第5章

Java API

本章介绍一些常用的 Java API，包括字符串、时间与日期、数学与随机数、正则表达式和容器等接口和类。

本章要点

- 字符串；
- 时间与日期；
- 数学与随机数；
- 系统相关；
- 正则表达式；
- 容器。

5.1 字符串

java.lang 包提供了字符串相关的一些类，如 String、StringBuffer、StringBuilder 等类。字符串是应用非常频繁的对象，本节主要介绍这些字符串类的使用方法。

5.1.1 String 类

String 是一个最终类(final 类型)，它表示一个字符串常量。Java 字符串常量使用双引号包括的一串字符，字符串由 0 个或任意多个字符组成，例如：

```
"Java Programming."
```

Java 编译器自动为每一个字符串常量生成一个 String 类的实例，因此可以用字符串常量直接初始化一个 String 对象，例如：

```
String s = "Java Programming.";
```

字符串常量，一旦创建之后其值就不能再改变。但 Java 提供了对字符串连接符号(+)用于连接其他字符串，例如：

```
String s = "Hello, " + name;
```

假设变量 name 的值是“Liming”，那么字符串变量 s 的值为“Hello, Liming”。

声明字符串除了上述方式，也可以使用 String 类的构造方法创建字符串对象，String 类

中提供了多个构造方法，常见的构造方法见表 5.1。

表 5.1 String 类常见的构造方法

方 法 名	说 明
String()	创建一个空字符串
String(byte[] bytes)	将一个 byte 数组构造为字符串
String(String s)	使用字符串构造为 String 对象
String(char[] chars)	将一个字符数组构造为字符串

String 类提供了很多操作字符串的方法，包括字符串的比较、查找、分隔等操作，表 5.2 列出 String 类的主要方法。

表 5.2 String 类的主要方法

方 法 名	说 明
byte[] getBytes()	返回字符串的 byte 数组形式
char charAt(int index)	返回指定索引处的字符
int compareTo(String s)	按照字母表的顺序比较两个字符串的大小关系，如相等返回 0，否则返回两个字符串对应字符的差值
boolean regionMatches(boolean ignoreCase, int toffset, String other, int offset, int len)	测试两个字符串的区域是否相等，即模式匹配，匹配成功则返回 true，否则返回 false
boolean startsWith(String prefix, int toffset)	判断字符串是否以指定前缀开始
boolean endsWith(String suffix)	判断该字符串是否以 suffix 后缀结束
int lastIndexOf(int ch)	返回 ch 在该字符串中最后一次出现时的索引值
String substring(int beginIndex, int endIndex)	取得该字符串在[beginIndex,endIndex)范围内的子字符串
String concat(String str)	将 str 连接到该字符串后并返回
String replace(char oldChar, char newChar)	该方法有重载，将字符 oldChar 替换为 newChar
String[] split(String regex)	将字符串根据给定的正则表达式进行拆分
String toLowerCase()	返回字符串的小写字母形式
String toUpperCase()	返回字符串的大写字母形式
String trim()	清除字符串两端的空格
char[] toCharArray()	返回字符串的字符数组形式
static String valueOf(type types)	该方法有重载，返回 types 的字符串形式
int length()	返回字符串长度
boolean equals(String s)	判断两个字符串的内容是否相等
int indexOf(int ch)	返回 ch 在该字符串中首次出现时的索引值

编写一个程序，输入一个身份证号，并实现如下功能：①验证身份证号的有效性；②获取其出生年月日；③判断其籍贯是否为北京市朝阳区。例 5.1 为 String 类的实例。

【例 5.1】 Example5_01.java

```
public class Example5_01 {
    public static void main(String[] args) {
        String id = input();
        if(valid(id)){
            LocalDate birth = getBirth(id);
            System.out.println("出生日期: " + birth.toString());
```

```
            //北京朝阳区地区编码 110105
            String chaoyang = "110105";
            System.out.println("他是朝阳群众吗?" + getRegion(id,chaoyang));
        }
        else{
            System.out.println("请输入合法身份证号码!");
            return;
        }
    }
    //输入身份证号码
    public static String input(){
        Scanner scanner = new Scanner(System.in);
        System.out.println("输入身份证号码: ");
        String id = null;
        id = scanner.nextLine();
        return id;
    }
    /**
     * 根据正则表达式验证身份证有效性
     * @param id 身份证号码
     * @return
     */
    public static boolean valid(String id){
        //身份证正则表达式
        String regex = "\\d{17}[\\d|x]|\\d{15}";
        if(id.matches(regex))
            return true;
        return false;
    }
    /**
     * 根据身份证号码得到出生日期
     * @param id 身份证号码
     * @return  LocalDate类型的日期
     */
    public static LocalDate getBirth(String id){
        int year = Integer.parseInt(id.substring(6,10));
        int month = Integer.parseInt(id.substring(10,12));
        int day = Integer.parseInt(id.substring(12,14));
        LocalDate birthdate = LocalDate.of(year,month,day);
        return birthdate;
    }
    /**
     * 判断身份证号码是否以 prefix 打头
     * @param id 身份证号码
     * @param prefix 前缀
     * @return
     */
    public static boolean getRegion(String id,String prefix){
        if(id.startsWith(prefix))
            return true;
        return false;
    }
}
```

运行结果：

```
输入身份证号码:
110105200808081256
出生日期: 2008 - 08 - 08
他是朝阳群众吗?true
```

5.1.2 StringBuffer 类

尽管 String 类提供了丰富的方法,但 String 代表的是字符串常量,所以无法对字符串进行插入、删除等操作。StringBuffer 类可用于操作字符串变量,特别是追加、插入等操作,另外,StringBuffer 也支持多线程,对于需要对字符串执行同步的程序来讲,使用 StringBuffer 是较好的选择。

创建 StringBuffer 对象时可为该对象提供一个字符串缓冲区,默认情况下其容量为 16 个字符,当然也可以指定缓冲区容量的大小。表 5.3 列出了 StringBuffer 类的构造方法。

表 5.3 StringBuffer 类的构造方法

方 法 名	说 明
StringBuffer()	构造一个空的 StringBuffer 对象,其缓冲区容量为 16 个字符
StringBuffer(int capacity)	构造一个缓冲区容量为 capacity 的 StringBuffer 对象
StringBuffer(String str)	构造一个初始内容为 str 的 StringBuffer 对象,容量为 16 加上 str 的长度

StringBuffer 类也提供了很多关于字符串操作的方法,有些方法与 String 类高度雷同,此处不再赘述。表 5.4 列出了 StringBuffer 类的追加、插入、删除、反转等常用方法。

表 5.4 StringBuffer 类的常用方法

方 法 名	说 明
StringBuffer append(type t)	将 t 追加到此字符串的尾部
StringBuffer insert(int offset,type t)	将 t 插入到此字符串索引为 offset 的位置
StringBuffer reverse()	返回此字符串的反转形式
StringBuffer delete(int start,int end)	删除此字符串指定区间[start,end)内的子字符串
StringBuffer deleteCharAt(int index)	删除此字符串指定索引处的字符

下面的案例遍历一个数组,并将其元素拼接为一个字符串形式,本程序分别使用 String 类与 StringBuffer 类两种方式实现,可通过结果对比二者的效率。

【例 5.2】 Example5_02.java

```
public class Example5_02 {
    public static void main(String[] args) {
        int len = 10000;
        int[] arr = new int[len];
        for (int i = 0;i < arr.length;i++){
            arr[i] = 2 * i + 1;
        }
        String s = arrayToStringBuffer(arr);
        String s1 = arrayToString(arr);
    }
    //使用 StringBuffer 方式拼接字符串
    public static String arrayToStringBuffer(int[] arr){
        StringBuffer sb = new StringBuffer();
```

```
        long begin = System.currentTimeMillis();
        sb.append("[");
        for (int x = 0; x < arr.length; x++) {
            //最后一个元素
            if (x == arr.length - 1) {
                sb.append(arr[x] + "]");
            } else {
                //拼接后为 StringBuffer 类型
                sb.append(arr[x]).append(", ");
            }
        }
        long end = System.currentTimeMillis();
        System.out.println("StringBuffer 方式消耗时间: " + (end - begin) + "ms");
        //StringBuffer 类下的 toString()方法,返回字符串 String 类型
        return sb.toString();
    }
    //使用 String 方式拼接字符串
    public static String arrayToString(int[] arr){
        String sb = "";
        long begin = System.currentTimeMillis();
        sb = sb + "[";
        for (int x = 0; x < arr.length; x++) {
            //最后一个元素
            if (x == arr.length - 1) {
                sb = sb + arr[x] + "]";
            } else {
                //拼接后为 StringBuffer 类型
                sb = sb + arr[x] + ", ";
            }
        }
        long end = System.currentTimeMillis();
        System.out.println("String 方式消耗时间: " + (end - begin) + "ms");
        //StringBuffer 类下的 toString()方法,返回字符串 String 类型
        return sb.toString();
    }
}
```

运行结果:

```
StringBuffer 方式消耗时间: 37ms
String 方式消耗时间: 232ms
```

本例分别使用 StringBuffer 类和 String 类将一个长度为 10000 的整型数组遍历后拼接为字符串形式。通过运行结果可以看到,StringBuffer 类的 append()方法追加字符串明显比 String 类使用“+”拼接的方式效率高。

5.1.3 StringBuilder 类

StringBuffer 是一个线程安全类,提供了对字符串操作的同步控制。同时,Java 也提供了另外一个字符串的可变对象,用来高效拼接字符串,它就是 StringBuilder 类。StringBuilder 类是线程不安全的,但支持链式操作,效率比 StringBuffer 更高。StringBuilder 类提供的方法与 StringBuffer 类似,本节不再说明。仍以例 5.2 的功能,增加一个方法使用 StringBuilder 类实现。

```
/**
 * 使用 StringBuilder 类实现字符串拼接
 * @param arr
 * @return
 */
public static String arrayToStringBuilder(int[] arr){
    StringBuilder sb = new StringBuilder();
    long begin = System.currentTimeMillis();
    sb.append("[");
    for (int x = 0; x < arr.length; x++) {
        //最后一个元素
        if (x == arr.length - 1) {
            sb.append(arr[x] + "]");
        } else {
            //拼接后为 StringBuffer 类型
            sb.append(arr[x]).append(", ");
        }
    }
    long end = System.currentTimeMillis();
    System.out.println("StringBuilder 方式消耗时间: " + (end - begin) + "ms");
    //StringBuilder 类下的 toString()方法,返回字符串 String 类型
    return sb.toString();
}
```

运行结果：

```
StringBuffer 方式消耗时间: 60ms
String 方式消耗时间: 213ms
StringBuilder 方式消耗时间: 4ms
```

通过对比，StringBuilder 类的效率最高，String 类的效率最低。说明如果大量对字符串进行拼接、插入和删除等操作时，使用 StringBuilder 是较好的选择。

注意：String 类与 StringBuffer 类、StringBuilder 类的不同之处如下。

- String 类重写了 equals()方法，比较两个字符串内容是否相等。StringBuffer 类和 StringBuilder 类没有重写 equals()方法，仍然比较两个对象的地址是否相等；
- StringBuffer 类和 StringBuilder 类用于字符串追加、插入、删除、反转等操作，且其内容可以改变，且字符串拼接的效率明显比 String 类高；
- StringBuffer 是 StringBuilder 的线程安全版本，现在很少使用。

5.1.4 StringTokenizer 类

java.util.StringTokenizer 类用于分隔字符串，可以指定分隔符，并提供了遍历字符串的方法。StringTokenizer 类的构造方法与其他方法如表 5.5 所示。

表 5.5 StringTokenizer 类的构造方法和其他方法

方 法 名	说 明
StringTokenizer(String str)	构造一个用来解析 str 的 StringTokenizer 对象。Java 默认的分隔符是“空格”“制表符('\t')”“换行符('\n')”“回车符('\r')”

续表

方 法 名	说 明
StringTokenizer(String str,String delim)	构造一个用来解析 str 的 StringTokenizer 对象,并提供一个指定的分隔符
StringTokenizer(String str, String delim, boolean retrunDelims)	构造一个用来解析 str 的 StringTokenizer 对象,并提供一个指定的分隔符,同时指定是否返回分隔符
int countTokens()	返回 nextToken 方法被调用的次数
boolean hasMoreTokens()	返回是否还有分隔符
boolean hasMoreElements()	返回是否还有元素
String nextToken()	返回从当前位置到下一个分隔符的字符串
String nextToken(String delim)	以指定的分隔符返回结果

【例 5.3】 Example5_03.java

```
import java.util.StringTokenizer;
public class Example5_03 {
    public static void main(String[] args) {
        String s = new String("Object - oriented programming (OOP) is a programming" +
"paradigm.");
        StringTokenizer st = new StringTokenizer(s);
        System.out.println("Token Total: " + st.countTokens());
        while(st.hasMoreElements()) {
            System.out.println(st.nextToken());
        }
    }
}
```

运行结果:

```
Token Total: 7
Object - oriented
programming
(OOP)
is
a
programming
paradigm.
```

本例构造了无参 StringTokenizer 对象,使用空格作为分隔符分隔字符串 s,通过使用 hasMoreElements()和 nextToken()方法遍历分隔后的字符串。hasMoreElements()和 hasMoreToken()方法是等价的,本例中二者可以互换。

5.2 时间与日期

时间和日期是应用程序中经常用到的对象,Java 提供了丰富的时间与日期类,包括日期、时间、日历等类。

5.2.1 java.util.Date 类

java.util.Date 类封装了当前的日期和时间,通过 Date 类可以构建当前的系统时间,该

类同时提供设置时间、获取时间、比较时间关系的方法，具体见表 5.6。

表 5.6 java.util.Date 类的构造方法和常用方法

方 法 名	说 明
Date()	使用当前日期和时间来初始化对象
Date(long ms)	参数是从 1970 年 1 月 1 日起的毫秒数，构造 Date 对象
void setTime(long ms)	用自 1970 年 1 月 1 日 00:00:00 GMT 以后 time 毫秒数设置时间和日期
long 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
boolean after(Date date)	若调用此方法的 Date 对象在指定日期之后返回 true，否则返回 false
boolean before(Date date)	若调用此方法的 Date 对象在指定日期之前返回 true，否则返回 false
int compareTo(Date date)	比较调用此方法的 Date 对象和指定日期。两者相等时返回 0。调用对象在指定日期之前则返回负数。调用对象在指定日期之后则返回正数

5.2.2 java.sql.Date 类

java.sql.Date 为 java.util.Date 的子类，该类是一个封装了毫秒值并支持 JDBC 操作的日期类，该类是为了与 SQL DATE 类型保持一致而特设的一个日期类，规范化的 java.sql.Date 只包含年月日信息，时分秒等都置零。该类的构造方法和常用方法见表 5.7。

表 5.7 java.sql.Date 类的构造方法和常用方法

方 法 名	说 明
Date(long ms)	使用给定的毫秒数构造 Date 对象
void setTime(long ms)	用给定的毫秒数设置日期
static Date valueOf(String s)	将 JDBC 日期转义格式的字符串转换为日期类型

5.2.3 Calendar 类

java.util.Calendar 类是一个抽象类，表示日历对象，它提供了丰富的常量，如 YEAR、MONTH、DATE_OF_MONTH 等表示年、月、日等，同时提供了操作日历字段的一些方法，如 get()、set()等方法。表 5.8 列出了 Calendar 类的常量和常用方法。

表 5.8 java.util.Calendar 类的常量和常用方法

常量和常用方法	说 明
Calendar.YEAR	年份
Calendar.MONTH	月份，注意月份范围为 0～11，0 代表 1 月，以此类推
Calendar.DATE	日期
Calendar.DAY_OF_MONTH	日期
Calendar.HOUR	12 小时制的小时
Calendar.HOUR_OF_DAY	24 小时制的小时
Calendar.MINUTE	分钟
Calendar.SECOND	秒
Calendar.DAY_OF_WEEK	星期几
getInstance()	返回一个日历对象实例
final void setTime(Date date)	用给定的日期设置日历
final java.util.Date getTime()	返回一个 java.util.Date 日期类型
void set(int year,int month,int day)	根据年、月、日设置日历
int get(int field)	根据日历常量值获取当前日历信息

【例 5.4】 Example5_04.java

```
import java.time.LocalDate;
import java.util.Calendar;
import java.util.Date;
public class Example5_04 {
    public static void main(String[] args) {
        java.util.Date date = new java.util.Date();
        System.out.println("Date 原始输出: " + date.toString());
        System.out.println("距 1970-1-1 0 时的毫秒数: " + date.getTime());
        //通过 java.util.Date 构造 java.sql.Date 对象
        java.sql.Date dbDate = new java.sql.Date(date.getTime());
        //可将 java.sql.Date 对象转换为 LocalDate 对象
        LocalDate localDate = dbDate.toLocalDate();
        System.out.println("LocalDate:" + localDate);
        //Calendar 类是抽象类
        Calendar calendar = Calendar.getInstance();
        //可通过如下两种方式为日历对象设置时间
        calendar.set(2021,11,06);
        calendar.setTime(date);
        //格式化日期 yyyy 年 M 月 d 日 hh:mm:ss
        String sDate = calendar.get(Calendar.YEAR) + "年" +
        (calendar.get(Calendar.MONTH) + 1) + "月" +
        calendar.get(Calendar.DAY_OF_MONTH) + "日" +
        calendar.get(Calendar.HOUR_OF_DAY) + ":" + calendar.get(Calendar.MINUTE) +
        ":" + calendar.get(Calendar.SECOND);
        System.out.println(sDate);
    }
}
```

运行结果:

```
Date 原始输出: Fri Nov 05 23:23:32 CST 2021
距 1970-1-1 0 时的毫秒数: 1636125812156
LocalDate:2021-11-05
2021 年 11 月 5 日 23:23:32
```

5.2.4 LocalDate 类

java.time 包提供了日期和时间的 API,包括的类型如下。

- 本地日期和时间: LocalDate、LocalTime、LocalDateTime;
- 带时区的日期和时间: ZonedDateTime;
- 时刻: Instant;
- 时区: ZoneId、ZoneOffset;
- 时间间隔: Duration。

java.time.LocalDate 是 Java 8 新增的一个日期类,该类提供了对日期(年月日)的简便操作。该类不能代表时间线上的即时信息,只是日期的描述。在 LocalDate 类中提供了两个获取日期对象的方法: now()和 of(int year, int month, int dayOfMonth),可通过这两个方法构造一个 LocalDate 对象,例如:

```
//构造日期为 2021-11-11
```

```
LocalDate date = LocalDate.of(2021,11,11);
//以默认时区的系统时间构造 LocalDate 对象
LocalDate now = LocalDate.now();
```

表 5.9 列出了 LocalDate 类的主要方法。

表 5.9 java.time.LocalDate 类的主要方法

方 法 名	说 明
static LocalDate now()	返回当前日期
static LocalDate of(int year,int m,int d)	根据参数指定的年月日设置日期
static LocalDate parse(CharSequence text)	将特定格式的字符串转换为 LocalDate 类型
int getDayOfMonth()	获取当前日期是所在月的第几天
int getMonthValue()	获取当前日期所在月份数值
boolean isLeapYear()	判断当前日期是否为闰年
LocalDate with(TemporalField t, long v)	给特定时间字段赋予新值
LocalDate withDayOfMonth(int day)	将当前日期的日替换为参数值
LocalDate withDayOfYear(int day)	将当前日期的天数替换为参数值
LocalDate withMonth(int month)	将当前日期的月份替换为参数值
LocalDate withYear(int year)	将当前日期的年份替换为参数值
LocalDate minusDays(long days)	将当前日期减少参数天
LocalDate minusWeeks(long weeks)	将当前日期减少参数周
LocalDate minusMonths(long months)	将当前日期减少参数月
LocalDate minusYears(Long years)	将当前日期减少参数年
LocalDate plusDays(long days)	将当前日期增加参数天
LocalDate plusWeeks(long weeks)	将当前日期增加参数周
LocalDate plusMonths(long months)	将当前日期增加参数月
LocalDate plusYear(long years)	将当前日期增加参数年

注意：Java 8 新修订的日期和时间类较旧版本做了如下方面的完善。

- 月份 Month 的范围用 1～12 表示 1～12 月，星期 Week 的范围用 1～7 表示星期一至星期日；
- 严格区分了时刻、本地日期、本地时间和带时区的日期和时间，对日期和时间的运算更加简便；
- 新 API 的类型几乎都是不变类型(类似于 String)，不必担心值被修改。

5.2.5 LocalTime 类

java.time.LocalTime 类是一个时间类，提供了包括时、分、秒和纳秒等时钟信息，与 LocalDate 类一样，该类不能代表时间线上的即时信息，只是时间的描述。在 LocalTime 类中提供了获取时间对象的方法，与 LocalDate 用法类似。同时 LocalTime 类也提供了与日期类相对应的时间格式化、增减时分秒等常用方法，这些方法与日期类相对应。

```
//构造时间为 10:10:35
LocalTime localTime = LocalTime.of(10,10,35);
//以默认时区的系统时钟构造 LocalTime 对象
LocalTime time = LocalDTime.now();
```

表 5.10 列出了 LocalTime 类的主要方法。

表 5.10 java.time.LocalTime 类的主要方法

方 法 名	说 明
static LocalTime now()	返回当前时间
static LocalTime of(int hour,int m,int s)	根据参数指定的时分秒设置时间
static LocalTime parse(CharSequence text)	将特定格式的字符串转换为 LocalTime 类型
int getNano()	获取当前时间的纳秒数
int getHour()	获取当前时间的小时
int getMinute()	获取当前时间的分钟
int getSecond()	获取当前时间的秒
LocalDateTime atDate(LocalDate date)	给特定日期字段赋予新值
LocalTime with(TemporalField t, long v)	给特定时间字段赋予新值
LocalTime withHour(int hour)	将当前时间的小时替换为参数值
LocalTime withMinute(int minute)	将当前时间的分钟替换为参数值
LocalTime withSecond(int second)	将当前时间的秒替换为参数值
LocalTime withNano(int nano)	将当前时间的纳秒替换为参数值
LocalTime minusHours(long hours)	将当前时间减少参数小时
LocalTime minusMinutes(long min)	将当前时间减少参数分钟
LocalTime minusSeconds(long sec)	将当前时间减少参数秒
LocalTime minusNanos(long nanos)	将当前时间减少参数纳秒
LocalTime plusHours (long hours)	将当前时间增加参数小时
LocalTime plus Minutes(long min)	将当前时间增加参数分钟
LocalTime plusSeconds(long sec)	将当前时间增加参数秒
LocalTime plusNanos(long nanos)	将当前时间增加参数纳秒
boolean isBefore(LocalTime time)	当前时间对象与参数比较,若在参数之前返回 true
boolean isAfter(LocalTime time)	当前时间对象与参数比较,若在参数之后返回 true

5.2.6 LocalDateTime 类

java.time.LocalDateTime 类表示一个本地日期和时间,相当于 LocalDate 和 LocalTime 二者的综合体。

【例 5.5】 Example5_05.java

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
public class Example5_05 {
    public static void main(String[] args) {
        LocalDateTime localDateTime1 = LocalDateTime.now();
        LocalDateTime localDateTime2 = LocalDateTime.of(2021, 11, 11, 18, 56, 52);
        System.out.println(localDateTime1);
        System.out.println(localDateTime2);
        System.out.println("localDateTime1 星期: " + localDateTime1.getDayOfWeek());
        //将 LocalDateTime 转换为 LocalDate 与 LocalTime
        LocalDate date = localDateTime1.toLocalDate();
        LocalTime time = localDateTime1.toLocalTime();
        System.out.println(date);
        System.out.println(time);
```

```
        int hour = time.getHour();
        int minute = time.getMinute();
        int second = time.getSecond();
        int nano = time.getNano();
        System.out.println("时 -> " + hour);
        System.out.println("分 -> " + minute);
        System.out.println("秒 -> " + second);
        System.out.println("Nano -> " + nano);
    }
}
```

运行结果：

```
2021-11-06T00:37:23.557737600
2021-11-11T18:56:52
localDateTime1 星期: SATURDAY
2021-11-06
00:37:23.557737600
时 -> 0
分 -> 37
秒 -> 23
Nano -> 557737600
```

注意：LocalDateTime、LocalDate 和 LocalTime 都严格按照国际标准 ISO 8601 规定的日期和时间格式进行打印，ISO 8601 规定日期和时间之间使用分隔符 T 隔开，标准格式如下。

- 日期：yyyy-MM-dd；
- 时间：HH:mm:ss；
- 带毫秒的时间：HH:mm:ss.SSS；
- 日期和时间：yyyy-MM-dd'T'HH:mm:ss；
- 带毫秒的日期和时间：yyyy-MM-dd'T'HH:mm:ss.SSS。

LocalDateTime、LocalDate 和 LocalTime 三个类都提供了 parse()方法，可将符合 ISO 8601 格式的字符串转换为相应的日期和时间类型，例如：

```
//将字符串转换为 LocalDateTime 类型
LocalDateTime dt = LocalDateTime.parse("2021-11-06T08:16:32");
//将字符串转换为 LocalDate 类型
LocalDate d = LocalDate.parse("2021-11-06");
//将字符串转换为 LocalTime 类型
LocalTime t = LocalTime.parse("08:16:32");
```

5.2.7 Instant 类

java.time.Instant 类代表某个时间。其内部由两个 long 字段组成，第一部分保存的是标准 Java 计算时代(就是 1970 年 1 月 1 日开始)到现在的秒数，第二部分保存的是纳秒数。表 5.10 列出了 Instant 类的主要方法。

表 5.11 java.time.Instant 类的主要方法

方 法 名	说 明
now()	从系统时钟获取当前瞬时
now(Clock clock)	从指定时钟获取当前瞬时
ofEpochSecond(long epochSecond)	使用从标准 Java 计算时代开始的秒数获得一个 Instant 的实例

续表

方 法 名	说 明
ofEpochMilli(long epochMilli)	使用从标准 Java 计算时代开始的秒数获得一个 Instant 的实例
getEpochSecond()	从 1970-01-01T00:00:00Z 的 Java 时代获取秒数
getNano()	获取 Instant 对象表示的纳秒数
parse(CharSequence text)	从一个指定格式的文本字符串获取一个 Instant 的实例
from(TemporalAccessor tenporal)	从时间对象获取一个 Instant 的实例

【例 5.6】 Example5_06.java

```
import java.time.Instant;
public class Example5_06 {
    public static void main(String[] args) {
        //创建 Instant 对象
        Instant instant = Instant.now();
        //以 ISO 8601 格式输出
        System.out.println(instant);
        //java.util.Date --> Instant 类型
        instant = Instant.ofEpochMilli(new java.util.Date().getTime());
        //将字符串转换为 Instant 类型
        instant = Instant.parse("2021-11-11T10:10:35Z");
        System.out.println(instant);
        Instant in = instant.plusSeconds(30);
        System.out.println(in);
    }
}
```

运行结果:

```
2021-11-05T17:19:56.009729500Z
2021-11-11T10:10:35Z
2021-11-11T10:11:05Z
```

5.2.8 Duration 类和 Period 类

Duration 类基于时间值,其作用范围是天、时、分、秒、毫秒和纳秒,而 Period 类基于日期类,计算两个日期之间的差值。

【例 5.7】 Example5_07.java

```
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.Period;
public class Example5_07 {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        LocalDateTime ago = LocalDateTime.of(1921,07,01,0,0,0);
    //创建 Duration 对象
        Duration duration = Duration.between(ago,now);
        System.out.println("间隔天: " + duration.toDays());
        System.out.println("间隔小时: " + duration.toHours());
        System.out.println("间隔分钟: " + duration.toMinutes());
    //创建 Period 对象
```

```
        Period period =
            Period.between(ago.toLocalDate(),now.toLocalDate());
        System.out.println("间隔年: " + period.getYears());
        System.out.println("间隔月: " + period.getMonths());
        System.out.println("间隔天: " + period.getDays());
    }
}
```

运行结果：

```
间隔天: 36653
间隔小时: 879673
间隔分钟: 52780394
间隔年: 100
间隔月: 4
间隔天: 5
```

5.2.9 日期格式化

1. DateFormat 类

java.text.DateFormat 类提供了两个功能：定义日期时间格式，实现 String 与日期时间之间的转换。DateFormat 是一个抽象类，SimpleDateFormat 是其实现类。该类的基本用法如下。

```
//创建 java.util.Date 对象
Date date = new Date();
//实例化 DateFormat 对象,并指定日期格式为"yyyy-MM-dd HH:mm:ss"
DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
//将 Date 类型转换为上述格式的字符串
String s = dateFormat.format(date);
//将"2021-11-06 13:10:35"转换为 java.util.Date 类型
String d = "2021-11-06 13:10:35";
date = dateFormat.parse(d);
```

可以看到，SimpleDateFormat 类提供的构造方法用来指定日期和时间格式，format()方法可以将一个 Date 对象转换为相应格式的字符串；反之，parse()方法可以将指定格式的字符串转换为 Date 类型。SimpleDateFormat 类的日期和时间格式由模式字符串指定，该类定义了模式字母，所有其他字符'A'～'Z'和'a'～'z'都被保留，具体见表 5.12。

表 5.12 模式字母表

字母	日期或时间元素	类 型	示 例
G	Era 标志符	Text	AD
y	年	Year	2021；21
M	年中的月份	Month	July；Jul；07
w	年中的周数	Number	36
W	月份中的周数	Number	2
D	年中的天数	Number	310
d	月份中的天数	Number	7
E	星期中的天数	Text	Sunday；Sun
a	AM/PM 标志	Text	PM
H	一天中的小时数(0～23)	Number	0

续表

字母	日期或时间元素	类　型	示　例
k	一天中的小时数(1～24)	Number	24
K	一天中的小时数(0～11)	Number	0
h	一天中的小时数(1～12)	Number	12
m	小时中的分钟数	Number	30
s	分钟中的秒数	Number	59
S	毫秒数	Number	988

2. DateTimeFormatter 类

java.text.DateFormat 类主要实现在 java.util.Date 进行格式化显示。如果要对 LocalDateTime 进行格式化显示,需要使用 DateTimeFormatter 类。DateTimeFormatter 是不变对象,并且是线程安全的。

【例 5.8】 Example5_08.java

```
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.TemporalAccessor;
import java.util.Date;

public class Example5_08 {
    public static void main(String[] args) {
        //创建 java.util.Date 对象
        Date date = new Date();
        //实例化 DateFormat 对象,并指定日期格式为"yyyy-MM-dd HH:mm:ss"
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        //将 Date 类型转换为上述格式的字符串
        String s = dateFormat.format(date);
        //将"2021-11-06 13:10:35"转换为 java.util.Date 类型
        String d = "2021-11-06 10:10:35";
        try {
            date = dateFormat.parse(d);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        //创建 LocalDateTime 对象
        LocalDateTime dt = LocalDateTime.now();
        //定义日期格式
        var f1 = DateTimeFormatter.ofPattern("yyyy 年 MM 月 dd 日");
        String s1 = dt.format(f1);
        System.out.println(s1);
        //定义时间格式
        DateTimeFormatter f2 = DateTimeFormatter.ofPattern("HH:mm:ss");
        String s2 = dt.format(f2);
        System.out.println(s2);
        TemporalAccessor ta = f1.parse("2021 年 09 月 10 日");
        System.out.println(ta);
    }
}
```

运行结果：

```
2021 年 11 月 07 日
14:10:22
{},ISO resolved to 2021 - 09 - 10
```

5.3 数值与随机数

本节主要介绍有关初等数学的相关函数操作类、随机类和包装类。包装类提供了对Java 的八种基本数据类型封装的方法。

5.3.1 Math 类

java.lang.Math 类提供了大量类方法，用于求解基本数学运算，如初等指数、对数、三角函数、平方根、绝对值、近似值等。Math 类的常见方法见表 5.13。

表 5.13 java.lang.Math 类的常见方法

方 法 名	说 明
abs(a)	计算绝对值
sqrt(a)	计算平方根
ceil(a,b)	计算大于参数的最小整数，简称天花板数
floor(a)	计算小于参数的最小整数，简称地板数
round(a)	计算小数进行四舍五入后的结果
max(a,b)	计算两个数的较大值
min(a,b)	计算两个数的较小值
random()	生成一个大于 0.0 且小于 1.0 的随机值
pow(a,b)	计算 a^b
log(a)	计算以自然数为底数的对数值
sin(a)	求参数的正弦值
cos(a)	求参数的余弦值
tan(a)	求参数的正切值
asin(a)	求参数的反正弦值
acos(a)	求参数的反余弦值
atan(a)	求参数的反正切值
toDegrees(a)	将参数转换为角度
toRadians(a)	将角度转换为弧度

5.3.2 Random 类

java.util.Random 类可以生成指定范围的随机数，包括整数和浮点数。Random 类中提供了两个构造方法和随机生成整数和浮点数的方法。Random 类的构造方法和常见方法见表 5.14。

表 5.14 java.util.Random 类的构造方法和常见方法

方 法 名	说 明
Random()	构造一个随机数生成器
Random(long seed)	用种子 seed 构造一个随机数生成器

续表

方 法 名	说 明
boolean nextBoolean()	生成一个 boolean 类型的随机数
float nextFloat()	生成一个 float 类型的随机数
double nextDouble()	生成一个 double 类型的随机数
int nextInt()	生成一个 int 类型的随机数
int nextInt(int bound)	生成一个[0,bound]范围内 int 类型的随机数
int nextLong()	生成一个 long 类型的随机数

Random 类的两个构造方法,无参构造方法具有更强的随机性,通过它创建的 Random 实例对象每次使用的种子都是随机的,因此每个对象产生的随机数不同。如果希望创建的多个 Random 实例对象产生相同的随机数,则使用带有种子值的构造方法,传入相同的种子值即可,如例 5.9 所示。

【例 5.9】 Example5_09.java

```
import java.util.Random;
public class Example5_09 {
    public static void main(String[] args) {
        testRandom();
    }
    public static void testRandom() {
        System.out.println("Random 不设置种子: ");
        for (int i = 0; i < 5; i++) {
            Random random = new Random();
            for (int j = 0; j < 10; j++) {
                System.out.print(" " + random.nextInt(100) + "\t");
            }
            System.out.println("");
        }
        System.out.println("");
        System.out.println("Random 设置种子: ");
        for (int i = 0; i < 5; i++) {
            Random random = new Random();
            random.setSeed(100);
            for (int j = 0; j < 10; j++) {
                System.out.print(" " + random.nextInt(100) + "\t");
            }
            System.out.println("");
        }
    }
}
```

某次运行结果如下。

```
Random 不设置种子:
  31   21   67   4    47   69   91   77   97   89
  58   17   80   63   44   77   65   73   98   67
  12   35   36   92   70   83   42   76   49   68
  8    97   57   35   71   76   70   59   2    57
  94   28   37   40   65   53   43   14   53   12
Random 设置种子:
```

```
15  50  74  88  91  66  36  88  23  13
15  50  74  88  91  66  36  88  23  13
15  50  74  88  91  66  36  88  23  13
15  50  74  88  91  66  36  88  23  13
15  50  74  88  91  66  36  88  23  13
```

例 5.10 中用 StringBuilder 和 Random 类提供了两种生成验证码的方法，其一是生成一个 4 位数字组成的验证码，其二是生成一个由数字和字母组成的 4 位验证码。

【例 5.10】 Example5_10.java

```
import java.util.Random;
public class Example5_10 {
    public static void main(String[] args) {
        System.out.println("四位数字: " + numberCode());
        System.out.println("四位字符: " + stringCode());
    }
    //生成 4 位数字组成的验证码
    static String  numberCode(){
        Random r1 = new Random();
        int i = 0 ;
        //随机数不能小于 1000
        i = r1.nextInt(10000);
        while(true){
            if(i<1000)
                i = r1.nextInt(10000);
            else
                break;
        }
        return String.valueOf(i);
    }
    //生成 4 位字符组成的验证码
    static String stringCode(){
        Random r2 = new Random();
        //构造随机数产生的范围
        String s = "abcdefghigklmnopqrstuvwxyz0123456789";
        StringBuilder s1 = new StringBuilder(s);
        StringBuilder code = new StringBuilder("");
        for(int i=0;i<4;i++){
            int index = r2.nextInt(36);
            code.append(s1.charAt(index));
        }
        return code.toString();
    }
}
```

某次运行结果如下。

```
四位数字: 7169
四位字符: i39s
```

5.3.3 包装类

Java 作为一种面向对象的语言，类将方法和数据有机整合为一体。但 Java 提供的 8 种基本数据类型并不符合面向对象的思想，特别是在一些场景下，需要把基本数据类型作为对

象来使用。为了解决这样的问题,JDK 提供了包装类,将基本数据类型封装为引用数据类型。表 5.15 列出了基本数据类型与包装类的对应关系。

表 5.15 基本数据类型对应的包装类

基本数据类型	包 装 类	基本数据类型	包 装 类
byte	Byte	float	Float
boolean	Boolean	int	Integer
char	Character	long	Long
double	Double	short	Short

包装类提供了丰富的方法和常量,本节以 Integer 类为例说明包装类的基本用法,表 5.16 列出了 Integer 类的主要方法。

表 5.16 Integer 类的主要方法

方 法 名	说 明
xxx xxxValue()	如 floatValue(),返回指定类型的数值
static int parseInt(String s)	将由数字组成的字符串转换为 int 类型
static Integer valueOf(String s)	将字符串 s 转换为 Integer 类型
static String toBinaryString(int i)	将参数 i 转换为二进制形式的字符串
static String boHexString(int i)	将参数 i 转换为十六进制形式的字符串
static String toOctalString(int i)	将参数 i 转换为八进制形式的字符串

【例 5.11】 Example5_11.java

```
public class Example5_11 {
    public static void main(String[] args) {
        Integer i = 100;
        //可以直接将 Integer 类型的变量赋给 int 类型变量
        int j = i;
        double d = i.doubleValue();
        String s = "1024";
        //将 String 类型转换为 int 类型
        j = Integer.parseInt(s);
        //使用 valueOf()方法将字符串 s 转换为 Integer 类型
        i = Integer.valueOf(s);
        //将 j 转换为十六进制的字符串
        System.out.println(Integer.toHexString(j));
    }
}
```

运行结果:

```
400
```

注意:基本数据类型与对应包装类的不同之处如下。

- 包装类型可以是 null,而基本数据类型不可以;
- 包装类可用于泛型,而基本数据类型不可以;
- 基本数据类型与包装类占用的内存空间不同,基本数据类型较包装类更高效。

从 Java 9 之后,Integer 不再推荐使用其构造方法创建 Integer 对象,这意味着 int 类型与 Integer 类型不再使用装箱、拆箱的方法进行转换,效率更高,对于其他包装类也是如此。

5.3.4 BigInteger 类与 BigDecimal 类

1. BigInteger 类

应用开发中难免遇到一些超大的数，超出基本数据类型的取值范围，那么 Java 提供了两个类：BigInteger 和 BigDecimal，分别表示超大的整数和浮点数。java.math.BigInteger 用于表示任意大小的整数，其内部使用一个 int[]数组来模拟一个大整数。java.math.BigDecimal 表示一个任意大小且精度完全准确的浮点数。BigInteger 类的构造方法和其他方法见表 5.17。

表 5.17 BigInteger 类的构造方法和其他方法

方 法 名	说 明
BigInteger(String val)	将字符串 val 构造为 BigInteger 对象
BigInteger abs()	返回大整数的绝对值
BigInteger add(BigInteger val)	返回两个大整数的和
BigInteger andNot(BigInteger val)	返回两个大整数的按位与非结果
BigInteger and(BigInteger val)	返回两个大整数的按位与的结果
BigInteger divide(BigInteger val)	返回两个大整数的商
BigInteger max(BigInteger val)	返回两个大整数的较大者
BigInteger min(BigInteger val)	返回两个大整数的较小者
BigInteger mod(BigInteger val)	用当前大整数对 val 求模
BigInteger multiply(BigInteger val)	返回两个大整数的积
BigInteger negate()	返回当前大整数的相反数
BigInteger not()	返回当前大整数的非
BigInteger or(BigInteger val)	返回两个大整数的按位或
BigInteger pow(BigInteger val)	返回当前大整数的 val 幂次方
BigInteger reminder(BigInteger val)	返回当前大整数除以 val 的余数
BigInteger xor(BigInteger val)	返回两个大整数的异或
BigInteger substract(BigInteger val)	返回两个大整数相减的结果
int intValue()	返回大整数的 int 类型的值
long longValue()	返回大整数的 long 类型的值
double doubleValue()	返回大整数的 double 类型的值
float floatValue()	返回大整数的 float 类型的值
byte[] toByteArray(BigInteger val)	将大整数二进制反码保存在 byte 类型的数组
String toString()	将当前大整数转换成十进制的字符串形式
long longValueExact()	返回大整数的准确 long 类型

通过表 5.17 可知，大整数的算术和逻辑运算不能再使用传统的计算方式，必须使用 BigInteger 类提供的方法进行相应计算，如例 5.12 所示。

【例 5.12】 Example5_12.java

```
import java.math.BigInteger;
public class Example5_12 {
    public static void main(String[] args) {
        BigInteger v1 = new BigInteger("99999");
        BigInteger v2 = new BigInteger("99");
        //幂运算
```

```
        BigInteger n = v1.pow(10);
        System.out.println("原始数据 BigInteger: " + n);
        System.out.println("转换为 long 类型: " + n.longValue());
        try{
            //使用 longValueExact()转换为 long 类型
            //超出 long 的范围时会抛出 ArithmeticException
            System.out.println(n.longValueExact());
        }catch (ArithmeticException e){
            System.out.println(e.getMessage());
        }
        //加法运算
        n = v1.add(v2);
        System.out.println(n);
        //非运算
        n = v2.not();
        System.out.println(n);
    }
}
```

运行结果：

```
原始数据 BigInteger: 999900004499880002099974800209998800004499990000001
转换为 long 类型: 485031440369308097
BigInteger out of long range
100098
-100
```

2. BigDecimal 类

java.math.BigDecimal 类用来对超过 16 位有效位的浮点数进行精确运算，与 BigInteger 类一样，我们也不能使用传统的算术运算直接对 BigDecimal 类型的对象进行计算，而必须调用其提供的相应方法计算，方法中的参数也必须是 BigDecimal 对象。表 5.18 列出了 BigDecimal 类的构造方法和主要方法。

表 5.18 BigDecimal 类的构造方法和主要方法

方 法 名	说 明
BigDecimal(String val)	创建一个具有参数所指定以字符串表示的数值的对象
BigDecimal(double val)	创建一个具有参数所指定双精度值的对象
BigDecimal setScale(int s,RoundingMode mode)	设置小数点保留位数及舍入方式
BigDecimal add(BigDecimal val)	返回两个大浮点数的和
BigDecimal substract(BigDecimal val)	返回两个大浮点数相减的结果
BigDecimal multiply(BigDecimal val)	返回两个大浮点数的积
BigDecimal divide(BigDecimal val)	返回两个大浮点数的商
BigDecimal divide(BigDecimal val,int scale, int mode)	返回两个大浮点数的商，参数 val 表示除数，参数 scale 表示小数点保留位数，参数 mode 表示舍入模式

例 5.13 以计算房屋公积金贷款的利息为例，计算本息合计金额。使用 NumberFormat 类设置货币符号及百分比格式，然后使用 BigDicemal 类表示大浮点数对象并进行算法运算。

【例 5.13】 Example5_13.java

```
import java.math.BigDecimal;
import java.math.RoundingMode;
```

```
import java.text.NumberFormat;
public class Example5_13 {
    public static void main(String[] args) {
        //建立货币格式化引用
        NumberFormat currency = NumberFormat.getCurrencyInstance();
        //建立百分比格式化引用
        NumberFormat percent = NumberFormat.getPercentInstance();
        //百分比小数点最多3位
        percent.setMaximumFractionDigits(4);
        //贷款金额
        BigDecimal loanAmount = new BigDecimal("395200.99");
        //公积金贷款利率
        BigDecimal interestRate = new BigDecimal("0.035");
        //计算贷款一年的利息
        BigDecimal interest = loanAmount.multiply(interestRate);
        //应还金额,小数点保留两位且四舍五入
        BigDecimal total = loanAmount.add(interest).setScale(2,RoundingMode.HALF_UP);
        System.out.println("贷款金额:\t" + currency.format(loanAmount));
        System.out.println("利率:\t" + percent.format(interestRate));
        System.out.println("利息:\t" + currency.format(interest));
        System.out.println("本息合计: \t" + total);
    }
}
```

运行结果：

```
贷款金额:￥395,200.99
利率:    3.5%
利息:    ￥13,832.03
本息合计:    409033.02
```

注意：BigDecimal 类提供了多种构造方法，建议使用参数类型为 String 的构造方法构造 BigDecimal 对象。由于浮点数无法使用二进制精确表示，计算机表示浮点数由两部分组成：指数和尾数，因此会失去一定的精确度，有些浮点数运算也会产生一定的误差。建议进行商业计算，特别是小数点保留指定位数时，使用 BigDecimal 类。

5.4 系统相关类

Java 提供了两个系统相关信息的类，System 类代表当前 Java 程序的运行平台，Runtime 类表示当前 JVM 的工作信息。

5.4.1 System 类

java.lang.System 类是应用最为广泛的一个类，该类提供的类变量 out 在很多案例中都有应用，System 类定义了一些与系统相关的属性和方法，具体见表 5.19。

表 5.19 System 类的方法

方 法 名	说 明
static void exit(int status)	该方法用于终止当前正在运行的 JVM，参数 status 表示状态码，若非 0 表示异常终止
static void gc()	运行垃圾回收器回收垃圾

续表

方 法 名	说 明
static long currentTimeMillis()	以毫秒为单位返回当前时刻距离 1970-01-01 0 时的时间间隔
static void arraycopy(Object src, int srcpos, Object dest, int destPos, int length)	从数组 src 复制到数组 dest,复制从指定位置开始,到目标数组的指定位置结束
static Properties getProperties()	获取当前的系统属性
static String getProperty(String key)	获取指定名称的系统属性

此外,System 类还提供了以下三个类变量。

- static PrintStream out: 标准输出流;
- static PrintStream err: 标准错误输出流;
- static InputStream in: 标准输入流。

out 和 err 都属于 PrintStream 类型,而 PrintStream 类提供了诸如 println()、printf()、print()等打印方法。因此,经常借助 System 类的类变量实现向控制台输出。

in 为 InputStream 类型,InputStream 提供了 read()方法,可以通过标准输入(键盘)接收一个字节的数据。

【例 5.14】 Example5_14.java

```
import java.io.IOException;
import java.util.Enumeration;
import java.util.Properties;
public class Example5_14 {
    public static void main(String[] args) {
        char src[] = new char[10];
        System.out.println("请输入 10 个字符: ");
        //从键盘读入 10 个字符
        for(int i = 0;i < src.length;i++){
            try{
                src[i] = (char)System.in.read();
            }catch (IOException e){
                e.printStackTrace();
            }
        }
        for(int i = 0;i < src.length;i++)
            System.out.print(src[i] + " ");
        System.out.println();
        //数组复制
        char[] dest = new char[5];
        //u 将数组 src 的前 5 个元素复制到数组 dest
        System.arraycopy(src,0,dest,0,5);
        String s = new String(dest);
        System.out.println("复制的数组内容: " + s);
        //获取系统日期
        long time = System.currentTimeMillis();
        System.out.println("从 1970 - 01 - 01 0 时到现在的毫秒数:" + time);
        //获取系统信息
        Properties p = System.getProperties();
        Enumeration en = p.propertyNames();
```

```
            while(en.hasMoreElements()){
                String key = (String)en.nextElement();
                if("java.vm.version".equals(key))//仅输出 JVM 版本信息
                    System.out.println(key + ":\t" + System.getProperty(key));
            }
        }
}
```

运行结果：

```
请输入 10 个字符:
hello,java
h e l l o , j a v a
复制的数组内容: hello
从 1970 - 01 - 01 0 时到现在的毫秒数:1636698485533
java.vm.version:16.0.2 + 7 - 67
```

上述例子使用 System 类从键盘上接收了 10 个字符存入数组 src，然后使用 arraycopy()方法将 src 的前 5 个字符复制到数组 dest。又使用 System 类获取系统时间和系统信息等。注意本例使用 Enumeration 接口遍历 Properties 对象，相关方法在 5.6 节介绍。

5.4.2 Runtime 类

Runtime 类表示 Java 虚拟机运行时的状态，它用于封装 Java 虚拟机的进程。注意，每次 java 命令启动虚拟机都对应一个 Runtime 实例，并且 Runtime 属于单例模式，有且仅有一个 Runtime 实例。Runtime 实例的创建采用如下方式。

```
Runtime runtime = Runtime.getRuntime();
```

表 5.20 列出了 Runtime 类的主要方法。

表 5.20 Runtime 类的主要方法

方 法 名	说 明
Runtime getRunTime()	该方法返回一个 Runtime 实例
Process exec(String command)throws IOException	根据指定路径 command 返回一个进程对象
long freeMemory()	以字节为单位返回当前 JVM 的空闲内存
long maxMemory()	以字节为单位返回 JVM 将尝试使用的最大内存量
long totalMemory()	以字节为单位返回 JVM 中内存总量
static void gc()	运行垃圾回收器回收垃圾
int availableProcessors()	返回 Java 虚拟机可用的处理器数

【例 5.15】 Example5_15.java

```
public class Example5_15 {
    public static void main(String[ ] args) {
        //创建 Runtime 实例
        Runtime runtime = Runtime.getRuntime();
        //获取 JVM 内存信息
        System.out.println("total memory:" +
                runtime.totalMemory()/(1024 * 1024) + "MB");
        System.out.println("max memory:" +
                runtime.maxMemory()/(1024 * 1024) + "MB");
        System.out.println("free memory:" +
```

```
                runtime.freeMemory()/(1024 * 1024) + "MB");
        //调用垃圾回收器
        runtime.gc();
        try {
            //运行计算器
            Process process = runtime.exec("calc.exe");
            Thread.sleep(1000 * 5);
            //5s 后关闭计算器进程
            process.destroy();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

运行结果：

```
total memory:64M
max memory:1010M
free memory:61M
```

注意：本例的运行结果在不同机器上有所不同。另外，本例还使用 exec()方法启动了计算器进程，5s 后关闭该进程。

5.5 正则表达式

正则表达式(Regular Expression)描述了一种字符串匹配模式(pattern)，可以用来检查一个串是否含有某种子串、将匹配的子串替换或者从某个串中取出符合某个条件的子串等。Java 提供了正则表达式校验有关类和方法。

5.5.1 元字符

构造正则表达式的方法和创建数学表达式的方法一样。使用多种元字符与运算符可以将子表达式结合在一起来创建复杂的表达式。正则表达式的组件可以是单个字符、字符集合、字符范围、字符间的选择或者所有这些组件的任意组合。

正则表达式是由普通字符(例如字符 a～z)以及特殊字符(称为“元字符”)组成的文字模式。模式描述在搜索文本时要匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。常见的正则表达式字符如表 5.21 所示。

表 5.21 常见的正则表达式字符

字 符	说 明
$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配'\n'或'\r'。要匹配 $ 字符本身，请使用\ $
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用\(和\)
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用\ *
+	匹配前面的子表达式一次或多次。要匹配＋字符，请使用\＋
.	匹配除换行符\n 之外的任何单字符。要匹配.，请使用\.
[	标记一个中括号表达式的开始。要匹配[，请使用\[

续表

字 符	说 明
?	匹配前面的子表达式零次或一次,或指明一个非贪婪限定符。要匹配? 字符,请使用\?
\	将下一个字符标记为特殊字符、原义字符、向后引用或八进制转义符。例如,'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\",而 '\(' 则匹配 "("
^	匹配输入字符串的开始位置,除非在方括号表达式中使用,当该符号在方括号表达式中使用时,表示不接受该方括号表达式中的字符集合。要匹配^字符本身,请使用\^
{	标记限定符表达式的开始。要匹配{,请使用\{
|	指明两项之间的一个选择。要匹配|,请使用|
*	匹配前面的子表达式零次或多次。例如,zo * 能匹配"z"以及"zoo"。 * 等价于{0,}
+	匹配前面的子表达式一次或多次。例如,'zo+'能匹配"zo"以及"zoo",但不能匹配"z"。+等价于{1,}
?	匹配前面的子表达式零次或一次。例如,"do(es)?"可以匹配"do" "does"中的"does" "doxy"中的"do"。? 等价于{0,1}
{n}	n 是一个非负整数。匹配确定的 n 次。例如,'o{2}' 不能匹配 "Bob" 中的 'o',但是能匹配 "food"中的两个 o
{n,}	n 是一个非负整数。至少匹配 n 次。例如,'o{2,}' 不能匹配 "Bob" 中的 'o',但能匹配 "foooood"中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o * '
{n,m}	m 和 n 均为非负整数,其中,n≤m。最少匹配 n 次且最多匹配 m 次。例如,"o{1,3}"将匹配"fooooood"中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格
x|y	匹配 x 或 y。例如,'z|food' 能匹配"z"或"food"。'(z|f)ood' 则匹配"zood"或"food"
[xyz]	字符集合。匹配所包含的任意一个字符。例如,'[abc]' 可以匹配 "plain" 中的 'a'
[^xyz]	负值字符集合。匹配未包含的任意字符。例如,'[^abc]' 可以匹配 "plain" 中的'p'、'l'、'i'、'n'
[a-z]	字符范围。匹配指定范围内的任意字符。例如,'[a-z]' 可以匹配 'a'～'z' 范围内的任意小写字母字符
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如,'[^a-z]' 可以匹配任何不在 'a'～'z' 范围内的任意字符
\w	匹配字母、数字、下画线。等价于'[A-Za-z0-9_]'
\W	匹配非字母、数字、下画线。等价于 '[^A-Za-z0-9_]'

基于上述正则表达式的字符说明,表 5.22 列出了一些常用的正则表达式。

表 5.22 常用的正则表达式

名 称	正则表达式
Email	\w[-\w.+] * @([A-Za-z0-9][-A-Za-z0-9]+\.)+[A-Za-z]{2,14}
URL	^((https|http|ftp|rtsp|mms)?://)[^\s]+
邮政编码	\d{6}
身份证号	\d{17}[\d|x]|\d{15}
国内手机号	0? 1(3|5|6|7|8|9)[0-9]{9}
6～18 位大小写字母、数字、下画线组成密码	^[a-zA-Z]\w{5,17}$

5.5.2 Pattern 类与 Matcher 类

1. Pattern 类

java.util.regex.Pattern 类是对正则表达式的编译表示。其用法是先将正则表达式使

用 Pattern 类的实例编译,然后使用生成的模式创建匹配器对象,该对象可以根据正则表达式匹配任意字符序列。采用如下方式创建匹配器对象:

```
Pattern pattern = Pattern.compile("a*b");
Matcher matcher = pattern.matcher("aaaaaab");
boolean result = matcher.matches();
```

表 5.23 列出了 Pattern 类的主要方法。

表 5.23 Pattern 类的主要方法

方 法 名	说 明
String[] split(CharSequence input, int limit)	将给定的输入序列分成这个模式的匹配,有 limit 参数时,表示只匹配前 limit(不含)次
Matcher matcher(CharSequence input)	提供了对正则表达式的分组支持,以及对正则表达式的多次匹配支持
static boolean matches(String regex, CharSequence input)	编译给定的正则表达式并尝试将给定的字符序列与之匹配

2. Matcher 类

java.util.regex.Matcher 类用于在给定的 Pattern 实例的模式控制下进行字符串的匹配工作。Matcher 对象由 Pattern 类的 matcher()方法创建,表 5.24 列出了 Matcher 类的主要方法。

表 5.24 Matcher 类的主要方法

方 法 名	说 明
boolean matches()	对整个字符串进行匹配,只有整个字符串均匹配才返回 true
boolean lookingAt()	对前面的字符串进行匹配,只有匹配到的字符串在最前面才返回 true
boolean find()	对字符串进行匹配,匹配到的字符串可以在任意位置
int end()	返回最后一个字符匹配后的偏移量
String group()	返回匹配到的子字符串
int start()	返回匹配到的子字符串在字符串中的索引位置

下面的程序使用 Pattern 类和 Matcher 类演示校验电子邮件、手机号、密码、邮编、日期等典型正则表达式的应用。

【例 5.16】 Example5_16.java

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class Example5_16 {
    public static void main(String[] args) {
        //邮编
        String postCodeRegex = "\\d{6}";
        //手机
        String telRegex = "1(3|5|6|7|8|9)[0-9]{9}";
        //E-mail
        String emailRegex =
        "\\w[-\\w.+]*@([A-Za-z0-9][-A-Za-z0-9]+\\.)+[A-Za-z]{2,14}";
        //密码,以字母开头,长度为6~18,只能包含字母、数字和下画线
        String passwordRegex = "^[a-zA-Z]\\w{5,17}$";
        //日期格式
        String dateRegex = "^\\d{4}-\\d{1,2}-\\d{1,2}";
```

```
        String post = "475006";
        //1. 使用 Pattern 和 Matcher 类校验
        Pattern p1 = Pattern.compile(postCodeRegex);
        Matcher m1 = p1.matcher(post);
        boolean r1 = m1.matches();
        System.out.println(r1);

        String tel = "14603711234";
        Pattern p2 = Pattern.compile(telRegex);
        Matcher m2 = p2.matcher(tel);
        boolean r2 = m2.matches();
        System.out.println(r2);

        String password = "Yq9zxV0";
        Pattern p3 = Pattern.compile(passwordRegex);
        Matcher m3 = p3.matcher(password);
        boolean r3 = m3.matches();
        System.out.println(r3);

        String date = "2021-11-12";
        Pattern p4 = Pattern.compile(dateRegex);
        Matcher m4 = p4.matcher(date);
        boolean r4 = m4.matches();
        System.out.println(r4);

        //2.也可使用 String 类的 matches()方法校验
        String email = "zhangsan@henu.edu.cn";
        boolean r6 = email.matches(emailRegex);
        System.out.println(r3);

        String str = "电话:3316889,地址:人民路 36 号,门牌号 10#楼东单元 601 房间";
        Pattern p5 = Pattern.compile("\\d+");
        //将 str 中的数字进行分隔
        String[] s = p5.split(str);
        for(String t: s){
            System.out.println(t);
        }
    }
}
```

运行结果：

```
true
false
true
true
true
电话:
,地址:人民路
号,门牌号
#楼东单元
房间
```

注意：String 类也提供了 matches()方法，可以应用特定正则表达式对字符串进行校验，例 5.16 的 E-mail 校验便是使用此方式进行校验。

5.6 集合

数组有两个重要特征：定长；数组中的元素数据类型均一致。正是数组的这两个特征，在程序开发中也带来了不便之处，程序开发人员必须预先判定出要处理的数据数量及其数据类型，否则便无法定义数组。同时由于程序运行时的未知性因素较多，对于声明的数组长度，也有可能产生数组越界或者是声明数组的长度远远大于实际的元素个数，从而导致内存的浪费。

Java 中的容器类能够有效解决数组的上述弊端。容器是一种非常实用的工具类，在 java.util 工具包中。它可以存储不同数据类型的元素，并且其容量可以变化。Java 的容器可以大致分为两类：单列结构 Collection 和双列结构 Map。其中，Collection 又分为两类：List 和 Set。List 表示有序、重复的集合，Set 表示无序、不可重复的集合。Map 则是一种具有映射关系的集合。

5.6.1 集合概述

集合类可存储不同数据类型的对象，并且集合的容量可以动态改变。Java 的集合主要由两个接口派生出来：Collection 和 Map。其中，Collection 是单列结构，Map 是双列结构，由 Key 和 Value(键值对)组成。Collection 和 Map 是 Java 集合层次关系中的两个根接口，这两个接口又派生大量子接口及其实现类。图 5.1 描述了 Java 集合的层次关系。

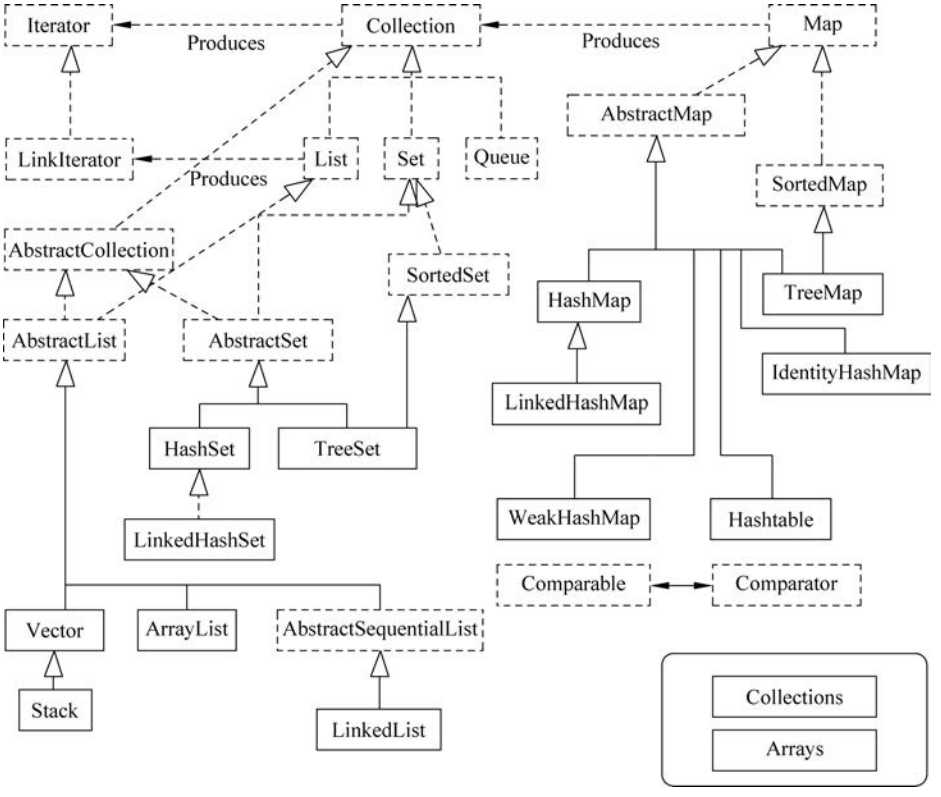

图 5.1 Java 集合的层次关系

在图 5.1 中给出了容器类层次结构关系，其中，Collection 是一组独立的元素，并且这些元素服从某种规则，如 List 必须保持元素特定的顺序，即存入的顺序与取出的顺序一致；而 Set 对象不能有重复的元素，元素存入的顺序与取出顺序也不一定相同；Map 表示一种映射关系，Map 是由键(Key)与值(Value)组成的映射关系，即 Map 是双列结构，且键不允许重复，从某种意义上来讲，Map 就像数据库的数据字典。图 5.2 给出了一些重点集合类型的层次结构及其特点，也是本节重点介绍的内容。

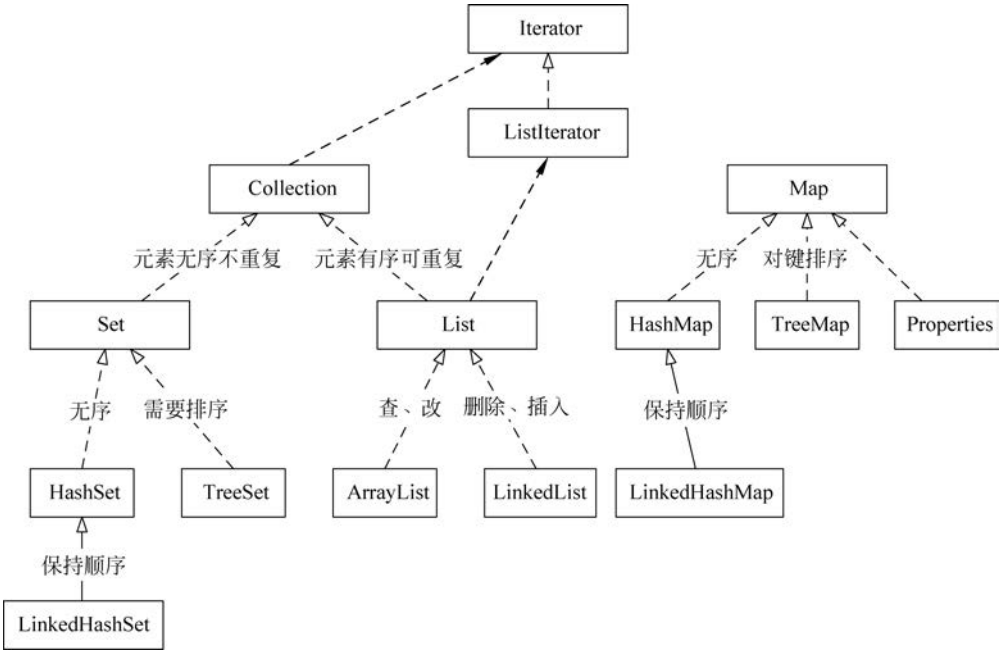

图 5.2 主要集合类型及其特点

注意：集合是一种长度可变、可存储不同数据类型的对象。按照存储结构包括两种类型：单列集合 Collection 和双列集合 Map。

Collection：单列集合的顶层父接口，用于存储一系列符合某种规则的元素。List 和 Set 是 Collection 其中两个应用最为广泛的子接口，List 元素有序且可重复，Set 元素无序且不能重复。

Map：双列集合的顶层父接口，由键(Key)和值(Value)两列组成。每个元素都包括一对键值，其中键的取值必须唯一。在 Map 集合中，可以通过键找到对应的值。

5.6.2 Collection 接口

Collection 作为单列集合的顶层父接口，提供了一些基础的公共方法，通过这些方法可以操作所有类型的单列集合。Collection 接口定义了操作集合元素的常用方法，具体见表 5.25。

表 5.25 Collection 的常用方法

方 法 名	说 明
boolean add(E o)	将对象 o 添加到此集合中
boolean addAll(Collection c)	将容器对象 c 中的所有元素添加到此集合中
void clear()	删除此集合中的所有元素

续表

方 法 名	说 明
boolean equals(Object o)	比较此集合与 o 是否相等
boolean isEmpty()	判断此集合是否为空集合
boolean contains(Object o)	判断此集合是否包含值为 o 的元素
Iterator iterator()	返回一个迭代器,用来访问容器中的各个元素
boolean remove(Object o)	如果此集合中有与 o 相匹配的元素,则删除此元素
boolean removeAll(Collection c)	删除此集合中那些也包含在指定 collection 中的所有元素
boolean retainAll(Collection c)	从此集合中删除容器对象 c 中不包含的元素
Object[] toArray()	返回一个内含此集合所有元素的数组

5.6.3 Iterator 接口

java.util.Iterator 接口称为迭代器,它也是 Java 集合框架的成员,Iterator 主要用来遍历并访问容器类的元素,因此 Collection 集合实现了 Iterator 接口,甚至 List 接口,还支持双向遍历的 ListIterator。使用 Iterator 遍历集合对象并不需要程序开发人员了解集合对象的底层结构,并且创建 Iterator 的代价较小,故 Iterator 被称为"轻量级"的对象。Iterator 接口中具体的常用方法见表 5.26。

表 5.26 Iterator 的常用方法

方 法 名	说 明
boolean hasNext()	判断集合中是否还有元素,如有返回 true,否则返回 false
E next()	获得集合中的下一个元素
void remove()	将迭代器新返回的元素删除

注意:集合遍历的方式有以下五种。

- 基本循环如 for/while 循环:这种方式功能上最为强大,遍历时也可以修改、删除元素。
- Iterator:比较简便的遍历方式,遍历时可以删除元素。
- ListIterator:Iterator 的子接口,专门用于 List 集合的遍历,支持双向遍历。
- Enumeration:遍历 Properties 等双列集合的方式,遍历时不能修改、删除元素。
- foreach:遍历方式上最为简便,同时功能上也最弱,遍历时不能修改、删除元素。

除了基本循环之外,其他方式遍历时,不可以通过集合对象的方法操作集合中的元素,因为会发生 ConcurrentModificationException 异常。Iterator 提供的方法是有限的,只能对元素进行判断、取出、删除的操作,如果想要其他的操作如添加、修改等,就需要使用其子接口 ListIterator。该接口只能通过 List 集合的 listIterator 方法获取。Enumeration 和 foreach 方式功能上更为单一,遍历时不能修改、删除元素。

5.6.4 List 接口

List 接口继承 Collection 接口,List 接口是一种允许有重复元素的有序集合。List 接口添加了面向位置的操作,允许用户对集合中每个元素的插入位置进行精确的控制,还可以根据元素的索引值访问元素,并搜索列表中的元素。表 5.27 列出了 List 接口的常用方法。

表 5.27 List 的常用方法

方 法 名	说 明
boolean add(E o)	将对象 o 追加到此集合的尾部
boolean add(int index, E element)	将对象 element 添加到此向量索引为 index 处
boolean addAll(Collection c)	将集合对象 c 中的全部元素追加到此集合的尾部
boolean addAll(int i,Collection c)	将集合对象 c 中的全部元素添加到此集合索引值为 i 处
Object[] toArray()	将此集合转换为数组
E get(int index)	取该集合索引为 index 的元素值
E set(int index, E element)	用指定的元素替换此集合中索引为 index 处的元素
boolean remove(int index)	删除此集合中索引为 index 的元素
boolean remove(Object o)	删除此集合中元素值为 o 的元素
void removeAll (Collection c)	从此集合中删除 c 的全部元素
boolean contains(Object obj)	测试 obj 是否为此集合中的元素,如是返回 true,否则返回 false
boolean equals(Object obj)	比较此集合与 obj 是否相等,如相等返回 true,否则返回 false
List subList(int from, int to)	以 List 形式返回此集合的部分元素,元素范围为[from,to)。如果 from 和 to 相等,则返回的 List 为空
int size()	返回此集合的大小
void clear()	清除此集合的所有元素
int indexOf(Object obj)	返回此集合中首次出现 obj 的索引值
Iterator iterator()	返回此集合的迭代器
Object[] toArray()	将集合转换为数组形式返回

下面分别简要介绍 List 接口的两个实现类 ArrayList 类和 LinkedList 类。

1. ArrayList 类

ArrayList 类封装了一个可动态改变大小的 Object 类型的数组,每个 ArrayList 都有一个表示其自身容量的数值,从某种意义上讲,ArrayList 就是一种特殊的数组,但是效率没有数组高,ArrayList 可以添加、删除和修改元素,并且其大小可动态改变。除了表 5.27 中的一些方法,ArrayList 还提供了一些方法,详见表 5.28。

表 5.28 ArrayList 类的主要方法

方 法 名	说 明
void ensureCapacity(int minCapacity)	将此 ArrayList 对象的容量增加 minCapacity
void trimToSize()	将此 ArrayList 对象的容量调整为列表的当前实际大小

例 5.17 是一个有关 ArrayList 的实例。

【例 5.17】 Example5_17.java

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;
public class Example5_17 {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        //添加元素
        list.add("zero");
        list.add("one");
        list.add("two");
```

```
        list.add("three");
        //集合中可以添加任意类型的元素
        list.add(4);
        //可添加重复元素,在索引为 3 的位置插入
        list.add(3,"one");
        //直接打印集合 list 中的元素
        System.out.println("集合中的元素: " + list );
        int pos = list.indexOf("one");
        System.out.println("one 首次出现的位置: " + pos);
        System.out.println("集合的元素个数: " + list.size());
        System.out.println("索引值为 2 的元素: " + list.get(2));
        System.out.println(" ============ ");
        //使用 for 循环遍历集合
        for(int i = 0;i < list.size();i++){
            System.out.print(list.get(i) + "  ");
        }
        System.out.println("\n ============ ");
        //使用 foreach 遍历集合
        for(Object obj: list)
            System.out.print(obj + "  ");
        System.out.println("\n ============ ");
        //使用 Iterator 遍历
        Iterator iterator = list.iterator();
        while (iterator.hasNext()){
            System.out.print(iterator.next() + "  ");
        }
        System.out.println("\n ============ ");
        //使用 ListIterator 遍历
        ListIterator listIterator = list.listIterator(6);
        while (listIterator.hasPrevious()){
            System.out.print(listIterator.previous() + "  ");
        }
    }
}
```

运行结果:

```
集合中的元素: [zero, one, two, one, three, 4]
one 首次出现的位置: 1
集合的元素个数: 6
索引值为 2 的元素: two
 ============
zero  one  two  one  three  4
 ============
zero  one  two  one  three  4
 ============
zero  one  two  one  three  4
 ============
4  three  one  two  one  zero
```

通过本例可以验证:集合中可以存放任意类型的元素,集合长度也是变长的。当元素存入集合后,元素的数据类型就转换为 Object 类型;同理,从集合中取出元素时,其数据类型仍然为 Object 类型。因此,如果元素取出后恢复原有类型,必须使用强制类型转换。

注意:从集合中取出元素并进行强制类型转换将给程序带来安全隐患,在集合对象实例中使用泛型是一种有效解决方案,例如:

```
ArrayList<String>  list = new ArrayList<String>();
```

那么，集合 list 只能存放 String 类型的元素，其他数据类型的元素添加到该集合时将报错。从该集合中取出的元素，其数据类型是 String 类型，而不再是 Object 类型。

例 5.17 提供了四种遍历集合的方法，分别是使用 for 循环、foreach、Iterator 和 ListIterator。特别是 ListIterator 还提供了从前向后遍历(使用 hasNext()和 next()方法)和从后向前遍历(使用 hasPrevious()和 previous()方法)。

注意：ArrayList 提供的 listIterator()方法进行了重载，以支持以下两种遍历方式。

- listIterator()：默认从索引值为 0 的位置开始遍历，此时调用 hasPrevious()方法返回 false，即不能向前遍历，只能向后遍历。
- listIterator(int index)：从指定索引处遍历，若 index 小于集合长度，可以向前或向后遍历。

使用 Iterator 迭代器遍历集合并删除其中元素时，一定要使用 Iterator 对象的 remove()方法删除指定元素，而不能使用集合的 remove()方法删除，如例 5.18 所示。

【例 5.18】 Example5_18.java

```
import java.util.ArrayList;
import java.util.Iterator;
public class Example5_18 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("one");
        list.add("two");
        list.add("three");
        list.add("four");
        list.add("five");
        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()){
            String value = iterator.next();
            if("three".equals(value))
                list.remove("three");
            System.out.print(value + "  ");
        }
        System.out.println();
        System.out.println("删除后的结果: " + list);    }
}
```

该程序执行时产生 ConcurrentModificationException，即并发修改异常，出现异常的原因是集合在迭代器运行期间删除了元素，导致迭代器预期的迭代次数发生改变，从而使迭代器的结果不正确。因此，程序中的 list.remove("three")；这一行代码需要进行修改。如果需要在集合的迭代期间对集合中的元素进行删除，可以使用迭代器 Iterator 自身提供的 remove()方法进行删除，将例 5.18 的该行程序替换为下行代码，即可解决该问题。

```
iterator.remove();
```

运行结果：

```
one  two  three  four  five
删除后的结果: [one, two, four, five]
```

2. LinkedList 类

由于 LinkedList 类充当了动态数组,并且在创建它时不必像 ArrayList 那样指定其大小,因此当动态添加或删除元素时,LinkedList 集合的大小将自动调整。而且,其内部元素不是以连续的方式存储的。LinkedList 实现了一个双向链表结构(如图 5.3 所示),链表中的每一个元素都使用引用的方式来记住它的前一个元素和后一个元素,从而将所有的元素彼此连接起来。当前插入一个新元素时,只需要修改元素之间的这种引用关系即可,同理,删除一个元素也是如此。因此 LinkedList 提供了一些处理集合首尾两端元素的方法。使用 LinkedList 集合进行元素的增加或删除操作时效率很高。

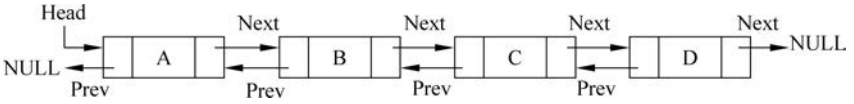

图 5.3 双向链表结构

表 5.29 给出了 LinkedList 类的常用方法。

表 5.29 LinkedList 类的常用方法

方 法 名	说 明
void addFirst(E o)	将 o 插入此集合的开头
void addLast(E o)	将 o 追加到此集合的结尾
E removeFirst()	删除并返回此集合的第一个元素
E removeLast()	删除并返回此集合的最后一个元素
E peek()	找到但不删除此集合的头
E poll()	找到并删除此集合的头
boolean offer(E o)	将指定元素添加到此集合的末尾

注意:以下情况建议使用 ArrayList。

- 频繁访问列表中的某一个元素;
- 只需在列表末尾进行添加和删除元素操作。

以下情况建议使用 LinkedList。

- 频繁地在列表首部、中间和末尾等位置进行添加和删除元素操作;
- 需要通过循环迭代来访问集合中的某些元素。

总之,与 ArrayList 相比,LinkedList 的增加和删除的操作效率更高,而查找和修改的操作效率较低。

下面是一个 LinkedList 类的实例。

【例 5.19】 Example5_19.java

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
public class Example5_19 {
    public static void main(String[] args) {
        LinkedList student = new LinkedList();
        //向 student 的尾部追加 Jimmy
        student.add("Jimmy");
        //向 student 的尾部追加 Eric
        student.offer("Eric");
```

```
        student.offer("Tom");
        //向 student 的头部追加 John
        student.addFirst("John");
        Iterator it = student.iterator();
        //遍历 student
        System.out.print("LinkedList 的所有元素: ");
        while(it.hasNext()) {
            System.out.print(it.next() + " ");
        }
        System.out.println();
        //打印第一个元素
        System.out.println("第 1 个元素: " + student.getFirst());
        System.out.println("peekLast 后的最后 1 个元素: " + student.peekLast());
        //访问并不删除第一个元素
        System.out.println("peekFirst 第 1 个元素: " + student.peekFirst());
        //访问并删除第一个元素
        System.out.println("poll 第 1 个元素: " + student.poll());
        System.out.println("删除后的所有元素为: " + student);
        ArrayList list = new ArrayList();
        long s1 = System.currentTimeMillis();
        for(int i = 0;i < 100000;i++){
            student.add(0,i);
        }
        long s2 = System.currentTimeMillis();
        System.out.println("LinkedList 执行插入元素消耗时间: " + (s2 - s1));
        long e1 = System.currentTimeMillis();
        for(int i = 0;i < 100000;i++){
            list.add(0,i);
        }
        long e2 = System.currentTimeMillis();
        System.out.println("ArrayList 执行插入元素消耗时间: " + (e2 - e1));
    }
}
```

运行结果:

```
LinkedList 的所有元素: John Jimmy Eric Tom
第 1 个元素: John
peekLast 后的最后 1 个元素: Tom
peekFirst 第 1 个元素: John
poll 第 1 个元素: John
删除后的所有元素为: [Jimmy, Eric, Tom]
LinkedList 执行插入元素消耗时间: 37
ArrayList 执行插入元素消耗时间: 1829
```

通过本例可以看到,批量插入多个元素时,LinkedList 要比 ArrayList 效率高出很多。同时,对于 peekFirst()方法和 poll()方法的区别,peekFirst()方法是获取 LinkedList 集合中的第一个元素;而 poll()方法也是获取 LinkedList 集合中的第一个元素,并且在集合中删除该元素。

注意:在实际应用中,究竟选取哪种 List 类型的集合?

实现 List 接口的实现类有两个:ArrayList 和 LinkedList。选取哪一种取决于特定的需要。如果要支持随机访问,而不必在除尾部的任何位置插入或删除元素,那么选用 ArrayList;如果要频繁地从列表的中间位置插入或删除元素,并且只要求以顺序的方式访问列表元素,那么最好选择 LinkedList。

5.6.5 Set接口

Set接口也继承自Collection接口,但与List接口相比,Set接口不允许集合中存在重复项,每个具体的Set实现类依赖添加对象的equals()方法来检查其唯一性。Set接口继承了Collection接口中的方法,没有添加新的方法,在此就不再重复列出Set接口的方法。Set接口可以分为两种类型:一种是使元素自动保持升序的集合SortedSet接口;另一种是HashSet类及其子类LinkedHashSet。

1. HashSet类及LinkedHashSet类

实现Set接口的实现类有HashSet类和TreeSet类(TreeSet同时也实现了SortedSet接口),以及HashSet类的子类LinkedHashSet。需要注意的是,HashSet集合中的元素先后顺序并不是固定不变的。一般来说,基于效率方面的考虑,添加到HashSet中的对象需要采用恰当分配哈希码的方式对Object类的hashCode()方法进行重写。

1) HashSet类

HashSet类主要用来快速查找集合中的元素,HashSet基于对象的散列值来确定元素在集合中的存储位置,因而具有较好的存取和查找性能。又由于Set集合中的元素不能有重复值,因此,存入HashSet的元素必须重写hashCode()方法,以此验证元素是否为重复元素。表5.30列出HashSet类的构造方法和常用方法。

表5.30 HashSet类的构造方法和常用方法

方法名	说明
HashSet()	构造一个新的空集合,默认初始容量为16,加载因子为0.75
HashSet(Collection c)	构造一个包含c中的元素的新集合
HashSet(int capacity,float Factor)	构造一个空集合,初始容量为capacity,加载因子为factor
HashSet(int capacity)	构造一个空集合,初始容量为capacity,加载因子为0.75
boolean add(E o)	如果此集合中还不包含o,则添加指定元素
boolean contains(Object o)	如果此集合包含o,则返回true
int size()	返回此集合中的元素的数量(集合的容量)

HashSet集合之所以能够确保不出现重复元素,是因为它在存入元素时做了一些计算与判断。当调用其add()方法存入元素时,首先调用当前存入元素的hashCode()方法获得对象的散列值,然后根据对象的散列值计算出存储位置。如果该存储位置上没有元素,则直接将元素存入。如果该存储位置上有元素存在,则会调用该元素的equals()方法比较将要存入的对象,如果返回值为false,则存入;否则说明二者值重复,将要存入的对象舍弃。HashSet存入元素的工作流程如图5.4所示。

下面通过一个例子介绍HashSet类的基本应用,了解一些常用方法的使用及其集合的遍历方法。

【例5.20】 Example5_20.java

```
import java.util.HashSet;
import java.util.Iterator;
public class Example5_20 {
    public static void main(String[] args) {
        HashSet set = new HashSet();
```

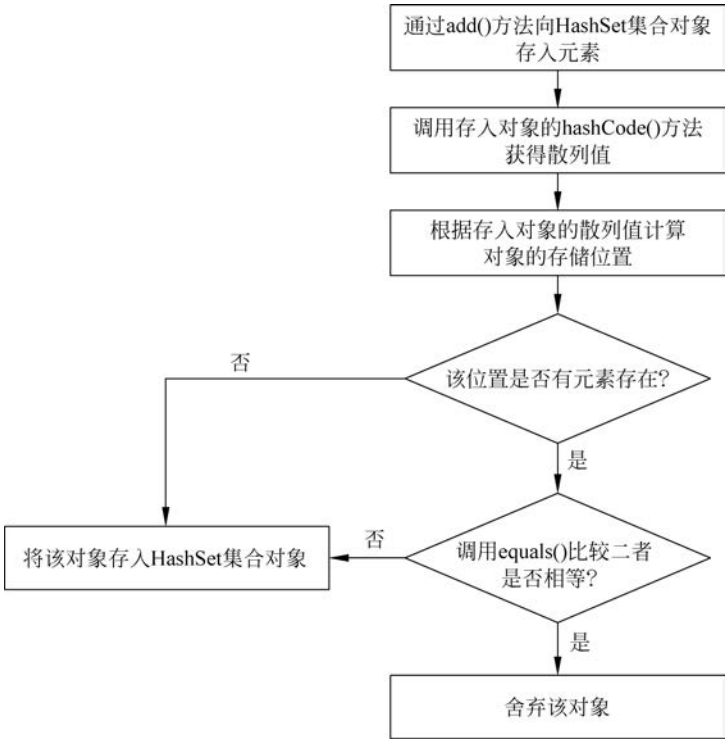

图 5.4 HashSet 存入元素的工作流程

```
        //添加下面的 6 个元素,注意有重复值
        set.add("white");
        set.add("red");
        set.add("blue");
        set.add("pink");
        set.add("orange");
        set.add("green");
        //添加重复的元素
        set.add(new String("blue"));
        System.out.println(set);
        System.out.println("容量: " + set.size());
        System.out.println("包含 red 吗?" + set.contains("red"));
        //遍历 Set 集合
        Iterator it = set.iterator();
        while (it.hasNext())
            System.out.print(it.next() + " ");
    }
}
```

运行结果:

```
[red, orange, pink, green, white, blue]
容量: 6
包含 red 吗?true
red orange pink green white blue
```

通过本例可以看到,向 HashSet 集合中添加 String 类型的元素,其重复值的元素如 blue 不再重复添加。原因在于 String 类重写了 hashCode()和 equals()方法,对于字符串常

量和创建 String 对象,若字符串内容相等,二者的散列值也是相等的,两个字符串也相等。因此,字符串"blue"不能重复存入 HashSet 集合。

那么,HashSet 集合存入其他引用类型的对象时,如何判断元素是否为重复项呢?这就要求存入的引用类型必须重写 hashCode()与 equals()方法。以 Student 类为例,假设将若干 Student 对象存入 HashSet 集合中,如果 Student 对象的学号与姓名均一致,则认为是相等的,下面通过一个程序看看能否存入相同的学生对象。

【例 5.21】 Example5_21.java

```
import java.util.HashSet;
public class Example5_21 {
    public static void main(String[] args) {
        HashSet<Student> set = new HashSet<Student>();
        Student s1 = new Student("201506899","zhangsan");
        Student s2 = new Student("201506890","lisi");
        Student s3 = new Student("201506893","wangwu");
        Student s4 = new Student("201506899","zhangsan");
        System.out.println("s1 hashCode:" + s1.hashCode());
        System.out.println("s4 hashCode:" + s4.hashCode());
        set.add(s1);
        set.add(s2);
        set.add(s3);
        set.add(s4);
        System.out.println(set);
    }
}

class Student{
    String no;
    String name;
    public Student(String no, String name){
        this.no = no;
        this.name = name;
    }
    @Override
    public String toString(){
        return "{No. " + no + ", Name: " + name + "}";
    }
}
```

运行结果:

```
s1 hashCode:1078694789
s4 hashCode:931919113
[{No. 201506899, Name: zhangsan}, {No. 201506893, Name: wangwu}, {No. 201506899, Name: zhangsan},
{No. 201506890, Name: lisi}]
```

本例的运行结果表明对于学号和姓名均相同的两个 Student 对象 s1 和 s4,却仍然能够存入 HashSet 集合中。s1 和 s4 的散列值表明了二者不相同,因此 HashSet 集合能够同时存入 s1 和 s4 对象。针对该问题,如何解决呢?解决方案是重写 Student 类的 hashCode()和 equals()方法。这里给出完善 Student 类的代码,主类无须做修改。

```
class Student{
```

```
    String no;
    String name;
    public Student(String no, String name){
        this.no = no;
        this.name = name;
    }
    @Override
    public String toString(){
        return "{No. " + no + ", Name: " + name + "}";
    }
    //重写 hashCode()方法,返回学号与姓名的散列值之差
    @Override
    public int hashCode() {
        return this.no.hashCode() - this.name.hashCode();
    }
    //重写 equals()方法,如果学号 no 和姓名 name 均相等,返回 true
    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Student){
            if(this.no.equals(((Student) obj).no)
                    && this.name.equals(((Student) obj).name))
                return true;
        }
        return false;
    }
}
```

在 Student 类中,重写了 hashCode()与 equals()方法。hashCode()方法返回的是学号与姓名的散列值之差;equals()方法则比较学号和姓名依次相等时返回 true。重新运行例 5.21,HashSet 集合中将不会存储重复的学生对象了。

2) LinkedHashSet 类

LinkedHashSet 类扩展了 HashSet 类,LinkedHashSet 类增加了跟踪添加到 HashSet 中的元素顺序的功能。LinkedHashSet 的迭代器按照元素的插入顺序来访问各个元素,它提供了一个可以快速访问各个元素的有序集合。当然也增加了实现的代价,因为哈希表元中的各个元素是通过双重链接式列表链接在一起的。LinkedHashSet 类的主要方法见表 5.31。

表 5.31 LinkedHashSet 类的主要方法

方 法 名	说 明
LinkedHashSet()	构造一个空集合,默认初始容量为 16,加载因子为 0.75
LinkedHashSet(Collection c)	构造一个包含 c 中的元素的集合
LinkedHashSet(int Capacity,float Factor)	构造一个初始容量为 Capacity,加载因子为 Factor 的空集合
LinkedHashSet(int Capacity)	构造一个初始容量为 Capacity,加载因子为 0.75 的空集合

【例 5.22】 Example5_22.java

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

public class Example5_22 {
```

```
    public static Set fill(Set s, int size) {
        for (int i = 0; i < size; i++) {
            s.add(new MySet(i));
        }
        return s;
    }
    public static void set(Set c) {
        fill(c,6);
        c.addAll(fill(new TreeSet(),6));
        System.out.println(c);
    }
    public static void main(String[] args) {
        System.out.print("HashSet:");
        set(new HashSet());
        System.out.print("LinkedHashSet:");
        set(new LinkedHashSet());
    }
}
class MySet implements Comparable {
    private int s;
    public MySet(int s) {
        this.s = s;
    }
    public boolean equals(Object obj){
        return (obj instanceof MySet) && (s == ((MySet)obj).s);
    }
    public String toString() {
        return s + " ";
    }
    public int hashCode() {
        return s;
    }
    public int compareTo(Object obj) {
        int t = ((MySet)obj).s;
        return (t < s ? -1:(t == s ? 0 : 1));
    }
}
```

运行结果：

```
HashSet:[0 , 1 , 2 , 3 , 4 , 5 ]
LinkedHashSet:[0 , 1 , 2 , 3 , 4 , 5 ]
```

2. SortedSet 接口及 TreeSet 类

SortedSet 接口能够使集合保持其元素为升序顺序，将元素添加到 SortedSet 接口的实现类 TreeSet 的实例中，无论元素添加的先后顺序如何，在集合对象中总是使这些元素保持为升序顺序。TreeSet 是 SortedSet 接口的唯一实现类。

TreeSet 基本上是一个自平衡二叉搜索树(如红黑树)的实现。因此，像添加、删除和搜索这样的操作需要 $O(\log N)$时间。原因是在自平衡树中，确保所有操作的树高度始终为 $O(\log N)$。因此，这被认为是存储海量排序数据并对其执行操作的最有效的数据结构之一。但是，像按排序顺序打印 N 个元素这样的操作，其时间复杂度为 $O(N)$。

下面简要介绍一下 TreeSet 的主要方法，详见表 5.32。

表 5.32 TreeSet 的主要方法

方 法 名	说 明
Comparator comparator()	返回此集合的 Comparator,或者返回 null,表示以自然方式排序
E first()	返回此集合的第一个元素
E last()	返回此集合的最后一个元素
SortedSet headSet(E toElement)	返回此集合的子集,由小于 toElement 的元素组成
SortedSet tailSet(E fromElement)	返回此集合的子集,由大于或等于 fromElement 的元素组成
SortedSet subSet(E from,E to)	返回此集合的子集,范围为[from, to]
E lower(E e)	返回小于给定元素值 e 的最近元素,如不存在,返回 null
E higher(E e)	返回大于给定元素值 e 的最近元素,如不存在,返回 null
E floor(E e)	返回小于或等于给定元素值 e 的最近元素,如不存在,返回 null
E ceiling(E e)	返回大于或等于给定元素值 e 的最近元素,如不存在,返回 null

【例 5.23】 Example5_23.java

```
import java.util.SortedSet;
import java.util.TreeSet;
public class Example5_23 {
    public static void main(String[] args) {
        //创建一个 SortedSet 集合 set
        SortedSet set =  new TreeSet();
        //添加下面的 6 个元素,注意有重复值
        set.add("white");
        set.add("red");
        set.add("blue");
        set.add("orange");
        set.add("green");
        set.add(new String("blue"));
        //输出集合
        System.out.println("集合的内容: " + set);
        System.out.println("第一个元素: " + set.first());
        System.out.println("最后一个元素: " + set.last());
        System.out.println("subSet(1,3): " + set.subSet("green","white"));
        set.remove("orange");
        System.out.println("删除 orange 元素后: " + set);
    }
}
```

运行结果:

```
集合的内容: [blue, green, orange, red, white]
第一个元素: blue
最后一个元素: white
subSet(1,3): [green, orange, red]
删除 orange 元素后: [blue, green, red, white]
```

从本例中可以看出,TreeSet 集合能够自动对添加的元素进行升序排序,这是因为添加的元素类型均为 String 类型,String 类实现了 Comparable 接口。Comparable 接口强行对实现它的每个类的对象进行整体排序,Comparable 接口的 compareTo(Object obj)方法是实现排序的核心方法。

注意:基本数据类型对应的包装类、String 类都实现了 Comparable 接口,这意味着向

TreeSet 集合中添加这些类型的元素时,TreeSet 集合将自动以这些类型的元素按自然排序的方式进行排序。

假设向 TreeSet 集合中添加其他引用类型的元素,那么这些类型的元素必须实现 Comparable 接口,并重写 compareTo(Object obj)方法定义该引用类型的排序方式,否则 TreeSet 集合无法对这些对象进行排序,从而产生 ClassCastException 异常。如例 5.24 所示,向 TreeSet 集合中添加 Student 对象。

【例 5.24】 Example5_24.java

```
import java.util.TreeSet;
public class Example5_24 {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<Integer>();
        set.add(23);
        set.add(56);
        set.add(-96);
        set.add(863);
        set.add(56);
        System.out.println(set);
    //向 TreeSet 集合中添加 Student 对象
        TreeSet<Student> ts = new TreeSet<Student>();
        Student s1 = new Student("201805323","zhangsan");
        Student s2 = new Student("201805324","lisi");
        Student s3 = new Student("201805325","wangwu");
        Student s4 = new Student("201805323","zhangsan");
        ts.add(s1);
        ts.add(s2);
        ts.add(s3);
        ts.add(s4);
        System.out.println(ts);
    }
}
```

如果仍然以例 5.21 中的 Student 类创建学生对象,并添加到 TreeSet 集合。程序运行时可以 Integer 类型的元素进行排序,但添加 Student 对象时将产生异常。因此,需要对 Student 类进一步完善,让 Student 类实现 Comparable 接口并重写 compareTo()方法。改进后的 Student 类如下。

```
class Student implements Comparable<Student>{
    String no;
    String name;
    public Student(String no, String name){
        this.no = no;
        this.name = name;
    }
    @Override
    public String toString(){
        return "{No. " + no + ", Name: " + name + "}";
    }
    //重写 hashCode()方法,返回学号与姓名的散列值之差
    @Override
    public int hashCode() {
        return this.no.hashCode() - this.name.hashCode();
```

```
    }
    //重写 equals()方法,如果学号 no 和姓名 name 均相等,返回 true
    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Students){
            if(this.no.equals(((Students) obj).no)
                    && this.name.equals(((Students) obj).name))
                return true;
        }
        return false;
    }
    //重写 compareTo()方法,比较学号 no 是否相等,如相等再比较姓名 name
    @Override
    public int compareTo(Student o) {
        int num = this.no.compareTo(o.no);
        return num == 0 ? this.name.compareTo(o.name) : num;
    }
}
```

重新运行例 5.24,此时程序可以正常运行,执行结果如下。可以看到无论是 Integer 类型的元素,还是 Student 类型的元素,均实现了排序且没有重复元素添加到集合中。

```
[-96, 23, 56, 863]
[{No. 201805323, Name: zhangsan}, {No. 201805324, Name: lisi}, {No. 201805325, Name: wangwu}]
```

5.6.6 Map 接口

Map 用于保存具有映射关系的数据,因此 Map 中每个元素由键(Key)和值(Value)两列数据组成,键和值是一对一的映射关系。Map 就像数据库中的数据表,而键就像数据表的主键,通过唯一的键就能找到相对应的值。Map 接口的常用方法如表 5.33 所示。

表 5.33 Map 接口的常用方法

方 法 名	说 明
void clear()	从此 Map 中移除所有映射关系
boolean containsKey(Object key)	如果此 Map 包含 key,则返回 true
boolean containsValue(Object value)	如果此 Map 包含指定 value 映射到一个或多个键,则返回 true
V get(Object key)	返回此 Map 中映射到键为 key 的值
V put(K key,V value)	将值 value 与此 Map 中的键 key 相关联
V remove(Object key)	如果存在键值为 key 的映射关系,则将其从映射中移除
void putAll(Map K)	把指定 Map 中的所有映射关系复制到此映射中
Set keySet()	返回此 Map 中所有键值组成的 Set 集合
Collection values()	返回此 Map 里所有值即 Value 组成的 Collection 集合
Set entrySet()	返回此 Map 中包含的键值对所组成的 Set 集合
boolean equals(Object o)	比较指定的对象与此 Map 是否相等

注意:Map 中键和值都可以为 null,但是键必须是唯一的,使用 put()方法添加一对映射关系时,如果映射关系中的键已经存在,那么与此键相关的新值将取代旧值。

1. HashMap 类及其 LinkedHashMap 类

HashMap 是 Map 接口的典型实现类,HashMap 自 JDK 1.2 以来成为 Java 集合的一部分,

该类位于 java.util 包中。它提供了 Java 映射接口的基本实现。它以“键-值”对的形式存储数据,我们可以通过键找到其对应的值,如果尝试插入重复键,它将替换相应键的元素。

HashMap 类似于 HashTable,但它是不同步的。它也允许存储空键,但是应该只有一个空键对象,但可以有任意数量的空值。此类不保证映射的顺序,要使用此类及其方法,需要导入 java.util.HashMap 包或其超类。

【例 5.25】 Example5_25.java

```
import java.util.*;
public class Example5_25 {
    public static void main(String[] args) {
        HashMap<String,String> map = new HashMap<String,String>();
        map.put("2","lisi");
        map.put("1","zhangsan");
        map.put("3","wangwu");
        //添加一个重复 key,相当于修改 key 对应的 value
        map.put("3","liming");
        System.out.println(map);
        //keySet()方式遍历 map
        Set<String> keyset = map.keySet();
        Iterator<String> it1 = keyset.iterator();
        while (it1.hasNext()){
            String key = it1.next();
            String value = map.get(key);
            System.out.println(key + "-->" + value);
        }
        System.out.println("============");
        //entrySet()方式遍历 map,迭代成员为 Map.Entry
        Set<Map.Entry<String,String>> entryset = map.entrySet();
        Iterator<Map.Entry<String,String>> it2 = entryset.iterator();
        while (it2.hasNext()){
            Map.Entry<String,String> entry = it2.next();
            String key = entry.getKey();
            String value = entry.getValue();
            System.out.println(key + "-->" + value);
        }
        System.out.println("============");
        //遍历值
        Collection<String> collection = map.values();
        for (String value:collection){
            System.out.println(value);
        }
        System.out.println("============");
        System.out.println(map.containsValue("zhangsan"));
        System.out.println(map.containsKey(6));
    }
}
```

运行结果:

```
{1=zhangsan, 2=lisi, 3=liming}
1-->zhangsan
2-->lisi
3-->liming
```

```
============
1-->zhangsan
2-->lisi
3-->liming
============
zhangsan
lisi
liming
============
true
false
```

本例先后使用 keySet()和 entrySet()两个方法返回的 Iterator 对象遍历 HashMap 对象。keySet()方法返回 HashMap 集合中的键并生成一个 Set 对象,然后使用迭代器遍历该 Set 对象; entrySet()方法将 HashMap 每个"键-值"对作为整体封装成 Map.Entry 对象,转存至 Set 集合中,然后使用迭代器遍历。

通过本例可以发现遍历 HashMap 集合迭代出来的元素顺序与存入的顺序是不一致的。如果想让这两个顺序一致,可以使用 LinkedHashMap 类,它是 HashMap 的子类,与 LinkedList 一样,它也使用双向链表来维护内部元素的关系,使 Map 元素迭代的顺序与存入的顺序一致。

如果在 Map 中插入、删除和定位元素,HashMap 是最佳选择。需要注意的是,使用 HashMap 类时要求添加的键(Key)要明确重写 hashCode()和 equals()两个方法。此外,HashMap 类的构造方法中有两个重要参数初始容量(initialCapactiy)和负载因子(loadFactor)。容量是指哈希表中桶(bucket)的数量,而初始容量只是哈希表在创建时的容量。负载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度,负载因子为 0 时表示为空的哈希表,为 0.5 时表示半满的哈希表。负载因子的默认值为 0.75,以寻求在时间和空间上达到折中,如果负载因子过高,虽然减少了空间开销,但同时也增加了查询成本。

2. SortedMap 接口及其实现类 TreeMap

SortedSet 接口有一个实现类 TreeSet,与之相似,SortedMap 接口也有一个实现类 TreeMap。同理,TreeMap 对该映射关系中的所有键进行排序,从而保证 TreeMap 中所有的映射关系保持有序状态。特别注意,添加到 SortedMap 对象的映射关系必须实现 Comparable 接口,否则必须给它的构造方法提供一个 Comparator 接口的实现。表 5.34 列出了 SortedMap 接口的常用方法。

表 5.34 SortedMap 接口的常用方法

方 法 名	说 明
Comparator comparator()	返回对关键字排序时使用的比较器
Object firstKey()	返回映射关系中第一个键
Object lastKey()	返回映射关系中最后一个键
SortedMap subMap(Object beginKey, Object endKey)	返回(beginKey,endKey)范围内的 SortedMap 子集
SortedMap headMap(Object endKey)	返回 SortedMap 的一个视图,其内各元素的键都小于 endKey
SortedMap tailMap(Object beginKey)	返回 SortedMap 的一个视图,其内各元素的键都大于或等于 beginKey
Collection values()	以 Collection 形式返回此映射包含的值

TreeMap 是 Java 集合框架的另一个重要成员,该类实现了 Map 接口、NavigableMap 接口和 SortedMap 接口,同时扩展了 AbstractMap 类。TreeMap 不允许键为 null,试图将 null 作为键存入 TreeMap 集合时会引发 NullPointerException 异常,但是值为 null 却不受限制。

【例 5.26】 Example5_26.java

```
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
public class Example5_26 {
    public static void main(String[] args) {
        TreeMap<Integer,String> treeMap = new TreeMap<Integer,String>();
        treeMap.put(12,"Melon");
        treeMap.put(1,"Tom");
        treeMap.put(41,"Eric");
        //键重复,键为 12 对应的值将更新
        treeMap.put(12,"Megan");
        //TreeMap 的键不能为 null
        //treeMap.put(null,"none");
        System.out.println(treeMap);
        //遍历 TreeMap 集合
        Set set = treeMap.entrySet();
        Iterator<Map.Entry<Integer,String>> iterator = set.iterator();
        while (iterator.hasNext()){
            Map.Entry<Integer,String> entry = iterator.next();
            System.out.println(entry.getKey() + ":" + entry.getValue());
        }
    }
}
```

运行结果:

```
{1=Tom, 12=Megan, 41=Eric}
1:Tom
12:Megan
41:Eric
```

在本例中,使用泛型对 TreeMap 进行了约束,键为 Integer 类型,值为 String 类型。由于 Integer 实现了 Comparable 接口,因此 TreeMap 集合可对键进行自然排序。同时也使用了 entrySet()方法,返回一个 Set 集合对象,进而使用迭代器对 TreeMap 集合进行遍历。

TreeMap 集合也可以对添加的元素的键进行排序,其实现与 TreeSet 一样。TreeMap 排序分为自然排序和比较排序。可以让添加的元素实现 Comparable 接口实现自然排序,或者让 TreeMap 对象实现 Comparator 接口实现比较排序。仍以 Student 类为例,前面对 Student 类实现了 Comparable 接口,因此当 Student 对象作为键存入 TreeMap 集合中时,将自动按 Student 的自然排序方式对集合中的键-值对进行排序。

【例 5.27】 Example5_27.java

```
import java.util.Comparator;
import java.util.TreeMap;
public class Example5_27 {
```

```
    public static void main(String[] args) {
        //使用匿名内部类定义排序规则：按姓名排序,如果姓名相同,再按年龄升序排列
        TreeMap treeMap = new TreeMap(new Comparator<Student>() {
            @Override
            public int compare(Student o1, Student o2) {
                int num = o1.getName().compareTo(o2.getName());
                return num == 0 ? o1.getAge() - o2.getAge():num;
            }
        });
        Student s1 = new Student("zhangsan",20);
        Student s2 = new Student("lisi",21);
        Student s3 = new Student("wangwu",19);
        Student s4 = new Student("zhangsan",18);
        treeMap.put(s1,"Kaifeng");
        treeMap.put(s2,"Zhengzhou");
        treeMap.put(s3,"Luoyang");
        treeMap.put(s4,"Xinxiang");
        System.out.println(treeMap);
    }
}

class Student {//implements Comparable<Student>{
    private String name;
    private int age;
    public Student(String name,int age){
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return  "[" + this.getName() + ":" + this.age + "]";
    }
    //排序规则：先按学生姓名升序排序,如果姓名相同,再按年龄升序排序
//    @Override
//    public int compareTo(Student o) {
//        int num = 0;
//        num = this.getName().compareTo(o.getName());
//        if(num == 0)
//            return this.getAge() - o.getAge();
//        else
//            return num;
//    }
}
```

运行结果：

```
{[lisi:21] = zhengzhou, [wangwu:19] = luoyang, [zhangsan:18] = xinxiang, [zhangsan:20] = kaifeng}
```

本例的 Student 类包括两个属性 name 和 age，在 compareTo(Object obj)方法中设置了排序规则：先按学生姓名 name 升序排序，如果 name 相同，再按年龄 age 升序排序。另外一种比较排序的方式是让 TreeMap 对象实现 Comparator 接口并重写 compare(Object o1, Object o2)方法。本例中的代码在比较排序的方式实现，在 TreeMap 实例化时以匿名内部类的方式实现了 Comparator 接口。倘若使用自然排序的方式，可以将匿名内部类的代码注释掉，将 Student 类代码的注释部分加上即可。

注意：TreeMap 集合对键对象排序的方式有两种：自然排序和比较排序。

- 自然排序是将作为键的引用类型实现 Comparable 接口，并重写 compareTo(Object obj)方法。以 Student 类为例，该类实现 Comparable 接口并重写相应方法。
- 比较排序是将 TreeMap 实现 Comparator 接口，并重写 compare(Object o1, Object o2)方法。一般在 TreeMap 对象实例化，以匿名内部类的方式实现。

3. Properties 类

java.util.Properties 类称为属性列表，继承于 Hashtable 类。Hashtable 类与 HashMap 比较相似，区别在于 Hashtable 是线程安全的，Hashtable 存取元素时速度较慢，目前已基本被 HashMap 所取代。Properties 类表示一个持久的属性列表，属性列表中的每个键及其对应值是一个字符串。Properties 类被许多 Java 类使用，例如前面介绍的 System 类的 getProperties()方法，其返回值就是 Properties 类型。

同时，Properties 类还提供 load()和 list()方法，可以读写磁盘上的资源文件(.properties)。在实际开发中，经常使用 Properties 对象来存取应用的配置项。表 5.35 列出 Properties 类的常用方法。

表 5.35 Properties 类的常用方法

方 法 名	说 明
Properties()	构造一个空属性列表
String getProperty(String key)	用指定的键在此属性列表中搜索属性
void list(PrintStream streamOut)	将属性列表输出到指定的输出流
void load(InputStream streamIn) throws IOException	从输入流中读取属性列表
Enumeration propertyNames()	返回属性列表中所有键的枚举
Object setProperty(String key,String value)	调用 Hashtable 的 put()方法

【例 5.28】 Example5_28.java

```
import java.io.*;
import java.util.Enumeration;
import java.util.Properties;
public class Example5_28 {
    public static void main(String[] args) {
        Properties p = new Properties();
        p.put("user","root");
        p.put("password","secret");
        p.put("url","jdbc:mysql//localhost:3306/db");
        p.put("driverClassName","com.mysql.jdbc.Driver");
```

```
        //将属性列表保存到磁盘上某个文件
        try{
            PrintStream ps = new PrintStream("c:/file.properties");
            p.list(ps);
            ps.close();
        }catch (IOException e){
            e.printStackTrace();
        }
        p.clear();
        //将磁盘上资源文件读取出来,加载到 Properties 对象中
        try{
            FileReader fr = new FileReader("c:/file.properties");
            p.load(fr);
            fr.close();
        }catch (IOException e){
            e.printStackTrace();
        }
        //遍历 Properties 对象
        Enumeration en = p.propertyNames();
        while (en.hasMoreElements()){
            Object obj = en.nextElement();
            System.out.println(obj + " = " + p.get(obj));
        }
    }
}
```

运行结果：

```
user = root
url = jdbc:mysql//localhost:3306/db
password = secret
driverClassName = com.mysql.jdbc.Driver
```

5.6.7 数组与容器的区别

数组和集合的差异，主要从以下几方面进行对比。

效率：数组是一种高效的存储和访问元素的数据类型，数组中的元素在内存中以连续方式存储，通过“数组名[index]”的方式可以轻松地访问数组中的元素，但是数组一旦定义并初始化后，其长度和元素的数据类型也就固定下来不能再改变。而集合存储和访问元素需要使用专门的方法如 add()、get()方法等。因此，从效率上讲，数组要比容器类的效率高。

类型：集合不以具体的数据类型来处理对象，而是把所有的数据类型都以 Object 类型来处理，正是集合这种处理数据的机制，使集合可以存储不同数据类型的元素。另外，基本数据类型的元素是不能直接存放到容器中的，而是转换成相对应的包装类对象后再存放到容器中。而数组既可以保存基本数据类型也可以保存引用类型。

使用：任何类型的元素存入集合后，其数据类型将自动转换为 Object 类型；元素从集合取出时，仍然是 Object 类型，元素要返回原有数据类型必须进行强制类型转换，这也给程序带来了安全风险，并且效率大幅下降。正基于此，集合又引入了泛型，从而约束集合中的元素必须是某种具体类型，避免了多次类型转换。而数组的使用规则必须先声明初始化再使用，元素存入和取出都不需要进行类型转换。

综合以上对比,可以认为数组是一种“轻量级”的数据类型,使用简单方便,但是功能上略微单薄;而集合是一种“重量级”的数据类型,它提供了丰富的接口,功能强大,性能卓越,但是使用时略显“笨重”,需要对各元素进行数据类型转换。

注意:何时使用集合?在下列情形下建议使用集合而不使用数组。

- 需要处理的对象数目不定,序列中的元素都是对象或可以表示为对象;
- 需要将不同类型的对象组合成一个数据序列;
- 需要做频繁的对象序列中元素的插入和删除;
- 经常需要定位序列中的对象和其他查找操作;
- 在不同的类之间传递大量的数据;
- 需要一些特定的操作,如要求数据序列中不能有重复元素、自动排序等。

5.7 泛型

Java 泛型(generics)是 JDK 1.5 之后增加的一个特性,泛型提供了编译时类型安全检测机制,该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型,也就是说,所操作的数据类型被指定为一个参数。Java 集合几乎全部支持泛型,在前面介绍集合的例子中也使用了泛型。集合使用泛型后,相当于为集合增加了约束,由原来可以存储任意数据类型的元素,更改为只能存储指定数据类型的元素。

在定义类、方法、接口时引入泛型,能够使程序具有更好的普适性。假设有这样一个需求:定义一个排序方法,能够对整型数组、字符组数组、日期类型数组等多种数据类型的数组进行排序,根据已学习的知识,很多人员想到了方法重载。但是,方法重载的缺陷也很明显,致使程序比较臃肿。泛型也可以解决该问题,并且可以做到代码极度简洁。

泛型允许程序员在使用强类型程序设计语言编写代码时定义一些可变部分,这些可变部分可以在运行前指定具体数据类型。在编程中使用泛型代替某个实际的数据类型,而后通过实际调用时传入或推导的类型来对泛型进行替换,从而达到代码复用的目的。Java 中常见泛型标记符如下。

- E:Element,多在集合中使用,表示存放的元素;
- T:Type,表示 Java 类;
- K:Key,表示键;
- V:Value,表示值;
- N:Number,表示数值类型;
- ?:表示不确定的 Java 类型。

除了这些标记符,其实也可以是任意的字母。在使用泛型的过程中,操作数据类型被指定为一个参数,这种参数类型在类、方法、接口中,分别称为泛型类、泛型方法和泛型接口。相对于传统的形参,泛型可以使参数具有更多类型上的变化,使代码更好地复用。

5.7.1 泛型类

泛型类是在传统类声明时通过使用泛型标记符表示类中某个属性的类型,或者是某个方法的返回值或形参类型。当开发人员在实例化该类的对象时,指定泛型指代的具体类型。

泛型类的声明格式具体如下。

```
[修饰符] class 类名<泛型标识符 1, 泛型标识符 2,…>{
            [修饰符] 泛型标识符 属性名称;
            [修饰符] 泛型标识符 方法名称(泛型标识符 参数名称,…){}
}
```

泛型类定义之后,创建该类的对象,语法格式如下。

```
类名<参数化类型> 对象名称 = new 类名<参数化类型>(参数列表);
```

例如,前面创建 TreeSet 对象时,引入泛型后,实例化后若所有元素均为 String 类型,实例化方式如下。

```
TreeSet<String> treeset = new TreeSet<String>();
```

下面的代码定义了一个泛型类 Generic,该类声明时使用了两个泛型标识符 T 和 K。

Generic.java

```
class Generic<T,K>{
    private T value;
    private K key;
    Set<K> hashset = new HashSet<K>();
    //泛型方法
    public T get(){
        return this.value;
    }
    //泛型方法
    public void set(T t){
        this.value = t;
    }
    public void print(){
        System.out.println(value);
    }
    public void add(K key){
        hashset.add(key);
    }
    public void list(){
        Iterator<K> iterator = hashset.iterator();
        while(iterator.hasNext())
            System.out.print(iterator.next() + "  ");
        System.out.println();
    }
}
```

5.7.2 泛型方法

前面介绍的泛型类中已包括泛型方法,泛型方法的定义与其所在类是否为泛型类没有任何关系。泛型方法的语法格式如下。

```
[修饰符] 泛型标识符 方法名称(泛型标识符 参数名称,…){}
```

例如,下面定义的 add(K key)方法就是一个泛型方法。

```
public void add(K key){
    hashset.add(key);
}
```

定义泛型方法的规则如下。

- 所有泛型方法声明都有一个类型参数声明部分(由尖括号分隔),该类型参数声明部分在方法返回类型之前。
- 每一个类型参数声明部分包含一个或多个类型参数,参数间用逗号隔开。一个泛型参数也被称为一个类型变量,是用于指定一个泛型类型名称的标识符。
- 类型参数能被用来声明返回值类型,并且能作为泛型方法得到的实际参数类型的占位符。
- 泛型方法体的声明和其他方法一样。注意类型参数只能代表引用型类型,不能是基本数据类型(如 int、double、char 等)。

5.7.3 泛型接口

Java 集合中的接口基本上都是泛型接口,当然,开发人员也可以自定义泛型接口,声明泛型接口与声明泛型类的语法格式类似,具体格式如下。

```
[修饰符] interface 接口名称<泛型标识符 1, 泛型标识符 2,…>{}
```

例如,创建一个泛型接口 Service:

```
interface Service<E>{
    public E add(E a);
}
```

【例 5.29】 Example5_29.java

```
import org.junit.Test;
import java.util.*;
//单元测试,JUnit
public class Example5_29 {
    @Test
    public void test1(){
        //实例化时,<String,Integer>替换声明时的<T,K>
        Generic<String,Integer> generic = new Generic<>();
        generic.set("hello");
        generic.print();
        generic.add(200);
        generic.add(90);
        generic.add(236);
        generic.list();
    }
    @Test
    public void test2(){
        Service<Integer> service = new Service<Integer>() {
            @Override
            public Integer add(Integer a) {
                return a + 100;
            }
        };
        System.out.println(service.add(98));
    }
}
//泛型类
```

```
class Generic<T,K>{
    private T value;
    private K key;
    private Set<K> hashset = new HashSet<K>();
    //泛型方法
    public T get(){
        return this.value;
    }
    //泛型方法
    public void set(T t){
        this.value = t;
    }
    public void print(){
        System.out.println(value);
    }
    public void add(K key){
        hashset.add(key);
    }
    public void list(){
        Iterator<K> iterator = hashset.iterator();
        while(iterator.hasNext())
            System.out.print(iterator.next() + "  ");
        System.out.println();
    }
}
//泛型接口
interface Service<E>{
    public E add(E a);
}
```

运行结果：

```
hello
200  90  236
198
```

本例定义了一个泛型类 Generic 和一个泛型接口 Service，类和接口中又定义了若干泛型方法。在测试类的 test1()和 test2()方法中分别实例化了 Generic 对象和 Service 对象。注意，为了简化操作，本例使用了 JUnit 单元测试的方法。

5.8 Lambda 表达式

Lambda 表达式是 JDK 8 新增的特性，Lambda 表达式允许把函数作为一个方法的参数，允许创建一个不属于任何类的函数。它主要实现具有唯一方法的接口(称为函数式接口)，Lambda 表达式可以取代大部分匿名内部类，使代码变得更加简洁紧凑。

Lambda 表达式由参数列表、箭头符号(->)和函数体组成。其中，函数体既可以是一个表达式，也可以是一个语句块。Lambda 表达式的语法格式如下。

```
(parameters) -> expression;
```

或

```
(parameters) -> {statements;};
```

以下是 Lambda 表达式的重要特征。

- 可选类型声明：不需要声明参数类型，编译器可以统一识别参数值；
- 可选的参数圆括号：一个参数无须定义圆括号，但多个参数需要定义圆括号；
- 可选的大括号：如果主体包含一个语句，就不需要使用大括号；
- 可选的返回关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定表达式返回了一个数值。

【例 5.30】 Example5_30.java

```
import org.junit.Test;
public class Example5_30 {
    //使用正常的实现 implements
    @Test
    public void test1(){
        Impl i = new Impl();
        i.sayHello("World!");
    }
    //匿名内部内的形式实现
    @Test
    public void test2(){
        Greeting greeting = new Greeting() {
            @Override
            public void sayHello(String s) {
                System.out.println("Hello, " + s);
            }
        };
        greeting.sayHello("Eric!");
    }
    //使用 Lambda 表达式的形式
    @Test
    public void test3(){
        Greeting h = (s) ->{ System.out.println("Hello, " + s); };
        h.sayHello("Java!");
    }
}
//实现 Greeting 接口
class Impl implements Greeting{
    @Override
    public void sayHello(String s) {
        System.out.println("Hello, " + s);
    }
}
//函数式接口
interface Greeting{
    public void sayHello(String s);
}
```

运行结果：

```
Hello, World!
Hello, Eric!
Hello, Java!
```

本例定义了一个接口 Greeting，该接口中仅有一个方法 sayHello()方法。而 Impl 类是 Greeting 接口的实现类。在测试类 Example5_30 的三个测试方法中，test1()方法基于实现类 Impl 创建对象并执行 Greeting 的 sayHello()方法；test2()方法基于匿名内部类的方式创建 Greeting 对象并执行 sayHello()方法；test3()方法基于 Lambda 表达式执行 Greeting 接口中的 sayHello()方法。对比可知，Lambda 表达式的方式更加简洁。

注意：Lambda 表达式实现接口中的方法，对接口有明确要求，即接口必须是函数式接口(Functional Interface)。所谓函数式接口，也就是接口中仅定义了一个抽象方法。

5.9 思政案例：保护环境，从垃圾分类做起

5.9.1 案例背景

随着人们生活水平的不断提高，对生活质量的追求也愈发强烈，美丽的生态环境和绿色的生活方式自然不能缺席。同时，随着人们生活质量的提高也产生越来越多的垃圾。如果垃圾没有妥善处理，不仅污染环境，还造成资源浪费。据有关部门统计，我国每年约有 300 万吨废钢铁，600 万吨废纸没得到利用。而我们经常随手丢弃的废干电池，每年就有 60 多亿只，里面总共含有 7 万多吨锌，10 万吨二氧化锰。这些资源如果都能被重新利用，将会成为巨大的社会财富！同时，一些有毒垃圾如果没有得到正确的分类处理，会增加填埋或焚烧的垃圾量，焚烧的垃圾越多，释放的有毒气体就越多，同时还会产生有害灰尘；而地下填埋也会污染地下水和土壤，这些都对我们的健康构成了极大威胁。

因此，垃圾分类处理刻不容缓！垃圾分类是指按照垃圾的成分、属性、利用价值、对环境的影响以及现有处理方式的要求，分离不同类别的若干种类。欧美发达国家与国内城市的垃圾分类经验告诉我们：垃圾分类是垃圾进行科学处理的前提，为垃圾的减量化、资源化、无害化处理奠定基础。垃圾分类处理有以下好处。

- 将易腐有机成分为主的厨房垃圾单独分类，为垃圾堆肥提供优质原料，生产出优质有机肥，有利于改善土壤肥力，减少化肥施用量。
- 将有害垃圾分类出来，减少了垃圾中的重金属、有机污染物、致病菌的含量，有利于垃圾的无害化处理，减少了垃圾处理的水、土壤、大气污染风险。
- 提高了废品回收利用的比例，减少了原材料的需求，减少二氧化碳的排放。
- 普及环保与垃圾的知识，提升全社会对环卫行业的认知，减少环卫工人的工作难度，形成尊重、关心环卫工人的氛围。

普及环保理念和垃圾分类知识，全民参与垃圾分类，养成绿色文明的生活方式，保护我们共同的生活家园。中国的垃圾分类回收还处于起步阶段，日常生活中很多民众还不知道如何将垃圾归类。通常，垃圾分为四个大类：可回收垃圾、厨余垃圾、有害垃圾和其他垃圾，对应四个不同颜色的垃圾桶。可回收垃圾又包括废纸、塑料、玻璃、金属和布料五类等。厨余垃圾包括剩菜剩饭、骨头、菜根菜叶、果皮等食品类废物，经生物技术处理成堆肥，每吨可生产 0.6～0.7t 有机肥料。有害垃圾指含有对人体健康有害的重金属、有毒的物质或者对环境造成现实危害或者潜在危害的废弃物，包括电池、荧光灯管、灯泡、水银温度计、油漆桶、家电类、过期药品、过期化妆品等。这些垃圾一般使用单独回收或填埋处理。其他垃圾包括除上述几类垃圾之外的砖瓦陶瓷、渣土、卫生间废纸纸巾等难以回收的废弃物，这类垃圾采

取卫生填埋可有效减少对地下水、地表水、土壤及空气的污染。

5.9.2 案例任务

本案例的任务是为大众设计一个垃圾识别分类程序,根据用户丢弃的垃圾,告诉用户这是什么类型的垃圾,需要投放到什么颜色的垃圾桶。

5.9.3 案例实现

分析:本案例主要基于垃圾名称来识别垃圾所属的分类。因此,可定义四种字符串类型的常量,代表四种类型的垃圾。当用户输入某种名称的垃圾时,通过与四种字符串常量匹配,从而确定垃圾应投放到哪个垃圾桶。

【例 5.31】 Example5_31.java

```
import javax.swing.*;
public class Example5_31 {
    public static void main(String[] args) {
        //调用输入对话框并输入垃圾名称
        String waste = JOptionPane.showInputDialog("请输入垃圾名称:");
        //调用消息对话框,显示分类结果
        JOptionPane.showMessageDialog(null,
            GarbageClassification.classfier(waste));
    }
}

class GarbageClassification{
    final static String RECYCLABLE = "报纸、期刊、图书、各种包装纸、塑料制桶、" +
            "制盆、制瓶、塑料衣架、玻璃瓶、碎玻璃片、镜子、暖瓶、易拉罐、罐头盒、" +
            "衣服、桌布、洗脸巾、书包、鞋";
    final static String KITCHEN_WASTE = "剩菜剩饭、骨头、菜根菜叶、果皮、食品";
    final static String HARMFULE_WASTE = "电池、荧光灯管、灯泡、水银温度计、" +
            "油漆桶、家电类、过期药品、过期化妆品";
    final static String OTHER_WASTE = "砖瓦陶瓷、渣土、纸巾";

    /**
     * 分类方法
     * @param waste 垃圾名称
     * @return 垃圾分类
     */
    public static String classfier(String waste){
       if(RECYCLABLE.contains(waste)){
           return waste + "是可回收垃圾,请投至蓝色垃圾桶!";
       }
       else if(KITCHEN_WASTE.contains(waste)) {
           return waste + "是厨余垃圾,请投至绿色垃圾桶!";
       }
       else if(HARMFULE_WASTE.contains(waste)){
           return waste + "是有毒垃圾,请投至红色垃圾桶!";
       }
       else{
           return waste + "是其他垃圾,请投至灰色垃圾桶!";
       }
    }
}
```

运行结果如图 5.5 所示。

图 5.5 垃圾分类的运行结果

小结

本章介绍了字符串相关的类，如 String、StringBuffer、StringBuilder 和 StringTokenizer 等类的使用方法。String 类主要处理不变字符串，而 StringBuffer 和 StringBuilder 主要处理可变字符串。String 类对 Object 类的 equals()方法进行了重写，用来比较两个字符串对象的内容是否相等，而 StringBuffer、StringBuilder 类仍继承了 Object 类的 equals()方法，比较的是两个字符串地址是否相等。

Java 提供了丰富的工具类，包括时间与日期、数值与随机数、正则表达式等，这些类是应用开发中使用非常频繁的类。

数组长度不能动态改变，数组中元素的类型也必须相同。由于这些限制，Java 提供了丰富的集合对象。Java 的集合包括单列集合 Collection 接口和双列集合 Map 接口，其中，Collection 接口又分为 List 接口和 Set 接口等。List 是一种保存有序可重复元素的集合，而 Set 是一种保存无序不可重复的元素集合。Map 是一种映射关系，它的每一个元素包括两个对象：键和值，它们是一种映射模型。

泛型进一步规范了集合中元素的数据类型，使用泛型定义类、接口或方法能够解决方法重载带来的代码臃肿问题。同理，Lambda 表达式在很多场合下可以替代匿名内部类，使代码更加简洁。