##

Java常用API

Math

- Math是一个帮助用于数学计算的工具类
- 私有化构造方法，所有的方法都是静态的
- 常用方法

方法名	返回值
abs(double a)	获取绝对值
ceil(double a)	向上取整，取坐标轴上离a最近最靠近正无穷的数
floor(double a)	向下取整，取坐标轴上离a最近最靠近负无穷的数
round(float a)	四舍五入
max(int a, int b)	获取两者的较大值
pow(double a, double b)	返回a的b次幂
sqrt(double a)	a的平方根
cbrt(double a)	a的立方根
random()	返回值为double的随机值，范围[0.0, 1.0)

System

- System也是一个工具类，提供了一个与系统相关的方法
- 常用方法

方法名	说明
`public static void exit(int status)`	终止当前运行的Java虚拟机
`public static long currentTimeMillis()`	返回当前系统的时间毫秒值形式
`punlic static void arraycopy(数据源数组，起始索引，目的地数组，起始索引，拷贝个数)`	数组拷贝

- 计算机的时间原点
  - `1970年1月1日 00:00:00`
  - 因为中国位于东八区，因此其起始时间为`1970年1月1日 08:00:00`
- 数组拷贝的细节
  - 如果数据源数组和目的地数组都是基本数据类型，那么两者类型必须一模一样

- 在拷贝的时候必须考虑数组的长度
- 如果都是引用数据类型（例如对象），那么子类类型可以赋值给父类（只是需要强转）

Runtime

- Runtime表示当前虚拟机的运行环境
- 常用方法

方法名	说明
`public static Runtime getRuntime()`	当前系统的运行环境对象
`public void exit(int status)`	停止虚拟机
`public int availableProcessors()`	获得CPU的线程数
`public long maxMemory()`	JVM能从系统中获取总内存大小（单位byte)
`public long totalMemory()`	JVM已经从系统中获取总内存大小（单位byte)
`public long freeMemory()`	JVM剩余内存大小(单位byte)
`public Process exec(String command)`	运行cmd命令

Object

- Object是Java中的顶级父类。所有的类都直接或间接的继承于Object类
- Object类中的方法可以被所有子类访问，所以我们要学习Object类和其中的方法
- Object只有无参构造
- Object的成员方法

方法名	说明
public String toString( )	返回对象的字符串表示
public boolean equals(Object obj)	比较两个对象是否相等（比较的是地址值）
protected Object clone(int a)	对象克隆

- toString方法的结论
  - 当我们打印一个对象的时候底层会调用对象的toString方法，把对象编变成字符串，然后再打印到控制台上，打印完毕换行处理
  - 在默认情况下，Object类中的toString方法返回的是地址值，如果想要知道对象内部的属性值，就需要重写Object中的toString方法
- equals方法的结论
  - Object类中的equals默认比较的是地址值，但是往往我们需要比较的是属性值，因此有需要的话就需要重写
  - 调用equals方法要看是谁调用的，如String调用，因为String中重写了equals方法，而StringBuilder中没有重写，所以两者判断方式也不相同
    - String调用：必须比较的双方都是字符串且具有相同字符序列才可以
    - StringBuilder调用：必须两者地址值相同才相等（使用的Object类中的方法）
- 对象克隆
  - 把A对象的属性值完全拷贝给B对象，也叫对象拷贝，对象复制

- 方法在底层会帮我们创建一个对象，并把原对象中的数据拷贝过去
- 书写细节
  - 重写Object中的clone方法
  - 让javabean类实现Cloneable接口
  - 创建元对象并调用clone即可
- 浅克隆
  - 不管对象内部的属性是基本数据类型还是引用数据类型，都完全拷贝过来（例如数组的地址也会相同赋值）
- 深克隆
  - 基本数据类型拷贝过来
  - 字符串复用
  - 引用数据类型会重新创建新的地址存储（例如数组的地址会重新申请而不与克隆的源对象相同）
- Object中的克隆默认是浅克隆，因此如果想要实现深克隆就需要重写clone方法

Objects

- Objects是一个工具类，提供了一些操作对象的方法
- Objects的成员方法

方法名	说明
`public static boolean equals(Object a, Object b)`	先做非空判断，再判断对象是否相同
`public static boolean isNull(Object obj)`	判断对象是否为空
`public static boolean nonNull(Object obj)`	判断对象是否不为空

- equals细节
  1. 方法的底层会判断s1是否为null，如果为null，直接返回false
  2. 如果s1不为null，那么就利用s1再次调用equals方法
  3. 如果s1是Student类型，那么最终还是会调用Studnet中的equals方法
  4. 如果没重写，则比较地址值，如果重写了，则用重写后的方法比较

BigInteger

- 在Java中，整数有四种类型：`byte`,`short`,`int`,`long`
- 在底层占用字节个数: `byte`1个字节、`short`2个字节、`int`4个字节、`long`8个字节
- BigInteger构造方法

方法名	说明
`public BigInteger(int num,Random rnd)`	获取随机大整数，范围:[0~2的num次方-1]
`public BigInteger(String val)`	获取指定的大整数（字符串中必须是整数）
`public BigInteger(string val, int radix)`	获取指定进制的大整数（字符串中必须是整数且必须跟进制吻合，例如八进制中每位不能大于8）
`public Static BigInteger`	静态方法获取BigInteger的对象，内部有优化

valueOf(long val)	
方法名	**说明**

- 对象一旦创建里面的数据不能发生改变
- valueOf方法
  - 能表示范围比较小，只能在`long`的取值范围之内，如果超出`long`的范围就不行了
  - 在内部对常用的数字：`-16 ~ 16`进行了优化
  - 提前把`-16 ~ 16`先创建好`BigInteger`的对象，如果多次获取不会重新创建新的
- BigInteger中的常见方法

方法名	说明
`public BigInteger add(BigInteger val)`	加法
`public BigInteger subtract(BigInteger val)`	减法
`public BigInteger multiply(BigInteger val)`	乘法
`public BigInteger divide(BigInteger val)`	除法，获取商
`public BigInteger[] divideAndRemainder（BigInteger val)`	除法，获取商和余数
`public booleanl equals(object x)`	比较是否相同
`public BigInteger pow (int exponent)`	次幂
`public BigInteger max/min(BigInteger val)`	返回较大值/较小值
`public int intValue(BigInteger val)`	转为int类型整数，超出范围数据有误

- BigInteger时如何存放的
  - 符号位专门存入一个变量中
  - 数据位从转换为二进制之后的最低位开始每隔32位存放进mag数组中

BigDecimal

- 计算机中的小数
  - 计算机的小数无法完全正确的表示一个数，使用BigDecimal可以精确存储
- BigDecimal构造方法

方法名	说明
`public static BigDecimal valueOf(double val)`	将字符串转换为对应的这通常是将doub1e（或f1oat）转化为BigDecima1 的首选方法
`public BigDecimal(double val)`	此构造方法的结果有一定的不可预知性

- 细节
  - 如果要表示的数字不大，没有超出double的取值范围，建议使用静态方法
  - 如果要表示的数字比较大，超出了double的取值范围，建议使用构造方法

    - 如果我们传递的是0~10之间的整数，包含0，包含10，那么方法会返回已经创建好的对象，不会重新new
- BigDecimal存储原理
    - 将每一位的数字（包括符号位和小数点位）对应的ASCII码值存储到一个byte类型的数组中

正则表达式

- 正则表达式可以校验字符串是否满足一定的规则，并用来校验数据格式的合法性。
- 字符类

```
[abc]               //只能是a、b、c
[^abc]              //除了a、b、c之外的任何字符
[a-zA-Z]            //a到zA到Z，包括（范围）
[a-d[m-p]]          //a到d，或m到p
[z-a&[def]]         //a-z，def和&符合（这就不是交集的意思了，只是一个简单的符号）
[z-a&&[def]]        //a-z和def的交集。为:d，e，f
[a-z&&[^bc]]        //a-z和非bc的交集。（等同于[ad-z])
[a-z&&[^m-p]]       //a到z和除了m到p的交集（等同于[a-lq-z]）
```

- 正则表达式是一个一个把字符串的字符与之匹配的，如果字符串中有多个字符，那么就需要多写几个与之匹配的规则
- 其他规则

```
(?i)        //忽略随后的字母的大小写
[]          //里面的条件出现一次
^           //取反
&&          //交集
|           //写在方括号外表示并集
()          //分组
a((?i)b)c   //只忽略b的大小写
```

- 预定义字符（只匹配一个）

```
.           //任何字符
\d          //一个数字
\D          //非数字
\s          //一个空白字符
\S          //非空白字符
\w          //[a-zA-Z_0-9]英文、数字、下划线
\W          //[^\w]一个非单词字符
```

- Java中`\`表示转义字符，只有`\\`才相当于实际字符`\`
- 因为`\d`表示一个数字，但是在java中匹配的时候必须用`\\d`来匹配
- 数量词

```
X?          //X出现一次或0次
X*          //X出现零次或多次
X+          //X出现一次或多次
X{n}        //X出现正好n次
X{n,}       //X出现至少n次
X{n,m}      //X出现至少n但不超过m次
```

- 举例

```
//匹配手机号
//  1表示手机号码只能从1开始
//  [3-9]表示手机号码第二位只能是3~9之间
//  \\d{9}任意数字可以连续出现九次，也只能出现九次，只能是数字
String regex1 = "1[3-9]\\d{9}"
"待校验的手机号".matches(refex1);

//匹配座机号码
//1.区号
//  0表示区号一定以0开头
//  \\d{2,3}表示第二位开始只能是数字，可以出现2到3次
//  -?表示-这个符号可以出现0次到1次
//2.号码
//  [1-9]表示号码第一位不能是0开头
//  \\d{4,9}表示号码从第二位看是可以实任意字符，号码总长度可以是5~10位
String regex2 = "0\\d{2,3}-?[1-9]\\d{4,9}"
"待校验的座机号号".matches(regex2);

//匹配邮箱号码
String regex3 = "\\w+@[\\w&&[^_]]{2,6}(\\.[a-zA-Z]{2,3}){1,2}"
"待匹配的邮箱账号".matches(regex3);
```

- pattern类
  - 在API帮助文档中查找Pattren类，里面有相关的正则内容
- 正则表达式在字符串方法中的使用

方法名	说明
`matches`	判断字符串是否满足正则表达式的要求
`repalceAll`	按照正则表达式的规则进行替换
`split`	按照正则表达式的规则切割字符串

- 分组
  - 每个小括号就是一个分组，每组都是有组号的
    - 从1开始，连续不间断
    - 以左括号为基准，最左边是第一组，其次为第二组
- 捕获分组
  - 正则内部使用：\组号
  - 正则外部使用：$组号

```
\\X          //把第X组的内容再拿出来匹配一次
$X           //把第X组的内容再拿出来用一次

//需要去除重复内容
String str = "我要学编编编编编程程程程程程";
String retStr = str.replaceAll("(.)\\1+", "$1");
System.out.println(retStr);//输出"我要学编程"
```

- 非捕获分组

- 分组之后不需要再用本组数据，仅仅是把数据括起来，不占用组号

```
(?=X)       //?表示其前面的某个字符串，该字符串后面需要有X，X不会被读取到
(?:X)       //?表示其前面的某个字符串，该字符串后面需要有X，X会被读取到
(?!X)       //?表示其前面的某个字符串，该字符串后面不能有X
```

爬虫

- 贪婪爬取
  - 在爬取数据的时候尽可能的多获取数据
- 非贪婪爬取
  - 在爬取数据的时候尽可能的少获取数据
- Java中默认的就是贪婪爬取，如果在数量词+ *的后面加上问号，那么就是非贪婪爬起
  - 例如正则表达式"ab+"
    - 默认情况下就是贪婪爬取
    - 如果改为"ab+?"就是非贪婪爬取

时间相关类

- 世界标准时间
  - 格林尼治时间/格林威治时间(Greenwich Mean Time）简称GMT
  - 目前世界标准时间（UTC)已经替换为:原子钟
- JDK前时间相关类
  - `Date`时间类
    - Date类是一个JDK写好的Javabean类，用来描述时间，精确到毫秒
    - 利用空参构造创建的对象，默认表示系统当前时间
    - 利用有参构造创建的对象，表示指定的时间
    - **如何创建日期对象**
      - `Date date = new Date();`
      - `Date date = new Date(指定毫秒值);`
    - **如何修改时间对象中的毫秒值**
      - `setTime(毫秒值);`
    - **如何获取时间对象中的毫秒值**
      - `getTime();`
  - `SimpleDateFormat`类
    - 格式化：把时间变成想要的格式
    - 解析：把字符串表示的时间变成`Date`对象
    - 如何创建对象
      - `SimplerDateFormat();`
      - `SimplerDateFormat(String pattren);`
    - 格式化（日期对象转换为字符串）
      - `public final String format(Date date);`
    - 解析
      - `public Date parse(String source);`
  - `Calendar`类
    - `Calendar`表示系统当前时间的日历对象，可以单独修改、获取时间中的年月日

        - `Calendar`是一个抽象类，不能直接创建对象
        - 获取`Calender`日历类对象的方法
            - `public static Calendar getInstance();`
        - `Calendar`常用方法
            - `get`方法
                - 细节1：月份是从0~11月，而不是1~12月
                - 细节2：得到的星期日期是从星期日开始的
            - `set`方法
                - 修改日期时间
            - `add`方法
                - 在原有的基础上增加或者减少
- JDK8新增时间相关类
    - `ZomeId`类
        - `getAvailableZoneIds();//获取所有的时区名称`
        - `systemDefault();//获取当前系统的默认时区`
        - `of(String zongId);//获取指定的时区`
    - `Instant`类

方法名	说明
`static Instant now()`	获取当前时间的Instant对象（标准时间）
`static Instant ofXxxx(long epochMilli)`	根据（秒/毫秒/纳秒）获取Instant对象
`ZonedDateTime atZone(ZoneId zone)`	指定时区
`boolean isXxx(Instant otherInstant)`	判断系列的方法
`Instant minusXxx(long millisToSubtract)`	减少时间系列的方法
`Instant plusXxx(long millisToSubtract)`	增加时间系列的方法

    - `ZoneDateTime`类

方法名	说明
`static ZonedDateTime now()`	获取当前时间的ZonedDateTime对象
`static ZonedDateTime ofXxxx(。。。)`	获取指定时间的ZonedDateTime对象
`ZonedDateTime withXxx(时间)`	修改时间系列的方法
`ZonedDateTime minusXxx(时间)`	减少时间系列的方法
`ZonedDateTime plusXxx(时间)`	增加时间系列的方法

    - `DateTimeFormatted`类

方法名	说明
`static DateTimeFormatter ofPattern(格式)`	获取格式对象
`String format(时间对象)`	按照指定方式格式化

- `LocalDate`类（年月日）、`LocalTime`类（时分秒）、`LocalDateTime`类

方法名	说明
`static XXX now()`	获取当前时间的对象
`static XXX of(。。。)`	获取指定时间的对象
`get开头的方法`	获取日历中的年、月、日、时、分、秒等信息
`isBefore, isAfter`	比较两个 LocalDate
`with开头的`	修改时间系列的方法
`minus开头的`	减少时间系列的方法
`plus开头的`	增加时间系列的方法

方法名	说明
public LocalDate toLocalDate()	LocalDateTime转换成一个LocalDate对象
public LocalTime toLocalTime()	LocalDateTime转换成一个LocalTime对象

- `Duration`类（时间间隔，秒、纳秒）
- `Period`类（时间间隔，年月日）
- `ChronoUnit`（时间间隔，所有单位）

包装类

- 包装类的本质就是用一个对象把一个基本数据类型给包装起来
- 包装类即基本数据类型对应的引用数据类型
- 获取Integer对象的方式（了解即可，这是JDK5以前的方法）

方法	说明
public Integer(int value)	根据传递的值创建一个Integer对象
public Integer(String s)	根据传递的字符串创建一个Integer对象
pubic static Integer valueOf(int i)	根据传递的值创建一个Integer对象
pubic static Integer valueOf(String s)	根据传递字符串创建一个Integer对象
pubic static Integer valueOf(String s, int radix)	根据传递的字符串按照进制创建一个Integer对象

- 在JDK5以后，Integer与int可以实现自动转换（Java底层会自动转换），因此可以看成是一个统一的对象
- Integer成员方法

方法名	说明
public static String toBinaryString(int i)	得到二进制
public static String toOctalString(int i)	得到八进制
public static String toHexString(int i)	得到十六进制
public static int parseInt(String s)	将字符串类型的整数转成int类型的整数

- Java中是强类型语言，即每种数据在java中都有各自的数据类型，即在计算的时候，如果不是同一种数据类型是无法直接计算的
- 8中包装类当中，除了Character都有对应的parseXxx的方法，进行类型转换
- 如果以后想要键盘录入，不管什么类型，统一使用nextLine

算法

查找算法

基本查找

- 核心思路是从0开始挨个往后查找

```
public static boolean basicSearch(int[] arr, int number){
    for (int i = 0; i < arr.length; i++) {
        if(arr[i] == number)
            return true;
    }
    return false;
}
```

- 如果要返回多个数据，则可以把这些数据放到数组（如果已知长度）或者集合中

二分查找/折半查找

- 前提条件：数据中的数据必须是有序的
- 核心逻辑：每次排除一半的查找范围

```
//min和max表示当前要查找的范围
//mid是在min和max中间的
//如果要查找元素的在mid的左边，则缩小范围时，min不变，max等于min-1
//如果要查找元素的在mid的左边，则缩小范围时，max不变，min等于min-1
public static int halfSearch(int[] arr, int number){
    int min = 0;
    int max = arr.length-1;
    int mid = (min+max)/2;
    while(min<=max){
        if(arr[mid] == number)
            return mid;
        else if(arr[mid] > number)
            max = mid-1;
        else
            min = mid+1;
        mid = (min+max)/2;
    }
    return -1;
```

```
}
```

二分查找改进之插值查找

```
//在计算时初步计算了待求值大约在数组中的位置
//其前提是要求数组中的数据分布比较均匀
mid = min + (key-arr[min])/(arr[max]-arr[min]) * (max-min);
```

二分查找改进之斐波那契查找

```
mid = min + 黄金分割点左边变长度 - 1;
```

- 二分查找、插值查找、斐波那契查询的特点
  - 相同点
    - 都是通过不断的缩小范围来查找对应的数据
  - 不同点
    - 计算mid的方式不一样
  - 二分查找
    - mid每次指向范围的中间位置
  - 插值查找
    - mid尽可能的靠近要查找的数据，但是要求数据尽可能的分布均匀
  - 斐波那契查找
    - 根据黄金分割点来计算mid指向的位置

分块查找

- 分块的原则
  - 前一块中的最大数据，小于后一块中所有的数据（块内无序，块间有序）
  - 块数数量一般等于数字的个数开根号
- 核心思路
  - 先确定要查找的元素在那一块，然后在块内挨个查找

```
//利用分块查找的原理，查询number的索引
private static int getIndex(Block[] blocks, int[] arr, int number) {
    //确定在哪个块中
    int indexBlock = findIndexBlock(blocks, number);
    System.out.println(indexBlock);
    if(indexBlock == -1)
        return -1;
    //获取块中的起始索引与结束索引
    int startIndex = blocks[indexBlock].getStartIndex();
    int endIndex = blocks[indexBlock].getEndIndex();
    //遍历大数组的起始索引与结束索引中的数据（需要包括结束索引对应的数据）
    for (int i = startIndex; i <= endIndex; i++) {
        if(arr[i] == number)
            return i;
    }
    return -1;
}
//定义一个方法，来确定number在哪一个块中
public static int findIndexBlock(Block[] blocks, int number){
```

```
        for (int i = 0; i < blocks.length; i++) {
            if(number <= blocks[i].getMax())
                return i;
        }
        return -1;
    }
}
//建立块对象用来描述块特征
class Block{
    private int max;
    private int startIndex;
    private int endIndex;

    public int getMax() {
        return max;
    }

    public void setMax(int max) {
        this.max = max;
    }

    public int getStartIndex() {
        return startIndex;
    }

    public void setStartIndex(int startIndex) {
        this.startIndex = startIndex;
    }

    public int getEndIndex() {
        return endIndex;
    }

    public void setEndIndex(int endIndex) {
        this.endIndex = endIndex;
    }

    public Block() {
    }

    public Block(int max, int startIndex, int endIndex) {
        this.max = max;
        this.startIndex = startIndex;
        this.endIndex = endIndex;
    }
}
```

扩展的分块查找（无规律的数据）

```
//数据无规律，即块并不是按照由小到大的顺序，那么也可以将每块设置成单独的区间，只要将块中的min和max都是相互独立的即可
class BlockSearch {
    private int min;
    private int max;
    private int startIndex;
    private int endIndex;
}
```

扩展的分块查找（查找的过程还需要添加数据）

- 思路：例如随机要从0~1000中存储100个数据且要求数据不重复，则可以使用下列方法：
  - 将待存储的100个数据分成10个块，分别是存储0~100，101~200,.....901~1000的区域
  - 如果随机到的数据在对应的区域中，则先在该块中判断是否已经存储了该值，如果有则不存储，没有则在该块中继续存储
- 总结：这属于简单的哈希查找

排序算法

冒泡排序

```
public static void bubbleSort(int[] arr){
    //外循环中表示跑的趟数，n个数据只需要跑n-1趟
    for (int i = 0; i < arr.length-1; i++) {
        //每次循环的过程中应该比上次比较的数据量少1
        //例如十个数据，第一趟相互比较9次，第二趟就只需要比较8次
        for (int j = 0; j < arr.length-1-i; j++) {
            if(arr[j] > arr[j+1]){
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

选择排序

- 从0索引开始，拿着每一个索引上的元素跟后面的元素依次比较，小的放前面，大的放后面，以此类推

```
public static void selectSort(int[] arr){
    //外循环表示这一轮要拿索引为i的数据去跟后面数据判断比较
    for (int i = 0; i < arr.length-1; i++) {
        //内循环表示要循环把索引为i后面的数据依次和索引i数据本身比较
        for (int j = i+1; j < arr.length; j++) {
            if(arr[i] > arr[j]){
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

插入排序

- 将0索引的元素到N索引的元素看做是有序的，把N+1索引的元素到最后一个当成是无序的。遍历无序的数据，将遍历到的元素插入有序序列中适当的位置，如遇到相同数据，插在后面
- N的范围：0~最大索引

```
public static void insertSort(int[] arr){
    //1，找到无序的数据是从那个索引开始的
    int startIndex = -1;
    for (int i = 0; i < arr.length; i++) {
        if(arr[i] > arr[i+1]){
            startIndex = i+1;
            break;
        }
    }
    //2，遍历从startIndex开始到最后一个元素的数据，即无序的数据
    for (int i = startIndex; i < arr.length; i++) {
        //记录要插入数据的索引
        int index = i;
        //如果索引不是起始位置同时无序数据仍然大于有序数据则继续交换
        while(index>0 && arr[index] < arr[index-1]){
            int temp = arr[index];
            arr[index] = arr[index-1];
            arr[index-1] = temp;
            //交换完毕之后索引减1继续判断
            index--;
        }
    }
}
```

递归算法

- 方法本身调用方法的过程就是递归
- 递归的注意点：递归一定要有出口，否则就会出现溢出
- 递归算法作用：把一个复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解
- 书写递归的核心
    - 找出口：什么时候不再调用方法
    - 找规律：如何把大问题变成规模较小的问题
- 递归的心得
    - 方法内部再次调用方法的时候，参数必须更加的靠近出口

快速排序

- 第一轮：把0索引的数组作为基准数，确定基准数在数组中正确的位置，比基准数小的全部在左边，比基准数大的全部在右边
    1. 将排序范围中的第一个数字作为基准数，再定义两个变量start,end
    2. start从前往后找比基准数大的，end从后往前找比基准数小的
    3. 找到之后交换start和end指向的元素，并循环这一过程，直到start和end处于同一个位置，该位置是基准数在数组中应存入的位置，再让基准数归位
    4. 归位后的效果:基准数左边的，比基准数小，基准数右边的，比基准数大
    5. 之后再分左右不断的递归使其归位，直至起始索引大于结束索引则退出递归

```
/*
```

```
第一个参数：待排序的数组
第二个参数：要排序的起始索引
第三个参数：要排序的结束索引
*/
public static void quickSort(int[] arr, int startIndex, int endIndex){
    //记录要查找的范围
    int start = startIndex;
    int end = endIndex;

    if(start > end)
        return;

    //记录基准数
    int baseNumber = arr[startIndex];

    //利用循环找到要交换的数字
    //其中的顺序也只能先end后判断start
    while(start != end){
        //利用end从后往前找，找比基准数小的数字
        while(true){
            if(end<=start || arr[end] < baseNumber)
                break;
            end--;
        }
        //利用start从前往后找，找比基准数大的数字
        while(true){
            if(end<=start || arr[start] > baseNumber)
                break;
            start++;
        }
        //把end和start指向的元素进行交换
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
    }
    //上面执行完毕之后，start和end就会指向基准数应该出现的索引
    //其左边小于基准数，右边大于基准数
    //基准数归位
    int temp = arr[startIndex];
    arr[startIndex] = arr[start];
    arr[start] = temp;
    //确定6左边的范围，重复上述行为
    quickSort(arr, startIndex, end-1);
    //确定6右边的范围，重复上述行为
    quickSort(arr, start+1, endIndex);
}
```

Arrays

- `public static String toString(数组)`
- 把数组拼接称一个字符串
- `public static int binarySearch(数组，查找的元素)`
    - 二分查找法
    - 细节：二分查找的前提是数组中的元素必须有序且必须是升序
    - 细节：如果数据存在则返回真是的索引，如果不存在则返回-插入点-1，-1是为了不返回-0

- `public static int[] copyOf(原数组，新数组长度)`
  - 拷贝数组
  - 参数一是老数组，参数二是新数组长度
  - 如果新数组长度小于新数组长度会部分拷贝
  - 如果新数组长度等于新数组长度会完全拷贝
  - 如果新数组长度大于新数组长度会补上默认初始值
- `public static int[] copyOfRange(原数组，起始索引，结束索引)`
  - 拷贝数组（指定范围）
  - 细节：包头不包尾
- `public static void fill(数组，元素)`
  - 填充数组
- `piblic static void sort(数组)`
  - 按照默认方式进行数字排序
  - 默认情况下，进行的是 升序排序，使用的是快速排序方法
- `public static void sort(数组，排序规则)`
  - 按照指定规则排序
  - 按照这种方法，第一个参数是数组，第二个参数是一个接口，所以传递的时候需要传递接口的实现类对象，但是因为只需要使用一次，所以不需要单独的写一个实现类，直接采取匿名内部类的方法即可
  - 底层原理:
    - 利用插入排序+二分查找的方式进行排序的
    - 默认把0索引的数据当做是有序的序列，1索引到最后认为是无序的序列
    - 遍历无序的序列得到里面的每一个元素，假设当前遍历得到的元素是A元素，把A往有序序列中进行插入，在插入的时候，是利用二分查找确定A元素的插入点
    - 拿着A元素，跟插入点的元素进行比较，比较的规则就是compare方法的方法体
    - 如果方法的返回值是负数，拿着A继续跟前面的数据进行比较
    - 如果方法的返回值是正数，拿着A继续跟后面的数据进行比较
    - 如果方法的返回值是0，也拿着A跟后面的数据进行比较，直到能确定A的最终位置为止

```
/*
参数o1：表示无序排列中得到的每一个元素
参数o2：有序序列中的元素
*/
Arrays.sort(arr, new Comparator<Integer>() ){
    @Override
    public int compara(Integer o1, Integer o2){
        return o1-o2;//此时是升序
        return o2-o1;//此时是降序
    }
}
```

Lambda

函数式编程

- 函数式编程是一种思想特点
- 面向对象：先找对象，让对象做事情
- 函数式编程思想忽略面向对象的复杂语法，强调做什么，而不是谁去做

Lambda表达式的标准格式

- Lambda是JDK8开始后的一种新语法形式
- Lambda变大时可以用来简化匿名内部类的书写
- Lambda百年大是只能简化函数式接口的匿名内部类的写法
- 如何判断是函数式接口：
    - 有且仅有一个抽象方法的接口叫做函数式接口，接口上方可以加@FunctionalInterface注解

```
//原匿名内部类的实现过程
Arrays.sort(arr, new Comparator<Integer>() ){
    @Override
    public int compara(Integer o1, Integer o2){
        return o1-o2;//此时是升序
        return o2-o1;//此时是降序
    }
}
//上述可以简化为
Arrays.sort(arr, (Integer o1, Integer o2)->{
        return o1-o2;//此时是升序
    }
}
```

- Lambda的好处
    - Lambda是一个匿名函数、我们可以把Lambda表达式理解为是一段可以传递的代码，它可以写出更简洁、更灵活的代码，作为一种更紧
      凑的代码风格，使Java语言表达能力得到了提升

Lambda表达式的省略写法

- lambda的省略规则：
    1. 参数类型可以省略不写
    2. 如果只有一个参数,参数类型可以省略,同时()也可以省略
    3. 如果Lambda表达式的方法体只有一行，大括号，分号，return可以省略不写，需要同时省略。

集合

集合体系结构

- collection（接口）
    - List（接口）
        - ArrayList（实现类）
        - LinkedList
        - Vector
    - Set（接口）
        - HashSet
            - LinkedHasnSet
        - TreeSet
- List系列集合
    - 添加的元素是有序（存取的顺序）、可重复、有索引
- Set系列集合
    - 添加的元素是无序、不重复、无索引

- 如果往List系列集合中添加数据，那么方法永远返回true，因为List允许重复
- 如果往Set系列集合中添加数据，如果要添加的数据已经存在则返回false，因为Set不允许重复

Collection

- Collection是单列集合的祖宗接口，它的功能是全部单列集合都可以继承使用的
- Collection是一个接口，因此不能直接创建他的对象，所以学习他的方法时只能创建它实现类的对象

```
//举例
Collection<String> coll = new ArrayList<>();
```

- 常用方法

```
public boolean add(E e)//添加数据

public void clear()//清空

//因为collection中是共性的方法，为了满足Set，因此不能使用索引删除
//删除成功则返回true，否则false
public boolean remove()//删除

//底层判断时候依赖的是Object类中的equals方法，其比较的是地址值，因此如果要判断自定义对象的话，就需要重写方法让其比较属性值
public boolean contains()//判断元素是否包含在集合中

public boolean isEmpty()//判断集合是否为空

public int size()//获取集合的长度
```

Collection的遍历方式

- 迭代器遍历

```
Iterator<E> iterator()//获取迭代器对象
boolean hasNext()//判断当前指向的位置是否有元素
E next()//获取当前指向的元素并移动指针
remove()//删除当前指向的元素
```

```
//创建迭代器对象，默认会有指针指向集合中的第一个元素的位置
Iterator<string> it = list.iterator();
//判断当前位置是否还有元素
while(it.hasNext()){
    //获取元素并移动指针
    string str = it.next();
    //输出获取到的数据
    system.out.println(str);
}
```

  - 细节：
    - 迭代器遍历完毕之后，迭代器指针不会复位
    - 循环中只能用一次next方法
    - 迭代器**遍历时**，不能用**集合的方法**进行增加或者删除，可以使用迭代器的方式进行删除
    - 如果要再次遍历集合那么需要重新申请新得迭代器对象

- 增强for遍历
  - 增强for的底层就是迭代器，为了简化迭代器的代码书写的
  - 它是JDK5之后出现的，其内部原理就是一个Iterator迭代器
  - 所有的**单列集合**和**数组**才能用增强for进行遍历
  - 格式

```
//变量名就是一个第三方变量，并不会影响集合中的数据本身
for(元素的数据类型 变量名:数组或者集合){
    //要执行的操作
}
```

  - 举例

```
for(String s : coll){
    sout(s);
}
```

- Lambda表达式遍历
  - forEach

```
default void forEach(Consumer<? super T> action)

//利用匿名内部类的方式
//Consumer接口中会依次得到集合中的每一个数据然后传递给下面的accept方法
coll.forEach(new Consumer<String>(){
    @Override
    public void accept(String s){
        //s依次表示集合中的每一个数据
        sout(s);
    }
})

//Lambda方式
//Consumer接口是一个函数式接口，所以可以使用Lambda方式书写
coll.forEach(s -> sout(s));
```

- 三种通用的遍历方式:
  - 在遍历的过程中需要删除元素，请使用迭代器
  - 仅仅想遍历，那么使用增强for或Lambda表达式

List集合

- List集合的特点
  - 有序:存和取的元素顺序一致
  - 索引:可以通过索引操作元素
  - 可重复:存储的元素可以重复
  - Collection的方法List都继承了
  - List集合因为有索引，所以多了很多索引操作的方法
- List集合的特有方法

```
//把元素添加在指定索引处，原来索引上的数据会依次往后移
void add(int index, E element)//在指定位置添加元素

//这是List中特有的根据索引删除
//在调用方法的时候，如果方法出现了重载现象，那么会优先调用实参跟形参类型一致的那个方法
E remove(int index)//删除对应索引的元素

E set(int index, E element)//修改指定索引处的元素

E get(int index)//返回指定索引处的元素
```

- List集合的遍历方式
  - 迭代器遍历（与collection一样）
  - Lambda表达式（与collection一样）
  - 增强for（与collection一样）
  - 普通for循环（使用for与get结合的方法）
  - 列表迭代器

```
//创建列表迭代器对象
ListIterator<String> it = list.listInerator();
//如果当前指针还存在列表元素
while(it.hasNext()){
    String str = it.next();//将当前指针的元素赋值给str并指针后移
    if(str == "aaa")
        it.add("xxx");//将指定的元素插入列表
    if(str == "bbb")
        it.remove();//从列表移除当前元素
}
```

  - 在遍历的过程中需要删除元素，请使用迭代器
  - 在遍历的过程中需要添加元素，请使用列表迭代器
  - 仅仅想遍历，那么使用增强for或Lambda表达式
  - 如果遍历的时候想操作索引，可以用普通for

ArrayList集合底层原理

- ArrayList集合底层原理
  - 利用空参创建的集合，在底层创建一个默认长度为0的数组
  - 添加第一个元素时，底层会创建一个新的长度为10的数组
  - 存满时，会扩容1.5倍
  - 如果一次添加多个元素，1.5倍还放不下，则新创建数组的长度以实际为准
-