Homework 3

Requirements

- 对String、StringBuilder以及StringBuffer进行源代码分析
  - 分析其主要数据组织及功能实现，有什么区别。
  - 说明为什么这样设计，这么设计对它们的影响。
  - 它们分别使用哪些场景。
- Question

```
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));
```

为什么结果返回的是false和true？

数据组织以及功能实现

对比1

String

```
//String
public final class String
{
    private final char value[];
    /*omitted*/
    public String(String original) {
          /*omitted*/
     }
    /*omitted*/
}
```

StringBuffer

```
//StringBuffer
public final class StringBuffer extends AbstractStringBuilder
{
    char value[]; //继承了父类AbstractStringBuilder中的value[]
    public StringBuffer(String str) {
        super(str.length() + 16); //继承父类的构造器，并创建一个大小为str.length()+16的value[]数组
        append(str); //将str切分成字符序列并加入到value[]中
    }
}
```

Noted: StringBuilder与StringBuffer类似。

分析：

- String和StringBuffer中，用字符串数组value[]储存字符串序列。
- 但是String中的是不可变的常量(final)数组;而StringBuffer中的数组是一个普通的数组，值得注意的是，通过StringBuffer构造的字符串长度会预留16个位置，为后续的append操作预留空间。

对比2

StringBuffer

```
public final class StringBuffer extends AbstractStringBuilder
{
    /*omitted*/
    public synchronized int length(){/*omitted*/}
    public synchronized int capacity(){/*omitted*/}
    ...
    public synchronized StringBuffer append(/*omitted*/){/*omitted*/}
    public synchronized int indexOf(/*omitted*/){/*omitted*/}
    ...
}
```

StringBuilder

```
public final class StringBuilder extends AbstractStringBuilder
{
    /*omitted*/
    public StringBuilder append(/*omitted*/){/*omitted*/}
    ...
    public int indexOf(/*omitted*/){/*omitted*/}
}
```

分析：

- 源代码中，StringBuffer的大部分方法都被关键字**synchronized**修饰，而在StringBuilder中却没有。
- 在操作系统中进程线程同步管理通过semaphore机制实现，**synchronized**功能相似： 每一个类对象都对应一把锁，当某个线程A调用类对象O中的synchronized方法M时，必须获得对象O的锁才能够执行M方法，否则线程A阻塞。一旦线程A开始执行M方法，将独占对象O的锁。使得其它需要调用O对象的M方法的线程阻塞。只有线程A执行完毕，释放锁后。那些阻塞线程才有机会重新调用M方法。
- 由此可见StringBuffer在线程安全性方面优于StringBuilder。

对比3

String、StringBuilder、StringBuffer三者的执行效率比较

```
package test01;

public class string_test {

    private static int time = 50000;
    public static void main(String[] args) {
        testString();
        testStringBuffer();
        testStringBuilder();
```

```
        test1String();
        test2String();
    }
    public static void testString () {
        String s="";
        long begin = System.currentTimeMillis();
        for(int i=0; i<time; i++){
            s += "java";
        }
        long over = System.currentTimeMillis();
        System.out.println("Operation: "+s.getClass().getName()+"->time:"+(over-begin)+"ms");
    }
    public static void testStringBuffer () {
        StringBuffer sb = new StringBuffer();
        long begin = System.currentTimeMillis();
        for(int i=0; i<time; i++){
            sb.append("java");
        }
        long over = System.currentTimeMillis();
        System.out.println("Operation: "+sb.getClass().getName()+"->time:"+(over-begin)+"ms");
    }
    public static void testStringBuilder () {
        StringBuilder sb = new StringBuilder();
        long begin = System.currentTimeMillis();
        for(int i=0; i<time; i++){
            sb.append("java");
        }
        long over = System.currentTimeMillis();
        System.out.println("Operation: "+sb.getClass().getName()+"->time:"+(over-begin)+"ms");
    }
    public static void test1String () {
        long begin = System.currentTimeMillis();
        for(int i=0; i<time; i++){
            String s = "I"+"love"+"java";
        }
        long over = System.currentTimeMillis();
        System.out.println("string plus directly: "+(over-begin)+"ms");
    }
    public static void test2String () {
        String s1 ="I";
        String s2 = "love";
        String s3 = "java";
        long begin = System.currentTimeMillis();
        for(int i=0; i<time; i++){
            String s = s1+s2+s3;
        }
        long over = System.currentTimeMillis();
        System.out.println("refrence plus: "+(over-begin)+"ms");
    }
}
```

运行结果

```
<terminated> string_test [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (2017年10月23日 下午3:32:28)
Operation: java.lang.String->time:3824ms
Operation: java.lang.StringBuffer->time:3ms
Operation: java.lang.StringBuilder->time:2ms
string plus directly: 0ms
refrence plus: 7ms
```

分析

- 对于直接相加字符串"I"+"love"+"java"，效率很高，因为编译阶段就已经连接，形成一个字符串常量并指向heap中的拘留字符串对象。
  相比之下，引用相加s1+s2+s3，效率低下，因为每次相加，StringBuilder会一次调用new、append、toString方法，效率下降。
- 通常情况下，三者的效率：
  StringBuilder > StringBuffer > String

使用场景

- 不需要频繁拼接字符串的时候使用String，相反则使用StringBuffer。
- StringBuffer和StringBuilder相比，前者是线程安全的，适合多线程下使用；后者在单线程下使用，效率高于StringBuffer。

Answer of Question

```
String s1 = "Welcome to Java";//1
```

1执行后，"Welcome to Java"被放入常量池中，s1储存的地址是常量池中该对象的地址。

```
String s2 = new String("Welcome to Java");//2
```

2执行后，JVM在heap中new一块空间存放"Welcome to Java"，并把地址给s2。

```
String s3 = "Welcome to Java";//3
```

3执行后，由于"Welcome to Java"已经在常量池中，s3直接得到常量池中该对象的地址。
因此，s1和s3是指向同一块内存，而s2指向heap中不为常量池部分的一块内存。

References

http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/String.java#String[1]
http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/StringBuilder.java#StringBuilder[2]
http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/StringBuffer.java#StringBuffer.indexOf%28java.lang.String%29[3]
http://blog.csdn.net/clam_clam/article/details/6831345[4]
http://blog.csdn.net/loveyaozu/article/details/47037957[5]