语言基础

1 类型转换

1. 小容量向大容量转换，称为**自动类型转换**，容量从小到大排序：
   > byte < short ，char < int < long < float < double
2. 大容量转换成小容量，叫做**强制类型转换**，需要加强制类型转换符，程序才能编译通过，但是在运行阶段可能会损失精度，所以谨慎使用。如：

```
double d = 3.14;
int i = (int) d;
```

3. byte，short，char混合运算的时候，各自先转换成int类型再做运算。比int大的就不会转。
4. 多种数据类型混合运算，先转换成容量最大的那种类型再做运算。

2 运算符

1. **逻辑运算符**

逻辑运算符	解释
&	逻辑与（两边的算子都是true，结果才是true）
!	逻辑非（取反，！false就是true，！true就是假，这是一个单目运算符）
^	逻辑异或（两边的算子只要不一样，结果就是true）
&&	短路与（第一个表达式执行结果是false，会发生短路与）
|	逻辑或（两边的算子只要有一个是true，结果就是true）
||	短路或（第一个表达式执行结果是true，会发生短路或）

1. **字符串连接运算符**
   +运算符在java语言中当中有两个作用

- 加法
- 字符串连接。可以将字符串和非字符串数据直接连接起来

2. 三元运算符/三目运算符/条件运算符

> 布尔表达式？表达式1：表达式2

如：

```
String result = a > c ? "a大于c" : "a小于等于c";
```

3 其他

- 所有浮点字面值，比如3.14，都是double型，如果想把这个浮点字面值当作float来处理，应该在其后面添加f/F，如3.14f
- 变量的作用域只要记住一句话：出了大括号就不认识了。

常用类

1 泛型集合

1.1 Collection - 单列集合

Collection是所有单列集合的祖宗类，有一些通用的功能。使用时候需要`import java.util.Collection;`

考试的时候可以直接`import java.util.*`

方法名	说明
add(E e)	向集合中添加元素
addAll(Collection<? extends E> c)	添加一个集合中的所有元素
remove(Object o)	移除指定元素
removeAll(Collection<?> c)	移除集合中所有与指定集合相同的元素
clear()	清空集合中的所有元素

方法名	说明
isEmpty()	判断集合是否为空
size()	返回集合中元素的个数
toArray()	将集合转换为数组
contains(Object o)	判断集合是否包含指定元素
containsAll(Collection<?> c)	判断集合是否包含指定集合的所有元素
iterator()	返回集合的迭代器
retainAll(Collection<?> c)	只保留集合中与指定集合相同的元素

第一种遍历，使用Iterator

对于iterator，当然也需要`import java.util.Iterator`，迭代器类有两个常用的函数：

1. `next()`，返回当前索引的值并且使得指针加一，类似于`it++`
2. `hasNext()`，返回是否可以往后移
3. `remove()`，删除当前迭代器所指的元素
   用该方法删除元素不会出现并发问题

第二种遍历，增强for循环，代码如下：

```
for (String s : list) {
    System.out.println(s);
}
```

第三种遍历，使用forEach方法，可以使用lambda简化

```
//最原始的使用
list.forEach(new Consumer<String>() {
    public void accept(String s) {
        System.out.println(s);
    }
});
//简化
list.forEach(s->System.out.println(s));
//再简化
list.forEach(System.out::println);
```

1.1.1 List集合

有序，可重复，有索引。Collection的所有功能都有。

需要`import java.util.List`

1. 定义集合

```
List<String> list=new ArrayList<>();
//添加数据
list.add("Hello");
list.add("World");
list.add(2,"whee!");//List特殊的add方式
//直接打印ArrayList
System.out.println(list);//[Hello, World]
```

2. 查看数据

```
String s1=list.get(0);
System.out.println(s1);
System.out.println(list.get(1));
```

3. 删除数据。有两种使用方式。

```
String removed = list.remove(0);//返回被删除数据的具体值
System.out.println(removed);//Hello
```

4. 修改数据

```
list.set(0,"Java");
System.out.println(list.get(0));//Java
```

5. `toArray()`：转为数组，有两种用法。
   第一种是直接使用Object来接收。`Object[] arr = list.toArray();`
   第二种是指定类型。`String[] arr=list.toArray(new String[0]);`
   注意，两种方法后续使用也有区别，第一种需要使用Object来接收值，第二种使用String。
6. `indexOf(Object o)`：返回元素第一次出现的位置
7. `lastIndexOf(Object o)`：返回元素最后一次出现的位置
8. `listIterator()` 和 `listIterator(int index)`：返回 ListIterator，支持双向遍历、previous、set、add 等
9. `sort(Comparator<? super E> c)`：就地排序（Java 8+）

1.1.1.1 ArrayList

基于数组存储数据，有如下特点：

1. 查询速度快，查询任意数据耗时相同
2. 增删效率低

1.1.1.2 LinkedList

基于双链表存储数据，有如下的特点：

1. 查询速度慢
2. 增删相对较快

新增了许多首位操作特有的办法

方法名	作用说明
addFirst(E e)	在头部添加元素
addLast(E e)	在尾部添加元素
getFirst()	获取头部元素
getLast()	获取尾部元素
removeFirst()	移除并返回头部元素

方法名	作用说明
removeLast()	移除并返回尾部元素
offerFirst(E e)	在头部插入元素（队列语义）
offerLast(E e)	在尾部插入元素（队列语义）
pollFirst()	移除并返回头部元素
pollLast()	移除并返回尾部元素
peekFirst()	查看头部元素，不移除
peekLast()	查看尾部元素，不移除
push(E e)	作为栈顶压入元素
pop()	作为栈顶弹出元素
descendingIterator()	反向迭代器（从尾到头）

1.1.2 Set集合

无序（添加数据的顺序和获取数据的顺序不一致），无索引，不重复。由于set无索引，所以大部分功能都来自于Collection。

1.1.2.1 HashSet

基于哈希表实现的一个Set容器

如果T设置成自定义的类，则不会自动去重，因为容器无法识别两个类是否相同。需要在类内重写一些函数。

```java
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Student student = (Student) o;
    return age == student.age && Objects.equals(name, student.name);
}

@Override
public int hashCode() {
    return Objects.hash(name, age);
}
```

1.1.2.2 LinkedHashSet

有序的集合

1.1.2.3 TreeSet

基于红黑树实现的集合，会自动排序，默认按照从小到大排序,如果是字符串，则按照字典序排序

1.2 Map - 双列集合

Map 是存储键值对（key-value）的集合，每个键唯一，值可重复。

1.2.1 基本特点

- **键值对结构**：每个元素包含一个键和一个值
- **键唯一性**：Map 中的键不能重复
- **值可重复**：不同的键可以对应相同的值
- **无序性**：大部分 Map 实现不保证顺序（除了 LinkedHashMap、TreeMap）

1.2.2 常用实现类

实现类	特点	适用场景
HashMap	无序，查询快，线程不安全	最常用的 Map 实现
LinkedHashMap	有序（插入顺序），查询快	需要保持插入顺序时

实现类	特点	适用场景
TreeMap	按键排序（自然顺序或自定义比较器）	需要按键排序时
Hashtable	线程安全，性能较低	多线程环境（已被 ConcurrentHashMap 替代）

1.2.3 基本操作

```
// 创建 Map
Map<String, Integer> map = new HashMap<>();

// 添加元素
map.put("Apple", 10);
map.put("Banana", 20);
map.put("Orange", 15);

// 获取元素
Integer count = map.get("Apple");  // 返回 10
Integer count2 = map.getOrDefault("Grape", 0);  // 如果不存在返回默认值 0

// 删除元素
map.remove("Banana");

// 检查键是否存在
boolean hasKey = map.containsKey("Apple");  // true
boolean hasValue = map.containsValue(10);     // true

// 获取大小
int size = map.size();  // 2

// 清空集合
map.clear();
```

1.2.4 遍历方式

1. 遍历键

```
for (String key : map.keySet()) {
    System.out.println(key + ": " + map.get(key));
}
```

2. 遍历值

```
for (Integer value : map.values()) {
    System.out.println(value);
}
```

3. 遍历键值对（推荐）

```
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

4. forEach 方法（Java 8+）

```
map.forEach((key, value) -> System.out.println(key + ": " + value));
```

1.2.5 常用方法总结

方法名	说明
put(K key, V value)	添加或更新键值对
get(Object key)	根据键获取值
getOrDefault(Object key, V defaultValue)	获取值或默认值
remove(Object key)	删除指定键的键值对
containsKey(Object key)	检查是否包含指定键
containsValue(Object value)	检查是否包含指定值
keySet()	返回所有键的集合
values()	返回所有值的集合

方法名	说明
entrySet()	返回所有键值对的集合
size()	返回键值对数量
isEmpty()	检查是否为空
clear()	清空所有键值对

1.2.6 实际应用示例

```
// 统计字符出现次数
String text = "hello world";
Map<Character, Integer> charCount = new HashMap<>();

for (char c : text.toCharArray()) {
    charCount.put(c, charCount.getOrDefault(c, 0) + 1);
}

// 结果: {h=1, e=1, l=3, o=2,  =1, w=1, r=1, d=1}
```

```
// 对象映射
Map<String, Person> personMap = new HashMap<>();
personMap.put("001", new Person("张三", 25));
personMap.put("002", new Person("李四", 30));

// 查找
Person person = personMap.get("001");
```

1.2.7 注意事项

1. **键的正确性**：作为键的对象应该正确实现 `equals()` 和 `hashCode()` 方法
2. **null 值**：HashMap 允许一个 null 键和多个 null 值
3. **线程安全**：多线程环境下使用 `ConcurrentHashMap` 或 `Collections.synchronizedMap()`
4. **性能考虑**：选择合适的 Map 实现类以获得最佳性能

2　集合工具类

我来详细解释Java中Arrays和Collections的区别：

3　Arrays和Collections的主要区别

3.1　基本概念

Arrays类：

- 位于`java.util`包中
- 是一个工具类，用于操作数组（包括基本类型数组和对象数组）
- 提供了静态方法来排序、搜索、比较、填充数组等

Collections类：

- 位于`java.util`包中
- 是一个工具类，用于操作集合（Collection框架）
- 提供了静态方法来排序、搜索、同步、不可修改集合等

3.2　操作对象不同

```
// Arrays操作数组
int[] array = {3, 1, 4, 1, 5};
Arrays.sort(array);                    // 排序数组
int index = Arrays.binarySearch(array, 4); // 二分查找

// Collections操作集合
List<Integer> list = new ArrayList<>(Arrays.asList(3, 1, 4, 1, 5));
Collections.sort(list);                // 排序集合
int index2 = Collections.binarySearch(list, 4); // 二分查找
```

3.3 功能对比表

功能	Arrays	Collections
排序	`sort()`	`sort()`
搜索	`binarySearch()`	`binarySearch()`
填充	`fill()`	`fill()`
复制	`copyOf()`	无直接对应
比较	`equals()`	无直接对应
转换	`asList()`	`toArray()`
同步	无	`synchronizedXXX()`
不可修改	无	`unmodifiableXXX()`
空集合	无	`emptyXXX()`
单元素集合	无	`singletonXXX()`

3.4 总结

特性	Arrays	Collections
操作对象	数组（基本类型和对象）	集合框架对象
方法类型	静态方法	静态方法
主要功能	数组操作（排序、搜索、复制等）	集合操作（排序、同步、包装等）
线程安全	不提供线程安全方法	提供synchronizedXXX()方法
不可修改	不支持	提供unmodifiableXXX()方法
特殊集合	无	提供emptyXXX()、singletonXXX()等

GUI编程

企业几乎不用java来写图形界面

java提供两套GUI编程包

1. AWT，在windows上使用，现在几乎不用
2. Swing，基于AWT，提供了更丰富的组件，不依赖于本地串口系统

以下部分均为swing的笔记

1 常用组件

- JFrame:窗口
- JPanel:用于组织其他组件的容器
- JButton:按钮组件
- JTextField:输入框
- JTable: 表格
- JLabel: 文本栏

2 布局管理器

目前常见的有四种

1. FlowLayout，流式布局管理

```
JLabel label3=new JLabel("请输入一元二次方程的系数c: ");
jf.add(label3);
```

2. BorderLayout，将容器划分为东、南、西、北、中五个区域

```
jf.add(new Button("请输入一元二次方程的系数a: "),BorderLayout.NORTH);
```

3. GridLayout，网格布局管理器

```
jf.setLayout(new GridLayout(2,3));
```

```
jf.add(new JButton("请输入一元二次方程的系数a: "));
```

4. BoxLayout,盒子布局，可以做竖向布局，配合panel使用

```
 JPanel panel=new JPanel();
```

```
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
panel.add(new JButton("Button 1"));
```

3　线程启动

```
SwingUtilities.invokeLater(new Runnable() {
        public void run() {
                new Main().setVisible(true);
        }
});

SwingUtilities.invokeLater(() -> {
        new Main().setVisible(true);
});
```

4　事件监听

常用的有两个事件监听器：

- 点击事件监听
- 按键事件监听

4.1　点击事件监听器ActionListener

```
clearButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        textA.setText("");
        textB.setText("");
        textC.setText("");
    }
});
```

4.2　按键事件监听器

```
//按键事件监听器
jf.addKeyListener(new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
        //每个键帽都有一个编号
        int keycode =e.getKeyCode();
        if (keycode == KeyEvent.VK_UP) {
            System.out.println("Up arrow key pressed");
        }
    }
})

//记得让窗口成为焦点
jf.requestFocus();
```

4.3　事件监听的三种写法

1. 使用匿名内部类

```
// 按钮事件处理，使用匿名内部类
solveButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            double a = Double.parseDouble(textA.getText());
            double b = Double.parseDouble(textB.getText());
            double c = Double.parseDouble(textC.getText());
            f equation = new f(a, b, c);
            String result = equation.cal();
            JOptionPane.showMessageDialog(jf, result, "求解结果", JOptionPanINFORMATION_MESSAGE);
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(jf, "请输入有效的数字", "输入错误", JOptionPanERROR_MESSAG
        }
    }
});
```

2. 使用lambda

```
// 使用 Lambda 表达式
clearButton.addActionListener(e -> {
    textA.setText("");
    textB.setText("");
    textC.setText("");
});
```

3. 事件源合而为一

5 示例代码

代码太长了，直接贴个文件得了

一元二次方程求解GUI

线程与并发编程

1 概念

线程（thread）是一个程序内部的一条执行流程

多线程是指从软硬件上实现的多条执行流程的技术（多条线程由CPU负责调度执行）

2 创建线程

有三种方法来创建多线程

2.1 继承Thread

1. 先定义出一个继承Thread的子类

```
class MyThread extends Thread{
    //重写run方法
    @Override
    public void run() {
        //线程要执行的代码
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread  running: " + i);
        }
    }
}
```

2. 然后再创建一个子类线程对象，然后通过start()函数来启动线程，注意并非run()。如果是调用run()函数，则会变成一个普通对象来调用，而不是启动一个新的线程。

```
    //创建线程对象
    MyThread t1 = new MyThread();
    //启动线程
    t1.start();
```

2.2 实现Runnable接口

1. 先定义一个线程任务类,这个类需要实现Runnable的接口

```
//定义一个线程任务类
class MyRunnable implements Runnable {
    @Override
    public void run() {
        //线程要执行的代码
        for (int i = 0; i < 10; i++) {
            System.out.println("Runnable running: " + i);
        }
    }
}
```

2. 创建线程任务类对象，然后把这个对象给一个线程对象

```
Runnable r=new MyRunnable();
Thread t2=new Thread(r);
t2.start();
```

这种方式可以继承其他的类，并且可以通过匿名内部类来简化写法：

```
Thread t2=new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Anonymous Runnable running: " + i);
        }
    }
});
t2.start();
```

可以进一步简化：

```
new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        System.out.println("Anonymous Runnable running: " + i);
    }
}).start();
```

2.3 实现Callable接口

前两个线程的创建方式的run()函数都是void类型，无法返回数据。注意，Callable来自java.util.concurrent.Callable，使用时需要import对应的包

1. 定义一个实现类实现Callable接口的call函数

```
class MyCallable implements Callable<String>{
    private int n;
    public MyCallable(int n){
        this.n = n;
    }
    public String call(){
        int sum = 0;
        for (int i = 0; i <= n; i++) {
            sum += i;
        }
        return String.valueOf(sum);
    }
}
```

2. 创建Callable对象并且将这个对象给真正的线程任务类对象FutureTask。其中，FutureTask来自`java.util.concurrent.FutureTask`。可以将FutureTask当作一个Runnable对象（事实上FutureTask继承了Runnable，可以通过多态来接受FutureTask对象）。

```
Callable<String> c1=new MyCallable(100);
FutureTask<String> f1=new FutureTask<>(c1);
//启动线程
new Thread(f1).start();
```

3. 通过FutureTask的get方法来取得返回数据

```
try {
    System.out.println("Callable result: " + f1.get());
} catch (Exception e) {
    e.printStackTrace();
}
```

最后得到的结果是等这个线程运行完之后的结果。

IO流

IO流用于处理文件的输入和输出操作。Java中的IO流分为字节流和字符流两大类。

使用IO流时需要导入：`import java.io.*;`

1 字节流

字节流以字节为单位读写数据，适合所有类型的文件（文本、图片、音频、视频等）。

1.1 FileInputStream - 字节输入流

用于从文件中读取数据。

基本使用步骤：

1. 创建字节输入流对象
2. 读取数据

3. 关闭流释放资源

方法一：一次读取一个字节

```
// 1. 创建流对象
FileInputStream fis = new FileInputStream("file.txt");

// 2. 读取数据
int b;
while ((b = fis.read()) != -1) {  // read()返回-1表示读取完毕
    System.out.print((char) b);
}

// 3. 关闭流
fis.close();
```

方法二：一次读取多个字节（推荐）

```
FileInputStream fis = new FileInputStream("file.txt");

byte[] buffer = new byte[1024];  // 定义缓冲区
int len;
while ((len = fis.read(buffer)) != -1) {  // len是实际读取的字节数
    System.out.print(new String(buffer, 0, len));
}

fis.close();
```

使用try-with-resources自动关闭流（推荐写法）

```
try (FileInputStream fis = new FileInputStream("file.txt")) {
    byte[] buffer = new byte[1024];
    int len;
    while ((len = fis.read(buffer)) != -1) {
        System.out.print(new String(buffer, 0, len));
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

1.2 FileOutputStream - 字节输出流

用于将数据写入文件。

基本用法：

```
// 创建输出流，会覆盖原文件内容
FileOutputStream fos = new FileOutputStream("output.txt");

// 写入单个字节
fos.write(97);  // 写入字符'a'

// 写入字节数组
byte[] bytes = "Hello World".getBytes();
fos.write(bytes);

// 写入字节数组的一部分
fos.write(bytes, 0, 5);  // 只写入"Hello"

// 关闭流
fos.close();
```

追加模式（不覆盖原文件）：

```
// 第二个参数为true表示追加模式
FileOutputStream fos = new FileOutputStream("output.txt", true);
fos.write("追加的内容".getBytes());
fos.close();
```

推荐写法（自动关闭）：

```
try (FileOutputStream fos = new FileOutputStream("output.txt", true)) {
    fos.write("Hello World\n".getBytes());
} catch (IOException e) {
    e.printStackTrace();
}
```

1.3 文件复制示例

```
try (FileInputStream fis = new FileInputStream("source.jpg");
     FileOutputStream fos = new FileOutputStream("copy.jpg")) {

    byte[] buffer = new byte[1024];
    int len;
    while ((len = fis.read(buffer)) != -1) {
        fos.write(buffer, 0, len);
    }
    System.out.println("文件复制完成");
} catch (IOException e) {
    e.printStackTrace();
}
```

1.4 常用方法总结

FileInputStream：

方法名	说明
read()	读取一个字节，返回-1表示读取完毕
read(byte[] b)	读取多个字节到数组，返回实际读取的字节数

方法名	说明
available()	返回可读取的剩余字节数
close()	关闭流释放资源

FileOutputStream：

方法名	说明
write(int b)	写入一个字节
write(byte[] b)	写入字节数组
write(byte[] b, int off, int len)	写入字节数组的一部分
flush()	刷新缓冲区
close()	关闭流释放资源

2 字符流

字符流以字符为单位读写数据，只适合文本文件。字符流会自动处理字符编码，避免乱码问题。

2.1 FileReader - 字符输入流

用于从文本文件中读取字符数据。

方法一：一次读取一个字符

```
try (FileReader fr = new FileReader("file.txt")) {
    int c;
    while ((c = fr.read()) != -1) {
        System.out.print((char) c);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

方法二：一次读取多个字符（推荐）

```
try (FileReader fr = new FileReader("file.txt")) {
    char[] buffer = new char[1024];
    int len;
    while ((len = fr.read(buffer)) != -1) {
        System.out.print(new String(buffer, 0, len));
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

2.2 FileWriter - 字符输出流

用于将字符数据写入文本文件。

基本用法：

```
try (FileWriter fw = new FileWriter("output.txt")) {
    // 写入单个字符
    fw.write('A');

    // 写入字符串
    fw.write("Hello World");

    // 写入字符数组
    char[] chars = {'J', 'a', 'v', 'a'};
    fw.write(chars);

    // 写入字符串的一部分
    fw.write("Hello World", 0, 5);  // 只写入"Hello"

    // 换行
    fw.write("\n");

} catch (IOException e) {
    e.printStackTrace();
}
```

追加模式：

```
try (FileWriter fw = new FileWriter("output.txt", true)) {
    fw.write("追加的内容\n");
} catch (IOException e) {
    e.printStackTrace();
}
```

2.3 BufferedReader - 缓冲字符输入流

提供缓冲功能，提高读取效率，并且可以按行读取。

按行读取（非常常用）：

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {  // readLine()读取一行，不包含换行符
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

2.4 BufferedWriter - 缓冲字符输出流

提供缓冲功能，提高写入效率。

基本用法：

```
try (BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"))) {
    bw.write("第一行内容");
    bw.newLine();  // 写入换行符（跨平台）
    bw.write("第二行内容");
    bw.newLine();
    bw.write("第三行内容");
} catch (IOException e) {
    e.printStackTrace();
}
```

2.5 文本文件复制示例

```
try (BufferedReader br = new BufferedReader(new FileReader("source.txt"));
     BufferedWriter bw = new BufferedWriter(new FileWriter("copy.txt"))) {

    String line;
    while ((line = br.readLine()) != null) {
        bw.write(line);
        bw.newLine();
    }
    System.out.println("文件复制完成");
} catch (IOException e) {
    e.printStackTrace();
}
```

2.6 常用方法总结

FileReader：

方法名	说明
read()	读取一个字符，返回-1表示读取完毕
read(char[] cbuf)	读取多个字符到数组，返回实际读取的字符数
close()	关闭流释放资源

FileWriter：

方法名	说明
write(int c)	写入一个字符
write(String str)	写入字符串
write(char[] cbuf)	写入字符数组
write(String str, int off, int len)	写入字符串的一部分
flush()	刷新缓冲区
close()	关闭流释放资源

BufferedReader：

方法名	说明
read()	读取一个字符
read(char[] cbuf)	读取多个字符到数组
readLine()	读取一行文本，不包含换行符
close()	关闭流释放资源

BufferedWriter：

方法名	说明
write(int c)	写入一个字符
write(String str)	写入字符串
write(char[] cbuf)	写入字符数组
newLine()	写入换行符（跨平台）
flush()	刷新缓冲区
close()	关闭流释放资源

3　字节字符转换流

字节字符转换流是连接字节流和字符流的桥梁，主要用于处理编码转换问题。

3.1　InputStreamReader - 字节输入流转字符输入流

将字节输入流转换为字符输入流，可以指定字符编码。

基本用法：

```
// 使用默认编码
try (InputStreamReader isr = new InputStreamReader(new FileInputStream("file.txt"));
     BufferedReader br = new BufferedReader(isr)) {

    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}

// 指定编码（推荐）
try (InputStreamReader isr = new InputStreamReader(
        new FileInputStream("file.txt"), "UTF-8");
     BufferedReader br = new BufferedReader(isr)) {

    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

3.2　OutputStreamWriter - 字符输出流转字节输出流

将字符输出流转换为字节输出流，可以指定字符编码。

基本用法：

```
// 使用默认编码
try (OutputStreamWriter osw = new OutputStreamWriter(
        new FileOutputStream("output.txt"));
     BufferedWriter bw = new BufferedWriter(osw)) {

    bw.write("Hello, 世界!");
    bw.newLine();
    bw.write("中文内容测试");
} catch (IOException e) {
    e.printStackTrace();
}

// 指定编码（推荐）
try (OutputStreamWriter osw = new OutputStreamWriter(
        new FileOutputStream("output.txt"), "UTF-8");
     BufferedWriter bw = new BufferedWriter(osw)) {

    bw.write("Hello, 世界!");
    bw.newLine();
    bw.write("指定UTF-8编码写入");
} catch (IOException e) {
    e.printStackTrace();
}
```

3.3 实际应用场景

3.3.1 读取不同编码的文件

```
public static void readFileWithEncoding(String filePath, String encoding) {
    try (InputStreamReader isr = new InputStreamReader(
            new FileInputStream(filePath), encoding);
         BufferedReader br = new BufferedReader(isr)) {

        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// 使用示例
readFileWithEncoding("chinese.txt", "GBK");    // 读取GBK编码文件
readFileWithEncoding("unicode.txt", "UTF-8");   // 读取UTF-8编码文件
```

3.3.2 文件编码转换

```
public static void convertEncoding(String sourceFile, String targetFile,
                                   String sourceEncoding, String targetEncoding) {
    try (InputStreamReader isr = new InputStreamReader(
            new FileInputStream(sourceFile), sourceEncoding);
         BufferedReader br = new BufferedReader(isr);
         OutputStreamWriter osw = new OutputStreamWriter(
            new FileOutputStream(targetFile), targetEncoding);
         BufferedWriter bw = new BufferedWriter(osw)) {

        String line;
        while ((line = br.readLine()) != null) {
            bw.write(line);
            bw.newLine();
        }
        System.out.println("编码转换完成");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// 使用示例：GBK转UTF-8
convertEncoding("gbk_file.txt", "utf8_file.txt", "GBK", "UTF-8");
```

3.4 常用编码格式

编码名称	说明	特点
UTF-8	Unicode变长编码	国际通用，支持所有字符
GBK	中文编码	主要用于简体中文
ISO-8859-1	西欧编码	单字节编码
ASCII	美国标准信息交换码	7位编码，只支持英文字符

类的进阶设计

1　匿名内部类

虽然是“类”，但是其实是个没有声明名字的对象。

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello from a thread!");
    }
}).start();
```

一般来讲接口需要有一个类来继承实现，但是匿名内部类可以直接用接口的“类名”来创建：

```
interface myinf{
    void cmp();
}

new myinf() {
    @Override
    public void cmp() {
        System.out.println("Anonymous class implementing myinf");
    }
}.cmp();
```

2　lambda表达式

Lambda表达式是Java 8引入的一个重要特性，它允许我们将函数作为方法参数传递，或者将代码作为数据对待。Lambda表达式提供了一种简洁、函数式的编程方式，**主要用于简化匿名内部类的写法**。

2.1　Lambda表达式的定义和核心目的

Lambda表达式本质上是一个匿名函数，它没有名称，但有参数列表、函数体和返回类型。Lambda

表达式的核心目的是：

- 简化代码：减少冗余的匿名内部类代码
- 函数式编程：支持将函数作为一等公民进行传递
- 提高可读性：使代码更加简洁明了

2.2 标准语法结构

Lambda表达式的基本语法结构：

```
(参数列表) -> { 函数体 }
```

- **参数列表**：指定Lambda表达式的参数，可以省略参数类型（类型推断）
- **箭头操作符**：`->` 分隔参数列表和函数体
- **函数体**：Lambda表达式要执行的代码，可以是一个表达式或代码块

语法示例：

```
// 无参数，无返回值
() -> System.out.println("Hello Lambda")

// 单个参数，无返回值
(name) -> System.out.println("Hello, " + name)
// 参数类型可省略
name -> System.out.println("Hello, " + name)

// 多个参数，有返回值
(a, b) -> a + b
// 当函数体只有一条语句时，return关键字可省略
(a, b) -> { return a + b; }

// 多行语句，需要使用代码块
(x, y) -> {
    int sum = x + y;
    int product = x * y;
    return sum + product;
}
```

2.3　函数式接口的原理及与Lambda的关联

函数式接口是指只有一个抽象方法的接口。Lambda表达式需要函数式接口的支持，因为Lambda表达式本质上是函数式接口中抽象方法的具体实现。

@FunctionalInterface注解：

`@FunctionalInterface`是一个注解，用于标识一个接口为函数式接口。它有两个作用：

1. 编译器检查：确保接口只有一个抽象方法，如果有多个，编译器会报错
2. 文档说明：明确表明该接口设计用于Lambda表达式

示例：

```
@FunctionalInterface
interface MyFunctionalInterface {
    // 唯一的抽象方法
    void doSomething(String str);
    // 可以包含默认方法
    default void defaultMethod() {
        System.out.println("This is a default method");
    }
    // 可以包含静态方法
    static void staticMethod() {
        System.out.println("This is a static method");
    }
    // 可以包含从Object类继承的方法
    @Override
    boolean equals(Object obj);
}

// 使用Lambda表达式实现函数式接口
MyFunctionalInterface myInterface = str -> System.out.println("Doing something with: " + str);
myInterface.doSomething("Lambda");
```

2.4 典型应用场景

2.4.1 集合遍历

```
List<String> list = Arrays.asList("Apple", "Banana", "Orange");

// 传统方式
for (String fruit : list) {
    System.out.println(fruit);
}

// Lambda方式
list.forEach(fruit -> System.out.println(fruit));

// 方法引用方式
list.forEach(System.out::println);
```

2.4.2 事件处理

```
// 按钮点击事件
JButton button = new JButton("Click me");

// 传统方式
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
    }
});

// Lambda方式
button.addActionListener(e -> System.out.println("Button clicked"));
```

2.4.3 线程创建

```
// 传统方式
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread is running");
    }
}).start();

// Lambda方式
new Thread(() -> System.out.println("Thread is running")).start();
```

2.5 实际代码示例

2.5.1 无参数无返回值

```
// 定义函数式接口
@FunctionalInterface
interface RunnableTask {
    void run();
}

public class LambdaExample1 {
    public static void main(String[] args) {
        // 使用Lambda表达式
        RunnableTask task = () -> {
            System.out.println("Task is running");
            System.out.println("Task completed");
        };

        task.run();
    }
}
```

2.5.2　单参数无返回值

```
// 定义函数式接口
@FunctionalInterface
interface Consumer<T> {
    void accept(T t);
}

public class LambdaExample2 {
    public static void main(String[] args) {
        // 使用Lambda表达式
        Consumer<String> printer = str -> System.out.println("Printing: " + str);

        printer.accept("Hello Lambda");
    }
}
```

2.5.3　多参数有返回值

```
// 定义函数式接口
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

public class LambdaExample3 {
    public static void main(String[] args) {
        // 加法
        Calculator addition = (a, b) -> a + b;
        System.out.println("10 + 5 = " + addition.calculate(10, 5));  // 15

        // 乘法
        Calculator multiplication = (a, b) -> a * b;
        System.out.println("10 * 5 = " + multiplication.calculate(10, 5));  // 50

        // 复杂计算
        Calculator complex = (a, b) -> {
            int sum = a + b;
            int product = a * b;
            return sum + product;
        };
        System.out.println("Complex(10, 5) = " + complex.calculate(10, 5));  // 65
    }
}
```

2.5.4 泛型函数式接口

```
// 定义泛型函数式接口
@FunctionalInterface
interface Function<T, R> {
    R apply(T t);
}

public class LambdaExample4 {
    public static void main(String[] args) {
        // String to Integer
        Function<String, Integer> stringToInt = Integer::parseInt;
        System.out.println("String '123' to Integer: " + stringToInt.apply("123"));

        // Integer to String
        Function<Integer, String> intToString = Object::toString;
        System.out.println("Integer 123 to String: " + intToString.apply(123));

        // 自定义转换
        Function<String, String> toUpperCase = str -> str.toUpperCase();
        System.out.println("Convert 'hello' to uppercase: " + toUpperCase.apply("hello"));
    }
}
```

2.6 变量捕获规则和this关键字特性

2.6.1 变量捕获规则

Lambda表达式可以访问外部变量，但有以下限制：

1. **访问final或等效final变量**：Lambda表达式可以访问外部final变量或事实上是final的变量（未被重新赋值）

```java
public class VariableCapture {
    public static void main(String[] args) {
        final int x = 10;  // 显式final
        int y = 20;        // 事实上final（未被重新赋值）

        Runnable r = () -> {
            System.out.println("x = " + x);  // 合法
            System.out.println("y = " + y);  // 合法
        };

        // y = 30;  // 如果取消注释，会导致Lambda表达式编译错误
        r.run();
    }
}
```

2. **不能修改外部变量**：Lambda表达式内部不能修改外部局部变量

```java
public class VariableModification {
    public static void main(String[] args) {
        int count = 0;

        // Runnable r = () -> count++;  // 编译错误

        // 解决方案：使用数组或对象
        int[] counter = {0};
        Runnable r = () -> counter[0]++;
        r.run();
        System.out.println("Counter: " + counter[0]);  // 1
    }
}
```

3. **可以访问实例变量和静态变量**：没有限制

```
public class InstanceVariables {
    private int instanceVar = 10;
    private static int staticVar = 20;

    public void test() {
        Runnable r = () -> {
            instanceVar++;  // 合法
            staticVar++;    // 合法
            System.out.println("instanceVar = " + instanceVar);
            System.out.println("staticVar = " + staticVar);
        };
        r.run();
    }

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

2.6.2 this关键字特性

在Lambda表达式中，`this`关键字引用的是创建Lambda表达式的外部类的实例，而不是Lambda表达式本身（因为Lambda表达式不是内部类）。

```java
public class ThisKeywordExample {
    private String name = "Outer Class";

    public void test() {
        Runnable r = () -> {
            // 这里的this引用的是ThisKeywordExample的实例
            System.out.println("this.name = " + this.name);
        };

        // 对比匿名内部类中的this
        Runnable r2 = new Runnable() {
            private String name = "Anonymous Class";

            @Override
            public void run() {
                // 这里的this引用的是匿名内部类的实例
                System.out.println("this.name = " + this.name);
                // 要访问外部类的name，需要使用OuterClassName.this
                System.out.println("Outer.this.name = " + ThisKeywordExample.this.name);
            }
        };

        r.run();
        r2.run();
    }

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