Java应用技术课程实验报告

姓名：应承峻

学号：3170103456

程序阅读：阅读String、StringBuffer与StringBuilder类的源程序，分析比较三个类的结构、功能设置、相同与不同。

`String`类和`StringBuffer`、`StringBuilder`一样，都由`final`关键字进行修饰，无法对类进行继承。

```
public final class String implements Serializable, Comparable, CharSequence
```

`String`类的结构与`StringBuffer`和`StringBuilder`一样，都通过字符数组`value`来存储字符串的值，并提供了一系列的方法。但是`String`类的`value`属性是由`final`修饰的。

```
/*String*/
final char[] value;
/*StringBuffer & StringBuilder */
char[] value;
```

因此`String`类的实例是不可变的，当我们在拼接字符串时，`String`类实际上是销毁了"abc"这个对象，并且重新创建了"abcd"这一个对象。如果在程序中我们对字符串进行大量操作时，`String`类的效率会非常低。因此为了提高字符串修改的效率，减少内存空间的消耗，我们需要使用`StringBuffer`和`StringBuilder`类。

```
s = "abc";
s += "d";
```

每一个`String`类的实例都有一个唯一对应的哈希值：该哈希值主要用来确定字符串对象的存储地址，以提高查询效率。

```
/**
   * Caches the result of hashCode().  If this value is zero, the hashcode
   * is considered uncached (even if 0 is the correct hash value).
   */
private int cachedHashCode;

/**
   * Holds the starting position for characters in value[].  Since
   * substring()'s are common, the use of offset allows the operation
   * to perform in O(1). Package access is granted for use by StringBuffer.
   */
final int offset;

/**
   * Computes the hashcode for this String. This is done with int arithmetic,
   * where ** represents exponentiation, by this formula:<br>
   * <code>s[0]*31**(n-1) + s[1]*31**(n-2) + ... + s[n-1]</code>.
   *
   * @return hashcode value of this String
   */
public int hashCode()
```

```
{
    if (cachedHashCode != 0)
        return cachedHashCode;

    // Compute the hash code using a local variable to be reentrant.
    int hashCode = 0;
    int limit = count + offset;
    for (int i = offset; i < limit; i++)
        hashCode = hashCode * 31 + value[i];
    return cachedHashCode = hashCode;
}
```

`String`类可以通过如下构造函数进行构造：这里值得学习的是其对代码的重复利用

```
/*构造一个空字符串*/
public String() {}

/*从原有字符串拷贝构造一个新字符串*/
public String(String str) {}

/*通过字符数组转换成字符串*/
public String(char[] data)
{
    this(data, 0, data.length, false);  //调用了其构造函数，提高了代码的利用率
}

/*从字符数组的第offset位置处的count个字符转换成字符串*/
public String(char[] data, int offset, int count)
{
    this(data, offset, data.count, false)
}

/*通过ASCII字符进行构造*/
public String(byte[] ascii, int hibyte, int offset, int count) {}
public String(byte[] ascii, int hibyte) {}
public String(byte[] data, int offset, int count, String encoding) {}
public String(byte[] data, String encoding) {}
public String(byte[] data, int offset, int count) {}
public String(byte[] data)

/*通过已有的StringBuffer类进行构造*/
/*通过StringBuffer构造时使用的synchronized我们可以看出StringBuffer是线程安全的*/
public String(StringBuffer buffer)
{
    synchronized (buffer)
    {
        offset = 0;
        count = buffer.count;
        // Share unless buffer is 3/4 empty.
        if ((count << 2) < buffer.value.length)
        {
            value = new char[count];
            VMSystem.arraycopy(buffer.value, 0, value, 0, count);
        }
        else
        {
            buffer.shared = true;
            value = buffer.value;
        }
    }
}

/*通过已有的StringBuilder类进行构造*/
public String(StringBuilder buffer)
```

```
{
    this(buffer.value, 0, buffer.count);
}

String(char[] data, int offset, int count, boolean dont_copy) {}
```

`String`类主要实现的一些功能：

```
public int length();  //得到字符串的长度
public char charAt(int index);  //获取字符串位于index处的字符
public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin); //将字符串的srcBegin处至srcEnd处的子串拷贝到dst数组中的起始处dstBegin
public void getBytes(int srcBegin, int srcEnd, byte dst[], int dstBegin);
public byte[] getBytes(String enc);
public boolean equals(Object anObject); //判断对象是否与字符串相等
public boolean contentEquals(StringBuffer buffer);  //判断序列是否一致
public boolean equalsIgnoreCase(String anotherString); //忽略大小写比较字符串
public int compareTo(String anotherString);  //比较函数，返回两串字典序差值
public int compareTo(Object o);  //重载compare
public int compareToIgnoreCase(String str);  //比较函数，返回两串字典序差值，忽略大小写
public boolean regionMatches(int toffset, String other, int ooffset, int len);
//比较该字符串从toofset处的len个字符是否与字符串other从ooffset处的len个字符相等
public boolean regionMatches(boolean ignoreCase, int toffset,String other, int ooffset, int len); //带是否忽略大小写的regionMatcher方法
public boolean startsWith(String prefix, int toffset); //判断字符串第toffset处是否是以prefix起始
public boolean startsWith(String prefix); //判断字符串是否是以prefix起始
public boolean endsWith(String suffix); //判断字符串是否是以suffi结束
public int indexOf(int ch); //找到字符ch首次出现的位置，若没出现则返回-1
public int indexOf(int ch, int fromIndex); //找到字符ch从formIndex开始首次出现的位置
public int lastIndexOf(int ch); //找到字符ch最后一次出现的位置
public int lastIndexOf(int ch, int fromIndex);//找到字符ch从formIndex开始最后一次出现出现的位置
public int indexOf(String str); //找到字符串第一次出现的位置
public int indexOf(String str, int fromIndex);
public int lastIndexOf(String str); //找到字符串最后一次出现的位置
public int lastIndexOf(String str, int fromIndex);
public String substring(int begin);  //截取从begin开始到字符串末尾的子串
public String substring(int beginIndex, int endIndex); //截取从beginIndex到endIndex的子串
public CharSequence subSequence(int begin, int end);
public String concat(String str); //将str字符串追加在字符串末尾
public String replace(char oldChar, char newChar);  //字符替换
public boolean matches(String regex);  //正则匹配
public String replaceFirst(String regex, String replacement);  //正则替换
public String replaceAll(String regex, String replacement);
public String[] split(String regex, int limit);  //字符串分割
public String[] split(String regex);
public String toLowerCase(Locale loc);  //转换成小写
public String toLowerCase();
public String toUpperCase(Locale loc);  //转换成大写
public String toUpperCase();
public String trim();  //去除首尾小于\u0020的字符，可以用作空格去除
public String toString();
public char[] toCharArray();  //转换成字符数组
public static String valueOf(Object obj);  //将数据类型转换成对应字符串形式
public static String valueOf(char[] data);
public static String valueOf(char[] data, int offset, int count);
public static String copyValueOf(char[] data, int offset, int count);
public static String copyValueOf(char[] data);
public static String valueOf(boolean b);
public static String valueOf(char c);
public static String valueOf(int i);
public static String valueOf(long l);
```

```
public static String valueOf(float f);
public static String valueOf(double d);
```

`StringBuffer`类、`StringBuilder`类与`String`类的功能基本一致，但是`StringBuffer`类的大部分函数都由`synchronized`关键字修饰。在`java`中，`synchronized`修饰的的代码在被线程执行之前，会去尝试获取一个对象的锁，如果成功，就进入并顺利执行代码，否则将会被阻塞在该对象上。因此`StringBuffer`类在多线程执行时，不会影响到最终的结果，能够保持对象的原子性，因此该类是线程安全的。而`StringBuilder`类没有`synchronized`关键字修饰，因此在并发执行时可能会造成结果不符合实际结果的情况，因此是线程不安全的。

`StringBuffer`和`StringBuilder`的相同之处在于他们都是可变的字符序列，且类的方法基本一致。但`String`是不可变字符序列。

但是从执行效率上来看`String`最慢，`StringBuffer`其次`StringBuilder`最快。

做一个总结：

- 对字符串做少量操作时用`String`
- 单线程操作大量数据使用`StringBuilder`
- 多线程操作大量数据使用`StringBuffer`