String

1. String类的理解(以JDK8为例说明)
1.1 类的声明
```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
```

> final:String是不可被继承的
> Serializable:可序列化的接口。凡是实现此接口的类的对象就可以通过网络或本地流进行数据的传输。
> Comparable:凡是实现此接口的类，其对象都可以比较大小。

1.2 内部声明的属性：
jdk8中：
```
private final char value[]; //存储字符串数据的容器
```
    > final : 指明此value数组一旦初始化，其地址就不可变。

jdk9开始：为了节省内存空间，做了优化
```
private final byte[] value; //存储字符串数据的容器。
```

2. 字符串常量的存储位置
> 字符串常量都存储在字符串常量池(StringTable)中
> 字符串常量池不允许存放两个相同的字符串常量。
> 字符串常量池，在不同的jdk版本中，存放位置不同。
  jdk7之前：字符串常量池存放在方法区
  jdk7及之后：字符串常量池存放在堆空间。

3. String的不可变性的理解
① 当对字符串变量重新赋值时，需要重新指定一个字符串常量的位置进行赋值，不能在原有的位置修改
② 当对现有的字符串进行拼接操作时，需要重新开辟空间保存拼接以后的字符串，不能在原有的位置修改
③ 当调用字符串的replace()替换现有的某个字符时，需要重新开辟空间保存修改以后的字符串，不能在原有的位置修改

4. String实例化的两种方式
第1种方式：String s1 = "hello";
第2种方式：String s2 = new String("hello");

【面试题】
String s2 = new String("hello");在内存中创建了几个对象？ 两个!
一个是堆空间中new的对象。另一个是在字符串常量池中生成的字面量。

5. String的连接操作:+
情况1：常量 + 常量：结果仍然存储在字符串常量池中，返回此字面量的地址。注：此时的常量可能是字面量，也可能是final修饰的常量
情况2：常量 + 变量 或 变量 + 变量 ：都会通过new的方式创建一个新的字符串，返回堆空间中此字符串对象的地址
情况3：调用字符串的intern():返回的是字符串常量池中字面量的地址
(了解)情况4：concat(xxx):不管是常量调用此方法，还是变量调用，同样不管参数是常量还是变量，总之，调用完concat()方法

都返回一个新new的对象，即结果在堆中。

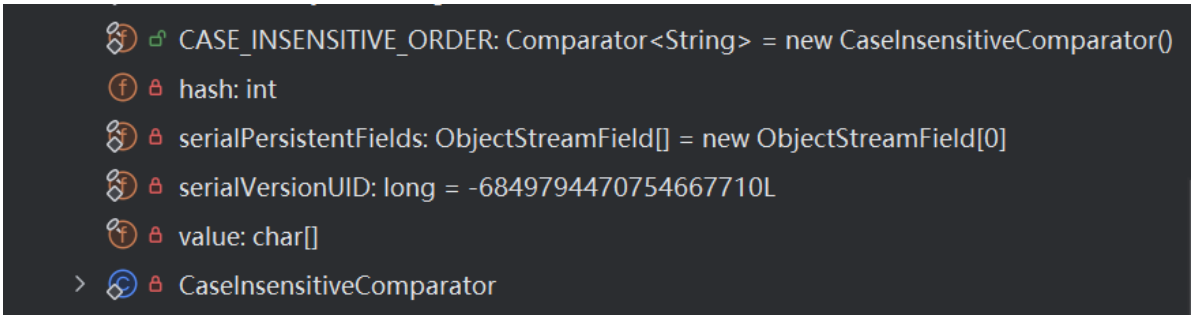
6. String的构造器和常用方法

6.1 构造器

* `public String() ` ：初始化新创建的 String对象，以使其表示空字符序列。
* `public String(String original)`： 初始化一个新创建的 `String` 对象，使其表示一个与参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的副本。
* `public String(char[] value) ` ：通过当前参数中的字符数组来构造新的String。
* `public String(char[] value,int offset, int count) ` ：通过字符数组的一部分来构造新的String。
* `public String(byte[] bytes) ` ：通过使用平台的**默认字符集**解码当前参数中的字节数组来构造新的String。
* `public String(byte[] bytes,String charsetName) ` ：通过使用指定的字符集解码当前参数中的字节数组来构造新的String。

6.2 常用方法

> String 类的常量属性




```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    private int hash; // Default to 0
    private final char value[];
    private static final ObjectStreamField[] serialPersistentFields =
            new ObjectStreamField[0];
    private static final long serialVersionUID = -6849794470754667710L;
}
```

> String类的构造方法

public String() ：初始化新创建的 String对象，以使其表示空字符序列。
String(String original)： 初始化一个新创建的 `String` 对象，使其表示一个与参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的副本。
public String(char[] value) ：通过当前参数中的字符数组来构造新的String。
public String(char[] value,int offset, int count) ：通过字符数组的一部分来构造新的String。
public String(byte[] bytes) ：通过使用平台的**默认字符集**解码当前参数中的字节数组来构造新的String。
public String(byte[] bytes,String charsetName) ：通过使用指定的字符集解码当前参数中的字节数组来构造新的String。

> String类的常用方法

（1）boolean isEmpty()：字符串是否为空
（2）int length()：返回字符串的长度
（3）String concat(xx)：拼接
（4）boolean equals(Object obj)：比较字符串是否相等，区分大小写
（5）boolean equalsIgnoreCase(Object obj)：比较字符串是否相等，不区分大小写
（6）int compareTo(String other)：比较字符串大小，区分大小写，按照Unicode编码值比较大小
（7）int compareToIgnoreCase(String other)：比较字符串大小，不区分大小写
（8）String toLowerCase()：将字符串中大写字母转为小写
（9）String toUpperCase()：将字符串中小写字母转为大写
（10）String trim()：去掉字符串前后空白符
（11）public String intern()：结果在常量池中共享

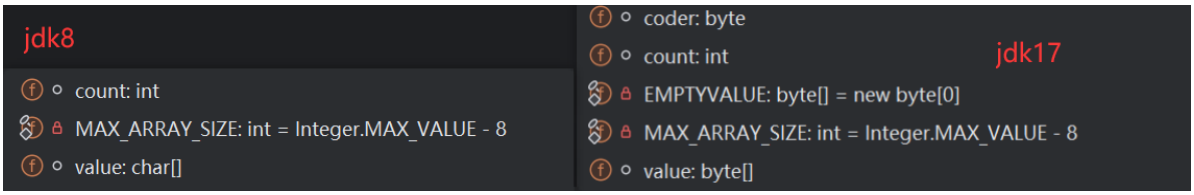
（12）boolean contains(xx)：是否包含xx
（13）int indexOf(xx)：从前往后找当前字符串中xx，即如果有返回第一次出现的下标，要是没有返回-1
（14）int indexOf(String str, int fromIndex)：返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始
（15）int lastIndexOf(xx)：从后往前找当前字符串中xx，即如果有返回最后一次出现的下标，要是没有返回-1
（16）int lastIndexOf(String str, int fromIndex)：返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索。
（17）String substring(int beginIndex) ：返回一个新的字符串，它是此字符串的从beginIndex开始截取到最后的一个子字符串。
（18）String substring(int beginIndex, int endIndex) ：返回一个新字符串，它是此字符串从beginIndex开始截取到endIndex(不包含)的一个子字符串。
（19）char charAt(index)：返回[index]位置的字符
（20）char[] toCharArray()： 将此字符串转换为一个新的字符数组返回
（22）static String valueOf(char[] data) ：返回指定数组中表示该字符序列的 String
（23）static String valueOf(char[] data, int offset, int count) ： 返回指定数组中表示该字符序列的 String
（24）static String copyValueOf(char[] data)： 返回指定数组中表示该字符序列的 String
（25）static String copyValueOf(char[] data, int offset, int count)：返回指定数组中表示该字符序列的 String
（26）boolean startsWith(xx)：测试此字符串是否以指定的前缀开始
（27）boolean startsWith(String prefix, int toffset)：测试此字符串从指定索引开始的子字符串是否以指定前缀开始
（28）boolean endsWith(xx)：测试此字符串是否以指定的后缀结束

（29）String replace(char oldChar, char newChar)：返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。 不支持正则。
（30）String replace(CharSequence target, CharSequence replacement)：使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。

（31）String replaceAll(String regex, String replacement)：使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。
（32）String replaceFirst(String regex, String replacement)：使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。

jdk8/17: AbstractStringBuilder类

> AbstractStringBuilder类的常见属性

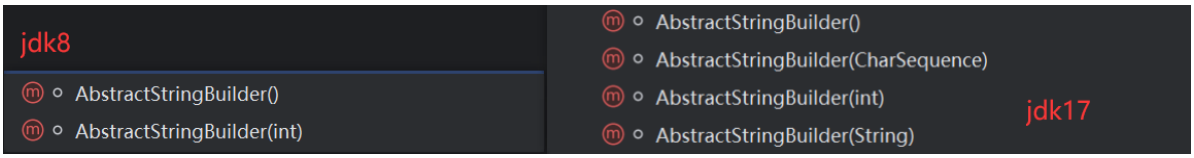



```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
    char[] value;
    int count; // 记录 value 数组中被使用的长度
    private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
}
```

> 下面是jdk17的属性分析

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
    byte[] value;
    byte coder;
    int count; // jdk9 之后加上的
    private static final byte[] EMPTYVALUE = new byte[0];
    private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
}
```

> AbstractStringBuilder类的构造方法



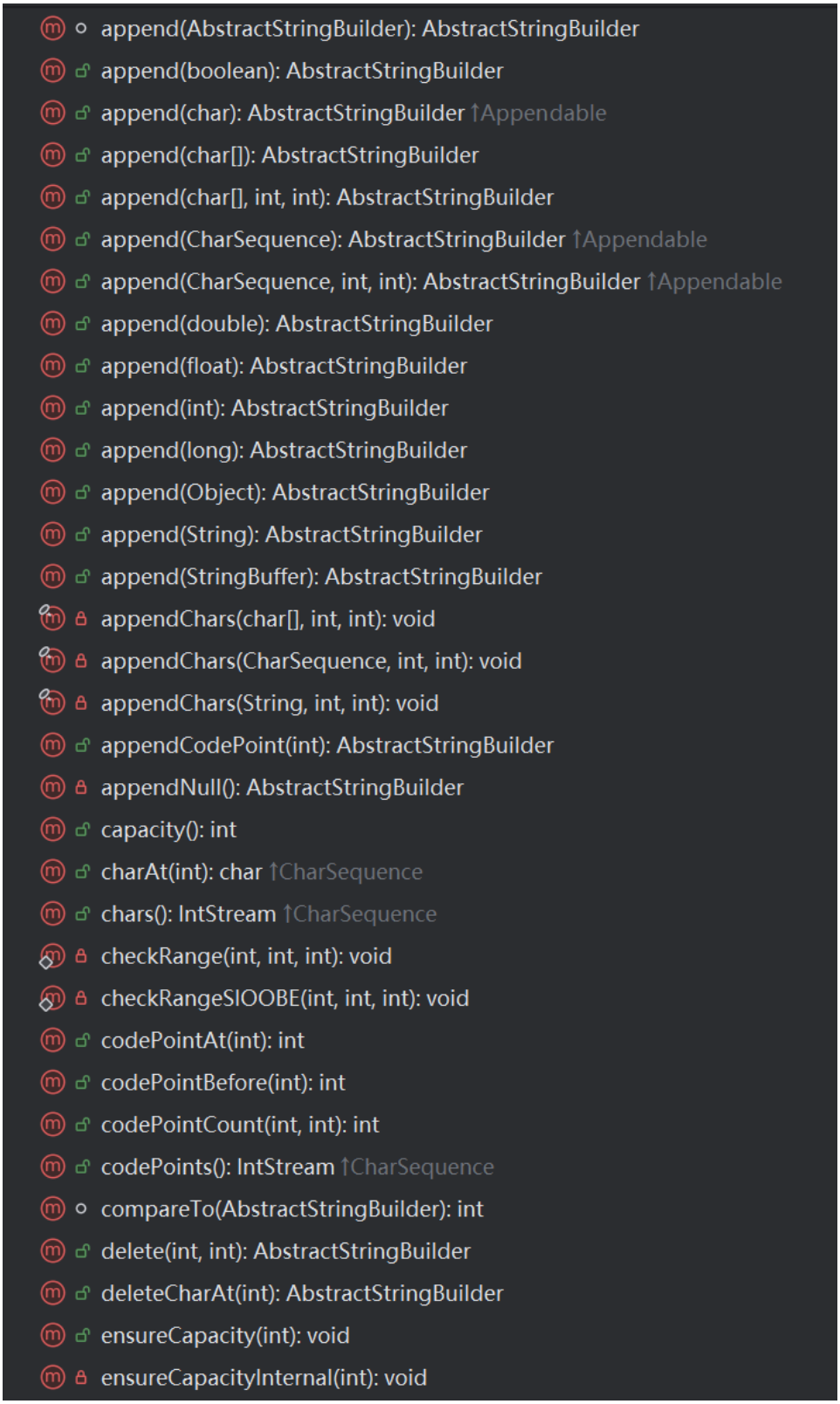

```
jdk8: AbstractStringBuilder() {
}
==========================================
AbstractStringBuilder(int capacity) {
    value = new char[capacity];
}
```

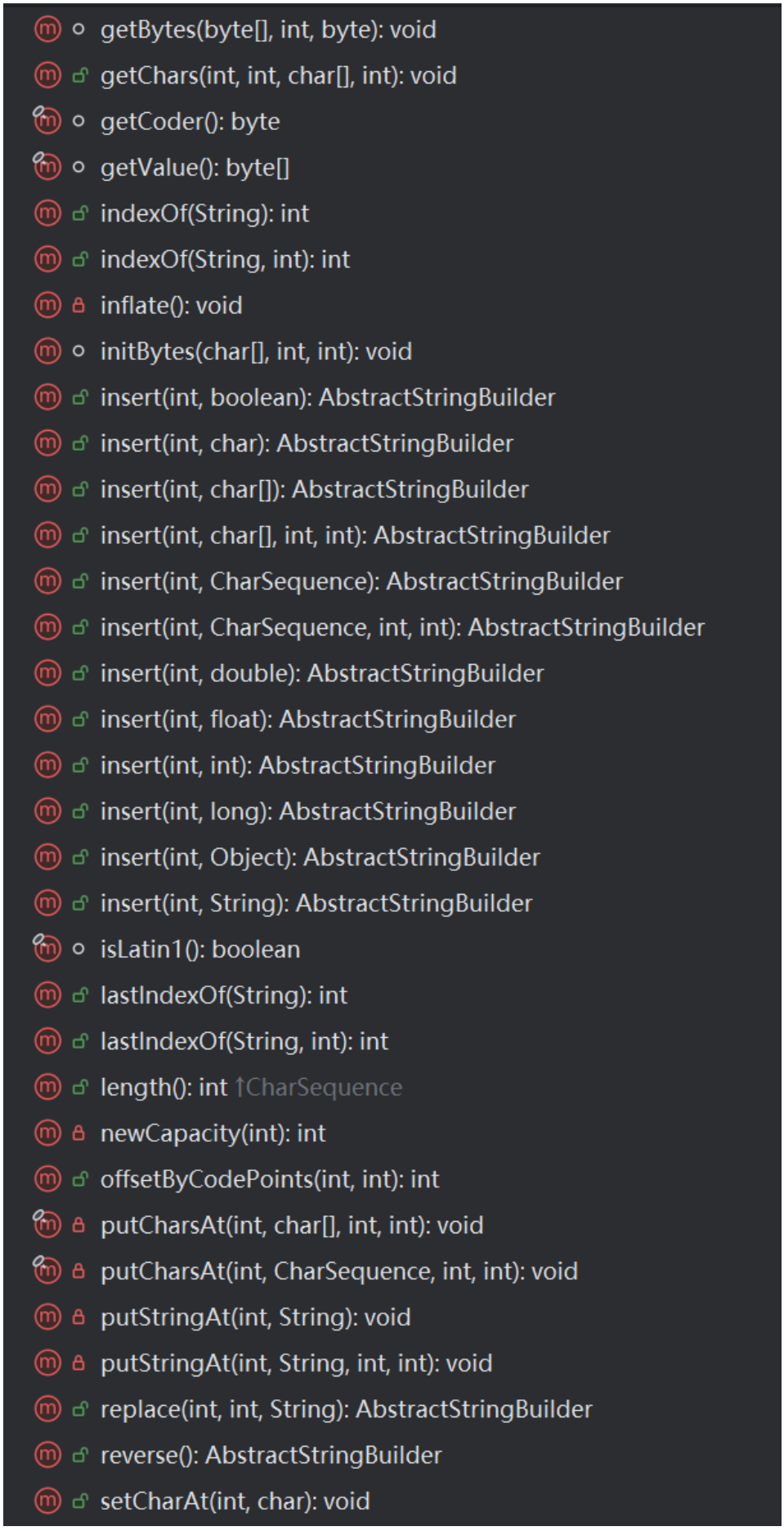
```
jdk17: AbstractStringBuilder() {
    value = EMPTYVALUE;
```

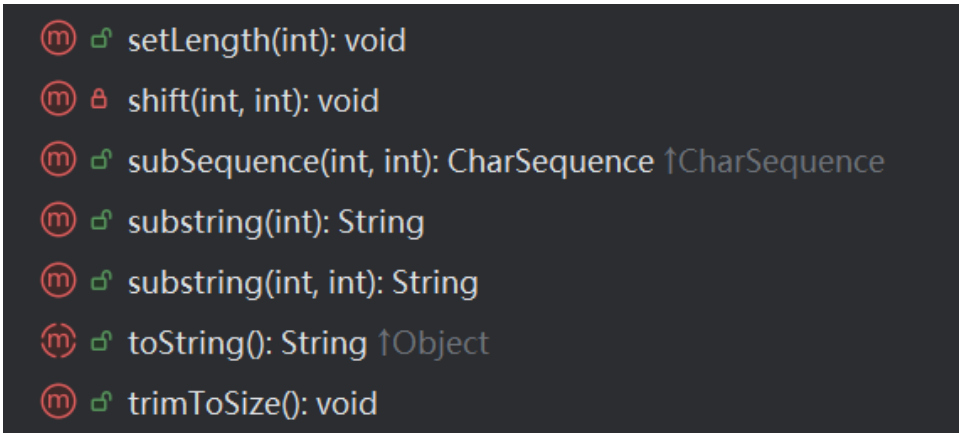
```
}
================================================
jdk17: AbstractStringBuilder(int capacity) {
    if (COMPACT_STRINGS) {
        value = new byte[capacity];
        coder = LATIN1;
    } else {
        value = StringUTF16.newBytesFor(capacity);
        coder = UTF16;
    }
}
================================================
 jdk17: AbstractStringBuilder(String str) {
    int length = str.length();
    int capacity = (length < Integer.MAX_VALUE - 16)
        ? length + 16 : Integer.MAX_VALUE;
    final byte initCoder = str.coder();
    coder = initCoder;
    value = (initCoder == LATIN1)
        ? new byte[capacity] : StringUTF16.newBytesFor(capacity);
    append(str);
}
================================================
jdk17: AbstractStringBuilder(CharSequence seq) { ..... }
```

> AbstractStringBuilder类的常用方法

```
```


append(AbstractStringBuilder): AbstractStringBuilder
append(boolean): AbstractStringBuilder
append(char): AbstractStringBuilder ↑Appendable
append(char[]): AbstractStringBuilder
append(char[], int, int): AbstractStringBuilder
append(CharSequence): AbstractStringBuilder ↑Appendable
append(CharSequence, int, int): AbstractStringBuilder ↑Appendable
append(double): AbstractStringBuilder
append(float): AbstractStringBuilder
append(int): AbstractStringBuilder
append(long): AbstractStringBuilder
append(Object): AbstractStringBuilder
append(String): AbstractStringBuilder
append(StringBuffer): AbstractStringBuilder
appendChars(char[], int, int): void
appendChars(CharSequence, int, int): void
appendChars(String, int, int): void
appendCodePoint(int): AbstractStringBuilder
appendNull(): AbstractStringBuilder
capacity(): int
charAt(int): char ↑CharSequence
chars(): IntStream ↑CharSequence
checkRange(int, int, int): void
checkRangeSIOOBE(int, int, int): void
codePointAt(int): int
codePointBefore(int): int
codePointCount(int, int): int
codePoints(): IntStream ↑CharSequence
compareTo(AbstractStringBuilder): int
delete(int, int): AbstractStringBuilder
deleteCharAt(int): AbstractStringBuilder
ensureCapacity(int): void
ensureCapacityInternal(int): void


getBytes(byte[], int, byte): void
getChars(int, int, char[], int): void
getCoder(): byte
getValue(): byte[]
indexOf(String): int
indexOf(String, int): int
inflate(): void
initBytes(char[], int, int): void
insert(int, boolean): AbstractStringBuilder
insert(int, char): AbstractStringBuilder
insert(int, char[]): AbstractStringBuilder
insert(int, char[], int, int): AbstractStringBuilder
insert(int, CharSequence): AbstractStringBuilder
insert(int, CharSequence, int, int): AbstractStringBuilder
insert(int, double): AbstractStringBuilder
insert(int, float): AbstractStringBuilder
insert(int, int): AbstractStringBuilder
insert(int, long): AbstractStringBuilder
insert(int, Object): AbstractStringBuilder
insert(int, String): AbstractStringBuilder
isLatin1(): boolean
lastIndexOf(String): int
lastIndexOf(String, int): int
length(): int ↑CharSequence
newCapacity(int): int
offsetByCodePoints(int, int): int
putCharsAt(int, char[], int, int): void
putCharsAt(int, CharSequence, int, int): void
putStringAt(int, String): void
putStringAt(int, String, int, int): void
replace(int, int, String): AbstractStringBuilder
reverse(): AbstractStringBuilder
setCharAt(int, char): void



Appendable接口

```
public interface Appendable {
    Appendable append(CharSequence csq) throws IOException;
    Appendable append(CharSequence csq, int start, int end) throws IOException;
    Appendable append(char c) throws IOException;
}
```

CharSequence接口

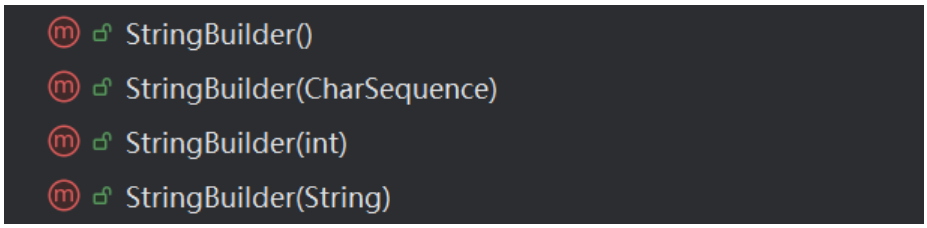
```
public interface CharSequence {
    int length();
    char charAt(int index);
    default boolean isEmpty() {
        return this.length() == 0;
    }
    CharSequence subSequence(int start, int end);
    public String toString();
    public default IntStream chars() { ..... }
    public default IntStream codePoints() { ..... }
    public static int compare(CharSequence cs1, CharSequence cs2) { .....}
}
```

字符串相关类之可变字符序列：StringBuffer、StringBuilder

> StringBuilder 的常见属性

```
public final class StringBuilder
    extends AbstractStringBuilder
    implements java.io.Serializable, Comparable<StringBuilder>, CharSequence
{
    @Serial
    static final long serialVersionUID = 4383685877147921099L;
}
```

StringBuilder 的常见构造方法




```
public StringBuilder() {
    super(16);
}
=======================================
public StringBuilder(int capacity) {
    super(capacity);
}
=======================================
public StringBuilder(String str) {
    super(str);
}
=======================================
public StringBuilder(CharSequence seq) {
    super(seq);
}
```

StringBuilder 的常见方法

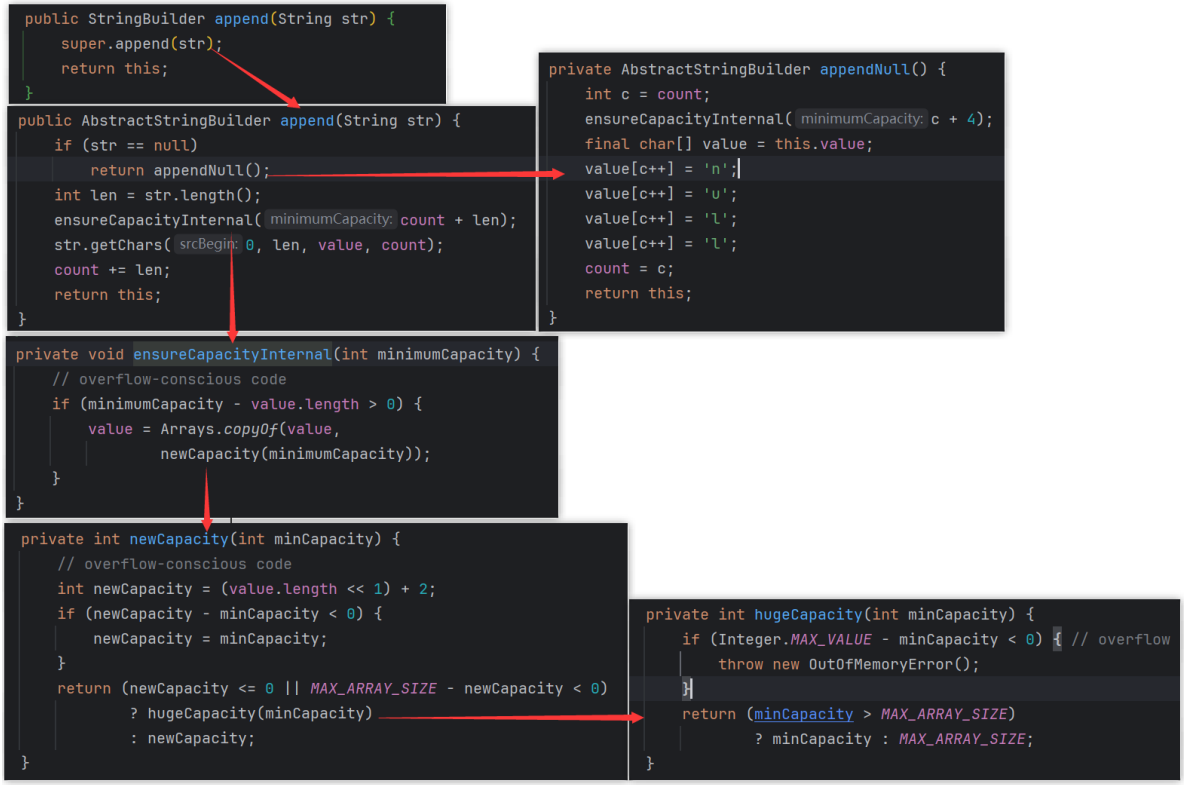
```
（1）StringBuffer append(xx)：提供了很多的append()方法，用于进行字符串追加的方式拼接
（2）StringBuffer delete(int start, int end)：删除[start,end)之间字符
（3）StringBuffer deleteCharAt(int index)：删除[index]位置字符
（4）StringBuffer replace(int start, int end, String str)：替换[start,end)范围的字符序列为str
（5）void setCharAt(int index, char c)：替换[index]位置字符
（6）char charAt(int index)：查找指定index位置上的字符
（7）StringBuffer insert(int index, xx)：在[index]位置插入xx
（8）int length()：返回存储的字符数据的长度
（9）StringBuffer reverse()：反转
```

- 当append和insert时，如果原来value数组长度不够，可扩容。
- 如上(1)(2)(3)(4)(9)这些方法支持`方法链操作`。原理：

```
@Override
public StringBuilder append(String str) {
    super.append(str);
    return this;
}
```

（1）int indexOf(String str)：在当前字符序列中查询str的第一次出现下标
（2）int indexOf(String str, int fromIndex)：在当前字符序列[fromIndex,最后]中查询str的第一次出现下标
（3）int lastIndexOf(String str)：在当前字符序列中查询str的最后一次出现下标
（4）int lastIndexOf(String str, int fromIndex)：在当前字符序列[fromIndex,最后]中查询str的最后一次出现下标
（5）String substring(int start)：截取当前字符序列[start,最后]
（6）String substring(int start, int end)：截取当前字符序列[start,end)
（7）String toString()：返回此序列中数据的字符串表示形式
（8）void setLength(int newLength) ：设置当前字符序列长度为newLength

jdk8



```
StringBuilder builder = new StringBuilder();          默认分配容量为 16 的数组
StringBuilder builder = new StringBuilder(5);         默认分配容量为 5 的数组
StringBuilder builder = new StringBuilder("str");     默认分配容量为 16 + str.length() 的数组
```

当 append 导致容量不够的时候，数组扩容机制是扩大到 (2 * value.length + 2)

可见调用 StringBuilder 调用 append 方法的时候，会调用父类 AbstractStringBuilder 的方法，如果要添加的字符串是 null，那么就会向 value[] 中加入 'n', 'u', 'l', 'l' 字符，如果数组容量不够，就会将其扩容到 (2 * value.length + 2)，如果扩容后还是小于数组元素个数，那么在其小于 MAX_ARRAY_SIZE 的时候，就会扩容为 当前数组元素个数大小

```
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
Integer.MAX_VALUE  = 0x7fffffff - 8 = 2,147,483,647 - 8 = 2的31次方-1 - 8
                                                          2的负31次方---2的31次方-1
```

扩容为什么要 +2
因为我们创建 StringBuilder 的时候,可以通过参数手动设置初始容量,如果你输入的容量为0,这个时候在扩容机制里面的通过位运算符向右移动1位还是0,所以会出现问题,所以需要加2,防止这种情况出现

jdk17