String 的基本属性和特征

String的主要属性

```
1. private final char[] value
2.     //以私有数组常量的形式记录了构成string的字符。由final易知String类不允许被其他类继承，并且被构造之后就不能再进行修改
3. private final int count;
4.     //count是用于表示string中字符数的常量。这里可见长度也是固定值，拼接string的时候需要重新生成。
5.  private final int offset;
6.     //记录字符串的起始位置，由于substring的操作比较常见，因此这样可以节约时间
7. private int hash;
8.     //存储字符串的哈希值
```

String的函数

构造函数

```
1. public String()
2. {
3.   this.value = "".value;
4. }
```

无参数构造字符串，初始化为空，长度和偏移量均初始化为0。

```
1. public String(String original)
2. {
3.   this.value = original.value;
4.   this.hash = original.hash;
5. }
```

拷贝构造函数，重新创建一个相同的String对象。

```
1. public String(char value[])
2. {
3.     this.value = Arrays.copyOf(value, value.length);
4. }
```

用字符数组构造String对象。

```
1. public String(StringBuffer buffer)
2. {
3.     synchronized(buffer)
4.         {
5.         this.value = Arrays.copyOf(buffer.getValue(), buffer.length());
6.         }
7. }
```

用StringBuffer类的对象构造String类对象，调用copyOf的方法，输入StringBuffer类的value和length即可。

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

用StringBuilder类的对象构造String类对象，同理调用copyOf的方法，输入StringBuilder类的value和length即可。

```
String(char[] data, int offset, int count, boolean dont_copy)
  {
    if (offset < 0 || count < 0 || offset + count > data.length)
      throw new StringIndexOutOfBoundsException();
    if (dont_copy)
      {
        value = data;
        this.offset = offset;
      }
    else
      {
        value = new char[count];
        VMSystem.arraycopy(data, offset, value, 0, count);
        this.offset = 0;
      }
    this.count = count;
  }
```

用字符数组构造一个String类对象，offset和count即新构造的String类属性，dont_copy的含义是是否要构造一个新的String类对象，还是作为输入的字符数组的substring。

length方法

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

得到string的长度，直接返回属性length

charAt方法

```
public char charAt(int index){
  if((index < 0) || (index >= value.length)){
    throw new StringIndexOutOfBoundsException(index);
  }
  return value[index];
}
```

返回特定下标的字符，首先判断下标是否合法，如果超过长度或为负数则抛出异常。然后返回输入下标的值

```
1. public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin)
2.   {
3.     if (srcBegin < 0 || srcBegin > srcEnd || srcEnd > count)
4.       throw new StringIndexOutOfBoundsException();
5.     VMSystem.arraycopy(value, srcBegin + offset,
6.                        dst, dstBegin, srcEnd - srcBegin);
7.   }
```

拷贝String的一部分，下标从srcBegin到srcEnd-1的子串到dst中下标从dstBegin开始的位置。

replace方法

```
1. public String replace(char oldChar, char newChar){
2.   if(oldChar != newChar){
3.     int len = value.length;
4.     int i = -1;
5.     char[] val = value; /*avoid get field opcode*/
6.     while (++i < len){
7.       if (val[i] == oldChar){
8.         break;
9.       }
10.     }
11.     if( i < len ){
12.       char buf[] = new char[len];
13.       for (intj=0; j<i; j++){
14.         buf[j] = val[j];
15.       }
16.       while (i < len){
17.         char c = val[i];
18.         buf[i] = (c == oldChar) ? newChar : c;
19.         i++;
20.       }
21.       return new String(buf,true);
22.     }
23.    }
24.   return this;
25. }
```

把原来String中的oldchar都替换成新的newchar，由于String类型无法直接更改的性质，同样也创建了一个新的String依次把字符数组复制过去，遇到oldchar的时候就用newchar代替。

compare方法

```
1. public int compare(Object o1, Object o2)
```

比较两个字符串，忽略大小写

```
1. public boolean equals(Object anObject) {
2.      if (this == anObject) {
3.          return true;
4.      }
5.     if (anObject instanceof String) {
6.        String anotherString = (String) anObject;
7.        int n = value.length;
8.        if (n == anotherString.value.length) {
```

```
 9.             char v1[] = value;
10.             char v2[] = anotherString.value;
11.             int i = 0;
12.             while (n-- != 0) {
13.               if (v1[i] != v2[i])
14.               return false;
15.               i++;
16.             }
17.             return true;
18.         }
19.     }
20.     return false;
21. }
```

比较两个字符串，大小写敏感。首先判断是不是相同的两个object，是则直接返回true；如果不是再判断object是不是String类型的实例，不是则返回false；再比较两个String的长度，不相等则返回false；最后才是逐字比较字符数组的每一项。这样可以节约很多判断的时间。

```
 1. public boolean contentEquals(StringBuffer buffer)
 2.   {
 3.     synchronized (buffer)
 4.       {
 5.         if (count != buffer.count)
 6.           return false;
 7.         if (value == buffer.value)
 8.           return true; // Possible if shared.
 9.         int i = count;
10.         int x = offset + count;
11.         while (--i >= 0)
12.           if (value[--x] != buffer.value[i])
13.             return false;
14.         return true;
15.       }
16.   }
```

区分大小写地比较stringbuffer和string的内容是否相同。首先判断两者的长度是否相同，不同则直接返回false；然后判断两者是否指向同一个String，是的话直接返回true。否则再两两对比依次比较。这样在一些特殊情况下可以节约很多时间。

```
1. public boolean equalsIgnoreCase(String anotherString) {
2.  return (this == anotherString) ? true : (anotherString != null) && (anotherString.val
   ue.length == value.length) && regionMatches(true, 0, anotherString, 0, value.length);
3.  }
```

忽略大小写，比较两个string是否相等 同样还是先判断anotherString和本身是不是指向相同的String，是则直接返回true；否则以此判断anotherString是否空，两者字符数是否相等，两者逐一字符数组是否相等来返回结果。由于&&操作符的特性，当顺序判断有一项不符合时就直接返回false，不再进行后续的判断，也能节约时间。

```
1. public String concat(String str) {
2.  int otherLen = str.length();
3.  if (otherLen == 0) {
4.      return this;
5.  }
6.  char buf[] = new char[count + otherLen];
7.  getChars(0, count, buf, 0);
```

```
8.  str.getChars(0, otherLen, buf, count);
9.  return new String(0, count + otherLen, buf);
10.     }
```

concat的功能是连接两个String。首先创建一个长度为需要拼接的string之和的string，然后拼接完成之后重新构造一个String后返回。

StringBuffer 的基本属性和特征

StringBuffer的主要属性包括

```
1. int count;
2. char[] value;
3. boolean shared;
4. //shared表示此StringBuffer在修改之前是否需要先拷贝
```

StringBuffer的主要函数包括

构造函数

```
1.   public StringBuffer(String str)
2.   {
3.     // Unfortunately, because the size is 16 larger, we cannot share.
4.     count = str.count;
5.     value = new char[count + DEFAULT_CAPACITY];
6.     str.getChars(0, count, value, 0);
7.   }
```

用String类的对象构造StringBuffer类的对象，两者的count数相等，为value申请空间后进行拷贝。

length和capacity

```
1.  public synchronized int capacity()
2.   {
3.     return value.length;
4.   }
5.   public synchronized int length()
6.   {
7.     return count;
8.   }
```

由上面的源码可知，capacity和length的区别在于capacity是该StringBuffer最大可以表示的字符串长度，lenght则是当前字符串的长度。

```
1.   public synchronized void setLength(int newLength)
2.   {
3.     if (newLength < 0)
4.       throw new StringIndexOutOfBoundsException(newLength);
5.     int valueLength = value.length;
```

```
    ensureCapacity_unsynchronized(newLength);
    if (newLength < valueLength)
    {
        count = newLength;
    }
    else
    {
        while (count < newLength)
          value[count++] = '\\0';
    }
  }
```

设置StringBuffer的长度。首先判断是否为负数，是则抛出异常；然后分成设置的长度小于当前长度和大于当前长度两种情况来处理，前者在结尾补\0表示字符串结束并；后者则补\0直到字符串达到输入值的长度。

append方法

```
public synchronized StringBuffer append(char[] data, int offset, int count)
  {
    if (offset < 0 || count < 0 || offset > data.length - count)
      throw new StringIndexOutOfBoundsException();
    ensureCapacity_unsynchronized(this.count + count);
    VMSystem.arraycopy(data, offset, value, this.count, count);
    this.count += count;
    return this;
  }
```

该方法的作用是追加内容到当前StringBuffer对象的末尾，类似于字符串的连接。同样首先判断是否合法， 保证偏移量不为负，拼接的字符数不为负以及没有越界之后，用arraycopy的方法拷贝之后增加表示字符数量的count。

```
public StringBuffer append(int inum)
  {
    return append(String.valueOf(inum));
  }
```

delete方法

```
  public synchronized StringBuffer delete(int start, int end)
  {
    if (start < 0 || start > count || start > end)
      throw new StringIndexOutOfBoundsException(start);
    if (end > count)
      end = count;
    ensureCapacity_unsynchronized(count);
    if (count - end != 0)
      VMSystem.arraycopy(value, end, value, start, count - end);
    count -= end - start;
    return this;
  }
```

delete函数的功能是删除StringBuffer里面从start到end-1部分的字符串，并把之后的部分向前补全。实现方法是把StringBuffer中end以后位置的字符拷贝到start的位置，然后对应地修改count。

```
1. public StringBuffer deleteCharAt(int index)
2.   {
3.     return delete(index, index + 1);
4.   }
```

deleteCharAt可以看作delete的一个特殊情况。删除index处的字符即等价于设置delete中的start为index，end为index-1。

substring方法

```
1.   public synchronized String substring(int beginIndex, int endIndex)
2.   {
3.     int len = endIndex - beginIndex;
4.     if (beginIndex < 0 || endIndex > count || endIndex < beginIndex)
5.       throw new StringIndexOutOfBoundsException();
6.     if (len == 0)
7.       return "";
8.     if (share_buffer)
9.       this.shared = true;
10.    // Package constructor avoids an array copy.
11.    return new String(value, beginIndex, len, share_buffer);
12.  }
```

同样是判断合法之后在进行子串的拷贝，由最后的return语句可知实际上是新创建了一个String之后进行返回。

```
1.   public String substring(int beginIndex)
2.   {
3.     return substring(beginIndex, count);
4.   }
```

只输入beginIndex的substring方法也是一个特例，相当于endIndex是String的末尾，即count。

返回从beginIndex开始一直到最后的子串。

insert方法

```
1.  public synchronized StringBuffer insert(int offse, char[] str, int str_offset, int le
   n)
2.   {
3.     if (offset < 0 || offset > count || len < 0
4.         || str_offset < 0 || str_offset > str.length - len)
5.       throw new StringIndexOutOfBoundsException();
6.     ensureCapacity_unsynchronized(count + len);
7.     VMSystem.arraycopy(value, offset, value, offset + len, count - offset);
8.     VMSystem.arraycopy(str, str_offset, value, offset, len);
9.     count += len;
10.    return this;
11.  }
```

该方法的作用是在StringBuffer对象的offset位置中插入str从str_offset长度为len的子串，形成新的字符串。首先修改StringBuffer的长度，在需要插入的地方空出位置，然后用arraycopy的方法进行插入操作。

```
1.   public StringBuffer insert(int offset, char[] data)
2.   {
3.     return insert(offset, data, 0, data.length);
4.   }
```

该函数的作用是在原StringBuffer中offset的位置插入data字符串的全部，同样可以看作是insert的特例，把str_offset设置为0就达到了相同的效果。

indexOf方法

```
1.   public synchronized int indexOf(String str, int fromIndex)
2.   {
3.     if (fromIndex < 0)
4.       fromIndex = 0;
5.     int limit = count - str.count;
6.     for ( ; fromIndex <= limit; fromIndex++)
7.       if (regionMatches(fromIndex, str))
8.         return fromIndex;
9.     return -1;
10.  }
```

该函数的功能是从StringBuffer的fromIndex开始向后查找第一个匹配的str的位置，并返回下标。首先判断如果fromIndex为负则设置为0，然后开始向后遍历，返回第一次找到str的下标。

```
1.   public synchronized int lastIndexOf(String str, int fromIndex)
2.   {
3.     fromIndex = Math.min(fromIndex, count - str.count);
4.     for ( ; fromIndex >= 0; fromIndex--)
5.       if (regionMatches(fromIndex, str))
6.         return fromIndex;
7.     return -1;
8.   }
```

该函数的功能是从和indexOf相反，从StringBuffer的fromIndex开始向前查找第一个匹配的str的位置，并返回下标，也即从前往后最后一个匹配的位置。首先判断如果fromIndex为负则设置为0，然后开始向前遍历，返回第一次找到str的下标。

reverse方法

```
1.   public synchronized StringBuffer reverse()
2.   {
3.     // Call ensureCapacity to enforce copy-on-write.
4.     ensureCapacity_unsynchronized(count);
5.     for (int i = count >> 1, j = count - i; --i >= 0; ++j)
6.       {
7.         char c = value[i];
8.         value[i] = value[j];
9.         value[j] = c;
10.      }
11.    return this;
12.  }
```

该方法的作用是将StringBuffer对象中的内容反转，然后形成新的字符串。方法是用i和j两个变量分别从下标为0和下标为count-1处开始遍历，同时交换两者顺序。

以下方法和String基本相同，除了加上了synchronized。

```
1. public synchronized void getChars(int srcOffset, int srcEnd, char[] dst, int dstOffset
   )
2. public synchronized char charAt(int index)
3. public synchronized void setCharAt(int index, char ch)
```

StringBuilder的基本属性和特征

StringBuilder的主要属性包括

```
1. int count;
2. char[] value;
3. private static final int DEFAULT_CAPACITY = 16;
```

构造方法

```
1. public StringBuilder(int capacity)
2. //构建一个指定容量的空字符串
3. public StringBuilder(String str)
4. //用输入的String构建一个指定字符串的字符串生成器
5. public StringBuilder(CharSequence seq)
6. //构建一个带制定字符串的字符串生成器
```

charAt方法

```
1. public void getChars(int srcOffset, int srcEnd, char[] dst, int dstOffset)
2. public void setCharAt(int index, char ch)
```

StringBuilder的getChars、setCharAt等函数和StringBuffer几乎完全相同，除了synchronized的修饰符。

其他方法

```
1. public StringBuilder append(String str)
2. public StringBuilder append(StringBuffer stringBuffer)
3. public StringBuilder deleteCharAt(int index)
4. public StringBuilder replace(int start, int end, String str)
5. public String substring(int beginIndex)
6. public StringBuilder insert(int offset, boolean bool)
7. public int indexOf(String str, int fromIndex)
```

常规StringBuffer有的函数StringBuilder也都有，功能和实现方法都几乎相同，所以不展开分析。

String, StringBuffer和StringBuilder

String和StringBuffer

String的内容是不可变的。用教材上的代码为例：

```
1. String s = "Java";
2. s = "HTML";
```

第二行代码实际上没有改变字符串的内容。第一条语句创建了内容为"Java"的String对象，并把它的引用赋值给了s；第二条语句创建了内容为"HTML"的String对象，并把它的引用赋值给了s。但是此时内容为"Java"的String对象仍然存在，只是不能再被访问，因为变量s指向了新的对象。 但是对于StringBuffer来说，它的每次修改都会改变对象自身，这点是和String类最大的区别。

```
1. StringBuffer s = "Java";
2. s = "HTML";
```

以上代码是对s本身进行修改，而没有重新创建新的对象。

因此在选择String和StringBuffer类的时候，如果字符串需要频繁的进行拼接、插入或删除操作，就推荐使用StringBuffer，可以减少对内存的占用，并且节约时间。

StringBuffer和StringBuilder

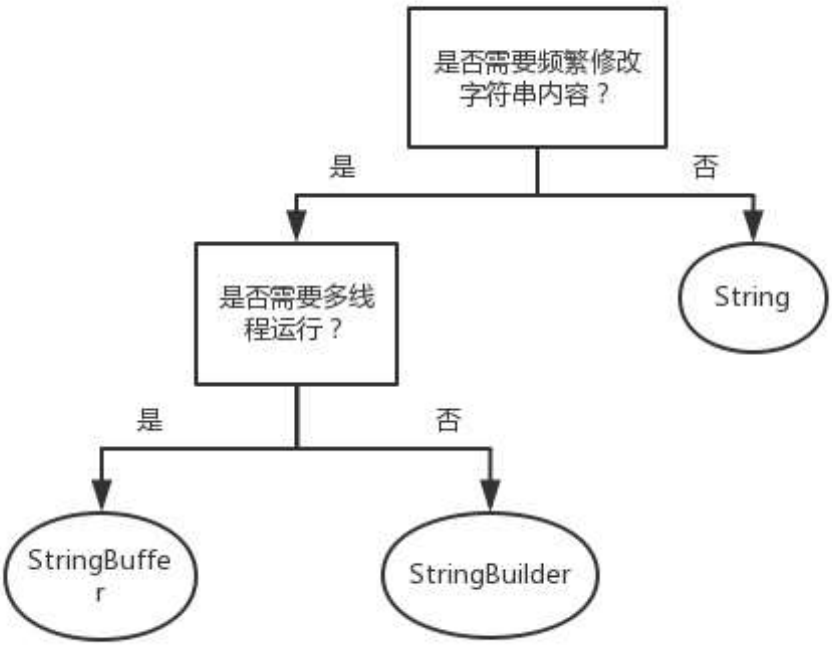
buffer和builder的主要不同主要在于，synchronized关键字作为了stringbuffer这个类里面函数的修饰符，来解决多线程共享数据同步问题。

synchronized 的存在就表示，每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁才能执行，否则所属线程阻塞；而方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程才能获得该锁，重新进入可执行状态。

这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态，从而有效避免了类成员变量的访问冲突。

因此，StringBuilder类不是线程安全的，但其在单线程中的性能比StringBuffer高。

如何选择


是否需要频繁修改字符串内容？
是
否
是否需要多线程运行？
String
是
否
StringBuffer
StringBuilder