public final class **String**

extends [Object](mk:@MSITStore:F:\my\mygit\enhancingProgram\JAVA\%5bJava参考文档%5dJDK_API_1_6_zh_CN.CHM::/java/lang/Object.html)

implements [Serializable](mk:@MSITStore:F:\my\mygit\enhancingProgram\JAVA\%5bJava参考文档%5dJDK_API_1_6_zh_CN.CHM::/java/io/Serializable.html), [Comparable](mk:@MSITStore:F:\my\mygit\enhancingProgram\JAVA\%5bJava参考文档%5dJDK_API_1_6_zh_CN.CHM::/java/lang/Comparable.html)<[String](mk:@MSITStore:F:\my\mygit\enhancingProgram\JAVA\%5bJava参考文档%5dJDK_API_1_6_zh_CN.CHM::/java/lang/String.html)>, [CharSequence](mk:@MSITStore:F:\my\mygit\enhancingProgram\JAVA\%5bJava参考文档%5dJDK_API_1_6_zh_CN.CHM::/java/lang/CharSequence.html)

1. final修饰的String类为最终类，不能被继承
2. String类实现了序列化接口，可比较接口和[CharSequence](mk:@MSITStore:F:\my\mygit\enhancingProgram\JAVA\%5bJava参考文档%5dJDK_API_1_6_zh_CN.CHM::/java/lang/CharSequence.html)接口
3. Java中所有的字符串字面值都是这个类的实例，如声明：

String str="abc" #则str就是一个string类的实例

上面的定义方法等效于：

char[] charArr={'a', 'b', 'c'}

String str2=new String(charArr)

1. 字符串是常量，他们的值在创建后不能修改
2. 字符串转换时通过toString方法实现的，该方法是由Object定义的，因此java中的所有类都可以继承并重写此方法

* 字段

|  |  |
| --- | --- |
| 字段类型 名定义 | 说明 |
| private final char value[]; | 真正存储字符串各个字符的类，字符串的底层实现为字符数组 |
| private int hash; |  |
| private static final long serialVersionUID = -6849794470754667710L; | 固定值：-6849794470754667710L |
| private static final ObjectStreamField[] serialPersistentFields = new ObjectStreamField[0]; |  |
| public static final Comparator<String> CASE\_INSENSITIVE\_ORDER |  |

方法、参数和说明以及示例

public String()

创建了一个新的空字符串对象

public String() {

this.value="".value();

}

public String(String original)

public String(String original){

this.value=original.value; #这些private变量可以在类的内部被访问

this.hash=original.hash;

}

#创建了一个新的对象，处理了String的核心value和hash

public String(char[] value)

将字符数组的值赋值给字符串的底层char[],由此可以看出字符串的底层实现是一个字符数组

public String(char value[]) {

this.value = Arrays.copyOf(value, value.length);

}

public String(char value[], int offset, int count)

拷贝字符数组的指定部分，初始化String对象

offset+count在数组范围之内

public String(char value[], int offset, int count) {

if (offset < 0) {

throw new StringIndexOutOfBoundsException(offset);

}

if (count <= 0) {

if (count < 0) {

throw new StringIndexOutOfBoundsException(count);

}

if (offset <= value.length) {

this.value = "".value;

return;

}

}

// Note: offset or count might be near -1>>>1.

if (offset > value.length - count) {

throw new StringIndexOutOfBoundsException(offset + count);

}

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

}

public String(int[] codePoints, int offset, int count)

通过数字数组的指定部分构建字符串对象

将指定数组转换为它的字符表示

最后新建的字符串对象字符的长度可能大于count，因为某些数字可能对应多个字符

public String(int[] codePoints, int offset, int count) {

if (offset < 0) {

throw new StringIndexOutOfBoundsException(offset);

}

if (count <= 0) {

if (count < 0) {

throw new StringIndexOutOfBoundsException(count);

}

if (offset <= codePoints.length) {

this.value = "".value;

return;

}

}

// Note: offset or count might be near -1>>>1.

if (offset > codePoints.length - count) {

throw new StringIndexOutOfBoundsException(offset + count);

}

final int end = offset + count;

// Pass 1: Compute precise size of char[]

int n = count;

for (int i = offset; i < end; i++) {

int c = codePoints[i];

if (Character.isBmpCodePoint(c))

continue;

else if (Character.isValidCodePoint(c))

n++;

else throw new IllegalArgumentException(Integer.toString(c));

}

// Pass 2: Allocate and fill in char[]

final char[] v = new char[n];

for (int i = offset, j = 0; i < end; i++, j++) {

int c = codePoints[i];

if (Character.isBmpCodePoint(c))

v[j] = (char)c;

else

Character.toSurrogates(c, v, j++);

}

this.value = v;

}

private static void checkBounds(byte[] bytes, int offset, int length)

内部在由字节数组新建字符串对象是调用，判断字节数组的范围是否越界

其实在将字符数组和数字数组转换为字符串对象的时候在代码的前一部分也有范围验证，也是代码重复，也应该合并到一起

private static void checkBounds(byte[] bytes, int offset, int length) {

if (length < 0)

throw new StringIndexOutOfBoundsException(length);

if (offset < 0)

throw new StringIndexOutOfBoundsException(offset);

if (offset > bytes.length - length)

throw new StringIndexOutOfBoundsException(offset + length);

}

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

由字节数组构建新的字符串对象，调用了上面的checkBounds方法检查字节数组的范围

传入的第三个参数为字符编码的名字

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

throws UnsupportedEncodingException {

if (charsetName == null)

throw new NullPointerException("charsetName");

checkBounds(bytes, offset, length);

this.value = StringCoding.decode(charsetName, bytes, offset, length);

}

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

由字节数组构建新的字符串对象，与上面的方法几乎一样，只是传递的第三参数是字符编码对象

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

if (charset == null)

throw new NullPointerException("charset");

checkBounds(bytes, offset, length);

this.value = StringCoding.decode(charset, bytes, offset, length);

}

public String(byte bytes[], String charsetName)

由字节数组构建新的字符串对象，是上面的方法的特殊形式，拷贝到是整个字节数组

public String(byte bytes[], String charsetName)

throws UnsupportedEncodingException {

this(bytes, 0, bytes.length, charsetName);

}

public String(byte bytes[], Charset charset)

由字节数组构建新的字符串对象，是上面几种方法的特殊形式，只是传递的参数为字符编码对象

public String(byte bytes[], Charset charset) {

this(bytes, 0, bytes.length, charset);

}

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

由字节数组构建字符串对象

public String(byte bytes[], int offset, int length) {

checkBounds(bytes, offset, length);

this.value = StringCoding.decode(bytes, offset, length);

}

public String(byte bytes[])

public String(byte bytes[]) {

this(bytes, 0, bytes.length);

}

public String(StringBuffer buffer)

由StringBuffer构建新的字符串对象

正是因为使用了synchronized关键字所有才是线程安全的

public String(StringBuffer buffer) {

synchronized(buffer) {

this.value = Arrays.copyOf(buffer.getValue(), buffer.length());

}

}

public String(StringBuilder builder)

由StringBuilder构建新的字符串对象

因为没有使用synchronized关键字，并且可拷贝的操作是非原子操作，所以是非线程安全的

public String(StringBuilder builder) {

this.value = Arrays.copyOf(builder.getValue(), builder.length());

}

String(char[] value, boolean share)

由字符数组构建新的字符串对象，貌似好像很少使用

这个构造方法是包内方法

String(char[] value, boolean share) {

// assert share : "unshared not supported";

this.value = value;

}

字符串类自身的非构造方法：

public int length()

返回字符串的长度，即字符数组元素的个数

public int length() {

return value.length;

}

public boolean isEmpty()

仅能够判断是否为空字符串，不能判断是否为null

public boolean isEmpty() {

return value.length == 0;

}

public char charAt(int index)

字符串指定索引处的字符，字符串的底层是字符数组因此有可以使用索引查找

public char charAt(int index) {

if ((index < 0) || (index >= value.length)) {

throw new StringIndexOutOfBoundsException(index);

}

return value[index];

}

public int codePointAt(int index)

字符串指定索引处字符的Unicode代码点

public int codePointAt(int index) {

if ((index < 0) || (index >= value.length)) {

throw new StringIndexOutOfBoundsException(index);

}

return Character.codePointAtImpl(value, index, value.length);

}

public int codePointBefore(int index)

字符串指定索引处字符的Unicode代码点的前一个代码点

public int codePointBefore(int index) {

int i = index - 1;

if ((i < 0) || (i >= value.length)) {

throw new StringIndexOutOfBoundsException(index);

}

return Character.codePointBeforeImpl(value, index, 0);

}

public int codePointCount(int beginIndex, int endIndex)

返回字符串指定文本范围内的Unicode代码点数

public int codePointCount(int beginIndex, int endIndex) {

if (beginIndex < 0 || endIndex > value.length || beginIndex > endIndex) {

throw new IndexOutOfBoundsException();

}

return Character.codePointCountImpl(value, beginIndex, endIndex - beginIndex);

}

??@public int offsetByCodePoints(int index, int codePointOffset)

返回字符串从index索引处偏移codePintOffset个代码点的索引

public int offsetByCodePoints(int index, int codePointOffset) {

if (index < 0 || index > value.length) {

throw new IndexOutOfBoundsException();

}

return Character.offsetByCodePointsImpl(value, 0, value.length,

index, codePointOffset);

}

void getChars(char dst[], int dstBegin)

包内方法，将字符串的字符数组拷贝到dst数组

void getChars(char dst[], int dstBegin) {

System.arraycopy(value, 0, dst, dstBegin, value.length);

}

public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin)

将当前字符串的数组元素从指定位置开始到结束为中拷贝到指定的数组的指定开始处

public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin) {

if (srcBegin < 0) {

throw new StringIndexOutOfBoundsException(srcBegin);

}

if (srcEnd > value.length) {

throw new StringIndexOutOfBoundsException(srcEnd);

}

if (srcBegin > srcEnd) {

throw new StringIndexOutOfBoundsException(srcEnd - srcBegin);

}

System.arraycopy(value, srcBegin, dst, dstBegin, srcEnd - srcBegin);

}

public byte[] getBytes(String charsetName)

以指定的字符编码返回字符串的字节数组，参数为编码的名字

public byte[] getBytes(String charsetName)

throws UnsupportedEncodingException {

if (charsetName == null) throw new NullPointerException();

return StringCoding.encode(charsetName, value, 0, value.length);

}

public byte[] getBytes(Charset charset)

以指定的编码格式返回字符串的字节流，参数为编码的对象

public byte[] getBytes(Charset charset) {

if (charset == null) throw new NullPointerException();

return StringCoding.encode(charset, value, 0, value.length);

}

public byte[] getBytes()

使用平台默认的编码格式返回字符串的字节序列数组

public byte[] getBytes() {

return StringCoding.encode(value, 0, value.length);

}

public boolean equals(Object anObject)

重写了object的equals方法，比较两个字符串，可以作为比较方法的模板：

先判断是否是同一个对象，然后判断是否instanceof 一个类，然后在具体比较对象的关键属性

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) { #以便在相等的时候推出循环

if (v1[i] != v2[i])

return false;

i++;

}

return true;

}

}

return false;

}

public boolean contentEquals(StringBuffer sb)

调用内部的方法和StringBuffer比较

public boolean contentEquals(StringBuffer sb) {

return contentEquals((CharSequence)sb);

}

private boolean nonSyncContentEquals(AbstractStringBuilder sb)

非同步与StringBuilder比较

因为stringBuilder不是线程安全的，所以与比较StringBuffer的方法不同

private boolean nonSyncContentEquals(AbstractStringBuilder sb) {

char v1[] = value;

char v2[] = sb.getValue();

int n = v1.length;

if (n != sb.length()) {

return false;

}

for (int i = 0; i < n; i++) {

if (v1[i] != v2[i]) {

return false;

}

}

return true;

}

public boolean contentEquals(CharSequence cs)

被上面比较StringBuffer的方法调用，比较与CharSequence对象是否相等

如果CharSequence是StringBuffer的话，则调用的是上面的同步比较方法

如果CharSequence是StringBuilder的话，则调用的是上面的非同步比较方法

如果CharSequence是String的话，则直接调用字符串的比较方法

public boolean contentEquals(CharSequence cs) {

// Argument is a StringBuffer, StringBuilder

if (cs instanceof AbstractStringBuilder) {

if (cs instanceof StringBuffer) {

synchronized(cs) { #比较StringBuffer时的同步锁

return nonSyncContentEquals((AbstractStringBuilder)cs);

}

} else {

return nonSyncContentEquals((AbstractStringBuilder)cs);

}

}

// Argument is a String

if (cs instanceof String) {

return equals(cs);

}

// Argument is a generic CharSequence

char v1[] = value;

int n = v1.length;

if (n != cs.length()) {

return false;

}

for (int i = 0; i < n; i++) {

if (v1[i] != cs.charAt(i)) {

return false;

}

}

return true;

}

public boolean equalsIgnoreCase(String anotherString)

忽略大小写的比较两个字符串

public boolean equalsIgnoreCase(String anotherString) {

return (this == anotherString) ? true

: (anotherString != null)

&& (anotherString.value.length == value.length)

&& regionMatches(true, 0, anotherString, 0, value.length);

}

public int compareTo(String anotherString)

比较两个字符串，一个字符一个字符的比较

如果两个字符串的值相等则返回0

如果两个字符串中叫较短字符串在较长字符串的位置一样则，返回两个字符串的长度差

如果两个字符串较短串与较长字符串同位置的字符不同，则返回两个不同字符串的差

public int compareTo(String anotherString) {

int len1 = value.length;

int len2 = anotherString.value.length;

int lim = Math.min(len1, len2);

char v1[] = value;

char v2[] = anotherString.value;

int k = 0;

while (k < lim) {

char c1 = v1[k];

char c2 = v2[k];

if (c1 != c2) {

return c1 - c2;

}

k++;

}

return len1 - len2;

}

public int compareToIgnoreCase(String str)

调用内部类比较两个字符串

public int compareToIgnoreCase(String str) {

return CASE\_INSENSITIVE\_ORDER.compare(this, str);

}

public boolean regionMatches(int toffset, String other, int ooffset,int len)

比较两个字符串指定范围的字符，部分大小写

public boolean regionMatches(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) {

if (ta[to++] != pa[po++]) {

return false;

}

}

return true;

}

public boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)

忽略大小写比较两个字符串的大小

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) {

// If characters don't match but case may be ignored,

// try converting both characters to uppercase.

// If the results match, then the comparison scan should

// continue.

char u1 = Character.toUpperCase(c1);

char u2 = Character.toUpperCase(c2);

if (u1 == u2) {

continue;

}

// Unfortunately, conversion to uppercase does not work properly

// for the Georgian alphabet, which has strange rules about case

// conversion. So we need to make one last check before

// exiting.

if (Character.toLowerCase(u1) == Character.toLowerCase(u2)) {

continue;

}

}

return false;

}

return true;

}

public boolean startsWith(String prefix, int toffset)

判断字符串指定偏移量是否已指定的字符串开头

public boolean startsWith(String prefix, int toffset) {

char ta[] = value;

int to = toffset;

char pa[] = prefix.value;

int po = 0;

int pc = prefix.value.length;

// Note: toffset might be near -1>>>1.

if ((toffset < 0) || (toffset > value.length - pc)) {

return false;

}

while (--pc >= 0) {

if (ta[to++] != pa[po++]) {

return false;

}

}

return true;

}

public boolean startsWith(String prefix)

判断字符串是否以指定的字符串开头

此方法是上面方法的特殊情况

public boolean startsWith(String prefix) {

return startsWith(prefix, 0);

}

public boolean endsWith(String suffix)

片段字符串是否以指定的字符串结尾

此方法是上面方法的特殊情况

public boolean endsWith(String suffix) {

return startsWith(suffix, value.length - suffix.value.length);

}

public int hashCode()

字符串的hashCode

public int hashCode() {

int h = hash;

if (h == 0 && value.length > 0) {

char val[] = value;

for (int i = 0; i < value.length; i++) {

h = 31 \* h + val[i];

}

hash = h;

}

return h;

}

public int indexOf(int ch)

从字符串开头开始，返回指定字符数字表示的索引

public int indexOf(int ch) {

return indexOf(ch, 0);

}

public int indexOf(int ch, int fromIndex)

返回指定数字对应字符从指定偏移量开始的索引

public int indexOf(int ch, int fromIndex) {

final int max = value.length;

if (fromIndex < 0) {

fromIndex = 0;

} else if (fromIndex >= max) {

// Note: fromIndex might be near -1>>>1.

return -1;

}

if (ch < Character.MIN\_SUPPLEMENTARY\_CODE\_POINT) {

// handle most cases here (ch is a BMP code point or a

// negative value (invalid code point))

final char[] value = this.value;

for (int i = fromIndex; i < max; i++) {

if (value[i] == ch) {

return i;

}

}

return -1;

} else {

return indexOfSupplementary(ch, fromIndex);

}

}

private int indexOfSupplementary(int ch, int fromIndex)

私有方法，判断指定数字对应的字符的索引

该方法被上面的方法调用

private int indexOfSupplementary(int ch, int fromIndex) {

if (Character.isValidCodePoint(ch)) {

final char[] value = this.value;

final char hi = Character.highSurrogate(ch);

final char lo = Character.lowSurrogate(ch);

final int max = value.length - 1;

for (int i = fromIndex; i < max; i++) {

if (value[i] == hi && value[i + 1] == lo) {

return i;

}

}

}

return -1;

}

public int lastIndexOf(int ch)

返回指定数字对应的字符在字符串中的最后一个索引

public int lastIndexOf(int ch) {

return lastIndexOf(ch, value.length - 1);

}

public int lastIndexOf(int ch, int fromIndex)

返回指定数字对应的字符从字符串指定位置开始向前查找的最后一个索引

public int lastIndexOf(int ch, int fromIndex) {

if (ch < Character.MIN\_SUPPLEMENTARY\_CODE\_POINT) {

// handle most cases here (ch is a BMP code point or a

// negative value (invalid code point))

final char[] value = this.value;

int i = Math.min(fromIndex, value.length - 1);

for (; i >= 0; i--) {

if (value[i] == ch) {

return i;

}

}

return -1;

} else {

return lastIndexOfSupplementary(ch, fromIndex);

}

}

private int lastIndexOfSupplementary(int ch, int fromIndex)

内部方法，该方法被上一个方法调用

private int lastIndexOfSupplementary(int ch, int fromIndex) {

if (Character.isValidCodePoint(ch)) {

final char[] value = this.value;

char hi = Character.highSurrogate(ch);

char lo = Character.lowSurrogate(ch);

int i = Math.min(fromIndex, value.length - 2);

for (; i >= 0; i--) {

if (value[i] == hi && value[i + 1] == lo) {

return i;

}

}

}

return -1;

}

public int indexOf(String str)

返回指定字符在字符串的索引

public int indexOf(String str) {

return indexOf(str, 0);

}

public int indexOf(String str, int fromIndex)

从指定偏移量开始查找，返回指定的字符在字符串中的索引

public int indexOf(String str, int fromIndex) {

return indexOf(value, 0, value.length,

str.value, 0, str.value.length, fromIndex);

}

//========20170911

static int indexOf(char[] source, int sourceOffset, int sourceCount, String target, int fromIndex)

static int indexOf(char[] source, int sourceOffset, int sourceCount,

String target, int fromIndex) {

return indexOf(source, sourceOffset, sourceCount,

target.value, 0, target.value.length,

fromIndex);

}

static int indexOf(char[] source, int sourceOffset, int sourceCount, char[] target, int targetOffset, int targetCount,

int fromIndex)

static int indexOf(char[] source, int sourceOffset, int sourceCount,

char[] target, int targetOffset, int targetCount,

int fromIndex) {

if (fromIndex >= sourceCount) {

return (targetCount == 0 ? sourceCount : -1);

}

if (fromIndex < 0) {

fromIndex = 0;

}

if (targetCount == 0) {

return fromIndex;

}

char first = target[targetOffset];

int max = sourceOffset + (sourceCount - targetCount);

for (int i = sourceOffset + fromIndex; i <= max; i++) {

/\* Look for first character. \*/

if (source[i] != first) {

while (++i <= max && source[i] != first);

}

/\* Found first character, now look at the rest of v2 \*/

if (i <= max) {

int j = i + 1;

int end = j + targetCount - 1;

for (int k = targetOffset + 1; j < end && source[j]

== target[k]; j++, k++);

if (j == end) {

/\* Found whole string. \*/

return i - sourceOffset;

}

}

}

return -1;

}

public int lastIndexOf(String str)

public int lastIndexOf(String str) {

return lastIndexOf(str, value.length);

}

public int lastIndexOf(String str, int fromIndex)

public int lastIndexOf(String str, int fromIndex) {

return lastIndexOf(value, 0, value.length,

str.value, 0, str.value.length, fromIndex);

}

static int lastIndexOf(char[] source, int sourceOffset, int sourceCount, String target, int fromIndex)

sstatic int lastIndexOf(char[] source, int sourceOffset, int sourceCount,

String target, int fromIndex) {

return lastIndexOf(source, sourceOffset, sourceCount,

target.value, 0, target.value.length,

fromIndex);

static int lastIndexOf(char[] source, int sourceOffset, int sourceCount,

String target, int fromIndex) {

return lastIndexOf(source, sourceOffset, sourceCount,

target.value, 0, target.value.length,

fromIndex);

}

static int lastIndexOf(char[] source, int sourceOffset, int sourceCount, char[] target, int targetOffset, int targetCount, int fromIndex)

static int lastIndexOf(char[] source, int sourceOffset, int sourceCount,

char[] target, int targetOffset, int targetCount,

int fromIndex) {

/\*

\* Check arguments; return immediately where possible. For

\* consistency, don't check for null str.

\*/

int rightIndex = sourceCount - targetCount;

if (fromIndex < 0) {

return -1;

}

if (fromIndex > rightIndex) {

fromIndex = rightIndex;

}

/\* Empty string always matches. \*/

if (targetCount == 0) {

return fromIndex;

}

int strLastIndex = targetOffset + targetCount - 1;

char strLastChar = target[strLastIndex];

int min = sourceOffset + targetCount - 1;

int i = min + fromIndex;

startSearchForLastChar:

while (true) {

while (i >= min && source[i] != strLastChar) {

i--;

}

if (i < min) {

return -1;

}

int j = i - 1;

int start = j - (targetCount - 1);

int k = strLastIndex - 1;

while (j > start) {

if (source[j--] != target[k--]) {

i--;

continue startSearchForLastChar;

}

}

return start - sourceOffset + 1;

}

}

public String substring(int beginIndex)

返回从指定位置开始到最后的子字符串

public String substring(int beginIndex) {

if (beginIndex < 0) {

throw new StringIndexOutOfBoundsException(beginIndex);

}

int subLen = value.length - beginIndex;

if (subLen < 0) {

throw new StringIndexOutOfBoundsException(subLen);

}

return (beginIndex == 0) ? this : new String(value, beginIndex, subLen);

}

public String substring(int beginIndex, int endIndex)

返回指定范围的字符串

public String substring(int beginIndex, int endIndex) {

if (beginIndex < 0) {

throw new StringIndexOutOfBoundsException(beginIndex);

}

if (endIndex > value.length) {

throw new StringIndexOutOfBoundsException(endIndex);

}

int subLen = endIndex - beginIndex;

if (subLen < 0) {

throw new StringIndexOutOfBoundsException(subLen);

}

return ((beginIndex == 0) && (endIndex == value.length)) ? this

: new String(value, beginIndex, subLen);

}

public CharSequence subSequence(int beginIndex, int endIndex)

返回字符串指定的范围的字符序列

实际调用的是字符串范围的方法，返回的也是字符串

public CharSequence subSequence(int beginIndex, int endIndex) {

return this.substring(beginIndex, endIndex);

}

public String concat(String str)

将指定的字符串拼接在字符串的后面

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);

}

public String replace(char oldChar, char newChar)

以就字符替换新的字符

public String replace(char oldChar, char newChar) {

if (oldChar != newChar) {

int len = value.length;

int i = -1;

char[] val = value; /\* avoid getfield opcode \*/

while (++i < len) {

if (val[i] == oldChar) {

break;

}

}

if (i < len) {

char buf[] = new char[len];

for (int j = 0; j < i; j++) {

buf[j] = val[j];

}

while (i < len) {

char c = val[i];

buf[i] = (c == oldChar) ? newChar : c;

i++;

}

return new String(buf, true);

}

}

return this;

}

public boolean matches(String regex)

判断字符串是否符合某种模式，利用了正则的matches方法

因为它查找的是模式所有使用正则的匹配无可厚非

public boolean matches(String regex) {

return Pattern.matches(regex, this);

}

public boolean contains(CharSequence s)

判断字符串是否包含指定的字符序列

public boolean contains(CharSequence s) {

return indexOf(s.toString()) > -1;

}

public String replaceFirst(String regex, String replacement)

public String replaceFirst(String regex, String replacement) {

return Pattern.compile(regex).matcher(this).replaceFirst(replacement);

}

public String replaceAll(String regex, String replacement)

public String replaceAll(String regex, String replacement) {

return Pattern.compile(regex).matcher(this).replaceAll(replacement);

}

public String replace(CharSequence target, CharSequence replacement)

public String replace(CharSequence target, CharSequence replacement) {

return Pattern.compile(target.toString(), Pattern.LITERAL).matcher(

this).replaceAll(Matcher.quoteReplacement(replacement.toString()));

}

public String[] split(String regex, int limit)

根据指定的格式分隔字符串，指定分分割的个数

传入的第一个参数为正则

public String[] split(String regex, int limit) {

/\* fastpath if the regex is a

(1)one-char String and this character is not one of the

RegEx's meta characters ".$|()[{^?\*+\\", or

(2)two-char String and the first char is the backslash and

the second is not the ascii digit or ascii letter.

\*/

char ch = 0;

if (((regex.value.length == 1 &&

".$|()[{^?\*+\\".indexOf(ch = regex.charAt(0)) == -1) ||

(regex.length() == 2 &&

regex.charAt(0) == '\\' &&

(((ch = regex.charAt(1))-'0')|('9'-ch)) < 0 &&

((ch-'a')|('z'-ch)) < 0 &&

((ch-'A')|('Z'-ch)) < 0)) &&

(ch < Character.MIN\_HIGH\_SURROGATE ||

ch > Character.MAX\_LOW\_SURROGATE))

{

int off = 0;

int next = 0;

boolean limited = limit > 0;

ArrayList<String> list = new ArrayList<>();

while ((next = indexOf(ch, off)) != -1) { #1个非正则特殊字符和2个字符其中第一个是转义符，直接使用字符查找索引较快

if (!limited || list.size() < limit - 1) {

list.add(substring(off, next));

off = next + 1;

} else { // last one， #list.size==limit-1则是最后一个

//assert (list.size() == limit - 1);

list.add(substring(off, value.length));

off = value.length;

break;

}

}

// If no match was found, return this

if (off == 0)

return new String[]{this};

// Add remaining segment

if (!limited || list.size() < limit)

list.add(substring(off, value.length));

// Construct result

int resultSize = list.size();

if (limit == 0) {

while (resultSize > 0 && list.get(resultSize - 1).length() == 0) {

resultSize--;

}

}

String[] result = new String[resultSize];

return list.subList(0, resultSize).toArray(result);

}

return Pattern.compile(regex).split(this, limit);

}

public String[] split(String regex)

public String[] split(String regex) {

return split(regex, 0);

}

public static String join(CharSequence delimiter, CharSequence... elements)

public static String join(CharSequence delimiter, CharSequence... elements) {

Objects.requireNonNull(delimiter);

Objects.requireNonNull(elements);

// Number of elements not likely worth Arrays.stream overhead.

StringJoiner joiner = new StringJoiner(delimiter);

for (CharSequence cs: elements) {

joiner.add(cs);

}

return joiner.toString();

}

public static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)

public static String join(CharSequence delimiter,

Iterable<? extends CharSequence> elements) {

Objects.requireNonNull(delimiter);

Objects.requireNonNull(elements);

StringJoiner joiner = new StringJoiner(delimiter);

for (CharSequence cs: elements) {

joiner.add(cs);

}

return joiner.toString();

}

public String toLowerCase(Locale locale)

public String toLowerCase(Locale locale) {

if (locale == null) {

throw new NullPointerException();

}

int firstUpper;

final int len = value.length;

/\* Now check if there are any characters that need to be changed. \*/

scan: {

for (firstUpper = 0 ; firstUpper < len; ) {

char c = value[firstUpper];

if ((c >= Character.MIN\_HIGH\_SURROGATE)

&& (c <= Character.MAX\_HIGH\_SURROGATE)) {

int supplChar = codePointAt(firstUpper);

if (supplChar != Character.toLowerCase(supplChar)) {

break scan;

}

firstUpper += Character.charCount(supplChar);

} else {

if (c != Character.toLowerCase(c)) {

break scan;

}

firstUpper++;

}

}

return this;

}

char[] result = new char[len];

int resultOffset = 0; /\* result may grow, so i+resultOffset

\* is the write location in result \*/

/\* Just copy the first few lowerCase characters. \*/

System.arraycopy(value, 0, result, 0, firstUpper);

String lang = locale.getLanguage();

boolean localeDependent =

(lang == "tr" || lang == "az" || lang == "lt");

char[] lowerCharArray;

int lowerChar;

int srcChar;

int srcCount;

for (int i = firstUpper; i < len; i += srcCount) {

srcChar = (int)value[i];

if ((char)srcChar >= Character.MIN\_HIGH\_SURROGATE

&& (char)srcChar <= Character.MAX\_HIGH\_SURROGATE) {

srcChar = codePointAt(i);

srcCount = Character.charCount(srcChar);

} else {

srcCount = 1;

}

if (localeDependent ||

srcChar == '\u03A3' || // GREEK CAPITAL LETTER SIGMA

srcChar == '\u0130') { // LATIN CAPITAL LETTER I WITH DOT ABOVE

lowerChar = ConditionalSpecialCasing.toLowerCaseEx(this, i, locale);

} else {

lowerChar = Character.toLowerCase(srcChar);

}

if ((lowerChar == Character.ERROR)

|| (lowerChar >= Character.MIN\_SUPPLEMENTARY\_CODE\_POINT)) {

if (lowerChar == Character.ERROR) {

lowerCharArray =

ConditionalSpecialCasing.toLowerCaseCharArray(this, i, locale);

} else if (srcCount == 2) {

resultOffset += Character.toChars(lowerChar, result, i + resultOffset) - srcCount;

continue;

} else {

lowerCharArray = Character.toChars(lowerChar);

}

/\* Grow result if needed \*/

int mapLen = lowerCharArray.length;

if (mapLen > srcCount) {

char[] result2 = new char[result.length + mapLen - srcCount];

System.arraycopy(result, 0, result2, 0, i + resultOffset);

result = result2;

}

for (int x = 0; x < mapLen; ++x) {

result[i + resultOffset + x] = lowerCharArray[x];

}

resultOffset += (mapLen - srcCount);

} else {

result[i + resultOffset] = (char)lowerChar;

}

}

return new String(result, 0, len + resultOffset);

}

public String toLowerCase()

public String toLowerCase() {

return toLowerCase(Locale.getDefault());

}

public String toUpperCase(Locale locale)

public String toUpperCase(Locale locale) {

if (locale == null) {

throw new NullPointerException();

}

int firstLower;

final int len = value.length;

/\* Now check if there are any characters that need to be changed. \*/

scan: {

for (firstLower = 0 ; firstLower < len; ) {

int c = (int)value[firstLower];

int srcCount;

if ((c >= Character.MIN\_HIGH\_SURROGATE)

&& (c <= Character.MAX\_HIGH\_SURROGATE)) {

c = codePointAt(firstLower);

srcCount = Character.charCount(c);

} else {

srcCount = 1;

}

int upperCaseChar = Character.toUpperCaseEx(c);

if ((upperCaseChar == Character.ERROR)

|| (c != upperCaseChar)) {

break scan;

}

firstLower += srcCount;

}

return this;

}

/\* result may grow, so i+resultOffset is the write location in result \*/

int resultOffset = 0;

char[] result = new char[len]; /\* may grow \*/

/\* Just copy the first few upperCase characters. \*/

System.arraycopy(value, 0, result, 0, firstLower);

String lang = locale.getLanguage();

boolean localeDependent =

(lang == "tr" || lang == "az" || lang == "lt");

char[] upperCharArray;

int upperChar;

int srcChar;

int srcCount;

for (int i = firstLower; i < len; i += srcCount) {

srcChar = (int)value[i];

if ((char)srcChar >= Character.MIN\_HIGH\_SURROGATE &&

(char)srcChar <= Character.MAX\_HIGH\_SURROGATE) {

srcChar = codePointAt(i);

srcCount = Character.charCount(srcChar);

} else {

srcCount = 1;

}

if (localeDependent) {

upperChar = ConditionalSpecialCasing.toUpperCaseEx(this, i, locale);

} else {

upperChar = Character.toUpperCaseEx(srcChar);

}

if ((upperChar == Character.ERROR)

|| (upperChar >= Character.MIN\_SUPPLEMENTARY\_CODE\_POINT)) {

if (upperChar == Character.ERROR) {

if (localeDependent) {

upperCharArray =

ConditionalSpecialCasing.toUpperCaseCharArray(this, i, locale);

} else {

upperCharArray = Character.toUpperCaseCharArray(srcChar);

}

} else if (srcCount == 2) {

resultOffset += Character.toChars(upperChar, result, i + resultOffset) - srcCount;

continue;

} else {

upperCharArray = Character.toChars(upperChar);

}

/\* Grow result if needed \*/

int mapLen = upperCharArray.length;

if (mapLen > srcCount) {

char[] result2 = new char[result.length + mapLen - srcCount];

System.arraycopy(result, 0, result2, 0, i + resultOffset);

result = result2;

}

for (int x = 0; x < mapLen; ++x) {

result[i + resultOffset + x] = upperCharArray[x];

}

resultOffset += (mapLen - srcCount);

} else {

result[i + resultOffset] = (char)upperChar;

}

}

return new String(result, 0, len + resultOffset);

}

public String toUpperCase()

public String toUpperCase() {

return toUpperCase(Locale.getDefault());

}

public String trim()

取出字符串两侧的空字符

public String trim() {

int len = value.length;

int st = 0;

char[] val = value; /\* avoid getfield opcode \*/

while ((st < len) && (val[st] <= ' ')) { #空字符的数字为32

st++;

}

while ((st < len) && (val[len - 1] <= ' ')) {

len--;

}

return ((st > 0) || (len < value.length)) ? substring(st, len) : this;

}

public String toString()

字符串的toString方法，重写了Object的toString方法

public String toString() {

return this;

}

public char[] toCharArray()

返回字符串的字符数组，这里对字符数组做了拷贝

public char[] toCharArray() {

// Cannot use Arrays.copyOf because of class initialization order issues

char result[] = new char[value.length];

System.arraycopy(value, 0, result, 0, value.length);

return result;

}

public static String format(String format, Object... args)

根据占位符和参数返回字符串

public static String format(String format, Object... args) {

return new Formatter().format(format, args).toString();

}

public static String format(Locale l, String format, Object... args)

public static String format(Locale l, String format, Object... args) {

return new Formatter(l).format(format, args).toString();

}

public static String valueOf(Object obj)

根据object构建字符串对象，调用的是object的toString方法

如果为null的话则直接返回null

public static String valueOf(Object obj) {

return (obj == null) ? "null" : obj.toString();

}

public static String valueOf(char data[])

根据字符数组构建新的字符串对象

public static String valueOf(char data[]) {

return new String(data);

}

public static String copyValueOf(char data[], int offset, int count)

静态方法，根据字符数组和偏移量、指定数量的元素构建新的字符串对象

public static String copyValueOf(char data[], int offset, int count) {

return new String(data, offset, count);

}

public static String copyValueOf(char data[])

静态方法，根据字符数组构建新的字符串对象

public static String copyValueOf(char data[]) {

return new String(data);

}

以下为各个基本类型的字符串值valueOf，包括：boolean char int float long double

这些方法调用的都是基本类型的包装类型的toString方法，

除了true直接判断返回true或者false；char利用的是字符串的String的构造方法

public static String valueOf(boolean b)

public static String valueOf(boolean b) {

return b ? "true" : "false";

}

public static String valueOf(char c)

public static String valueOf(char c) {

char data[] = {c};

return new String(data, true);

}

public static String valueOf(int i)

public static String valueOf(int i) {

return Integer.toString(i);

}

public static String valueOf(long l)

public static String valueOf(long l) {

return Long.toString(l);

}

public static String valueOf(float f)

public static String valueOf(float f) {

return Float.toString(f);

}

public static String valueOf(double d)

public static String valueOf(double d) {

return Double.toString(d);

}

public native String intern();

返回字符串对象的规范表示

内部类

private static class CaseInsensitiveComparator

private static class CaseInsensitiveComparator

implements Comparator<String>, java.io.Serializable {

// use serialVersionUID from JDK 1.2.2 for interoperability

private static final long serialVersionUID = 8575799808933029326L;

public int compare(String s1, String s2) {

int n1 = s1.length();

int n2 = s2.length();

int min = Math.min(n1, n2);

for (int i = 0; i < min; i++) {

char c1 = s1.charAt(i);

char c2 = s2.charAt(i);

if (c1 != c2) {

c1 = Character.toUpperCase(c1);

c2 = Character.toUpperCase(c2);

if (c1 != c2) {

c1 = Character.toLowerCase(c1);

c2 = Character.toLowerCase(c2);

if (c1 != c2) {

// No overflow because of numeric promotion

return c1 - c2;

}

}

}

}

return n1 - n2;

}

/\*\* Replaces the de-serialized object. \*/

private Object readResolve() { return CASE\_INSENSITIVE\_ORDER; }

}