《Java应用技术》第三次作业

姓名	学号	班级
沈子衿	3160104734	软件工程1601

1.String, StringBuffer, StringBuilder源码分析

1.1 String类

Java的String类相关代码可以通过对String.class反编译得到，主流IDE也大多提供了直接反编译查看.class文件对应源代码的支持。本次作业，我们使用Eclipse作为工具，来对String类进行分析。

外部引用

在文件的开头，可以看到String类的打包信息和引用的外部包，如下：

```
package java.lang;
// 将String类打包到java.lang.*下
import java.io.ObjectStreamField;
//I/O包，负责对可序列化字段的处理
import java.io.UnsupportedEncodingException;
//I/O包，负责在数据编码不支持的情况下抛出异常
import java.nio.charset.Charset;
//java字符集支持
import java.util.ArrayList;
//java ArrayList类支持
import java.util.Arrays;
//java Arrays类支持
import java.util.Comparator;
// 比较器，用于排序、分组等功能
import java.util.Formatter;
// 格式化类，可用于数字、日期、时间等的格式化
import java.util.Locale;
// 提供多地区、文化支持
import java.util.Objects;
//java Objects类
import java.util.StringJoiner;
// 用于构造一个由分隔符分隔的字符序列
import java.util.regex.Matcher;
//正则匹配
import java.util.regex.Pattern;
// 正则编译支持
import java.util.regex.PatternSyntaxException;
// 处理正则表达式编译可能出现的语法错误
```

总体来说，String类引用的外部包主要有四种用途：第一种，用于底层数据结构和数据内容（如编码等）的组织，如Charset类、StringJoiner类和ObjectStreamField类，第二种，用于实现字符串的一些基本操作和功能（剪裁、匹配、排序、分组、格式化等）如Comparator类、Foramtter类和两个regex类；第三种，用于实现同其他数据结构的互动与相互转化，如Arrays和ArrayList类；第四种，用于其他底层支持，如Objects类。不同的用途均有其对应的异常处理支持。

在类主体开始之前，我们看到了注释中有指向StringBuffer和StringBuilder类的链接：

```
/* @author  Lee Boynton
 * @author  Arthur van Hoff
 * @author  Martin Buchholz
 * @author  Ulf Zibis
 * @see     java.lang.Object#toString()
 * @see     java.lang.StringBuffer
 * @see     java.lang.StringBuilder
 * @see     java.nio.charset.Charset
 * @since   JDK1.0
 */
```

这体现了三个类之间密不可分的关系。

成员变量

这之后，我们开始分析String类的具体实现，首先观察类头：

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

这说明String类调用了java.io.Serializable, Comparable, CharSequence这三个接口。

进入类体，首先是三个String类最核心的四个成员变量：

```
    private final char value[];
    private int hash; // Default to 0
    private static final long serialVersionUID = -6849794470754667710L;
    private static final ObjectStreamField[] serialPersistentFields =
        new ObjectStreamField[0];
```

从注释中我们可以了解到，第一个value数组用来储存字符串中的字符信息，是String类的核心存储数据结构，注意到这个变量是private的且被final修饰，因此我们不能直接通过成员访问符访问value变量，也不能修改他的值，**这便是String类生成的字符串不可更改的原因**；第二个hash变量用于存储这个字符串的哈希值（默认为0），其目的是暂存hashCode方法的值，避免反复调用这一具有巨大时间复杂度的方法。第三个变量serialVersionUID用于在序列化验证中判断类的版本一致性，确保版本更新不会导致相关引用出错；第四个变量也是用于序列化验证的，同本文主题不一致，相关资料中的叙述也比较繁琐，故略去不表。

构造方法

接下来，源代码描述了String类的几个主要的构造方法：

```
public String() {
    this.value = "".value;
}
```

本方法是String的默认构造方法。在创建对象且没有初始值的的情况下，调用该构造方法，创建一个空字符串。

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

本方法接收一个现有字符串，创建一个和源字符串完全相同的新字符串。注意，该字符串是原来的复制，而非一个引用。看似新字符串的value是原字符串value的引用，但考虑到字符串的不可更改性，所以直接让value数组复制源字符串的引用，并没有什么不妥的。

```
public String(char value[]) {
    this.value = Arrays.copyOf(value, value.length);
}
```

本方法接收一个字符数组，创建一个和字符数组中各字符对应的字符串。

```
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);
}
```

本方法接受一个字符数组、一个偏移量以及一个计数变量，其功能是将字符数组中从offset开始的count个字符摘取出来，组成一个字符串。如果count等于0且偏移量不大于value的长度，返回空字符串。其他异常情况，抛出下标异常。不支持负向拷贝。

```
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;
    }
```

本方法分配一个新的String，其中包含Unicode codePoint数组参数的子数组中的字符。 offset参数是子数组的第一个codePoint的索引，count参数指定子数组的长度。 子数组的内容转换为char s; 该int数组的后续修改不会影响新创建的字符串。

```
    @Deprecated
    public String(byte ascii[], int hibyte, int offset, int count) {
        checkBounds(ascii, offset, count);
        char value[] = new char[count];

        if (hibyte == 0) {
            for (int i = count; i-- > 0;) {
                value[i] = (char)(ascii[i + offset] & 0xff);
            }
```

```
    } else {
        hibyte <<= 8;
        for (int i = count; i-- > 0;) {
            value[i] = (char)(hibyte | (ascii[i + offset] & 0xff));
        }
    }
    this.value = value;
}

@Deprecated
public String(byte ascii[], int hibyte) {
    this(ascii, hibyte, 0, ascii.length);
}
```

接下来，代码还描述了两个过往版本的构造方法。考虑到当前版本中（JDK1.1之后）官方已经不推荐使用（带有@Deprecated标记），故略去不表。两个方法提供的byte数组转化为字符串的解决方案已经为以下的方法所取代：

```
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) {
        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)
            throws UnsupportedEncodingException {
        this(bytes, 0, bytes.length, charsetName);
}

public String(byte bytes[], Charset charset) {
        this(bytes, 0, bytes.length, charset);
}

public String(byte bytes[], int offset, int length) {
        checkBounds(bytes, offset, length);
        this.value = StringCoding.decode(bytes, offset, length);
}

public String(byte bytes[]) {
        this(bytes, 0, bytes.length);
}
```

上面的方法使用了更新的特性，也更加安全。它们主要实现了byte字节数组向字符串的转换，支持自定义字符集（通过字符集名称或Charset对象表征）、长度、偏移量等，提供了基本但完备的异常处理机制。

```
public String(StringBuffer buffer) {
        synchronized(buffer) {
            this.value = Arrays.copyOf(buffer.getValue(), buffer.length());
        }
}
 public String(StringBuilder builder) {
        this.value = Arrays.copyOf(builder.getValue(), builder.length());
}
```

这两个方法支持通过StringBuffer和StringBuilder对象创建String对象。值得注意的是，StringBuffer中设置了synchronized关键字，这表示最多只有一个进程可以访问对应缓冲区，体现了StringBuffer作为内存缓冲区的特性。

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

这个方法为StringBuffer的toString方法提供加速。当share为true时，构造函数将会使用StringBuffer寄存器toStringCache里的值，而非value里的值。这将加速转换的过程。

以上就是String类的全部构造方法。可以看出，它们已经涵盖了基本需求。

在构造方法之间，还有这样一个方法：

```
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);
}
```

该方法用于在字节数组中进行边界检查。

接下来，代码描述了一些对String对象基本信息的获取方法（由于篇幅限制，部分方法体不得不加以省略）：

基本属性方法

```
 public int length() {
        return value.length;
}

public boolean isEmpty() {
        return value.length == 0;
    }

public char charAt(int index) {
```

```
        if ((index < 0) || (index >= value.length)) {
            throw new StringIndexOutOfBoundsException(index);
        }
        return value[index];
}
public int codePointAt(int index);
public int codePointBefore(int index);
public int codePointCount(int beginIndex, int endIndex);
public int offsetByCodePoints(int index, int codePointOffset);
```

上面的前三个方法分别可以获取字符串的长度、是否为空以及指定位置的字符；下面的四个方法则为Unicode字符串的对应操作。

getChars方法

```
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) {
        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);
}
```

getChars方法获取字符串中指定一段字符序列，存入目标字符数组中的指定位置，支持复制整个字符串以及只截取一段字符串。

getBytes方法

```
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) {
        if (charset == null) throw new NullPointerException();
        return StringCoding.encode(charset, value, 0, value.length);
}
public byte[] getBytes() {
        return StringCoding.encode(value, 0, value.length);
}
```

getBytes支持在指定字符集下读取字符串对应的byte流，支持自定义字符集，也支持智能检测（根据encode方法逻辑，首先方法会检测最符合字符串编码的字符集，否则会依次尝试使用utf-8或ISO-8859-1编码）。

Equals方法

```
public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
        if (anObject instanceof String) {
            String anotherString = (String)anObject;
            int n = value.length;
            if (n == anotherString.value.length) {
                char v1[] = value;
                char v2[] = anotherString.value;
                int i = 0;
                while (n-- != 0) {
                    if (v1[i] != v2[i])
                        return false;
                    i++;
                }
                return true;
            }
        }
        return false;
}
```

equals方法通过逐个比较字符的方式判断两个字符串是否相等，是检测字符串相等的通用方法。当两个字符串变量是同一对象的不同引用时，方法会直接判断正确。值得注意的是，参数也可以是其他对象，但方法会检测对象是否为String实例，若不是，它会返回false，而不是抛出异常。

contentEquals方法

```
public boolean contentEquals(StringBuffer sb) {
        return contentEquals((CharSequence)sb);
}

public boolean contentEquals(CharSequence cs) {
        // Argument is a StringBuffer, StringBuilder
        if (cs instanceof AbstractStringBuilder) {
            if (cs instanceof StringBuffer) {
                synchronized(cs) {
                   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;
}
```

contentEquals通过将sb转换为CharSequence对象的方式判断其内容是否和String对象的内容相同。CharSequence和String内容相等的判断依然使用逐个字符比较的方式，只是在判断该CharSequence同时是AbstractStringBuilder、StringBuffer或String实例的情况下会调用类对应的方法。

nonSyncContentEquals方法

```
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;
}
```

nonSyncContentEquals方法直接操作StringBuilder和StringBuffer的共同抽象父类AbstractStringBuilder，通过逐个字符判断的方式确定其内容是否和String类内容相等。由于只有在StringBuffer的情况下该方法才能实现“同步”，因此操作抽象父类的方法名冠以“nonSync-”的前缀。

equalsIgnoreCase方法

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

该方法比较两个字符串是否相等，无视大小写。

compareTo方法

```
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;
    }
```

该方法计算两个字符串的“差”。“差”定义如下:若从第一个字符开始到第k个字符都相同，但第k+1个不同，则差为本字符串的该字符-另一个字符串的该字符（可为负）；若一个字符串是另一个的子集（包含完全相等的情况），则差为两个字符串的长度差（可为负）。差以int类型返回。

CaseInsensitiveComparator子类

String类中定义了CaseInsensitiveComparator子类：

```
    public static final Comparator<String> CASE_INSENSITIVE_ORDER
                                         = new 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; }
}
```

该类的唯一成员方法compare使用和compareTo相同的逻辑实现了两个字符串之间的通用"减"方法。值得注意的是，本compare方法忽略大小写。这里之所以同时转化为UpperCase又转化为LowerCase再比较一次，是因为大写转换在处理格鲁吉亚字母时可能存在问题。与之对应，String类也实现了相关方法:

compareToIgnoreCase方法

```
public int compareToIgnoreCase(String str) {
        return CASE_INSENSITIVE_ORDER.compare(this, str);
}
```

该方法通过CaseInsensitiveComparator实例CASE_INSENSITIVE_ORDER调用Comparator的compare方法执行无视大小写的比较。

regionMatches方法

```
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);
```

regionMatches方法通过指定两个String的起点偏移量和区域长度来比较两个字符串的指定区域是否相等。另有一个重载版本，可以指定是否无视大小写。

startsWith方法

```
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) {
            return startsWith(prefix, 0);
    }
    public boolean endsWith(String suffix) {
            return startsWith(suffix, value.length - suffix.value.length);
    }
```

startsWith方法检测某一字符串中从指定下标开始的子字符串以一串指定字符（用字符串表示）为前缀。默认下标为0（通过重载实现）。与之对应，还有检测是否有相应后缀的方法。方法是将下标置于和尾部距离等于后缀长度的位置，然后调用startsWith方法。

hashCode方法

```
    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;
    }
```

hashCode方法生成当前字符串的哈希值。当字符串的哈希值已被计算时，直接返回计算值；当未被计算时（字符串非空但哈希值为0），则进行计算，将计算的哈希值存储在hash中。考虑到字符串可能很长，而计算哈希值的最坏时间复杂度为O(length),毫无疑问，这是为了降低计算哈希的摊还时间复杂度（之后的单次时间复杂度就只是1了），最大限度节省资源。

indexOf与lastIndexOf方法

```
    public int indexOf(int ch) {
            return indexOf(ch, 0);
    }
```

```
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);
        }
}
/**
     * Handles (rare) calls of indexOf with a supplementary character.
     */
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;
}
```

indexOf方法返回从某一下标开始指定字符在字符串中第一次出现的位置，默认起始位置为0。如果没有找到对应字符，则返回-1。indexOfSupplementary方法处理存在ASCII中补充字符的情况，但如代码中注释所言，这种情况是稀少(rare)的。

```
public int lastIndexOf(int ch) {
        return lastIndexOf(ch, value.length - 1);
}
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) {
        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;
    }
```

与indexOf对应，也有相对的lastIndexOf方法.值得注意的是，方法的遍历方向是从fromindex向前，而非indexOf中的向后。此外，fromindex可以大于最大下标而不报错，此时方法会自动将起始点纠正为默认起始点（最后一个字符）。

```
    public int indexOf(String str) {
        return indexOf(str, 0);
    }
    public int indexOf(String str, int fromIndex) {
        return indexOf(value, 0, value.length,
                str.value, 0, str.value.length, 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) {/*-----*/}
    /*****************************************/
    public int lastIndexOf(String str) {
        return lastIndexOf(str, value.length);
    }
    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) {
        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){/*---*/}
```

当indexOf的参数为字符串时，其返回的值为参数子串在字符串中第一次出现的位置。你可以自定义起始匹配地址，也可以调用静态方法指定匹配区段的起始偏移量和长度，这个指定在静态方法中同时适用于被匹配串和匹配串。你也可以选择对匹配串的完全匹配。和字符一样，字串匹配也同样支持lastIndexOf方法。

substring方法

```
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) {
        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) {
        return this.substring(beginIndex, endIndex);
}
```

subString方法返回String中从一个下标参数开始到**另一个下标参数的前一个下标**结束的字串。你可以选择提供endindex，也可以选择不提供。但你提供的下标不可以小于零，也不能大于最大的下标，更不能让后下标小于前下标（但可以等于，返回空串）。另一个值得注意的点是，返回的子串是一个新串，而非原串的引用。此外，代码提供了对CharSequence对象的重载。

concat方法

```
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);
}
```

concat方法将参数中的字符串连接到this字符串之后并返回链接后的字符串。它实现的方法是先把this字符串复制到一段字符数组中，然后使用getChars方法吧str连接在后面。注意，返回的是一个新串，而不是老串，因为String类的字符串是不可修改的。

replace方法

```
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;
}
```

replace方法可以指定一个旧字符和一个新字符，然后将一个字符串中所有出现的旧字符全部替换为新字符。方法是首先找到第一个旧字符，然后从这个旧字符开始一直到字符串末尾一个一个验证，如果是旧字符就加以替换。**注意，这种情况下返回的引用是新生成的字符串**，而非旧串。**但如果传入的旧字符和新字符相同，或没有找到旧字符，即没有任何字符被替换，此时返回的则是旧字符串的引用。**

matches方法

```
public boolean matches(String regex) {
        return Pattern.matches(regex, this);
}
```

该方法检验字符串是否和给定正则表达式匹配。

contains方法

```
public boolean contains(CharSequence s) {
        return indexOf(s.toString()) > -1;
}
```

该方法检测字符串中是否包含和给定字符序列内容相同的字串。

replace及相关方法

```
public String replaceFirst(String regex, String replacement) {
        return Pattern.compile(regex).matcher(this).replaceFirst(replacement);
}
public String replaceAll(String regex, String replacement) {
        return Pattern.compile(regex).matcher(this).replaceAll(replacement);
}
```

replacefirst用replacement字符串替换this字符串中第一个和正则表达式regex匹配的子串；replaceAll用replacement字符串替换this字符串中所有和正则表达式regex匹配的字串；注意反斜杠`\`和美元符号`$`可能导致匹配问题，因为它们是正则表达式关键字的组成部分。

```
public String replace(CharSequence target, CharSequence replacement) {
        return Pattern.compile(target.toString(), Pattern.LITERAL).matcher(
                this).replaceAll(Matcher.quoteReplacement(replacement.toString()));
}
```

CharSequence的replace使用正则表达式替换CharSequence中所有指定字符序列（子串）。

split方法

```
public String[] split(String regex, int limit);
public String[] split(String regex) {
        return split(regex, 0);
}
```

split方法按照给定的regex正则表达式匹配的子串切分字符串，并把切分后的字符串存在新创立的字符串数组对象中返回。你可以选择指定参数limit，他表示了array数组的最大长度，即取前`min(split(regex).length, limit)`段结果存入array中。顺序和在字符串中出现的顺序一致。 注意：如果表达式与输入的任何部分都不匹配，那么结果数组只有一个元素，即该字符串。 当在该字符串的开头存在正宽度匹配时，在结果数组的开头包含空的前导子字符串。 然而，开头的零宽度匹配从不会产生这样的空前导子串。

join方法

```
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) {
        Objects.requireNonNull(delimiter);
        Objects.requireNonNull(elements);
        StringJoiner joiner = new StringJoiner(delimiter);
        for (CharSequence cs: elements) {
            joiner.add(cs);
        }
        return joiner.toString();
}
```

静态方法join方法的功能是将多个CharSequence的内容依次连接，再连接到delimiter尾部，最后返回连接完成的新构造字符串。这里应用了java的一个语法：不定数量多参数。使用方法是在参数的类型名之后加上`...`。 值得注意的是，本方法将检测delimiter和elements是否为空。否则将会抛出异常。 在调用join方法时，你也可以不使用传入多个参数的方式，也可以使用Interable列表的方式。其中`<? extends T>`表示类型的上界，表示参数化类型的可能是T 或是 T的子类。

大小写转换方法

```
public String toLowerCase(Locale locale){/*---*/}
 public String toUpperCase(Locale locale){/*---*/}
public String toLowerCase() {
        return toLowerCase(Locale.getDefault());
}
public String toUpperCase() {
        return toUpperCase(Locale.getDefault());
}
```

java的toLowerCase和toUpperCase是支持多语言的，它支持希腊字母、拉丁字母和西里尔(斯拉夫)字母的大小写转换。Locale类就是为了实现多语言支持而引入的类。其基本逻辑如下：方法首先通过Locale类的getLanguage方法获取语言种类，然后使用Unicode字符类Character中的toChars方法对每个可进行大小写转换的字母依次进行判断和转换。同时，为了防止补充字符干扰，该方法提供了补充字符的额外处理机制。 当不提供地域信息时，Locale可以省略，这时使用Locale类的静态方法getDefault获取默认语言。获取默认语言的过程是复杂的，其中包含获取用户系统语言、scriptKey和所在地域等，由于偏离本文主题，这里略去不表。

trim方法

```
public String trim() {
    int len = value.length;
    int st = 0;
    char[] val = value;    /* avoid getfield opcode */

    while ((st < len) && (val[st] <= ' ')) {
        st++;
    }
    while ((st < len) && (val[len - 1] <= ' ')) {
        len--;
    }
    return ((st > 0) || (len < value.length)) ? substring(st, len) : this;
}
```

trim方法可以去除字符串两端的空白字符（所有ascii值小于空格的都被判断为空白字符），返回剪切后的新字符串。如果没有作剪切，返回原字符串的引用。

toString与toCharArray方法

```
public String toString() {
        return this;
}
```

本方法是为了重写Object.toString方法而存在的，本质上毫无用处。

```
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;
}
```

本方法将一个String字符串对象转化为一个字符数组并返回。

格式化（format）方法

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

本方法将多个args（各参数类型和参数数量视格式化字符串要求而定）利用format格式化字符串格式化为指定形式的字符串并返回。你可以选择指定地域信息，也可以不指定。指定地域信息可以对字符串进行特殊的格式化。

valueOf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);
}
public static String copyValueOf(char data[]) {
    return new String(data);
}
public static String valueOf(boolean b) {
return b ? "true" : "false";
}
public static String valueOf(char c) {
char data[] = {c};
return new String(data, true);
}
public static String valueOf(int i) {
    return Integer.toString(i);
}
public static String valueOf(long l) {
    return Long.toString(l);
}
public static String valueOf(float f) {
    return Float.toString(f);
}
    public static String valueOf(double d) {
    return Double.toString(d);
}
```

valueOf方法返回参数对象调用`toString()`方法后返回的结果。对于字符数组，你可以指定转换的偏移量和长度。copyValueOf()和valueOf是等价的。

intern方法

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

intern 方法会从字符串常量池中查询当前字符串是否存在，若不存在就会将当前字符串放入常量池中.这样可以加速访问。

1.2 StringBuffer和StringBuilder类

StringBuffer类和StringBuilder类（以下简称Buffer类和Builder类）是抽象类AbstractStringBuilder的共同子类。Java从JDK1.0开始支持StringBuffer。从代码注释中看，该类的撰写者将StringBuffer描述为“一个线程安全，可互斥的字符序列（A thread-safe, mutable sequence of characters）”。和String相比，StringBuffer有以下特点：

1. 支持Java多线程同步与互斥，比String更线程安全，多线程编程中多用StringBuffer；
2. 元素可修改；
3. 同时具有Capacity和length两个属性，length <= capacity，但当length大于Capacity时，buffer的capacity会自动加长。

StringBuilder和StringBuffer本质上是一样的，只是前者适用单线程方法，且直到JDK5才被支持。值得注意的是，StringBuilder的执行更快，因为它不需要进程同步的开销。

此外，由于AbstractStringBuilder类中的许多方法实现和String中基本相似，由于篇幅限制，我们不再对其进行细致分析，而着眼于比较它的两个子类与String的区别。

成员变量

首先观察它们的成员变量：

```
// 继承自AbstractStringBuilder
char[] value;
int count;

// StringBuffer
private transient char[] toStringCache;
static final long serialVersionUID = 3388685877147921107L;

// StringBuilder
static final long serialVersionUID = 4383685877147921099L;
```

Buffer类和Builder类的数据存储结构均为父类的value[]数组。它们都有一个成员变量count，表示当前数组中的字符个数（字符串的长度）。注意，capacity并不是成员变量，它只是value数组初始化时的一个属性值。之后我们会讨论这个问题。 Buffer类和Builder类都有表征版本的序列号，但Buffer比Builder多一个私有的toStringCache字符数组。注意到它拥有一个表示其不可被序列化的transient关键字，初步判断该数组的作用是在Buffer的toString操作中充当缓冲区。

构造方法

下面观察Buffer和Builder的构造方法：

```
// 继承自AbstractStringBuilder
AbstractStringBuilder() {
}
AbstractStringBuilder(int capacity) {
value = new char[capacity];
}
//StringBuffer和StringBuilder（两者方法逻辑一致）
public StringBuffer() {
    super(16);
}
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);
}
```

Builder和Buffer的默认构造方法来自其父类。调用者可以指定capacity参数，它确定了char数组的初始大小（并不是长度，默认capacity为16）。如果length没有超过capacity，那么相关操作的效率是可以保证的。如果超出了这一范围，虽然Buffer和Builder可以检测这一溢出并且动态扩大capacity，但这会导致额外的时间和空间开销。 此外，你也可以使用String对象和CharSequence对象初始化Builder和Buffer。这种情况下的初始capacity为16+给定字符串/字符序列长度。

基本属性方法、Capacity

```
// 仅StringBuffer
@Override
public synchronized int length() {
return count;
}
@Override
public synchronized int capacity() {
return value.length;
}
@Override
public synchronized void ensureCapacity(int minimumCapacity) {
super.ensureCapacity(minimumCapacity);
}
@Override
public synchronized void trimToSize() {
super.trimToSize();
}
@Override
public synchronized void setLength(int newLength) {
toStringCache = null;
super.setLength(newLength);
}
@Override
public synchronized char charAt(int index) {
if ((index < 0) || (index >= count))
    throw new StringIndexOutOfBoundsException(index);
return value[index];
}
@Override
public synchronized int codePointAt(int index) {
return super.codePointAt(index);
}
@Override
public synchronized int offsetByCodePoints(int index, int codePointOffset) {
    return super.offsetByCodePoints(index, codePointOffset);
}
@Override
public synchronized void getChars(int srcBegin, int srcEnd, char[] dst,
                                  int dstBegin)
{
    super.getChars(srcBegin, srcEnd, dst, dstBegin);
}
@Override
public synchronized void setCharAt(int index, char ch) {
    if ((index < 0) || (index >= count))
```

```
            throw new StringIndexOutOfBoundsException(index);
        toStringCache = null;
        value[index] = ch;
    }
```

以上方法都在Abstract中有过定义，StringBuilder也没有重写，但在StringBuffer中被重写了。其原因是：StringBuilder是面向多线程编程的，需要确保线程安全，一个方法同时只能有一个线程访问，因此需要重写方法并加上synchronized关键字。 在观察这些方法时，我们还发现，有一部分直接通过调用super对应方法来重写方法，有一些则完全重写了代码逻辑————因为它们需要将toStringCache置为null。这可能同线程安全的需求有关。

append方法

```
// StringBuffer和StringBuilder
@Override
public synchronized StringBuffer append(Object obj);
@Override
public synchronized StringBuffer append(String str);
public synchronized StringBuffer append(StringBuffer sb);
// StringBuilder有对应的public StringBuffer append(StringBuilder sb);
@Override
synchronized StringBuffer append(AbstractStringBuilder asb);
@Override
public synchronized StringBuffer append(CharSequence s);
@Override
public synchronized StringBuffer append(CharSequence s, int start, int end);
@Override
public synchronized StringBuffer append(char[] str);
@Override
public synchronized StringBuffer append(char[] str, int offset, int len);
@Override
public synchronized StringBuffer append(boolean b);
@Override
public synchronized StringBuffer append(char c);
@Override
public synchronized StringBuffer append(int i);
@Override
public synchronized StringBuffer appendCodePoint(int codePoint);
@Override
public synchronized StringBuffer append(long lng);
@Override
public synchronized StringBuffer append(float f);
@Override
public synchronized StringBuffer append(double d);
```

append方法体现了StringBuffer和StringBuilder可变的特性。StringBuilder和StringBuffer关于append方法的实现是几乎完全一致的，只有两点不同：第一，StringBuilder方法没有synchronized关键字；第二，StringBuilder方法没有将toStringCache置为null的操作。 Buffer和Builder操作的append方法可以接收众多参数。包括一般对象、String字符串、Builder和Buffer类字符串、Charsequence字符序列、字符数组、布尔值、字符、整数、长整数和浮点数等。其功能是将参数的字符串形式连接到当前字符串的尾部并返回。**与String的类似操作不同的是，Buffer和Builder执行操作后，返回的不是一个新创建的对象，而是原来的对象。** 从抽象Builder类中的任一append方法定义就可以看出：

```
public AbstractStringBuilder append(String str) {
    if (str == null)
        return appendNull();
    int len = str.length();
    ensureCapacityInternal(count + len);
    str.getChars(0, len, value, count);
    count += len;
    return this; // 返回的是原本的对象
}
```

return this表示这个方法返回的是原来的对象。 此外，append支持连接Unicode的CodePoint，也可以通过指定offset和length的方式截取用作连接的片段。

delete方法

```
// StringBuffer和StringBuilder
@Override
public synchronized StringBuffer delete(int start, int end) {
    toStringCache = null;
    super.delete(start, end);
    return this;
}
@Override
public synchronized StringBuffer deleteCharAt(int index) {
    toStringCache = null;
    super.deleteCharAt(index);
    return this;
}
```

这些方法实现的是删除和更改Buffer或Builder字符串中的字符。他们返回的都是修改后的原对象本身。

substring及相关方法

```
// 仅StringBuffer重写
@Override
public synchronized String substring(int start) {
    return substring(start, count);
}
@Override
public synchronized CharSequence subSequence(int start, int end) {
    return super.substring(start, end);
}
@Override
public synchronized String substring(int start, int end) {
    return super.substring(start, end);
}
```

这些方法只有StringBuffer进行了多线程重写，实际上Builder也有相关功能。它支持指定截取的片段，也支持返回不同类型的结果（String和CharSequence）。

insert方法

```
// StringBuffer和StringBuilder
@Override
public synchronized StringBuffer insert(int offset, Object obj);
@Override
public synchronized StringBuffer insert(int offset, String str);
@Override
public synchronized StringBuffer insert(int offset, char[] str);
@Override
public StringBuffer insert(int dstOffset, CharSequence s);
@Override
public synchronized StringBuffer insert(int dstOffset, CharSequence s,
    int start, int end);
@Override
public  StringBuffer insert(int offset, boolean b);
@Override
public synchronized StringBuffer insert(int offset, char c);
@Override
public StringBuffer insert(int offset, int i);
@Override
public StringBuffer insert(int offset, long l);
@Override
public StringBuffer insert(int offset, float f);
@Override
public StringBuffer insert(int offset, double d);
```

这些方法实现了Buffer和Builder的另一个基本功能：插入。该方法接收一个参数offset表征插入位置，再接收一个对象作为插入主体。这个插入主体的类型可以是一般对象、String字符串、Builder和Buffer类字符串、Charsequence字符序列、字符数组、布尔值、字符、整数、长整数和浮点数等。`insert(int offset, String str)`方法在抽象类中的定义如下：

```
public AbstractStringBuilder insert(int offset, String str) {
    if ((offset < 0) || (offset > length()))
        throw new StringIndexOutOfBoundsException(offset);
    if (str == null)
        str = "null";
    int len = str.length();
    ensureCapacityInternal(count + len);
    System.arraycopy(value, offset, value, offset + len, count - offset);
    str.getChars(value, offset);
    count += len;
    return this;
}
```

该方法调用ensureCapacityInternal方法扩展Buffer/Builder存储空间，然后调用arrayCopy方法将尾部字符串向后移动腾出空间，最后使用getChars方法将待插入字符串放入指定位置。最后返回原对象。

indexOf方法

```
// StringBuffer和StringBuilder
@Override
public int indexOf(String str) {
```

```
    // Note, synchronization achieved via invocations of other StringBuffer methods
    return super.indexOf(str);
}
@Override
public synchronized int indexOf(String str, int fromIndex) {
    return super.indexOf(str, fromIndex);
}
@Override
public int lastIndexOf(String str) {
    // Note, synchronization achieved via invocations of other StringBuffer methods
    return lastIndexOf(str, count);
}
@Override
public synchronized int lastIndexOf(String str, int fromIndex) {
    return super.lastIndexOf(str, fromIndex);
}
```

这些方法获取某一字符串在Buffer/builder中首次/最后一次出现的位置。并可指定判断的起始位置。该方法的判断逻辑参见String类的indexOf和lastIndexOf的实现。

reverse方法

```
@Override
public synchronized StringBuffer reverse() {
    toStringCache = null;
    super.reverse();
    return this;
}
```

reverse方法可以将Buffer或Builder内的字符倒序，并且返回原对象。相关实现代码如下：

```
public AbstractStringBuilder reverse() {
    boolean hasSurrogates = false;
    int n = count - 1;
    for (int j = (n-1) >> 1; j >= 0; j--) {
        int k = n - j;
        char cj = value[j];
        char ck = value[k];
        value[j] = ck;
        value[k] = cj;
        if (Character.isSurrogate(cj) ||
            Character.isSurrogate(ck)) {
            hasSurrogates = true;
        }
    }
    if (hasSurrogates) {
        reverseAllValidSurrogatePairs();
    }
    return this;
}
```

该方法是通过依次交换字符串两端的字符实现倒序的。同时该方法也支持Unicode Surrogate 编码。

toString方法，toStringCache

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

toString方法通过直接调用String类的构造方法创建一个和Buffer或Builder内容相同的String对象。 这里我们可以看出toStringCache的作用原理：可以看出，如果多次连续调用toString方法的时候由于这个字段的缓存就可以少了Arrays.copyOfRange的操作。(每次调用其他的修改StringBuffer对象的方法时，这些方法的第一步都会先将toStringCache设置为null，如上面的append方法等等）new String中的第二个参数true代表每次调用toString方法返回的String对象共享toStringCache的值。String支持share的构造函数的注释中也提到：shares value array for speed。

writeObject方法与readObject方法

```
// StringBuffer和StringBuilder
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    s.defaultWriteObject();
    s.writeInt(count);
    s.writeObject(value);
}
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    s.defaultReadObject();
    count = s.readInt();
    value = (char[]) s.readObject();
}
```

readObject的功能是从流中恢复Buffer/Builder的状态。writeObject的功能是将当前Buffer/Builder的状态存入流中。

可序列化字段（serialPersistentFields）

```
// 仅StringBuffer
private static final java.io.ObjectStreamField[] serialPersistentFields =
{
    new java.io.ObjectStreamField("value", char[].class),
    new java.io.ObjectStreamField("count", Integer.TYPE),
    new java.io.ObjectStreamField("shared", Boolean.TYPE),
};
```

该成员为StringBuffer的可序列化字段。

2. 对比

根据源码分析的结果，我将String，StringBuffer和StringBuilder的特性总结如下：

选项	String	StringBuffer	StringBuilder
作用	存储字符串	存储与修改字符串	存储与修改字符串
底层数据结构	字符数组	字符数组和寄存器	字符数组
使用场景	单线程和多线程	主要用于多线程	仅用于单线程
基本属性	长度	长度与容量	长度与容量
可否修改（删、改字符）	否，如果要修改必须创建新的String对象	是	是
可否扩展大小	否	是，但长度超过当前数组容量后效率会降低	是，但长度超过当前数组容量后效率会降低
和常见数据类型的转换	支持	支持	支持
数据读取效率	中	中	中
数据操作效率	低，往往需要重新建立对象	较高	单线程环境下比StringBuffer高
默认操作种类	较少	多	多
正则匹配	支持	支持	支持

总的来说，如果需要在单线程环境下高效地操作数据，可以选择StringBuilder，多线程环境下，为确保安全，可以选择StringBuffer；如果对字符串操作地需求较小，或额外开销可忽略不计，最好选择使用方便、易于上手、且在单线程、多线程环境下都适用的String。

3. 问题

```
String s1 = "Welcome to Java";
 String s2 = new String("Welcome to Java");
 String s3 = "Welcome to Java";
 System.out.println("s1 == s2 is " + (s1 == s2)); System.out.println("s1 == s3 is " + (s1
== s3));
 为什么s1==s2 返回false，而s1==s3返回true?
```

答：s1定义时，"Welcome to Java"被放入常量池中；s3在定义时，由于s1的生命周期尚未结束，常量池中的"Welcome to Java"还存在，因此s3直接指向常量池中的"Welcome to Java"，而s2对应的字符串是保存在堆上的，地址同s1和s3的不同，且由于常量池中已经存在"Welcome to Java"，所以新分配的字符串所在堆区域不被算作常量池的一部分（因此，如果之前没有s1的定义，s3将使用s2的地址）所以s1=s2返回false，而s1=s3返回

true。