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;

}

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; }

}