# 第十章 常用类

学习API更深的意义在于复习面向对象的知识。查看源代码，学习源代码。

## 10.1包装类Wrapper

包装类其实就是将基本数据类型封装成类的一个属性，然后定义了一系列的方法来操作该基本类型的值

以Integer为例：

|  |
| --- |
| **//内部维护的属性**private final int value;  **//构造器**public Integer(int value) {  this.value = value; } |

### 10.1.1 为什么需要包装类

* Java并不是**纯**面向对象的语言。

Java语言是一个面向对象的语言，但是Java中的基本数据类型却是不面向对象的。

基本数据类型有它的优势：性能（效率高，节省空间）。

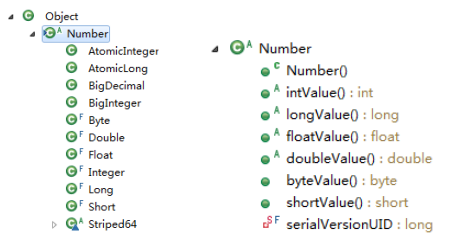
* 使用包装类将基本数据类型转化成对象，便于操作。
  + 集合的操作
  + 使用Object类型接收任意类型的数据等
  + 泛型实参
* 这时，我们就需要将基本数据类型数据转化为对象。

### 10.1.2 包装类

包装类均位于java.lang包，包装类和基本数据类型的对应关系，如图10-1：



其中有六个都是继承自java.lang.Number类：



### 10.1.3自动装箱（auto\_boxing）与自动拆箱（unboxing）

JDK1.5之前需要手动装箱与拆箱，JDK1.5之后支持自动装箱与自动拆箱。

装箱：基本数据类型=>包装类对象

拆箱：包装类对象=>基本数据类型

|  |
| --- |
| public class Main {  public static void main(String[] args) {  //自动装箱  Integer total = 99;  //自定拆箱  int totalprim = total;  }  } |

执行上面那句代码的时候，系统为我们执行了：   
Integer total = Integer.valueOf(99);

int totalprim = total;   
执行上面那句代码的时候，系统为我们执行了：   
int totalprim = total.intValue();

### 10.1.4 缓存问题

现象：

|  |
| --- |
| package \_10\_常用类.包装类;  import org.junit.Test;  public class \_05\_缓存问题 {  @Test  public void test01() {  int a = 1;  int b = 1;  System.*out*.println(a==b); *//true* int c = 130;  int d = 130;  System.*out*.println(c == d ); *//true* Integer e = 1;  Integer f = 1;  System.*out*.println(e==f);*//true 比较的是地址* Integer h = 130;  Integer i = 130;  System.*out*.println(h == i );*//false 比较的是地址*  //todo 1.自动装箱这种情况  // Byte. Short. Integer Long 等都有缓存对象  // 缓存范围为-128~127Integer h1 = new Integer(120);  Integer h2 = new Integer(120);  System.*out*.println(h1 == h2); *//false* Integer a1 = Integer.*valueOf*("2");  Integer a2 = Integer.*valueOf*("2");  System.*out*.println(a1 == a2); *//true* }  } |

当我们使用自动装箱的时候，会发现：

|  |
| --- |
| Integer e = 1;  Integer f = 1;  System.*out*.println(e==f);*//true 比较的是地址* Integer h = 130;  Integer i = 130;  System.*out*.println(h == i );*//false 比较的是地址* |

原因是：

自动装箱valueOf()方法中使用了缓存。

|  |
| --- |
| public static Integer valueOf(int i) {  if (i >= IntegerCache.*low* && i <= IntegerCache.*high*)  return IntegerCache.*cache*[i + (-IntegerCache.*low*)];  return new Integer(i); } |

当自动装箱的基本数据类型在其上下限范围内，就会使用缓存，否则重新new 一个包装类对象。

* IntegerCache类

此类是Integer类的内部类，对于别的包装类也有自己的内部缓存类。

该缓存类定义了：

1. 该数据类型的上下限取值
2. 该包装类的数组 cache[]

在静态代码块中就将cache初始化了。这意味着一旦我们使用过Integer类之后，就会在内存中存在这个数组。而自动

|  |
| --- |
| private static class IntegerCache {  static final int *low* = -128;  static final int *high*;  static final Integer *cache*[];   static {  *// high value may be configured by property* int h = 127;  String integerCacheHighPropValue =  sun.misc.VM.*getSavedProperty*("java.lang.Integer.IntegerCache.high");  if (integerCacheHighPropValue != null) {  try {  int i = *parseInt*(integerCacheHighPropValue);  i = Math.*max*(i, 127);  *// Maximum array size is Integer.MAX\_VALUE* h = Math.*min*(i, Integer.*MAX\_VALUE* - (-*low*) -1);  } catch( NumberFormatException nfe) {  *// If the property cannot be parsed into an int, ignore it.* }  }  *high* = h;   *cache* = new Integer[(*high* - *low*) + 1];  int j = *low*;  for(int k = 0; k < *cache*.length; k++)  *cache*[k] = new Integer(j++);   *// range [-128, 127] must be interned (JLS7 5.1.7)* assert IntegerCache.*high* >= 127;  }   private IntegerCache() {} } |
| public static Integer valueOf(int i) {  if (i >= IntegerCache.*low* && i <= IntegerCache.*high*)  return IntegerCache.*cache*[i + (-IntegerCache.*low*)];  return new Integer(i); } |

* 关于所有包装类，缓存数据范围如下：
* byte Byte -128–127
* short Short -128–127
* int Integer -128—127
* long Long -128—127
* float Float 没有缓存
* double Double 没有缓存
* char Character 0–127
* boolean Boolean true,false

### 10.1.5 包装类的作用

#### 1.构造器

|  |
| --- |
| public Integer(int value) {  this.value = value; }  public Integer(String s) throws NumberFormatException {  this.value = *parseInt*(s, 10); } |

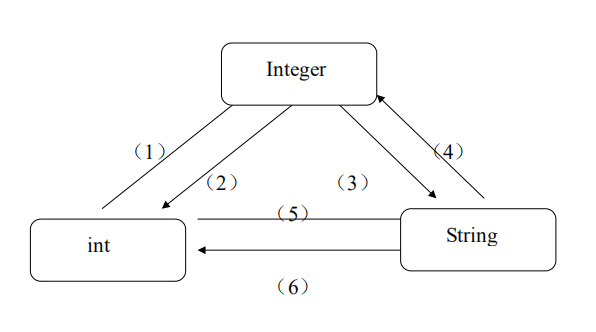
#### 将字符串转为基本数据类型 parseInt

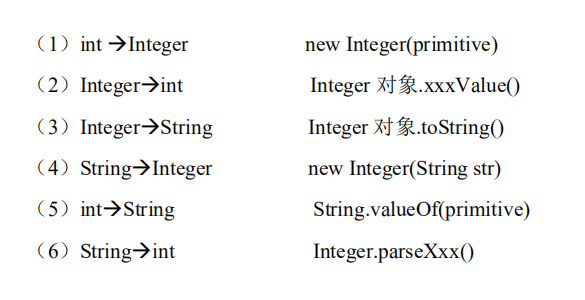
|  |
| --- |
| public static int parseInt(String s) throws NumberFormatException {  return *parseInt*(s,10); }  //构造器，参数为String，底层调用的也是parseInt  public Integer(String s) throws NumberFormatException {  this.value = *parseInt*(s, 10); } |

#### 将字符串转成包装类对象

* 将字符串转为包装类有两种方式：
  + 包装类的构造方法
  + 用包装类的静态方法 valueOf

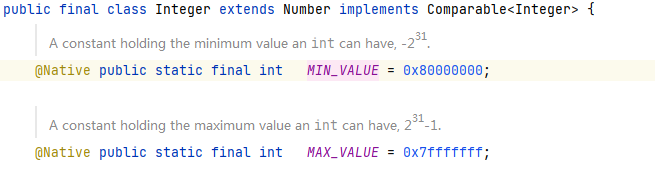
|  |
| --- |
| //1.构造器方式  public Integer(String s) throws NumberFormatException {  this.value = *parseInt*(s, 10); }  //2. valueOf()方式  //param:String  //return: Integer  public static Integer valueOf(String s, int radix) throws NumberFormatException {  return Integer.*valueOf*(*parseInt*(s,radix)); }  public static Integer valueOf(String s) throws NumberFormatException {  return Integer.*valueOf*(*parseInt*(s, 10)); }  底层其实是先转为基本数据类型，再转为Integer类型，所以底层调用的其实都是下面这个方法：  public static Integer valueOf(int i) {  if (i >= IntegerCache.*low* && i <= IntegerCache.*high*)  return IntegerCache.*cache*[i + (-IntegerCache.*low*)];  return new Integer(i); }  结论：valueOf(int i)当i在[-128,127]范围内，都不new 新的Integer对象。  所以：  Integer a1 = Integer.*valueOf*("2"); Integer a2 = Integer.*valueOf*("2"); System.*out*.println(a1 == a2); *//true*  （1）使用包装类型的构造方法  除了Character类型，其他7中类型都有1个构造方法，其参数是字符串类型，可以将字符串转为包装类  例如：  Integer t2=new Integer("500");//参数是字符串，字符串的值是必须对应的数值  Integer t3=new Integer("abc");// java.lang.NumberFormatException: For input string: "abc"  （2）使用包装类的valueOf方法  例如：Integer i=Integer.valueOf("500");  **2、字符串转成基本数据类型**  通过包装类的parseXxx(String s)静态方法  例如：int i=Integer.parseInt("500"); |





#### 4.获取该数据类型的取值范围

* 包装类中有静态常量，可以获取该类型数值的最大最小值：MIN\_VALUE、MAX\_VALUE



* Float和Double中还有正无穷大POSITIVE\_INFINITY、负无穷大NEGATIVE\_INFINITY，还NaN，是Not a Number的缩写。NaN 用于处理计算中出现的错误情况，比如 0.0 除以 0.0 或者求负数的平方根。

程序员可以利用这种定制的 NaN 值中的特定位模式来表达某些诊断信息。

#### 5.基本数据类型转换

|  |
| --- |
| */\*\*  \* Returns the value of this {@code Integer} as a {@code byte}  \* after a narrowing primitive conversion.  \* @jls 5.1.3 Narrowing Primitive Conversions  \*/* public byte byteValue() {  return (byte)value; }  */\*\*  \* Returns the value of this {@code Integer} as a {@code short}  \* after a narrowing primitive conversion.  \* @jls 5.1.3 Narrowing Primitive Conversions  \*/* public short shortValue() {  return (short)value; }  */\*\*  \* Returns the value of this {@code Integer} as an  \* {@code int}.  \*/* public int intValue() {  return value; }  */\*\*  \* Returns the value of this {@code Integer} as a {@code long}  \* after a widening primitive conversion.  \* @jls 5.1.2 Widening Primitive Conversions  \* @see Integer#toUnsignedLong(int)  \*/* public long longValue() {  return (long)value; }  */\*\*  \* Returns the value of this {@code Integer} as a {@code float}  \* after a widening primitive conversion.  \* @jls 5.1.2 Widening Primitive Conversions  \*/* public float floatValue() {  return (float)value; }  */\*\*  \* Returns the value of this {@code Integer} as a {@code double}  \* after a widening primitive conversion.  \* @jls 5.1.2 Widening Primitive Conversions  \*/* public double doubleValue() {  return (double)value; } |

#### 6.包装类的其他方法

1、Integer类型

public static String toBinaryString(int i) //把十进制转成二进制

public static String toHexString(int i) //把十进制转成十六进制

public static String toOctalString(int i) //把十进制转成八进制

2、Character类型

public static char toUpperCase(char ch) //转成大写字母

public static char toLowerCase(char ch) //转成小写字母

其他的查看相关API文档即可

3、equals

按照包装的基本数据类型的值比较

4、compareTo

按照包装的基本数据类型的值比较

## 10.2 字符串String类

### 10.2.1 字符串String类的特点

字符串的学习，有的同学就看看API，记下方法，有的同学看看源代码，还有的同学画画图，自然学的深度是不一样的。

|  |
| --- |
| /\*\*  \* The {@code String} class represents character strings. All  \* string literals in Java programs, such as {@code "abc"}, are  \* implemented as instances of this class.  \* <p>  \* Strings are constant; their values cannot be changed after they  \* are created. String buffers support mutable strings.  \* Because String objects are immutable they can be shared. For example:  \* <blockquote><pre>  \* String str = "abc";  \* </pre></blockquote><p>  \* is equivalent to:  \* <blockquote><pre>  \* char data[] = {'a', 'b', 'c'};  \* String str = new String(data);  \* </pre></blockquote><p>  \* Here are some more examples of how strings can be used:  \* <blockquote><pre>  \* System.out.println("abc");  \* String cde = "cde";  \* System.out.println("abc" + cde);  \* String c = "abc".substring(2,3);  \* String d = cde.substring(1, 2);  \* </pre></blockquote>  \* <p>  \* The class {@code String} includes methods for examining  \* individual characters of the sequence, for comparing strings, for  \* searching strings, for extracting substrings, and for creating a  \* copy of a string with all characters translated to uppercase or to  \* lowercase. Case mapping is based on the Unicode Standard version  \* specified by the {@link java.lang.Character Character} class.  \* <p>  \* The Java language provides special support for the string  \* concatenation operator (&nbsp;+&nbsp;), and for conversion of  \* other objects to strings. String concatenation is implemented  \* through the {@code StringBuilder}(or {@code StringBuffer})  \* class and its {@code append} method.  \* String conversions are implemented through the method  \* {@code toString}, defined by {@code Object} and  \* inherited by all classes in Java. For additional information on  \* string concatenation and conversion, see Gosling, Joy, and Steele,  \* <i>The Java Language Specification</i>. |
| String 类代表字符串。**Java 程序中的所有字符串字面值（如 "abc" ）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。例如：**  String str = "abc";  等效于：  char data[] = {'a', 'b', 'c'};  String str = new String(data);  下面给出了一些如何使用字符串的更多示例：  System.out.println("abc");  String cde = "cde";  System.out.println("abc" + cde);  String c = "abc".substring(2,3);  String d = cde.substring(1, 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。  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。有关字符串串联和转换的更多信息，请参阅 Gosling、Joy 和 Steele 合著的 The Java Language Specification。 |

#### 1、String是个final类

#### 2、String是不可变的字符序列

|  |
| --- |
| **public** **final** **class** String  **implements** java.io.Serializable, Comparable<String>, CharSequence {  /\*\* The value is used for character storage. \*/  **private** **final** **char** value[];  /\*\* Cache the hash code for the string \*/  **private** **int** hash; // Default to 0 |

String对象的字符内容是存储在一个字符数组中的。

private意味着外面无法直接获取字符数组，而且String没有提供value的get和set方法，

final意味着字符数组的引用不可改变，即通过让value指向新的数组对象来实现修改String对象，

而且String也没有提供方法来修改value数组某个元素值，因此字符串的字符数组内容也不可变。

疑问？那么字符串的拼接、字符串的截取、字符串的替换等操作是如何实现的呢？

每次修改都创建一个新的char数组表示修改结果。

#### 3、String对象的创建

String str = “hello”;

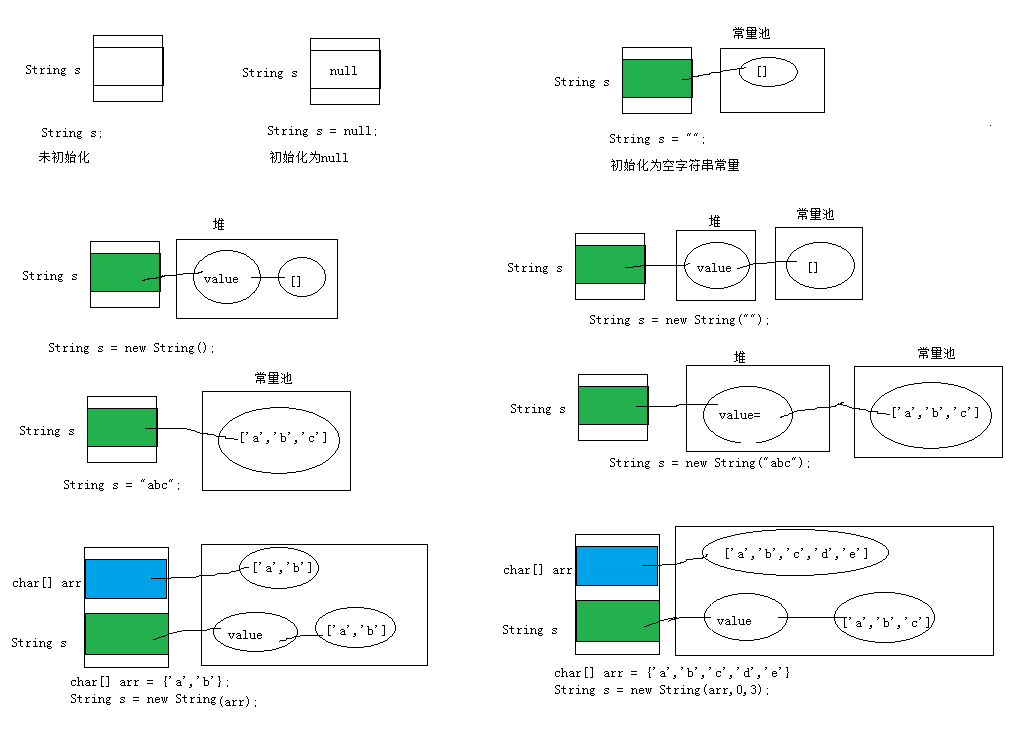
String s1 = new String(); // 本质上 this.value = new char[0];

String s2 = new String(String original); //this.value = original.value;

String s3 = new String(char[] a); //this.value = Arrays.copyOf(value, value.length); //相当于copy了一份，

String s4 = new String(char[] a,int startIndex,int count)

.......



String s = “ ”; 变量s直接指向常量池中的字符串数组

String s = new String(); 变量S指向堆中的value对象，value对象指向堆中字符串数组的地址

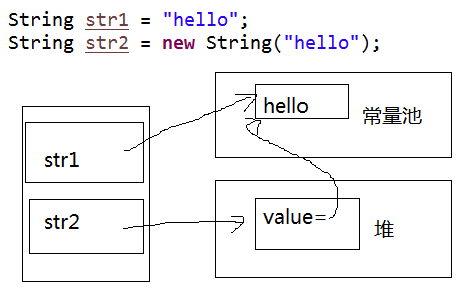
String s = new String(“ ”); 可以分两步看，首先在常量池中有一个字符串 然后堆中创建了String对象，其value属性指向常量池中的地址。

也就是说只要有“”出现，字符串都是存储在常量池中，然后用字符串变量指向

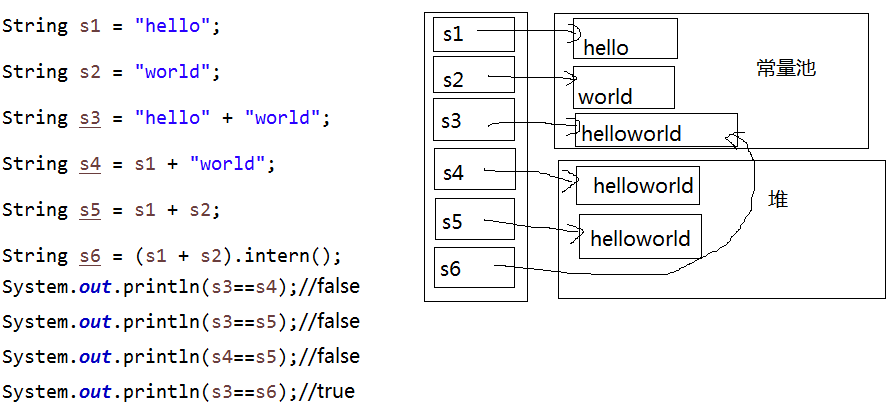
#### 4、字符串对象是如何存储的

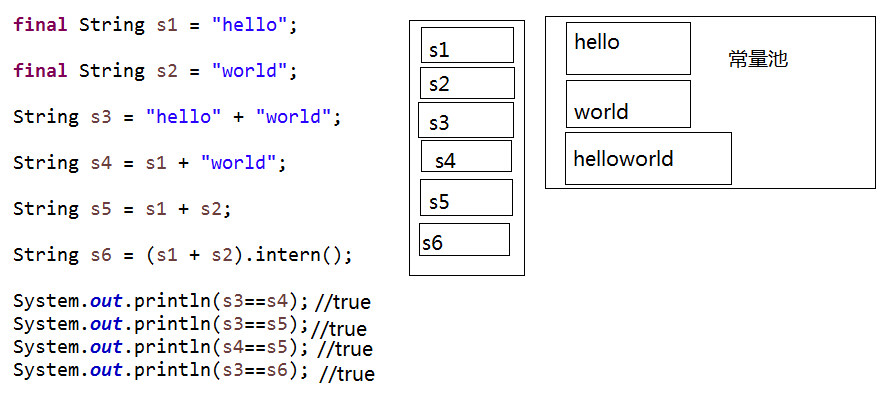
**字符串常量**存储在字符串常量池，目的是共享

**字符串非常量对象**存储在堆中。



#### 5、String的拼接



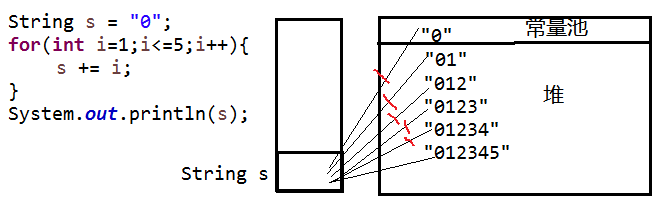


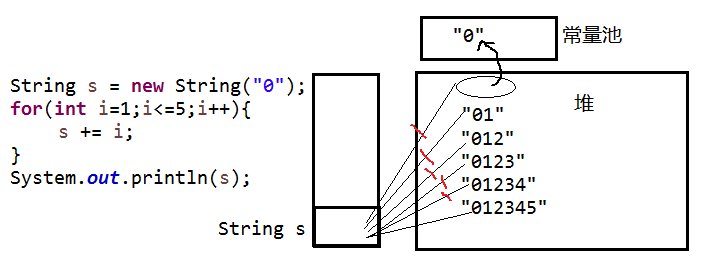
结论：

常量与常量的拼接结果在常量池

只要其中有一个是变量，结果就在堆中

如果拼接的结果调用intern()方法，就在常量池中





#### 6、String对象的比较

==比较的是地址。

equals比较的是字符串的内容，重写了Object的equals方法。

|  |
| --- |
| **public** **boolean** equals(Object anObject) {  **if** (**this** == anObject) {  **return** **true**;  }  **if** (anObject **instanceof** String) {  String anotherString = (String)anObject;  **int** n = value.length;  **if** (n == anotherString.value.length) {  **char** v1[] = value;  **char** v2[] = anotherString.value;  **int** i = 0;  **while** (n-- != 0) {  **if** (v1[i] != v2[i])  **return** **false**;  i++;  }  **return** **true**;  }  }  **return** **false**;  } |

### 10.2.2 String类的常用方法

#### 1、常用方法系列之一

* int length()：返回字符串的长度： return value.length;
* boolean isEmpty()：判断是否是空字符串：return value.length == 0;
* 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。
* 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。
* String trim()：返回字符串的副本，忽略前导空白和尾部空白。
* boolean equals(Object obj)：比较字符串的内容
* boolean equalsIgnoreCase(String anotherString)：与equals方法类似，忽略大小写
* String concat(String str)：将指定字符串连接到此字符串的结尾。 等价于用“+”

#### 2、String类和字符相关操作

* char charAt(int index)： 返回某索引处的字符return value[index];
* char[] toCharArray()：将此字符串转换为一个新的字符数组
* 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。
* 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。

#### 3、String类字节与字符串操作方法

编码：把字符-->字节

* 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。
* byte[] getBytes(Charset charset) ：使用给定的 charset 将此 String 编码到 byte 序列，并将结果存储到新的 byte 数组。
* 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。

解码：把字节-->字符

* 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。
* String(byte[] bytes, Charset charset)：通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。
* 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。
* String(byte[] bytes, int offset, int length, Charset charset)：通过使用指定的 charset 解码指定的 byte 子数组，构造一个新的 String。
* String(byte[] bytes, int offset, int length, String charsetName)：通过使用指定的字符集解码指定的 byte 子数组，构造一个新的 String。
* 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。

|  |
| --- |
| **public** **static** **void** main(String[] args)**throws** Exception {  String str = "中";  System.***out***.println(str.getBytes("ISO8859-1").length);//-128~127  System.***out***.println(str.getBytes("GBK").length);  System.***out***.println(str.getBytes("UTF-8").length);    System.***out***.println(**new** String(str.getBytes("ISO8859-1"),"ISO8859-1"));//乱码，表示不了中文  System.***out***.println(**new** String(str.getBytes("GBK"),"GBK"));  System.***out***.println(**new** String(str.getBytes("UTF-8"),"UTF-8"));  } |
| **public** **static** **void** main(String[] args)**throws** Exception {  String str = "中";  **byte**[] bytes = str.getBytes("GBK");//2个字节 客户端编码  String string = **new** String(bytes,"ISO8859-1");//尝试转成2个字符，错 tomcat服务器6默认尝试用ISO8859-1解码  //在servlet中用代码处理  **byte**[] bytes2 = string.getBytes("ISO8859-1");//重新变回2个字节 再次用ISO8859-1编码  String string2 = **new** String(bytes2,"GBK");//再变成1个字 再次用GBK解码  System.***out***.println(string2);  } |
| **public** **static** **void** main(String[] args)**throws** Exception {  String str = "中";  **byte**[] bytes = str.getBytes("GBK");//2个字节 浏览器客户端编码  String string = **new** String(bytes,"UTF-8");//尝试转成2个字符，错 好比tomcat服务器8默认尝试用UTF-8解码  System.***out***.println("string = " + string);//乱码    //在servlet中用Java代码处理  **byte**[] bytes2 = string.getBytes("UTF-8");//重新解码变成6个字节 再次用UTF-8编码  String string2 = **new** String(bytes2,"GBK");//尝试变成3个字 再次用GBK解码  System.***out***.println(string2);//乱码  } |

#### 4、String类判断是否以指定内容开头或结尾

* boolean endsWith(String suffix)：测试此字符串是否以指定的后缀结束。
* boolean startsWith(String prefix)：测试此字符串是否以指定的前缀开始。
* 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。

#### 5、String类字符串查找操作

* 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。
* 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。
* 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。
* 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。
* 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。
* 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。
* 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。
* 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。
* 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。

indexOf和lastIndexOf方法如果未找到都是返回-1

#### 6、String类字符串截取操作

* 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。

* 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。

#### 7、String类是否匹配正则

* boolean matches(String regex)：告知此字符串是否匹配给定的正则表达式。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  String str = "12345";  //判断str字符串中是否全部有数字组成，即有1-n个数字组成  **boolean** matches = str.matches("\\d+");  System.***out***.println(matches);  String tel = "0571-4534289";  //判断这是否是一个杭州的固定电话  **boolean** result = tel.matches("0571-\\d{7,8}");  System.***out***.println(result);  } |

#### 8、String类替换操作

* 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。

* 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。

* 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。

* 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。

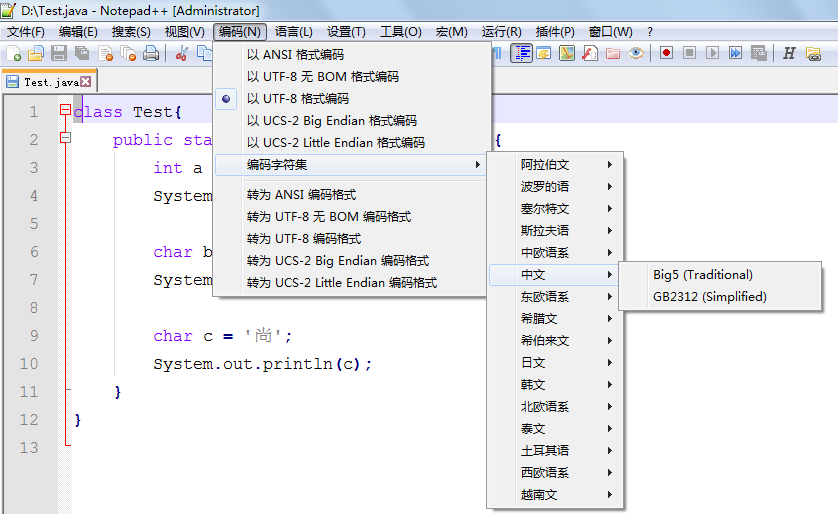
|  |
| --- |
| **public** **static** **void** main(String[] args) {  String str = "12hello34world5java7891mysql456";  //把字符串中的数字替换成,，如果结果中开头和结尾有，的话去掉  String string = str.replaceAll("\\d+", ",").replaceAll("^,|,$", "");  System.***out***.println(string);  } |

#### 9、String类字符串拆分操作

* String[] split(String regex)：根据给定正则表达式的匹配拆分此字符串。
* 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，最多不超过limit个，如果超过了，剩下的全部都放到最后一个元素中。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  String str = "hello|world|java";  String[] strings = str.split("\\|");  **for** (String string : strings) {  System.***out***.println(string);  }  String str2 = "hello.world.java";  String[] strings2 = str2.split("\\.");  **for** (String string : strings2) {  System.***out***.println(string);  }  } |

#### 10、附录：字符编码（了解）



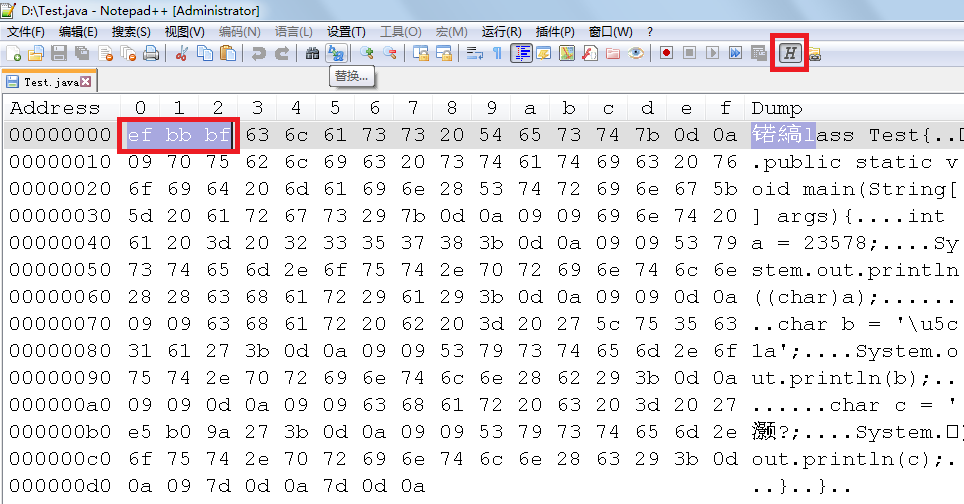
首先来看一下常用的编码有哪些，截图自Notepad++。其中ANSI在中国大陆即为GBK（以前是GB2312），最常用的是 GBK 和 UTF8无BOM 编码格式。后面三个都是有BOM头的文本格式，UCS-2即为人们常说的Unicode编码，又分为大端、小端。

所谓BOM头（Byte Order Mark）就是文本文件中开始的几个并不表示任何字符的字节，用二进制编辑器（如bz.exe）就能看到了。

UTF8的BOM头为 0xEF 0xBB 0xBF

Unicode大端模式为 0xFE 0xFF

Unicode小端模式为 0xFF 0xFE



##### ASCII码

计算机一开始发明的时候是用来解决数字计算的问题，后来人们发现，计算机还可以做更多的事，例如文本处理。但由于计算机只识“数”，因此人们必须告诉计算机哪个数字来代表哪个特定字符，例如65代表字母‘A’，66代表字母‘B’，以此类推。但是计算机之间字符-数字的对应关系必须得一致，否则就会造成同一段数字在不同计算机上显示出来的字符不一样。因此美国国家标准协会ANSI制定了一个标准，规定了常用字符的集合以及每个字符对应的编号，这就是ASCII字符集（Character Set），也称ASCII码。

那时候的字符编解码系统非常简单，就是简单的查表过程。例如将字符序列编码为二进制流写入存储设备，只需要在ASCII字符集中依次找到字符对应的字节，然后直接将该字节写入存储设备即可。解码二进制流的过程也是类似。

其中：

* 0～31及127(共33个)是控制字符或通信专用字符（其余为可显示字符），如控制符：LF（换行）、CR（回车）、FF（换页）、DEL（删除）、BS（退格)
* 32～126(共95个)是字符(32是空格），其中48～57为0到9十个阿拉伯数字。
* 65～90为26个大写英文字母，97～122号为26个小写英文字母，其余为一些标点符号、运算符号等。
* 后128个称为扩展ASCII码。许多基于x86的系统都支持使用扩展（或“高”）ASCII。扩展ASCII 码允许将每个字符的第8 位用于确定附加的128 个特殊符号字符、外来语字母和图形符号。

##### OEM字符集的衍生

当计算机开始发展起来的时候，人们逐渐发现，ASCII字符集里那可怜的128个字符已经不能再满足他们的需求了。人们就在想，一个字节能够表示的数字（编号）有256个，而ASCII字符只用到了0x00~0x7F，也就是占用了前128个，后面128个数字不用白不用，因此很多人打起了后面这128个数字的主意。可是问题在于，很多人同时有这样的想法，但是大家对于0x80-0xFF这后面的128个数字分别对应什么样的字符，却有各自的想法。这就导致了当时销往世界各地的机器上出现了大量各式各样的OEM字符集。

大家对于0x00~0x7F这个范围的解释基本是相同的，而对于后半部分0x80~0xFF的解释却不一定相同。甚至有时候同样的字符在不同OEM字符集中对应的字节也是不同的。

不同的OEM字符集导致人们无法跨机器交流各种文档。例如职员甲发了一封简历résumés给职员乙，结果职员乙看到的却是r?sum?s，因为é字符在职员甲机器上的OEM字符集中对应的字节是0x82，而在职员乙的机器上，由于使用的OEM字符集不同，对0x82字节解码后得到的字符却是?。

##### 多字节字符集（MBCS）和中文字符集

上面我们提到的字符集都是基于单字节编码，也就是说，一个字节翻译成一个字符。这对于拉丁语系国家来说可能没有什么问题，因为他们通过扩展第8个比特，就可以得到256个字符了，足够用了。但是对于亚洲国家来说，256个字符是远远不够用的。因此这些国家的人为了用上电脑，又要保持和ASCII字符集的兼容，就发明了多字节编码方式，相应的字符集就称为多字节字符集（Muilti-Bytes Charecter Set）。例如中国使用的就是双字节字符集编码。

例如目前最常用的中文字符集GB2312，涵盖了所有简体字符以及一部分其他字符；GBK（K代表扩展的意思）则在GB2312的基础上加入了对繁体字符等其他非简体字符。这两个字符集的字符都是使用1-2个字节来表示。Windows系统采用936代码页来实现对GBK字符集的编解码。在解析字节流的时候，如果遇到字节的最高位是0的话，那么就使用936代码页中的第1张码表进行解码，这就和单字节字符集的编解码方式一致了。如果遇到字节的最高位是1的话，那么就表示需要两个字节值才能对应一个字符。

假如你使用GB2312写了这么一句话：

我叫ABC

它的二进制编码是这样的：

11001110 11010010 10111101 11010000 01000001 01000002 01000003

全角？

全角是一种电脑字符，且每个全角字符占用两个标准字符（或半角字符）位置。通常的英文字母、数字键、符号键都是半角的，半角的显示内码都是一个字节。为了排列整齐，英文和其它拉丁文的字符和标点也提供了全角格式。在中文输入法中，切换全角和半角格式的快捷键为SHIFT+空格。

##### ANSI标准、国家标准、ISO标准

不同ASCII衍生字符集的出现，让文档交流变得非常困难，因此各种组织都陆续进行了标准化流程。例如美国ANSI组织制定了ANSI标准字符编码（注意，我们现在通常说到ANSI编码，通常指的是平台的默认编码，例如英文操作系统中是ISO-8859-1，中文系统是GBK），ISO组织制定的各种ISO标准字符编码，还有各国也会制定一些国家标准字符集，例如中国的GBK，GB2312和GB18030。

操作系统在发布的时候，通常会往机器里预装这些标准的字符集还有平台专用的字符集，这样只要你的文档是使用标准字符集编写的，通用性就比较高了。例如你用GB2312字符集编写的文档，在中国大陆内的任何机器上都能正确显示。同时，我们也可以在一台机器上阅读多个国家不同语言的文档了，前提是本机必须安装该文档使用的字符集。

##### Unicode的出现

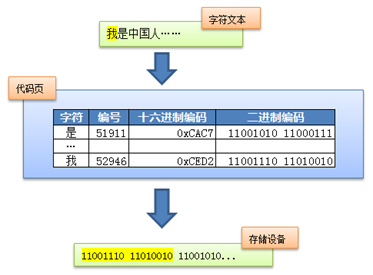
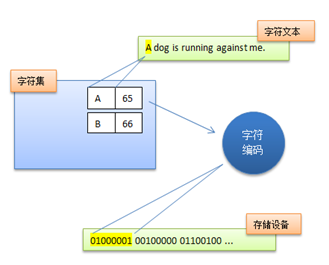
虽然通过使用不同字符集，我们可以在一台机器上查阅不同语言的文档，但是我们仍然**无法解决一个问题：如果一份文档中含有不同国家的不同语言的字符，那么无法在一份文档中显示所有字符**。为了解决这个问题，我们需要一个全人类达成共识的巨大的字符集，这就是Unicode字符集。

Unicode字符集涵盖了目前人类使用的所有字符，并为每个字符进行统一编号，分配唯一的字符码（Code Point）。Unicode字符集将所有字符按照使用上的频繁度划分为17个层面（Plane），每个层面上有216=65536个字符码空间。

其中第0个层面BMP，基本涵盖了当今世界用到的所有字符。其他的层面要么是用来表示一些远古时期的文字，要么是留作扩展。我们平常用到的Unicode字符，一般都是位于BMP层面上的。目前Unicode字符集中尚有大量字符空间未使用。

##### 编码系统的变化

**在Unicode出现之前，所有的字符集都是和具体编码方案绑定在一起的（即字符集≈编码方式）**，都是直接将字符和最终字节流绑定死了，例如ASCII编码系统规定使用7比特来编码ASCII字符集；GB2312以及GBK字符集，限定了使用**最多**2个字节来编码所有字符，并且规定了字节序。这样的编码系统通常用简单的查表，也就是通过代码页就可以直接将字符映射为存储设备上的字节流了。例如下面这个例子：

Unicode同样也不完美，这里就有三个的问题，一个是，我们已经知道，英文字母只用一个字节表示就够了，第二个问题是如何才能区别Unicode和ASCII？计算机怎么知道两个字节表示一个符号，而不是分别表示两个符号呢？第三个，如果和GBK等双字节编码方式一样，用最高位是1或0表示两个字节和一个字节，就少了很多值无法用于表示字符，不够表示所有字符。Unicode在很长一段时间内无法推广，直到互联网的出现，为解决Unicode如何在网络上传输的问题，于是面向传输的众多 UTF（UCS Transfer Format）标准出现了，顾名思义，UTF-8就是每次8个位传输数据，而UTF-16就是每次16个位。UTF-8就是在互联网上使用最广的一种Unicode的实现方式，这是为传输而设计的编码，并使编码无国界，这样就可以显示全世界上所有文化的字符了。

UTF-8最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号。从unicode到uft-8并不是直接的对应，而是要过一些算法和规则来转换（**即Uncidoe字符集≠UTF-8编码方式**）。

Unicode符号范围 | UTF-8编码方式

(十六进制) | （二进制）

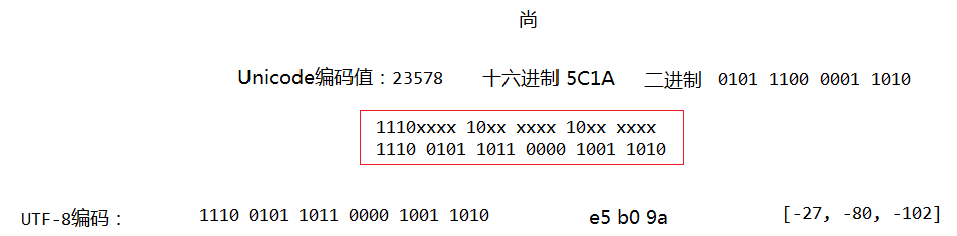
—————————————————————–

0000 0000-0000 007F | 0xxxxxxx（兼容原来的ASCII）

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx



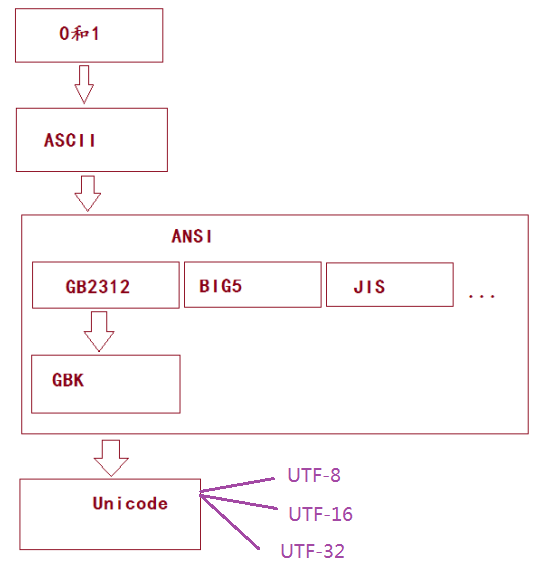
因此，Unicode只是定义了一个庞大的、全球通用的字符集，并为每个字符规定了唯一确定的编号，具体存储成什么样的字节流，取决于字符编码方案。推荐的Unicode编码是UTF-16和UTF-8。

早期字符编码、字符集和代码页等概念都是表达同一个意思。例如GB2312字符集、GB2312编码，936代码页，实际上说的是同个东西。

但是对于Unicode则不同，Unicode字符集只是定义了字符的集合和唯一编号，Unicode编码，则是对UTF-8、UCS-2/UTF-16等具体编码方案的统称而已，并不是具体的编码方案。所以当需要用到字符编码的时候，你可以写gb2312，codepage936，utf-8，utf-16，但请不要写Unicode。

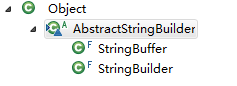
造成乱码的原因就是因为使用了错误的字符编码去解码字节流，因此当我们在思考任何跟文本显示有关的问题时，请时刻保持清醒：当前使用的字符编码是什么。只有这样，我们才能正确分析和处理乱码问题。

常见CharSet有：GBK、GB2312、US-ASCII、ISO-8859-1、UTF-8、UTF-16BE、UTF-16LE、UTF-16

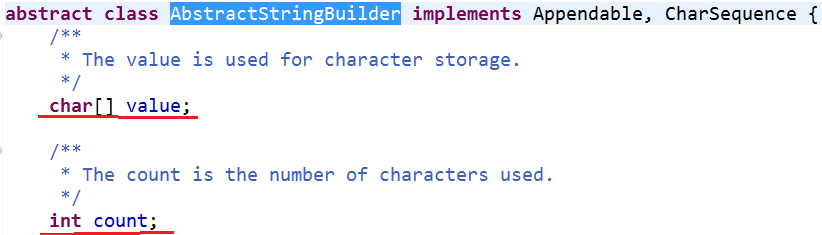


## 10.3 可变字符序列：字符串缓冲区

### 10.3.1 StringBuffer、StringBuilder、String的区别



1、一个类似于 String 的字符串缓冲区，但能被修改。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，所有对StringBuffer或StringBuilder对象的字符序列的修改不会产生新的StringBuffer或StringBuilder对象，这点和String很大的不同。



**value没有final声明,value可以不断扩容，count记录有效字符的个数。**

**2、StringBuilder：线程安全，效率低；StringBuffer：线程不安全，效率高**

### 10.3.2 StringBuffer和StringBuilder不同的API

#### 1、构造器

StringBuffer或StringBuilder对象的创建必须用new+构造器。

StringBuffer() 初始容量为16的字符串缓冲区

StringBuffer(int size) 构造指定容量的字符串缓冲区

StringBuffer(String str) 将内容初始化为指定字符串内容

#### 2、和字符序列改变有关的方法

而且所有这些方法支持方法链

StringBuffer append(xx)：提供了很多的append()方法，用于进行字符串拼接

StringBuffer delete(int start,int end)：删除指定位置的内容

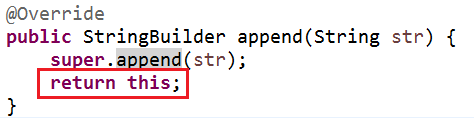
StringBuffer insert(int offset, xx)：在指定位置插入xx

StringBuffer replace(int start, int end, String str)：把[start,end)位置替换为str

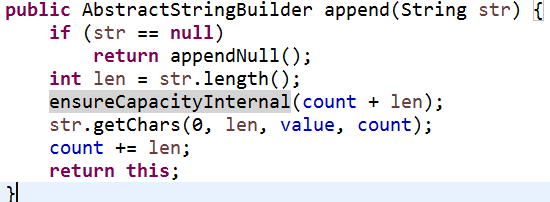
StringBuffer reverse() ：把当前字符序列逆转

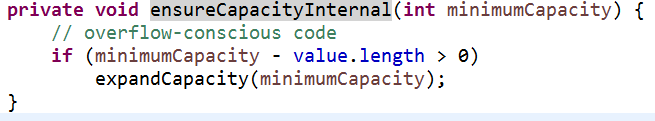
|  |
| --- |
| **public** **static** **void** main(String[] args) {  StringBuffer sf = **new** StringBuffer();  sf.append("hello").append("world").append("java");  System.***out***.println(sf);  } |

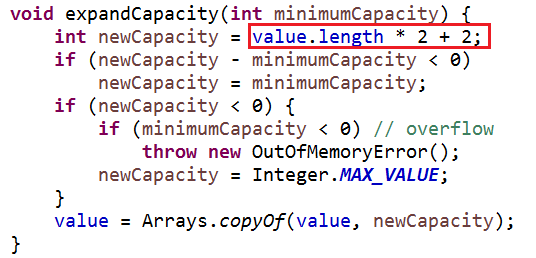
**方法链的原理：StringBuffer对象进行方法操作后，得到的是操作后的对象，所以可以进行连锁调用方法。**



当append和insert时，如果原来value数组长度不够，可扩容







### 10.3.3 效率测试

|  |
| --- |
| **package** com.api.string;  **public** **class** TestStringBuilder {  **public** **static** **void** main(String[] args) {  *testStringBuilder*();  *testStringBuffer*();  *testString*();  }  **public** **static** **void** testString(){  **long** start = System.*currentTimeMillis*();  String s = **new** String("0");  **for**(**int** i=1;i<=10000;i++){  s += i;  }  System.***out***.println(s);  **long** end = System.*currentTimeMillis*();  System.***out***.println("String拼接+用时："+(end-start));  **long** memory = Runtime.*getRuntime*().totalMemory() - Runtime.*getRuntime*().freeMemory();  System.***out***.println("String拼接+memory: " + memory);  }  **public** **static** **void** testStringBuilder(){  **long** start = System.*currentTimeMillis*();  StringBuilder s = **new** StringBuilder("0");  **for**(**int** i=1;i<=10000;i++){  s.append(i);  }  System.***out***.println(s);  **long** end = System.*currentTimeMillis*();  System.***out***.println("StringBuilder拼接+用时："+(end-start));  **long** memory = Runtime.*getRuntime*().totalMemory() - Runtime.*getRuntime*().freeMemory();  System.***out***.println("StringBuilder拼接+memory: " + memory);  }  **public** **static** **void** testStringBuffer(){  **long** start = System.*currentTimeMillis*();  StringBuffer s = **new** StringBuffer("0");  **for**(**int** i=1;i<=10000;i++){  s.append(i);  }  System.***out***.println(s);  **long** end = System.*currentTimeMillis*();  System.***out***.println("StringBuffer拼接+用时："+(end-start));  **long** memory = Runtime.*getRuntime*().totalMemory() - Runtime.*getRuntime*().freeMemory();  System.***out***.println("StringBuffer拼接+memory: " + memory);  }  } |

## 10.4 System系统类

System 类包含一些有用的类字段和方法。它不能被实例化。

在 System 类提供的设施中，有标准输入、标准输出和错误输出流；对外部定义的属性和环境变量的访问；加载文件和库的方法；还有快速复制数组的一部分的实用方法。

### 10.4.1 标准输入、标准输出和错误输出流对象

PrintStream err：“标准”错误输出流。

InputStream in：“标准”输入流。

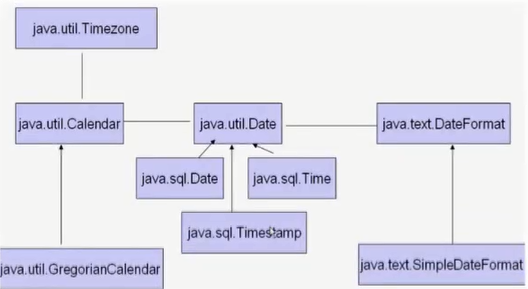
PrintStream out：“标准”输出流。

|  |
| --- |
| import java.util.Scanner;  class TestSystem{  public static void main(String[] args){  Scanner input = new Scanner(System.in);  System.out.print("请输入成绩：");  int score = input.nextInt();    if(score>=0 && score<=100){  System.out.println("成绩是：" + score);  }else{  System.err.println("输入有误，成绩应该在[0,100]");  }  }  } |

### 10.4.2 常用方法

* void arraycopy(Object src, int srcPos, Object dest, int destPos, int length) ：从指定源数组中复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。
* long currentTimeMillis() ：返回以毫秒为单位的当前时间。
* void exit(int status) ：终止当前正在运行的 Java 虚拟机。
* void gc()：运行垃圾回收器。
* Properties getProperties()：确定当前的系统属性。
* String getProperty(String key)：获取指定键指示的系统属性。
* void setErr(PrintStream err) ：重新分配“标准”错误输出流。
* void setIn(InputStream in)：重新分配“标准”输入流。
* void setOut(PrintStream out)：重新分配“标准”输出流。
* String setProperty(String key, String value) ：设置指定键指示的系统属性。

## 10.5 JDK1.8之前日期时间类



### 10.5.1 java.lang.System类

System类提供的public static long currentTimeMillis()用来返回当前时间与1970年1月1日0时0分0秒GMT之间以毫秒为单位的时间差。此方法适于计算时间差。

### 10.5.2 java.util.Date

它的对象表示一个特定的瞬间，精确到毫秒。

Java中时间的表示说白了也是数字，是从标准纪元1970年1月1日0时0分0秒GMT到某个时刻的毫秒数，类型是long

理解：一维的时间轴，选择1970年1月1日0时0分0秒时间为0刻度，1毫秒一刻度

构造方法：

* Date()： 源代码：this(System.currentTimeMillis());
* Date(long date)

常用方法：

* getTime():返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。
* toString():把此 Date 对象转换为以下形式的 String： dow mon dd hh:mm:ss zzz yyyy 其中：

dow 是一周中的某一天 (Sun, Mon, Tue, Wed, Thu, Fri, Sat)， zzz是时间标准。

已过时的方法：

* public Date(int year,int month,int date,int hrs,int min,int sec)

参数：

year - 减 1900 的年份。

month - 0-11 之间的月份。

date - 一月中 1-31 之间的某一天。

hrs - 0-23 之间的小时数。

min - 0-59 之间的分钟数。

sec - 0-59 之间的秒数。

* getYear() 从 JDK 1.1 开始，由 Calendar.get(Calendar.YEAR) - 1900 取代。
* getMonth()从 JDK 1.1 开始，由 Calendar.get(Calendar.MONTH) 取代。返回的值在 0 和 11 之间，值 0 表示 1 月。
* getDate() JDK 1.1 开始，由 Calendar.get(Calendar.DAY\_OF\_MONTH) 取代。返回的值在 1 和 31 之间
* getDay()从 JDK 1.1 开始，由 Calendar.get(Calendar.DAY\_OF\_WEEK) 取代。 返回值 (0 = Sunday, 1 = Monday, 2 = Tuesday, 3 = Wednesday, 4 = Thursday, 5 = Friday, 6 = Saturday) 表示一周中的某一天
* getHours()从 JDK 1.1 开始，由 Calendar.get(Calendar.HOUR\_OF\_DAY) 取代。返回值是一个数字（0 至 23）
* getMinutes()从 JDK 1.1 开始，由 Calendar.get(Calendar.MINUTE) 取代。 返回值在 0 和 59 之间。
* getSeconds()从 JDK 1.1 开始，由 Calendar.get(Calendar.SECOND) 取代。回的值在 0 和 61 之间。值 60 和 61 只可能发生在考虑了闰秒的 Java 虚拟机上。

### 10.5.3 java.util.TimeZone和Locale

Locale 对象表示了特定的地理、政治和文化地区。需要 Locale 来执行其任务的操作称为语言环境敏感的 操作，它使用 Locale 为用户量身定制信息。例如，显示一个数值，日期就是语言环境敏感的操作，应该根据用户的国家、地区或文化的风俗/传统来格式化该数值。

获取Locale对象：

Locale(String language)

Locale(String language, String country)

Locale.CHINA、Locale.US等

|  |
| --- |
| public static void main(String[] args) {  Locale loc = Locale.CHINA;  System.out.println(loc);//zh\_CN  System.out.println(Locale.US);//en\_US  System.out.println(Locale.JAPAN);//ja\_JP    Locale c = new Locale("zh","CN");  System.out.println(c);  } |

通常，使用 TimeZone的getDefault 获取 TimeZone，getDefault 基于程序运行所在的时区创建 TimeZone。例如，对于在日本运行的程序，getDefault 基于日本标准时间创建 TimeZone 对象。

也可以用TimeZone的 getTimeZone 及时区 ID 获取 TimeZone 。例如美国太平洋时区的时区 ID 是 "America/Los\_Angeles"。因此，可以使用下面语句获得美国太平洋时间 TimeZone 对象：

TimeZone tz = TimeZone.getTimeZone("America/Los\_Angeles");

|  |
| --- |
| public static void main(String[] args) {  TimeZone t = TimeZone.getDefault();  System.out.println(t);    TimeZone t1 = TimeZone.getTimeZone("America/New\_York");  System.out.println(t1);    TimeZone t2 = TimeZone.getTimeZone("Asia/Shanghai");  System.out.println(t2);  } |
| String[] all = TimeZone.getAvailableIDs();  for (String id : all) {  System.out.println(id);  } |

### 10.5.4 java.util.Calendar

Calendar 类是一个抽象类，它为特定瞬间与一组诸如 YEAR、MONTH、DAY\_OF\_MONTH、HOUR 等 日历字段之间的转换提供了一些方法，并为操作日历字段（例如获得下星期的日期）提供了一些方法。瞬间可用毫秒值来表示，它是距历元（即格林威治标准时间 1970 年 1 月 1 日的 00:00:00.000，格里高利历）的偏移量。

人们对于时间的认识是：某年某月某日，这样的日期概念。计算机是long类型的数字。通过Calendar在二者之间搭起桥梁。而且Calendar提供了很多关于日期时间计算的方法。

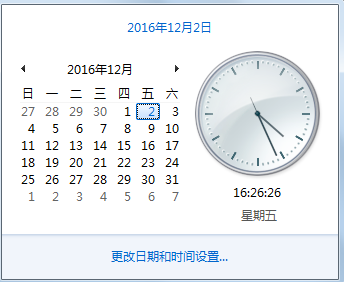


GregorianCalendar（公历）是Calendar的一个具体子类，提供了世界上大多数国家/地区使用的标准日历系统。

注意：

月份：一月是0，二月是1，以此类推，12月是11

星期：周日是1，周二是2,。。。。周六是7



|  |
| --- |
| //默认语言环境的时间（时区）  Calendar c = new GregorianCalendar();  /\*  \* java.util.GregorianCalendar[  \* time=1480667849712,  \* areFieldsSet=true,  \* areAllFieldsSet=true,  \* lenient=true,  \* zone=sun.util.calendar.ZoneInfo[id="Asia/Shanghai",offset=28800000,dstSavings=0,useDaylight=false,transitions=19,lastRule=null],  \* firstDayOfWeek=1,  \* minimalDaysInFirstWeek=1,  \* ERA=1,  \* YEAR=2016,  \* MONTH=11,  \* WEEK\_OF\_YEAR=49,//本年第49周  \* WEEK\_OF\_MONTH=1,//本月第1周  \* DAY\_OF\_MONTH=2,  \* DAY\_OF\_YEAR=337,//本年第337天  \* DAY\_OF\_WEEK=6,  \* DAY\_OF\_WEEK\_IN\_MONTH=1,  \* AM\_PM=1, //下午  \* HOUR=4,  \* HOUR\_OF\_DAY=16, //HOUR是12小时制， HOUR\_OF\_DAY是24小时制  \* MINUTE=37,  \* SECOND=29,  \* MILLISECOND=712,  \* ZONE\_OFFSET=28800000,  \* DST\_OFFSET=0]  \*/ |
| **public** **static** **void** main(String[] args) {  //默认语言环境的时间（时区）  Calendar c = **new** GregorianCalendar();    **int** year=c.get(Calendar.***YEAR***);  **int** month=c.get(Calendar.***MONTH***);  **int** date=c.get(Calendar.***DAY\_OF\_MONTH***);    **int** hour=c.get(Calendar.***HOUR\_OF\_DAY***);  **int** minute=c.get(Calendar.***MINUTE***);  **int** second=c.get(Calendar.***SECOND***);  **int** mill=c.get(Calendar.***MILLISECOND***);  **int** week=c.get(Calendar.***DAY\_OF\_WEEK***);    StringBuffer dateStr=**new** StringBuffer();  dateStr.append(year).append("年");  dateStr.append(month+1).append("月");  dateStr.append(date).append("日").append(" ");  dateStr.append(hour).append("时");  dateStr.append(minute).append("分");  dateStr.append(second).append("秒");  dateStr.append(mill).append("毫秒").append(" ");    String[] weeks={"日","一","二","","四","五","六"};  dateStr.append("星期").append(weeks[week-1]);    System.***out***.println(dateStr);  } |
| public static void main(String[] args) {  Calendar c = new GregorianCalendar(2015, 6, 13);  // c.set(2016, Calendar.DECEMBER, 4, 12, 12, 0);  // c.setTime(new Date());  //15天之后  //c.add(Calendar.DATE, 15);  //2个月之前  //c.add(Calendar.DAY\_OF\_MONTH, -2);  //12小时之后  c.add(Calendar.HOUR, 12);    Date time = c.getTime();//转成日期  System.out.println(time);  } |

public static Calendar getInstance()使用默认时区和语言环境获得一个日历。返回的 Calendar 基于当前时间，使用了默认时区和默认语言环境。

public static Calendar getInstance(TimeZone zone, Locale aLocale)使用指定时区和语言环境获得一个日历。返回的 Calendar 基于当前时间，使用了给定的时区和给定的语言环境。

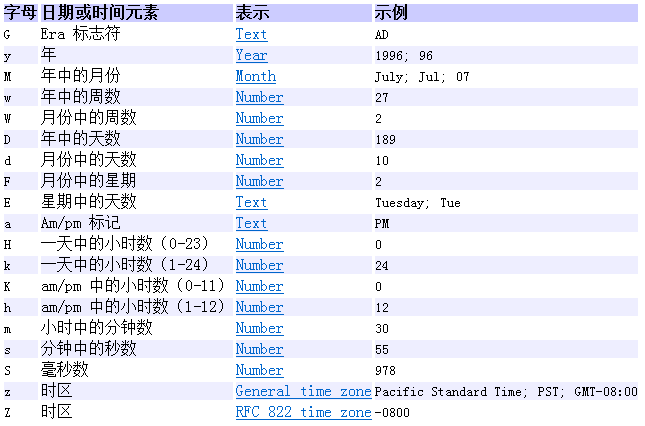
|  |
| --- |
| public static void main(String[] args) {  Calendar c = Calendar.getInstance();  System.out.println(c);    Calendar c2 = Calendar.getInstance(TimeZone.getTimeZone("Asia/Shanghai"), Locale.CHINA);  System.out.println(c2);    Calendar c3 = Calendar.getInstance(TimeZone.getTimeZone("America/New\_York"), Locale.US);  System.out.println(c3);  } |

### 10.5.5 java.text.DateFormat和SimpleDateFormat

完成字符串和时间对象的转化：

* String format(date)
* Date parse(string)

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Date date = **new** Date();  SimpleDateFormat sf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss 是本年的第几D");  System.***out***.println(sf.format(date));    String s = "2016-12-01 14:12:23";  SimpleDateFormat sf2 = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  **try** {  Date d = sf2.parse(s);  System.***out***.println(d);  } **catch** (ParseException e) {  e.printStackTrace();  }  } |



## 10.6 数学相关类Math、BigInteger、BigDecimal

### 10.6.1 java.lang.Math类

java.lang.Math提供了一系列静态方法用于科学计算；其方法的参数和返回值类型一般为double型。

* abs 绝对值
* acos,asin,atan,cos,sin,tan 三角函数
* sqrt 平方根
* pow(double a,doble b) a的b次幂
* log 自然对数
* exp e为底指数
* max(double a,double b)
* min(double a,double b)
* random() 返回0.0到1.0的随机数
* long round(double a) double型数据a转换为long型（四舍五入）
* toDegrees(double angrad) 弧度—>角度
* toRadians(double angdeg) 角度—>弧度

|  |
| --- |
| @Test  **public** **void** test1() {  System.***out***.println(Math.*random*());//随机值  System.***out***.println(Math.*round*(1.8));//四舍五入 保留整数部分  System.***out***.println(Math.*floor*(1.2));//1.0 向下取  System.***out***.println(Math.*ceil*(1.2));//2.0 向上取  System.***out***.println(Math.*floor*(-2.4));//-3.0  System.***out***.println(Math.*ceil*(-2.4));//-2.0  } |

### 10.6.2 java.math包的BigInteger和BigDecimal

Integer类作为int的包装类，能存储的最大整型值为231-1，Long类也是有限的，最大为263-1如果要表示再大的整数，不管是基本数据类型还是他们的包装类都无能为力，更不用说进行运算了。

java.math包的BigInteger可以表示不可变的任意精度的整数。BigInteger 提供所有 Java 的基本整数操作符的对应物，并提供 java.lang.Math 的所有相关方法。另外，BigInteger 还提供以下运算：模算术、GCD 计算、质数测试、素数生成、位操作以及一些其他操作。

* 构造方法
  + BigInteger(String val)：根据字符串构建BigInteger对象
* 常用方法
  + BigInteger add(BigInteger val) ：返回其值为 (this + val) 的 BigInteger。
  + BigInteger subtract(BigInteger val) ：返回其值为 (this - val) 的 BigInteger。
  + BigInteger multiply(BigInteger val) ：返回其值为 (this \* val) 的 BigInteger。
  + BigInteger divide(BigInteger val) ：返回其值为 (this / val) 的 BigInteger。整数相除只保留整数部分。
  + BigInteger remainder(BigInteger val) ：返回其值为 (this % val) 的 BigInteger。
  + BigInteger[] divideAndRemainder(BigInteger val)：返回包含 (this / val) 后跟 (this % val) 的两个 BigInteger 的数组。
  + BigInteger pow(int exponent) ：返回其值为 (thisexponent) 的 BigInteger。

|  |
| --- |
| @Test  public void test2(){  // long num1 = 12345678901234567890L;//out of range 超过long的范围  BigInteger num1 = new BigInteger("12345678901234567890");  BigInteger num2 = new BigInteger("92345678901234567890");    // System.out.println("和：" + (num1 + num2));//错误的  System.out.println("和：" + num1.add(num2));  System.out.println("减：" + num1.subtract(num2));  System.out.println("乘：" + num1.multiply(num2));  System.out.println("除：" + num2.divide(num1));//两个整数相除只保留整数部分  System.out.println("幂次方：" + num2.pow(5));  } |

一般的Float类和Double类可以用来做科学计算或工程计算，但是在商业计算中，要求数字精度比较高，所以用到java.math.BigDecimal类。BigDecimal类支持不可变的、任意精度的有符号十进制定点数。

* 构造器
  + BigDecimal(double val)
  + BigDecimal(String val)
* 常用方法
  + BigDecimal add(BigDecimal augend) :返回一个 BigDecimal，其值为 (this + augend)，其标度为 max(this.scale(), augend.scale())。
  + BigDecimal subtract(BigDecimal subtrahend) ：返回一个 BigDecimal，其值为 (this - subtrahend)，其标度为 max(this.scale(), subtrahend.scale())。
  + BigDecimal multiply(BigDecimal multiplicand)：返回一个 BigDecimal，其值为 (this × multiplicand)，其标度为 (this.scale() + multiplicand.scale())。
  + BigDecimal pow(int n) ：返回其值为 (thisn) 的 BigDecimal，准确计算该幂，使其具有无限精度。
  + BigDecimal divide(BigDecimal divisor)： 返回一个 BigDecimal，其值为 (this / divisor)，其首选标度为 (this.scale() - divisor.scale())；如果无法表示准确的商值（因为它有无穷的十进制扩展），则抛出 ArithmeticException。
  + BigDecimal divide(BigDecimal divisor, int roundingMode) ：返回一个 BigDecimal，其值为 (this / divisor)，其标度为 this.scale()。
  + BigDecimal divide(BigDecimal divisor, int scale, int roundingMode) ：返回一个 BigDecimal，其值为 (this / divisor)，其标度为指定标度。

|  |
| --- |
| @Test  public void test3(){  BigDecimal num1 = new BigDecimal("-12.1234567890123456567899554544444332");  BigDecimal num2 = new BigDecimal("89.6734567890123456567899554544444333");  System.out.println("和：" + num1.add(num2));  System.out.println("减：" + num1.subtract(num2));  System.out.println("乘：" + num1.multiply(num2));  System.out.println("除：" + num2.divide(new BigDecimal("2")));//可以整除（除尽）就对，不能整除就报异常  System.out.println("除：" + num2.divide(num1,BigDecimal.ROUND\_HALF\_UP));  System.out.println("除：" + num2.divide(num1,BigDecimal.ROUND\_DOWN));//往零的方向舍去  System.out.println("除：" + num2.divide(num1,BigDecimal.ROUND\_FLOOR));//往小的方向舍去  System.out.println("除：" + num2.divide(num1,BigDecimal.ROUND\_CEILING));//往大的方向舍去  } |

## 10.7 比较器：自然排序与定制排序

### 10.7.1 自然排序：java.lang.Comparable

Comparable接口强行对实现它的每个类的对象进行整体排序。这种排序被称为类的**自然排序**，类的 compareTo(T t) 方法被称为它的自然比较方法。**当前对象this与指定对象t比较“大小”，如果当前对象this大于指定对象t，则返回正整数，如果当前对象this小于指定对象t，则返回负整数，如果当前对象this等于指定对象t，则返回零**。

实现Comparable接口的对象列表（和数组）可以通过 Collections.sort（和 Arrays.sort）进行自动排序。实现此接口的对象可以用作**有序映射**中的键或**有序集合**中的元素，无需指定比较器。

Comparable的典型实现：

* String：按照字符串中字符的Unicode值进行比较
* Character：按照字符的Unicode值来进行比较
* 数值类型对应的包装类以及BigInteger、BigDecimal：按照它们对应的数值大小进行比较
* Date、Time等：后面的日期时间比前面的日期时间大

### 10.7.2 定制排序：java.util.Compartor

强行对某个对象 collection 进行整体排序 的比较函数。可以将 Comparator 传递给 sort 方法（如 Collections.sort 或 Arrays.sort），从而允许在排序顺序上实现精确控制。还可以使用 Comparator 来控制某些数据结构（如有序 set或有序映射）的顺序，或者为那些没有自然顺序的对象 collection 提供排序。

当元素的类型没有实现java.lang.Comparable接口而又不方便修改代码，或者实现了java.lang.Comparable接口的排序规则不适合当前的操作，那么可以考虑使用 Comparator 的对象来排序。

### 10.7.3 示例

|  |
| --- |
| **package** com.api.compare;  **import** java.text.Collator;  **import** java.util.Arrays;  **import** java.util.Comparator;  **import** java.util.Locale;  **public** **class** TestCompare {  @SuppressWarnings("unchecked")  **public** **static** **void** main(String[] args) {  Goods[] all = **new** Goods[4];  all[0] = **new** Goods("《红楼梦》",100);  all[1] = **new** Goods("《西游记》",80);  all[2] = **new** Goods("《三国演义》",140);  all[3] = **new** Goods("《水浒传》",120);    Arrays.*sort*(all);    System.***out***.println(Arrays.*toString*(all));    Arrays.*sort*(all , **new** Comparator() {  @Override  **public** **int** compare(Object o1, Object o2) {  Goods g1 = (Goods) o1;  Goods g2 = (Goods) o2;    **return** Collator.*getInstance*(Locale.***CHINA***).compare(g1.getName(),g2.getName());  }  });    System.***out***.println(Arrays.*toString*(all));  }  }  **class** Goods **implements** Comparable{  **private** String name;  **private** **double** price;  **public** Goods(String name, **double** price) {  **super**();  **this**.name = name;  **this**.price = price;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **double** getPrice() {  **return** price;  }  **public** **void** setPrice(**double** price) {  **this**.price = price;  }  @Override  **public** String toString() {  **return** "商品名：" + name + ", 价格：" + price;  }  @Override  **public** **int** compareTo(Object o) {  Goods other = (Goods) o;  **if**(**this**.price > other.price){  **return** 1;  }**else** **if**(**this**.price < other.price){  **return** -1;  }  **return** 0;  }    } |

## 10.8 JDK1.8新增日期时间类型

Java1.0中包含了一个Date类，但是它的大多数方法已经在Java 1.1引入Calendar类之后被弃用了。而Calendar并不比Date好多少。它们面临的问题是：

* 可变性：象日期和时间这样的类对象应该是不可变的。Calendar类中可以使用三种方法更改日历字段：set()、add() 和 roll()。
* 偏移性：Date中的年份是从1900开始的，而月份都是从0开始的。
* 格式化：格式化只对Date有用，Calendar则不行。
* 此外，它们也不是线程安全的，不能处理闰秒等。

|  |
| --- |
| Date的API：  尽管 Date 类打算反映协调世界时 (UTC)，但无法做到如此准确，这取决于 Java 虚拟机的主机环境。当前几乎所有操作系统都假定 1 天 = 24 × 60 × 60 = 86400 秒。但对于 UTC，大约每一两年出现一次额外的一秒，称为“闰秒”。闰秒始终作为当天的最后一秒增加，并且始终在 12 月 31 日或 6 月 30 日增加。例如，1995 年的最后一分钟是 61 秒，因为增加了闰秒。大多数计算机时钟不是特别的准确，因此不能反映闰秒的差别。 |
| 在类 Date 所有可以接受或返回年、月、日期、小时、分钟和秒值的方法中，将使用下面的表示形式：   * 年份 y 由整数 y - 1900 表示。 * 月份由从 0 至 11 的整数表示；0 是一月、1 是二月等等；因此 11 是十二月。 * 日期（一月中的某天）按通常方式由整数 1 至 31 表示。 * 小时由从 0 至 23 的整数表示。因此，从午夜到 1 a.m. 的时间是 0 点，从中午到 1 p.m. 的时间是 12 点。 * 分钟按通常方式由 0 至 59 的整数表示。 * 秒由 0 至 61 的整数表示；值 60 和 61 只对闰秒发生，尽管那样，也只用在实际正确跟踪闰秒的 Java 实现中。于按当前引入闰秒的方式，两个闰秒在同一分钟内发生是极不可能的，但此规范遵循 ISO C 的日期和时间约定。   在所有情形中，针对这些目的赋予方法的参数不需要在指定的范围内；例如，可以把日期指定为 1 月 32 日，并把它解释为 2 月 1 日的相同含义。 |
| Date date = new Date(2017-1900,8-1,28);  System.out.println(date);//Mon Aug 28 00:00:00 CST 2017 |

可以说，对日期和时间的操作一直是Java程序员最痛苦的地方之一。第三次引入的API是成功的，并且java 8中引入的java.time API 已经纠正了过去的缺陷，将来很长一段时间内它都会为我们服务。

Java 8 吸收了 Joda-Time 的精华，以一个新的开始为 Java 创建优秀的 API。

* java.time – 包含值对象的基础包
* java.time.chrono – 提供对不同的日历系统的访问。
* java.time.format – 格式化和解析时间和日期
* java.time.temporal – 包括底层框架和扩展特性
* java.time.zone – 包含时区支持的类

Java 8 吸收了 Joda-Time 的精华，以一个新的开始为 Java 创建优秀的 API。新的 java.time 中包含了所有关于时钟（Clock），本地日期（LocalDate）、本地时间（LocalTime）、本地日期时间（LocalDateTime）、时区（ZonedDateTime）和持续时间（Duration）的类。历史悠久的 Date 类新增了 toInstant() 方法，用于把 Date 转换成新的表示形式。这些新增的本地化时间日期 API 大大简化了了日期时间和本地化的管理。

说明：大多数开发者只会用到基础包和format包，也可能会用到temporal包。因此，尽管有68个新的公开类型，大多数开发者，大概将只会用到其中的很少一部分。

|  |  |
| --- | --- |
| java.time | 注：ISO-8601日历系统是国际标准化组织制定的现代公民的日期和时间的表示法，也就是公历。 |
| java.time.chrono |  |

### 10.8.1 java.time

#### 1、本地日期（LocalDate）、本地时间（LocalTime）、本地日期时间（LocalDateTime）

LocalDate代表IOS格式（yyyy-MM-dd）的日期,可以存储“生日、纪念日”等日期。

LocalTime表示一个时间，而不是日期

LocalDateTime是用来表示日期和时间的，这是一个最常用的类之一。

|  |  |
| --- | --- |
|  | **描述** |
| now() / now(ZoneId zone) | 静态方法，根据当前时间创建对象/指定时区的对象 |
| of() | 静态方法，根据指定日期/时间创建对象 |
| getDayOfMonth()/getDayOfYear() | 获得月份天数(1-31) /获得年份天数(1-366) |
| getDayOfWeek() | 获得星期几(返回一个 DayOfWeek 枚举值) |
| getMonth() | 获得月份, 返回一个 Month 枚举值 |
| getMonthValue() / getYear() | 获得月份(1-12) /获得年份 |
| getHours()/getMinute()/getSecond() | 获得当前对象对应的小时、分钟、秒 |
| withDayOfMonth()/withDayOfYear()/withMonth()/withYear() | 将月份天数、年份天数、月份、年份修改为指定的值并返回新的对象 |
| with(TemporalAdjuster  t) | 将当前日期时间设置为校对器指定的日期时间 |
| plusDays(), plusWeeks(), plusMonths(), plusYears(),plusHours() | 向当前对象添加几天、几周、几个月、几年、几小时 |
| minusMonths() / minusWeeks()/minusDays()/minusYears()/minusHours() | 从当前对象减去几月、几周、几天、几年、几小时 |
| plus(TemporalAmount t)/minus(TemporalAmount t) | 添加或减少一个 Duration 或 Period |
| isBefore()/isAfter() | 比较两个 LocalDate |
| isLeapYear() | 判断是否是闰年（在LocalDate类中声明） |
| format(DateTimeFormatter  t) | 格式化本地日期、时间，返回一个字符串 |
| parse(Charsequence text) | 将指定格式的字符串解析为日期、时间 |

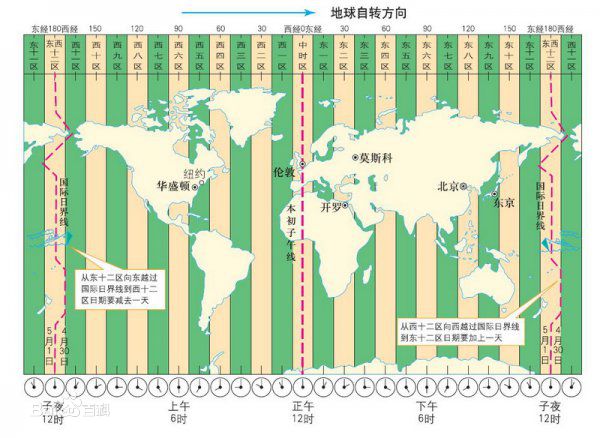
|  |
| --- |
| //now()  @Test  public void testLocalDateTime(){  LocalDate date = LocalDate.now();  LocalTime time = LocalTime.now();  LocalDateTime datetime = LocalDateTime.now();  } |
| //of()或parse  @Test  public void testLocalDate() {  // LocalDate date = LocalDate.now();  // LocalDate date = LocalDate.of(2017, 3, 20);  LocalDate date = LocalDate.parse("2017-03-12");  } |
| public static void main(String[] args) {  LocalDateTime t = LocalDateTime.now();  System.out.println("这一天是这一年的第几天："+t.getDayOfYear());  System.out.println("年："+t.getYear());  System.out.println("月："+t.getMonth());  System.out.println("月份值："+t.getMonthValue());  System.out.println("日："+t.getDayOfMonth());  System.out.println("星期："+t.getDayOfWeek());  System.out.println("时："+t.getHour());  System.out.println("分："+t.getMinute());  System.out.println("秒："+t.getSecond());  System.out.println(t.getMonthValue());  } |
| @Test  public void testLocalDate2() {  LocalDate date = LocalDate.now();  //withXxx()方法，不改变原来的date对象，返回一个新的对象，不可变性  // LocalDate date2 = date.withDayOfMonth(1);//获取这个月的第一天  LocalDate date2 = date.with(TemporalAdjusters.firstDayOfMonth());// 获取这个月的第一天  System.out.println(date2);  // 获取这个月的最后一天  LocalDate date3 = date.with(TemporalAdjusters.lastDayOfMonth());  System.out.println(date3);  //45天后的日期  LocalDate date4 = date.plusDays(45);  System.out.println(date4);  //20天前的日期  LocalDate date5 = date.minusDays(20);  System.out.println(date5);    boolean before = date.isBefore(date5);  System.out.println(date+"是否比"+date5+"早" + before);    System.out.println(date+"是否是闰年："+date.isLeapYear());    } |
| MonthDay month = MonthDay.of(8, 14);  MonthDay today = MonthDay.from(date);  System.out.println("今天是否是生日：" + month.equals(today)); |

#### 2、瞬时：Instant

Instant：时间线上的一个瞬时点。 这可能被用来记录应用程序中的事件时间戳。

在处理时间和日期的时候，我们通常会想到年,月,日,时,分,秒。然而，这只是时间的一个模型，是面向人类的。第二种通用模型是面向机器的，或者说是连续的。在此模型中，时间线中的一个点表示为一个很大的数，这有利于计算机处理。在UNIX中，这个数从1970年开始，以秒为的单位；同样的，在Java中，也是从1970年开始，但以毫秒为单位。

java.time包通过值类型Instant提供机器视图。Instant表示时间线上的一点，而不需要任何上下文信息，例如，时区。概念上讲，它只是简单的表示自1970年1月1日0时0分0秒（UTC）开始的秒数。因为java.time包是基于纳秒计算的，所以Instant的精度可以达到纳秒级。





时间戳：指格林威治时间1970年01月01日00时00分00秒（北京时间1970年01月01日08时00分00秒）起至现在的总秒数。

|  |
| --- |
| public static void main(String[] args) {  Instant t = Instant.now();  System.out.println(t);    //偏移8个小时  OffsetDateTime atOffset = t.atOffset(ZoneOffset.ofHours(8));  System.out.println(atOffset);    long milli = t.toEpochMilli();  System.out.println(milli);    Instant in2 = Instant.ofEpochSecond(10000000);  System.out.println(in2);  } |

#### 3、带时区的日期、时间的处理

作为一个开发者，如果不用去处理时区和它带来的复杂性，那是幸运的。java.time包下的LocalDate、LocalTime、LocalDateTime和Instant基本能满足需求。当你不可避免时区时，ZonedDateTime等类可以满足我们的需求。

ZonedDateTime：一个在ISO-8601日历系统时区的日期时间，如 2007-12-03T10:15:30+01:00 Europe/Paris。

* 其中每个时区都对应着ID，地区ID都为“{区域}/{城市}”的格式，例如：Asia/Shanghai等
* now()：使用系统时间获取当前的ZonedDateTime
* now(ZoneId)：返回指定时区的ZonedDateTime

ZoneId：该类中包含了所有的时区信息，一个时区的ID，如 Europe/Paris

* getAvailableZoneIds()：静态方法，可以获取所有时区信息
* of(String id)：静态方法，用指定的时区信息获取ZoneId对象

Clock：使用时区提供对当前即时、日期和时间的访问的时钟。

|  |
| --- |
| public static void main(String[] args) {  Set<String> availableZoneIds = ZoneId.getAvailableZoneIds();  for (String string : availableZoneIds) {  System.out.println(string);  }    ZonedDateTime t = ZonedDateTime.now();  System.out.println(t);    ZonedDateTime t1 = ZonedDateTime.now(ZoneId.of("America/New\_York"));  System.out.println(t1);  // Clock clock = Clock.systemDefaultZone();  Clock c = Clock.system(ZoneId.of("America/New\_York"));  System.out.println(c.getZone());  System.out.println(c.instant());  } |

#### 4、持续时间：Duration

Duration:用于计算两个“时间”间隔

|  |
| --- |
| public static void main(String[] args) {  LocalDateTime t1 = LocalDateTime.now();  LocalDateTime t2 = LocalDateTime.of(2017, 8, 29, 0, 0, 0, 0);  Duration between = Duration.between(t1, t2);  System.out.println(between);    System.out.println("相差的总天数："+between.toDays());  System.out.println("相差的总小时数："+between.toHours());  System.out.println("相差的总分钟数："+between.toMinutes());  System.out.println("相差的总秒数："+between.getSeconds());  System.out.println("相差的总毫秒数："+between.toMillis());  System.out.println("相差的总纳秒数："+between.toNanos());  System.out.println("不够一秒的纳秒数："+between.getNano());  } |

#### 5、日期间隔：Period

Period:用于计算两个“日期”间隔

|  |
| --- |
| public static void main(String[] args) {  LocalDate t1 = LocalDate.now();  LocalDate t2 = LocalDate.of(2018, 12, 31);  Period between = Period.between(t1, t2);  System.out.println(between);    System.out.println("相差的年数："+between.getYears());//1年  System.out.println("相差的月数："+between.getMonths());//又7个月  System.out.println("相差的天数："+between.getDays());//零25天  System.out.println("相差的总数："+between.toTotalMonths());//总共19个月  } |

### 10.8.2 java.time.temporal .TemporalAdjuster : 时间校正器

TemporalAdjuster : 时间校正器。有时我们可能需要获取例如：将日期调整到“下一个工作日”等操作。

TemporalAdjusters : 该类通过静态方法(firstDayOfXxx()/lastDayOfXxx()/nextXxx())提供了大量的常用 TemporalAdjuster 的实现。

|  |
| --- |
| public static void main(String[] args) {  LocalDate now = LocalDate.now();  System.out.println("下一个周日:"+now.with(TemporalAdjusters.next(DayOfWeek.SUNDAY)));  System.out.println("下周五" + now.with(TemporalAdjusters.next(DayOfWeek.FRIDAY)));  System.out.println("本月最后一天:"+now.with(TemporalAdjusters.lastDayOfMonth()));//2017-08-31  LocalDate week = LocalDate.now().with(new TemporalAdjuster(){  @Override  public Temporal adjustInto(Temporal temporal) {  LocalDate date = (LocalDate) temporal;  if(date.getDayOfWeek().equals(DayOfWeek.MONDAY)){  return date.plusDays(7);  }else if(date.getDayOfWeek().equals(DayOfWeek.TUESDAY)){  return date.plusDays(6);  }else if(date.getDayOfWeek().equals(DayOfWeek.WEDNESDAY)){  return date.plusDays(5);  }else if(date.getDayOfWeek().equals(DayOfWeek.THURSDAY)){  return date.plusDays(4);  }else if(date.getDayOfWeek().equals(DayOfWeek.FRIDAY)){  return date.plusDays(3);  }else if(date.getDayOfWeek().equals(DayOfWeek.SATURDAY)){  return date.plusDays(2);  }else{  return date.plusDays(1);  }  }  });  System.out.println("下一个工作日：" + week);  } |

### 10.8.3 java.time.format.DateTimeFormatter 类

该类提供了三种格式化方法：

预定义的标准格式。如：ISO\_DATE\_TIME;ISO\_DATE

本地化相关的格式。如：ofLocalizedDate(FormatStyle.MEDIUM)

自定义的格式。如：ofPattern(“yyyy-MM-dd hh:mm:ss”)

|  |
| --- |
| public static void main(String[] args) {  System.out.println(DateTimeFormatter.ISO\_DATE.format(LocalDate.now()));  System.out.println(DateTimeFormatter.ISO\_DATE\_TIME.format(LocalDateTime.now()));    //FULL和SHORT适用于LocalDate和LocalTime  System.out.println(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL).format(LocalDate.now()));  System.out.println(DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT).format(LocalTime.now()));    //LONG和MEDIUM适用于LocalDateTime  System.out.println(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM).format(LocalDateTime.now()));    DateTimeFormatter op = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");  System.out.println(op.format(LocalDateTime.now()));  } |

### 10.8.4 与传统日期处理的转换

|  |  |  |
| --- | --- | --- |
| **类** | **To 遗留类** | **From 遗留类** |
| **java.time.Instant与java.util.Date** | Date.from(instant) | date.toInstant() |
| **java.time.Instant与java.sql.Timestamp** | Timestamp.from(instant) | timestamp.toInstant() |
| **java.time.ZonedDateTime与java.util.GregorianCalendar** | GregorianCalendar.from(zonedDateTime) | cal.toZonedDateTime() |
| **java.time.LocalDate与java.sql.Time** | Date.valueOf(localDate) | date.toLocalDate() |
| **java.time.LocalTime与java.sql.Time** | Date.valueOf(localDate) | date.toLocalTime() |
| **java.time.LocalDateTime与java.sql.Timestamp** | Timestamp.valueOf(localDateTime) | timestamp.toLocalDateTime() |
| **java.time.ZoneId与java.util.TimeZone** | Timezone.getTimeZone(id) | timeZone.toZoneId() |
| **java.time.format.DateTimeFormatter与java.text.DateFormat** | formatter.toFormat() | 无 |

## 10.9 练习

1. 模拟一个trim方法，去除字符串两端的空格。
2. 将一个字符串进行反转。将字符串中指定部分进行反转。比如将“abcdefgho”反转为”abfedcgho”
3. 获取一个字符串在另一个字符串中出现的次数。

比如：获取"ab"在 “abababkkcadkabkebfkabkskab”中出现的次数

4、获取两个字符串中最大相同子串。比如：

str1 = "abcwerthelloyuiodef“;str2 = "cvhellobnm"

提示：将短的那个串进行长度依次递减的子串与较长的串比较。

|  |
| --- |
| public class TestString5 {  public static void main(String[] args) {  String str =" he llo ";  System.out.println(myTrim(str));  System.out.println(myTrim2(str));  System.out.println(myTrim3(str));  }    public static String myTrim3(String str){  //利用正则表达式  //^表示开头 \s表示 空白符 \*表示0次或多次 |表示或者 $表示结尾  return str.replaceAll("(^\\s\*)|(\\s\*$)", "");  }    public static String myTrim2(String str){  while(str.startsWith(" ")){  str = str.replaceFirst(" ", "");  }  while(str.endsWith(" ")){  str = str.substring(0, str.length()-1);  }  return str;  }    public static String myTrim(String str){  char[] array = str.toCharArray();  int start =0;  for(int i=0;i<array.length;i++){  if(array[i] == ' '){  start++;  }else{  break;  }  }  int end = array.length-1;  for(int i=end;i>=0;i--){  if(array[i] == ' '){  end--;  }else{  break;  }  }    String result = str.substring(start,end+1);    return result;  }  } |
| public class TestString6 {  public static void main(String[] args) {  String str ="abcdefgho";  System.out.println(str);  System.out.println(reverse(str,2,5));  }    //从第start个字符，到第end个字符  public static String reverse(String str,int start,int end){  char[] array = str.toCharArray();  for(int i = start,j=end;i< j;i++,j--){  char temp =array[i];  array[i]=array[j];  array[j]=temp;  }  String s = new String(array);  return s;  }  } |
| public class TestString7 {  public static void main(String[] args) {  String str1="ab";  String str2="abababkkcadkabkebfkabkskab";  System.out.println(count(str1,str2));  }    public static int count(String str1,String str2){  int count =0;  do{  int index = str2.indexOf(str1);  if(index !=-1){  count++;  str2 = str2.substring(index + str1.length());  }else{  break;  }    }while(true);  return count;  }  } |
| public class TestString8 {  public static void main(String[] args) {  String str=findMaxSubString("abcwerthelloyuiodef","cvhellobnm");  System.out.println(str);  }    //提示：将短的那个串进行长度依次递减的子串与较长的串比较。  public static String findMaxSubString(String str1,String str2){  String result="";    String mixStr = str1.length()<str2.length()?str1:str2;  String maxStr = str1.length()>str2.length()?str1:str2;    //外循环控制从左到右的下标，内循环控制从右到左的下标  for(int i=0;i<mixStr.length();i++){  for(int j=mixStr.length();j>=i;j--){  String str=mixStr.substring(i, j);  if(maxStr.contains(str)){  //找出最大相同子串  if(result.length()<str.length()){  result = str;  }  }  }  }  return result;  }  } |