02 常用类

大纲	知识点
包装类	包装类基本用法
	Number 类
	自动装箱和拆箱
	包装类的缓存问题
字符串相关类	String 类
	StringBuilder 类和 StringBuffer 类
	不可变字符序列使用陷阱
时间处理相关类	Date 类
	DateFormat 和 SimpleDateFormat
	Calendar
其他常用类	Math 类、Random 类
	File 类
	递归打印目录树
	枚举

基本数据类型的包装类

我们前面学习的八种基本数据类型并不是对象，为了将基本类型数据和对象之间实现互相转化，Java 为每一个基本数据类型提供了相应的包装类。

包装类基本知识

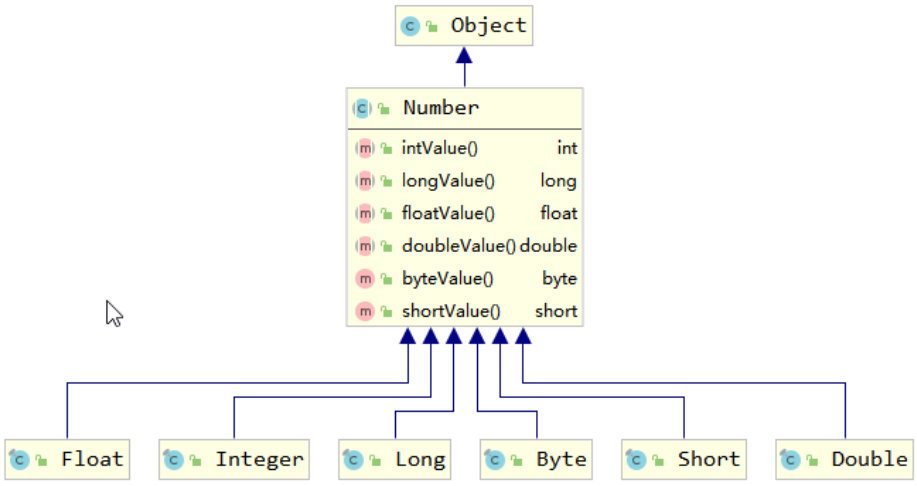
Java 是面向对象的语言，但并不是“纯面向对象”的，因为我们经常用到的基本数据类型就不是对象。但是我们在实际应用中经常需要将基本数据转化成对象，以便于操作。比如：将基本数据类型存储到 Object[ ]数组或集合中的操作等等。

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)。

包装类位于 java.lang 包，八种包装类和基本数据类型的对应关系：

基本数据类型对应的包装类	
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。



Number 类是抽象类，因此它的抽象方法，所有子类都需要提供实现。Number 类提供了抽象方法：intValue()、longValue()、floatValue()、doubleValue()，意味着所有的“数字型”包装类都可以互相转型。

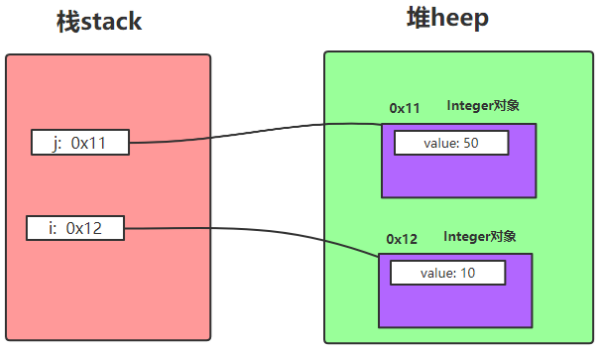
下面我们通过一个简单的示例认识一下包装类。

【示例】初识包装类

```
public class WrapperClassTest {
    public static void main(String[ ] args) {
```

```
        Integer i = new Integer(10);      //从 java9 开始被废弃
        Integer j = Integer.valueOf(50); //官方推荐
    }
}
```

示例内存分析如图所示：



包装类的用途

对于包装类来说，这些类的用途主要包含两种：

1. 作为和基本数据类型对应的类型存在，方便涉及到对象的操作，如 Object[ ]、集合等的操作。
2. 包含每种基本数据类型的相关属性如最大值、最小值等，以及相关的操作方法（这些操作方法的作用是在基本数据类型、包装类对象、字符串之间提供相互之间的转化！）。

【示例】包装类的使用

```
public class Test {
    /** 测试 Integer 的用法，其他包装类与 Integer 类似 */
    void testInteger() {
        // 基本类型转化成 Integer 对象
        Integer int1 = new Integer(10);    //已经被废弃，不推荐使用
        Integer int2 = Integer.valueOf(20); // 官方推荐这种写法
        // Integer 对象转化成 int
        int a = int1.intValue();
        // 字符串转化成 Integer 对象
        Integer int3 = Integer.parseInt("334");
        Integer int4 = new Integer("999");
        // Integer 对象转化成字符串
        String str1 = int3.toString();
       // 一些常见 int 类型相关的常量
        System.out.println("int 能表示的最大整数：" + Integer.MAX_VALUE);
```

```
    }
    public static void main(String[ ] args) {
        Test test   = new Test();
        test.testInteger();
    }
}
```

执行结果如图所示：

```
int能表示的最大整数：2147483647
```

自动装箱和拆箱

自动装箱(autoboxing)和拆箱(unboxing):将基本数据类型和包装类自动转换。

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”。

我们以 Integer 为例：

```
Integer i = 5
```

编译器会自动转成：Integer i = Integer.valueOf(5)，这就是 Java 的自动装箱。

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。

```
Integer i = Integer.valueOf(5);
int j = i;
```

编译器会自动转成：int j = i.intValue();

这样的过程就是自动拆箱。

自动装箱/拆箱的本质是：

自动装箱与拆箱的功能是编译器来帮忙，编译器在编译时依据您所编写的语法，决定是

否进行装箱或拆箱动作。

【示例】自动装箱

```
Integer i = 100;//自动装箱
//相当于编译器自动为您作以下的语法编译：
Integer i = Integer.valueOf(100);//调用的是 valueOf(100)，而不是 new Integer(100)
```

【示例】自动拆箱

```
Integer i = 100;
int j = i;//自动拆箱
//相当于编译器自动为您作以下的语法编译：
int j = i.intValue();
```

自动装箱与拆箱的功能是所谓的“编译器蜜糖(Compiler Sugar)”，虽然使用这个功能很方便，但在程序运行阶段您得了解 Java 的语义。如下所示的程序是可以通过编译的：

【示例】包装类空指针异常问题

```
public class Test1 {
    public static void main(String[ ] args) {
        Integer i = null;
        int j = i;
    }
}
```

执行结果如图所示：

```
Exception in thread "main" java.lang.NullPointerException
        at cn.sxt.gao1.Test1.main(Test1.java:5)
```

运行结果之所以会出现空指针异常，是因为如上代码相当于：

```
public class Test1 {
    public static void main(String[ ] args) {
        /*示例 8-5 的代码在编译时期是合法的，但是在运行时期会有错误
          因为其相当于下面两行代码*/
        Integer i = null;
        int j = i.intValue();
    }
}
```

包装类的缓存问题

整型、char类型所对应的包装类，在自动装箱时，对于-128~127之间的值会进行缓存处理，其目的是提高效率。

缓存原理为：如果数据在-128~127这个区间，那么在类加载时就已经为该区间的每个数值创建了对象，并将这256个对象存放到一个名为cache的数组中。每当自动装箱过程发生时（或者手动调用valueOf()时），就会先判断数据是否在该区间，如果在则直接获取数组中对应的包装类对象的引用，如果不在该区间，则会通过new调用包装类的构造方法来创建对象。

下面我们以Integer类为例，看一看Java为我们提供的源码，加深对缓存技术的理解，如示例所示。

【示例】Integer 类相关源码

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

这段代码中我们需要解释下面几个问题：

1. IntegerCache类为Integer类的一个静态内部类，仅供Integer类使用。
2. 一般情况下 IntegerCache.low为-128，IntegerCache.high为127，

IntegerCache.cache为内部类的一个静态属性，如示例所示。

【示例】IntegerCache 类相关源码

```
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() {}
}
```

由上面的源码我们可以看到，静态代码块的目的就是初始化数组cache的，这个过程会在类加载时完成。

下面我们做一下代码测试，如示例所示。

【示例 8-9】包装类的缓存测试

```
public class Test3 {
    public static void main(String[ ] args) {
        Integer in1 = -128;
        Integer in2 = -128;
        System.out.println(in1 == in2);//true 因为 123 在缓存范围内
        System.out.println(in1.equals(in2));//true
        Integer in3 = 1234;
        Integer in4 = 1234;
        System.out.println(in3 == in4);//false 因为 1234 不在缓存范围内
        System.out.println(in3.equals(in4));//true
    }
}
```

总结

- ❑ 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。

自定义一个简单的包装类

仅限于练习，没有实际意义！

```
public class MyInteger {

    private int value;
    private static MyInteger[] cache = new MyInteger[256];

    public static final int LOW = -128;
    public static final int HIGH = 127;


    static {
        //[-128,127]
        for(int i=MyInteger.LOW;i<=HIGH;i++){
            //-128,0;-127,1;-126,2;
            cache[i+128] = new MyInteger(i);
        }
    }

    public static MyInteger valueOf(int i) {
        if(i>=LOW&&i<=HIGH) {
            return cache[i+128];
        }
        return new MyInteger(i);
    }
```

```
    @Override
    public String toString() {
        return this.value+"";
    }

    public int intValue(){
        return value;
    }

    private MyInteger(int i) {
        this.value = i;
    }

    public static void main(String[] args) {
        MyInteger m = MyInteger.valueOf(30);
        System.out.println(m);

    }
}
```

字符串相关类

String 类代表不可变的字符序列

StringBuilder 类和 StringBuffer 类代表可变字符序列。

这三个类的用法，在笔试面试以及实际开发中经常用到，必须掌握好。

String 类源码分析

String 类对象代表不可变的 Unicode 字符序列，因此我们可以将 String 对象称为“不可变对象”。 那什么叫做“不可变对象”呢？指的是对象内部的成员变量的值无法再改变。

我们打开 String 类的源码，如图所示：

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
{
    /** The value is used for character storage. */
    private final char value[];

    /** The offset is the first index of the storage that is used. */
    private final int offset;

    /** The count is the number of characters in the String. */
    private final int count;
}
```

String 类的部分源码

我们发现字符串内容全部存储到 value[ ]数组中，而变量 value 是 final 类型的，也就是常量(即只能被赋值一次)。 这就是“不可变对象”的典型定义方式。

我们发现在前面学习 String 的某些方法，比如：substring()是对字符串的截取操作，但本质是读取原字符串内容生成了新的字符串。测试代码如下：

【示例】String 类的简单使用

```
public class TestString1 {
    public static void main(String[ ] args) {
        String s1 = new String("abcdef");
        String s2 = s1.substring(2, 4);
        // 打印：ab199863
        System.out.println(Integer.toHexString(s1.hashCode()));
        // 打印：c61，显然 s1 和 s2 不是同一个对象
        System.out.println(Integer.toHexString(s2.hashCode()));
    }
}
```

在遇到字符串常量之间的拼接时，编译器会做出优化，即在编译期间就会完成字符串的拼接。因此，在使用==进行 String 对象之间的比较时，我们要特别注意，如示例所示。

【示例】字符串常量拼接时的优化

```
public class TestString2 {
    public static void main(String[ ] args) {
        //编译器做了优化,直接在编译的时候将字符串进行拼接
        String str1 = "hello" + " java";//相当于 str1 = "hello java";
        String str2 = "hellojava";
        System.out.println(str1 == str2);//true
        String str3 = "hello";
```

```
        String str4 = " java";
        //编译的时候不知道变量中存储的是什么,所以没办法在编译的时候优化
        String str5 = str3 + str4;
        System.out.println(str2 == str5);//false
    }
}
```

StringBuffer 和 StringBuilder 可变字符序列

StringBuffer 和 StringBuilder 都是可变的字符序列。

- StringBuffer **线程安全，做线程同步检查， 效率较低。**
- StringBuilder **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。

- **常用方法列表：**
  - 重载的 public StringBuilder append(...)方法
  - 可以为该 StringBuilder 对象添加字符序列，**仍然返回自身对象。**
  - 方法 public StringBuilder delete(int start,int end)
  - 可以删除从 start 开始到 end-1 为止的一段字符序列，**仍然返回自身对象。**
  - 方法 public StringBuilder deleteCharAt(int index)
  - 移除此序列指定位置上的 char，仍然返回自身对象。
  - 重载的 public StringBuilder insert(...)方法
    可以为该 StringBuilder 对象在指定位置插入字符序列，**仍然返回自身对象。**
  - 方法 public StringBuilder reverse()
  - 用于将字符序列逆序，仍然返回自身对象。
  - 方法 public String toString() 返回此序列中数据的字符串表示形式。
  - 和 String 类含义类似的方法：

```
public int indexOf(String str)
public int indexOf(String str,int fromIndex)
public String substring(int start)
public String substring(int start,int end)
public int length()
char charAt(int index)
```

【示例】StringBuffer/StringBuilder 基本用法

```
public class TestStringBufferAndBuilder{
    public static void main(String[ ] args) {
        /**StringBuilder*/
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 7; i++) {
            sb.append((char) ('a' + i));//追加单个字符
        }
        System.out.println(sb.toString());//转换成 String 输出
        sb.append(", I can sing my abc!");//追加字符串
        System.out.println(sb.toString());
        /**StringBuffer，下面的方法同样适用 StringBuilder*/
        StringBuffer sb2 = new StringBuffer("北京尚学堂");
        sb2.insert(0, "爱").insert(0, "我");//插入字符串
        System.out.println(sb2);
        sb2.delete(0, 2);//删除子字符串
        System.out.println(sb2);
        sb2.deleteCharAt(0).deleteCharAt(0);//删除某个字符
        System.out.println(sb2.charAt(0));//获取某个字符
        System.out.println(sb2.reverse());//字符串逆序
    }
}
```

执行结果如图所示：

```
abcdefg
abcdefg, I can sing my abc!
我爱北京尚学堂
北京尚学堂
尚
堂学尚
```

不可变和可变字符序列使用陷阱

- **String 使用的陷阱**

String 一经初始化后，就不会再改变其内容了。对 String 字符串的操作实际上是对其副本（原始拷贝）的操作，原来的字符串一点都没有改变。比如：

String s ="a"；创建了一个字符串

s = s+"b"；实际上原来的"a"字符串对象已经丢弃了，现在又产生了另一个字符串 s+"b"（也就是"ab"）。 如果多次执行这些改变串内容的操作，会导致大量副本字符串对象存留在内存中，降低效率。如果这样的操作放到循环中，会极大影响程序的时间和空间性能，甚至会造成服务器的崩溃。

相反，StringBuilder 和 StringBuffer 类是对原字符串本身操作的，可以对字符串进行修改而不产生副本拷贝或者产生少量的副本。因此可以在循环中使用。

【示例】String 和 StringBuilder 在字符串频繁修改时的效率测试

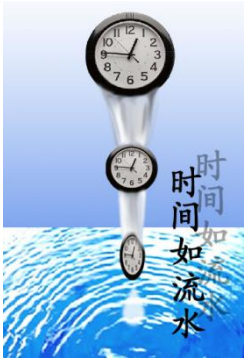
```
public class Test {
    public static void main(String[ ] args) {
        /**使用 String 进行字符串的拼接*/
        String str8 = "";

        long num1 = Runtime.getRuntime().freeMemory();//获取系统剩余内存空间
        long time1 = System.currentTimeMillis();//获取系统的当前时间
        for (int i = 0; i < 5000; i++) {
            str8 = str8 + i;//相当于产生了 5000 个对象
        }
        long num2 = Runtime.getRuntime().freeMemory();
        long time2 = System.currentTimeMillis();
        System.out.println("String 占用内存 ：" + (num1 - num2));
        System.out.println("String 占用时间 ：" + (time2 - time1));
        /**使用 StringBuilder 进行字符串的拼接*/
```

```
        StringBuilder sb1 = new StringBuilder("");
        long num3 = Runtime.getRuntime().freeMemory();
        long time3 = System.currentTimeMillis();
        for (int i = 0; i < 5000; i++) {
            sb1.append(i);
        }
        long num4 = Runtime.getRuntime().freeMemory();
        long time4 = System.currentTimeMillis();
        System.out.println("StringBuilder 占用内存：" + (num3 - num4));
        System.out.println("StringBuilder 占用时间：" + (time4 - time3));
    }
}
```

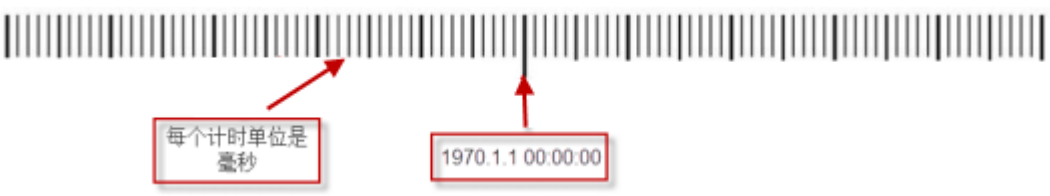
执行结果如图所示：

```
String占用内存： 6110984
String占用时间： 84
StringBuilder占用内存： 0
StringBuilder占用时间： 1
```

时间处理相关类



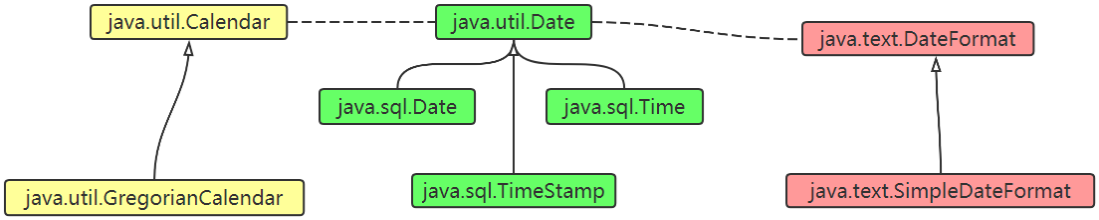
“时间如流水，一去不复返”，时间是一维的。所以，我们需要一把刻度尺来表达和度量时间。在计算机世界，我们把 1970 年 1 月 1 日 00:00:00 定为基准时间，每个度量单位是毫秒(1 秒的千分之一)，如图所示。



计算机的时间概念

我们用 long 类型的变量来表示时间，从基准时间前后几亿年都能表示。
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：



日期时间相关类

Date 时间类(java.util.Date)

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。

- Date() 分配一个 Date 对象，并初始化此对象为系统当前的日期和时间，可以精确到毫秒)。
- Date(long date) 分配 Date 对象并初始化此对象，以表示自从标准基准时间以来的毫秒数。
- boolean equals(Object obj) 比较两个日期的相等性。
- long getTime() 返回毫秒数。
- String 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 。

【示例】Date 类的使用

```
long  nowNum = System.currentTimeMillis();  //当前时刻对应的毫秒数

Date d = new Date();                  //当前时刻的对象
System.out.println(d.getTime());     //返回时间对应的毫秒数

Date d2 = new Date(1000L * 3600 * 24 * 365 * 150);  //距离1970年150年
System.out.println(d2);
```

DateFormat 类和 SimpleDateFormat 类

•DateFormat 类的作用

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。

DateFormat 是一个抽象类，一般使用它的的子类 SimpleDateFormat 类来实现。

【示例】DateFormat 类和 SimpleDateFormat 类的使用

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class TestDateFormat {
    public static void main(String[ ] args) throws ParseException {
        // new 出 SimpleDateFormat 对象
        SimpleDateFormat s1 = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
        SimpleDateFormat s2 = new SimpleDateFormat("yyyy-MM-dd");
        // 将时间对象转换成字符串
        String daytime = s1.format(new Date());
        System.out.println(daytime);
        System.out.println(s2.format(new Date()));
        System.out.println(new SimpleDateFormat("hh:mm:ss").format(new Date()));
        // 将符合指定格式的字符串转成成时间对象.字符串格式需要和指定格式一致。
        String time = "2049-10-1";
        Date date = s2.parse(time);
        System.out.println("date1: " + date);
        time = "2049-10-1 20:15:30";
        date = s1.parse(time);
        System.out.println("date2: " + date);
    }
}
```

代码中的格式化字符的具体含义见表：

格式化字符的含义

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

时间格式字符也可以为我们提供其他的便利。比如：获得当前时间是今年的第几天。

【示例】获取今天时本年度第几天

```
import java.text.SimpleDateFormat;
import java.util.Date;
public class TestDateFormat2 {
    public static void main(String[ ] args) {
        SimpleDateFormat s1 = new SimpleDateFormat("D");
        String daytime = s1.format(new Date());
        System.out.println(daytime);
    }
}
```

执行结果如图所示：

```
137
```

Calendar 日历类

Calendar 类是一个抽象类，为我们提供了关于日期计算的功能，比如：年、月、日、时、分、秒的展示和计算。
GregorianCalendar 是 Calendar 的子类，表示公历。

菜鸟雷区

注意月份的表示，一月是 0，二月是 1，以此类推，12 月是 11。 因为大多数人习惯于使用单词而不是使用数字来表示月份，这样程序也许更易读，父类 Calendar 使用常量来表示月份：JANUARY、FEBRUARY 等等。

【示例】GregorianCalendar 类和 Calendar 类的使用

```
import java.util.*;
public class TestCalendar {
    public static void main(String[ ] args) {
        // 得到相关日期元素
        GregorianCalendar calendar = new GregorianCalendar(2049, 9, 1, 22, 10, 50);
        int year = calendar.get(Calendar.YEAR); // 打印：2049
        int month = calendar.get(Calendar.MONTH); // 打印：9
        int day = calendar.get(Calendar.DAY_OF_MONTH); // 打印：1
        int day2 = calendar.get(Calendar.DATE); // 打印：1
        // 日：Calendar.DATE 和 Calendar.DAY_OF_MONTH 同义
        int date = calendar.get(Calendar.DAY_OF_WEEK); // 打印：1
        // 星期几 这里是：1-7.周日是 1，周一是 2，。。。周六是 7
        System.out.println(year);
        System.out.println(month);
        System.out.println(day);
        System.out.println(day2);
        System.out.println(date);
        // 设置日期
        GregorianCalendar calendar2 = new GregorianCalendar();
        calendar2.set(Calendar.YEAR, 2049);
```

```
            calendar2.set(Calendar.MONTH, Calendar.OCTOBER); // 月份数：0-11
            calendar2.set(Calendar.DATE, 1);
            calendar2.set(Calendar.HOUR_OF_DAY, 10);
            calendar2.set(Calendar.MINUTE, 20);
            calendar2.set(Calendar.SECOND, 23);
            printCalendar(calendar2);
            // 日期计算
            GregorianCalendar calendar3 = new GregorianCalendar(2049, 9, 1, 22, 10,
50);
            calendar3.add(Calendar.MONTH, -7); // 月份减 7
            calendar3.add(Calendar.DATE, 7); // 增加 7 天
            printCalendar(calendar3);
            // 日历对象和时间对象转化
            Date d = calendar3.getTime();
            GregorianCalendar calendar4 = new GregorianCalendar();
            calendar4.setTime(new Date());
        }
        static void printCalendar(Calendar calendar) {
            int year = calendar.get(Calendar.YEAR);
            int month = calendar.get(Calendar.MONTH) + 1;
            int day = calendar.get(Calendar.DAY_OF_MONTH);
            int date = calendar.get(Calendar.DAY_OF_WEEK) - 1; // 星期几
            String week = "" + ((date == 0) ? "日" : date);
            int hour = calendar.get(Calendar.HOUR);
            int minute = calendar.get(Calendar.MINUTE);
            int second = calendar.get(Calendar.SECOND);
            System.out.printf("%d 年%d 月%d 日,星期%s %d:%d:%d\n", year, month, day,
                    week, hour, minute, second);
        }
}
```

其他常用类

Math 类

java.lang.Math 提供了一系列静态方法用于科学计算；常用方法如下：

- abs 绝对值
- acos,asin,atan,cos,sin,tan 三角函数
- sqrt 平方根
- pow(double a, double b) a 的 b 次幂
- max(double a, double b) 取大值
- min(double a, double b) 取小值
- ceil(double a) 大于 a 的最小整数
- floor(double a) 小于 a 的最大整数
- random() 返回 0.0 到 1.0 的随机数
- long round(double a) double 型的数据 a 转换为 long 型（四舍五入）
- toDegrees(double angrad) 弧度->角度
- toRadians(double angdeg) 角度->弧度

【示例】Math 类的常用方法

```
public class TestMath {
    public static void main(String[ ] args) {
        //取整相关操作
        System.out.println(Math.ceil(3.2));
        System.out.println(Math.floor(3.2));
        System.out.println(Math.round(3.2));
        System.out.println(Math.round(3.8));
        //绝对值、开方、a 的 b 次幂等操作
        System.out.println(Math.abs(-45));
```

```
        System.out.println(Math.sqrt(64));
        System.out.println(Math.pow(5, 2));
        System.out.println(Math.pow(2, 5));
        //Math 类中常用的常量
        System.out.println(Math.PI);
        System.out.println(Math.E);
        //随机数
        System.out.println(Math.random());// [0,1)
    }
}
```

执行结果如图所示：

```
4.0
3.0
3
4
45
8.0
25.0
32.0
3.141592653589793
2.718281828459045
0.19363107489981546
```

Random 类

Random 类：专门用来生成随机数。

```
import java.util.Random;
public class TestRandom {
    public static void main(String[ ] args) {
        Random rand = new Random();
        //随机生成[0,1)之间的 double 类型的数据
        System.out.println(rand.nextDouble());
        //随机生成 int 类型允许范围之内的整型数据
        System.out.println(rand.nextInt());
        //随机生成[0,1)之间的 float 类型的数据
        System.out.println(rand.nextFloat());
        //随机生成 false 或者 true
        System.out.println(rand.nextBoolean());
        //随机生成[0,10)之间的 int 类型的数据
        System.out.print(rand.nextInt(10));
```

```
        //随机生成[20,30)之间的 int 类型的数据
        System.out.print(20 + rand.nextInt(10));
    }
}
```

注意

Random 类位于 java.util 包下。

File 类

File 类用来代表文件和目录。

File 类的基本用法

java.io.File 类：代表文件和目录，用于：读取文件、创建文件、删除文件、修改文件。

【示例】使用 File 类创建文件

File 类的常见构造方法：public File(String pathname)

以 pathname 为路径创建 File 对象，如果 pathname 是相对路径，则默认的当前路径在系统属性 user.dir 中存储。

```
import java.io.File;
public class TestFile1 {
    public static void main(String[ ] args) throws Exception {
        System.out.println(System.getProperty("user.dir"));
        File f = new File("a.txt"); //相对路径：默认放到 user.dir 目录下面
        f.createNewFile();//创建文件
        File f2 = new File("d:/b.txt");//绝对路径
        f2.createNewFile();
    }
```

```
}
```

user.dir 就是本项目的目录。上面代码执行后，在本项目和 D 盘下都生成了新的文件。

❑ **通过 File 对象可以访问文件的属性：**

File 类访问属性的方法列表

方法	说明
public boolean exists()	判断 File 是否存在
public boolean isDirectory()	判断 File 是否是目录
public boolean isFile()	判断 File 是否是文件
public long lastModified()	返回 File 最后修改时间
public long length()	返回 File 大小
public String getName()	返回文件名
public String getPath()	返回文件的目录路径

【示例】使用 File 类访问文件或目录属性

```
import java.io.File;
import java.util.Date;
public class TestFile2 {
    public static void main(String[ ] args) throws Exception {
        File f = new File("d:/b.txt");
        System.out.println("File 是否存在："+f.exists());
        System.out.println("File 是否是目录："+f.isDirectory());
        System.out.println("File 是否是文件："+f.isFile());
        System.out.println("File 最后修改时间："+new Date(f.lastModified()));
        System.out.println("File 的大小："+f.length());
        System.out.println("File 的文件名："+f.getName());
        System.out.println("File 的目录路径："+f.getAbsolutePath());
    }
}
```

执行结果如图所示：

```
File是否存在：true
File是否是目录：false
File是否是文件：true
File最后修改时间：Thu May 18 14:25:26 CST
File的大小：0
File的文件名：b.txt
File的目录路径：d:\b.txt
```

❑ **通过 File 对象创建空文件或目录（在该对象所指的文件或目录不存在的情况下）**

表 File 类创建文件或目录的方法列表

方法	说明
createNewFile()	创建新的 File
delete()	删除 File 对应的文件
mkdir()	创建一个目录；中间某个目录缺失，则创建失败
mkdirs()	创建多个目录；中间某个目录缺失，则创建该缺失目录

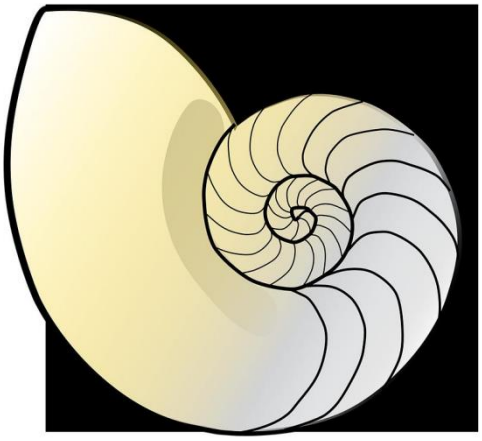
【示例】使用 mkdir 创建目录

```
import java.io.File;
public class TestFile3 {
    public static void main(String[ ] args) throws Exception {
        File f = new File("d:/c.txt");
        f.createNewFile(); // 会在 d 盘下面生成 c.txt 文件
        f.delete(); // 将该文件或目录从硬盘上删除
        File f2 = new File("d:/电影/华语/大陆");
        boolean flag = f2.mkdir(); //目录结构中有一个不存在，则不会创建整个目录树
        System.out.println(flag);//创建失败
    }
}
```

【示例】使用 mkdirs 创建目录

```
import java.io.File;
public class TestFile4 {
    public static void main(String[ ] args) throws Exception {
        File f = new File("d:/c.txt");
        f.createNewFile(); // 会在 d 盘下面生成 c.txt 文件
        f.delete(); // 将该文件或目录从硬盘上删除
        File f2 = new File("d:/电影/华语/大陆");
        boolean flag = f2.mkdirs();//目录结构中有一个不存在也没关系；创建整个目录树
        System.out.println(flag);//创建成功
    }
}
```

递归遍历目录结构和树状展现



本节结合前面给大家讲的递归算法，展示目录结构。大家可以先建立一个目录，下面增加几个子文件夹或者文件，用于测试。

【示例】使用递归算法，以树状结构展示目录树

```
import java.io.File;
public class TestFile6 {
    public static void main(String[ ] args) {
        File f = new File("d:/电影");
        printFile(f, 0);
    }
    /**
     * 打印文件信息
     * @param file 文件名称
     * @param level 层次数(实际就是：第几次递归调用)
     */
    static void printFile(File file, int level) {
        //输出层次数
        for (int i = 0; i < level; i++) {
            System.out.print("-");
        }
        //输出文件名
        System.out.println(file.getName());
        //如果 file 是目录，则获取子文件列表，并对每个子文件进行相同的操作
        if (file.isDirectory()) {
            File[ ] files = file.listFiles();
            for (File temp : files) {
                //递归调用该方法：注意要+1
                printFile(temp, level + 1);
```

```
            }
        }
    }
}
```

执行结果如图所示：

```
电影
-华语
--大陆
---人民的名义.mp4
---尚学堂传奇.mp4
---程序员统治世界.mp4
-好莱坞
--国王的演讲.mp4
--速度与激情8.mp4
```

枚举

JDK1.5 引入了枚举类型。枚举类型的定义包括枚举声明和枚举体。格式如下：

```
enum   枚举名 {
        枚举体（常量列表）
}
```

枚举体就是放置一些常量。我们可以写出我们的第一个枚举类型，如示例所示：

【示例】创建枚举类型

```
enum Season {
    SPRING, SUMMER, AUTUMN, WINTER
```

```
}
```

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。

老鸟建议

- ❑ 当你需要定义一组常量时，可以使用枚举类型。
- ❑ 尽量不要使用枚举的高级特性，事实上高级特性都可以使用普通类来实现，没有必要引入枚举，增加程序的复杂性！

【示例】枚举的使用

```
import java.util.Random;
public class TestEnum {
    public static void main(String[ ] args) {
        // 枚举遍历
        for (Week k : Week.values()) {
            System.out.println(k);
        }
        // switch 语句中使用枚举
        int a = new Random().nextInt(4); // 生成 0，1，2，3 的随机数
        switch (Season.values()[a]) {
        case SPRING:
            System.out.println("春天");
            break;
        case SUMMER:
            System.out.println("夏天");
            break;
        case AUTUMN:
            System.out.println("秋天");
            break;
        case WINTER:
            System.out.println("冬天");
            break;
        }
    }
}
/**季节*/
enum Season {
    SPRING, SUMMER, AUTUMN, WINTER
```

```
}
/**星期*/
enum Week {
    星期一, 星期二, 星期三, 星期四, 星期五, 星期六, 星期日
}
```