# 第06天【常用类】

## 主要内容

1. 包装类
2. String类
3. StringBuffer和StringBuilder类
4. Date类和DateFormat类
5. Calendar类
6. JDK8的日期类
7. Math类和Random类
8. 枚举

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 引入包装类 | 理解 |
| 包装类的使用 | 掌握 |
| 自动装箱和自动拆箱 | 掌握 |
| 阅读包装类源码 | 理解 |
| String的常用方法 | 掌握 |
| 阅读String类源码 | 理解 |
| 理解String的==和equals | 掌握 |
| StringBuilder的使用 | 掌握 |
| 理解StringBuilder的原理 | 理解 |
| Date类 | 掌握 |
| DateFormat类 | 掌握 |
| Calendar类 | 了解 |
| Math类 | 了解 |
| JDK8的日期类 | 了解 |
| Random类 | 了解 |
| 枚举 | 了解 |

## 一、包装类

### 1.1引入包装类

#### 1.1.1什么是包装类

Java是面向对象的语言，但并不是“纯面向对象”的，因为我们经常用到的基本数据类型就不是对象。但我们在实际应用中经常需要将基本数据转化成对象，以便于操作。

为了解决这个不足，Java在设计类时为每个基本数据类型设计了一个对应的类进行代表，这样八个和基本数据类型对应的类统称为包装类(Wrapper Class)。8种基本数据类型，分别对应一个包装类。包装类均位于java.lang包

|  |  |
| --- | --- |
| **基本数据类型** | **包装类** |
| **byte** | **Byte** |
| **boolean** | **Boolean** |
| **short** | **Short** |
| **char** | **Character** |
| **int** | **Integer** |
| **long** | **Long** |
| **float** | **Float** |
| **double** | **Double** |

在这八个类名中，除了Integer和Character类以外，其它六个类的类名和基本数据类型一致，只是类名的第一个字母大写而已

在这八个类中，除了Character和Boolean以外，其他的都是“数字型“，“数字型”都是java.lang.Number的子类。

#### 1.1.2为什么需要包装类（优点）

1. 某些方法的参数必须是对象，为了让基本数据类型的数据能作为参数，提供包装类。
2. 包装类还可以提供更多的功能
3. 其他特别重要的功能：比如可以实现字符串和基本数据类型之间的转换

#### 【示例1】认识包装类

|  |
| --- |
| **public class** TestWrapper1 {  **int num**;  Integer **in** ;  **public static void** main(String[] args) {  //1.某些方法参数是对象，为了让基本数据类型能作为参数  List list = **new** ArrayList();  list.add(**new** Integer(56)); //int 56  list.add(**new** Integer(100));  list.add(**new** Double(67.5));  list.add(99);  System.**out**.println(list);  //2.包装类还可以提供更多的功能  System.**out**.println(Integer.**SIZE**);  System.**out**.println(Integer.**MIN\_VALUE**);  System.**out**.println(Integer.**MAX\_VALUE**);  System.**out**.println(Integer.toBinaryString(123));   System.**out**.println(Integer.toOctalString(123));  System.**out**.println(Integer.toHexString(123));   //3.其他特别重要的功能：可以实现字符串和基本数据类型之间的转换  String str = **"123"**; //----123  **int** num2 = Integer.parseInt(str);  System.**out**.println(num2);  String str2 = **"123.45"**;  **double** d = Double.parseDouble(str2);  System.**out**.println(d);  } } |

|  |
| --- |
| **注意**   * 注意1：包装类的对象需要占用栈内存和堆内存，而基本数据类型的（局部）变量只占用栈内存；基本数据类型的变量占用空间少，更简单，更灵活，更高效。 * 注意2：作为成员变量，初始值不同。int 0；Integer null。 |

### 1.2使用包装类

#### 1.2.1自动装箱和自动拆箱

自动装箱和拆箱就是将基本数据类型和包装类之间进行自动的互相转换。JDK1.5后，Java引入了自动装箱(autoboxing)/拆箱(unboxing)。

自动装箱：基本类型的数据处于需要对象的环境中时，会自动转为“对象”。

自动拆箱：每当需要一个值时，对象会自动转成基本数据类型，没必要再去显式调用intValue()、doubleValue()等转型方法。

如 Integer i = 5;int j = i; 这样的过程就是自动拆箱。

我们可以用一句话总结自动装箱/拆箱：

自动装箱过程是通过调用包装类valueOf()方法实现的，而自动拆箱过程是通过调用包装类 xxxValue()方法实现的（xxx代表对应的基本数据类型，如intValue()、doubleValue()等）。

#### 【示例2】包装类的使用

|  |
| --- |
| **public class** TestWrapper2 {  **public static void** main(String[] args) {  //1.自动装箱和自动拆箱  Integer in = 5;  Integer in2 = **new** Integer(5);//valueOf()  **int** i = in2;  **int** i2 = in2.intValue();  //2.== equals  Integer in3 = **new** Integer(56); Integer in4 = **new** Integer(56);  System.**out**.println(in3==in4); //false  System.**out**.println(in3.equals(in4));//true  Integer in5 = 25; Integer in6 = 25;  System.**out**.println(in5 ==in6);//true false ???  System.**out**.println(in5.equals(in6));//true  Integer in7 = 256; Integer in8 = 256;  System.**out**.println(in7 ==in8);//true false ???  System.**out**.println(in7.equals(in8));//true  } } |

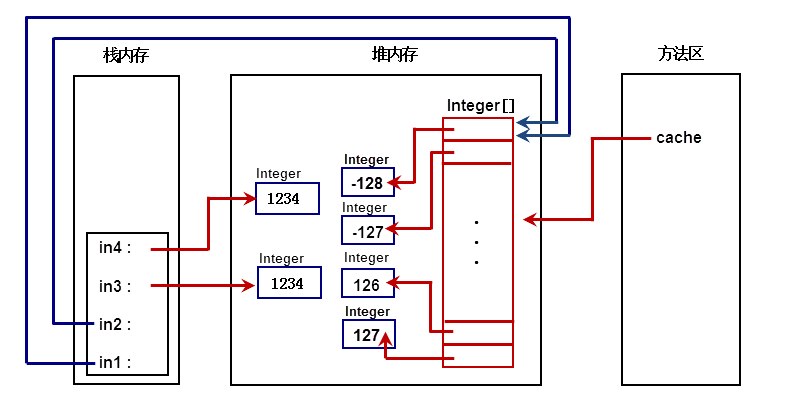
#### 1.2.2理解Integer源码

* Integer的父类是Number类；底层就是封装了一个int类型的value常量，可以通过构造方法、intValue()等赋值取值。

|  |
| --- |
| **public final class** Integer **extends** Number {  **private final int value**;  **public** Integer(**int** value) {  **this**.**value** = value;  }  @Override  **public int** intValue() {  **return value**;  } } |

* Integer类提供了一个静态内部类IntegerCache，对于定义一个静态数组cache，长度为256，赋值为-128—127。对于自动装箱时如果是-128—127范围内的数据，直接获取数组的指定值；对于这个范围之外的数据，通过new Integer()重新创建对象。这么做的目的是提高效率。

|  |
| --- |
| **public class** Integer **extends** Number {   **private static class** IntegerCache {  **static final int *low*** = -128;  **static final int *high***;  **static final** java.lang.Integer ***cache***[];  **static** {  **int** h = 127;  ***high*** = h;  ***cache*** = **new** java.lang.Integer[(***high*** - ***low***) + 1];  **int** j = ***low***;  **for**(**int** k = 0; k < ***cache***.**length**; k++)  ***cache***[k] = **new** java.lang.Integer(j++);  }  **private** IntegerCache() {}  }  **public static** java.lang.Integer valueOf(**int** i) {  **if** (i >= IntegerCache.***low*** && i <= IntegerCache.***high***)  **return** IntegerCache.***cache***[i + (-IntegerCache.***low***)];  **return new** java.lang.Integer(i);  } } |

****

|  |
| --- |
| **注意**   * JDK1.5以后，增加了自动装箱与拆箱功能，如：   Integer i = 100; int j = new Integer(100);   * 自动装箱调用的是valueOf()方法，而不是new Integer()方法。 * 自动拆箱调用的xxxValue()方法。 * 包装类在自动装箱时为了提高效率，对于-128~127之间的值会进行缓存处理。超过范围后，对象之间不能再使用==进行数值的比较，而是使用equals方法。 |

### 本节作业

1. 为什么需要包装类
2. 理解自动装箱和自动拆箱
3. IntegerCache的作用

## 二、字符串相关类

String类、StringBuilder类、StringBuffer类是三个字符串相关类。String类是的对象代表不可变的字符序列，StringBuilder类和StringBuffer类代表可变字符序列。关于这三个类详细的用法，在笔试面试以及实际开发中经常用到，我们必须掌握好它们。

### 2.1 String类的使用

#### 【示例3】String类的使用

|  |
| --- |
| **public class** TestString1 {  **public static void** main(String[] args) {  //1.创建一个String对象  String str = **"北京尚学堂BJsxtBJsxt"**; //字符的个数  //str = null;  //str = "";  //String str2 = new String("北京尚学堂bjsxt");  //2.操作该String对象  //2.1最简单的方法  System.**out**.println(str.length());//10  System.**out**.println(str.isEmpty());//length==0  System.**out**.println(str.startsWith(**"bj"**));  System.**out**.println(str.endsWith(**"sxt"**));  System.**out**.println(str.toUpperCase());  System.**out**.println(str.toLowerCase());  //str = str.toLowerCase();  System.**out**.println(str);  //2.2根据索引确定子串  System.**out**.println(str.charAt(3));  //从beginIndex,到endIndex-1结束，共计输出字符个  //endIndex-beginIndex  System.**out**.println(str.substring(5,7));   System.**out**.println(str.substring(7));  //2.3根据子串确定索引  System.**out**.println(str.indexOf(**"学"**));  System.**out**.println(str.indexOf(**"尚学堂"**));  //从第一个字符开始查找，返回找到的第一个匹配的子串的索引  System.**out**.println(str.indexOf(**"sxt"**));  //如果不存在，就返回-1  System.**out**.println(str.indexOf(**"SXT"**));   System.**out**.println(str.lastIndexOf(**"sxt"**));  System.**out**.println(str.indexOf(**"sxt"**,8));  //2.4其他  //str.compareTo()  **byte** [] bytes = str.getBytes();//字符串----byte[]  System.**out**.println(bytes.**length**); // 5\*3+5\*1=20  System.**out**.println(Arrays.toString(bytes));  //String str2 = new String(bytes); //byte[] ---String  String str2 = **new** String(bytes,9,6);  System.**out**.println(str2);  System.**out**.println(str);   **boolean** flag = str.contains(**"sxt1"**);  System.**out**.println(flag); // str = str.concat("京南校区"); // str = str.concat("亦庄科创十四街"); // str = str.concat("赛蒂科技园");  str = str.concat(**"京南校区"**).concat(**"亦庄科创十四街"**)  .concat(**"赛蒂科技园"**);  System.**out**.println(str);  str = str.replace(**"BJsxt"**,**"北京尚学堂"**);  System.**out**.println(str);  String str3 = **" bj sxt "**;  System.**out**.println(str3.length());  System.**out**.println(str3.trim());  } } |

#### 【示例4】String类的使用2

|  |
| --- |
| **public class** TestString2 {  **public static void** main(String[] args) {  String str1 = **"bjsxt"**;  String str2 = **"bjsxt"**;  System.**out**.println(str1 == str2); //!!!true !!!!!!  System.**out**.println(str1.equals(str2));//true   String str3 = **new** String(**"bjsxt"**);  String str4 = **new** String(**"bjsxt"**);  System.**out**.println(str3 == str4); //false  System.**out**.println(str3.equals(str4));//true   String str5 = **""**;  String str6 = **null**;  System.**out**.println(str5.isEmpty());//true  //System.out.println(str6.isEmpty());//NullPointerException   String str7 = **"123"**;  str7 = str7.concat(**"321"**);//最后追加  str7 = str7.concat(**"abc"**);  str7 = str7.concat(**"xyz"**);  System.**out**.println(str7);  } } |

|  |  |
| --- | --- |
| **方法摘要** | |
| char | **[charAt](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "charAt(int))**(int index)            返回指定索引处的 char 值。 |
| int | **[compareTo](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "compareTo(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) anotherString)            按字典顺序比较两个字符串。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[concat](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "concat(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) str)            将指定字符串连接到此字符串的结尾。 |
| boolean | **[contains](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "contains(java.lang.CharSequence))**([CharSequence](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/CharSequence.html" \o "java.lang 中的接口) s)            当且仅当此字符串包含指定的 char 值序列时，返回 true。 |
| boolean | **[endsWith](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "endsWith(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) suffix)            测试此字符串是否以指定的后缀结束。 |
| boolean | **[equals](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "equals(java.lang.Object))**([Object](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html" \o "java.lang 中的类) anObject)            将此字符串与指定的对象比较。 |
| boolean | **[equalsIgnoreCase](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "equalsIgnoreCase(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) anotherString)            将此 String 与另一个 String 比较，不考虑大小写。 |
| int | **[hashCode](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "hashCode())**()            返回此字符串的哈希码。 |
| int | **[indexOf](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "indexOf(int))**(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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "indexOf(int, int))**(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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "indexOf(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) 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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "indexOf(java.lang.String, int))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) 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。 |
| boolean | **[isEmpty](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "isEmpty())**()            当且仅当 [length()](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "length()) 为 0 时返回 true。 |
| int | **[lastIndexOf](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "lastIndexOf(int))**(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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "lastIndexOf(int, int))**(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，从指定索引处开始进行反向搜索。 |
| int | **[lastIndexOf](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "lastIndexOf(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) 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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "lastIndexOf(java.lang.String, int))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) 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。 |
| int | **[length](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "length())**()            返回此字符串的长度。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[replace](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "replace(char, char))**(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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[replace](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "replace(java.lang.CharSequence, java.lang.CharSequence))**([CharSequence](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/CharSequence.html" \o "java.lang 中的接口) target, [CharSequence](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/CharSequence.html" \o "java.lang 中的接口) 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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[replaceAll](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "replaceAll(java.lang.String, java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) regex, [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) 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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Pattern.html" \l "sum)的子字符串。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类)[] | **[split](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "split(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) regex)            根据给定[正则表达式](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Pattern.html" \l "sum)的匹配拆分此字符串。 |
| boolean | **[startsWith](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "startsWith(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) prefix)            测试此字符串是否以指定的前缀开始。 |
| boolean | **[startsWith](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "startsWith(java.lang.String, int))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) 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。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[substring](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "substring(int))**(int beginIndex)            返回一个新的字符串，它是此字符串的一个子字符串。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[substring](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "substring(int, int))**(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。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[toLowerCase](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[toString](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "toString())**()            返回此对象本身（它已经是一个字符串！）。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[toUpperCase](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "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](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[trim](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "trim())**()            返回字符串的副本，忽略前导空白和尾部空白。 |

### 2.2理解String类源码

* String类是一个final类，意味着该类不能再有子类。
* String类底层是一个字符数组value。各种方法的操作其实都是对该数组的操作。

|  |
| --- |
| **public final class** String  **implements** java.io.Serializable, Comparable, CharSequence {  **private final char value**[];  **public** String() {  **this**.**value** = **""**.**value**;  }  **public** String(java.lang.String original) {  **this**.**value** = original.**value**;  *//this.hash = original.hash;* }  **public boolean** isEmpty() {  **return value**.**length** == 0;  }  **public int** length() {  **return value**.**length**;  }  **public char** charAt(**int** index) {  **if** ((index < 0) || (index >= **value**.**length**)) {  **throw new** StringIndexOutOfBoundsException(index);  }  **return value**[index];  }  **public** java.lang.String toString() {  **return this**;  } } |

* String类的equals()方法其实就是比较底层的字符数组的各个元素是否相同，只要发现一个字符不相同，就返回false。如果所有字符都相同，返回true。但是如果两个变量指向了同一个字符数组，直接返回true。

|  |
| --- |
| **public final class** String  **implements** java.io.Serializable, Comparable, CharSequence {  **public boolean** equals(Object anObject) {  **if** (**this** == anObject) {  **return true**;  }  **if** (anObject **instanceof** java.lang.String) {  java.lang.String anotherString =  (java.lang.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**;  } } |

* String类的concat()是创建一个新的字符数组，存放原来字符数组和新加入的字符数组内容，然后以该新数组创建一个新的字符串。

|  |
| --- |
| **public final class** String  **implements** java.io.Serializable, Comparable<java.lang.String>, CharSequence {  **public** java.lang.String concat(java.lang.String str) {  **int** otherLen = str.length();  **if** (otherLen == 0) {  **return this**;  }  **int** len = **value**.**length**;  **char** buf[] = Arrays.*copyOf*(**value**, len + otherLen);  str.getChars(buf, len);  **return new** java.lang.String(buf, **true**);  }  } |

* JDK9时String类底层由char数组变为byte数组，节省空间，同时通过一个coder成员变量作为编码格式的标识，使用LATIN1还是UTF-16，这个是在String生成时自动的，如果字符串中都是能用LATIN1就能表示的就是0，否则就是UTF-16。

### 2.3 使用StringBuilder类

StringBuffer和StringBuilder非常类似，均代表可变的字符序列。这两个类都是抽象类AbstractStringBuilder的子类，方法几乎一模一样。

两个类的区别主要是：

* StringBuffer JDK1.0提供的类，线程安全，做线程同步检查，效率较低。
* StringBuilder JDK1.5提供的类，线程不安全，不做线程同步检查，因此效率较高。 建议采用该类。

#### 【示例5】StringBuilder类的使用

|  |
| --- |
| **public class** TestStringBuilder {  **public static void** main(String[] args) {  //String----StringBuilder  //StringBuilder builder = new StringBuilder("123");  StringBuffer builder = **new** StringBuffer(**"123"**);  builder.append(**"321"**);//在末尾追加子串  System.**out**.println(builder);   builder.append(**"abc"**);  builder.append(**"xyz"**);  System.**out**.println(builder);  builder.insert(3,**"sxt"**);  System.**out**.println(builder);  builder.setCharAt(3,**'S'**);  System.**out**.println(builder);  builder.replace(3,6,**"SXT"**);  System.**out**.println(builder);   builder.deleteCharAt(3);  System.**out**.println(builder);  builder.delete(6,builder.length());  System.**out**.println(builder);   builder.reverse();  System.**out**.println(builder.toString());  //StringBuilder ----->String  String str =builder.toString();  System.**out**.println(str);  } } |

|  |
| --- |
| **注意**   * 实际开发中StringBuilder的使用场合：字符串的拼接（SQL语句）   StringBuilder builder = new StringBuilder("123");  builder.append("321");  builder.append("abc");  String str =builder.toString(); |

|  |  |
| --- | --- |
| **方法摘要** | |
| [StringBuilder](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \o "java.lang 中的类) | **[append](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "append(boolean))**(boolean b)            将 boolean 参数的字符串表示形式追加到序列。 |
| int | **[capacity](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "capacity())**()            返回当前容量。 |
| char | **[charAt](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "charAt(int))**(int index)            返回此序列中指定索引处的 char 值。 |
| [StringBuilder](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \o "java.lang 中的类) | **[delete](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "delete(int, int))**(int start, int end)            移除此序列的子字符串中的字符。 |
| [StringBuilder](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \o "java.lang 中的类) | **[deleteCharAt](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "deleteCharAt(int))**(int index)            移除此序列指定位置上的 char。 |
| void | **[ensureCapacity](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "ensureCapacity(int))**(int minimumCapacity)            确保容量至少等于指定的最小值。 |
| int | **[indexOf](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "indexOf(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) str)            返回第一次出现的指定子字符串在该字符串中的索引。 |
| int | **[indexOf](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "indexOf(java.lang.String, int))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) str, int fromIndex)            从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。 |
| [StringBuilder](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \o "java.lang 中的类) | **[insert](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "insert(int, boolean))**(int offset, boolean b)            将 boolean 参数的字符串表示形式插入此序列中。 |
| int | **[lastIndexOf](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "lastIndexOf(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) str)            返回最右边出现的指定子字符串在此字符串中的索引。 |
| int | **[lastIndexOf](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "lastIndexOf(java.lang.String, int))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) str, int fromIndex)            返回最后一次出现的指定子字符串在此字符串中的索引。 |
| int | **[length](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "length())**()            返回长度（字符数）。 |
| [StringBuilder](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \o "java.lang 中的类) | **[replace](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "replace(int, int, java.lang.String))**(int start, int end, [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) str)            使用给定 String 中的字符替换此序列的子字符串中的字符。 |
| [StringBuilder](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \o "java.lang 中的类) | **[reverse](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "reverse())**()            将此字符序列用其反转形式取代。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[substring](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "substring(int))**(int start)            返回一个新的 String，它包含此字符序列当前所包含字符的子序列。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[substring](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "substring(int, int))**(int start, int end)            返回一个新的 String，它包含此序列当前所包含字符的子序列。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[toString](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/StringBuilder.html" \l "toString())**()            返回此序列中数据的字符串表示形式。 |

### 2.4 理解StringBuilder源码

* StringBuilder类底层和String类一样，也是一个字符数组value，但不是final的。变量count表示的是底层字符数组的元素的真实个数，不是底层字符数组的长度。
* 默认数组的长度是16。也可以通过构造方法直接指定初始长度。length()方法返回的是字符数组元素的真实个数，capacity()返回的是底层数组的长度；

|  |
| --- |
| **public final class** StringBuilder {//省略父类AbstractStringBuilder  **char**[] **value**;  **int count**;  **public** StringBuilder() {  **super**(16);  //**value** = **new char**[capacity];  }  **public** StringBuilder(**int** capacity) {  **super**(capacity);  }  **public int** length() {  **return count**;  }  **public int** capacity() {  **return value**.**length**;  }  **public** String toString() {  **return new** String(**value**, 0, **count**);  } } |

* 每次添加字符串时要扩容，扩容的默认策略时候增加到原来长度的2倍再加2。

|  |
| --- |
| **public** AbstractStringBuilder append(String str) {  //如果添加的字符串是null，特殊处理  **if** (str == **null**)  **return** appendNull();  //获取要新增的字符串的长度  **int** len = str.length();  //要扩容，要扩容多少呢？  ensureCapacityInternal(**count** + len);  //扩容后将新的字符串追加到最后  str.getChars(0, len, **value**, **count**);  //字符串的长度增加了  **count** += len;  //返回最新的StringBuilder  **return this**; }  **private void** ensureCapacityInternal(**int** minimumCapacity) {  //如果长度不足，就扩容  **if** (minimumCapacity - **value**.**length** > 0) {  **value** = Arrays.copyOf(**value**,  newCapacity(minimumCapacity));  } }  **private int** newCapacity(**int** minCapacity) {  //默认策略：数组原来长度的2倍+2  **int** newCapacity = (**value**.**length** << 1) + 2;  **if** (newCapacity - minCapacity < 0) {  newCapacity = minCapacity;  }  **return** (newCapacity <= 0 ||**MAX\_ARRAY\_SIZE**-newCapacity < 0)  ? hugeCapacity(minCapacity)  : newCapacity; } |

|  |
| --- |
| **要点**   * String：不可变字符序列。 * StringBuffer：可变字符序列，并且线程安全，但是效率低。 * StringBuilder：可变字符序列，线程不安全，但是效率高（一般用它）。 |

### 本节作业

1. 练习String类的常用方法
2. 理解String类底层的源码
3. String、StringBuilder、StringBuffer的区别和练习
4. 理解StringBuilder的底层原理，尤其是数组扩容的机制

## 三、日期类

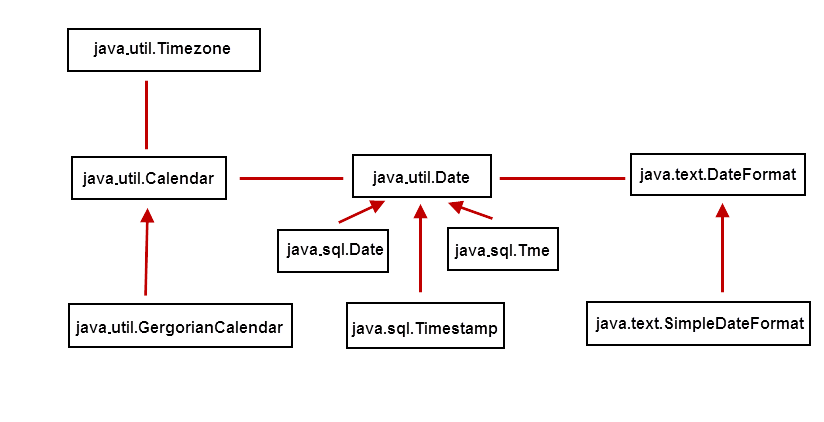
“时间如流水，一去不复返”，时间是一个一维的东东。所以，我们需要一把刻度尺来表达和度量时间。在计算机世界，我们把1970 年 1 月 1 日 00:00:00定为基准时间，每个度量单位是毫秒(1秒的千分之一)，如图所示。

|  |
| --- |
| 8-13 |
| **图8-13 计算机的时间概念** |

我们用long类型的变量来表示时间，从基准时间往前几亿年，往后几亿年都能表示。如果想获得现在时刻的“时刻数值”，可以使用：

**long** now = System.*currentTimeMillis*();

这个“时刻数值”是所有时间类的核心值，年月日都是根据这个“数值”计算出来的。我们工作学习涉及的时间相关类有如下这些：



### 3.1Date类

在标准Java类库中包含一个Date类。它的对象表示一个特定的瞬间，精确到毫秒。在网上商城下单时，在对报销单进行审核时，都需要获取当前的时间，通过Date类就可以完成。

#### 【示例5】Date类的使用

|  |
| --- |
| **public class** TestDate {  **public static void** main(String[] args) {  //获取当前的时间  Date now = **new** Date();  //输出当前的时间  System.**out**.println(now.toString());  System.**out**.println(now.toLocaleString());  System.**out**.println(now.getYear()); // 1900+119=2019  System.**out**.println(now.getMonth());// 0---11  System.**out**.println(now.getDate());//日  //星期几 星期日是0，一周是从星期日开始  System.**out**.println(now.getDay());  System.**out**.println(now.getHours());  System.**out**.println(now.getMinutes());  System.**out**.println(now.getSeconds());  //其他的方法  **long** l = now.getTime();  // 1970年1月1日 00:00:00 GMT以来的毫秒数。  System.**out**.println(l);  System.**out**.println(System.currentTimeMillis());  System.**out**.println(System.nanoTime());   Date date1 = **new** Date(-(**long**)(1000\*60\*60\*24)\*31);  System.**out**.println(date1.toLocaleString());  //util.Date yyyyMMdd hh:mm:ss  // 一共有三个子类 Date Time Timestamp  //sql.Date yyyyMMdd  //sql.Time hh:mm:ss  //sql/Timestamp 时间戳 yyyyMMdd hh:mm:ss  //java.sql.Date sDate = new java.sql.Date(l);  java.sql.Date sDate =  **new** java.sql.Date(System.currentTimeMillis());  System.**out**.println(sDate);  java.sql.Date sDate2 =  java.sql.Date.valueOf(**"2019-8-06"**);//对格式有要求  System.**out**.println(sDate2);  } } |

查看API文档大家可以看到其实Date类中的很多方法都已经过时了。JDK1.1之前的Date包含了：日期操作、字符串转化成时间对象等操作。JDK1.1之后，日期操作一般使用Calendar类，而字符串的转化使用DateFormat类。

#### 【示例6】理解Date类的源码

|  |
| --- |
| **public class** Date  **implements** java.io.Serializable, Cloneable, Comparable<Date> {  **private transient long fastTime**;  **public** Date() {  **this**(System.*currentTimeMillis*());  }  **public** Date(**long** date) {  **fastTime** = date;  }  **public long** getTime() {  **return** getTimeImpl();  }  **private final long** getTimeImpl() {  **if** (cdate != **null** && !cdate.isNormalized()) {  normalize();  }  **return fastTime**;  }  } |

Date类中提供一个成员变量fastTime，表示相应时间对应的毫秒数。

### 3.2.DateFormat类

DateFormat是一个抽象类，一般使用它的的子类SimpleDateFormat类来实现。主要作用就是把时间对象转化成指定格式的字符串。反之，把指定格式的字符串转化成时间对象。

需求1：网页注册，需要输入日期，在网页上输入的所有内容，xiaoming,23, 1999-12-23,1999/12/23，但是传送到后台接收到都是字符串 "xiaoming", "23", "1999-12-23","1999/12/23"，需要把字符串----->日期

需求2：注册成功后，在网页上显示日期，可以不同格式来显示"1999-12-23", "1999/12/23"，"1999年12月23日"，需要把日期----->字符串。

#### 【示例7】DateFormat类的使用

|  |
| --- |
| **public class** TestDateFormat {  **public static void** main(String[] args) {  // int age = Integer.parseInt("23abc");  // System.out.println(age);  //有缺陷：对日期格式有限制，不能表示时分秒 // String str = "1999-12-23 "; // //String ----Date // java.sql.Date date = java.sql.Date.valueOf(str); // //Date ---String // System.out.println(date.toString());   DateFormat sdf =  **new** SimpleDateFormat(**"yyyy-MM-dd hh:mm:ss"**);  String str = **"1999-12-23 09:23:45"**;  //String ---Date  Date date =**null**;  **try** {  date = sdf.parse(str);  } **catch** (ParseException e) {  e.printStackTrace();  } // System.out.println(date.toString()); // System.out.println(date.toLocaleString());  //Date--String  DateFormat sdf2 = **new** SimpleDateFormat(**"yyyy年MM月dd日 hh时mm分ss秒"**);  String str2 = sdf2.format(date);  System.**out**.println(str2);  } } |

**关于日期和时间模式**

|  |  |  |  |
| --- | --- | --- | --- |
| **字母** | **日期或时间元素** | **表示** | **示例** |
| G | Era 标志符 | [Text](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "text) | AD |
| y | 年 | [Year](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "year) | 1996; 96 |
| M | 年中的月份 | [Month](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "month) | July; Jul; 07 |
| w | 年中的周数 | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 27 |
| W | 月份中的周数 | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 2 |
| D | 年中的天数 | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 189 |
| d | 月份中的天数 | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 10 |
| F | 月份中的星期 | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 2 |
| E | 星期中的天数 | [Text](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "text) | Tuesday; Tue |
| a | Am/pm 标记 | [Text](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "text) | PM |
| H | 一天中的小时数（0-23） | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 0 |
| k | 一天中的小时数（1-24） | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 24 |
| K | am/pm 中的小时数（0-11） | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 0 |
| h | am/pm 中的小时数（1-12） | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 12 |
| m | 小时中的分钟数 | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 30 |
| s | 分钟中的秒数 | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 55 |
| S | 毫秒数 | [Number](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "number) | 978 |
| z | 时区 | [General time zone](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "timezone) | Pacific Standard Time; PST; GMT-08:00 |
| Z | 时区 | [RFC 822 time zone](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \l "rfc822timezone) | -0800 |

### 3.3 Calendar类

Calendar 类是一个抽象类，为我们提供了关于日期计算的相关功能，比如：年、月、日、时、分、秒的展示和计算。GregorianCalendar 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。

#### 【示例8】Calendar类的使用

|  |
| --- |
| **public class** TestCalendar {  **public static void** main(String[] args) {  //创建Calendar对象  Calendar cal = **new** GregorianCalendar();  //输出日期中的各个部分  System.**out**.println(cal.get(Calendar.**YEAR**));  System.**out**.println(cal.get(Calendar.**MONTH**));  System.**out**.println(cal.get(Calendar.**DATE**));  System.**out**.println(cal.get(Calendar.**HOUR**));  System.**out**.println(cal.get(Calendar.**MINUTE**));  System.**out**.println(cal.get(Calendar.**SECOND**));  System.**out**.println(cal.get(Calendar.**MONTH**));  System.**out**.println(cal.get(Calendar.**DAY\_OF\_WEEK**));  //修改日期中各个部分  cal.set(Calendar.**YEAR**,2000);  cal.set(Calendar.**MONTH**,1);  cal.set(Calendar.**DATE**,1); //2019-08-01  **int** max =cal.getActualMaximum(Calendar.**DATE**);  **for**(**int** i=1;i<=max;i++){  System.**out**.println(cal.get(Calendar.**DAY\_OF\_WEEK**));  cal.add(Calendar.**DATE**,1);  }  System.**out**.println(cal.get(Calendar.**DAY\_OF\_WEEK**));  System.**out**.println(cal.getActualMinimum(Calendar.**DATE**));  System.**out**.println(cal.getActualMaximum(Calendar.**DATE**));  //输出Calendar对象  System.**out**.println(cal);  //Date--Calendar之间的转换  //Date ---->Calendar  Date now = **new** Date();  cal.setTime(now);  //Calendar---->Date  Date now2 = cal.getTime();  } } |

### 3.4 JDK8的新的日期类

针对JDK8以前的时间日期类设计的不足，比如不支持时区，线程不安全等，JDK8引入了java.time包来作为新的日期时间处理类。

|  |  |
| --- | --- |
| **属性** | **含义** |
| Instant | 代表的是时间，相当于Date |
| LocalDate | 代表日期，比如2020-01-14 |
| LocalTime | 代表时刻，比如12:59:59 |
| LocalDateTime | 代表具体时间 2020-01-12 12:22:26 |
| ZonedDateTime | 代表一个包含时区的完整的日期时间，偏移量是以UTC/ 格林威治时间为基准的 |
| Period | 代表时间段 |
| ZoneOffset | 代表时区偏移量，比如：+8:00 |
| Clock | 代表时钟，比如获取目前美国纽约的时间 |
| DateTimeFormatter | 日期和字符串格式转换 |

#### 【示例10】使用Calendar类实现万年历的功能

|  |
| --- |
| **public class** TestNewDate {  **public static void** main(String[] args) {  //Instant  Instant instant = Instant.*now*();  System.***out***.println(instant);  System.***out***.println(Instant.***MAX***);  System.***out***.println(Instant.***MIN***);  System.***out***.println(Instant.***EPOCH***);  //LocalDate  LocalDate localDate = LocalDate.*now*();  System.***out***.println(localDate);  //DateTimeFormatter  DateTimeFormatter dftr = DateTimeFormatter.*ofPattern*(**"yyyy年MM月dd日"**);  //LocalDate--->String  String str = localDate.format(dftr);  System.***out***.println(str);  //String--->LocalDate  LocalDate localDate1 = LocalDate.*parse*(str,dftr);  System.***out***.println(localDate1);  } } |

### 本节作业

1. Date、DateFormat、Calendar的作用。
2. 使用DateFormat实现字符串和时间之间的转换

## 四、其他常用类

### 4.1Math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。如果需要更加强大的数学运算能力，计算高等数学中的相关内容，可以使用apache commons下面的Math类库。

#### 【示例11】Math类

|  |
| --- |
| **public class** TestMath {  **public static void** main(String[] args) {  System.***out***.println(Math.***PI***);  System.***out***.println(Math.*round*(3.14));  System.***out***.println(Math.*ceil*(3.14));  System.***out***.println(Math.*floor*(3.14));   System.***out***.println(Math.*sqrt*(64));*//开平方* System.***out***.println(Math.*pow*(2,5));*//幂* System.***out***.println(Math.*abs*(-5));  System.***out***.println(Math.*max*(30,40));  System.***out***.println(Math.*min*(30,40));  System.***out***.println(Math.*random*());*//[0.0,1.0)* } } |

### 4.2 Random类

Math类中虽然为我们提供了产生随机数的方法Math.random()，但是通常我们需要的随机数范围并不是[0, 1)之间的double类型的数据，这就需要对其进行一些复杂的运算。如果使用Math.random()计算过于复杂的话，我们可以使用例外一种方式得到随机数，即Random类，这个类是专门用来生成随机数的，并且Math.random()底层调用的就是Random的nextDouble()方法。

#### 【示例12】使用Random生成[10,40)之间的随机正数

|  |
| --- |
| **public class** TestRandom {  **public static void** main(String[] args) {  //Random rand = new Random(10000);  Random rand = **new** Random();  System.**out**.println(rand.nextInt());  System.**out**.println(rand.nextInt(10));  System.**out**.println(rand.nextDouble());  System.**out**.println(rand.nextFloat());  **for** (**int** i = 0; i < 10; i++) {  System.**out**.print(rand.nextInt(10)+**" "**);  }  } } |

|  |  |
| --- | --- |
| **构造方法摘要** | |
| **[Random](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Random.html" \l "Random())**()            创建一个新的随机数生成器。 |  |
| **[Random](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Random.html" \l "Random(long))**(long seed)            使用单个 long 种子创建一个新的随机数生成器。 |  |

发现只要种子数和nextInt()中的参数一致的话，每次生成的随机数都是一样的（所以这是伪随机数）。

### 4.3枚举

JDK1.5引入枚举类型。枚举类型的定义包括枚举声明和枚举体。枚举体就是放置一些常量。

定义枚举要使用enum关键字。对于性别、季节、星期几等内容，如果定义为字符串类型，是很难限制其取值的。采用枚举可以轻松解决该问题。

#### 【示例13】定义和使用性别枚举类

|  |
| --- |
| **public enum** Gender {  **男**,**女** }  **public class** Person {  String **name**;  //String sex;//gender  Gender **sex**;  **int age**;  **public** String getName() {  **return name**;  }  **public void** setName(String name) {  **this**.**name** = name;  }  **public** Gender getSex() {  **return sex**;  }  **public void** setSex(Gender sex) {  **this**.**sex** = sex;  }  **public int** getAge() {  **return age**;  }  **public void** setAge(**int** age) {  **this**.**age** = age;  }  **public static void** main(String[] args) {  Person person = **new** Person();  //person.sex = "adfadfad";  person.**sex** = Gender.**男**;  System.**out**.println(person.getSex());  } } |

#### 【示例14】定义和使用季节枚举类

|  |
| --- |
| **public enum** Season {  ***春***,***夏***,***秋***,***冬*** }  **public class** TestSeason {  **public static void** main(String[] args) {  Season season = Season.***夏***;  **switch**(season){  **case *春*** :  System.***out***.println(**"春暖花开 踏青 春游 "**);  **break**;  **case *夏*** :  System.***out***.println(**"夏日炎炎 吃雪糕 游泳"**);  **break**;  **case *秋*** :  System.***out***.println(**"秋高气爽 收庄稼 赏月"**);  **break**;  **case *冬*** :  System.***out***.println(**"冬雪皑皑 泡温泉 打雪仗"**);  **break**;  }  } } |

所有的枚举类型隐性地继承自 java.lang.Enum。枚举实质上还是类！而每个被枚举的成员实质就是一个枚举类型的实例，他们默认都是public static final修饰的。可以直接通过枚举类型名使用它们。

|  |
| --- |
| **注意**   * 强烈建议当你需要定义一组常量时，使用枚举类型； * 另外尽量不要使用枚举的高级特性 |

### 本节作业

1. 练习Math类方法
2. 使用Random类生产指定范围的正数，比如[10,30]。
3. 定义季节、性别枚举并使用它