程序阅读

要求：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。

初步分析：
String 类实现了 Serializable, Comparable, CharSequence 三个接口。
StringBuffer 类实现了 Serializable, CharSequence 两个接口。
StringBuilder 类实现了 Serializable, CharSequence 两个接口。

在未阅读源码之前先简单说一下自己对这三个类的初步印象，String 类是最常见最广泛使用的字符串类型，StringBuffer 是线程安全的，StringBuilder 是线程安全的，这是学习 java 基础课程中就了解到的。
在实际使用也能发现 StringBuffer 和 StringBuilder 的有点，比如要尝试将两个字符串相连接，String 类可以直接使用 + 运算将两个字符串相连接，StringBuffer 类则可以使用 append 方法来连接两个字符串，在力扣刷题过程中是可以发现使用 StringBuffer 类来连接两个字符串操作更方便，也更快捷，这说明 StringBuffer 和 StringBuilder 类相对 String 类是有优点相对优势的，这也是为什么 Java 存在多种字符串类。

具体阅读源码分析

一、String 类

实现三个接口，与 StringBuffer 和 StringBuilder 不同的是还多实现了一个接口 Comparable

在 IDEA 快捷键 ctrl + B 即可查看接口 Comparable, 里面只有一个方法 compareTo，返回类型是 Int 类型。截图如下

```
@Contract(pure = true)
public int compareTo( @NotNull T o);
```

其次 String 类是不可变类，实例在被创建后就不可被修改。没有提供任何修改对象状态的方法，保证类不会被拓展，所有的域都是 final，所有的域都是私有的，确保对于任何可变组件的互斥访问。

里面重点是缓存 hashcode
保证每次使用同一个字符串对象总是相同的，在使用中不用考虑其是否发生变化，不需要每次都去计算以便 hashcode，使程序更加高效。

二、StringBuffer 和 StringBuider

StringBuffer 字符串变量（线程安全）
StringBuilder 字符串变量（非线程安全）

StringBuilder 类在 Java 5 中被提出，它和 StringBuffer 之间的最大不同在于 StringBuilder 的方法不是线程安全的（不能同步访问）。由于 StringBuilder 相较于 StringBuffer 有速度优势，所以多数情况下建议使用 StringBuilder 类。然而在应用程序要求线程安全的情况下，则必须使用 StringBuffer 类。

当对字符串进行修改的时候，特别是字符串对象经常改变的情况下，需要使用 StringBuffer 和 StringBuilder 类。
和 String 类不同的是，StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。、

从 StringBuffer.m 网页代码截图来看，我们可以发现里面的子类都有添加 synchronzied 这一关键字，这是一个同步锁。而在与之相似的 StringBuider.m 中则没有添加 sychronzied 这一关键字，所以这是线程不安全的。

下面分别是 StringBuffer 和 StringBuider 的截图，观察对比其不同。

```
 */
public synchronized int capacity()
{
  return value.length;
}

/**
 * Increase the capacity of this <code>StringBuffer</code>. This will
 * ensure that an expensive growing operation will not occur until
 * <code>minimumCapacity</code> is reached. The buffer is grown to the
 * larger of <code>minimumCapacity</code> and
 * <code>capacity() * 2 + 2</code>, if it is not already large enough.
 *
 * @param minimumCapacity the new capacity
 * @see #capacity()
 */
public synchronized void ensureCapacity(int minimumCapacity)
{
  ensureCapacity_unsynchronized(minimumCapacity);
}
```

```
187  public int capacity()
188  {
189    return value.length;
190  }
191
192  /**
193   * Increase the capacity of this <code>StringBuilder</code>. This will
194   * ensure that an expensive growing operation will not occur until
195   * <code>minimumCapacity</code> is reached. The buffer is grown to the
196   * larger of <code>minimumCapacity</code> and
197   * <code>capacity() * 2 + 2</code>, if it is not already large enough.
198   *
199   * @param minimumCapacity the new capacity
200   * @see #capacity()
201   */
202  public void ensureCapacity(int minimumCapacity)
203  {
204    if (minimumCapacity > value.length)
205      {
206        int max = value.length * 2 + 2;
207        minimumCapacity = (minimumCapacity < max ? max : minimumCapacity);
208        char[] nb = new char[minimumCapacity];
209        System.arraycopy(value, 0, nb, 0, count);
210        value = nb;
211      }
212  }
213
```

这样就保证了同一时刻最多只有一个线程执行同步代码，能保证多线程坏境下并发安全的效

果。