个对象的结果。每个对象都可以通过一个不同的类实现，而且不能保证每个类能够测试其实例与其他类的实例的相等性。因此，**使用任意 CharSequence 实例作为集合中的元素或映射中的键是不合适的**。下面是所有实现CharSequence接口的类，我们在后面一一认识。



## 7.0.1 接口的方法摘要

|  |  |
| --- | --- |
| 方法摘要 | |
| char | **[charAt](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \l "charAt(int))**(int index) 返回指定索引的 char 值。 |
| Int | **[length](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \l "length())**() 返回此字符序列的长度。长度是序列中的16位char的数量 |
| [CharSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) | **[subSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \l "subSequence(int, int))**(int start, int end) 返回一个新的 CharSequence，它是此序列的子序列。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[toString](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \l "toString())**() 返回一个包含此序列中字符的字符串，该字符串与此序列的顺序相同。 |

# 7.1 字符串

从概念上讲,Java字符串就是Unicode字符序列。例如，串“Java\u2122”由5个Unicode字符J,a,v,a和TM组成。Java并没有内置的字符串类型，而是在标准Java类库中提供了一个预定义类，很自然的叫做String。每个

# 7.1 字符串的声明与创建

学习String的第一步就是创建(声明)字符串,我们在这里之所以分为创建和声明是因为String是一个很特殊的类,它的对象产生在五种创建对象之外,还有另外一种方式,下面我们就来详细了解一下.

## 7.1.1 声明字符串

在Java语言中字符串必须包含在一对””双引号中.例如:

“23.3”,”adc”,”ad%-”,”1+3”.”你好,安静”

以上都是字符串常量,字符串常量是系统能够显示的任何文字信息,甚至是单个字符。在这里我们必须再强调一下凡是被双引号“”包含的都是字符串，不能作为其它数据类型使用，若要使用需要转型（可能出现类型转换异常），如上“1+2”如果输出的话是不会输出3的，输出结果是1+2.

可以通过一下语法格式来声明字符串:

String str = [null];

* String:描述符,指定该变量为字符串变量;
* Str:任意有效的标识符,表示字符串变量的名称;
* =:赋值运算符,在数学上我们可以理解为等号,在内存层面我们可以理解为将str指向一个内存地址;
* Null:在这里表示赋值的内容,如果未赋值,默认是为初始化,为null.否则表明声明的字符串值为null.注意:成员字段可以不初始化,虚拟机给予初始化,局部变量必须初始化.

|  |
| --- |
| **package** cn.stringPractise.create;  //字符串创建的两种方法--声明字符串  **public** **static** **void** test1(){  String str1 = "abc";  String str2 = **null**;  // String str3;如果没有初始化就是用,会提示未曾初始化错误  //System.out.println(str3);//The local variable str3 may not have been initialized  System.***out***.println("str1:"+str1+"--hashcode:"+str1.hashCode());  System.***out***.println(str2);  //获取hashcode,此时null根本没有分配内存,自然是一个空指针,下面会报出空指针异常  // System.out.println("str2:"+str2+"--hashcode:"+str2.hashCode()); java.lang.NullPointerException  } |



## 7.1.2 创建字符串

字符串除了可以直接赋值外,还可以由另一种方式来创建,下面我们来介绍一下.

在Java中将字符串作为对象来管理,因此可以像创建其它类对象一样来创建字符串对象。创建对象要用到构造方法。String类的常用构造方法如下：

String（char a[]）方法

用一个字符数组a创建String对象

String(char a[],int offset,int length)

提取字符数组的一部分创建一个字符串对象.参数offset表示开始截取字符串的位置,length表示截取字符串的长度.

String(char[] value)

该构造方法可分配一个新的String对象,使其表示字符数组参数中所有的元素连接的结果。

除了上面几种使用String类的构造方法来创建字符串变量外，还可通过字符串常量的引用赋值给一个字符串变量。

【例7.1.2.1】验证创建字符串的几种方式。

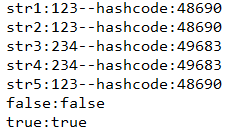
|  |
| --- |
| //字符串创建的第二种方法--创建字符串  **public** **static** **void** test2(){  **char**[] ch = {'花','褪','残','红','青','杏','小'};  //使用字符数组创建字符串对象  String str1 = **new** String(ch);  System.***out***.println(str1);  //提取字符数组中的一部分,创建爱你字符串对象  String str2 = **new** String(ch, 1, 3);  System.***out***.println(str2);  String str3 = str1;  System.***out***.println(str3);  System.***out***.println(str1 == str3);  } |



## 7.1.3 两种字符串创建的对比

上面我们分别针对两种字符串的声明进行了分析,下面我们在来看一段代码,然后进行内存分析:

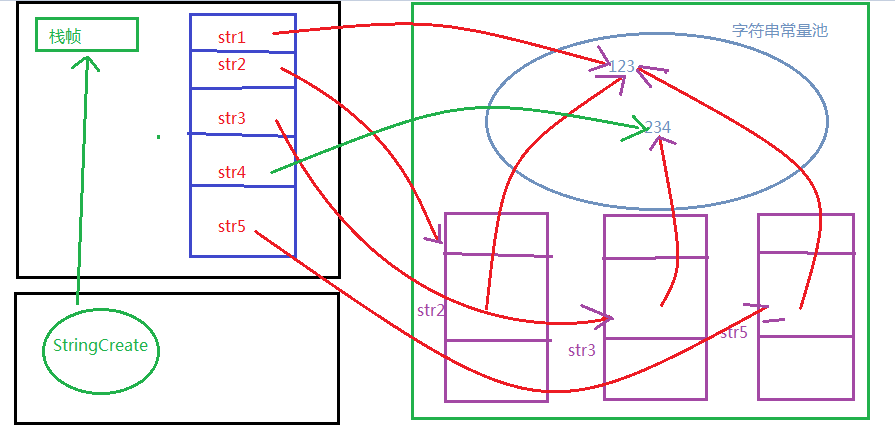
|  |
| --- |
| //两种字符串创建的对比  **public** **static** **void** test3(){  String str1 = "123";  String str2 = **new** String("123");  String str3 = **new** String("234");  String str4 = "234";  System.***out***.println("str1:"+str1+"--hashcode:"+str1.hashCode());  System.***out***.println("str2:"+str2+"--hashcode:"+str2.hashCode());  System.***out***.println("str3:"+str3+"--hashcode:"+str3.hashCode());  System.***out***.println("str4:"+str4+"--hashcode:"+str4.hashCode());  String str5 = **new** String("123");  System.***out***.println("str5:"+str5+"--hashcode:"+str5.hashCode());  System.***out***.println((str2 == str5)+":"+(str1 == str2));  System.***out***.println((str2.equals(str5))+":"+(str1.equals(str2)));  } |



通过上面的比对,我们明白了一个问题,那就是hashcode绝不是真实的内存地址,我们知道String中”==”比较的是地址的内存地址,而equals()的源码如下:

|  |
| --- |
| **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])  **return** **false**;  i++;  }  **return** **true**;  }  }  **return** **false**;  } |

String重写了Object的equals(),通过比对hashcode和字符串内容来确定equals的返回值,当他们有一点满足true就是true.下面我们来分析一下上面的test3()在内存中的情形:



美术功底太差,将就看吧,从上面我们可以看出,通过声明字符串的形式进行字符串的创建的时候,会先在字符串常量池中查看是否有这个字符串,如果有那么就将这个引用指向这个字符串的地址,如果没有那么就在字符串常量池中创建该字符串,并将指针指向该字符串的内存地址.

而通过new创建字符串则不同,它会在堆中创建一个字符串对象,然后才是在字符串常量池中查找是否有这个字符串,如果有那么会在堆中的对象中放入一个该字符串的地址,如果没有在字符串常量池中创建字符串并将地址放在堆中对象里边,而我们的指针是指向堆中的字符串对象的.有了这个分析,我们再来看一道面试题吧:

## 7.1.4 String的构造方法简略

|  |  |
| --- | --- |
| **构造方法摘要** | |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String())**()            初始化一个新创建的 String 对象，使其表示一个空字符序列。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(byte[]))**(byte[] bytes)            通过使用平台的默认字符集解码指定的 byte 数组，构造一个新的 String。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(byte[], java.nio.charset.Charset))**(byte[] bytes, [Charset](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset 中的类) charset)            通过使用指定的 [charset](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset 中的类) 解码指定的 byte 数组，构造一个新的 String。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(byte[], int))**(byte[] ascii, int hibyte)            **已过时。** *该方法无法将字节正确地转换为字符。从 JDK 1.1 开始，完成该转换的首选方法是使用带有 [Charset](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset 中的类)、字符集名称，或使用平台默认字符集的 String 构造方法。* |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(byte[], int, int))**(byte[] bytes, int offset, int length)            通过使用平台的默认字符集解码指定的 byte 子数组，构造一个新的 String。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(byte[], int, int, java.nio.charset.Charset))**(byte[] bytes, int offset, int length, [Charset](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset 中的类) charset)            通过使用指定的 [charset](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset 中的类) 解码指定的 byte 子数组，构造一个新的 String。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(byte[], int, int, int))**(byte[] ascii, int hibyte, int offset, int count)            **已过时。** *该方法无法将字节正确地转换为字符。从 JDK 1.1 开始，完成该转换的首选方法是使用带有 [Charset](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset 中的类)、字符集名称，或使用平台默认字符集的 String 构造方法。* |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(byte[], int, int, java.lang.String))**(byte[] bytes, int offset, int length, [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) charsetName)            通过使用指定的字符集解码指定的 byte 子数组，构造一个新的 String。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(byte[], java.lang.String))**(byte[] bytes, [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) charsetName)            通过使用指定的 [charset](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset 中的类) 解码指定的 byte 数组，构造一个新的 String。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(char[]))**(char[] value)            分配一个新的 String，使其表示字符数组参数中当前包含的字符序列。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(char[], int, int))**(char[] value, int offset, int count)            分配一个新的 String，它包含取自字符数组参数一个子数组的字符。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(int[], int, int))**(int[] codePoints, int offset, int count)            分配一个新的 String，它包含 Unicode 代码点数组参数一个子数组的字符。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) original)            初始化一个新创建的 String 对象，使其表示一个与参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的副本。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(java.lang.StringBuffer))**([StringBuffer](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) buffer)            分配一个新的字符串，它包含字符串缓冲区参数中当前包含的字符序列。 |  |
| **[String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "String(java.lang.StringBuilder))**([StringBuilder](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/StringBuilder.html" \o "java.lang 中的类) builder)            分配一个新的字符串，它包含字符串生成器参数中当前包含的字符序列。 |  |

注意:String(byte[] bytes,String charsetName),可以用来将一个字节数组按照指定编码集,创建一个String.

## 7.1.5 字符串的底层是char数组

我们经常会听人说字符串的底层是字符数组，那么事实是这样吗？下面我们通过字符串的构造函数来了解一下；

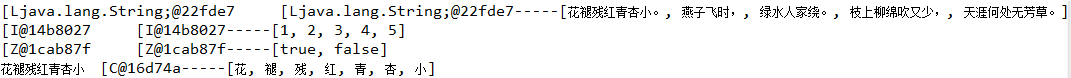
|  |
| --- |
| **private** **final** **char** value[];  **public** String(**char** value[]) {  **this**.value = Arrays.*copyOf*(value, value.length);  } |

在这个构造函数中使用到了数组的copyOf（）方法，该方法的具体作用如下：



那么这个构造函数其实就是以我们传递进来的字符数组为范本然后创建一个新的字符数组，内容一样，然后赋值给String中已经声明但是未经初始化的字符数组，所以我们平常看到的字符串，其底层实际上是一个字符数组，那么这是我们可能会有这么一个疑问，那就是为什么输出的时候输出字符串，在控制台上还是以字符串出现呢，数组的输出不是通常会输出一个16进制的hashCode吗，这和我们所说的不是矛盾吗？下面我们以一段代码来分析：

|  |
| --- |
| /\*\*  \* String的底层是char数组  \*/  @Test  **public** **void** fun3(){  String[] str = {"花褪残红青杏小。","燕子飞时，","绿水人家绕。","枝上柳绵吹又少，","天涯何处无芳草。"};  **int**[] i = {1,2,3,4,5};  **boolean**[] boo = {**true**,**false**};  **char**[] ch = {'花','褪','残','红','青','杏','小'};  System.*out*.print(str);  System.*out*.println(" "+str+"-----"+Arrays.*toString*(str));  System.*out*.print(i);  System.*out*.println(" "+i+"-----"+Arrays.*toString*(i));  System.*out*.print(boo);  System.*out*.println(" "+boo+"-----"+Arrays.*toString*(boo));  System.*out*.print(ch);  System.*out*.println(" "+ch+"-----"+Arrays.*toString*(ch));  } |



查看结果我们发现，当我们数组对象时，大部分数组都会输出一个看起来像是地址的字符串，只有char数组，输出了一个字符串，字符串内容是字符数组的元素。这是为什么呢？然我们一步步通过代码来解析：

首选看输出语句打开看它的底层代码如下：printStraem.class

|  |
| --- |
| **public** **void** println(Object x) {  String s = String.*valueOf*(x);  **synchronized** (**this**) {  print(s);  newLine();  }  } |

观察上面的代码,我们知道输出会将一个object通过String.valueOf()方法转变为String,观察该方法

|  |
| --- |
| **public** **static** String valueOf(Object obj) {  **return** (obj == **null**) ? "null" : obj.toString();  } |
| **public** **static** String valueOf(**char** data[]) {  **return** **new** String(data);  } |
| **public** **static** String valueOf(**char** data[], **int** offset, **int** count) {  **return** **new** String(data, offset, count);  } |

在String类中涉及到的valueOf方法有这三个,观察发现,其中两个都是关于char数组的,而一个是涉及的其它对象的,那么数组的输出在这里划出了两条路线,观察字符发现,它会把字符数组作为参数构建一个字符串对象.观察那个参数是Object的,如果不是null,那么会返回一个字符串,内容是Object的toString方法

|  |
| --- |
| **public** String toString() {  **return** getClass().getName() + "@" + Integer.*toHexString*(hashCode());  } |

那么此时到了这里,字符数组变成了一个字符串对象,底层仍然是字符数组,而其他类型则变成了一个字符串,内容是16进制的HashCode,接下来我们看对字符串的打印输出

|  |
| --- |
| **public** **void** print(String s) {  **if** (s == **null**) {  s = "null";  }  write(s);  } |
| **private** **void** write(String s) {  **try** {  **synchronized** (**this**) {  ensureOpen();  textOut.write(s);  textOut.flushBuffer();  charOut.flushBuffer();  **if** (autoFlush && (s.indexOf('\n') >= 0))  out.flush();  }  }  **catch** (InterruptedIOException x) {  Thread.*currentThread*().interrupt();  }  **catch** (IOException x) {  trouble = **true**;  }  } |

到了这里我们大概就明白了,使用字符流输出到控制台字符数组变成了字符串,而其他对象输出的确实一个看起来像是地址的HashCode(HashCode和地址不一样,它并非是真实的物理地址).

# 7.2 字符串的深入了解

不要以为字符串很简单我们就可以掉以轻心,String给我们的惊喜还有很多,下面我们就来了解一下;

## 7.2.1 不可变的String

不可变的String是什么意思呢?查看String的帮助文档我们发现

public final class String

extends [Object](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类)

implements [Serializable](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/io/Serializable.html" \o "java.io 中的接口), [Comparable](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/Comparable.html" \o "java.lang 中的接口)<[String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类)>, [CharSequence](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口)

String类是被final修饰过的,查看String的底层char数组

**private** **final** **char** value[];

String对象是不可变的。查看JDK文档你就会发现，String类中每一个看起来会修改String值的方法，实际上都是创建一个全新的String对象，以包含修改后的字符串内容。而最初的String对象则丝毫未动（如果没有指针指向，下一步面临的就是垃圾回收器的回收了）。

|  |
| --- |
| **package** cn.stringPractise.Commonoperation;  **public** **class** Immutable {  **public** **static** String upcase(String s){  **return** s.toUpperCase();//字符串大写  }  **public** **static** **void** main(String[] args) {  String q = "howdy";  System.*out*.println(q);  String qq = *upcase*(q);  System.*out*.println(qq);  System.*out*.println(q);  System.*out*.println(q == qq);  }  } |



当把q传给upcase()方法时,实际传递的是引用的一个拷贝.其实,每当把String对象作为方法的参数时,都会复制一份引用,而该引用所指的对象其实一直待在单一的物理位置上,从未动过.

回到upcase()的定义,传入其中的引用有了名字s,只有upcase()运行的时候,局部引用s才存在.一旦upcase()运行结束,s就消失了.当然了,upcase()的返回值,其实只是最终结果的引用.这足以说明,upcase()返回的引用已经指向了一个新的对象,而原本的q则还在原地.

String的这种行为方式其实正是我们想要的.例如:

String s = “asdf”;

String x = Immutable.upcase(s);

难道你真的希望upcase()改变其参数吗?**对于一个方法而言,参数是为该方法提供信息的,而不是想让该方法改变自己的**.在阅读这段代码时,读者自然就会有这样的感觉.这一点很重要,正是有了这种保障,才使得代码易于编写与阅读.

## 7.2.2 重载”+”与StringBuilder

String对象是不可变的,你可以给一个String对象加任意多的别名.因为String对象具有只读特性,所以指向它的任何引用都不可能改变它的值,因此,也就不会对其他的引用有什么影响(例如两个别名指向同一个引用,一个别名有了改变这个引用的操作,那么不可变性就保证了另一个别名引用的安全).

不可变性会带来一定的效率问题.为String对象重载的”+”操作符就是一个例子.重载的意思是,一个操作符在应用于特定的类时,被赋予了特殊的意义(用于String的”+”与”+=”是Java中仅有的两个重载过的操作符,而Java并不允许程序员重载任何操作符).+在数学中用来两个数的相加,在字符串中用来连接String:

|  |
| --- |
| **package** cn.string.two;  **public** **class** Concatenation {  **public** **static** **void** main(String[] args) {  String mango = "mango";  String s = "abc" + mango + "def" + 47;  System.*out*.println(s);  }  } |



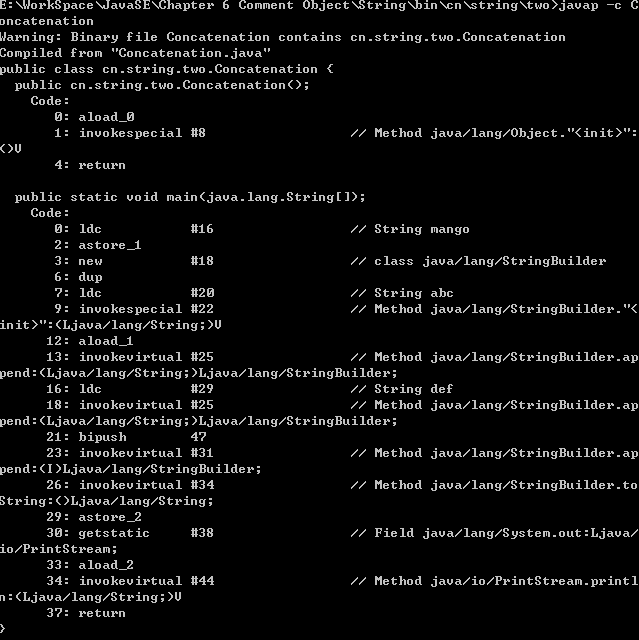
可以想象一下,这段代码可能是这样工作的:String可能有一个append()方法,它会生成一个新的String对象,以包含”abc”与mango连接后的字符串.然后,该对象再与”def”相连,生成另一个新的String对象,依次类推.

这种工作方式当然也行得通,但是为了生成最终的String,此方式会产生一大堆需要垃圾回收的中间对象.我猜想,Java设计师一开始就是这么做的(这也是软件设计中的一个教训:除非你用代码将系统实现,并让它动起来,否则你无法真正了解它会有什么问题),然后他们发现其性能相当糟糕.

想看看以上代码到底是如何工作的吗,可以用JDK自带的工具javap来反编译以上代码.命令如下:



这里的-c标志表示将生成JVM字节码.我剔除掉了不感兴趣的部分,然后作了一点点修改,于是有了以下的字节码:



如果有汇编语言的经验,以上代码一定看着眼熟,其中的dup与invokevirtural语句相当于Java虚拟机上的汇编语句.即使你完全不了解汇编语言也无需担心,需要注意的重点是:编译器自动引入了java.lang.StringBuilder类.虽然我们在源代码中并没有使用StringBuilder类,但是编译器却自作主张地使用了它,因为它更高效.

在这个例子中,编译器创建了一个StringBuilder对象,用以构造最终的String,并为每个字符串调用一次StringBuilder的append()方法,总计四次.最后调用toString()生成结果,并存在s(使用的命令为astore\_2)

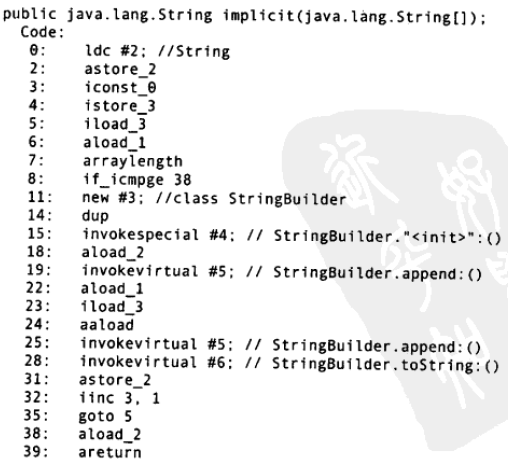
现在,也许你会觉得可以随意使用String对象,反正编译器会为你自动地优化性能.可是在这之前,让我们更深入地看看编译器能为我们优化到什么程度.下面的程序采用两种方式生成一个String:方法一使用了多个String对象,方法二在代码中使用了StringBuilder.

|  |
| --- |
| **package** cn.stringPractise.Commonoperation;  **public** **class** WhitherStringBuilder {  **public** **static** **void** main(String[] args) {  String[] str = {"长安古道马迟迟","高柳乱蝉嘶","夕阳岛外","秋风原上","目断四天垂",  "归云一去无踪迹","何处是前期","狎兴生疏","酒徒萧索","不似去年时。"};  System.*out*.println(*implicit*(str));  System.*out*.println(*explicit*(str));  }  **public** **static** String implicit(String[] fields){  String result = "";  **for** (**int** i = 0; i < fields.length; i++) {  result += fields[i];  }  **return** result;  }  **public** **static** String explicit(String[] fields){  StringBuilder sb = **new** StringBuilder();  **for** (**int** i = 0; i < fields.length; i++) {  sb.append(fields[i]);  }  **return** sb.toString();  }  } |
| **public** **static** **void** main(String[] args) {  String[] str = {"长安古道马迟迟","高柳乱蝉嘶","夕阳岛外","秋风原上","目断四天垂",  "归云一去无踪迹","何处是前期","狎兴生疏","酒徒萧索","不似去年时。"};  String[] str1 = **new** String[20000];  **for** (**int** i = 0; i < 20000; i++) {  str1[i] = Integer.*toString*(i);  }  **long** start = System.*currentTimeMillis*();  // System.out.println(implicit(str1));  *implicit*(str1);  **long** end = System.*currentTimeMillis*();  System.*out*.println(end-start);  start = System.*currentTimeMillis*();  *explicit*(str1);  // System.out.println(explicit(str1));  end = System.*currentTimeMillis*();  System.*out*.println(end-start);  } |



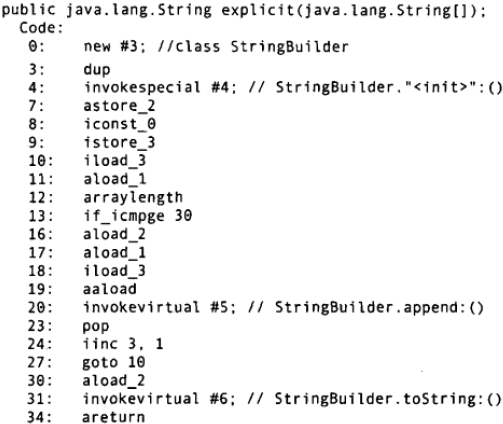


现在运行javap -c WitherStringBuilder,可以看到两个方法对应的(简化过的)字节码.首先是implicit()方法:



注意从第8行到第35行构成了一个循环体.第8行:对堆栈中的操作数进行”大于或等于的整数比较运算”,循环结束时跳到第38行.第35行:返回循环体的起始点(第5行).要注意的重点是:StringBuilder是在循环体内构成的,这意味着每经过一次循环,就会创建一个新的StringBuilder对象.

下面是explicit()方法对应的字节码:



可以看到,不仅循环部分的代码更简短、更简单，而且它只生成了一个StringBuilder对象。显式的创建StringBuilder还允许你预先为其指定大小.如果你已经知道最终的字符串大概有多长,那预先指定StringBuilder的大小可以避免多长重新分配缓冲.

因此,当你为一个类编写toString()方法时,如果字符串操作比较简单,那就可以信赖编译器,它会为你合理地构造最终的字符串结果.但是,如果你要在toString()方法中使用循环,那么最好自己创建一个StringBuilder对象,用它来构造最终的结果.参考一下示例:

|  |
| --- |
| **package** cn.stringPractise.Commonoperation;  **import** java.util.Random;  **public** **class** UsingStringBuilder {  **public** **static** Random *rand* = **new** Random(47);  @Override  **public** String toString() {  StringBuilder builder = **new** StringBuilder("[");  **for** (**int** i = 0; i < 25; i++) {  builder.append(*rand*.nextInt(100));  builder.append(",");  }  builder.delete(builder.length()/2, builder.length());  builder.append("]");  **return** builder.toString();  }    **public** **static** **void** main(String[] args) {  UsingStringBuilder usingStringBuilder = **new** UsingStringBuilder();  System.*out*.println(usingStringBuilder);  }  } |



|  |  |
| --- | --- |
| **public** **void** println(Object x) {  String s = String.*valueOf*(x);  **synchronized** (**this**) {  print(s);  newLine();  }  } | **public** **static** String valueOf(Object obj) {  **return** (obj == **null**) ? "null" : obj.toString();  } |

最终的结果是用append()语句一点点拼接起来的.如果你想走捷径,例如append(a+”:”+c),那编译器就会掉入陷阱,从而为你另外创建一个StringBuilder对象处理括号内的字符串操作.

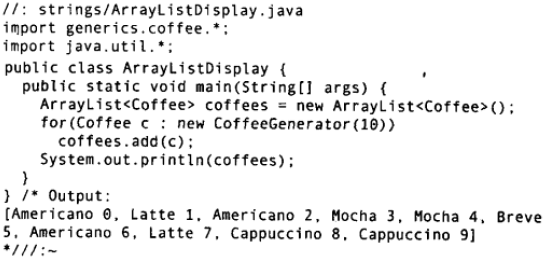
如果拿不准该用哪种方式,随时可以用javap来分析你的程序.

StringBuilder提供了丰富而全面的方法,包括insert()、repleace()、substring()甚至reverse()，但是最常用的还是append()和toString().还有delete()方法,上面的例子中我们用它删除最后一个逗号与空格,以便添加右括号.

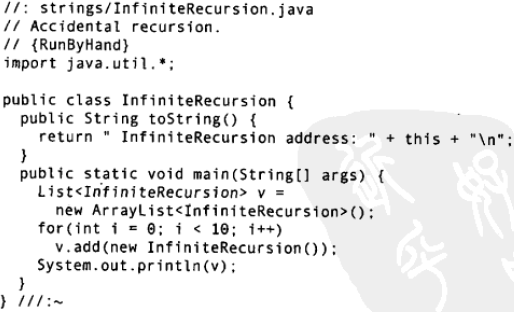
StringBuilder是Java SE5引入的,在这之前Java用的是StringBuffer.后者是线程安全的,因此开销也会大些,所以在Java SE5/6中,字符串操作应该还会更快一点.

## 7.2.3 无意识的递归

Java中的每个类从根本上都是继承自Object,标准容器类自然也不例外.因此容器类都有toString()方法,并且覆写了该方法,使得它生成的String结果能够表达容器自身,以及容器所包含的对象.例如ArrayList.toString(),它会遍历ArrayList中包含的所有对象,调用每个元素上的toString()方法.



如果你希望toString()打印出对象的内存地址,也许你会考虑使用this关键字:



当你创建了InfiniteRecursion对象,并将其打印出来的时候,你会得到一串非常长的异常.如果你将该InfiniteRecursion对象存入一个ArrayList中,然后打印该ArrayList,你也会得到同样的异常.其实,当如下代码运行时:



这里发生了自动类型转换.由InfiniteRecursion类型转换成String类型.因为编译器看到一个String对象后面跟着一个”+”,而再后面的对象不是String,于是编译器试着将this转换成一个String.它怎么转换呢,正是通过this上的toString()方法,于是就发生了递归调用.

如果你真的想要打印出对象的内存地址,应该调用Object.toString()方法,这才是负责此任务的方法,所以你不该使用this,而是应该调用super.toString()方法.⑪⑪

　　我们观察String的源码，发现String的底层数据结构是字符数组，在成员字段中声明了一个字符数组（声明但为初始化），我们创建String对象时，会将字符数组变量指向一个动态创建的数组地址，我们的String变量的内容实际上是指向该字符数组的内存地址的。而该字符数组是被声明为final，final意味着常量，意味着一经初始化是无法改变的，但是对象的指向是可以发生改变的。我们通过一副内存图来说明：

# 7.3 String上的常见操作

在字符串上进行操作,使我们对字符串最直接的一种运用,下面我们就来一一介绍一下字符串中的常见方法.除了实现接口中的方法外，还有一些特有的方法，我们一一认识一下。

## 7.3.1 连接字符串

### 7.3.1.1 使用重载+连接字符串

连接字符串是我们很常见的一种字符串操作方式,字符串和字符串的连接好理解,那么字符串和其它类型的连接呢?

在Java中的数学运算符中+是用于数字的相加的,在String中+被重载可以用于字符串的相加,如果是字符串和其它数据类型的相加,那么结果数据类型转变为String,底层是StringBuilder.append().toString;**String类是没有append()方法的**.

使用“+”运算符实现连接多个字符串的功能，“+”运算符可以连接多个运算符并产生一个String对象。一句相连的字符串不能分开在两行写，是错误的，可以通过运算符“+”实现

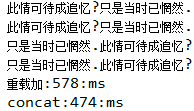
|  |
| --- |
| **package** cn.stringPractise.append;  **import** org.junit.Test;  **public** **class** StringAppend {    /\*\*  \* 连接字符串  \* 字符串之间的连接  \* 字符串和其它类型的连接  \*/  @Test  **public** **void** fun2(){  String str1 = "归云一去无踪迹,何处是前期?";  String str2 = "夕阳岛外,秋风原上,目断四天垂.";  **int** i = 90;  **boolean** boo = **true**;  **char** ch= '1';  **boolean**[] booArr = {**true**,**false**};  String str3 = str1+i+boo+ch+booArr;  System.***out***.println(str3);  StringBuilder sb = **new** StringBuilder();  str3 = sb.append(str1).append(i).append(boo).append(ch).append(booArr).toString();  System.***out***.println(str3);  }  } |



### 7.3.1.2 使用concat连接字符串



|  |
| --- |
| /\*\*  \* 字符串连接  \*/  @Test  **public** **void** fun15(){  String str1 = "此情可待成追忆?";  String str2 = "只是当时已惘然.";  System.***out***.println(str1.concat(str2));  System.***out***.println(str1+str2);  System.***out***.println(str2.concat(str1));  System.***out***.println(str2+str1);  //验证+和concat的效率问题  **long** t1 = System.*currentTimeMillis*();  **for**(**int** i = 0;i<10000;i++){  str1 += Integer.*toString*(i);  }  **long** t2 = System.*currentTimeMillis*();  System.***out***.println("重载加:"+(t2-t1)+":ms");  t1 = System.*currentTimeMillis*();  **for**(**int** i = 0;i<10000;i++){  str1.concat(Integer.*toString*(i));  }  System.***out***.println("concat:"+(System.*currentTimeMillis*()-t1)+":ms");  } |



重载加底层是new StringBuilder.append(),而concat()的底层代码如下:

|  |
| --- |
| **public** String concat(String str) {  **int** otherLen = str.length();  **if** (otherLen == 0) {  **return** **this**;  }  **int** len = value.length;  **char** buf[] = Arrays.*copyOf*(value, len + otherLen);  str.getChars(buf, len);  **return** **new** String(buf, **true**);  } |

这中间省去了创建StringBuilder的时间,所以看起来concat的效率和内存空间使用方面更具有优势.

## 7.3.2 获取字符串信息

字符串变量作为对象,可通过相应方法获取字符串的有效信息,如获取字符串的长度,获取指定字符的索引位置等.

### 7.3.2.1 求字符串长度

我们知道字符串底层其实是字符数组,那么既然是数组自然就能够获取长度字符串获取长度的方法和字符串获取长度的方法相同,都是length();

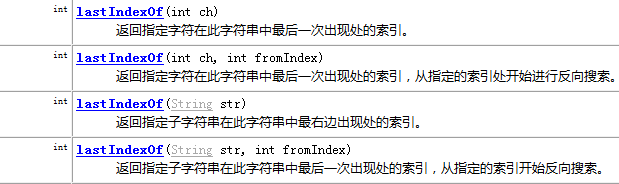
|  |
| --- |
| //获取字符串长度  @Test  **public** **void** fun1(){  String str = "林花谢了春红,太匆匆! ";//空格也算一个字符,只要在引号内占位置  System.*out*.println("字符串长度:"+str.length());  } |



### 7.3.2.2 根据给定字符获取索引位置

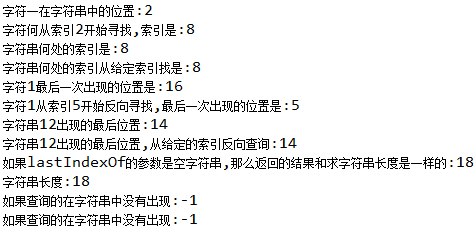
String类提供了indexOf()和lastIndexOf()方法来获取指定字符的索引位置,不同的是,indexOf()方法返回的搜索的字符首次出现位置的索引,而lastIndexOf()方法返回的是搜索的字符最后出现的位置的索引.





注意:所有的参数都是要查找的字符或字符串,注意不是索引

|  |
| --- |
| **package** cn.stringPractise.index;  /\*\*  \* 根据给定信息查询字符串  \*/  @Test  **public** **void** fun3(){  String str1 = "归云一去无踪迹,何处是前期?";  String str2 = "123242543534121214";  System.*out*.println("字符一在字符串中的位置:"+str1.indexOf('一'));  System.*out*.println("字符何从索引2开始寻找,索引是:"+str1.indexOf('何', 2));  System.*out*.println("字符串何处的索引是:"+str1.indexOf("何处"));  System.*out*.println("字符串何处的索引从给定索引找是:"+str1.indexOf("何处", 3));  System.***out***.println("----"+str1.indexOf("何处", 9));//查找不到返回-1  System.*out*.println("字符1最后一次出现的位置是:"+str2.lastIndexOf('1'));  System.*out*.println("字符1从索引5开始反向寻找,最后一次出现的位置是:"+str2.lastIndexOf('2', 5));  System.*out*.println("字符串12出现的最后位置:"+str2.lastIndexOf("12"));  System.*out*.println("字符串12出现的最后位置,从给定的索引反向查询:"+str2.lastIndexOf("12", 15));  System.*out*.println("如果lastIndexOf的参数是空字符串,那么返回的结果和求字符串长度是一样的:"+str2.lastIndexOf(""));  System.*out*.println("字符串长度:"+str2.length());  System.*out*.println("如果查询的在字符串中没有出现:"+str1.indexOf('1'));  System.*out*.println("如果查询的在字符串中没有出现:"+str2.lastIndexOf(" "));  } |
| //查找指定字符在字符串中第一次出现的位置,若字符串中没有要查找的字符返回-1  System.out.println("j第一次出现的位置是："+str.indexOf('。'));  //返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。若字符串中没有要查找的字符返回-1  System.out.println("o在脚标2后第一次出现的索引："+str.indexOf('o', 2)); |



注意: System.*out*.println(str2.lastIndexOf(100));//索引越界不会异常,会返回负值-1

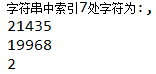
字符串中凡是查询不到那么就返回-1

### 7.3.2.3 根据给定位置获取字符

使用String类的charAt()方法可获取指定索引处的字符,返回字符的索引.【这是接口方法，所有实现CharSequence接口的类都有该方法】

|  |  |
| --- | --- |
| int | **[codePointAt](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "codePointAt(int))**(int index)            返回指定索引处的字符（Unicode 代码点）。 |
| int | **[codePointBefore](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "codePointBefore(int))**(int index)            返回指定索引之前的字符（Unicode 代码点）。 |
| int | **[codePointCount](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "codePointCount(int, int))**(int beginIndex, int endIndex)            返回此 String 的指定文本范围中的 Unicode 代码点数。 |

|  |
| --- |
| /\*\*  \* 根据索引获取字符  \*/  @Test  **public** **void** fun4(){  String str1 = "归云一去无踪迹,何处是前期?";  System.*out*.println("字符串中索引7处字符为:"+str1.charAt(7));  System.*out*.println(str1.codePointAt(3));  System.*out*.println(str1.codePointBefore(3));  System.*out*.println(str1.codePointCount(2, 4));//计头不计尾  } |



注意:charAt还可用于字符串的遍历

|  |
| --- |
| //遍历字符串  for(int i=0;i<str.length();i++){  System.out.print(str.charAt(i)+"、");  } |

## 7.3.3 字符串操作

字符串除了可以查询外,还有许多其它的”修改”功能,下面我们就来认识一下;

### 7.3.3.1 字符串去除空格

去掉字符串中的空格有两种,一是去除字符串的前导、尾部空格，二是去除字符串中的所有空格，可以用不同的方法来实现。

#### 7.3.3.1.1 去除前导、尾部空格

|  |  |
| --- | --- |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[trim](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "trim())**()            返回字符串的副本，忽略前导空白和尾部空白。 |

|  |
| --- |
| **package** cn.stringPractise.space;  /\*\*  \* 去除空格--前导 后置  \*/  @Test  **public** **void** fun5(){  String str = " 红藕香残玉簟秋 轻解罗裳独上兰舟 ! ";  System.***out***.println("去除空格前长度:"+str.length()+"--去除空格后长度:"+str.trim().length());  } |

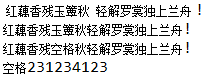


#### 7.3.3.1.2 去除字符串中所有空格

|  |  |
| --- | --- |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[replace](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "replace(char, char))**(char oldChar, char newChar)            返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[replace](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "replace(java.lang.CharSequence, java.lang.CharSequence))**([CharSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) target, [CharSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) replacement)            使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[replaceAll](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "replaceAll(java.lang.String, java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) regex, [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) replacement)            使用给定的 replacement 替换此字符串所有匹配给定的[正则表达式](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/util/regex/Pattern.html" \l "sum)的子字符串。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[replaceFirst](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "replaceFirst(java.lang.String, java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) regex, [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) replacement) |

|  |
| --- |
| /\*\*  \* 去除空格--字符串中所有的空格使用split截取字符串获取字符串数组,然后trim每个字符串,再相加  \*/  @Test  **public** **void** fun6(){  String str = " 红藕香残玉簟秋 轻解罗裳独上兰舟 ! ";  //方法一:按照空格截取字符串  String[] strArr = str.split(" ");  **for** (**int** i = 0; i < strArr.length; i++) {  strArr[i] = strArr[i].trim();  }  System.***out***.println(Arrays.*toString*(strArr));  String str1 = "";  **for** (**int** i = 0; i < strArr.length; i++) {  **if** (strArr[i] != **null** && strArr[i].trim() != **null**) {  str1 +=strArr[i];  }  }  System.***out***.println(str1);  }  /\*\*  \* 方式二:使用替换的方式去除空格  \* replaceAll(String regex, String replacement)  使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。  \*/  @Test  **public** **void** fun7(){  String str = " 红藕香残玉簟秋 轻解罗裳独上兰舟 ! ";  System.***out***.println(str);  str = str.replaceAll(" ", "");  System.***out***.println(str);  str = str.replace("玉簟", "空格");  System.***out***.println(str);  str = "1231234123";  System.***out***.println(str.replaceFirst("1", "空格"));  } |



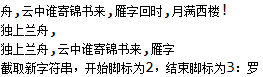


### 7.3.3.2 获取子字符串

我们知道字符串底层是字符数组，那么我们可不可以获取数组的一部分呢？也就是获取字符串的一部分呢？实际上是可以的，查看API发现有以下几种方式可以用于获取子字符串；

|  |  |
| --- | --- |
| [CharSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) | **[subSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "subSequence(int, int))**(int beginIndex, int endIndex)            返回一个新的字符序列，它是此序列的一个子序列。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[substring](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "substring(int))**(int beginIndex)            返回一个新的字符串，它是此字符串的一个子字符串。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[substring](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "substring(int, int))**(int beginIndex, int endIndex)            返回一个新字符串，它是此字符串的一个子字符串。 |

|  |
| --- |
| **package** cn.stringPractise.subString;  **import** org.junit.Test;  **public** **class** SubString {    /\*\*  \* 获取字符串子序列:截取顾头不顾尾  \* CharSequence是一个接口,String是该接口的一个实现类，截取返回接口这是多态的体现  \*/  @Test  **public** **void** fun8(){  String str = "轻解罗裳独上兰舟,云中谁寄锦书来,雁字回时,月满西楼!";  String str1 = str.substring(7);  String str2 = str.substring(4, 9);  CharSequence str3 = str.subSequence(4, 19);  System.***out***.println(str1);  System.***out***.println(str2);  System.***out***.println(str3);  //截取字符串，从指定脚标截取到指定脚标，包含开始脚标，不包含结束脚标,注意结束脚标不能比字符串长度大，也不能比开始脚标小，否则会脚标越界异常  System.***out***.println("截取新字符串，开始脚标为2，结束脚标为3："+str.substring(2, 3));  // System.out.println(str.substring(-1));  // System.out.println(str.substring(0, 28));StringIndexOutOfBoundException  }  } |



注意：字符串中空格占据一个索引位置；

截取的时候顾头不顾尾；

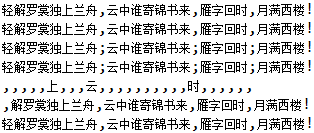
### 7.3.3.3 字符串替换

字符串既然是数组，那么可不可以对其中一个元素进行替换呢？答案是可以的；查看API有如下方法：

|  |  |
| --- | --- |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[replace](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "replace(char, char))**(char oldChar, char newChar)            返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[replace](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "replace(java.lang.CharSequence, java.lang.CharSequence))**([CharSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) target, [CharSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) replacement)            使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[replaceAll](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "replaceAll(java.lang.String, java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) regex, [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) replacement)            使用给定的 replacement 替换此字符串所有匹配给定的[正则表达式](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/util/regex/Pattern.html" \l "sum)的子字符串。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[replaceFirst](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "replaceFirst(java.lang.String, java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) regex, [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) replacement)            使用给定的 replacement 替换此字符串匹配给定的[正则表达式](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/util/regex/Pattern.html" \l "sum)的第一个子字符串。 |

//替换功能，把老的字符或字符串，替换成新的字符或字符串,注意字符只能替换成字符，字符串只能替换成字符串（对象）

|  |
| --- |
| /\*\*  \* 字符串替换  \* String replace(char oldChar, char newChar)  返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。  String replace(CharSequence target, CharSequence replacement)  使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。  String replaceAll(String regex, String replacement)  使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。  replaceFirst(String regex, String replacement)  使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。  \*/  @Test  **public** **void** fun9(){  String str = "轻解罗裳独上兰舟,云中谁寄锦书来,雁字回时,月满西楼!";  String str1 = str.replace(",", ";");  System.***out***.println(str.replace(";;", "qq"));  String str2 = str.replace(",", ";");  String str3 = str.replaceAll("[^上云时]", ",");//替换除了上云时的三个字符的任意一个字符  String str4 = str.replaceFirst("[^上云时]", ",");//替换除了上云时的三个字符的任意一个字符的第一个  System.***out***.println(str);  System.***out***.println(str1);  System.***out***.println(str2);  System.***out***.println(str3);  System.***out***.println(str4);  str1 = str.replace("A", "B");//如果OldChar没有,那么会返回原字符串  System.***out***.println(str1);  } |



注意：oldChar如果不存在会返回原字符串

OldChar如果多次出现，那么会全部替换

要替换的OldChar和字符串中的应该保持大小写一致，字符串区别大小写，A和a是不同的两个字符

### 7.3.3.4 字符串比较判断

如果我们有两个字符串,我们应该怎么比较这两个字符串呢?当然我指的是内容的比较,下面我们就来看看字符串的比较;【关于有内容的对象，我们往往有两种比较方法，一种是内容的比较“equals(),hashcode()”，一种是内存地址的比较“==”】

#### 7.3.3.4.1 判断字符串的开始和结尾

判断字符串的开始和结尾也是我们经常用到的一种判断,查看API知道一下方法可以用来判断字符串的开始和结尾;

|  |  |
| --- | --- |
| boolean | **[startsWith](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "startsWith(java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) prefix)            测试此字符串是否以指定的前缀开始。 |
| boolean | **[startsWith](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "startsWith(java.lang.String, int))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) prefix, int toffset)            测试此字符串从指定索引开始的子字符串是否以指定前缀开始。 |
| boolean | **[endsWith](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "endsWith(java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) suffix)            测试此字符串是否以指定的后缀结束。 |

|  |
| --- |
| /\*\*  \* 判断字符串的开始和结尾  \*/  @Test  **public** **void** fun10(){  String str = "轻解罗裳独上兰舟,云中谁寄锦书来,雁字回时,月满西楼!";  System.*out*.println("字符串str是否是以字符轻开始和结尾:"+str.startsWith("轻")+"---"+str.endsWith("一"));  System.*out*.println("字符串str是否是以字符!开始和结尾:"+str.startsWith("!")+"---"+str.endsWith("!")+"---"+str.startsWith("上", 5));  **if**(str.indexOf(0) == '!'){  System.*out*.println("字符串str是以!开始的!");  }**else** **if** (str.lastIndexOf("!") == (str.length()-1)) {  System.*out*.println("字符串str是以!结尾的");  }  } |



注意:Java中涉及到开始索引和结束索引的问题都会遵循顾头不顾尾的原则!

#### 7.3.3.4.2 判断字符串是否相等

在Java中对字符串最常用的判断还是判断字符串的内容是否相同,比如我们在登录中的校验等等;Java中比较字符串分为两种情况,区分大小写和不区分大小写;

|  |  |
| --- | --- |
| boolean | **[equals](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "equals(java.lang.Object))**([Object](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) anObject)            将此字符串与指定的对象比较。 |
| boolean | **[equalsIgnoreCase](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "equalsIgnoreCase(java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) anotherString)            将此 String 与另一个 String 比较，不考虑大小写。 |

|  |
| --- |
| /\*\*  \* 判定字符串内容  \*/  @Test  **public** **void** fun11(){  String str1 = "abc";  String str2 = **new** String("abc");  String str3 = **new** String("Abc");  String str4 = "abc";  System.*out*.println(str1.equals(str2));  System.*out*.println(str1.equals(str3));  System.*out*.println(str1 == str2);  System.*out*.println(str1 == str4);  System.*out*.println(str1.equalsIgnoreCase(str3));  } |



贴出equalsIgnoreCase源码:

|  |
| --- |
| **public** **boolean** equalsIgnoreCase(String anotherString) {  //地址相同自然为true  **return** (**this** == anotherString) ? **true**  **//String为null,那就是空指针异常了,也没意义**  : (anotherString != **null**)  //长度不同,自然false  && (anotherString.value.length == value.length)  //两个字符串转变大写,然后比较  && regionMatches(**true**, 0, anotherString, 0, value.length);  } |
| **public** **boolean** regionMatches(**boolean** ignoreCase, **int** toffset,  String other, **int** ooffset, **int** len) {  **char** ta[] = value;  **int** to = toffset;  **char** pa[] = other.value;  **int** po = ooffset;  // Note: toffset, ooffset, or len might be near -1>>>1.  **if** ((ooffset < 0) || (toffset < 0)  || (toffset > (**long**)value.length - len)  || (ooffset > (**long**)other.value.length - len)) {  **return** **false**;  }  **while** (len-- > 0) {  **char** c1 = ta[to++];  **char** c2 = pa[po++];  **if** (c1 == c2) {  **continue**;  }  **if** (ignoreCase) {  **char** u1 = Character.*toUpperCase*(c1);  **char** u2 = Character.*toUpperCase*(c2);  **if** (u1 == u2) {  **continue**;  }  **if** (Character.*toLowerCase*(u1) == Character.*toLowerCase*(u2)) {  **continue**;  }  }  **return** **false**;  }  **return** **true**;  } |

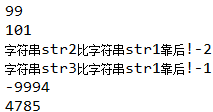
#### 7.3.3.4.3 按字典顺序判断字符串

|  |  |
| --- | --- |
| int | **[compareTo](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "compareTo(java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) anotherString)            按字典顺序比较两个字符串。 |
| int | **[compareToIgnoreCase](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "compareToIgnoreCase(java.lang.String))**([String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) str)            按字典顺序比较两个字符串，不考虑大小写。 |

查看API发现String有两种方法比较字符串的顺序,如下:

按字典顺序比较两个字符串。该比较基于字符串中各个字符的 Unicode 值。按字典顺序将此 String 对象表示的字符序列与参数字符串所表示的字符序列进行比较。如果按字典顺序此 String 对象位于参数字符串之前，则比较结果为一个负整数。如果按字典顺序此 String 对象位于参数字符串之后，则比较结果为一个正整数。如果这两个字符串相等，则结果为 0；compareTo 只在方法 [equals(Object)](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "equals(java.lang.Object)) 返回 true 时才返回 0。

|  |
| --- |
| **package** cn.stringPractise.compare;  **import** org.junit.Test;  **public** **class** StringCompare {    /\*\*  \* 按照字典顺序比较字符串  \* A.compareTo(B);如果B比A靠前,那么返回的是靠前的位数,比如c比e靠前两位返回2,如果靠后,返回的靠后的位数,靠前返回正值,靠后返回负值  \* 比较的时候,如果第一位相同,比较第二位,依次类推  \*/  @Test  **public** **void** fun12(){  String str1 = "abc";  String str2 = "abec";  String str3 = "bace";  System.***out***.println(str1.codePointAt(2));  System.***out***.println(str2.codePointAt(2));  // System.out.println(str3.codePoints());  **if**(str1.compareTo(str2) > 0){  System.***out***.println("字符串str2比字符串str1靠前!"+str1.compareTo(str2));  }**else** **if**(str1.compareTo(str3)>0){  System.***out***.println("字符串str3比字符串str1靠前!"+str1.compareTo(str3));  }**else** {  System.***out***.println("字符串str2比字符串str1靠后!"+str1.compareTo(str2));  System.***out***.println("字符串str3比字符串str1靠后!"+str1.compareTo(str3));  }  str1 = "我是中国人!";  str2 = "我是男人!";  str3 = "你是中国人吗?是";  System.***out***.println(str1.compareTo(str2));  System.***out***.println(str1.compareTo(str3));  }  } |



注意:compareTo()只会在equals()返回true时才会返回0;

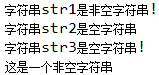
我们知道String的底层是字符数组,而字符在计算机中是以Unicode编码集进行编码的,所以String提供了几个获取Unicode对应编码的方法,这个上面我们在7.3.2.3中有介绍

#### 7.3.3.4.4 对字符串进行非空判断

我们在获取到一个字符串时,会首选对其进行一个非空判断,如果是空字符串,那么就下来的做多就是没意义的;在String中对字符串进行非空判断使用双重判断:null和isEmpty()

|  |  |
| --- | --- |
| boolean | **[isEmpty](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "isEmpty())**()            当且仅当 [length()](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "length()) 为 0 时返回 true。 |

|  |
| --- |
| /\*\*  \* 非空判断  \*/  @Test  **public** **void** fun14(){  String str1 = " ";  String str2 = **null**;  String str3 = "";  **if**(str1.isEmpty() || str1 == **null**){  System.*out*.println("字符串str1是空字符串!");  }**else** {  System.*out*.println("字符串str1是非空字符串!");  }  **if**(str2 == **null** || str2.isEmpty()) {  System.*out*.println("字符串str2是空字符串");  }  **if**(str3.isEmpty()||str3 == **null**){  System.*out*.println("字符串str3是空字符串!");  }  //非空判断  **if**(str1 != **null** && str1.length() != 0){  System.*out*.println("这是一个非空字符串");  }  } |



注意:我们在进行一个空字符串判断时,最好将null判断放在前面,因为如果字符串真是一个null值得字符串,此时null判断放在后面会出现下面异常

// System.out.println(str2.isEmpty());注意对null的操作会空指针异常



实际上开发过程中我们从表单中获取的字符串比较非空使用的是,String != null && String.trim.isEmpty,【要考虑空格】

#### 7.3.3.4.5 字符串中的其它比较

|  |  |
| --- | --- |
| boolean | **[contains](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "contains(java.lang.CharSequence))**([CharSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) s)            当且仅当此字符串**包含**指定的 char 值序列时，返回 true。 |
| boolean | **[contentEquals](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "contentEquals(java.lang.CharSequence))**([CharSequence](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) cs)            将此**字符串**与指定的 CharSequence 比较。 |
| boolean | **[contentEquals](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "contentEquals(java.lang.StringBuffer))**([StringBuffer](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) sb)            将此字符串与指定的 StringBuffer 比较。 |

|  |
| --- |
| /\*\*  \* 字符串中的其它比较  \*/  @Test  **public** **void** fun16(){  String str = "此情可待成追忆?";  System.*out*.println(str.contains("可追"));  System.*out*.println(str.contains("可待"));  System.*out*.println(str.contentEquals("可待"));  System.*out*.println(str.contentEquals("此情可待成追忆?"));  StringBuffer sb = **new** StringBuffer("此情可待成追忆?");  StringBuffer sb2 = **new** StringBuffer("此情追忆");  System.*out*.println(str.contentEquals(sb));  System.*out*.println(str.contentEquals(sb2));  } |

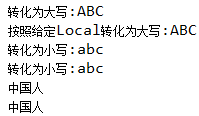


### 7.3.3.5 对字符串中字母进行大小写转换

我们在开发中经常用到字符串,有时候我们需要对收到的字符串进行整理后才好操作,这个整理就包含字符串字母的大小写转换,查看API发现对字符串大小写的转换有如下方法:

|  |  |
| --- | --- |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[toLowerCase](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "toLowerCase())**()            使用默认语言环境的规则将此 String 中的所有字符都转换为小写。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[toLowerCase](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "toLowerCase(java.util.Locale))**([Locale](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/util/Locale.html" \o "java.util 中的类) locale)            使用给定 Locale 的规则将此 String 中的所有字符都转换为小写。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[toString](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "toString())**()            返回此对象本身（它已经是一个字符串！）。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[toUpperCase](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "toUpperCase())**()            使用默认语言环境的规则将此 String 中的所有字符都转换为大写。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[toUpperCase](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/String.html" \l "toUpperCase(java.util.Locale))**([Locale](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/util/Locale.html" \o "java.util 中的类) locale)            使用给定 Locale 的规则将此 String 中的所有字符都转换为大写。 |

|  |
| --- |
| /\*\*  \* 字符串字母大小写转换  \*/  @Test  **public** **void** fun13(){  String str = "aBc";  System.*out*.println("转化为大写:"+str.toUpperCase());  System.*out*.println("按照给定Local转化为大写:"+str.toUpperCase(Locale.*US*));  System.*out*.println("转化为小写:"+str.toLowerCase());  System.*out*.println("转化为小写:"+str.toLowerCase(Locale.*CHINA*));  str = "中国人";  System.*out*.println(str.toLowerCase(Locale.*US*));  System.*out*.println(str.toUpperCase());  } |



注意:对字符进行大小写转换是非英文字符不受影响,原字符符合规定的原样输出.

### 7.3.3.6 字符串分割

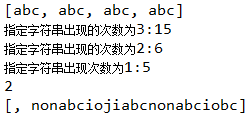
Split()方法根据指定的分隔符对字符进行分割,并将分割后的结果存放在字符串数组中.split()方法提供了两种重载形式.

|  |  |
| --- | --- |
| [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类)[] | **[split](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "split(java.lang.String))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) regex)            根据给定[正则表达式](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../util/regex/Pattern.html" \l "sum)的匹配拆分此字符串。 |
| [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类)[] | **[split](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "split(java.lang.String, int))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) regex, int limit)            根据匹配给定的[正则表达式](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/util/regex/Pattern.html" \l "sum)来拆分此字符串。 |

前者是完全分割,后者限制拆分的次数,拆分后的次数,实际就是结果数组的长度.再者参数可以是分割的字符,也可以是正则表达式.

注意:没有统一的对字符进行分割的符号.如果想定义多个分隔符,可使用符号”|”,例如,”,|=”表示分隔符分别为”,”和”=”.

|  |
| --- |
| /\*\*  \* 字符串分割练习  \*/  @Test  **public** **void** fun21(){  String str = "cnonabciojiabcnonabciocnonabciojiabcnonabciocnonabciojiabcnonabciocnonabciojiabcnonabciocnonabciojiabcnonabcio";  String str1 = "abcnonabciojiabcnonabcioabc";  String str2 = "abcnonabciojiabcnonabciobc";  String str3 = "abc.abc.abc.abc";  System.*out*.println(Arrays.*toString*(str3.split("\\.")));  test(str);  test(str1);  test(str2);  test1(str2);  }  /\*\*  \* 分割字符串的三种情况  \* 头部存在字符串 abc123abc123abc123 次数按照abc分割,那么结果数组长度是3,但是abc出现次数arr.length  \* 尾部存在字符串 123abc123abc123abc 按照abc分割,那么结果数组长度3,abc出现次数arr.length  \* 头尾都存在字符串 abc123abc123abc123abc123abc 按照abc分割,数组长度4,abc出现次数arr.length+1  \* 头尾都没有指定字符串 123abc123abc123abc123 按照abc分割,数组长度4,abc出现次数arr.length-1  \*/  **public** **void** test(String str){  //分割字符串  String[] strArr = str.split("abc");  **if** (str.startsWith("abc") && str.endsWith("abc")) {  System.*out*.println("指定字符串出现的次数为2:"+(strArr.length+1));  //在这里要用单或,因为如果是头尾都是abc那么还是真,所以我们把头尾的判断放在前面,单或放后边,表面了这个错误  }**else** **if**(str.startsWith("abc") | str.endsWith("abc")){  System.*out*.println("指定字符串出现次数为1:"+strArr.length);  }**else** {  System.*out*.println("指定字符串出现的次数为3:"+(strArr.length-1));  }  }  /\*\*  \* 限定次数分割,给定的参数是分隔后的字符串长度  \*/  **public** **void** test1(String str){  //分割字符串  String[] strArr = str.split("abc",2);  System.*out*.println(strArr.length);  System.*out*.println(Arrays.*toString*(strArr));  } |



## 7.3.4 字符串转变为其它类型

我们将其它数据类型转变为字符串只需要一个+号就可以,那么将字符串转变为其它数据类型呢?这可就复杂了点,String提供了很多方法用来转换,我们下面来一一认识一下.

### 7.3.4.1 字符串转变为字节数组

在Java中一个字符占有两个字节(为什么占两个字节后面我们在Question中单独介绍)即16位,那么我们知道在Java中有两种流,字符流和字节流,那么当我们在字符流和字节流之间转换的时候其实就是将字符转换为字节,网络中传输只能是字节流,所以字符转换成字节是很必要的;查看API我们发现String提供了几种方法供我们将字符转换为字节数组;

|  |  |  |  |
| --- | --- | --- | --- |
| byte[] | | | **[getBytes](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "getBytes())**()            使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。 |
| byte[] | | | **[getBytes](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "getBytes(java.nio.charset.Charset))**([Charset](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset 中的类) charset)            使用给定的 [charset](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset 中的类) 将此 String 编码到 byte 序列，并将结果存储到新的 byte 数组。 |
| void | **[getBytes](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "getBytes(int, int, byte[], int))**(int srcBegin, int srcEnd, byte[] dst, int dstBegin)            **已过时。** *该方法无法将字符正确转换为字节。从 JDK 1.1 起，完成该转换的首选方法是通过 [getBytes()](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "getBytes()) 方法，该方法使用平台的默认字符集。* | | |
| byte[] | | **[getBytes](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "getBytes(java.lang.String))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) charsetName)            使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。 | |

查看平台默认字符集的方法:

在命令提示符的“菜单”上，右击-----》属性----》当前代码页 936（ANSI/OEM -简体中文GBK）

|  |  |
| --- | --- |
| /\*\*  \* 将字符串转变为字节数组  \* **@throws** UnsupportedEncodingException  \*/  @Test  **public** **void** fun17() **throws** UnsupportedEncodingException{  look();  String str = "走了,记着你的好,也希望你想起我也不都是恶心!";  **byte**[] bytes = str.getBytes();//使用平台默认字符集编码GBK  System.*out*.println(Arrays.*toString*(bytes));  String str1 = **new** String(bytes, "utf-8");//解码字符集和编码字符集不同,乱码  System.*out*.println(str1);  String str2 = **new** String(bytes, "GBK");  System.*out*.println(str2);  bytes = str.getBytes("utf-8");//使用utf-8编码  String str3 = **new** String(bytes);//使用平台默认字符集解码GBK,乱码  String str4 = **new** String(bytes, "utf-8");//使用utf-8解码  System.*out*.println(str3);  System.*out*.println(str4);  }  //使用java查询系统默认字符集  **public** **void** look(){  Properties prop = System.*getProperties*();  prop.list(System.*out*);  System.*out*.println("-------以上为JVM的所有属性值-------");  String name = System.*getProperty*("sun.jnu.encoding");  System.*out*.println("系统默认的字符集为:"+name);  } | |
| -- listing properties --  java.runtime.name=Java(TM) SE Runtime Environment  sun.boot.library.path=D:\Java\JDK\jre\bin  java.vm.version=24.45-b08  java.vm.vendor=Oracle Corporation  java.vendor.url=http://java.oracle.com/  path.separator=;  java.vm.name=Java HotSpot(TM) Client VM  file.encoding.pkg=sun.io  user.script=  user.country=CN  sun.java.launcher=SUN\_STANDARD  sun.os.patch.level=Service Pack 1  java.vm.specification.name=Java Virtual Machine Specification  user.dir=E:\WorkSpace\JavaSE\Chapter 6 Comment...  java.runtime.version=1.7.0\_45-b18  java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment  java.endorsed.dirs=D:\Java\JDK\jre\lib\endorsed  os.arch=x86  java.io.tmpdir=C:\Users\You\AppData\Local\Temp\  line.separator=  java.vm.specification.vendor=Oracle Corporation  user.variant=  os.name=Windows 7  sun.jnu.encoding=GBK  java.library.path=D:\Java\JDK\bin;C:\Windows\Sun\Java\b...  java.specification.name=Java Platform API Specification  java.class.version=51.0  sun.management.compiler=HotSpot Client Compiler  os.version=6.1  user.home=C:\Users\You  user.timezone=  java.awt.printerjob=sun.awt.windows.WPrinterJob  file.encoding=GBK  java.specification.version=1.7  user.name=You  java.class.path=E:\WorkSpace\JavaSE\Chapter 6 Comment...  java.vm.specification.version=1.7  sun.arch.data.model=32  java.home=D:\Java\JDK\jre  sun.java.command=org.eclipse.jdt.internal.junit.runner...  java.specification.vendor=Oracle Corporation  user.language=zh  awt.toolkit=sun.awt.windows.WToolkit  java.vm.info=mixed mode  java.version=1.7.0\_45  java.ext.dirs=D:\Java\JDK\jre\lib\ext;C:\Windows\Su...  sun.boot.class.path=D:\Java\JDK\jre\lib\resources.jar;D:\...  java.vendor=Oracle Corporation  file.separator=\  java.vendor.url.bug=http://bugreport.sun.com/bugreport/  sun.cpu.endian=little  sun.io.unicode.encoding=UnicodeLittle  sun.desktop=windows  sun.cpu.isalist=pentium\_pro+mmx pentium\_pro pentium+m...  -------以上为JVM的所有属性值-------  系统默认的字符集为:GBK  [-41, -33, -63, -53, 44, -68, -57, -41, -59, -60, -29, -75, -60, -70, -61, 44, -46, -78, -49, -93, -51, -5, -60, -29, -49, -21, -58, -16, -50, -46, -46, -78, -78, -69, -74, -68, -54, -57, -74, -15, -48, -60, 33]  ????,????????,??????????????????????!  走了,记着你的好,也希望你想起我也不都是恶心!  璧颁簡,璁扮潃浣犵殑濂?涔熷笇鏈涗綘鎯宠捣鎴戜篃涓嶉兘鏄伓蹇?  走了,记着你的好,也希望你想起我也不都是恶心! |

### 7.3.4.2 从字符串中将字符串片段添加到字符数组中

|  |  |
| --- | --- |
| void | **[getChars](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "getChars(int, int, char[], int))**(int srcBegin, int srcEnd, char[] dst, int dstBegin)            将字符从此字符串复制到目标字符数组。 |

public void getChars(int srcBegin,

int srcEnd,

char[] dst,

int dstBegin)

将字符从此字符串复制到目标字符数组。

要复制的第一个字符位于索引 srcBegin 处；要复制的最后一个字符位于索引 srcEnd-1 处（因此要复制的字符总数是 srcEnd-srcBegin）。要复制到 dst 子数组的字符从索引 dstBegin 处开始，并结束于索引：

dstbegin + (srcEnd-srcBegin) - 1

|  |
| --- |
| /\*\*  \* 将字符串片段添加到字符数组  \*/  @Test  **public** **void** fun18(){  String str = "记得你的好,希望留给你的也不都是恶心!做了一场梦似的,咱俩结的梁子解了.";  **char**[] ch = **new** **char**[50];  ch[0] = '走';  ch[1] = '了';  ch[2] = ',';  ch[3] = '就';  ch[4] = '跟';  ch[45] = ';';  str.getChars(0, 19, ch, 3);//如果覆盖数组位置上有元素,那么会替代,而不是向后推移  System.*out*.println(ch);  } |



注意:这个方法没有返回值,这个方法调用,那么数组就已经被改变;

|  |
| --- |
| //getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 将字符从此字符串复制到目标字符数组。但是截取的字符串，不能比字符数组长度减去存放其实脚标的值大，否则脚标越界异常 |

注意索引越界异常:

0<srcEnd-srcBegin+dstBegin<=dst.length

|  |
| --- |
| str.getChars(0, 19, ch, 40);ArrayIndexOutOfBoundException  System.***out***.println(ch); |

假如头尾相等的话,此时相当于没有改变,如下:

|  |
| --- |
| str.getChars(0, 0, ch, 0);  System.***out***.println(ch); |



### 7.3.4.3 将字符串转变为字符数组

我们知道字符串的底层是字符数组,但是字符串是一个String的对象,而字符数组是Arrays的对象,那么我们可不可以将字符串转变为Arrays对象呢?可以的,这样我们经过转变就可以进行数组的一些操作了.

|  |  |
| --- | --- |
| char[] | **[toCharArray](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "toCharArray())**()            将此字符串转换为一个新的字符数组。 |

|  |
| --- |
| /\*\*  \* 将字符串转变为字符数组  \*/  @Test  **public** **void** fun19(){  String str ="走了,记得你的好,希望你想起我也不都是恶心!";  **char**[] ch = str.toCharArray();  System.*out*.println(ch);  **int** i = Arrays.*binarySearch*(ch, '是');  System.*out*.println("字符是出现的索引为:"+i);  *stringDe*();  }  //对字符串中字符进行自然排序 我们把字符串转换成字符数组，然后排序，最后再转换成字符串  **public** **static** **void** stringDe(){  String str = "1221aasafd!";  **char**[] ch = str.toCharArray();  Arrays.*sort*(ch);//将字符数组排序  str = String.*copyValueOf*(ch);  //str = String.valueOf(ch);上下输出一样  System.*out*.println("输出字符串："+str);  } |



## 7.3.5 其它类型向字符串的转变

我们知道其它类型向字符串转变最简单的是+,但是如果是数组呢?那么+只会把数组的HashCode值加在字符串上,这是将数组转变成字符串就很必要了,这就是本节学习的一个点,查看API,有如下方法:

|  |  |
| --- | --- |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[copyValueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "copyValueOf(char[]))**(char[] data)            返回指定数组中表示该字符序列的 String。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[copyValueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "copyValueOf(char[], int, int))**(char[] data, int offset, int count)            返回指定数组中表示该字符序列的 String。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[valueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "valueOf(boolean))**(boolean b)            返回 boolean 参数的字符串表示形式。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[valueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "valueOf(char))**(char c)            返回 char 参数的字符串表示形式。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[valueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "valueOf(char[]))**(char[] data)            返回 char 数组参数的字符串表示形式。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[valueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "valueOf(char[], int, int))**(char[] data, int offset, int count)            返回 char 数组参数的特定子数组的字符串表示形式。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[valueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "valueOf(double))**(double d)            返回 double 参数的字符串表示形式。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[valueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "valueOf(float))**(float f)            返回 float 参数的字符串表示形式。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[valueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "valueOf(int))**(int i)            返回 int 参数的字符串表示形式。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[valueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "valueOf(long))**(long l)            返回 long 参数的字符串表示形式。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[valueOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "valueOf(java.lang.Object))**([Object](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) obj)            返回 Object 参数的字符串表示形式。 |

注意:这些方法都是static,那么意味着只要类名称就可以调用;

|  |
| --- |
| /\*\*  \* 将其它数据类型转变为字符串  \*/  @Test  **public** **void** fun20(){  Object object = **new** Object();  **char**[] ch = {'我','好','类','啊','啊','啊'};  System.*out*.println(String.*valueOf*(object));//会先调用Object的toString(),然后输出String.  System.*out*.println(String.*copyValueOf*(ch));  System.*out*.println(String.*copyValueOf*(ch, 2, 4));//注意索引越界异常  } |



## 7.3.6 格式化字符串

格式化字符串是程序中经常用到的,主要包括日期格式化、时间格式化、日期/时间组合的格式化和常规类型格式化.

### 7.3.6.1 时间日期格式化

在应用程序设计中,经常需要显示时间和日期。format()方法通过给定的特殊换符作为参数来实现对日期和时间的格式化。format()方法有以下几种重载形式.

|  |  |
| --- | --- |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[format](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "format(java.util.Locale, java.lang.String, java.lang.Object...))**([Locale](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/util/Locale.html" \o "java.util 中的类) l, [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) format, [Object](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类)... args)            使用指定的语言环境、格式字符串和参数返回一个格式化字符串。 |
| static [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[format](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \l "format(java.lang.String, java.lang.Object...))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) format, [Object](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类)... args)            使用指定的格式字符串和参数返回一个格式化字符串。 |

下面我们就来分别针对日期和时间的格式化来讲解一下:

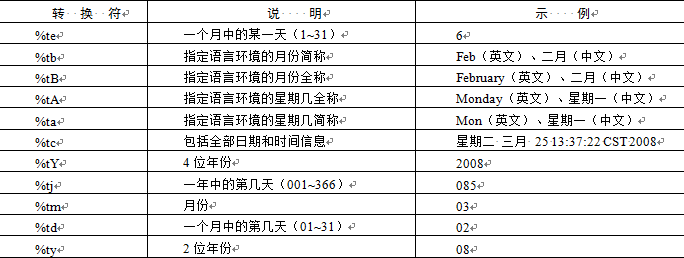
#### 7.3.6.1.1 日期格式化

我们知道Date可以获取当前时间,那么它得到的是一个以毫秒为单位的long值,我们怎么将其转换为我们常见的的显而已懂的时间格式呢?这就是这章节我们要探讨的,废话不多,代码先行;

|  |
| --- |
| /\*\*  \* 时间类型格式化  \*/  @Test  **public** **void** fun1(){  Date date = **new** Date();  String time = String.*format*("%te", date);  System.*out*.println("格式化后的时间--天数:"+time);  } |



上述代码中我们发现format中两个参数后边的是我们获取到的当前时间,前面的我们不认识,但是观察结果,我们发现的到的是当前的日期天,那么可以猜测这个字符串是用来转换时间为天的,常见的日期转换符有如下几种:



知道了上边的转换符,我们试着将时间以4位年份,月份全称,2位日期的形式返回:

|  |
| --- |
| @Test  **public** **void** fun2(){  Date date = **new** Date();  String year = String.*format*("%tY", date);  String month = String.*format*("%tB", date);  String day = String.*format*("%td", date);  System.*out*.println("日期的格式:"+(year+month+day));  //试着给定平台环境输出  year = String.*format*(Locale.*US*,"%tY", date);  month = String.*format*(Locale.*US*,"%tB", date);  day = String.*format*(Locale.*US*,"%td", date);  System.*out*.println("美国日期环境的格式:"+(year+month+day));  } |



#### 7.3.6.1.2 时间格式化

使用format()方法不仅可以完成日期的格式化,也可以实现时间的格式化.时间格式化转换符要比日期转换符更多、更精确，它可以将日期格式化为时，分，秒，毫秒。格式化时间的转换符如下：



|  |
| --- |
| /\*\*  \* 时间转换  \*/  @Test  **public** **void** fun3(){  Date date = **new** Date();  String hour = String.*format*("%tH", date);  String minute = String.*format*("%tM", date);  String second = String.*format*("%tS", date);  String msec = String.*format*("%tL", date);  System.*out*.println("时间格式化:"+(hour+":"+minute+":"+second+":"+msec));  hour = String.*format*(Locale.*UK*,"%tH", date);  minute = String.*format*(Locale.*UK*,"%tM", date);  second = String.*format*(Locale.*UK*,"%tS", date);  msec = String.*format*(Locale.*UK*,"%tL", date);  System.*out*.println("时间格式化:"+(hour+":"+minute+":"+second+":"+msec));  } |



#### 7.3.6.1.3 日期和时间的组合格式化

在上面我们知道了格式化日期和时间的方法，那么将日期和时间放在一起呢？毕竟以后很多时候我们还是要格式化日期和时间的组合的，常见的日期和时间的格式如下：

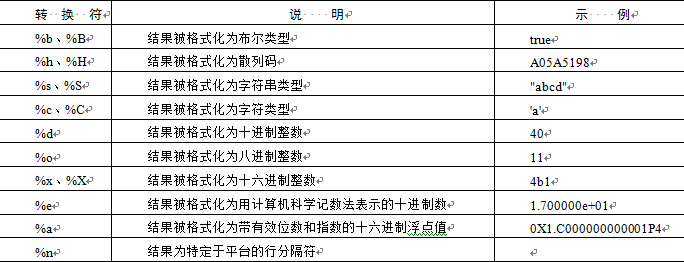


|  |
| --- |
| /\*\*  \* 日期和时间的组合格式化  \*/  @Test  **public** **void** fun4(){  Date date = **new** Date();  String year = String.*format*(Locale.*US*,"%tF", date);  String time = String.*format*(Locale.*US*,"%tT", date);  System.*out*.println(year+" "+time);  } |



#### 7.3.6.1.4 常规类型格式化

在程序设计过程中，经常需要对常规类型的数据进行格式化，如格式化为整数、格式化为科学计数表示等，在Java语言中可以使用常规类型的格式化转换符来实现。下图列出了常规类型转换符。



|  |
| --- |
| /\*\*  \* 常规类型格式化  \*/  @Test  **public** **void** fun5(){  String str1 = String.*format*("%d", 400/2);  String str2 = String.*format*("%b", 3>5);  String str3 = String.*format*("%x", 400);  System.*out*.println("结果以十进制格式表示!"+str1);  System.*out*.println("结果以boolean类型显示:"+str2);  System.*out*.println("结果以16进制显示:"+str3);  } |



### 7.3.6.2 格式化输出

#### 7.3.6.2.1 printf()

学过C的知道在C中有一种方法printf（）用于格式化输出。Java源于C自然也会具有C的一些优点了，JavaSE5推出了与C语言中风格一致的printf（）方法，这不仅使得控制输出的代码更加简单，同时也给与Java开发者对于输出格式与排列更强大的控制能力。

C语言中的printf（）并不能像Java那样连接字符串，它使用一个简单的格式化字符串，加上要插入其中的值，然后将其格式化输出。Printf（）并不使用重载的“+”操作符（C没有重载）来连接引号内的字符串或者字符串变量，而是使用特殊的占位符来表示数据将来的位置。而且它还将插入格式化字符串的参数，以逗号分隔，排成一行。

例如：

|  |
| --- |
| //System.*out*.printf("Row 1: [%d %f]\n", x,y);  System.*out*.printf("Row 1: [%d %f]\n", 20,2.3); |

这一会代码在运行的时候，首先将x的值插入到%d的位置,然后将y的值插入到%f的位置.这些占位符被称为格式修饰符,它们不但说明了插入数据的位置,同时还说明了将插入什么类型的变量,以及如何对其格式化.在这个例子中,%d表示x是一个整数,%f表示y是以一个浮点数(float或者double).

|  |
| --- |
| /\*\*  \* printf()  \*/  @Test  **public** **void** fun6(){  String str = "x";  System.*out*.printf("Row 1: [%d %f]\n", 20,2.3);  } |



#### 7.3.6.2.2 System.out.format()

JavaSE5引入的format方法可用于PrintStream或PrintWriter对象其中也包括System.out对象。Format（）方法模仿自C的printf().如果你比较怀旧,也可以使用printf().一下是一个简单实例:

|  |
| --- |
| /\*\*  \* syso.format()  \*/  @Test  **public** **void** fun7(){  **int** x = 5;  **double** y = 5.332542;  System.*out*.println("Row 1:[" + x + " " + y +"]");//笨办法  System.*out*.printf("Row 1:[%d,%f]\n",x,y);  System.*out*.format("Row 1:[%d,%f]",x,y);  } |



可以看到,format()与printf()是等价的,它们只需要一个简单的格式化字符串,加上一串参数即可,每个参数对应一个格式修饰符.

#### 7.3.6.2.3 格式化的底层Formatter类

在上面我们知道了字符串的格式化和常见的输出格式化,那么我们有没有想过这些格式化的底层都是什么呢?下面是底层代码片段:

|  |
| --- |
| **public** PrintStream printf(String format, Object ... args) {  **return** format(format, args);  } |
| **public** PrintStream format(String format, Object ... args) {  **try** {  **synchronized** (**this**) {  ensureOpen();  **if** ((formatter == **null**)  || (formatter.locale() != Locale.*getDefault*()))  formatter = **new** Formatter((Appendable) **this**);  formatter.format(Locale.*getDefault*(), format, args);  }  } **catch** (InterruptedIOException x) {  Thread.*currentThread*().interrupt();  } **catch** (IOException x) {  trouble = **true**;  }  **return** **this**;  } |
| **public** **static** String format(String format, Object... args) {  **return** **new** Formatter().format(format, args).toString();  } |

上面是几种格式化的底层实现,我们发现最后都是回到了Formatter类上.实际上在Java中所有的格式化功能都是由java.util.Formatter类处理.可以将Formatter看作一个翻译器,它将你的格式化字符串与数据翻译成你需要的结果.当你创建一个Formatter对象的时候,需要向其构造器传递一些信息,告诉它最终的结果将向哪里输出:

|  |
| --- |
| **package** cn.string.three;  **import** java.io.PrintStream;  **import** java.util.Formatter;  **public** **class** Turtle {  **private** String name;  **private** Formatter f;  **public** Turtle(String name, Formatter f) {  **super**();  **this**.name = name;  **this**.f = f;  }  **public** **void** move(**int** x,**int** y){  f.format("%s The Turtle is at (%d,%d)\n", name,x,y);  }  **public** **static** **void** main(String[] args) {  PrintStream ps = System.*out*;  Turtle tommy = **new** Turtle("Terry", **new** Formatter(ps));  tommy.move(0, 0);  tommy.move(4, 8);  }  } |



所有的tommy都将输出到ps也即System.out,而所有的terry则都输出到System.out的一个别名中.Formatter的构造器经过重载可以接受多种输出目的地,不过最常用的还是PrintStream()、OutputStream和File.

#### 7.3.6.2.4 格式化说明符

在插入数据的时候,如果想要控制空格和对齐,你需要更精细复杂的格式修饰符.一下是其抽象的语法:

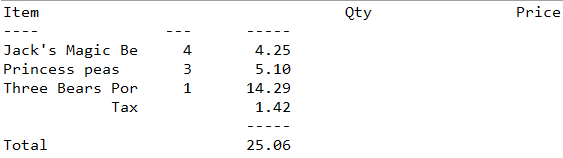
%[argument\_index$][flags][width][.precision]conversion

最常见的应用是控制一个域的最小尺寸,这可以通过指定width来实现.Formatter对象通过在必要时添加空格,来确保一个域至少达到某个长度。在默认的情况下，数据是右对齐，不过可以通过使用“-”符号来改变对齐方向。

与width相对应的是precision，它用来指明最大尺寸。Width可以用于各种类型的数据转换，并且其行为方式都一样。Precision则不然，不是所有类型的数据都能使用precision，而且，应用于不同类型的数据转换时，precision的意义也不同。在将precision应用于String时，它表示打印String时输出的字符的最大数量。而在将precision应用于浮点数时，它表示小数部分要显示出来的位数（默认是6位小数），如果小数位数过多则会舍入，太少则会在尾部补0.由于整数没有小数部分，所以precision无法应用于整数，如果你对整数应用precision，则会触发异常。

下面的程序应用格式修饰符来打印一个购物收据：

|  |
| --- |
| **package** cn.string.three;  **import** java.util.Formatter;  **public** **class** Receipt {  **private** **double** total = 0;  **private** Formatter f = **new** Formatter(System.*out*);  **public** **void** printTitle(){  //width控制字符串的最小尺寸,所以结果会错位,precision控制输出字符串的最大字符数量  f.format("%-20s %20s %20s\n", "Item","Qty","Price");  f.format("%-15s %5s %10s\n", "----","---","-----");  }  **public** **void** print(String name,**int** qty,**double** price){  f.format("%-15.15s %5d %10.2f\n", name,qty,price);  total += price;  }  /\*\*  \* 默认右对齐,-可以改为左对齐,有.说明有precision,  \*/  **public** **void** printTotal(){  f.format("%15s %5s %10.2f\n", "Tax","",total\*0.06);  f.format("%-15s %5s %10s\n", "","","-----");  f.format("%-15s %5s %10.2f\n", "Total","",total\*1.06);  }  **public** **static** **void** main(String[] args) {  Receipt receipt = **new** Receipt();  receipt.printTitle();  receipt.print("Jack's Magic Beans", 4, 4.25);  receipt.print("Princess peas", 3, 5.1);  receipt.print("Three Bears Porridge", 1, 14.29);  receipt.printTotal();  }  } |



查看结果，通过相当简洁的语法，Formatter提供了对空格和对齐的强大控制能力。在该程序中，为了恰当的控制间隔，格式化字符串被重复的利用了多遍。

#### 7.3.6.2.5 Formatter转换

下面的表格包含了最常用的类型转换:

|  |  |
| --- | --- |
| 类型转换字符 | |
| d 整数型(十进制)  c Unicode字符  b Boolean值  s String值  f 浮点数(十进制) | e 浮点数(科学计数)  x 整数(十六进制)  h 散列码(十六进制)  % 字符% |
| **package** cn.stringPractise.format;  **import** java.math.BigInteger;  **import** java.util.Formatter;  **public** **class** Conversion {  **public** **static** **void** main(String[] args) {  Formatter f = **new** Formatter(System.***out***);  **char** u = 'a';  System.***out***.println("u = 'a'");//u = 'a'  f.format("s:%s\n", u);//s:a  // f.format("d:%d\n", u); java.util.IllegalFormatConversionException d != java.lang.Character  f.format("c:%c\n", u);//c:a  f.format("b:%b\n", u);//b:true  // f.format("f:%f\n", u);f != java.lang.Character  // f.format("e:%e\n", u);  // f.format("x:%x\n", u);  f.format("h:%h\n", u);//h:61  System.***out***.println("-------------------------------");  **int** v = 121;  System.***out***.println("v = 121");//v = 121  f.format("d:%d\n", v);//d:121  f.format("c:%c\n", v); //c:y  f.format("b:%b\n", v);//b:true  f.format("s:%s\n", v);//s:121  // f.format("f:%f\n", v);  // f.format("e:%e\n", v);  f.format("x:%x\n", v);//x:79  f.format("h:%h\n", v);//h:79  System.***out***.println("-------------------------------");  BigInteger w = **new** BigInteger("50000000000000");  System.***out***.println("w = new BigInteger(\"50000000000000\")");//w = new BigInteger("50000000000000")  f.format("d: %d\n", w);//d: 50000000000000  // f.format("c: %c\n", w);  f.format("b: %b\n", w);//b: true  f.format("s: %s\n", w);//s: 50000000000000  // f.format("f: %f\n", w);  // f.format("e: %e\n", w);  f.format("x: %x\n", w);//x: 2d79883d2000  f.format("h: %h\n", w);//h: 8842a1a7  System.***out***.println("----------------------------------");  **double** x = 179.543;  System.***out***.println(" x = 179.543");//x = 179.543  // f.format("d: %d\n", x);  // f.format("c: %c\n", x);  f.format("b: %b\n", x);//b: true  f.format("s: %s\n", x);//s: 179.543  f.format("f: %f\n", x);//f: 179.543000  f.format("e: %e\n", x);//e: 1.795430e+02  // f.format("x: %x\n", x);  f.format("h: %h\n", x);//h: 1ef462c  System.***out***.println("----------------------------------");  Conversion y = **new** Conversion();  System.***out***.println("y = new Conversion()");//y = new Conversion()  // f.format("d: %d\n", y);  // f.format("c: %c\n", y);  f.format("b: %b\n", y);//b: true  f.format("s: %s\n", y);//s: cn.stringPractise.format.Conversion@4aa298b7  // f.format("f: %f\n", y);  // f.format("e: %e\n", y);  // f.format("x: %x\n", y);  f.format("h: %h\n", y);//h: 4aa298b7  System.***out***.println("----------------------------------");  **boolean** z = **false**;  System.***out***.println(" z = false");// z = false  // f.format("d: %d\n", z);  // f.format("c: %c\n", z);  f.format("b: %b\n", z);//b: false  f.format("s: %s\n", z);//s: false  // f.format("f: %f\n", z);  // f.format("e: %e\n", z);  // f.format("x: %x\n", z);  f.format("h: %h\n", z);//h: 4d5  System.***out***.println("----------------------------------");  }  } | |

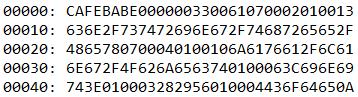
被注释的代码表示,针对相应类型的变量,这些转换是无效的。如果执行这些转换，则会触发异常。

注意：程序中每个变量都用到了b转换。虽然它对各种类型都是合法的，但其行为却不一定与你想像的一致。对于boolean基本类型或者Boolean对象,其转换结果是对应的true或false.但是,对其它类型的参数,只要该参数不为null,那转换的结果就永远都是true。即使是数字0，转换结果依然是true，而在其他语言中往往转换为false。所以将b应用于非布尔类型对象时请格外小心.

#### 7.3.6.2.6 十六进制转换小工具

在研究Class文件结构时,我们经常会查看它的16进制文件,那么我们可不可以通过String.format()将字节数组以16进制打印出来:

|  |
| --- |
| **package** cn.string.three;  **import** java.io.File;  **import** java.io.IOException;  **import** net.mindview.util.BinaryFile;  **public** **class** Hex {  **public** **static** String format(**byte**[] data){  StringBuilder sb = **new** StringBuilder();  **int** n = 0;  **for** (**byte** b : data) {  **if**(n%16 == 0){  sb.append(String.*format*("%05X: ", n));  }  sb.append(String.*format*("%02X", b));  n++;  **if**(n % 16 == 0){  sb.append("\n");  }  }  sb.append("\n");  **return** sb.toString();  }  **public** **static** **void** main(String[] args) **throws** IOException {  **if**(args.length == 0){  System.*out*.println(*format*(BinaryFile.*read*("E:/WorkSpace/JavaSE/Chapter 6 Comment Object/String/bin/cn/string/three/Hex.class")));  }**else** {  System.*out*.println(*format*(BinaryFile.*read*(**new** File(args[0]))));  }  }  } |



# 7.4 String与Question

## Question 20171113 Java中char为什么占用两个字节?

Java是使用Unicode编码的所以Java会将源代码以Unicode的方式编码成字节码文件,而Unicode编码规定任意一个字符占用的都是两字节,这也是为什么Java中char类型占用两字节的原因了.

## Question 20171113 String的长度有限制吗?方法,类名,字段名称有限制吗?

我们在学习Java的时候很多时候会接触到容器的概念,那么String呢?其实我们可以将String也看作一个容器,既然是容器那么有限制吗?

我们知道String的底层是char[]那么我们实际上只要知道数组的长度限制即可,我们知道数组的长度为arr.length,返回值是int值,那么数组长度原则上最大值就是int的最大值了,但实际上的最大值,是要取决于JVM的内存的,下面我们就通过代码来实验一下:

|  |
| --- |
| /\*\*  \* 字符串的长度  \*/  @Test  **public** **void** fun1(){  //当我们申请数组长度超出int最大值会出现异常,数组下标超出  // char[] ch1 = new char[Integer.MAX\_VALUE+1];  // System.out.println(ch1.length);  //提示  // char[] ch2 = new char[Integer.MAX\_VALUE];  // System.out.println(ch2.length);  **char**[] ch3 = **new** **char**[268435456];  System.*out*.println(ch3.length);  } |

第一个会报索引溢出异常



第二个会爆出内存溢出异常

java.lang.OutOfMemoryError: Requested array size exceeds VM limit

### 方法,类名,字段名称限制问题

了解Class文件的同志大概会知道,Class文件中方法和字段等都需要引用CONSTANT\_Utf8\_info型常量来描述名称,所以CONSTANT\_Utf8\_info型常量的最大长度也就是Java中方法,字段名称的最大长度,而这里的最大长度就是length的最大值,即u2类型所能表达的最大值65535。所以Java程序中如果定义了超过64KB英文字符的变量或方法名，将会无法编译。

## Question 20171119 String和CharSequence的区别与联系?

String 继承于CharSequence，也就是说String也是CharSequence类型。CharSequence是一个接口，它只包括length(), charAt(int index), subSequence(int start, int end)这几个API接口。除了String实现了CharSequence之外，StringBuffer和StringBuilder也实现了 CharSequence接口。需要说明的是，CharSequence就是字符序列，String, StringBuilder和StringBuffer本质上都是通过字符数组实现的！CharSequence 是 char 值的一个可读序列。此接口对许多不同种类的 char 序列提供统一的只读访问（个人理解应该算得上是char类型得一个集合，是interface）

String 类代表字符串。Java 程序中的所有字符串字面值（如 "abc" ）都作为此类的实例来实现。CharSequence是String实现的一个接口，相当简单，就是要求是一串字符。所以每个参数类型是CharSequence的方法，都可以实际代入String对象。为什么这些方法不干脆定义String作为参数类型？因为还有其他的CharSequence类型的类，比如StringBuffer和StringBuilder这两个很重要的类。String对象是不可变的，这两个可变，所以我们在构造字符串的过程中往往要用到StringBuffer和StringBuilder。如果那些方法定义String作为参数类型，那么就没法对它们用那些方法，先得转化成String才能用。但StringBuffer和StringBuilder转换为String再转换过来很化时间的，用它们而不是直接用String的“加法”来构造新String本来就是为了省时间，所以如果用String作为参数类型就杯具了。

# 7.5 正则表达式

提到正则表达式可能很多人都会头疼,的确正则表达式看起来毫无规律可循,而且伴随着大量的符号,枯燥无味,但是正则表达式却是我们操作字符串不可逾越的一个知识点,我们下面就来认识一下正则。

## 7.5.1 正则的用途

在程序设计过程中，经常需要对输入的数据格式进行检验（最简单的例子，校验邮箱格式是否合格，这个可以放在JavaScript中，不用单独请求），这时就会用到正则表达式，匹配正则表达式则数据格式正确，否则格式错误。

## 7.5.2 判断是否符合正则表达式的方法

为了检查输入的数据是否满足某种格式，从JDK1.4开始可以使用String类的matches（）方法进行判断。语法格式如下：

boolean matches(Strign regex)

* regex:指定的正则表达式
* 返回值:返回boolean类型

该方法用于告知当前字符串是否匹配参数regex指定的正则表达式。返回值是boolean类型，如果当前字符串与正则表达式匹配，则该方法返回true，否则返回false。

## 7.5.3 正则表达式的元字符

正则表达式是由一些含有特殊意义的字符组成的字符串，这些含有特殊意义的字符称为元字符，下表列出了正则表达式的部分元字符。



注意：在正则表达式中，“.”代表任何一个字符,因此在正则表达式中如果想使用普通意义的点字符”.”,必须使用转义字符”\”.

【例7.5.3.1】创建Decision类,判断指定字符串是否合法,要求字符串由7个字符组成,并且第一位必须是大写字母,2~4位必须是小写字母,后3位必须是数字字符,并在控制台输出结果.

|  |
| --- |
| /\*\*  \* 判断指定字符串是否合法,要求字符串由7个字符组成,并且第一位必须是大写字母,2~4位必须是小写字母,  \* 后3位必须是数字字符  \*/  @Test  **public** **void** fun1(){  String str1 = "Mrkj007";  String str2 = "mrkj007";  String str3 = "Mikr0078";  String regex = "\\p{Upper}\\p{Lower}\\p{Lower}\\p{Lower}\\d\\d\\d";  System.*out*.println("str1是否满足要求:"+str1.matches(regex));  System.*out*.println("str2是否满足要求:"+str2.matches(regex));  System.*out*.println("str3是否满足要求:"+str3.matches(regex));  } |



结果如上图所示,要求首字母为大写的英文字母,所以在regex中我们用\\p{Upper}表示一个大写的英文字符,要求2~4位是小写的英文字符,所以我们用\\p{Lower}表示,要求后三位是数字,所以我们用\\d来表示,要求七位,所以我们的元字符只能判定7位.字符串str1是符合规定的,str2首字母不符合规定,str3长度不符合规定.

## 7.5.4 正则表达式的限定符

在使用正则表达式时，如果需要某一类型的元字符多次输出，逐个输入就相当麻烦，这时可以使用正则表达式的限定元字符来重复次数。下表列出了常用限定符及其含义。



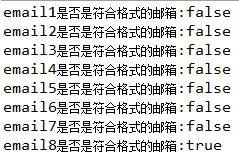
像例7.5.3.1中的语句 String regex = “\\p{Upper}\\p{Lower}\\p{Lower}\\p{Lower}\\d\\d\\d”;可以改写为:

String regex = "\\p{Upper}\\p{Lower}{3}\\d{3}";

运行发现结果是一致的,但是写起来更简单,阅读也方便,尤其在于重复次数多的时候简洁许多.

【例7.5.4.1】在Decision类中，在测试方法fun2中创建String对象，实现使用正则表达式来判断指定的变量是否是合法的Email地址。

|  |
| --- |
| /\*\*  \* 判断给定的字符串是否是Email格式  \* 分析:常见Email格式xxxx@xxxx.xxxx,这里我们简单比较,实际比较取决于JavaWeb中我们对于邮箱的规定  \* 也就是开始以任何数量的任意字符除了$,然后是@符,然后是3到4位的英文小写字符,然后是.然后是3到4位的英文小写字符  \*/  @Test  **public** **void** fun2(){  String regex = "\\w{0,}\\@\\p{Lower}{3,4}\\.{1}\\p{Lower}{3,4}";  String email1 = "doundo$@yeah.net";  String email2 = "dnon@163..com";  String email3 = "dsonscoasn123.com";  String email4 = "scnajnc@yeah.asdfg";  String email5 = "youjob@yeah .net";  String email6 = "youjob@ye&ah.net";  String email7 = "youjob@yeah.ne&t";  String email8 = "youjob@yeah.net";  System.*out*.println("email1是否是符合格式的邮箱:"+email1.matches(regex));  System.*out*.println("email2是否是符合格式的邮箱:"+email2.matches(regex));  System.*out*.println("email3是否是符合格式的邮箱:"+email3.matches(regex));  System.*out*.println("email4是否是符合格式的邮箱:"+email4.matches(regex));  System.*out*.println("email5是否是符合格式的邮箱:"+email5.matches(regex));  System.*out*.println("email6是否是符合格式的邮箱:"+email6.matches(regex));  System.*out*.println("email7是否是符合格式的邮箱:"+email7.matches(regex));  System.*out*.println("email8是否是符合格式的邮箱:"+email8.matches(regex));  } |



结果如上图所示，由于@之前我们要求是任意数量的任意字符除了$,所以格式为//w{0,},然后跟随一个@然后是3到4位的小写英文字符,所以是//p{Lower}{3,4},然后是一个.,注意.我们要转义,然后是3到4位的小写英文字符.茶看结果和代码对比,我们发现只有最后一个符合规定.

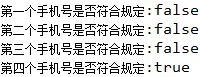
## 7.5.5 方括号中元字符的含义

在正则表达式中还可以用方括号把多个字符括起来，方括号中各种正则表达式代表不同的含义。下表列出了方括号中元字符及其含义。



【例7.5.5.1】在类Decision中，创建测试方法fun3，在测试方法中创建String对象，使用正则表达式对手机号的合法性进行验证，在使用正则表达式时需要用到一些元字符和方括号中单独元字符。

|  |
| --- |
| /\*\*  \* 使用正则验证手机号:  \* 首字母是1  \* `后面紧跟的字符为3,或5,或8  \* 后面紧跟9位数字  \*/  @Test  **public** **void** fun3(){  String regex = "1[3,5,8]\\d{9}$";//$是结尾匹配符,代表d{9}后匹配结束  String num1 = "1378988901";//长度不够  String num2 = "14123456789";//第二部分不符合规定  String num3 = "23123456789";//首字符不符合  String num4 = "15123456789";//符合规定  System.*out*.println("第一个手机号是否符合规定:"+num1.matches(regex));  System.*out*.println("第二个手机号是否符合规定:"+num2.matches(regex));  System.*out*.println("第三个手机号是否符合规定:"+num3.matches(regex));  System.*out*.println("第四个手机号是否符合规定:"+num4.matches(regex));  } |



【例7.5.5.2】IP地址是网络上每台计算机的标识,在浏览器中输入的网址也要经过DNS服务器转换为IP地址才能找到服务器.对IP地址的验证要比对电话号码的验证复杂一些.因为IP地址的各数字之间使用”.”号进行连接,并且3种元字符都会用到.

在我们代码实现之前,先来看看现在IP的格式

iPv4的ip地址都是（1~255）.（0~255）.（0~255）.（0~255）的格式

那么IP分为四块:

第一块:个位数[1-9]

两位数[1-9]\\d

三位数1\\d{2} 2[0-4]\\d 25[0-5]

第二块:个位数\\d

两个数[1-9]\\d

三个数1\\d{2} 2[0-4]\\d 25[0-5]

第三块和第四块定义格式和第二块相同;

有了格式我们组合正则表达式如下:注意不要少了.

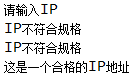
String regex = “(^[1-9]|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])\\.”+

”(\\d|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])\\.”+

”(\\d|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])\\.”+”(\\d|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])”;

在类Decision中,创建测试方法fun4,判断输入的IP地址每一个字符是否匹配IP地址的正则表达式,每个字符都匹配方是正确的IP地址.

|  |
| --- |
| /\*\*  \* 匹配IP地址  \*/  @Test  **public** **void** fun4(){  String ip = "01.192.168.98";  String ip2 = "192.1234.123.67";  String ip3 = "192.168.1.34";  System.*out*.println(test(""));  System.*out*.println(test(ip));  System.*out*.println(test(ip2));  System.*out*.println(test(ip3));  }  **public** String test(String ip){  //判断IP不为空  **if**(ip != **null** && !ip.isEmpty()){  String regex = "(^[1-9]|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])\\."+  "(\\d|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])\\."+  "(\\d|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])\\."+  "(\\d|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])";  **if** (ip.matches(regex)) {  **return** "这是一个合格的IP地址";  }**else** {  **return** "IP不符合规格";  }  }  **return** "请输入IP";  } |



## 7.5.6 边界匹配符

在前面我们看到了两个字符”^”和”$”,这两个字符一个表示行的开始,一个表示行的结尾,这是边界匹配符的两个最基本的字符,边界匹配符,顾名思义用于匹配边界的.边界匹配符有如下几种:



# 7.6. String的练习

## 7.6.1 登录小练习

|  |
| --- |
| **package** cn.string.practice;  **import** java.util.Scanner;  **public** **class** LoginPractice {  **public** **static** **void** main(String[] args) {  *login*();  }  //登录小练习  **public** **static** **void** login(){  //循环控制登录机会  **for** (**int** i = 0; i < 2; i++) {  //已知的用户名和密码,用于对比  String user = "admin";  String pass = "admin";  //声明一个Scanner类,获取键盘输入  Scanner scanner = **new** Scanner(System.*in*);  System.*out*.println("请输入用户名:");  String inputUsername = scanner.nextLine();  System.*out*.println("请输入密码:");  String inputPassword = scanner.nextLine();  **if**(inputUsername.equals(user)&&inputPassword.equals(pass)){  System.*out*.println("恭喜你登录成功!");  }**else** {  **if**(i == 0){  System.*out*.println("请检查你的用户名和密码,重新输入!");  }**else** {  System.*out*.println("对不起两次登录机会已用完!");  }  }    }  }  } |

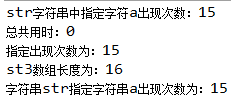


## 7.6.2 获取字符串中指定字符串出现次数

在7.2.3.6中我们已经初步学习了字符串的分割查找了,那么在这里我们再深入了解一下,有没有除了split以外的方式.

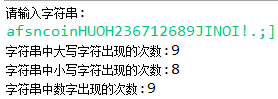
字符串中字符的出现次数:

|  |
| --- |
| **package** cn.string.practice;  /\*\*  \* 如果是获取字符串中的字符串,那么我们就只能截取字符串了  \* 但是如果要求的是字符的话,那么我们就有很多发挥空间了,我们可以首先将该字符串转变为字符数组,  \* 方式1:循环遍历比较  \* 方式2:替换,将字符替换为"",然后比较先后长度  \* 方式3:split截取  \* 注意我们看不出时间上的变化是因为我们给定的字符串太小，小差别忽略到我们看不到  \*/  **public** **class** GetCharCount {  **public** **static** **void** main(String[] args) {  String str = "cnonabciojiabcnonabciocnonabciojiabcnonabciocnonabciojiabcnonabciocnonabciojiabcnonabciocnonabciojiabcnonabcio";  **char** c = 'a';  **int** num = 0;  //计时开始  **long** startTime = System.*currentTimeMillis*();  **char**[] ch = str.toCharArray();  **for**(**int** i = 0;i<ch.length;i++){  **if**(ch[i]==c){  num++;  }  }  **long** endTime = System.*currentTimeMillis*();  System.*out*.println("str字符串中指定字符a出现次数："+num);  **long** time = endTime-startTime;  System.*out*.println("总共用时："+time);  //第二种，  String strTest = str.replaceAll("a", "");  //出现次数  **int** num1 = str.length()-strTest.length();  System.*out*.println("指定出现次数为："+num1);  //第三种  //通过字符串str2来验证  String[] st3 = str.split("a");  System.*out*.println("st3数组长度为："+st3.length);  **if**(str.startsWith("a")&&str.endsWith("a")){  System.*out*.println("字符串str指定字符串a出现次数为："+(st3.length+1));  }**else** **if** (str.startsWith("a")|str.endsWith("a")) {  System.*out*.println("字符串str指定字符串a出现次数为："+st3.length);  }**else** {  System.*out*.println("字符串str指定字符串a出现次数为："+(st3.length-1));  }  }  } |



## 7.6.3 统计字符大小写和数字的个数

|  |
| --- |
| **package** cn.string.practice;  **import** java.util.Scanner;  **public** **class** GetCharNumber {  **public** **static** **void** main(String[] args) {  *getNum*();  }  /\*\*  \* 获取字符串中大写,小写,数字的个数  \* 计算字符串中大写字母个数，小写字母个数，数字个数  \* 分析：首先我们要有一个字符串，其次，我们要分析，怎么判断一个字符他是不是字母，怎么判断他是不是大小写  \* 我们知道字符串是Unicode编码，那么字母其实对应的是数字，所以我们只有判断他的区间，就可以判断大小写  \*/  **public** **static** **void** getNum(){  Scanner sc = **new** Scanner(System.*in*);  System.*out*.println("请输入字符串:");  String str = sc.nextLine();  //声明三个int类型变量来作为大写字母个数 小写字母个数 非字母个数  **int** bigCount = 0;  **int** smallCount = 0;  **int** numCount = 0;  //将字符串转变为字符数组,然后循环判断  **char**[] ch = str.toCharArray();  **for** (**int** i = 0; i < ch.length; i++) {  **if**(ch[i] >= 'A' && ch[i] <= 'Z'){  bigCount++;  }**else** **if** (ch[i] >= 'a' && ch[i] <= 'z') {  smallCount++;  }**else** **if**(ch[i] >= '0' && ch[i] <= '9'){  numCount++;  }  }  System.*out*.println("字符串中大写字符出现的次数:"+bigCount);  System.*out*.println("字符串中小写字符出现的次数:"+smallCount);  System.*out*.println("字符串中数字出现的次数:"+numCount);  }  } |



## 7.6.4 将字符串首字母大写,其余小写

|  |
| --- |
| **package** cn.string.practice;  **import** java.util.Scanner;  **public** **class** CharChange {  **public** **static** **void** main(String[] args) {  *change*();  }  //将字符串首字母大写,后面的字母小写  **public** **static** **void** change(){  Scanner sc = **new** Scanner(System.*in*);  System.*out*.println("请输入字符串:");  String str = sc.nextLine();  //获取第一个字符  String str2 = str.substring(0, 1);  //获取第一个字符后面的字符  String str3 = str.substring(1);  **char**[] ch = str2.toCharArray();  //判断首字符是字母,然后转换  **if**((ch[0] >= 'A' && ch[0] <= 'Z') || (ch[0] >= 'a' && ch[0] <= 'z')){  str2 = String.*valueOf*(ch).toUpperCase();  str3 = str3.toLowerCase();  str = str2 + str3;  System.*out*.println(str);  }**else** {  System.*out*.println("请检查您的输入首字符是不是字母!");  }  }  } |

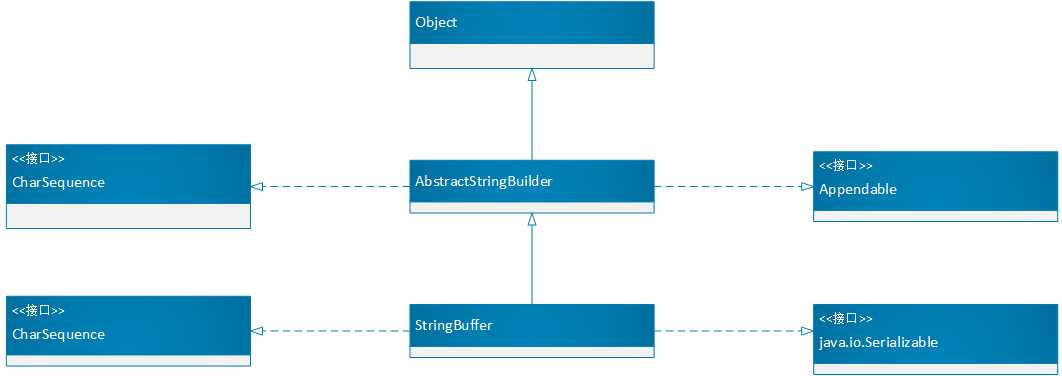


我们学习了Java常见对象String后,我们知道String是不可改变的,那么这会不会造成内存的压力呢?有没有可以修改的字符串呢?实际上CharSequence接口提供了两个实现类StringBuffer和StringBuilder来做这项任务,下面我们来深入学习一下StringBuffer和StringBuilder,并且对比这两个类.

# 7.7 StringBuffer

## 7.7.1 StringBuffer概述

StringBuffer，由名字可以看出，是一个String的缓冲区，也就是说一个类似于String的字符串缓冲区，和String不同的是，它可以被修改，而且是线程安全的。StringBuffer在任意时刻都有一个特定的字符串序列，不过这个序列和它的长度可以通过一些函数调用进行修改。它的结构层次如下图：



StringBuffer是线程安全的，因此如果有几个线程同时操作StringBuffer，对它来说也只是一个操作序列，所有操作串行发生。每一个StringBuffer都有一个容量，如果内容的大小不超过容量，StringBuffer就不会分配更大容量的缓冲区；如果需要更大的容量，StringBuffer会自动增加容量。和StringBuffer类似的有StringBuilder，两者之间的操作相同，不过StringBuilder不是线程安全的。虽然如此，由于StringBuilder没有同步，所以它的速度更快一些。

当发生与源序列有关的操作（如源序列中的追加或插入操作）时，该类只在执行此操作的字符串缓冲区上而不是在源上实现同步。

每个字符串缓冲区都有一定的容量。只要字符串缓冲区所包含的字符序列的长度没有超出此容量，就无需分配新的内部缓冲区数组。如果内部缓冲区溢出，则此容量自动增大。从 JDK 5 开始，为该类补充了一个单个线程使用的等价类，即 StringBuilder。与该类相比，通常应该优先使用 StringBuilder 类，因为它支持所有相同的操作，但由于它不执行同步，所以速度更快。

## 7.7.2 StringBuffer的初始容量和扩容机制

### 7.7.2.1 StringBuffer的初始容量

既然是容器,那么是一定会有个初始容量的,目的在于避免在内存中过度占用内存.容器的初始容量有默认和使用构造函数申明两种.

查看API我们知道StringBuffer有以下几种构造方法:

|  |
| --- |
| **[StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "StringBuffer())**()            构造一个其中不带字符的字符串缓冲区，其初始容量为 16 个字符。 |
| **[StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "StringBuffer(java.lang.CharSequence))**([CharSequence](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) seq)            public java.lang.StringBuilder(CharSequence seq) 构造一个字符串缓冲区，它包含与指定的 CharSequence 相同的字符。 |
| **[StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "StringBuffer(int))**(int capacity)            构造一个不带字符，但具有指定初始容量的字符串缓冲区。 |
| **[StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "StringBuffer(java.lang.String))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) str)            构造一个字符串缓冲区，并将其内容初始化为指定的字符串内容。 |

知道了构造方法,我们来了解一下StringBuffer的容量的底层原理;

#### 7.7.2.1.1 不声明长度,默认分配16

查看StringBuffer的空参构造,发现其容量受父类AbstractStringBuilder控制

|  |
| --- |
| **public** StringBuffer() {  **super**(16);  } |
| AbstractStringBuilder(**int** capacity) {  value = **new** **char**[capacity];  } |

查看上面代码发现默认容量实际上就是创建了一个长度16的字符数组;

#### 7.7.2.1.2 声明长度

查看StringBuffer的有参构造

|  |
| --- |
| **public** StringBuffer(**int** capacity) {  **super**(capacity);  } |
| **public** StringBuffer(String str) {  **super**(str.length() + 16);  append(str);  } |
| **public** StringBuffer(CharSequence seq) {  **this**(seq.length() + 16);  append(seq);  } |
| AbstractStringBuilder(**int** capacity) {  value = **new** **char**[capacity];  } |

我们发现有参构造有三个,直接给予长度的很好理解,底层是根据这个长度来创建了一个字符数组,而使用字符串的创建呢,底层的数组长度是字符串长度+16,如果是用字符序列那么和字符串是一样的,然后执行append操作.

### 7.7.2.2 StringBuffer的原理

StringBuffer继承了抽象类AbstractStringBuilder，在AbstractStringBuilder类中，有两个字段分别是char[]类型的value和int类型的count，也就是说，StringBuffer本质上是一个字符数组：

|  |
| --- |
| **char**[] value;  **int** count; |

value用来存储字符，而count表示数组中已有内容的大小，也就是长度。StringBuffer的主要操作有append、insert等，这些操作都是在value上进行的，而不是像String一样每次操作都要new一个String，因此，StringBuffer在效率上要高于String。有了append、insert等操作，value的大小就会改变，那么StringBuffer是如何操作容量的改变的呢？ 我们发现最后所有的方法,其实都是在AbstractStringBuilder类中的,以前总结过一次是以JDK1.7位蓝本总结的,这次以JDK1.8为蓝本总结,发现有不同,以此次为准;

|  |
| --- |
| //执行插入操作offset - 偏移量。 str - 一个 string。  **public** AbstractStringBuilder insert(**int** offset, String str) {  //offset指的是我们 要将字符串插入到的原字符串的位置,所以要注意索引越界异常  **if** ((offset < 0) || (offset > length()))  **throw** **new** StringIndexOutOfBoundsException(offset);  **if** (str == **null**)//注意这一点,如果插入的字符串是空字符串null.那么这里不会有nullPointException,而是掺入一个字符串”null”  str = "null";  **int** len = str.length();//获取要插入的字符串的长度  ensureCapacityInternal(count + len);**//注意了:这是扩容扩容机制**  //**[arraycopy](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/System.html" \l "arraycopy(java.lang.Object, int, java.lang.Object, int, int))**([Object](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) src, int srcPos, [Object](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) dest, int destPos, int length)            从指定源数组中复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。  public static void arraycopy([Object](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) src,  int srcPos,  [Object](mk:@MSITStore:D:\\Java-API\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) dest,  int destPos,  int length)  从 src 引用的源数组到 dest 引用的目标数组，数组组件的一个子序列被复制下来。被复制的组件的编号等于 length 参数。源数组中位置在 srcPos 到 srcPos+length-1 之间的组件被分别复制到目标数组中的 destPos 到 destPos+length-1 位置。  System.*arraycopy*(value, offset, value, offset + len, count - offset);  str.getChars(value, offset);  count += len;  **return** **this**;  } |
| **//测试null字符串的插入**  **public** **static** **void** main(String[] args) {  StringBuffer sb = **new** StringBuffer();  sb.append("西风多少恨,吹不散眉弯!");  String str = **null**;  sb.insert(2, str);  System.***out***.println(sb);//西风null多少恨,吹不散眉弯!  } |
| **//添加**  **public** AbstractStringBuilder append(String str) {  **if** (str == **null**)//假如添加的是null字符串  **return** appendNull();//添加字符串”null”  **int** len = str.length();  ensureCapacityInternal(count + len);//扩容  str.getChars(0, len, value, count);//复制数组到指定数组,这就是为什么这些字符串是可变的,是通过扩容机制重新创建了数组,然后将原来的数组指向新数组,旧数组会被回收  count += len;  **return** **this**;  } |
| **private** AbstractStringBuilder appendNull() {  **int** c = count;  ensureCapacityInternal(c + 4);  **final** **char**[] value = **this**.value;  value[c++] = 'n';  value[c++] = 'u';  value[c++] = 'l';  value[c++] = 'l';  count = c;  **return** **this**;  } |
| **/\*\***StringBuffer和StringBuilder除了通过插入和添加字符串动态扩容外,还可以通过动态设置容量的方法,来扩容,源码如下,我们观察和上面其实是一致的,这里不再分析:  **\*/**  **public** **void** setLength(**int** newLength) {  **if** (newLength < 0)  **throw** **new** StringIndexOutOfBoundsException(newLength);  ensureCapacityInternal(newLength);  **if** (count < newLength) {  Arrays.*fill*(value, count, newLength, '\0');  }  count = newLength;  } |
| **//重头戏,扩容机制**  **//**minimumCapacity = count +length (setLength的minimumCapacity = 给定的length)是原字符串长度+插入或添加的字符串长度,注意是字符串长度不是字符数组长度  **private** **void** ensureCapacityInternal(**int** minimumCapacity) {  // overflow-conscious code  **if** (minimumCapacity - value.length > 0) {//假如我们的参数长度大于未扩容前的字符数组长度,那么复制原字符数组,到一个指定长度newCapacity(minimumCapacity)的新字符数组  value = Arrays.*copyOf*(value,  newCapacity(minimumCapacity));  }  } |
| //对于字符串的长度,是返回的字符数组中有效的字符个数,而非字符长度  @Override  **public** **synchronized** **int** length() {  **return** count;  } |
| **private** **static** **final** **int** ***MAX\_ARRAY\_SIZE*** = Integer.***MAX\_VALUE*** - 8; |
| **//这是扩容中的新字符数组创建,扩容的本质是创建一个新数组,然后将原来的数据添加到新数组,这里是确定新数组的长度**  **private** **int** newCapacity(**int** minCapacity) {  // overflow-conscious code  **int** newCapacity = (value.length << 1) + 2;//长度=原长度\*2+2  **if** (newCapacity - minCapacity < 0) {//假如新长度比原有字符串长度和新加字符串长度小,那么以后者为准  newCapacity = minCapacity;  }  //这里是要返回的新字符数组的长度,如果长度小于等于0,或者newCapacity比***MAX\_ARRAY\_SIZE大,那么返回***hugeCapacity(minCapacity),否则返回长度newCapacity(一般返回后者,返回前者的都”不寻常”)  **return** (newCapacity <= 0 || ***MAX\_ARRAY\_SIZE*** - newCapacity < 0)  ? hugeCapacity(minCapacity)  : newCapacity;  } |
| **private** **int** hugeCapacity(**int** minCapacity) {  //如果原字符串和新字符串长度超过String的最大值,那么就内存溢出异常  **if** (Integer.***MAX\_VALUE*** - minCapacity < 0) { // overflow  **throw** **new** OutOfMemoryError();  }  //否则根据原字符串与新字符串的长度与 ***MAX\_ARRAY\_SIZE的判定决定返回的值***  **return** (minCapacity > ***MAX\_ARRAY\_SIZE***)  ? minCapacity : ***MAX\_ARRAY\_SIZE***;  } |

总结:StringBuffer的扩容实际上就是新建了一个数组,将原来旧数组的内容复制到新数组,扩容机制根据当前数组长度的2倍+2和新增加字符串长度+原有数组长度进行比较,如果前者小于后者,那么扩容后的长度就是后者,如果前者大于后者那么扩容后的数组长度就是前者,每次append或者insert会再次进行比较.

## 7.7.3 StringBuffer方法

了解了StringBuffer的扩容,下面我们来了解一下StringBuffer的常见方法.

### 7.7.3.1 追加字符串

StringBuffer中提供了append()方法来向缓存字符串中追加字符串,append添加的扩容,我们可以查看6.1.2中的扩容,下面我们来认识一下:

|  |  |
| --- | --- |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(boolean))**(boolean b)            将 boolean 参数的字符串表示形式追加到序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(char))**(char c)            将 char 参数的字符串表示形式追加到此序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(char[]))**(char[] str)            将 char 数组参数的字符串表示形式追加到此序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(char[], int, int))**(char[] str, int offset, int len) offset开始索引,len追加的字符长度           将 char 数组参数的子数组的字符串表示形式追加到此序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(java.lang.CharSequence))**([CharSequence](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) s)            将指定的 CharSequence 追加到该序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(java.lang.CharSequence, int, int))**([CharSequence](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) s, int start, int end) start开始索引,end结束索引,要注意索引越界异常,如果头尾相等,则不追加,如果开始索引大于结束索引,角标越界异常           将指定 CharSequence 的子序列追加到此序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(double))**(double d)            将 double 参数的字符串表示形式追加到此序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(float))**(float f)            将 float 参数的字符串表示形式追加到此序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(int))**(int i)            将 int 参数的字符串表示形式追加到此序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(long))**(long lng)            将 long 参数的字符串表示形式追加到此序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(java.lang.Object))**([Object](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) obj) Object先调用toString,然后将字符串追加结尾           追加 Object 参数的字符串表示形式。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(java.lang.String))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) str)            将指定的字符串追加到此字符序列。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "append(java.lang.StringBuffer))**([StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) sb)            将指定的 StringBuffer 追加到此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[appendCodePoint](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "appendCodePoint(int))**(int codePoint)            将 codePoint 参数的字符串表示形式追加到此序列。 |

offset - 要追加的第一个 char 的索引

len - 要追加的 char 的数量。

codePoint - 一个 Unicode 代码点

|  |
| --- |
| /\*\*  \* StringBuffer的缓存探讨  \*/  @Test  **public** **void** fun1(){  //给定缓存长度创建StringBuffer对象  StringBuffer sb = **new** StringBuffer(18);  String str1 = "相逢不语,";  **char**[] ch = {'一','朵','芙','蓉','著','秋','雨','。'};  **char** ch1 = '小';  StringBuilder sb1 = **new** StringBuilder("晕红潮");  **double** dou = 2.34;  **float** f = 2.3F;  **int** i = 90;  **long** l = 900;  **boolean** boo = **true**;  StringBufferDemo sbd = **new** StringBufferDemo();  System.*out*.println(sb.append(boo).toString());  System.*out*.println(sb.append(str1));  System.*out*.println(sb.append(ch));  System.*out*.println(sb.append(ch1));  System.*out*.println(sb.append(ch,2,4));//注意角标越界异常,开始索引和长度都会产生越界异常  System.*out*.println(sb.append(sb1));  System.*out*.println(sb.append(sb1,2,2));//顾头不顾尾,头尾相同则不追加  System.*out*.println(sb.append(sb1,1,2));  System.*out*.println(sb.append(dou));  System.*out*.println(sb.append(f));  System.*out*.println(sb.append(i));  System.*out*.println(sb.append(l));  System.*out*.println(sb.append(sbd));  System.*out*.println(sb.append(sb));  System.*out*.println(sb.appendCodePoint(i));  } |
| true  true相逢不语,  true相逢不语,一朵芙蓉著秋雨。  true相逢不语,一朵芙蓉著秋雨。小  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.34  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.3  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.390  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.390900  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.390900cn.sb.capacity.StringBufferDemo@18a79e  [true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.390900cn.sb.capacity.StringBufferDemo@18a79etrue相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.390900cn.sb.capacity.StringBufferDemo@18a79e](mailto:true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.390900cn.sb.capacity.StringBufferDemo@18a79etrue相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.390900cn.sb.capacity.StringBufferDemo@18a79e)  true相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.390900cn.sb.capacity.StringBufferDemo@18a79etrue相逢不语,一朵芙蓉著秋雨。小芙蓉著秋晕红潮红2.342.390900cn.sb.capacity.StringBufferDemo@18a79eZ |

### 7.7.3.2 与长度及容量有关的方法

StringBuffer是个缓存,所以就存在了长度和容量的问题,长度指的是字符数组中元素的个数,容量是指这个数组声明的长度,下面我们来看看

|  |  |
| --- | --- |
| int | **[capacity](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "capacity())**()            返回当前容量。 |
| int | **[length](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "length())**()            返回长度（字符数）。 |
| void | **[setLength](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "setLength(int))**(int newLength)            设置字符序列的长度。 |
| void | **[trimToSize](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "trimToSize())**()            尝试减少用于字符序列的存储空间。 |

|  |  |
| --- | --- |
| void | **[ensureCapacity](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "ensureCapacity(int))**(int minimumCapacity)            确保容量至少等于指定的最小值。 |

|  |
| --- |
| /\*\*  \* 长度和容量  \*/  @Test  **public** **void** fun2(){  StringBuffer sb = **new** StringBuffer(100);  String str ="相逢不语，一朵芙蓉著秋雨。小晕红潮，斜溜鬟心只凤翘。待将低唤，直为凝情恐人见。欲诉幽怀，转过回阑叩玉钗。";  sb.append(str);  System.*out*.println("字符串缓冲的容量是:"+sb.capacity());  System.*out*.println("字符串缓冲的长度是:"+sb.length());  //重新设置长度  sb.setLength(200);  System.*out*.println("容量:"+sb.capacity()+" 长度:"+sb.length());  //尝试减少字符序列空间,减少的就是那个2  sb.trimToSize();  System.*out*.println("容量:"+sb.capacity()+" 长度:"+sb.length());  sb.trimToSize();  System.*out*.println("容量:"+sb.capacity()+" 长度:"+sb.length());  sb.ensureCapacity(150);  System.*out*.println("容量:"+sb.capacity()+" 长度:"+sb.length());  sb.ensureCapacity(250);  System.*out*.println("容量:"+sb.capacity()+" 长度:"+sb.length());  sb.trimToSize();  System.***out***.println("容量:"+sb.capacity()+" 长度:"+sb.length());  } |
| **public** **void** trimToSize() {  **if** (count < value.length) {  value = Arrays.*copyOf*(value, count);  }  } |



发现容量和我们定义有的不同,这是由Stringbuffer和StringBuilder的扩容决定的.

#### 7.7.3.2.1 对比setlength()和ensureCapacity()

StringBuilder和StringBuffer中有一个方法ensureCapacity()是用于扩容的,研究过JDK1.7的扩容机制会发现,JDK1.7的扩容和JDK1.8是有很大不同的,在JDK1.8中ensureCapacity()有点鸡肋,没有取消大概是作为一个过渡吧,不过既然在,我们就来看一下setlength()和ensureCapacity()这两种的区别.

setlength()和ensureCapacity()都能动态扩大StringBuffer和StringBuilder的容量,那么区别在哪呢?我们把源码粘贴出来看一下

|  |
| --- |
| **public** **void** setLength(**int** newLength) {  **if** (newLength < 0)  **throw** **new** StringIndexOutOfBoundsException(newLength);  ensureCapacityInternal(newLength);  **if** (count < newLength) {  Arrays.*fill*(value, count, newLength, '\0');  }  count = newLength;  } |
| **public** **void** ensureCapacity(**int** minimumCapacity) {  **if** (minimumCapacity > 0)  ensureCapacityInternal(minimumCapacity);  } |

对比发现setLength会在新数组中复制newLength的字符个数,若旧数组元素个数不够,那就以空格代替;而ensureCapacity只是一个扩容,将原字符数组复制到新字符数组,至于多余的长度,不会有什么添加,我们通过一段代码来分析:

|  |
| --- |
| @Test  **public** **void** test(){  StringBuffer sb = **new** StringBuffer("我是谁!");  System.***out***.println("容量:"+sb.capacity()+" 长度:"+sb.length());  sb.setLength(100);  System.***out***.println("容量:"+sb.capacity()+" 长度:"+sb.length());  sb.ensureCapacity(500);  System.***out***.println("容量:"+sb.capacity()+" 长度:"+sb.length());  } |



我们发现的确是如此,setlength,在扩容的同时将字符数组中填充,是每个索引都有输出,而单纯的扩容,仅仅是实现了一次容量的增量罢了.

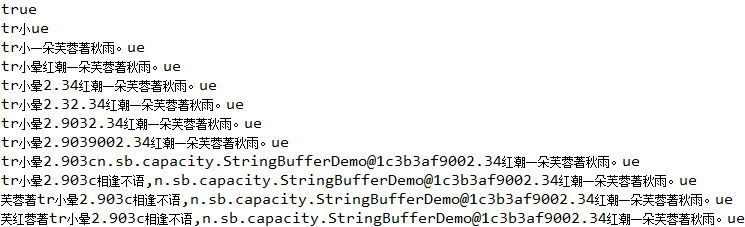
### 7.7.3.3 StringBuffer的插入操作

|  |  |
| --- | --- |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, boolean))**(int offset, boolean b) offset - 偏移量。           将 boolean 参数的字符串表示形式插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, char))**(int offset, char c)            将 char 参数的字符串表示形式插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, char[]))**(int offset, char[] str)            将 char 数组参数的字符串表示形式插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, char[], int, int))**(int index, char[] str, int offset, int len)            将数组参数 str 的子数组的字符串表示形式插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, java.lang.CharSequence))**(int dstOffset, [CharSequence](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) s)            将指定 CharSequence 插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, java.lang.CharSequence, int, int))**(int dstOffset, [CharSequence](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) s, int start, int end)            将指定 CharSequence 的子序列插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, double))**(int offset, double d)            将 double 参数的字符串表示形式插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, float))**(int offset, float f)            将 float 参数的字符串表示形式插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, int))**(int offset, int i)            将 int 参数的字符串表示形式插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, long))**(int offset, long l)            将 long 参数的字符串表示形式插入此序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, java.lang.Object))**(int offset, [Object](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/Object.html" \o "java.lang 中的类) obj)            将 Object 参数的字符串表示形式插入此字符序列中。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "insert(int, java.lang.String))**(int offset, [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) str)            将字符串插入此字符序列中。 |

Offset:开始索引,start开始索引,end:结束索引

插入操作也是字符串经常见到的一个操作,插入的意思就是在制定位置上插入新元素,原有元素向后推,每一次插入都有一次扩容,所以实际也是将元素复制到新数组的过程,下面我们来实地操作一下:

|  |
| --- |
| /\*\*  \* 插入操作  \*/  @Test  **public** **void** fun3(){  //给定缓存长度创建StringBuffer对象  StringBuffer sb = **new** StringBuffer(18);  String str1 = "相逢不语,";  **char**[] ch = {'一','朵','芙','蓉','著','秋','雨','。'};  **char** ch1 = '小';  StringBuilder sb1 = **new** StringBuilder("晕红潮");  **double** dou = 2.34;  **float** f = 2.3F;  **int** i = 90;  **long** l = 900;  **boolean** boo = **true**;  StringBufferDemo sbd = **new** StringBufferDemo();  System.***out***.println(sb.capacity()+"--"+sb.length()+"--"+sb.insert(0, boo).toString());//如果字符串缓冲没有元素,此时有容量没元素,那么开始索引只能是0,否则角标越界异常18--0--true  System.*out*.println(sb.insert(2, ch1).toString());  System.*out*.println(sb.insert(3, ch).toString());//String中getChars是覆盖,而这里是插入,将原有元素后退  System.*out*.println(sb.insert(3, sb1).toString());  System.*out*.println(sb.insert(4, dou).toString());  System.*out*.println(sb.insert(7, f).toString());  System.*out*.println(sb.insert(6, i).toString());  System.*out*.println(sb.insert(9, l).toString());  System.*out*.println(sb.insert(9, sbd).toString());  System.*out*.println(sb.insert(10, str1).toString());  System.*out*.println(sb.insert(0, ch, 2, 3).toString());//注意索引越界  System.*out*.println(sb.insert(1, sb1, 1, 2).toString());  } |



### 7.7.3.4 查询

|  |  |
| --- | --- |
| char | **[charAt](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "charAt(int))**(int index)            返回此序列中指定索引处的 char 值。 |
| int | **[codePointAt](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "codePointAt(int))**(int index)            返回指定索引处的字符（统一代码点）。 |
| int | **[codePointBefore](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "codePointBefore(int))**(int index)            返回指定索引前的字符（统一代码点）。 |
| int | **[codePointCount](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "codePointCount(int, int))**(int beginIndex, int endIndex)            返回此序列指定文本范围内的统一代码点。 |
| int | **[indexOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "indexOf(java.lang.String))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) str)            返回第一次出现的指定子字符串在该字符串中的索引。 |
| int | **[indexOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "indexOf(java.lang.String, int))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) str, int fromIndex)            从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。 |
| int | **[lastIndexOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "lastIndexOf(java.lang.String))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) str)            返回最右边出现的指定子字符串在此字符串中的索引。 |
| int | **[lastIndexOf](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "lastIndexOf(java.lang.String, int))**([String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) str, int fromIndex)            返回最后一次出现的指定子字符串在此字符串中的索引。 |

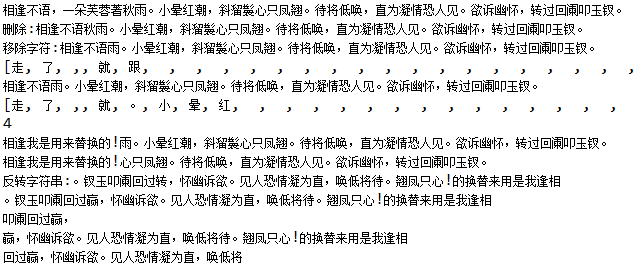
|  |
| --- |
| /\*\*  \* 查询操作  \*/  @Test  **public** **void** fun4(){  StringBuffer sb = **new** StringBuffer(100);  String str ="相逢不语，一朵芙蓉著秋雨。小晕红潮，斜溜鬟心只凤翘。待将低唤，直为凝情恐人见。欲诉幽怀，转过回阑叩玉钗。";  sb.append(str);  //返回指定索引处的char值  **char** ch = sb.charAt(7);  System.*out*.println(ch);  //返回指定索引处的字符（统一代码点）  System.*out*.println(sb.codePointAt(7));  //返回指定索引前的字符（统一代码点）  System.*out*.println(sb.codePointBefore(7));  //返回此序列指定文本范围内的统一代码点  System.*out*.println(sb.codePointCount(5, 10));  //返回第一次出现的指定子字符串在该字符串中的索引。  System.*out*.println(sb.indexOf("一")+" "+sb.indexOf("静"));//若查找字符不存在返回-1  //从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。  System.*out*.println(sb.indexOf("一", 10));  //返回最右边出现的指定子字符串在此字符串中的索引。  System.*out*.println(sb.lastIndexOf("。"));  System.*out*.println(sb.lastIndexOf("。", 15));//反向查找  } |



### 7.7.3.5 StringBuffer的操作

|  |  |
| --- | --- |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[delete](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "delete(int, int))**(int start, int end)            移除此序列的子字符串中的字符。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[deleteCharAt](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "deleteCharAt(int))**(int index)            移除此序列指定位置的 char。 |
| void | **[getChars](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "getChars(int, int, char[], int))**(int srcBegin, int srcEnd, char[] dst, int dstBegin)            将字符从此序列复制到目标字符数组 dst。 |
| int | **[offsetByCodePoints](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "offsetByCodePoints(int, int))**(int index, int codePointOffset)            返回此序列中的一个索引，该索引是从给定 index 偏移 codePointOffset 个代码点后得到的。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[replace](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "replace(int, int, java.lang.String))**(int start, int end, [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) str)            使用给定 String 中的字符替换此序列的子字符串中的字符。 |
| [StringBuffer](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \o "java.lang 中的类) | **[reverse](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "reverse())**()            将此字符序列用其反转形式取代。 |
| void | **[setCharAt](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "setCharAt(int, char))**(int index, char ch)            将给定索引处的字符设置为 ch。 |
| [CharSequence](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/CharSequence.html" \o "java.lang 中的接口) | **[subSequence](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "subSequence(int, int))**(int start, int end)            返回一个新的字符序列，该字符序列是此序列的子序列。 |
| [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[substring](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "substring(int))**(int start)            返回一个新的 String，它包含此字符序列当前所包含的字符子序列。 |
| [String](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/String.html" \o "java.lang 中的类) | **[substring](mk:@MSITStore:E:\\各种API\\J2SE6.0_CN.chm::/java/lang/../../java/lang/StringBuffer.html" \l "substring(int, int))**(int start, int end)            返回一个新的 String，它包含此序列当前所包含的字符子序列。 |

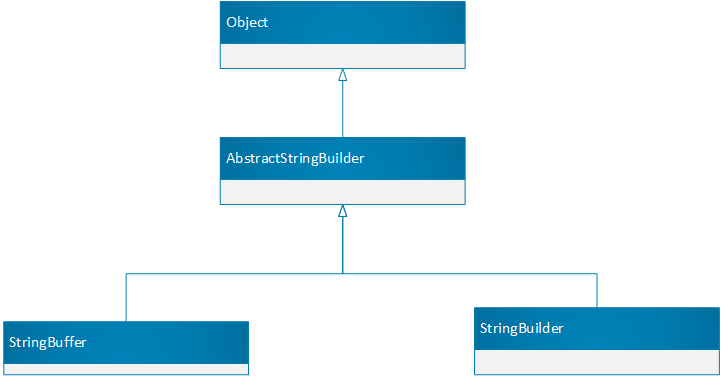
|  |
| --- |
| /\*\*  \* StringBuffer的操作  \*/  @Test  **public** **void** fun5(){  StringBuffer sb = **new** StringBuffer(100);  String str ="相逢不语，一朵芙蓉著秋雨。小晕红潮，斜溜鬟心只凤翘。待将低唤，直为凝情恐人见。欲诉幽怀，转过回阑叩玉钗。";  sb.append(str);  System.*out*.println(sb.toString());  sb.delete(4, 10);  System.*out*.println("删除:"+sb.toString());  sb.deleteCharAt(4);  System.*out*.println("移除字符:"+sb);  **char**[] ch = **new** **char**[50];  ch[0] = '走';  ch[1] = '了';  ch[2] = ',';  ch[3] = '就';  ch[4] = '跟';  ch[45] = ';';  System.*out*.println(Arrays.*toString*(ch));  //截取字符串到字符数组  sb.getChars(5, 9, ch, 4);//顾头不顾尾  System.*out*.println(sb.toString());  System.*out*.println(Arrays.*toString*(ch));  System.*out*.println(sb.offsetByCodePoints(2, 2));  String str1 = "我是用来替换的!";  System.*out*.println(sb.replace(2, 4, str1).toString());//会用一个字符串填充到给定的区间中,如果长度大于这个区间,那么元素相应后移  System.*out*.println(sb.replace(2, 20, str1).toString());//字符串小于区间,相应元素前移  System.*out*.println("反转字符串:"+sb.reverse().toString());  //将给定索引处的字符设置为 ch。  sb.setCharAt(7, '赢');  System.*out*.println(sb.toString());  // 返回一个新的字符序列，该字符序列是此序列的子序列  CharSequence cs = sb.subSequence(3, 9);  System.*out*.println(cs.toString());  str = sb.substring(7);  System.*out*.println(str);  str = sb.substring(5, 25);  System.*out*.println(str);  } |



# 7.8 StringBuilder

## 7.8.1 StringBuilder概述

StringBuilder和StringBuffer一样，都是继承自抽象类AbstractStringBuilder类，也是一个可变的字符序列。StringBuilder和StringBuffer非常相似，甚至有互相兼容的API，不过，StringBuilder不是线程安全的，这是和StringBuffer的主要区别。StringBuilder的层次结构如下：



StringBuilder类被设计成在不需要多线程安全的情况下替换StringBuffer类的。如果程序中不需要多线程安全，推荐使用StringBuilder类，因为没有了同步的操作而速度上比较快。StringBuilder和StringBuffer一样，主要的操作有append和insert等。

StringBuilder是JDK1.5新增加的功能,

创建成功的String对象,其长度是固定的,内容不能被改变和编译。虽然使用“+”可以达到附加新字符或字符串的目的，但“+”会产生一个新的String实例，会在内存中创建新的字符串对象。如果重复地对字符串进行修改，将极大的增加系统开销。而JavaSE 5.0新增了可变的字符序列StringBuilder类，大大提高了频繁增加字符串的效率。也许我们会问那JavaSE5.0之前呢？实际在JavaSE5.0之前有StringBuffer，但是StringBuffer是线程安全的，这就会造成效率的底下，所以JavaSE5.0新增了StringBuilder来弥补StringBuffer的不足。

## 7.8.2 StringBuilder的原理

StringBuilder的原理和StringBuffer一样，不同之处在于StringBuilder不需要考虑线程安全。我们拿一个扩容来看一下就知道了:

|  |
| --- |
| **public** **void** ensureCapacity(**int** minimumCapacity) {  **if** (minimumCapacity > 0)  ensureCapacityInternal(minimumCapacity);  } |

对比StringBuffer发现一个有关键字synchronized ,一个没有这就决定了一个线程安全,一个线程不安全,其余的都一样,参考StringBuffer,有兴趣可以去底层看看,

## 7.8.3 StringBuilder的方法

上面简单介绍了StringBuilder，下面我们说一下StringBuilder的方法。StringBuilder的方法实际上和StringBuffer只差一个关键字synchronized ，所以在这里我们不多做说明了，参见StringBuffer即可。

## 7.2.4 String&&StringBuffer&&StringBuilder

### Question 20171114 String&&StringBuffer&&StringBuilder的区别和联系

创建成功的String对象,其长度是固定的,内容不能被改变和编译。虽然使用“+”可以达到附加新字符或字符串的目的，但“+”会产生一个新的String实例，会在内存中创建新的字符串对象。如果重复地对字符串进行修改，将极大的增加系统开销。而JavaSE 5.0新增了可变的字符序列StringBuilder类，大大提高了频繁增加字符串的效率。也许我们会问那JavaSE5.0之前呢？实际在JavaSE5.0之前有StringBuffer，但是StringBuffer是线程安全的，这就会造成效率的底下，所以JavaSE5.0新增了StringBuilder来弥补StringBuffer的不足。

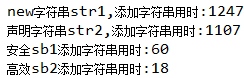
String ：不可改变

StringBuffer：可改变 线程安全

StringBuilder：可改变 线程不安全

验证连接字符串，String之+和StringBuffer的append谁的效率高

|  |
| --- |
| **package** cn.sb.capacity;  **public** **class** StringAndStringBuffer {  **public** **static** **void** main(String[] args) {  *fun1*();  // fun2();  // fun3();  // fun4();  }  /\*\*  \* new String  \*/  **public** **static** **void** fun1(){  String str1 = **new** String("");  **long** start = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 20000; i++) {  str1 += i;  }  **long** end = System.*currentTimeMillis*();  System.*out*.println("new字符串str1,添加字符串用时:"+(end-start));  }  /\*\*  \* 声明字符串  \*/  **public** **static** **void** fun2(){  String str2 = "";  **long** start = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 20000; i++) {  str2 += i;  }  **long** end = System.*currentTimeMillis*();  System.*out*.println("声明字符串str2,添加字符串用时:"+(end-start));  }  /\*\*  \* 安全字符串缓冲  \*/  **public** **static** **void** fun3(){  StringBuffer sb1 = **new** StringBuffer("");  **long** start = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 200000; i++) {  sb1.append(i);  }  **long** end = System.*currentTimeMillis*();  System.*out*.println("安全sb1添加字符串用时:"+(end-start));  }  /\*\*  \* 高效字符串缓冲  \*/  **public** **static** **void** fun4(){  StringBuilder sb2 = **new** StringBuilder("");  **long** start = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 200000; i++) {  sb2.append(i);  }  **long** end = System.*currentTimeMillis*();  System.*out*.println("高效sb2添加字符串用时:"+(end-start));  }  } |



|  |  |
| --- | --- |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [format](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "format-java.util.Locale-java.lang.String-java.lang.Object...-)([Locale](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/util/Locale.html" \o "class in java.util) l, [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) format, [Object](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/Object.html" \o "class in java.lang)... args)  使用指定的区域设置，格式字符串和参数返回格式化的字符串。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [format](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "format-java.lang.String-java.lang.Object...-)([String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) format, [Object](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/Object.html" \o "class in java.lang)... args)  使用指定的格式字符串和参数返回格式化的字符串。 |
| byte[] | [getBytes](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "getBytes--)()  使用平台的默认字符集将此 String编码为字节序列，将结果存储到新的字节数组中。 |
| byte[] | [getBytes](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "getBytes-java.nio.charset.Charset-)([Charset](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/nio/charset/Charset.html" \o "class in java.nio.charset) charset)  使用给定的[charset](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/nio/charset/Charset.html" \o "java.nio.charset中的类)将该String编码为字节序列，将结果存储到新的字节数组中。 |
| void | [getBytes](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "getBytes-int-int-byte:A-int-)(int srcBegin, int srcEnd, byte[] dst, int dstBegin)  已弃用  此方法无法将字符正确转换为字节。 从JDK 1.1开始，首选的方法是通过[getBytes()](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "getBytes--)方法，该方法使用平台的默认字符集。 |
| byte[] | [getBytes](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "getBytes-java.lang.String-)([String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) charsetName)  使用命名的字符集将此 String编码为字节序列，将结果存储到新的字节数组中。 |
| void | [getChars](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "getChars-int-int-char:A-int-)(int srcBegin, int srcEnd, char[] dst, int dstBegin)  将此字符串中的字符复制到目标字符数组中。 |
| int | [hashCode](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "hashCode--)()  返回此字符串的哈希码。 |
| char[] | [toCharArray](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "toCharArray--)()  将此字符串转换为新的字符数组。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [toLowerCase](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "toLowerCase--)()  将所有在此字符 String使用默认语言环境的规则，以小写。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [toLowerCase](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "toLowerCase-java.util.Locale-)([Locale](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/util/Locale.html" \o "class in java.util) locale)  将所有在此字符 String ，以降低使用给定的规则情况下 Locale 。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [toString](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "toString--)()  此对象（已经是字符串！）本身已被返回。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [toUpperCase](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "toUpperCase--)()  将所有在此字符 String使用默认语言环境的规则大写。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [toUpperCase](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "toUpperCase-java.util.Locale-)([Locale](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/util/Locale.html" \o "class in java.util) locale)  将所有在此字符 String使用给定的规则，大写 Locale 。 |
| [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [trim](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "trim--)()  返回一个字符串，其值为此字符串，并删除任何前导和尾随空格。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [valueOf](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "valueOf-boolean-)(boolean b)  返回 boolean参数的字符串 boolean形式。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [valueOf](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "valueOf-char-)(char c)  返回 char参数的字符串 char形式。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [valueOf](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "valueOf-char:A-)(char[] data)  返回 char数组参数的字符串 char形式。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [valueOf](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "valueOf-char:A-int-int-)(char[] data, int offset, int count)  返回 char数组参数的特定子阵列的字符串 char形式。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [valueOf](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "valueOf-double-)(double d)  返回 double参数的字符串 double形式。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [valueOf](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "valueOf-float-)(float f)  返回 float参数的字符串 float形式。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [valueOf](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "valueOf-int-)(int i)  返回 int参数的字符串 int形式。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [valueOf](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "valueOf-long-)(long l)  返回 long参数的字符串 long形式。 |
| static [String](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \o "class in java.lang) | [valueOf](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/String.html" \l "valueOf-java.lang.Object-)([Object](mk:@MSITStore:D:\\Java-API\\JDK%20API%201.8.CHM::/java/lang/../../java/lang/Object.html" \o "class in java.lang) obj)  返回 Object参数的字符串 Object形式。 |