在Java中字符串属于对象，也是常量。Java提供了String类来创建和操作字符串。任何类型遇到String转成String（String A = null；）

创建字符串

JDK6：

JDK7：只有一个value变量。还有一个hash成员变量，是该String对象的哈希值的缓存。

不同字符串可能共享同一个底层char数组。eg.

String s=”abc” 与 s.substring(1) 共享同一个char数组：char[] c = {‘a’,’b’,’c’}。

---

1. **字面值形式：** JVM会自动根据字符串常量池中字符串的实际情况来决定是否创建新对象 （要么不创建，要么创建一个对象，关键要看常量池中有没有）

```
String s = "abc";
```

等价于：

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

否则，在堆中创建char数组 data，然后在堆中创建一个String对象object，它由 data 数组支持，紧接着这个String对象 object 被存放进字符串常量池，最后将 s 指向这个对象。

---

2. **通过 new 创建字符串对象** ： 在堆中创建新对象，无论字符串字面值是否相等 （要么创建一个，要么创建两个对象，关键要看常量池中有没有）。有各自的独立空间。

```
String s = new String("abc");
```

等价于：

```
String original = "abc";
String s = new String(original);
```

通过 new 操作产生一个字符串（“abc”）时，会先去常量池中查找是否有“abc”对象，如果没有，则创建一个此字符串对象并放入常量池中。然后，在堆中再创建“abc”对

象，并返回该对象的地址赋予栈中的引用S。所以，如果常量池中原来没有”abc”，则会产生两个对象（一个在常量池中，一个在堆中）；否则，产生一个对象。 以及创建一个引用。

经典例题：

String str1 = new String("A"+"B") ; 会创建多少个对象?

String str2 = new String("ABC") + "ABC" ; 会创建多少个对象?

str1:

字符串常量池："A","B","AB" : 3个

堆：new String("AB") : 1个

引用： str1 : 1个

总共 ： 5个

str2 :

字符串常量池："ABC" : 1个

堆：new String("ABC") : 1个

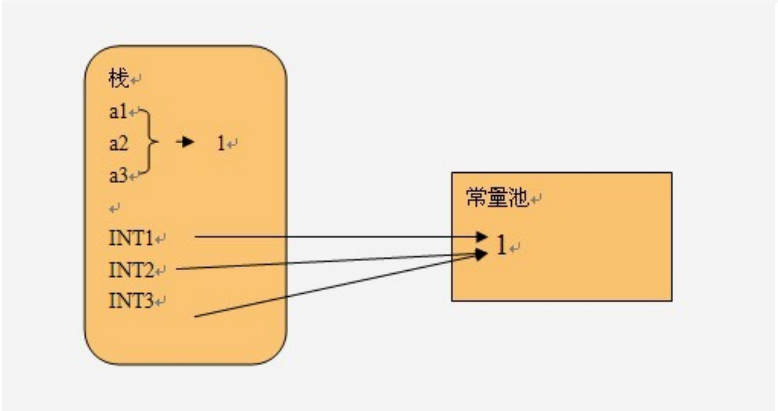
引用： str2 : 1个

总共 ： 3个

引用存储在栈中，常量存储在常量池中。（基本数据类型的局部变量存在栈中常量池。基本数据类型的成员变量存在方法区的常量池）

```
int a1 = 1;
int a2 = 1;
int a3 = 1;

public static int INT1 =1 ;
public static int INT2 =1 ;
public static int INT3 =1 ;
```



- String.intern()

通过new操作符创建的字符串对象不指向字符串池中的任何对象，但是可以通过使用字符串的intern()方法来指向其中的某一个。java.lang.String.intern()返回一个保留池字符串，就

是一个在全局字符串池中有了一个入口。如果以前没有在全局字符串池中（用 equals（Object）方法确定），那么它就会被添加到里面。对于任意两个字符串 s 和 t ，当且仅当 s.equals(t) 为 true 时，s.intern() == t.intern() 才为 true 。

```
        // Create three strings in three different ways.
        String s1 = "Hello";//常量池中
        String s2 = new StringBuffer("He").append("llo").toString();//堆中
        String s3 = s2.intern();

        // Determine which strings are equivalent using the ==
        // operator
        System.out.println("s1 == s2? " + (s1 == s2)); // false
        System.out.println("s1 == s3? " + (s1 == s3)); // true
```

经典例题：

```
            String m = "hello,world";
            String n = "hello,world";
            String u = new String(m);
            String v = new String("hello,world");

            System.out.println(m == n); //true
            System.out.println(m == u); //false
            System.out.println(m == v); //false
            System.out.println(u == v); //false
    //==比较地址
```

结论：

- m和n是同一个对象
- m,u,v都是不同的对象
- m,u,v,n但都使用了同样的字符数组，并且用equal判断的话也会返回true

3）情景三：字符串连接符“+”

```
    String str2 = "ab";           //1个对象
    String str3 = "cd";           //1个对象
    String str4 = str2+str3;
  //Class文件反编译后：      String str4 = (new StringBuilder(String.valueOf(str2))).append(str3).toString ();
    String str5 = "abcd";     //指向的是字符串常量池中字面值”abcd”所对应的字符串对象。
    System.out.println("str4 = str5 : " + (str4==str5)); // false。内存中存在五个字符串对象：  三个在字符串常量池中的String对象、一个在堆中的String对象和一个在堆中的StringBuilder对象。
```

(str2+str3)被分解成五个步骤：

(1). 调用 String 类的静态方法 String.valueOf() 将 str2 转换为字符串表示；

(2). JVM 在堆中创建一个 StringBuilder对象，同时用str2指向转换后的字符串对象进行初始化；

(3). 调用StringBuilder对象的append方法完成与str3所指向的字符串对象的合并；

(4). 调用 StringBuilder 的 toString() 方法在堆中创建一个 String对象；

(5). 将刚刚生成的String对象的堆地址存赋给局部变量引用str4。

故使用字符串连接符效率低下

4）情景四：字符串的编译期优化

```
    String str1 = "ab" + "cd";  //1个对象
//编译器已确定等价于： String str1 = “abcd”;
    String str11 = "abcd";
    System.out.println("str1 = str11 : "+ (str1 == str11));   // true

    final String str8 = "cd";
    String str9 = "ab" + str8;
//编译器已确定等价于（clas文件反编译）： String str9 = “abcd”;
    String str89 = "abcd";
    System.out.println("str9 = str89 : "+ (str9 == str89));     // true
    //↑str8为常量变量，编译期会被优化

    String str6 = "b";
    String str7 = "a" + str6;
    String str67 = "ab";
    System.out.println("str7 = str67 : "+ (str7 == str67));     // false
    //↑str6为变量，在运行期才会被解析。
```

Java 编译器对于类似“常量+字面值”的组合，其值在编译的时候就能够被确定了。str1 和 str9 的值在编译时就可以被确定。

Java 编译器对于含有 “String引用”的组合，则在运行期会产生新的对象（通过调用StringBuilder类的toString()方法），因此这个对象存储在堆中。

4、小结

- 使用字面值形式创建的字符串与通过 new 创建的字符串一定是不同的：前者在方法区，不一定会创建对象，但其引用一定指向位于字符串常量池的某个对象（常量池中已存在的字面量）；后者在堆，一定会创建对象，甚至可能会同时创建两个对象（一个位于字符串常量池中，一个位于堆中）；

- 字符串常量池的理念是 《享元模式》；

- Java 编译器对"常量+字面值"的组合会当成常量表达式直接求值来优化的；对于含有"String引用"的组合，其在编译期不能被确定，会在运行期创建新对象。

equals方法 ： 先比较引用是否相同(是否是同一对象)，再检查是否为同一类型（str instanceof String）， 最后比较内容是否一致（String 的各个成员变量的值或内容是否相同）。这也同样适用于诸如 Integer 等的八种包装器类。

String 与 (深)克隆

克隆就是制造一个对象的副本。一般地，根据所要克隆的对象的成员变量中是否含有引用类型，可以将克隆分为两种：浅克隆(Shallow Clone) 和 深克隆(Deep Clone)，默认情况下使用Object中的clone方法进行克隆就是浅克隆，即完成对象域对域的拷贝。

Cloneable 接口

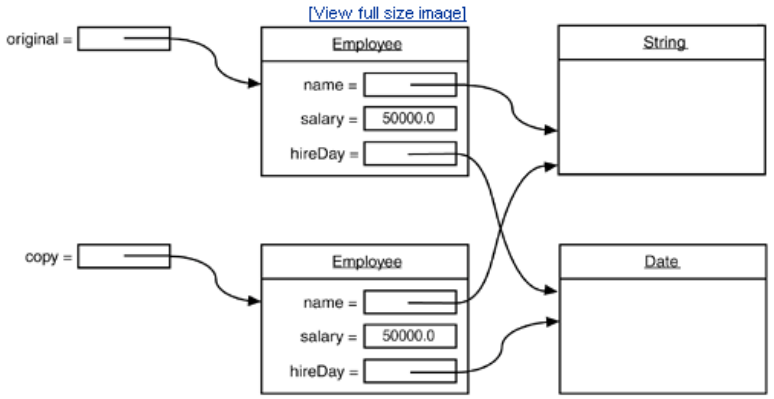
Cloneable 接口是一个标识性接口，即该接口不包含任何方法（甚至没有clone()方法），但是如果一个类想合法的进行克隆，那么就必须实现这个接口。在使用clone()方法时，若该类未实现 Cloneable 接口，则抛出 java.lang.CloneNotSupportedException 异常。

Clone & Copy

假设Employee tobby = new Employee(“CMTobby”,5000)，这样赋值Employee tom=tobby，只是copy了一下reference，tom 和 tobby 都指向内存中同一个object，这样tom或者tobby对对象的修改都会影响到对方。如果要得到tobby所指向的对象的一个精确拷贝，同时两者互不影响，需要使用Clone方法。Employee cindy=tobby.clone()，这时会生成一个新的Employee对象，并且和tobby具有相同的属性值和方法。

Shallow Clone & Deep Clone

Object的clone()方法在对某个对象实施克隆时对其是一无所知的，仅仅是简单地执行域对域的copy，这就是Shallow Clone。但以Employee为例，它里面有一个域hireDay不是基本类型的变量，而是一个reference变量，经过Clone之后克隆类只会产生一个新的Date类型的引用，它和原始引用都指向同一个 Date 对象，这样克隆类就和原始类共享了一部分信息。过程下图所示：



进行 Deep Clone ，重新定义 clone方法：

```
public Object clone() throws CloneNotSupportedException {
        Employee cloned = (Employee) super.clone();
        // Date 支持克隆且重写了clone()方法，Date 的定义是：
        // public class Date implements java.io.Serializable, Cloneable,
Comparable<Date>
        cloned.hireDay = (Date) hireDay.clone() ;
        return cloned;
    }
```

String 在克隆时只是克隆了它的引用。String是在内存中不可改变的对象。当修改值的时候，并不会改变被复制对象的值。所以克隆时，可将 String类型 视为基本类型，只需浅克隆即可。

Clone()方法的保护机制

在Object中clone()是被申明为 protected 的，这样做是有一定的道理的。以 Employee 类为例，如果我们在Employee中重写了protected Object clone()方法， ，就大大限制了可以“克隆”Employee对象的范围，即可以保证只有在和Employee类在同一包中类及Employee类的子类里面才能“克隆”Employee对象。进一步地，如果我们没有在Employee类重写clone()方法，则只有Employee类及其子类才能够“克隆”Employee对象。

连接字符串

String类提供了连接两个字符串的方法：

```
string1.concat(string2);
```

返回string2连接string1的新字符串。也可以对字符串常量使用concat()方法，如：

```
"My name is ".concat("Zara");
```

更常用的是使用'+'操作符来连接字符串。

创建格式化字符串

输出格式化数字可以使用printf()和format()方法。String类使用静态方法format()返回一个String对象而不是PrintStream对象，能用来创建可复用的格式化字符串。

```
System.out.printf("The value of the float variable is " +
                  "%f, while the value of the integer " +
                  "variable is %d, and the string " +
                  "is %s", floatVar, intVar, stringVar);
```

转化：

```
String fs;
fs = String.format("The value of the float variable is " +
                   "%f, while the value of the integer " +
                   "variable is %d, and the string " +
                   "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

String 方法

方法	解释说明
char charAt(int index)	返回字符串中第index个字符
boolean equals(String other)	如果字符串与other相等，返回true；否则，返回false。
boolean equalsIgnoreCase(String other)	如果字符串与other相等（忽略大小写），则返回true；否则，返回false。
int indexOf(String str)	返回从头开始查找第一个子字符串str在字符串中的索引位置。如果未找到子字符串str，则返回-1。
lastIndexOf()	返回从末尾开始查找第一个子字符串str在字符串中的索引位置。如果未找到子字符串str，则返回-1。
int length()	返回字符串的长度。
String replace(char oldChar,char newChar)	返回一个新串，它是通过用 newChar 替换此字符串中出现的所有oldChar而生成的。
boolean startsWith(String prefix)	如果字符串以prefix开始，则返回true。
boolean endsWith(String prefix)	如果字符串以prefix结尾，则返回true。
String substring(int beginIndex)	返回一个新字符串，该串包含从原始字符串beginIndex到串尾。
String substring(int beginIndex,int endIndex)	返回一个新字符串，该串包含从原始字符串beginIndex到串尾或endIndex-1的所有字符。
String toLowerCase()	返回一个新字符串，该串将原始字符串中的所有大写字母改成小写字母。
String toUpperCase()	返回一个新字符串，该串将原始字符串中的所有小写字母改成大写字母。
String trim()	返回一个新字符串，该串删除了原始字符串头部和尾部的空格。

SN(序号)	方法描述
1	char charAt(int index) 返回指定索引处的 char 值。

2	int compareTo(Object o) 把这个字符串和另一个对象比较。
3	int compareTo(String anotherString) 按字典顺序比较两个字符串。
4	int compareToIgnoreCase(String str) 按字典顺序比较两个字符串，不考虑大小写。
5	String concat(String str) 将指定字符串连接到此字符串的结尾。
6	boolean contentEquals(StringBuffer sb) 当且仅当字符串与指定的StringButter有相同顺序的字符时候返回真。
7	static String copyValueOf(char[] data) 返回指定数组中表示该字符序列的 String。
8	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。
9	boolean endsWith(String suffix) 测试此字符串是否以指定的后缀结束。
10	boolean equals(Object anObject) 将此字符串与指定的对象比较。
11	boolean equalsIgnoreCase(String anotherString) 将此 String 与另一个 String 比较，不考虑大小写。
12	byte[] getBytes() 使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
13	byte[] getBytes(String charsetName) 使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 将字符从此字符串复制到目标字符数组。
15	int hashCode() 返回此字符串的哈希码。
16	int indexOf(int ch) 返回指定字符在此字符串中第一次出现处的索引。
17	int indexOf(int ch, int fromIndex) 返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。
18	int indexOf(String str) 返回指定子字符串在此字符串中第一次出现处的索引。
19	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。
20	String intern() 返回字符串对象的规范化表示形式。
21	int lastIndexOf(int ch) 返回指定字符在此字符串中最后一次出现处的索引。
22	int lastIndexOf(int ch, int fromIndex) 返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索。
23	int lastIndexOf(String str) 返回指定子字符串在此字符串中最右边出现处的索引。
24	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。
25	int length() 返回此字符串的长度。
26	boolean matches(String regex) 告知此字符串是否匹配给定的正则表达式。
27	boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) 测试两个字符串区域是否相等。
28	boolean regionMatches(int toffset, String other, int ooffset, int len) 测试两个字符串区域是否相等。
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。
30	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。
31	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。
32	String[] split(String regex) 根据给定正则表达式的匹配拆分此字符串。
33	String[] split(String regex, int limit) 根据匹配给定的正则表达式来拆分此字符串。
34	boolean startsWith(String prefix) 测试此字符串是否以指定的前缀开始。
35	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。
36	CharSequence subSequence(int beginIndex, int endIndex) 返回一个新的字符序列，它是此序列的一个子序列。
37	String substring(int beginIndex) 返回一个新的字符串，它是此字符串的一个子字符串。
38	String substring(int beginIndex, int endIndex) 返回一个新字符串，它是此字符串的一个子字符串。
39	char[] toCharArray() 将此字符串转换为一个新的字符数组。
40	String toLowerCase() 使用默认语言环境的规则将此 String 中的所有字符都转换为小写。
41	String toLowerCase(Locale locale) 使用给定 Locale 的规则将此 String 中的所有字符都转换为小写。
42	String toString() 返回此对象本身（它已经是一个字符串！）。
43	String toUpperCase() 使用默认语言环境的规则将此 String 中的所有字符都转换为大写。
44	String toUpperCase(Locale locale) 使用给定 Locale 的规则将此 String 中的所有字符都转换为大写。
45	String trim() 返回字符串的副本，忽略前导空白和尾部空白。
46	static String valueOf(primitive data type x) 返回给定data type类型x参数的字符串表示形式。