# java空字符串和null的区别

1、类型

null表示的是一个对象的值，而并不是一个字符串。例如声明一个对象的引用，String a = null ;

""表示的是一个空字符串，也就是说它的长度为0。例如声明一个字符串String str = "" ;

2、内存分配

String str = null ; 表示声明一个字符串对象的引用，但指向为null，也就是说还没有指向任何的内存空间。

String str = "";    表示声明一个字符串类型的引用，其值为""空字符串，这个str引用指向的是空字符串的内存空间。

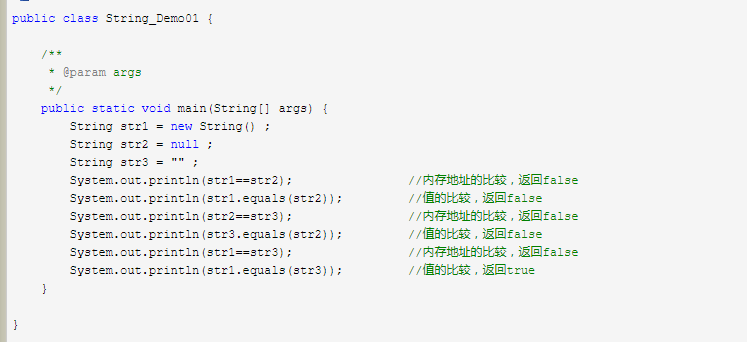
在java中变量和引用变量是存在栈中（stack），而对象（new产生的）都是存放在堆中（heap）。

就如下：

String str = new String("abc") ;

ps：=左边的是存放在栈中（stack），=右边是存放在堆中（heap）

示例代码：



通过如上的程序可以得出如下结论：

（1）字符串对象与null的值不相等，且内存地址也不相等。

（2）空字符串对象与null的值不相等，且内存地址也不相等。

（3）new String()创建一个字符串对象，它的默认值为""。

（4）String类型成员变量的初始值为null。

# 判断字符串为空的方法

## 第一种方法

最多人使用的一个方法, 直观, 方便, 但效率很低

if(s == null || s.equals(""));

## 第二种方法

比较字符串长度, 效率高, 是我知道的最好一个方法

**if(s == null || s.length() == 0);**

## 第三种方法

Java SE 6.0才开始提供的方法, 效率和方法二几乎相等, 但出于兼容性考虑, 推荐使用方法二

if(s == null || s.isEmpty());

## 第四种方法

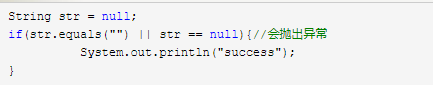
这是一种比较直观,简便的方法,而且效率也非常的高,与方法二、三的效率差不多。

if (s == null || s == "");

注意:s == null 是有必要存在的.

如果 String 类型为null, 而去进行 equals(String) 或 length() 等操作会抛出java.lang.NullPointerException，并且s==null 的顺序必须出现在前面，不然同样会抛出java.lang.NullPointerException。

如下Java代码:



# 如何比较两个字符串相等

简单来说，“==”是用来检测两个引用是不是指向内存中的同一个对象，而equals()方法则检测的是两个对象的值是否相等。如果你想检测两个字符串的值是不是相等，你就必须用equals()方法。

示例：

String str = "jinshihui";

String str1 = "jinshihui";

System.out.println(str.equals(str1));//true

System.out.println(str==str1);//true

# String类

## 基本概念

（1）String类代表字符串，Java程序中的所有字符串字面值都作为此类的实例实现。字符串是常量，它们的值在创建之后不能更改，字符串缓冲区支持可变的字符串，因为String对象是不可变的，所以可以共享。String的本质是char[]（字符数组）。

（2）String类包括的方法可用于检查序列的单个字符、比较字符串、搜索字符串、提取子字符串、创建字符串副本并将所有字符全部转换为大写或小写。大小写映射基于Character类指定的Unicode标准版。

（3）Java语言提供对字符串串联符号（“+”）以及将其他对象转换为字符串的特殊支持，字符串串联是通过StringBuilder（或StringBuffer）类及其append方法实现的。字符串转换是通过toString方法实现的，该方法由Object类定义，并可被Java中的所有类继承。

（4）String 类型默认初始值为 null，即为空未分配对象，如果是string s = "";则是分配了一个对象引用，但其值是个空字符。

## String对象的构造方法

### （1）String()

初始化一个新创建的String对象，使其表示一个空字符序列。

### （2）String(byte[] bytes)

通过使用平台的默认字符集解码指定的byte数组，构造一个新的String。

### （3）String(byte[] bytes,Charset charset)

通过使用指定的字符集解码指定的byte数组，构造一个新的String。

### （4）String(byte[] bytes,int offset,int length)

通过使用平台的默认字符集解码指定的byte子数组，构造一个新的String。

### （5）String(byte[] bytes,String charsetName)

通过使用指定的charset解码指定的byte数组，构造一个新的String。

### （6）String(char[] value)

分配一个新的 String，使其表示字符数组参数中当前包含的字符序列，将字符数组转换为字符串。

示例：

char[] a = {'a','b'};

String str = new String(a);

System.out.println(str);

### （7）String(char[] value,int offset,int count)

分配一个新的 String，它包含取自字符数组参数一个子数组的字符。

## String对象常用的方法（重点）

### （1）char charAt(int index)

返回指定索引处的字符值，从0开始搜索。

### （2）int compareTo(String anotherString)

按字典顺序比较两个字符串。

返回：如果此字符串等于字符串参数，则值为0; 如果此字符串按字典顺序小于字符串参数，则小于0; 如果此字符串按字典顺序大于字符串参数，则值大于0。

### （3）String concat(String str)

将指定字符串连接到此字符串的结尾。

### （4）boolean contains(CharSequence s)

当且仅当此字符串包含指定的char值序列时，返回true。

### （5）boolean equals(Object obj)

将此字符串与指定的对象比较。

### （6）String format(Locale l, String format, Object... args)

使用指定的语言环境、格式字符串和参数返回一个格式化字符串。

### （7）String format(String format, Object... args)

使用指定的格式字符串和参数返回一个格式化字符串。

### （8）int hashCode()

返回此字符串的哈希码

### （9）int indexOf(String str)

返回指定子字符串在此字符串中第一次出现的索引。

### （10）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。

### （11）boolean isEmpty()

判断字符串是否为空，当且仅当length()为0时返回true，否则返回false。

### （12）int length()

返回字符串的长度。

### （13）boolean matches(String regex)

字符串是否匹配给定的正则表达式。

//示例

boolean a = str.matches("^[a-z]\*$");

System.out.println(a);

### （14）String replace(char oldChar, char newChar)

返回一个新的字符串，它是通过用newChar替换字符串中出现的所有oldChar得到的。

### （15）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。

### （16）String[] split(String regex)

根据给定正则表达式的匹配拆分字符串。

### （17）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。

### （18）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，从0开始起始索引。

### （19）String substring(int beginIndex,int endIndex)

返回一个新的字符串，它是此字符串的一个子字符串。

### （20）String trim()

返回字符串的副本，忽略前后空白。

### （21）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。

示例：

String str = "jinshihui";

byte[] b = str.getBytes("UTF-8");

for(int i=0;i<b.length;i++){

System.out.println(b[i]);

}

### （22）char[] toCharArray()

//将字符串转换为一个新的字符数组

### （23）public native String intern();

//与此字符串具有相同内容的字符串，但保证来自唯一字符串池。

原理：String.intern()是一个Native方法，底层调用C++的 StringTable::intern方法实现。当通过语句str.intern()调用intern()方法后，JVM 就会在当前类的常量池中查找是否存在与str等值的String，若存在则直接返回常量池中相应Strnig的引用；若不存在，则会在常量池中创建一个等值的String，然后返回这个String在常量池中的引用。因此只要是等值的String对象，使用intern()方法返回的都是常量池中同一个String引用，所以这些等值的String对象通过intern()后使用==是可以匹配的。

总结：String.intern()方法是一种手动将字符串加入常量池中的方法，原理如下：如果在常量池中存在与调用intern()方法的字符串等值的字符串，就直接返回常量池中相应字符串的引用，否则在常量池中复制一份该字符串，并将其引用返回（Jdk7中会直接在常量池中保存当前字符串的引用）；Jdk6 中常量池位于PremGen区，大小受限，不建议使用String.intern()方法，不过Jdk7 将常量池移到了Java堆区，大小可控，可以重新考虑使用String.intern()方法，但是由对比测试可知，使用该方法的耗时不容忽视，所以需要慎重考虑该方法的使用；String.intern()方法主要适用于程序中需要保存有限个会被反复使用的值的场景，这样可以减少内存消耗，同时在进行比较操作时减少时耗，提高程序性能。

### （24）public int codePointCount(int beginIndex, int endIndex)

返回此String的指定文本范围内的Unicode代码点数，也是计算字符串个数的方法。

参数：

beginIndex：文本范围的第一个char的索引

endIndex：文本范围的最后一个char的索引

返回值：指定文本范围内的Unicode代码点数

### （25）public int codePointAt(int index)

返回指定索引位置的字符（Unicode代码点数）

参数：字符值的索引

返回：在指定索引处的字符的代码点值。

# String类length与codePointCount的区别

对于普通字符串这两种方法得到的值是一样的，但对于UniCode编码来说，还是有一点区别。

length()方法返回的是使用的是UTF-16编码的字符代码单元数量，不一定是实际上我们认为的字符个数。codePointCount()方法返回的是代码点个数，是实际上的字符个数。

例如：String str = “/uD835/uDD6B”，那么机器会识别它是2个代码单元代理的1个代码点”Z“，故而，length的结果是代码单元数量2，而codePointCount()的结果是代码点数量1。

# 为什么安全敏感的字符串信息用char[]会比String对象更好

String对象是不可变的，意味着直到垃圾回收器过来清扫之前它们都不会发生变化的。用数组的话，就可以很明确的修改它任何位置的字符元素。这样的话，如密码等安全敏感的信息就不会出现在系统的任何地方。

# 字符串对象能否用在switch表达式中

从JDK7开始，我们就可以在switch条件表达式中使用字符串了，也就是说7之前的版本是不可以的。

# 将字符串转换为整型数值

int n = Integer.parseInt("10");

# 快速重复构造一段字符串

在Java编程中，我们可以使用来自Apache Commons Lang包中的StringUtils类的repeat()方法进行重复一段字符串。

示例：

String str = "abcd";

String repeated = StringUtils.repeat(str,3);

//abcdabcdabcd

# 将时间格式的字符串转换成date对象

String str = "Sep 17, 2013";

Date date = new SimpleDateFormat("MMMM d, yy", Locale.ENGLISH).parse(str);

System.out.println(date);

//Tue Sep 17 00:00:00 EDT 2013

# 计数一个字符在某个字符串中出现的次数

使用Apache Commons Lang包中的StringUtils类中的countMatches方法就可以完成这个工作。

int n = StringUtils.countMatches("11112222", "1");

System.out.println(n);//4

# StringUtils类中的常用方法（重点）

说明：StringUtils这个类在commons-lang3.jar这个包中。

## （1）public static boolean isEmpty(String str)

判断某字符串是否为空，为空的标准是 str==null 或 str.length()==0

示例代码：

String str = "jin";

boolean b = StringUtils.isEmpty(str);

System.out.println(b);

下面是StringUtils判断是否为空的示例：

StringUtils.isEmpty(null) = true

StringUtils.isEmpty("") = true

StringUtils.isEmpty(" ") = false //注意在StringUtils中空格作非空处理

StringUtils.isEmpty(" ") = false

StringUtils.isEmpty("bob") = false

StringUtils.isEmpty(" bob ") = false

## （2）public static boolean isNotEmpty(String str)

判断某字符串是否不为空，等于 !isEmpty(String str)

下面是示例：

StringUtils.isNotEmpty(null) = false

StringUtils.isNotEmpty("") = false

StringUtils.isNotEmpty(" ") = true

StringUtils.isNotEmpty(" ") = true

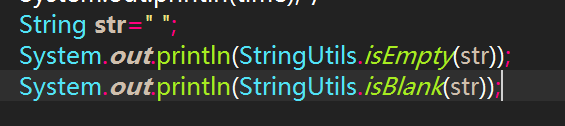
StringUtils.isNotEmpty("bob") = true

StringUtils.isNotEmpty(" bob ") = true

## （3）public static boolean isBlank(String str)

//判断某个字符串是否为空，为空的标准是：

与isEmpty的区别



结果：false true

## （4）public static boolean isNotBlank(String str)

//判断某个字符串是否不为空

## （5）public static String capitalize(String str)

//将首字母大写

## （6）public static String uncapitalize(String str)

//首字母不大写

## （7）public static String upperCase(String str)

//全部转换为大写字母

## （8）public static String lowerCase(String str)

//全部转换为小写字母

## （9）public static String swapCase(String str)

//大小写字母互转

## （10）public static String remove(String str，char remove)

//删除字符

## （11）public static String remove(String str，String remove)

//删除字符串

## （12）public static String removeEnd(String str，String remove)

//删除结尾匹配的字符串

## （13）public static String trim(String str)

去掉字符串两端的控制符(control characters, char <= 32) ,包含去掉字符串两端的空格，如果输入为null，则返回null。

示例：

StringUtils.trim(null) = null

StringUtils.trim("") = ""

StringUtils.trim(" ") = ""

StringUtils.trim(" \b \t \n \f \r ") = ""

StringUtils.trim(" \n\tss \b") = "ss"

StringUtils.trim(" d d dd ") = "d d dd"

StringUtils.trim("dd ") = "dd"

StringUtils.trim(" dd ") = "dd"

# 获取系统当前时间

SimpleDateFormat是一个与语言环境有关的方式来格式化和解析日期的具体类，它允许进行格式化（日期--文本）、解析（文本--日期）和规范化。

SimpleDateFormat date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String time = date.format(new Date());

System.out.println(time);//返回当前系统的日期时间

# 字符串编码与解码

String name = "张三";

byte[] b1 = name.getBytes("UTF-8");//将字符串编码存储到字节数组中

String name1 = new String(b1, "UTF-8"); //将字符串进行解码，编码解码相同，正常显示

System.out.println(name1);

String name2 = new String(b1, "GBK"); //将字符串进行解码，编码解码不同，乱码

System.out.println(name2);

# 字符数组转换为字符串

char[] a = {'a','b'};

String str = new String(a);

System.out.println(str);

# 字符串数组转换为字符串

思路：通过循环去转换

String[] a = {"靳世辉","中国","山西"};

StringBuilder sbBuilder = new StringBuilder();

for(int i=0;i<a.length;i++){

sbBuilder.append(a[i]);

}

System.out.println(sbBuilder.toString());

# StringBuffer

## 基本概念

（1）StringBuffer是线程安全的可变字符序列，虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。它可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。

（2）StringBuffer的主要操作是append和insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而insert方法则在指定的位置添加字符。

例如，如果z引用一个当前内容是“start”的字符串缓冲区对象，则调用 z.append("le") 会使字符串缓冲区包含“startle”，而 z.insert(4, "le") 将更改字符串缓冲区，使之包含“starlet”。

在大部分情况下 StringBuilder > StringBuffer。

## 方法

### StringBuffer append(Object o)

//追加Object字符串

### int capacity()

//返回当前容量，容量指可用于最新插入的字符的存储量，超过这一容量就需要再次进行分配。默认容量的大小是16。

### StringBuffer insert(int offset Object o)

//将字符串插入到指定的位置

# StringBuilder

## 基本概念

StringBuilder是一个可变的字符序列，是java5.0新增的。此类提供一个与 StringBuffer兼容的API，但不保证同步。该类被设计用作 StringBuffer的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能建议优先采用该类，因为在大多数实现中它比 StringBuffer 要快，两者的方法基本相同。

String,StringBuffer与StringBuilder三者的执行效率：

StringBuilder>StringBuffer>String

# String、StringBuffer、StringBuilder说明

（1）String是字符串常量，StringBuffer是字符串变量（线程安全），StringBuilder是字符串变量（非线程安全）。

（2）简要的说， String类型和StringBuffer类型的主要区别在于 String 是不可变的对象, 因此每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无用对象多了以后， JVM的GC就会开始工作，那速度是相当慢的。而如果是使用 StringBuffer 类则结果就不一样了，每次结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再改变对象引用。所以在一般情况下我们推荐使用StringBuffer，特别是字符串经常改变的情况下。而在某些特别情况下，String 对象的字符串拼接其实是被 JVM 解释成了 StringBuffer 对象的拼接，所以这些时候 String 对象的速度并不会比 StringBuffer 对象慢，而特别是以下的字符串对象生成中， String 效率是远要比 StringBuffer 快的。

String S1 = “This is only a” + “ simple” + “ test”;

StringBuffer Sb = new StringBuilder(“This is only a”).append(“ simple”).append(“ test”);

你会很惊讶的发现，生成 String S1 对象的速度简直太快了，而这个时候 StringBuffer 居然速度上根本一点都不占优势。其实这是 JVM 的一个把戏，在 JVM 眼里，这个

String S1 = “This is only a” + “ simple” + “test”; 其实就是：

String S1 = “This is only a simple test”; 所以当然不需要太多的时间了。但大家这里要注意的是，如果你的字符串是来自另外的 String 对象的话，速度就没那么快了，譬如：

String S2 = “This is only a”;

String S3 = “ simple”;

String S4 = “ test”;

String S1 = S2 +S3 + S4;

这时候 JVM 会规规矩矩的按照原来的方式去做。

在大部分情况下 StringBuffer > String。

# CharSequence接口

## 基本概念

CharSequence是一个接口，表示char值的一个可读序列。此接口对许多不同种类的char序列提供统一的自读访问。它有几个实现类：CharBuffer、String、StringBuffer、StringBuilder。CharSequence与String都能用于定义字符串，但CharSequence的值是可读可写序列，而String的值是只读序列。对于一个抽象类或者是接口类，不能使用new来进行赋值，但是可以通过以下方式来进行实例的创建：

CharSequence cs="hello";

## 方法

### int length()

返回字符的数目

### char charAt(int index)

指定字符的值

### CharSequence subSequence(int start, int end)

指定的子序列

### public default IntStream codePoints()

返回这个序列中的一个Unicode编码流