# 概述

JDK9之后String底层使用byte[]存储，之前使用char[]存储。

1. String：字符串，使用一对“”表示
2. String类 是被final修饰的，说明String类中已经拥有对字符串的非常完备的操作方法了，不需要被继承后修改或者完善。
3. String类implements了Serializable接口，凡是实现了这个接口的，都可以被序列化，这个接口的意思也是可序列化的。可序列化的类的对象都可以通过IO流进行网络传输。

String类还implements了Comparable接口，表示String是可以比较大小的。

1. String内最重要的是定义了final 修饰的char[]，这个数组保存所有字符，把所有的char[]数组的保存的值输出就是保存的字符串。
2. Char【】是被final修饰的，也代表了字符串的不可变性。是不可变的字符序列。

## String不可变性

String s1=“abc“；

String s2=“abc“；s1与s2变量存放在栈中，”abc“这个字符串是存放在char【】。

详解：第一次s1创建的时候，在栈中声明s1，在方法区中的字符串常量池中new一个char【】，将这个数组的地址赋值给s1。

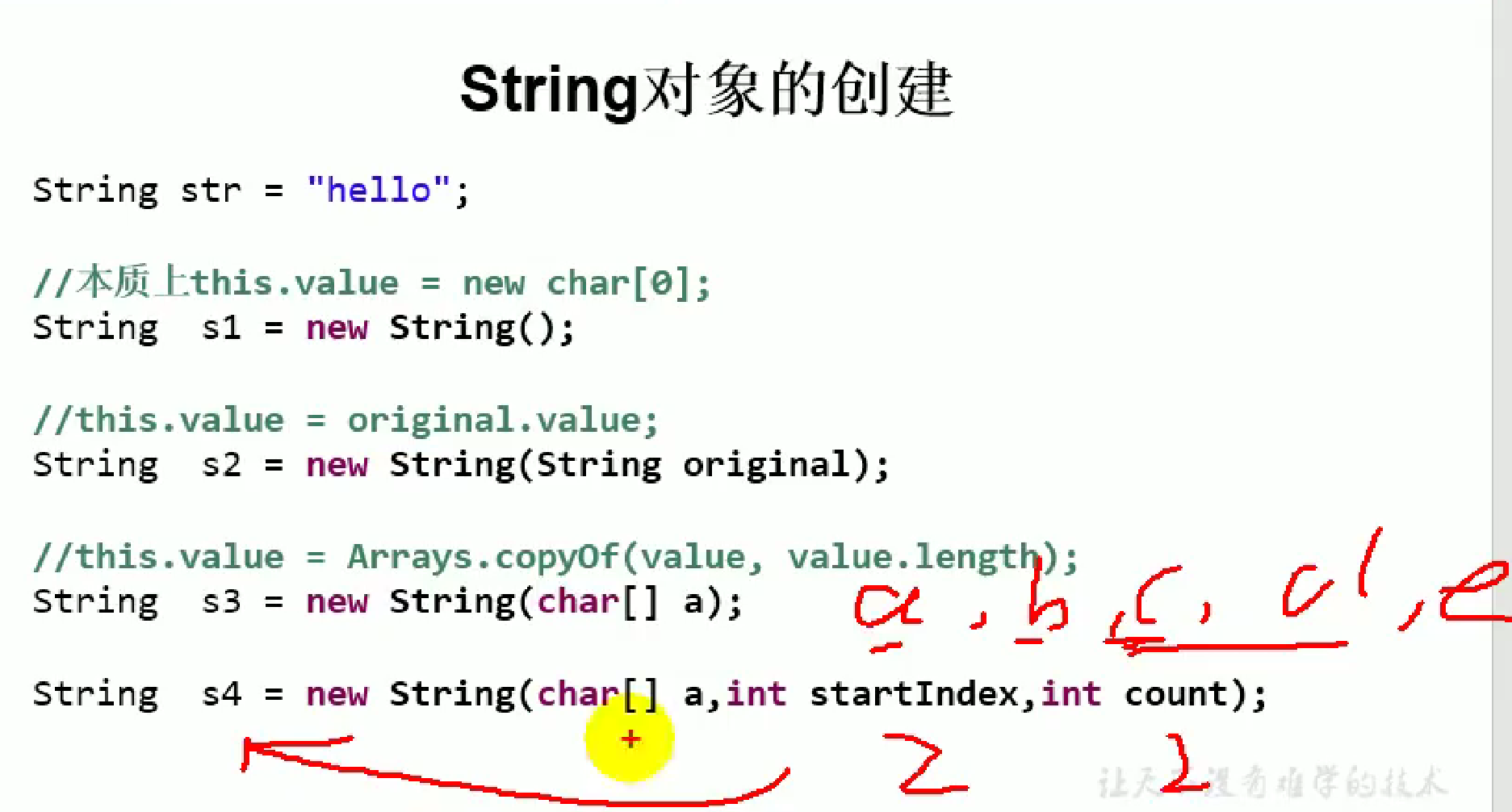
第二次s2创建的时候，栈声明s2，编译器在方法区的字符串常量池中找是否有值为abc的char数组，发现有，就将这个数组地址值给s2，此时s1与s2保存的地址是是一样的，都是方法区常量池中相同的char数组。String s2过程中相当于并没有new char【】对象。

S1=“bcd“，这时我们修改了char数组的值，这时相当于在方法区常量池中new了一个char【】对象存进bcd，然后将这个数组地址值重新赋值给s1，此时s1与s2的地址值就不再像等了。

S3=“abc“，s3+=”def“，此时s3储存的地址是方法区常量池中新new的char数组值为abcdef。

S4=“abc”，sout一下s4.replace（‘a’，‘b’）为bbc，但是sout一下s4仍然为abc，这就说明调用方法返回的是一个在方法区中常量池的新的char【】。我们用s5接收一下，输出s5就是bbc了。也说明调用方法改变s4中的某个字符，必须用一个新的char【】，而不能真的修改原本的s4的cahr【】。

## String的赋值



String s1=“abc”；

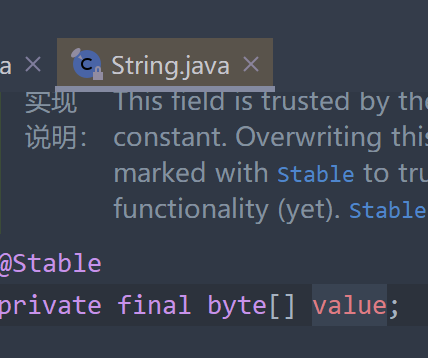
String s2=“abc”；

String s3=new String（“abc”）；

String s4=new String（“abc”）；

地址值相同的：s1和s2

其他都各不相同。



S3和s4根本就是两个不同的对象，必然地址不相同。

S3：在堆空间中new一个String类的对象，存到栈中s3变量中，通过构造器为s3对象的char【】 value属性 新new一个char【】对象，并将abc存到char【】对象中。此时new的char【】对象就放在方法区常量池中，而地址值则是在s3对象的value属性中。

此时若判断s3的value==s1，结果是true，当然没有这个方法调用value。



String s1=”abcd“，String s2=”ab“+”cd“。此时s1==s2是成立的，因为仅涉及到字面量的赋值操作都是在方法区常量池完成的。常量池中又不会存储两份一样的字符串数组，所以s2==s1成立。

String s1=“ab“ s2=”cd“

String s3=”abcd“

String s4=s1+s2 s3==s4就是false

S4储存的是在堆空间中存在的jvm帮我们new的Stirng对象的地址，s1，s2，s3存的是常量池中的数组地址。

此时s4的value属性char【】的地址与常量池中abcd的地址相同，即s4的value存的地址与s3的村的地址相同。

而s4与其他三个都不同，他存String类型的对象地址。

拿s4举例，因为涉及到对String 类型的变量操作，就需要在堆空间中new一个String对象，赋值给s4，把String类型变量s1，s2在方法区中的数组的值提取出来，赋值给s4对象的value值，这样才完成工作。

即如果我们直接拿“字符串“这样的字面量去执行赋值操作，那么就相当于仅仅是把字符存起来，在常量池中就可以完成，并且将地址给String类型的变量，以便通过变量能调用String中的方法，这时候要是再拿String 类型的变量去执行赋值操作的话，就相当于操作引用类型变量（操作对象），而不是简单的操作字符串了。所以操作引用类型的变量（储存对象地址），就需要new一个String类型的对象，如果等式右边有字面量，将字面量放到new的对象的value数组中，将等式右边的引用类型的变量的数组的值存到new的对象的value数组中。

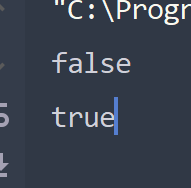
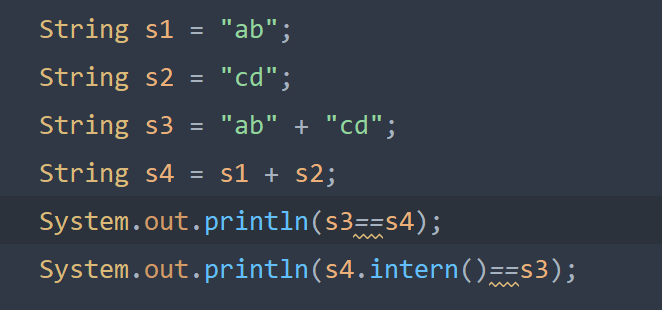
String s3=s1+“ABC“；

只要等式右边涉及到对String引用类型变量的操作，jvm就必须要new一个String类型的对象s3，虽然我们没手动new，但是jvm会做。所以s3就是一个存在堆空间中实实在在的的对象。

特例：final String s=“ab”；String s1=“abcd” String s2=s+“cd“

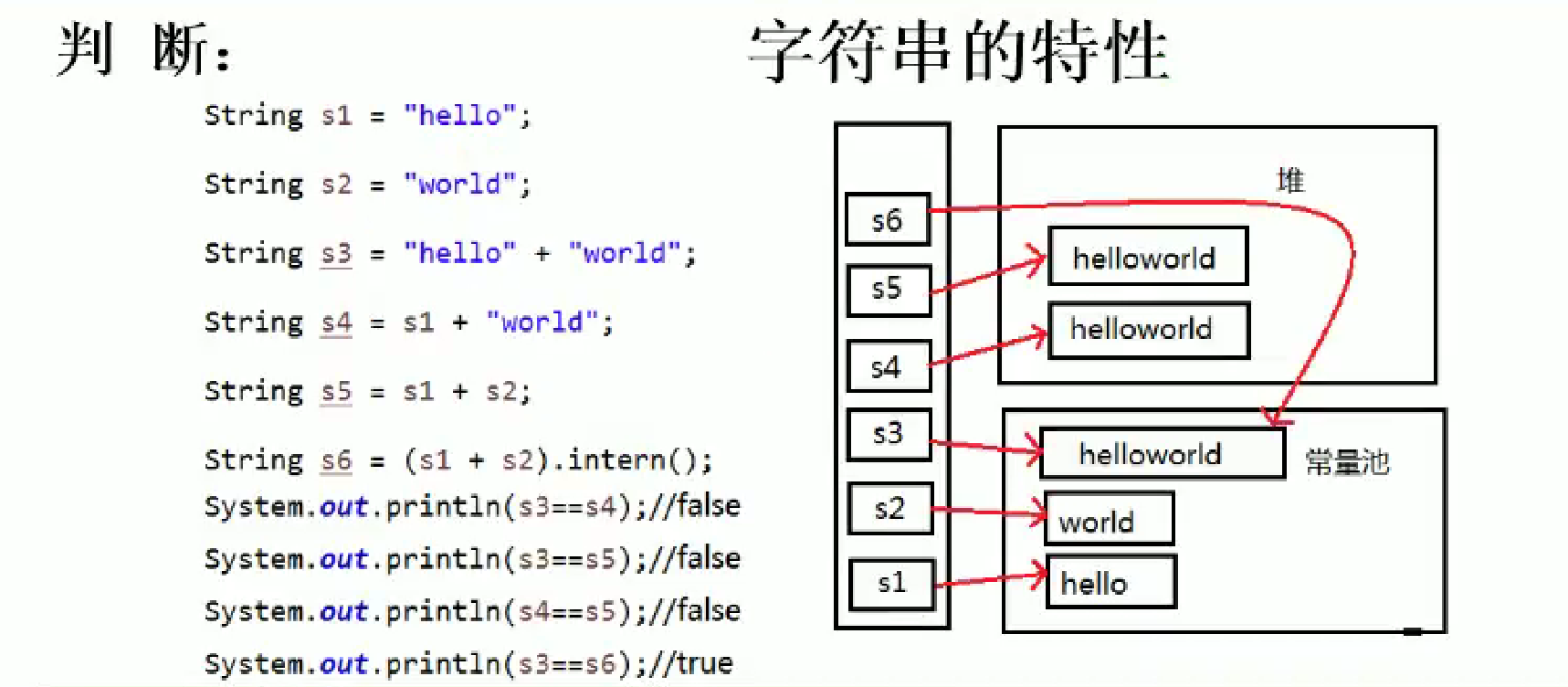
由于s被final修饰已经是一个常量，所以s2其实是两个字面量的运算，s2保存的地址也是在常量池中。

Intern方法

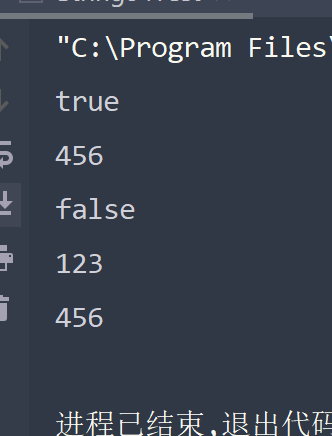


上面解释可以得知，s3存的是常量池中数组地址。s4存的是堆中对象地址，所以第一次输出肯定是false。

然后s4调用String的intern方法，使得方法返回的是s4对象中的value地址（常量池中数组地址），这时候实际上s4数组地址与s3是一样的在常量池中，所以第二次输出就是true。



题目



先看char数组，复制数组地址给形参，通过方法更换形参数组的值为456，此时chars1和chars2存的地址值是一样的，所以chars2变更值chars1值也变了。

再看s1，s1通过方法将地址复制给了s2，s2修改了常量值中的值456，并且在方法中输出s2是456已经更改了。但是输出s1还是123，为啥呢，他们两个存放的确实是同一个对象地址，但是别忘了，字符串都是存放在常量池中的，并且因为字符串不可变性，s2更改之后，先查看常量池中有无456，发现没有，在常量池中增加字符串456，并新new对象将地址值给s2，这时候s2已经是一个新的String类型变量了。并且存着新的对象地址。与s1完全不同。

上述进入方法时先输出一下时true，s2修改后又输出一下结果就是false了，说明此时s1，s2存着完全不同对象的地址。

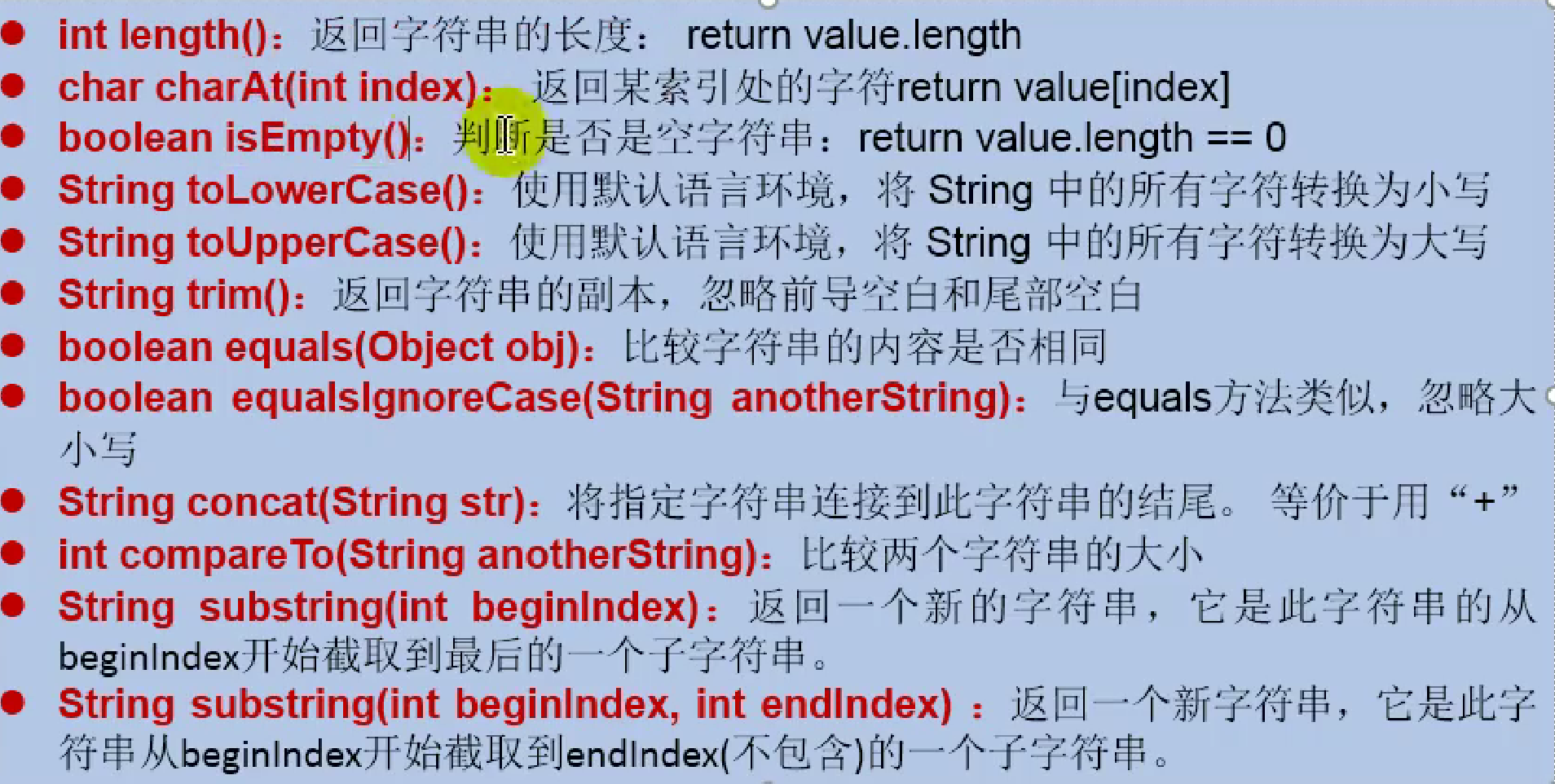
即字符串因为不可变性，变量s1最终指向的字符串数组，是final的不可更改的！就算用另一个变量s2复制一份s1地址然后去修改发现没报错，是因为jvm给s2重新new了一个对象！

如果s1直接用字面量赋值的，赋值s1地址给s2，修改s2值，s1值仍不改变，因为s1指向的常量池中数组依然是final！不可变，就算s2复制地址然后改变值仍然是jvm在常量池中新造char【】并将地址给s2.

要更改s1的值，不能将其地址复制给另外的变量s2，使用s2修改，**而必须直接使用s1修改。即s1=“xxx”。**

# String常用方法

## 第一部分



String s=“HelloWorld”；

获取长度：s.length()

获取字符串指定位置字符：s.charAt(char数组下标)

通过char[]长度判断字符串是否为空：s.isEmpty()

所有字符转大写：s.toUpperCase()

所有字符转小写：s.toLowerCase() 注意，转化操作因为字符串不可变性需要用新的String s2接收一下返回值。原来的s字符串没变。 不可变性

去除字符串首部和尾部的空格，除了中间的：s.trim() 需要另外的变量接收一下。不可变性

比较两字符串内容区分大小写：s.equals(s1)

比较两字符串内容忽略大小学：s.equalsIgnoreCase(s1)

连接方法中的字符串等价于+一般不用：s.concat(s1)等价s2=s+s1; 需新变量接收不可变性

单纯比较两个字符串大小：s.compareTo(s1) 前面减后面，就是每个字符按ASCII表顺序相减 负数相当于前面的小

手机通讯录排序

从字符串指定下标开始(包含)截取到最后形成新的字符串并返回：String s1=s.substring()

不可变性

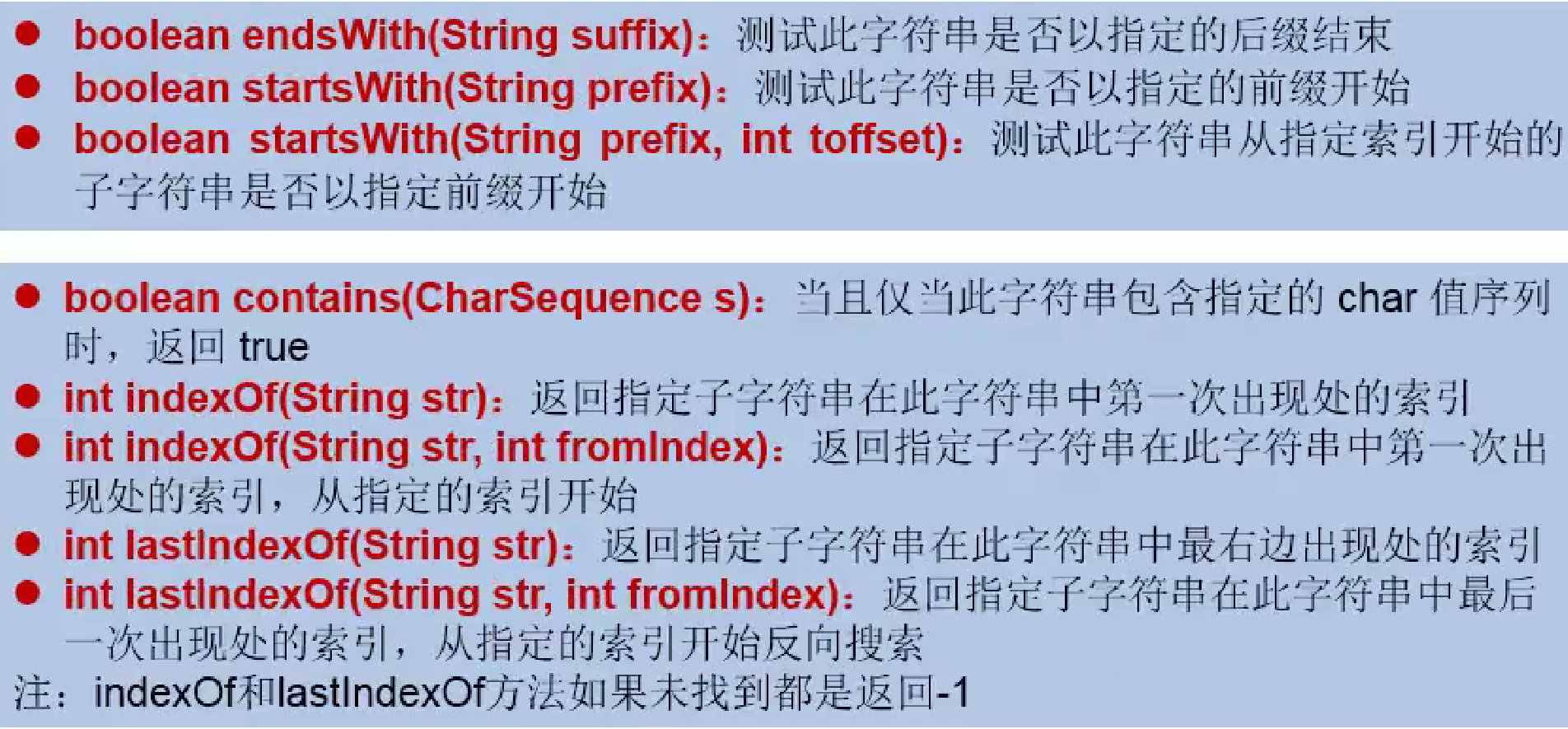
从字符串指定下标开始（包含）到截止下标的后一个下标形成新的字符串并返回（最大就是字符串的长度）：

String s=“我是zzh啊“;

String s1=s.substring(2,5); 输出s1就是zzh

不可变性 从a开始到b结束。包含a不包含b。

## 第二部分



String s=”helloworld”

注意区分大小写

判断字符串是否以指定的字符作为结尾（不限定字符个数）：s.endsWith(“d”)返回true

判断字符串是否以指定的字符作为开头（不限定字符个数）:s.startsWith(“he”) 返回true

判断字符串在指定的位置是否以指定的字符作为开头：s.startsWith(“ll”,2)返回true

判断字符串是否含有指定的子字符串：s.contains(“llo”);返回true

返回指定子字符串在字符串的哪个下标开始：s.indexOf(“lo”) 返回3

返回指定子字符串是否在字符串中并返回下标，从指定下标开始往后看是否存在：s.indexOf(“wo”,2) 返回5 注意不是3返回的就是下标值

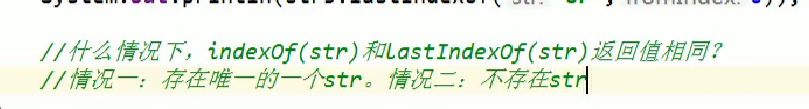
返回 指定子字符串在字符串中下标，jvm是从后往前找的，返回的下标是从前往后的：

String s1=“121212“;

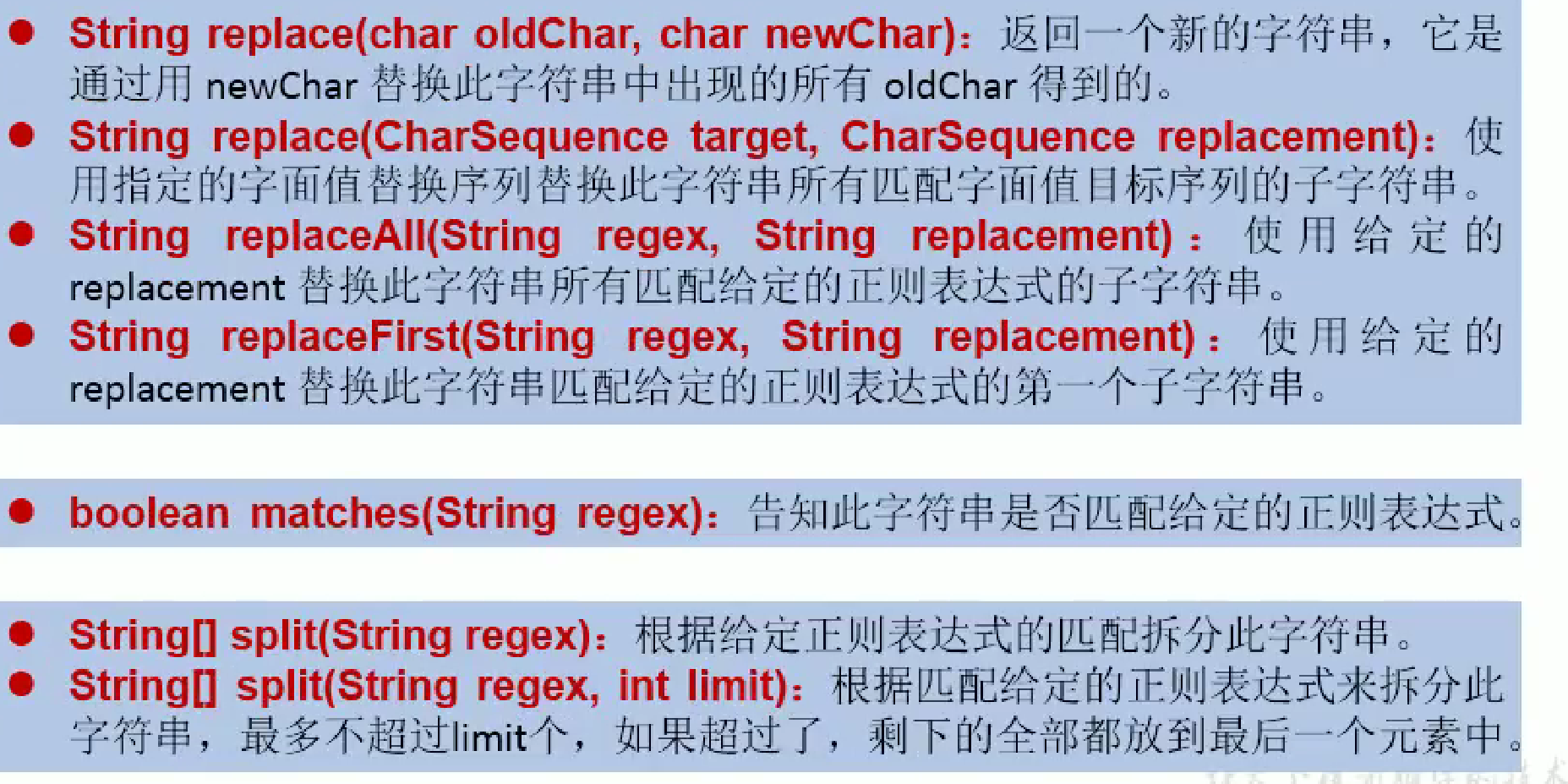
S1.lastIndexOf(“12”) 返回4

返回 指定子字符串在字符串中下标，并且指定从下标多少往前找，jvm是从后往前找的，返回的下标是从前往后的：

S1.lastIndexOf(“12”,3) 返回2 S1.lastIndexOf(“12”,4) 返回4



## 第三部分



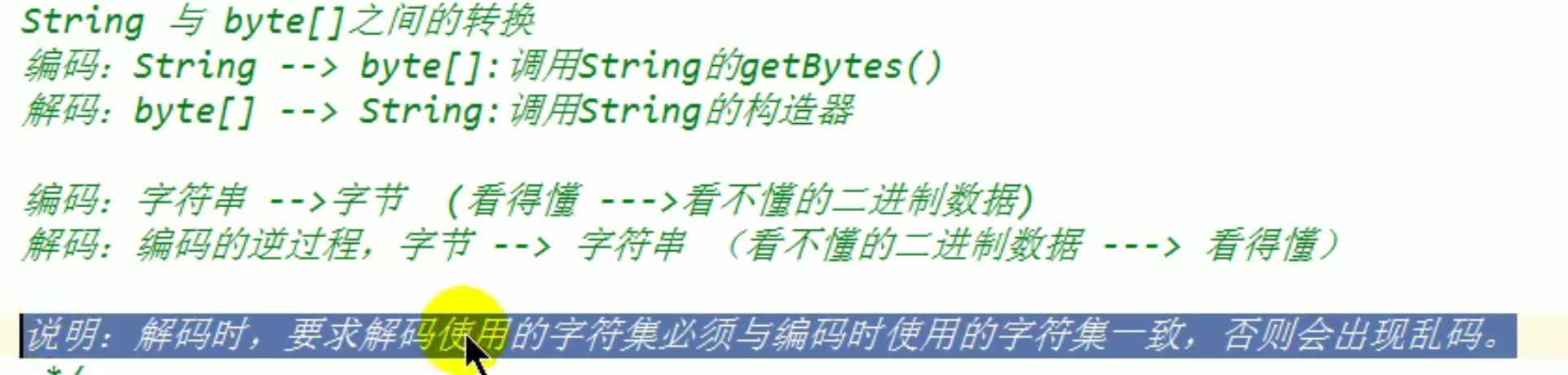
String s=”啊我是zzh啊”;

将字符串中字符替换为指定字符返回另一个字符串：s.replace(‘啊’，‘吖’)，用新变量接收 不可变性

将字符串值指定子字符串替换为另一个子字符串：s.replace(“啊我“，”啊我是“)，用新变量接收不可变性

还有正则表达式方法暂时不列举

# String转换



基本数据类型-》String String.valueOf(数据)

String-》基本数据类型 Int型：Interger.parseInt(字符串)

String转char【】 ：char[] chars =s.toCharArry()

Char[]转String ： String s=new String(chars);

String转byte[] : byte【】 bytes=s.getBytes();

String转特定编码的byte[] : byte[] bytes=s.getBytes(“gbk”); 以gbk模式转为byte[]

编码就是将看得懂的数据转为二进制 String-》byte[]

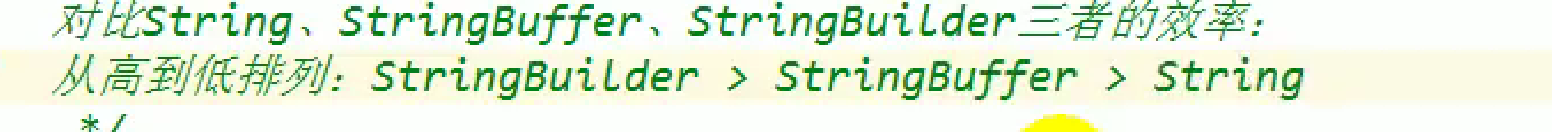
解码就是将二进制转为看得懂的数据 byte[]->String

Byte[]转String : String s=new Sting(bytes);

注意如果String转byte[]用的是gbk转码集，解码的时候，即byte[]转String的时候也必须用gbk

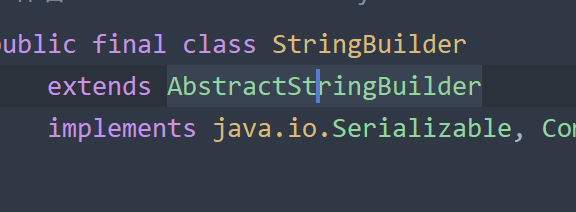
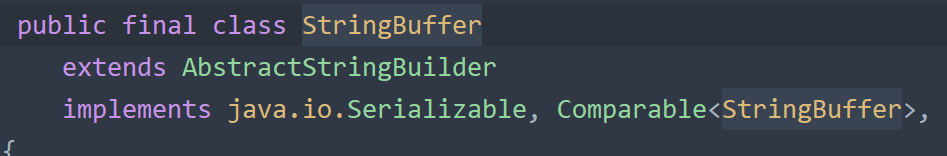
String s=new String（bytes,”gbk”）; 转码编码过程必须都是用统一编码解码集

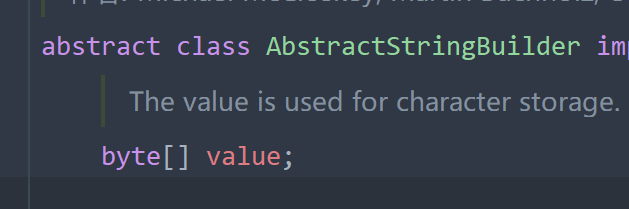
# StringBuffer和StringBuild



String:底层使用final的char[]存储字符，不可修改原有值，修改方法只能返回新的数组。

StringBuffer和StringBuilder

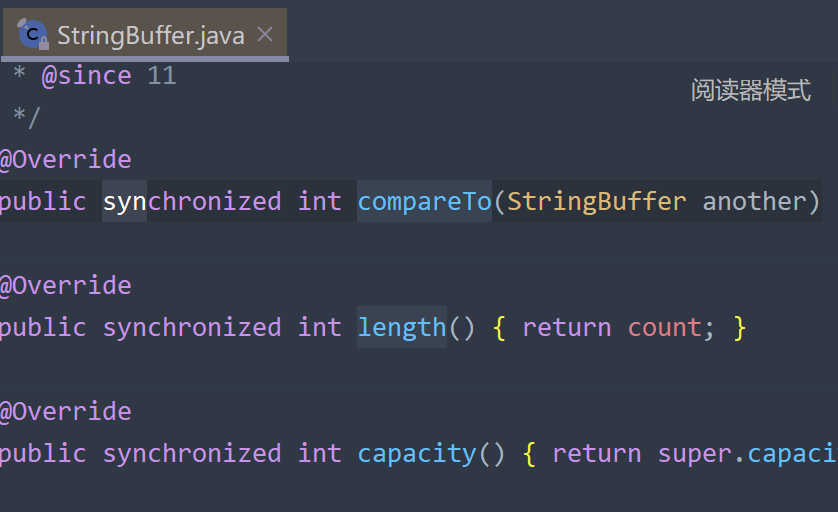




都继承抽象类AbstractStringBuilder，底层使用char【】存储数据（jdk5之前），并且没有被final修饰，说明StringBuffer和StringBuilder声明的变量指向的地址是可以被另一个相同地址的变量的更改而改变值。

所以两两者是可变的。

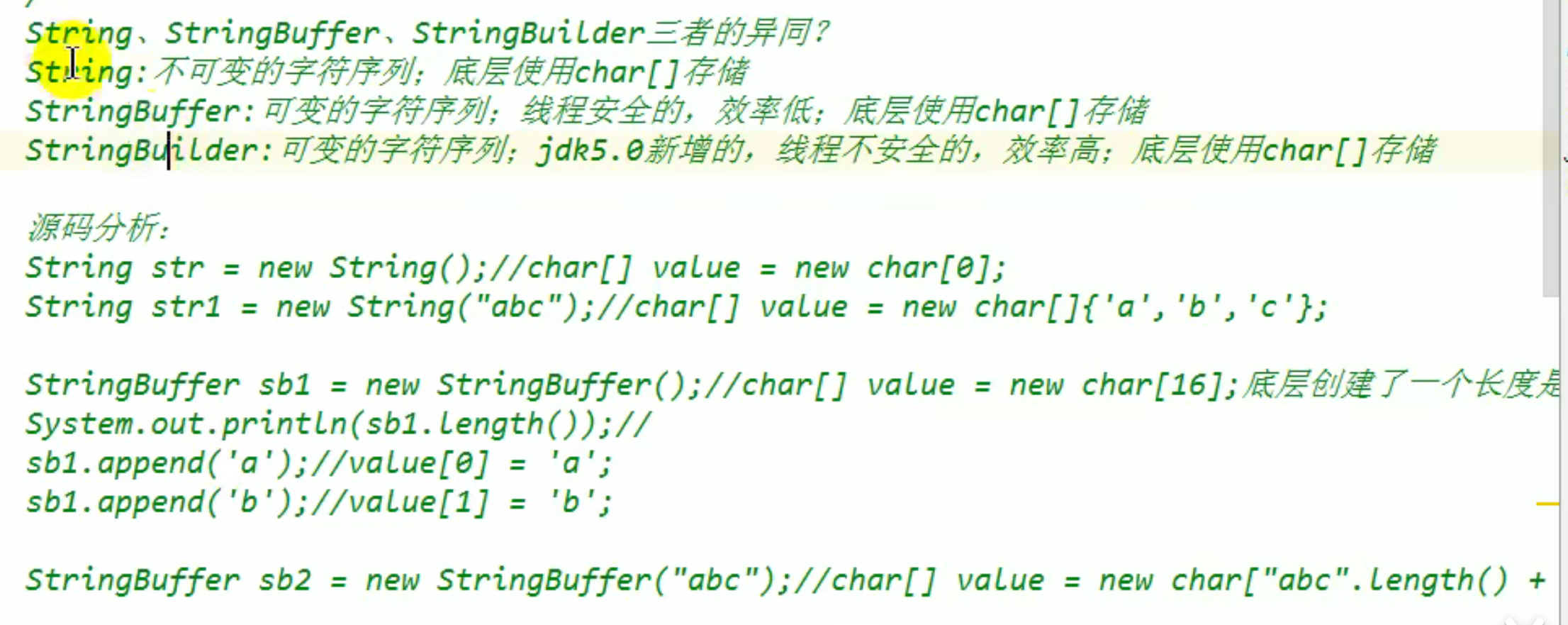
但是StringBuffer是线程安全的，他的大多数方法都被synchronized修饰。

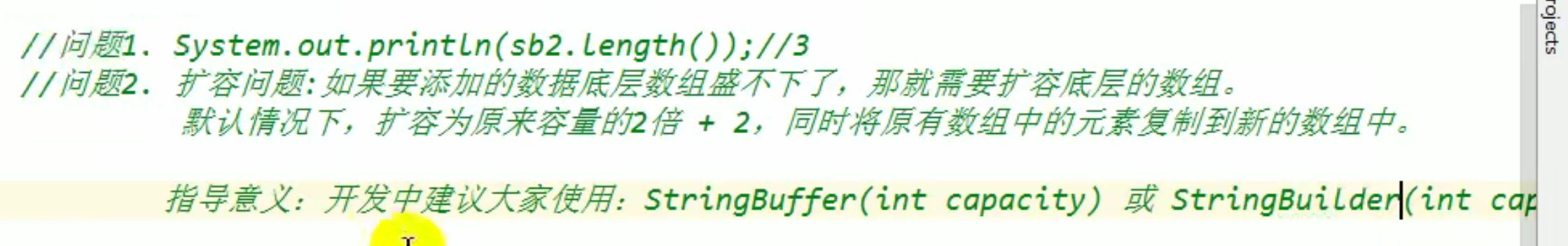


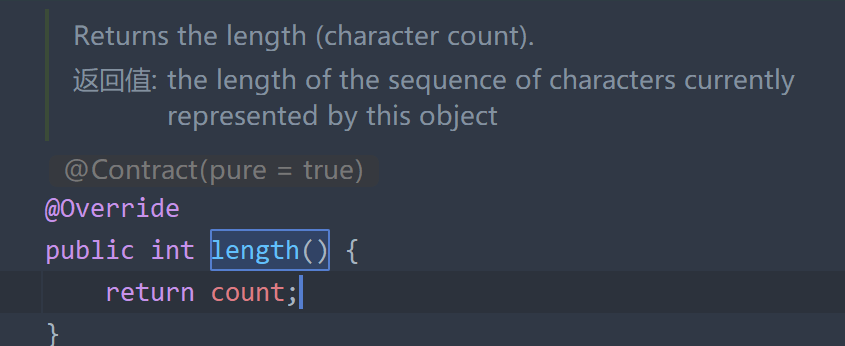
StringBuilder 类里面没有被synchronized修饰的方法，允许多个线程同时操作。

所以当确定不是多线程操作共享数据时，一般用StirngBuilder。线程不安全的。

## 三者区别







String str = new String();//final修饰的，char value=new char[0]在常量池中创建了一个长度为0的cahr[]，并将数组地址给了str对象的value属性

String str2 = new String("abc");//final修饰的。char value=new char[]{'a','b','c'}在常量池中创建了一个长度为3的cahr[]赋值abc，并将数组地址给了str对象的value属性

StringBuffer sb = new StringBuffer();//没有final修饰。char[] value=new char[16],常量池中创建了长度为16的char[]，将数组地址给了sb对象中的value属性。

System.out.println(sb.length());//获取长度获取的时属性count的值，每往数组增加一个字符count就+1，而不是返回的数组的长度，那就成16+1了。此时输出结果为0

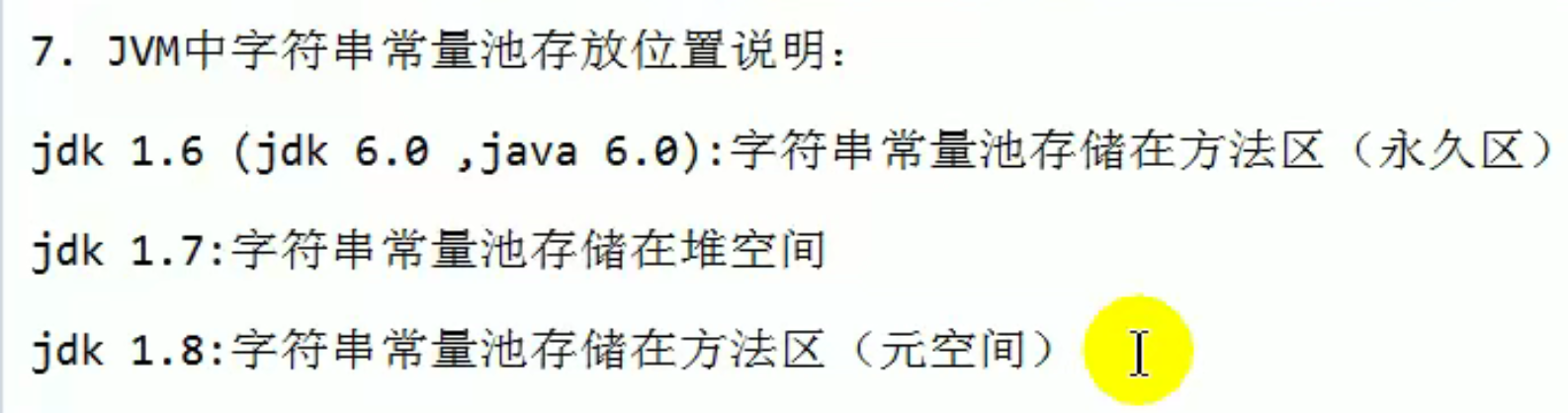
sb.append('a');//将常量池中数组char[0]='a'并不会返回新的地址，不用新的变量接收，直接在原有数组上进行修改;

sb.append('b');//将常量池中数组char[1]='b';

StringBuffer sb2 = new StringBuffer("abc");//底层在常量池中新建一个char[]，长度为abc长度加16，为19，赋值abc，并将此数组地址给sb2对象的value属性。

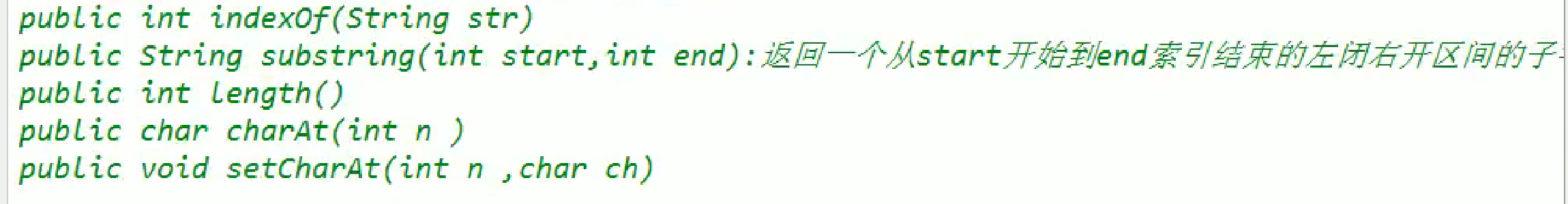
sb2.append("qwertyuiouiotry");

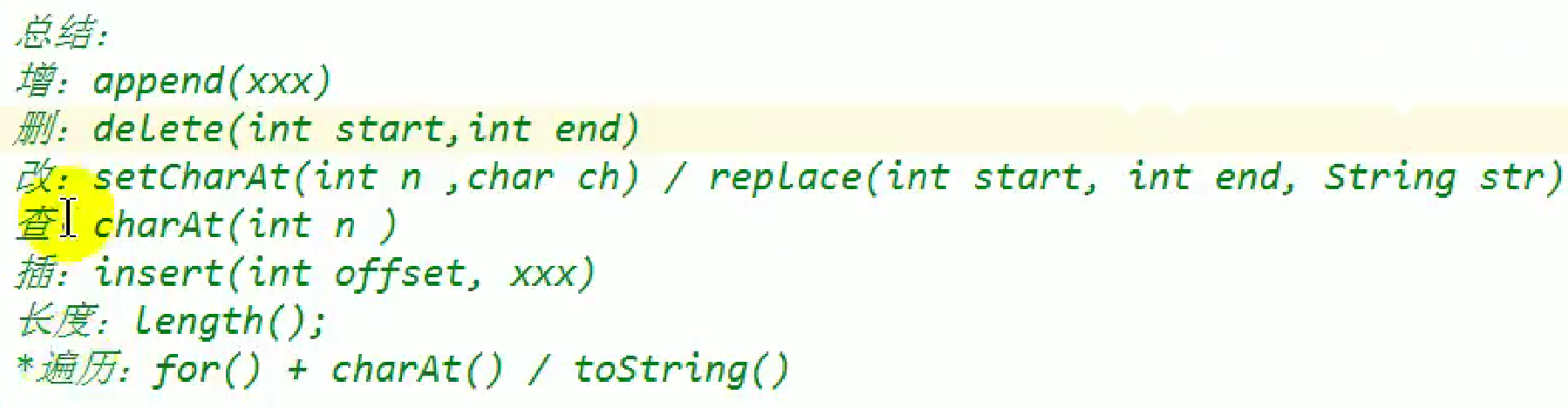
StringBuffer sb3 = new StringBuffer(50);//在常量池中创建一个指定长度的数组，以免因为频繁添append字符串，使得数组频繁自动扩容。

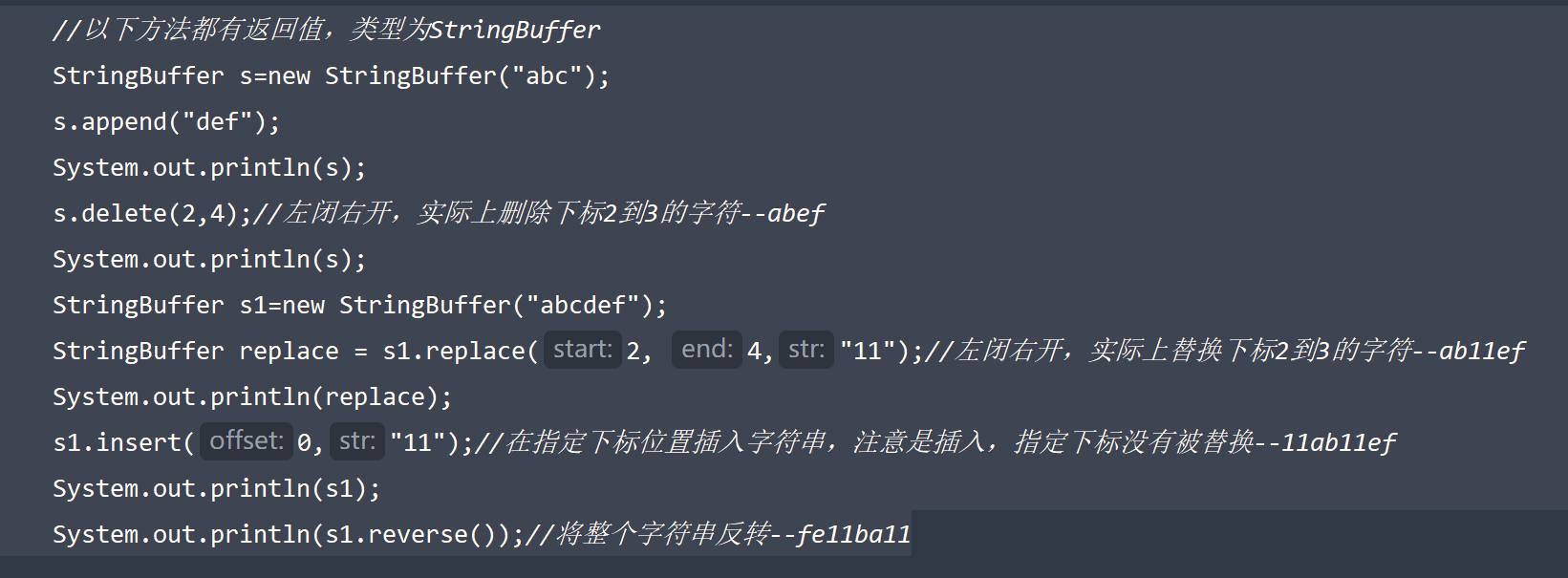


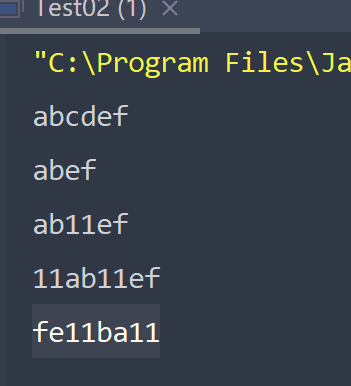
## 常用方法





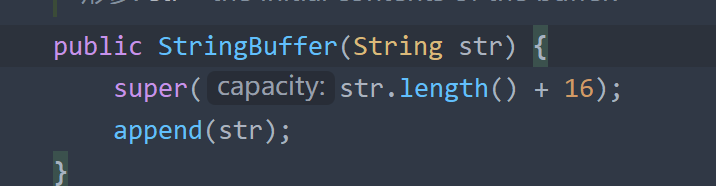






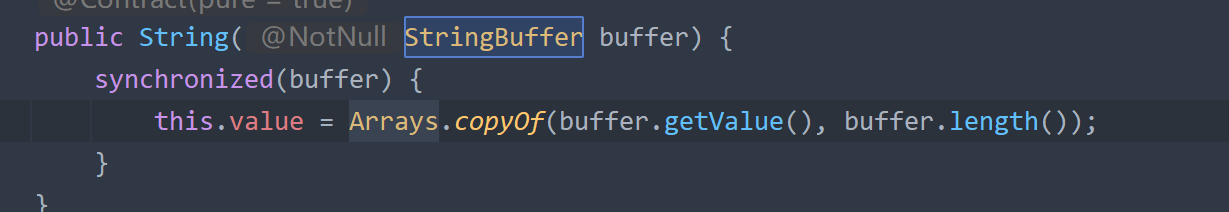
## 转换

String转为StringBuffer/Builder：



直接将String对象放到StringBuffer/Builder的构造器中就行。

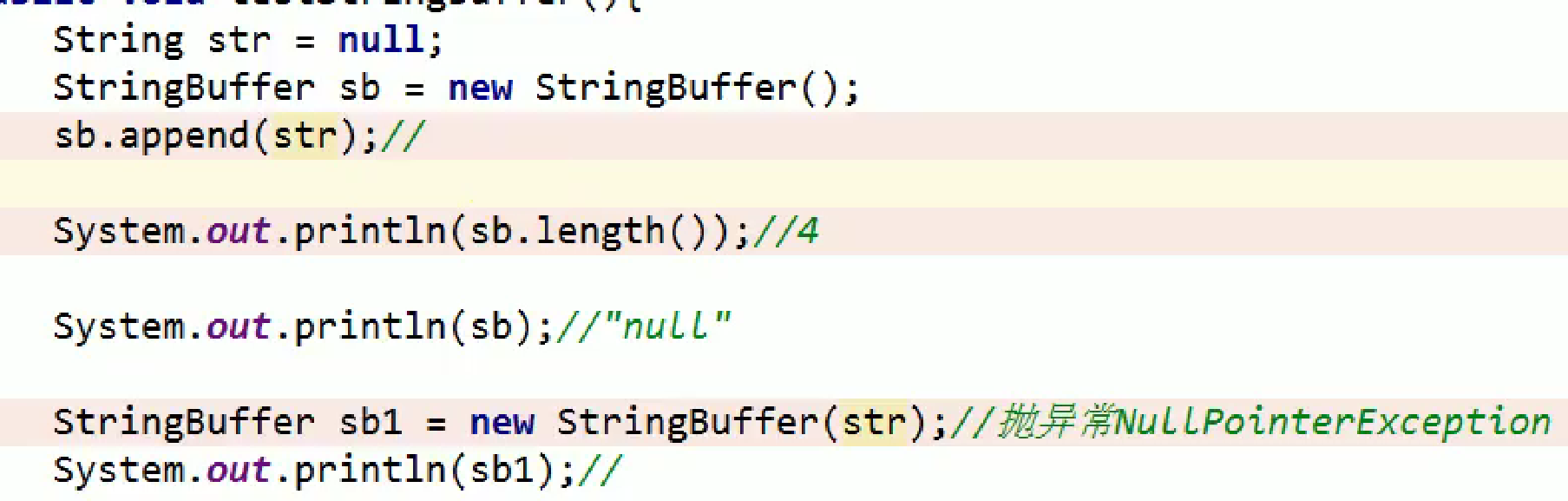
StringBuffer/Builder转为String



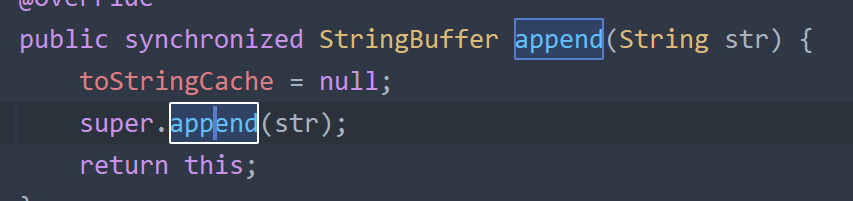
直接将StringBuffer/Builder对象放到String的构造器中就行。

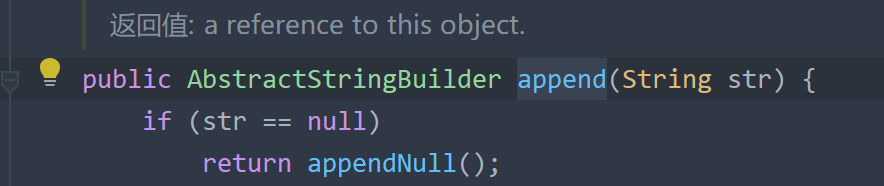
或者直接调用StringBuffer/Builder的toString方法，返回的就是一个字符串

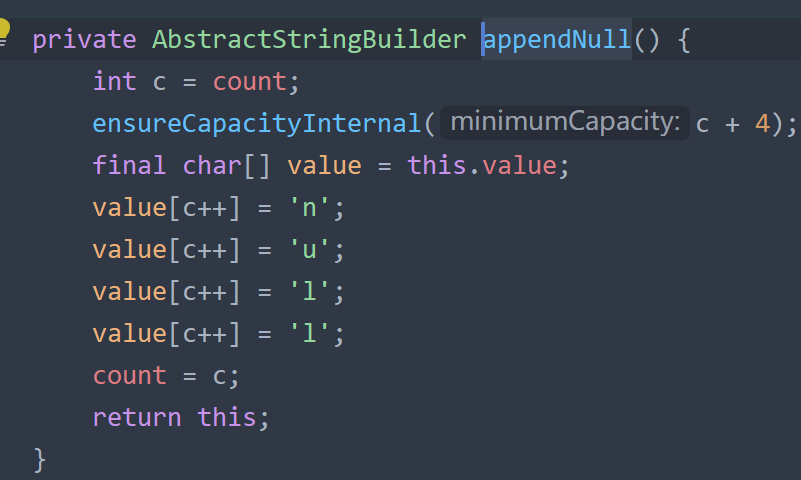
# 面试题



StrignBuffer中的构造器String类型参数：





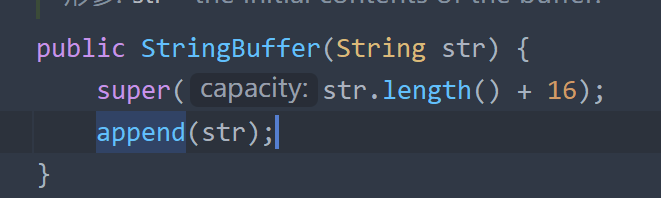


打开append方法源码，可以看到str如果是null。调用父类的appendNull方法。声明一个final修饰的char【】 value，赋值value属性的地址给这个value，注意final修饰的变量不能被重新赋值，即这个新声明的value数组的地址永远只能是value属性的地址，但是数组的内容不影响，只是地址定死了。

这样我们相当于将value数组的前四个下标值改成了‘n’,’u’,’l’,’l’。

输出sb的长度，就是4，输出sb就是“null“这个字符串

但是如果直接将空地址的str放入构造器



可以看到第一行就得调str的长度，但是str是空地址，所以就抛异常了。