6. 常用类库

6.1. 字符串操作——String类

- String可以表示一个字符串
- String类实际是使用字符数组存储的
- String的两种赋值方式：
  - 直接赋值：

```
String name = "小白";
```

  - 通过关键字new调用String的构造方法赋值：

```
String name = new String("小白");
```

- String类编译期与运行期分析：

```
// 代码示例：4种情况分析：直接赋值字符串连接时，考虑编译期和运行期
// 如果在编译期，值可以被确定，那么就使用已有的对象，否则会创建新的对象
String a = "a";
String a1 = a + 1;
String a2 = "a1";
System.out.println(a1 == a2);// false
System.out.println("----------------------");

final String b = "b";
String b1 = b + 1;
String b2 = "b1";
System.out.println(b1 == b2);// true
System.out.println("----------------------");

String c = getC();
String c1 = c + 1;
```

```
String c2 = "c1";
System.out.println(c1 == c2);// false
System.out.println("--------------------");

final String d = getD();
String d1 = d + 1;
String d2 = "d1";
System.out.println(d1 == d2);// false

private static String getC() {
    return "c";
}

private static String getD() {
    return "d";
}
```

- String类字符与字符串操作方法：

No.	方法名称	类型	描述
1	public char charAt(int index)	普通	根据下标找到指定字符
2	public char[] toCharArray()	普通	以字符数组的形式返回全部的字符串内容
3	public String(char[] value)	构造	将全部的字符数组变为字符串
4	public String(char[] value, int offset, int count)	构造	将指定范围的字符数组变为字符串

- String类字节与字符串操作方法：

No.	方法名称	类型	描述
1	public byte[] getBytes()	普通	将字符串变为字节数组
2	public String(byte[] bytes)	构造	将字节数组变为字符串
3	public String(byte[] bytes, int offset, int length)	构造	将指定范围内的字节数组变为字符串
4	public String(byte[] bytes, String charsetName)	构造	通过使用指定的charset解码指定的byte数组，构造一个新的String

- String类判断是否以指定内容开头或结尾：

No.	方法名称	类型	描述
1	public boolean startsWith(String prefix)	普通	从第一个位置开始判断是否以指定的内容开头
2	public boolean startWith(String prefix, int toffset)	普通	从指定的位置开始判断是否以指定的内容开头
3	public boolean endsWith(String suffix)	普通	判断是否以指定的内容结尾

- String类替换操作：

No.	方法名称	类型	描述
1	public String replace(char oldChar, char newChar)	普通	替换指定字符
2	public String replace(charSequence target, charSequence replacement)	普通	替换指定字符串
3	public String replaceAll(String regex, String replacement)	普通	替换指定的字符串
4	public String replaceFirst(String regex, String replacement)	普通	替换第一个满足条件的字符串

- String类字符串截取操作：

No.	方法名称	类型	描述
1	public String substring(int beginIndex)	普通	从指定位置开始一直截取到末尾
2	public String substring(int beginIndex, int endIndex)	普通	截取指定范围的字符串

- String类字符串拆分操作：

No.	方法名称	类型	描述
1	public String[] split(String regex)	普通	按照指定的字符串拆分
2	public String[] split(String regex, int limit)	普通	拆分字符串，并指定拆分的个数

- String类字符串查找操作：

No.	方法名称	类	描述

No.	方法名称	类型	描述
1	public boolean contains(String s)	普通	返回一个字符串是否存在
2	public int indexOf(int ch)	普通	从头查找指定的字符是否存在，chat→int，如果存在则返回位置，如果不存在则返回“-1”
3	public int indexOf(int ch, int fromIndex)	普通	从指定位置查找指定的字符是否存在，char→int，如果存在则返回位置，如果不存在则返回“-1"
4	public int indexOf(String str)	普通	从头查找指定的字符串是否存在，如果存在则返回位置，如果不存在则返回”-1“
5	public int indexOf(String str, int fromIndex)	普通	从指定位置查找字符串是否存在，如果存在则返回位置，如果不存在则返回”-1“
6	public int lastIndexOf(int ch)	普通	从字符串的最后向前查找，指定的字符是否存在，如果存在则返回位置，如果不存在则返回”-1“
7	public int lastIndexOf(int ch, int fromIndex)	普通	从字符串的指定的末尾向前查找，指定的字符是否存在，如果存在则返回位置，如果不存在则返回”-1“
8	public int lastIndex(String str)	普通	从字符串的最后向前查找，指定的字符串是否存在，如果存在则返回位置，如果不存在则返回”-1“
9	public int lastIndexOf(String str, int fromIndex)	普通	从字符串的指定的末尾向前查找，指定的字符串是否存在，如果存在则返回位置，如果不存在则返回”-1”

- String类其他操作：

No.	方法名称	类型	描述
1	public boolean isEmpty()	普通	判断是否为空，指的是内容为空”“
2	public int length()	普通	取得字符串的长度
3	public String toLowerCase()	普通	转小写
4	public String toUpperCase()	普通	转大写
5	public String trim()	普通	去掉开头和结尾的空格，中间的空格不去
6	public String concat(String str)	普通	字符串连接操作

6.2. 字符串操作——StringBuffer类

- 由于使用String连接字符串，代码性能非常低，所以采用StringBuffer
- StringBuffer常用操作方法：

方法名称	描述

方法名称	描述
public StringBuffer()	构造一个空的StringBuffer对象
public StringBuffer(String str)	将指定的String变为StringBuffer的内容
public StringBuffer(CharSequence seq)	接收CharSequence接口的实例
public StringBuffer append(数据类型 b)	提供了很多append()方法，用于进行字符串连接
public StringBuffer delete(int start, int end)	删除指定位置的内容
public int indexOf(String str)	字符串的查询功能
public StringBuffer insert(int offset, 数据类型 b)	在指定位置上增加一个内容
public StringBuffer replace(int start, int end, String str)	将指定范围内的内容换成其他内容
public String substring(int start, int end)	截取指定范围的字符串
public String substring(int start)	字符串截取
public StringBuffer reverse()	字符串反转

6.3. 字符串操作——StringBuilder类

- StringBuffer的兄弟类：StringBuilder
  - 一个可变的字符序列，此类提供一个与StringBuffer兼容的API，但不保证同步。该类被设计用作StringBuffer的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先使用该类，因为在大多数实现中，它比StringBuffer要快
- 面试题：StringBuffer和StringBuilder的区别？
  - StringBuffer：线程安全，性能低，适用于多线程，JDK1.0
  - StringBuilder：线程不安全，性能高，适用于但线程，JDK1.5

6.4. 程序国际化

- 对国际化(internationalization)的理解：同一套程序代码可以在各个语言环境下使用。各个语言环境下，只是语言显示的不同，那么具体的程序操作本身都是一样的，那么国际化程序完成的就是这样一个功能
- Locale类：

- Locale对象表示了特定的地理、政治和文化地区。需要Locale来执行其任务的操作称为语言环境敏感操作，它使用Locale为用户量身定制信息
- 使用此类的中的构造方法来创建Locale对象：

```
Locale(String language);
LOcale(String language, String country);
```

- 通过静态方法创建Locale：

```
getDefault();
```

- ResourceBundle类：
  - 国际化的实现核心在于显示的语言上，通常的做法是将其定义成若干个属性文件（文件后缀是*.properties），属性文件的格式采用“key=value”的格式进行操作
  - ResourceBundle类表示的是一个资源文件的读取操作，所有的资源文件需要使用ResourceBundle进行读取，读取的时候不需要加上文件的后缀

```
getBundle(String baseName);
getBundle(String baseName, Locale locale);
getString(String key);
```

  - 处理动态文本：
    - 进行动态文本处理，必须使用java.text.MessageFormat类完成
- 国际化示例：
  - 主类：

```
package day06_常用类库.i18n;

import java.text.MessageFormat;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.Scanner;

/**
 * @author xiao儿
 * @date 2019年8月26日 上午9:02:44
 * @description 程序国际化
 * 1.Locale
 * 2.Properties 文件：属性文件（配置文件），内容以键值对的形式存放：key:value
 * 3.ResourceBundle 工具类：来绑定属性文件，并指定 Locale 对象，来自动选择使用那个属性文件，默认将使用与操作系统相同的语言环境
```

```
 *  getString() 方法从属性文件中使用key来获取value
 * 注意：配置文件是只读的
 */
public class Internationalization {
    public static void main(String[] args) {
        // 创建一个本地语言环境对象，该对象会根据参数设置来自动选择与之相关的语言环境
        // 参数：语言，地区
        // Locale locale_CN = new Locale("zh", "CN");
        Locale locale_US = new Locale("en", "US");
        // 获取当前系统默认的语言环境
        // Locale locale_default = Locale.getDefault();

        Scanner scanner = new Scanner(System.in);
        // 用户绑定属性文件的工具类（参数：属性文件的基本名（就是前缀，比如：info））
        ResourceBundle resourceBundle = ResourceBundle.getBundle("day06_常用类库.i18n.info", locale_US);

System.out.println(resourceBundle.getString("System.name"));

System.out.println(resourceBundle.getString("input.username"));
        String username = scanner.nextLine();

System.out.println(resourceBundle.getString("input.password"));
        String password = scanner.nextLine();

        if ("admin".equals(username) && "123456".equals(password)) {

System.out.println(resourceBundle.getString("login.success"));
            String welcome = resourceBundle.getString("welcome");
            // 动态文本格式化
            welcome = MessageFormat.format(welcome, username);
            System.out.println(welcome);
        } else {

System.out.println(resourceBundle.getString("login.error"));
        }

        scanner.close();
    }
}
```

- 配置文件：

```
// info_zh_CN.properties
System.name = \u5458\u5DE5\u7BA1\u7406\u7CFB\u7EDF
input.username = \u8F93\u5165\u7528\u6237\u540D\uFF1A
input.password = \u8F93\u5165\u5BC6\u7801\uFF1A
login.success = \u767B\u5F55\u6210\u529F\uFF01
login.error = \u767B\u5F55\u9519\u8BEF
welcome = \u6B22\u8FCE\u60A8\uFF0C{0}

// info_en_US.properties
System.name = EMP Manager System
input.username = Input UserName:
```

```
input.password = Input Password:
login.success = Login Success!
login.error = Login Error
welcome = welcome,{0}
```

6.5. Math和Random类

- Math类：
  - Math类包含用于执行基本数学运算的方法，如初等函数、对数、平方根和三角函数
  - 使用Math类有两种方式：
    - 直接使用（Math所在的包java.lang为默认引入的包）
    - 使用import static java.lang.Matn.abs;静态引入

static double PI	比任何其他值都更接近pi的double值
abs(double a)	返回double值的绝对值
random()	返回带正号的dobule值，该值大于等于0.0且小于1.0
round(double a)	返回最接近参数并等于某一整数的double值
sqrt(double a)	返回正确舍入的dobule值的正

- Random类：
  - 此实例用于生成伪随机数流

nextLong()	返回下一个伪随机数的long值
nextBoolean()	返回下一个伪随机数boolean值
nextDouble()	返回下一个伪随机数，在0.0和1.0之间的double值
nextFloat()	返回下一个伪随机数，在0.0和1.0之间的float值
nextInt()	返回下一个伪随机数，int值
nextInt(int n)	返回下一个伪随机数，在0（包括）和指定值分布的int值

6.6. 日期操作类

- Date类：
  - 类Date表示特定的瞬间，精确到毫秒，也就是程序运行时的当前时间

```
// 实例化Date对象，表示当前时间
Date date = new Date();
```

- Calendar类：
  - Calendar，日历类，使用此类可以将时间精确到毫秒显示

```
// 两种实例化方式
Calendar c = Calendar.getInstance();
Calendar c = new GregorianCalendar();
```

- DateFormat类及子类SimpleDateFormat

```
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateFormatDemo {
    public static void main(String[] args) {
        DateFormat dateFormat = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss SSS");
        String nowDate = dateFormat.format(new Date());
        System.out.println(nowDate);
    }
}
```

6.7. 对象比较器

- 对两个或多个数据项进行比较，以确定它们是否相等，或确定它们之间的大小关系及排列顺序称为比较
- Comparable接口：
  - 此接口强行对实现它的每个类的对象进行整体排序。这种排序被称为类的自然排序，类的comparaTo方法被称为它的自然比较方法
  - 示例：

```
public class Cat implements Comparable<Cat> {
    private String name;
    private int age;

    public Cat() {
    }

    public Cat(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
20
21      public int getAge() {
22          return age;
23      }
24
25      public void setAge(int age) {
26          this.age = age;
27      }
28
29      @Override
30      public String toString() {
31          return "Cat [name=" + name + ", age=" + age + "]";
32      }
33
34      @Override
35      public int compareTo(Cat cat) {
36  //      if (this.age < cat.age)
37  //          return -1;
38  //      if (this.age > cat.age)
39  //          return 1;
40  //      return 0;
41          return this.age - cat.age;
42      }
43  }
```

- Comparator接口：
  - Comparator接口：强行对某个对象collection进行整体排序的比较
  - 注意：适用于如果该类已经实现好了，不想进行修改时，可以实现Comparator接口
  - 示例：

```
1   // Dog
2   public class Dog {
3       private String name;
4       private int age;
5
6       public Dog() {
7       }
8
9       public Dog(String name, int age) {
10          this.name = name;
11          this.age = age;
12      }
13
14      public String getName() {
15          return name;
16      }
17
18      public void setName(String name) {
19          this.name = name;
20      }
21
22      public int getAge() {
```

```
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Dog [name=" + name + ", age=" + age + "]";
    }
}

// DogComparator
import java.util.Comparator;

public class DogComparator implements Comparator<Dog> {

    @Override
    public int compare(Dog o1, Dog o2) {
        return o1.getAge() - o2.getAge();
    }
}
```

6.8. 对象的克隆

- 将一个对象复制一份，称为对象的克隆技术
- 在Object类中存在一个clone()方法：

```
protected Object clone() throws CloneNotSupportedException
```

  - 如果某个类的对象要想被克隆，则对象所在的类必须实现Cloneable接口。此接口没有任何方法，是一个标记接口
- 示例：

```
// Cat
/**
 * @author xiao儿
 * @date 2019年8月26日 下午5:43:19
 * @description Cat对象
 * 对象需要具备克隆功能:
 * 1.实现 Cloneable 接口（标记接口）
 * 2.重写 Object 类中的 clone() 方法
 */
public class Cat implements Cloneable {
    private String name;
    private int age;

    public Cat() {
    }

    public Cat(String name, int age) {
        this.name = name;
        this.age = age;
```

```
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Cat [name=" + name + ", age=" + age + "]";
    }

    // 重写 Object 中的 clone() 方法
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

// CloneDemo
public class CloneDemo {
    public static void main(String[] args) {
        Cat cat = new Cat("喵喵", 3);
        try {
            Cat newCat = (Cat) cat.clone();
            System.out.println("cat=" + cat);
            System.out.println("newCat=" + newCat);
            System.out.println(cat == newCat);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

6.9. System与Runtime类

- System类：
  - System类代表系统，系统级的很多属性和控制方法都放置在该类的内部。该类位于java.lang包
  - 成员变量：
    - System类内部包含in、out和err三个成员变量，分别代表标准输入流（键盘输入），标准输出流（显示器）和标准错误输出流

- 成员方法：
  - System类提供了一些系统级的操作方法：

```
// 该方法的作用是拷贝数组，也就是将一个数组中的内容复制到另一个数组中的指定位置，由于该方法是native方法，所以性能上比使用循环高效
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length);

// 该方法的作用是返回当前的计算机时间，时间的表达格式为当前计算机时间和GMT时间（格林威治时间）1970年1月1号0时0分0秒所差的毫秒数
public static long currentTimeMills();

// 该方法的作用是退出程序。其中status的值为0代表正常退出，非零代表异常退出。使用该方法可以在图形界面编程中实现程序的退出功能等
public static void exit(int status);

// 该方法的作用是请求系统进行垃圾回收。至于系统是否立刻回收，则取决于系统中垃圾回收算法的实现以及系统执行时的情况
public static void gc();

// 该方法的作用是获得系统中属性名为key的属性对应的值
public static String getProperty(String key);
    java.version    java运行时环境版本
    java.home       java安装目录
    os.name         操作系统的名称
    os.version      操作系统的版本
    user.name       用户的账户名称
    user.home       用户的主目录
    user.dir        用户的当前工作目录
```

- Runtime类：
  - 每个Java应用程序都有一个Runtime类实例，使应用程序能够与其运行的环境相连接

```
// 获取Java运行时相关的运行时对象// 获取Java运行时相关的运行时对象
Runtime runtime = Runtime.getRuntime();
System.out.println("处理器数量: " + runtime.availableProcessors() + "个");
System.out.println("JVM总内存数: " + runtime.totalMemory() + "byte");
System.out.println("JVM空闲内存数: " + runtime.freeMemory() + "byte");
System.out.println("JVM可用最大内存数: " + runtime.maxMemory() + "byte");

// 在单独的进程中执行指定的字符串命令
runtime.exec("notepad");
```

6.10. 数字处理工具类

- BigInteger：可以让超过Integer范围内的数据进行运算

```java
// 构造方法
public BigInteger(String val);

// 常用方法
public BigInteger add(BigInteger val);
public BigInteger subtract(BigInteger val);
public BigInteger multiply(BigInteger val);
public BigInteger divide(BigInteger val);
public BigInteger[] divideAndRemainder(BigInteger val);
```

- BigDecimal：由于在运算的时候，float类型和double很容易丢失精度，为了能精确地表示、计算浮点数，Java提供了BigDecimal，不可变的、任意精度的有符号的十进制数

```java
// 构造方法
public BigDecimal(String val);

// 常用方法
public BigDecimal add(BigDecimal augend);
public BigDecimal subtract(BigDecimal subtrahend);
public BigDecimal multiply(BigDecimal multiplicand);
public BigDecimal divide(BigDecimal divisor);
```

- DecimalFormat：Java提供DecimalFormat类，快速格式化数据

```java
// 圆周率
double pi = 3.141592657;
// 取一位整数，结果：3
System.out.println(new DecimalFormat("0").format(pi));
// 取一位整数和两位小数，结果：3.14
System.out.println(new DecimalFormat("0.00").format(pi));
// 取两位整数和三位小数，整数不足部分以0填补，结果：03.142
System.out.println(new DecimalFormat("00.000").format(pi));
// 取所有整数部分，结果：3
System.out.println(new DecimalFormat("#").format(pi));
// 以百分比方式计数，并取两位小数，结果：314.16%
System.out.println(new DecimalFormat("#.##%").format(pi));
```

6.11. MD5工具类

- MD5的全称是Message-Digest Algorithm 5（信息-摘要算法）

```java
MessageDigest md5 = MessageDigest.getInstance("MD5");// SHA-1
// 通过MD5计算摘要
byte[] bytes = md5.digest(password.getBytes("utf-8"));
System.out.println(Arrays.toString(bytes));
// String newPassword = new String(bytes);
// System.out.println(newPassword);
// a-z  A-Z 0-9 / * BASE64编码算法
// JDK1.8版本
String newPassword = Base64.getEncoder().encodeToString(bytes);
System.out.println(newPassword);
// 解码
byte[] byte2 = Base64.getDecoder().decode(newPassword);
System.out.println(Arrays.toString(byte2));
```

6.12. 数据结构之二叉树实现

- 树是一种重要的线性表数据结构，直观来说，它是数据元素（在树中称为结点）按分支关系组织起来的结构。二叉树（Binary Tree）是每个结点最多有两个子树的有序树。通常子树被称作“左子树”和“右子树”
- 二叉树算法的排序规则：
  - 选择第一个元素作为根结点
  - 之后如果元素大于根结点放在右子树，如果元素小于根结点，则放在左子树
  - 最后按照中序遍历的方式进行输出，则可以得到排序的结果（左➡根➡右）
- 示例：

```java
public class BinaryTree {
    private Node root;

    public void add(int data) {
        if (root == null) {
            root = new Node(data);
        } else {
            root.addNode(data);
        }
    }

    // 输出结点
    public void print() {
        root.printNode();
    }

    private class Node {
        private int data;
        private Node left;
        private Node right;

        public Node(int data) {
            this.data = data;
        }
```

```
25
26          public void addNode(int data) {
27              if (this.data > data) {
28                  if (this.left == null) {
29                      this.left = new Node(data);
30                  } else {
31                      this.left.addNode(data);
32                  }
33              } else {
34                  if (this.right == null) {
35                      this.right = new Node(data);
36                  } else {
37                      this.right.addNode(data);
38                  }
39              }
40          }
41
42          // 中序遍历
43          public void printNode() {
44              if (this.left != null) {
45                  this.left.printNode();
46              }
47              System.out.print(this.data + "->");
48              if (this.right != null) {
49                  this.right.printNode();
50              }
51          }
52      }
53  }
```

6.13. JDK1.8新特性

- Lambda表达式:
  - Lambda表达式（也称为闭包）是整个Java 8发行版中最受期待的在Java语言层面上的改变，Lambda允许把函数作为一个方法的参数（函数作为参数传递进方法中），或者把代码看成数据。Lambda表达式用于简化Java中接口式的匿名内部类。被称为函数式接口的概念。函数式接口就是一个具有一个方法的普通接口。像这样的接口，可以被隐式转换为Lambda表达式

```
// 语法
(参数1, 参数2,...) -> {...}

1.没有参数时使用Lambda表达式
2.带参数时使用Lambda表达式
3.代码块中只一句代码时使用Lambda表达式
4.代码块中有多句代码时使用Lambda表达式
5.有返回值的代码块
6.参数中使用final关键字
```

- 示例:

```java
package day06_常用类库.lambda;

public class LambdaDemo {
    public static void main(String[] args) {
        IEat iEat = new IEatImpl();
        iEat.eat();

        IEat iEat2 = new IEat() {
            @Override
            public void eat() {
                System.out.println("eat banana");
            }
        };
        iEat2.eat();

        // Lambda 表达式
        // 好处：1.代码更简洁；2.不会单独生成class文件
//      IEat iEat3 = () -> {
//          System.out.println("eat apple banana");
//      };
        // 没有参数时使用
        IEat iEat3 = () -> System.out.println("eat apple banana");
        iEat3.eat();

        // 带参数时使用
        IDrink iDrink = (thing, name) -> {
            System.out.println("drink--" + thing + "--");
            System.out.println(name);
        };
        iDrink.drink("water", "大冰");

        // 但返回值的方法
        IPlay iPlay = (thing, name) -> {
            System.out.println(name + " play " + thing);
            return 10;
        };
        iPlay.play("足球", "大冰");

        // 带返回值的方法只有一句代码实现
//      IPlay iPlay2 = (thing, name) -> {
//          return 10;
//      };
        // IPlay iPlay2 = (thing, name) -> 10;
        IPlay iPlay2 = (thing, name) -> thing == null ? 1 : 0;
        iPlay2.play("篮球", "大冰");

        // 带final关键字
        IPlay iPlay3 = (final String thing, final String name) -> 10;
        iPlay3.play("乒乓球", "大冰");
    }
}

// 只有一个抽象方法的接口
interface IEat {
    void eat();
    // 默认方法不影响
    public default void print() {
        System.out.println("默认的方法");
```

```
59      }
60      // 静态方法也不影响
61      public static void method() {
62          System.out.println("静态的方法");
63      }
64  }
65
66  class IEatImpl implements IEat {
67      @Override
68      public void eat() {
69          System.out.println("eat apple");
70      }
71  }
72
73  interface IDrink {
74      void drink(String thing, String name);
75  }
76
77  interface IPlay {
78      int play(String thing, String name);
79  }
```