1. **Object类、常用的API**

根据JDK源代码及Object类的API文档，Object类当中包含的方法有11个。今天我们主要学习其中的2个：

\* `public String toString()`：返回该对象的字符串表示。

\* `public boolean equals(Object obj)`：指示其他某个对象是否与此对象“相等”。

在比较两个对象的时候，Object的equals方法容易抛出空指针异常，而Objects类中的equals方法就优化了这个问题。方法如下：

\* `public static boolean equals(Object a, Object b)`:判断两个对象是否相等。

源码：

~~~java

public static boolean equals(Object a, Object b) {

return (a == b) || (a != null && a.equals(b));

}

日期时间类

```java

import java.util.Date;

public class Demo01Date {

public static void main(String[] args) {

// 创建日期对象，把当前的时间

System.out.println(new Date()); // Tue Jan 16 14:37:35 CST 2018

// 创建日期对象，把当前的毫秒值转成日期对象

System.out.println(new Date(0L)); // Thu Jan 01 08:00:00 CST 1970

}

}

```

> tips:在使用println方法时，会自动调用Date类中的toString方法。Date类对Object类中的toString方法进行了覆盖重写，所以结果为指定格式的字符串。

### 常用方法

Date类中的多数方法已经过时，常用的方法有：

\* `public long getTime()` 把日期对象转换成对应的时间毫秒值。

\* `public SimpleDateFormat(String pattern)`：用给定的模式和默认语言环境的日期格式符号构造SimpleDateFormat。

### 格式规则

常用的格式规则为：

| 标识字母（区分大小写） | 含义 |

| ----------- | ---- |

| y | 年 |

| M | 月 |

| d | 日 |

| H | 时 |

| m | 分 |

| s | 秒 |

> 备注：更详细的格式规则，可以参考SimpleDateFormat类的API文档0。

创建SimpleDateFormat对象的代码如：

```java

import java.text.DateFormat;

import java.text.SimpleDateFormat;

public class Demo02SimpleDateFormat {

public static void main(String[] args) {

// 对应的日期格式如：2018-01-16 15:06:38

DateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

}

}

```

### 常用方法

DateFormat类的常用方法有：

- `public String format(Date date)`：将Date对象格式化为字符串。

- `public Date parse(String source)`：将字符串解析为Date对象。

#### format方法

使用format方法的代码为：

```java

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

/\*

把Date对象转换成String

\*/

public class Demo03DateFormatMethod {

public static void main(String[] args) {

Date date = new Date();

// 创建日期格式化对象,在获取格式化对象时可以指定风格

DateFormat df = new SimpleDateFormat("yyyy年MM月dd日");

String str = df.format(date);

System.out.println(str); // 2008年1月23日

}

}

```

#### parse方法

使用parse方法的代码为：

```java

import java.text.DateFormat;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Date;

/\*

把String转换成Date对象

\*/

public class Demo04DateFormatMethod {

public static void main(String[] args) throws ParseException {

DateFormat df = new SimpleDateFormat("yyyy年MM月dd日");

String str = "2018年12月11日";

Date date = df.parse(str);

System.out.println(date); // Tue Dec 11 00:00:00 CST 2018

}

}

```

**System类**

`java.lang.System`类中提供了大量的静态方法，可以获取与系统相关的信息或系统级操作，在System类的API文档中，常用的方法有：

- `public static long currentTimeMillis()`：返回以毫秒为单位的当前时间。

- `public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`：将数组中指定的数据拷贝到另一个数组中。**其实是替换元素**

数组的拷贝动作是系统级的，性能很高。System.arraycopy方法具有5个参数，含义分别为：

| 参数序号 | 参数名称 | 参数类型 | 参数含义 |

| ---- | ------- | ------ | ---------- |

| 1 | src | Object | 源数组 |

| 2 | srcPos | int | 源数组索引起始位置 |

| 3 | dest | Object | 目标数组 |

| 4 | destPos | int | 目标数组索引起始位置 |

| 5 | length | int | 复制元素个数 |

**StringBuilder**常用的方法有2个：

- `public StringBuilder append(...)`：添加任意类型数据的字符串形式，并返回当前对象自身。

- `public String toString()`：将当前StringBuilder对象转换为String对象。

**包装类：**

| 基本类型 | 对应的包装类（位于java.lang包中） |

| ------- | --------------------- |

| byte | Byte |

| short | Short |

| int | \*\*Integer\*\* |

| long | Long |

| float | Float |

| double | Double |

| char | \*\*Character\*\* |

| boolean | Boolean |

装箱与拆箱

基本类型与对应的包装类对象之间，来回转换的过程称为”装箱“与”拆箱“：

\* \*\*装箱\*\*：从基本类型转换为对应的包装类对象。

\* \*\*拆箱\*\*：从包装类对象转换为对应的基本类型。

String转换成对应的基本类型

除了Character类之外，其他所有包装类都具有parseXxx静态方法可以将字符串参数转换为对应的基本类型：

- `public static byte parseByte(String s)`：将字符串参数转换为对应的byte基本类型。

- `public static short parseShort(String s)`：将字符串参数转换为对应的short基本类型。

- `public static int parseInt(String s)`：将字符串参数转换为对应的int基本类型。

- `public static long parseLong(String s)`：将字符串参数转换为对应的long基本类型。

- `public static float parseFloat(String s)`：将字符串参数转换为对应的float基本类型。

- `public static double parseDouble(String s)`：将字符串参数转换为对应的double基本类型。

- `public static boolean parseBoolean(String s)`：将字符串参数转换为对应的boolean基本类型。

代码使用（仅以Integer类的静态方法parseXxx为例）如：

```java

public class Demo18WrapperParse {

public static void main(String[] args) {

int num = Integer.parseInt("100");

}

}

```

> 注意:如果字符串参数的内容无法正确转换为对应的基本类型，则会抛出`java.lang.NumberFormatException`异常。

>

1. **Collection集合、泛型**

**集合和数组既然都是容器，它们有啥区别呢？**

\* 数组的长度是固定的。集合的长度是可变的。

\* 数组中存储的是同一类型的元素，可以存储基本数据类型值。集合存储的都是对象。而且对象的类型可以不一致。在开发中一般当对象多的时候，使用集合进行存储。

**集合按照其存储结构可以分为两大类，分别是单列集合`java.util.Collection`和双列集合`java.util.Map`。**

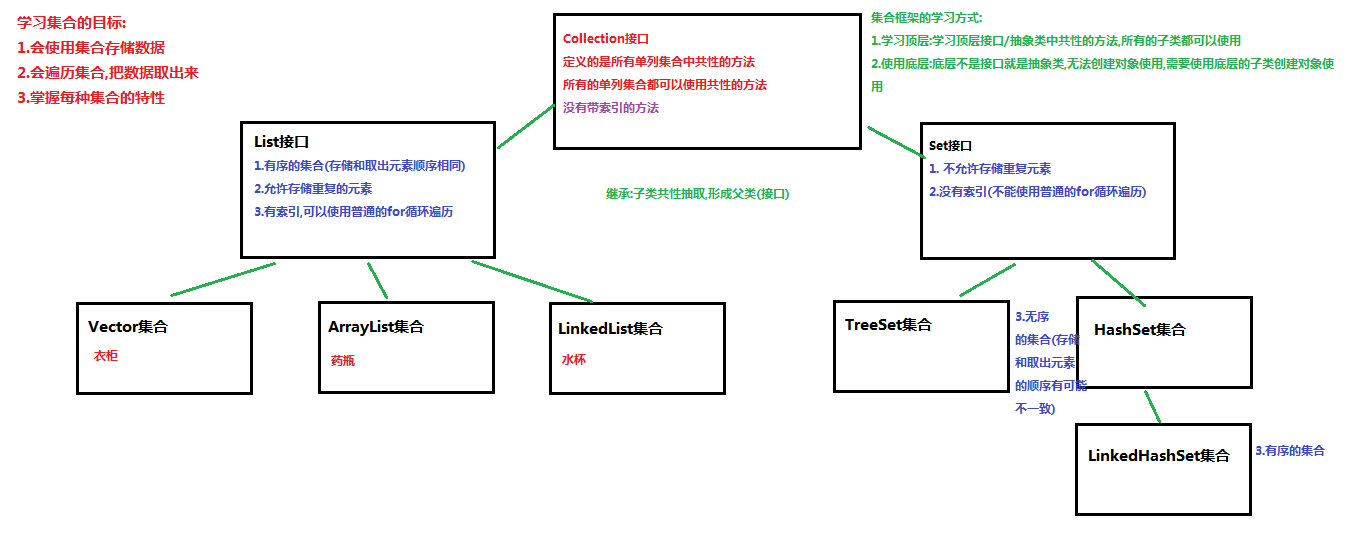
**\* \*\*Collection\*\*：单列集合类的根接口，用于存储一系列符合某种规则的元素，它有两个重要的子接口，分别是`java.util.List`和`java.util.Set`。**

**其中，`List`的特点是元素有序、元素可重复。**

**`Set`的特点是元素无序，而且不可重复。**

**`List`接口的主要实现类有`java.util.ArrayList`和`java.util.LinkedList`，**

**`Set`接口的主要实现类有`java.util.HashSet`和`java.util.TreeSet`。**



Collection是所有单列集合的父接口，因此在Collection中定义了单列集合(List和Set)通用的一些方法，这些方法可用于操作所有的单列集合。方法如下：

\* `public boolean add(E e)`： 把给定的对象添加到当前集合中 。

\* `public void clear()` :清空集合中所有的元素。

\* `public boolean remove(E e)`: 把给定的对象在当前集合中删除。

\* `public boolean contains(E e)`: 判断当前集合中是否包含给定的对象。

\* `public boolean isEmpty()`: 判断当前集合是否为空。

\* `public int size()`: 返回集合中元素的个数。

\* `public Object[] toArray()`: 把集合中的元素，存储到数组中。

例如：

~~~java

import java.util.ArrayList;

import java.util.Collection;

public class Demo1Collection {

public static void main(String[] args) {

// 创建集合对象

// 使用多态形式

Collection<String> coll = new ArrayList<String>();

// 使用方法

// 添加功能 boolean add(String s)

coll.add("小李广");

coll.add("扫地僧");

coll.add("石破天");

System.out.println(coll);

// boolean contains(E e) 判断o是否在集合中存在

System.out.println("判断 扫地僧 是否在集合中"+coll.contains("扫地僧"));

//boolean remove(E e) 删除在集合中的o元素

System.out.println("删除石破天："+coll.remove("石破天"));

System.out.println("操作之后集合中元素:"+coll);

// size() 集合中有几个元素

System.out.println("集合中有"+coll.size()+"个元素");

// Object[] toArray()转换成一个Object数组

Object[] objects = coll.toArray();

// 遍历数组

for (int i = 0; i < objects.length; i++) {

System.out.println(objects[i]);

}

// void clear() 清空集合

coll.clear();

System.out.println("集合中内容为："+coll);

// boolean isEmpty() 判断是否为空

System.out.println(coll.isEmpty());

}

}

**Iterator迭代器**

在程序开发中，经常需要遍历集合中的所有元素。针对这种需求，JDK专门提供了一个接口`java.util.Iterator`。`Iterator`接口也是Java集合中的一员，但它与`Collection`、`Map`接口有所不同，`Collection`接口与`Map`接口主要用于存储元素，而`Iterator`**主要用于迭代访问（即遍历）`Collection`中的元素**，因此`Iterator`对象也被称为迭代器。

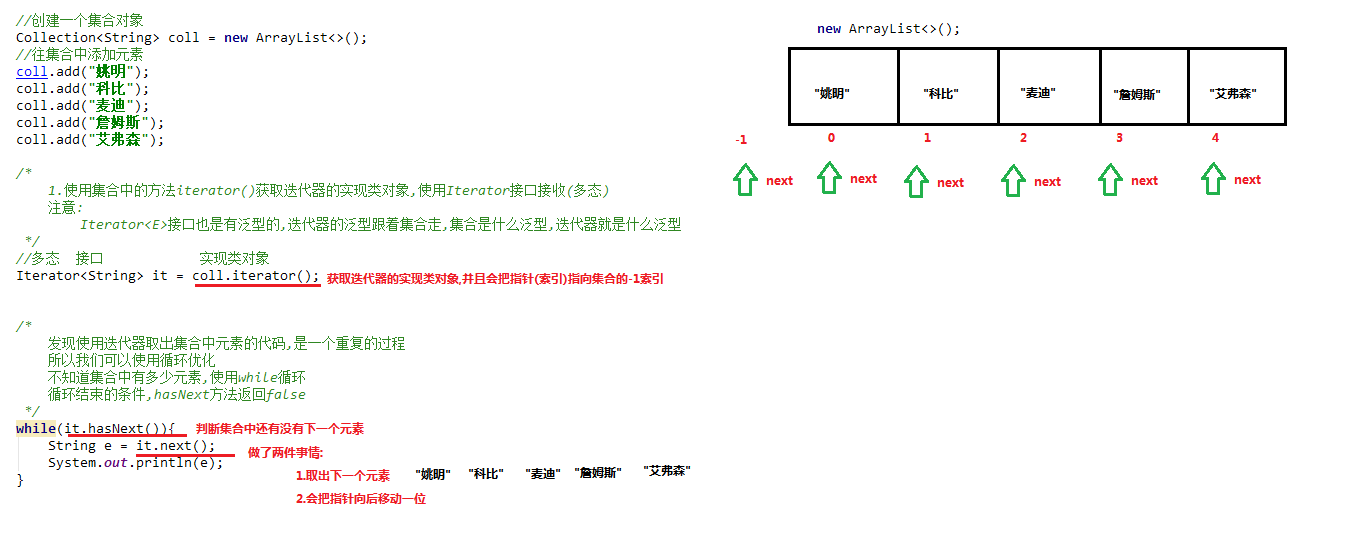
想要遍历Collection集合，那么就要获取该集合迭代器完成迭代操作，下面介绍一下获取迭代器的方法：

\* `public Iterator iterator()`: 获取集合对应的迭代器，用来遍历集合中的元素的。

**Iterator接口的常用方法如下：**

\* `public E next()`:返回迭代的下一个元素。

\* `public boolean hasNext()`:如果仍有元素可以迭代，则返回 true。



**泛型：**



~~~java

class ArrayList<E>{

public boolean add(E e){ }

public E get(int index){ }

....

}

例如，

~~~java

public interface MyGenericInterface<E>{

public abstract void add(E e);

public abstract E getE();

}

**Tip:**

**泛型不存在继承关系 Collection<Object> list = new ArrayList<String>();这种是错误的。**

*## 4.2 案例分析*

*\* 准备牌：*

*牌可以设计为一个ArrayList<String>,每个字符串为一张牌。*

*每张牌由花色数字两部分组成，我们可以使用花色集合与数字集合嵌套迭代完成每张牌的组装。*

*牌由Collections类的shuffle方法进行随机排序。*

*\* 发牌*

*将每个人以及底牌设计为ArrayList<String>,将最后3张牌直接存放于底牌，剩余牌通过对3取模依次发牌。*

*\* 看牌*

*直接打印每个集合。*

*## 4.3 代码实现*

*~~~java*

*import java.util.ArrayList;*

*import java.util.Collections;*

*public class Poker {*

*public static void main(String[] args) {*

*/\**

*\* 1: 准备牌操作*

*\*/*

*//1.1 创建牌盒 将来存储牌面的*

*ArrayList<String> pokerBox = new ArrayList<String>();*

*//1.2 创建花色集合*

*ArrayList<String> colors = new ArrayList<String>();*

*//1.3 创建数字集合*

*ArrayList<String> numbers = new ArrayList<String>();*

*//1.4 分别给花色 以及 数字集合添加元素*

*colors.add("♥");*

*colors.add("♦");*

*colors.add("♠");*

*colors.add("♣");*

*for(int i = 2;i<=10;i++){*

*numbers.add(i+"");*

*}*

*numbers.add("J");*

*numbers.add("Q");*

*numbers.add("K");*

*numbers.add("A");*

*//1.5 创造牌 拼接牌操作*

*// 拿出每一个花色 然后跟每一个数字 进行结合 存储到牌盒中*

*for (String color : colors) {*

*//color每一个花色*

*//遍历数字集合*

*for(String number : numbers){*

*//结合*

*String card = color+number;*

*//存储到牌盒中*

*pokerBox.add(card);*

*}*

*}*

*//1.6大王小王*

*pokerBox.add("小☺");*

*pokerBox.add("大☠");*

*// System.out.println(pokerBox);*

*//洗牌 是不是就是将 牌盒中 牌的索引打乱*

*// Collections类 工具类 都是 静态方法*

*// shuffer方法*

*/\**

*\* static void shuffle(List<?> list)*

*\* 使用默认随机源对指定列表进行置换。*

*\*/*

*//2:洗牌*

*Collections.shuffle(pokerBox);*

*//3 发牌*

*//3.1 创建 三个 玩家集合 创建一个底牌集合*

*ArrayList<String> player1 = new ArrayList<String>();*

*ArrayList<String> player2 = new ArrayList<String>();*

*ArrayList<String> player3 = new ArrayList<String>();*

*ArrayList<String> dipai = new ArrayList<String>();*

*//遍历 牌盒 必须知道索引*

*for(int i = 0;i<pokerBox.size();i++){*

*//获取 牌面*

*String card = pokerBox.get(i);*

*//留出三张底牌 存到 底牌集合中*

*if(i>=51){//存到底牌集合中*

*dipai.add(card);*

*} else {*

*//玩家1 %3 ==0*

*if(i%3==0){*

*player1.add(card);*

*}else if(i%3==1){//玩家2*

*player2.add(card);*

*}else{//玩家3*

*player3.add(card);*

*}*

*}*

*}*

*//看看*

*System.out.println("令狐冲："+player1);*

*System.out.println("田伯光："+player2);*

*System.out.println("绿竹翁："+player3);*

*System.out.println("底牌："+dipai);*

*}*

*}*

**三、【List、Set、数据结构、Collections】**

## 2.2 常见的数据结构

数据存储的常用结构有：栈、队列、数组、链表和红黑树。我们分别来了解一下：

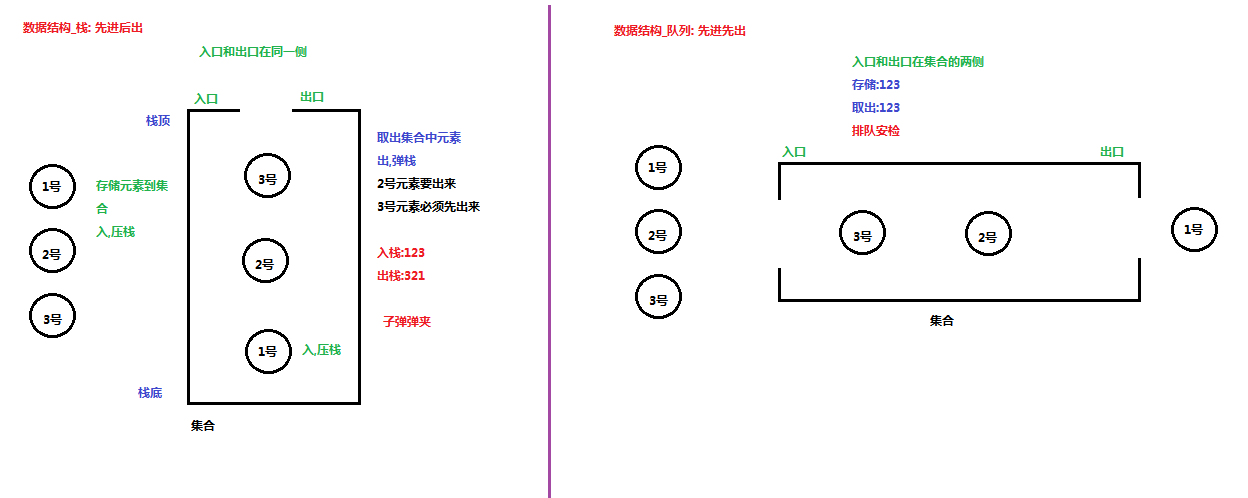
#### 栈

\* \*\*栈\*\*：\*\*stack\*\*,又称堆栈，它是运算受限的线性表，其限制是仅允许在表的一端进行插入和删除操作，不允许在其他任何位置进行添加、查找、删除等操作。

简单的说：采用该结构的集合，对元素的存取有如下的特点

\* 先进后出（即，存进去的元素，要在后它后面的元素依次取出后，才能取出该元素）。例如，子弹压进弹夹，先压进去的子弹在下面，后压进去的子弹在上面，当开枪时，先弹出上面的子弹，然后才能弹出下面的子弹。

\* 栈的入口、出口的都是栈的顶端位置。



这里两个名词需要注意：

\* \*\*压栈\*\*：就是存元素。即，把元素存储到栈的顶端位置，栈中已有元素依次向栈底方向移动一个位置。

\* \*\*弹栈\*\*：就是取元素。即，把栈的顶端位置元素取出，栈中已有元素依次向栈顶方向移动一个位置。

#### 队列

\* \*\*队列\*\*：\*\*queue\*\*,简称队，它同堆栈一样，也是一种运算受限的线性表，其限制是仅允许在表的一端进行插入，而在表的另一端进行删除。

简单的说，采用该结构的集合，对元素的存取有如下的特点：

\* 先进先出（即，存进去的元素，要在后它前面的元素依次取出后，才能取出该元素）。例如，小火车过山洞，车头先进去，车尾后进去；车头先出来，车尾后出来。

\* 队列的入口、出口各占一侧。例如，下图中的左侧为入口，右侧为出口。

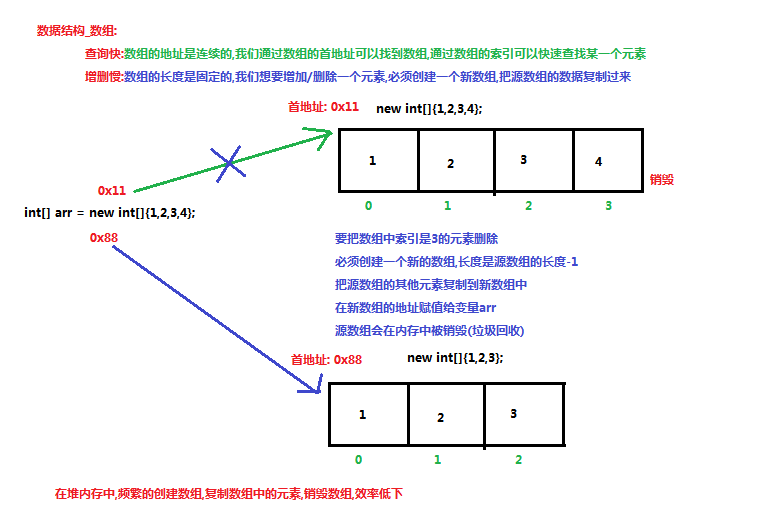
![](img\队列图.bmp)

#### 数组

\* \*\*数组\*\*:\*\*Array\*\*,是有序的元素序列，数组是在内存中开辟一段连续的空间，并在此空间存放元素。就像是一排出租屋，有100个房间，从001到100每个房间都有固定编号，通过编号就可以快速找到租房子的人。

简单的说,采用该结构的集合，对元素的存取有如下的特点：

\* 查找元素快：通过索引，可以快速访问指定位置的元素



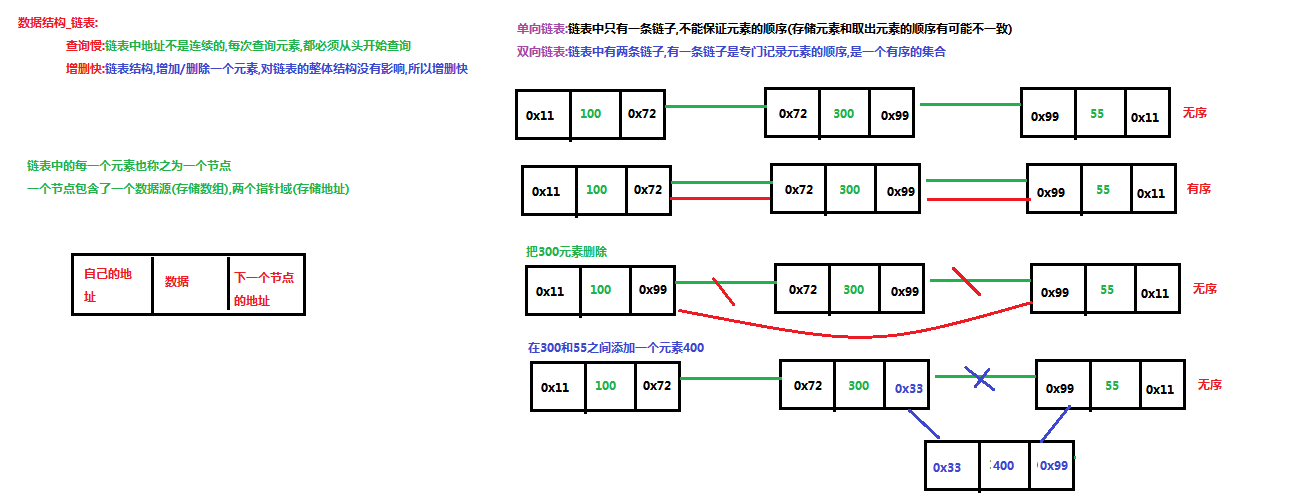
\* 增删元素慢

\* \*\*指定索引位置增加元素\*\*：需要创建一个新数组，将指定新元素存储在指定索引位置，再把原数组元素根据索引，复制到新数组对应索引的位置。如下图![](img/数组添加.png)

\* \*\*指定索引位置删除元素：\*\*需要创建一个新数组，把原数组元素根据索引，复制到新数组对应索引的位置，原数组中指定索引位置元素不复制到新数组中。如下图![](img/数组删除.png)

#### 链表

\* \*\*链表\*\*:\*\*linked list\*\*,由一系列结点node（链表中每一个元素称为结点）组成，结点可以在运行时i动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。我们常说的链表结构有单向链表与双向链表，那么这里给大家介绍的是\*\*单向链表\*\*。



简单的说，采用该结构的集合，对元素的存取有如下的特点：

\* 多个结点之间，通过地址进行连接。例如，多个人手拉手，每个人使用自己的右手拉住下个人的左手，依次类推，这样多个人就连在一起了。

![](img\单链表结构.png)

\* 查找元素慢：想查找某个元素，需要通过连接的节点，依次向后查找指定元素

\* 增删元素快：

\* 增加元素：只需要修改连接下个元素的地址即可。

![](img\增加结点.png)

\* 删除元素：只需要修改连接下个元素的地址即可。

![](img\删除结点.bmp)

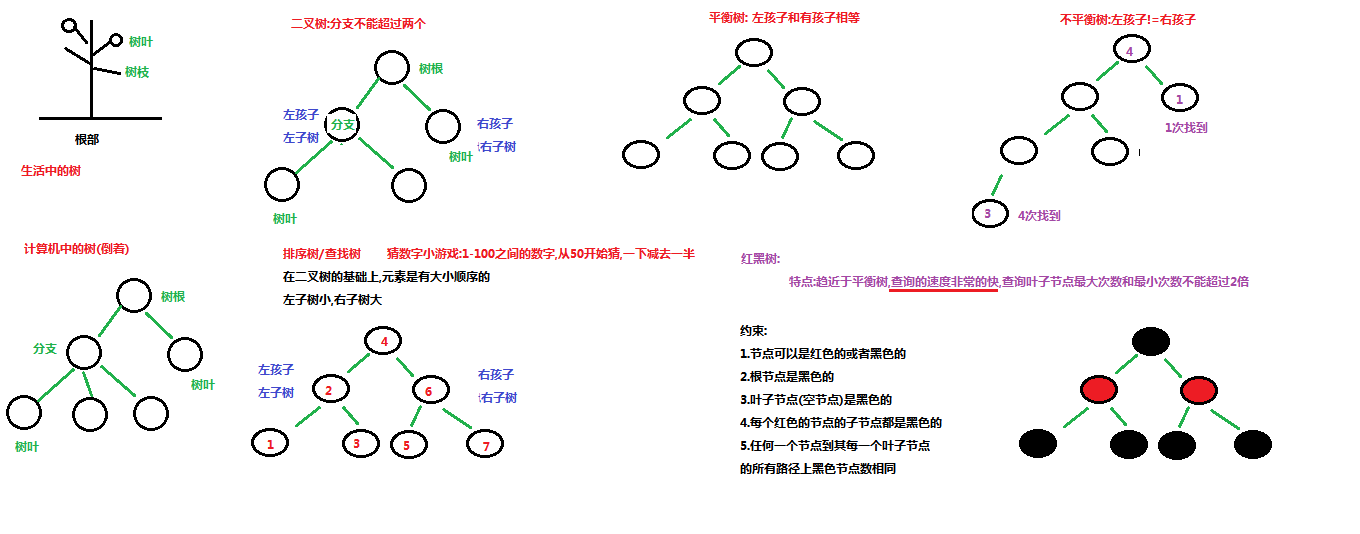
#### 红黑树

\* \*\*二叉树\*\*：\*\*binary tree\*\* ,是每个结点不超过2的有序\*\*树（tree）\*\* 。

简单的理解，就是一种类似于我们生活中树的结构，只不过每个结点上都最多只能有两个子结点。

二叉树是每个节点最多有两个子树的树结构。顶上的叫根结点，两边被称作“左子树”和“右子树”。

如图：



我们要说的是二叉树的一种比较有意思的叫做\*\*红黑树\*\*，红黑树本身就是一颗二叉查找树，将节点插入后，该树仍然是一颗二叉查找树。也就意味着，树的键值仍然是有序的。

红黑树的约束:

1. 节点可以是红色的或者黑色的

2. 根节点是黑色的

3. 叶子节点(特指空节点)是黑色的

4. 每个红色节点的子节点都是黑色的

5. 任何一个节点到其每一个叶子节点的所有路径上黑色节点数相同

红黑树的特点:

​ 速度特别快,趋近平衡树,查找叶子元素最少和最多次数不多于二倍

**List集合**

List接口特点：

1. 它是一个元素存取有序的集合。例如，存元素的顺序是11、22、33。那么集合中，元素的存储就是按照11、22、33的顺序完成的）。

2. 它是一个带有索引的集合，通过索引就可以精确的操作集合中的元素（与数组的索引是一个道理）。

3. 集合中可以有重复的元素，通过元素的equals方法，来比较是否为重复的元素。

**List接口中常用方法：**

public void add(int index, E element)`: 将指定的元素，添加到该集合中的指定位置上。

- `public E get(int index)`:返回集合中指定位置的元素。

- `public E remove(int index)`: 移除列表中指定位置的元素, 返回的是被移除的元素。

- `public E set(int index, E element)`:用指定元素替换集合中指定位置的元素,返回值的更新前的元素。

LinkedList集合：

\* `public void addFirst(E e)`:将指定元素插入此列表的开头。

\* `public void addLast(E e)`:将指定元素添加到此列表的结尾。

\* `public E getFirst()`:返回此列表的第一个元素。

\* `public E getLast()`:返回此列表的最后一个元素。

\* `public E removeFirst()`:移除并返回此列表的第一个元素。

\* `public E removeLast()`:移除并返回此列表的最后一个元素。

\* `public E pop()`:从此列表所表示的堆栈处弹出一个元素。

\* `public void push(E e)`:将元素推入此列表所表示的堆栈。

\* `public boolean isEmpty()`：如果列表不包含元素，则返回true。

方法演示：

~~~java

public class LinkedListDemo {

public static void main(String[] args) {

LinkedList<String> link = new LinkedList<String>();

//添加元素

link.addFirst("abc1");

link.addFirst("abc2");

link.addFirst("abc3");

System.out.println(link);

// 获取元素

System.out.println(link.getFirst());

System.out.println(link.getLast());

// 删除元素

System.out.println(link.removeFirst());

System.out.println(link.removeLast());

while (!link.isEmpty()) { //判断集合是否为空

System.out.println(link.pop()); //弹出集合中的栈顶元素

}

System.out.println(link);

}

}

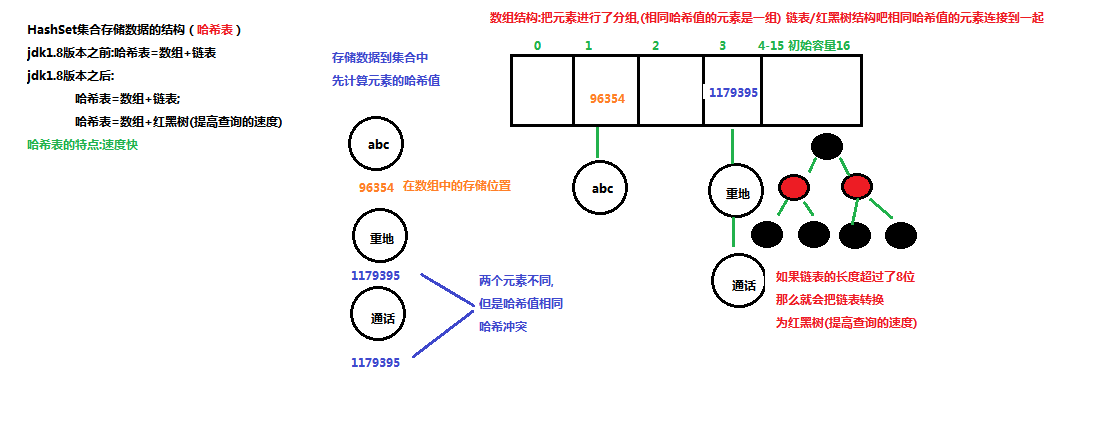
Set接口

`Set`集合有多个子类，这里我们介绍其中的`java.util.HashSet`、`java.util.LinkedHashSet`这两个集合。

**Set集合取出元素的方式可以采用：迭代器、增强for。**

**`HashSet`是根据对象的哈希值来确定元素在集合中的存储位置，因此具有良好的存取和查找性能。保证元素唯一性的方式依赖于：`hashCode`与`equals`方法。**

**JDK1.8中，哈希表存储采用数组+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样大大减少了查找时间。**





LinkedHashSet

在HashSet下面有一个子类`java.util.LinkedHashSet`，它是链表和哈希表组合的一个数据存储结构。

Collections

`java.utils.Collections`是集合工具类，用来对集合进行操作。部分方法如下：

- `public static <T> boolean addAll(Collection<T> c, T... elements) `:往集合中添加一些元素。

- `public static void shuffle(List<?> list) 打乱顺序`:打乱集合顺序。

- `public static <T> void sort(List<T> list)`:将集合中元素按照默认规则排序。

- `public static <T> void sort(List<T> list，Comparator<? super T> )`:将集合中元素按照指定规则排序。

例如：

```java

public class CollectionsDemo {

public static void main(String[] args) {

ArrayList<Integer> list = new ArrayList<Integer>();

//原来写法

//list.add(12);

//list.add(14);

//list.add(15);

//list.add(1000);

//采用工具类 完成 往集合中添加元素

Collections.addAll(list, 5, 222, 1，2);

System.out.println(list);

//排序方法

Collections.sort(list);

System.out.println(list);

}

}

结果：

[5, 222, 1, 2]

[1, 2, 5, 222]

Comparator比较器

\*\*Comparable\*\*：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compareTo方法被称为它的自然比较方法。只能在类中实现compareTo()一次，不能经常修改类的代码实现自己想要的排序。实现此接口的对象列表（和数组）可以通过Collections.sort（和Arrays.sort）进行自动排序，对象可以用作有序映射中的键或有序集合中的元素，无需指定比较器。

\*\*Comparator\*\*强行对某个对象进行整体排序。可以将Comparator 传递给sort方法（如Collections.sort或 Arrays.sort），从而允许在排序顺序上实现精确控制。还可以使用Comparator来控制某些数据结构（如有序set或有序映射）的顺序，或者为那些没有自然顺序的对象collection提供排序。

四、【Map】



\* `Collection`中的集合，元素是孤立存在的（理解为单身），向集合中存储元素采用一个个元素的方式存储。

\* `Map`中的集合，元素是成对存在的(理解为夫妻)。每个元素由键与值两部分组成，通过键可以找对所对应的值。

\* `Collection`中的集合称为单列集合，`Map`中的集合称为双列集合。

\* 需要注意的是，`Map`中的集合不能包含重复的键，值可以重复；每个键只能对应一个值。

\* \*\*HashMap<K,V>\*\*：存储数据采用的哈希表结构，元素的存取顺序不能保证一致。由于要保证键的唯一、不重复，需要重写键的hashCode()方法、equals()方法。

\* \*\*LinkedHashMap<K,V>\*\*：HashMap下有个子类LinkedHashMap，存储数据采用的哈希表结构+链表结构。通过链表结构可以保证元素的存取顺序一致；通过哈希表结构可以保证的键的唯一、不重复，需要重写键的hashCode()方法、equals()方法。

> tips：Map接口中的集合都有两个泛型变量<K,V>,在使用时，要为两个泛型变量赋予数据类型。两个泛型变量<K,V>的数据类型可以相同，也可以不同。

Map接口中的常用方法

Map接口中定义了很多方法，常用的如下：

\* `public V put(K key, V value)`: 把指定的键与指定的值添加到Map集合中。

\* `public V remove(Object key)`: 把指定的键 所对应的键值对元素 在Map集合中删除，返回被删除元素的值。

\* `public V get(Object key)` 根据指定的键，在Map集合中获取对应的值。

\* `boolean containsKey(Object key) ` 判断集合中是否包含指定的键。

\* `public Set<K> keySet()`: 获取Map集合中所有的键，存储到Set集合中。

\* `public Set<Map.Entry<K,V>> entrySet()`: 获取到Map集合中所有的键值对对象的集合(Set集合)。

tips:

>

> 使用put方法时，若指定的键(key)在集合中没有，则没有这个键对应的值，返回null，并把指定的键值添加到集合中；

>

> 若指定的键(key)在集合中存在，则返回值为集合中键对应的值（该值为替换前的值），并把指定键所对应的值，替换成指定的新值。

键找值方式：即通过元素中的键，获取键所对应的值

分析步骤：

1. 获取Map中所有的键，由于键是唯一的，所以返回一个Set集合存储所有的键。方法提示:`keyset()`

2. 遍历键的Set集合，得到每一个键。

3. 根据键，获取键所对应的值。方法提示:`get(K key)`

代码演示：

~~~java

public class MapDemo01 {

public static void main(String[] args) {

//创建Map集合对象

HashMap<String, String> map = new HashMap<String,String>();

//添加元素到集合

map.put("胡歌", "霍建华");

map.put("郭德纲", "于谦");

map.put("薛之谦", "大张伟");

//获取所有的键 获取键集

Set<String> keys = map.keySet();

// 遍历键集 得到 每一个键

for (String key : keys) {

//key 就是键

//获取对应值

String value = map.get(key);

System.out.println(key+"的CP是："+value);

}

}

}

Entry键值对对象

我们已经知道，`Map`中存放的是两种对象，一种称为\*\*key\*\*(键)，一种称为\*\*value\*\*(值)，它们在在`Map`中是一一对应关系，这一对对象又称做`Map`中的一个`Entry(项)`。`Entry`将键值对的对应关系封装成了对象。即键值对对象，这样我们在遍历`Map`集合时，就可以从每一个键值对（`Entry`）对象中获取对应的键与对应的值。

既然Entry表示了一对键和值，那么也同样提供了获取对应键和对应值得方法：

\* `public K getKey()`：获取Entry对象中的键。

\* `public V getValue()`：获取Entry对象中的值。

在Map集合中也提供了获取所有Entry对象的方法：

\* `public Set<Map.Entry<K,V>> entrySet()`: 获取到Map集合中所有的键值对对象的集合(Set集合)。

## 1.6 Map集合遍历键值对方式

键值对方式：即通过集合中每个键值对(Entry)对象，获取键值对(Entry)对象中的键与值。

操作步骤与图解：

1. 获取Map集合中，所有的键值对(Entry)对象，以Set集合形式返回。方法提示:`entrySet()`。

2. 遍历包含键值对(Entry)对象的Set集合，得到每一个键值对(Entry)对象。

3. 通过键值对(Entry)对象，获取Entry对象中的键与值。 方法提示:`getkey() getValue()`

~~~java

public class MapDemo02 {

public static void main(String[] args) {

// 创建Map集合对象

HashMap<String, String> map = new HashMap<String,String>();

// 添加元素到集合

map.put("胡歌", "霍建华");

map.put("郭德纲", "于谦");

map.put("薛之谦", "大张伟");

// 获取 所有的 entry对象 entrySet

Set<Entry<String,String>> entrySet = map.entrySet();

// 遍历得到每一个entry对象

for (Entry<String, String> entry : entrySet) {

// 解析

String key = entry.getKey();

String value = entry.getValue();

System.out.println(key+"的CP是:"+value);

}

}

}

~~~

五、【异常、线程】



finally 代码块

\*\*finally\*\*：有一些特定的代码无论异常是否发生，都需要执行。另外，因为异常会引发程序跳转，导致有些语句执行不到。而finally就是解决这个问题的，在finally代码块中存放的代码都是一定会被执行的。

\* 运行时异常被抛出可以不处理。即不捕获也不声明抛出。

\* 如果finally有return语句,永远返回finally中的结果,避免该情况.

\* 如果父类抛出了多个异常,子类重写父类方法时,抛出和父类相同的异常或者是父类异常的子类或者不抛出异常。

\* 父类方法没有抛出异常，子类重写父类该方法时也不可抛出异常。此时子类产生该异常，只能捕获处理，不能声明抛出。

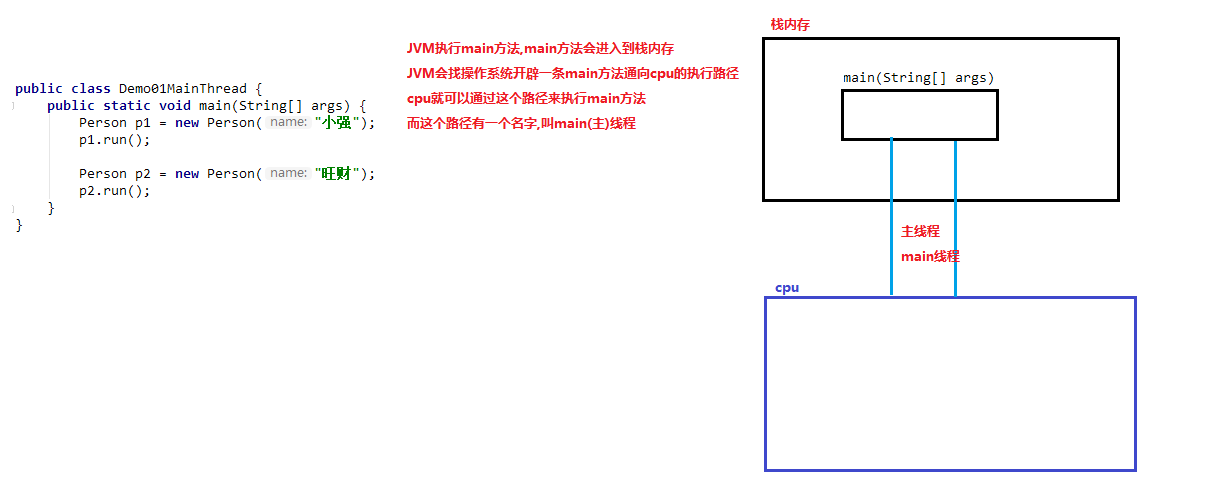
多线程：

并发与并行：

\* \*\*并发\*\*：指两个或多个事件在\*\*同一个时间段内\*\*发生。

\* \*\*并行\*\*：指两个或多个事件在\*\*同一时刻\*\*发生（同时发生）。

线程与进程



\* \*\*进程\*\*：是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，一个应用程序可以同时运行多个进程；进程也是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创建、运行到消亡的过程。

\* \*\*线程\*\*：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

简而言之：一个程序运行后至少有一个进程，一个进程中可以包含多个线程 。

线程调度:\*\*

- 分时调度

所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间。

- 抢占式调度

优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个(线程随机性)，Java使用的为抢占式调度。

- 设置线程的优先级

![设置线程优先级](img/设置线程优先级.bmp)

- 抢占式调度详解

大部分操作系统都支持多进程并发运行，现在的操作系统几乎都支持同时运行多个程序。比如：现在我们上课一边使用编辑器，一边使用录屏软件，同时还开着画图板，dos窗口等软件。此时，这些程序是在同时运行，”感觉这些软件好像在同一时刻运行着“。

实际上，CPU(中央处理器)使用抢占式调度模式在多个线程间进行着高速的切换。对于CPU的一个核而言，某个时刻，只能执行一个线程，而 CPU的在多个线程间切换速度相对我们的感觉要快，看上去就是在同一时刻运行。

其实，多线程程序并不能提高程序的运行速度，但能够提高程序运行效率，让CPU的使用率更高。

创建线程类

Java使用`java.lang.Thread`类代表\*\*线程\*\*，所有的线程对象都必须是Thread类或其子类的实例。每个线程的作用是完成一定的任务，实际上就是执行一段程序流即一段顺序执行的代码。Java使用线程执行体来代表这段程序流。Java中通过继承Thread类来\*\*创建\*\*并\*\*启动多线程\*\*的步骤如下：

1. 定义Thread类的子类，并重写该类的run()方法，该run()方法的方法体就代表了线程需要完成的任务,因此把run()方法称为线程执行体。

2. 创建Thread子类的实例，即创建了线程对象

3. 调用线程对象的start()方法来启动该线程

代码如下：

测试类：

~~~java

public class Demo01 {

public static void main(String[] args) {

//创建自定义线程对象

MyThread mt = new MyThread("新的线程！");

//开启新线程

mt.start();

//在主方法中执行for循环

for (int i = 0; i < 10; i++) {

System.out.println("main线程！"+i);

}

}

}

~~~

自定义线程类：

~~~java

public class MyThread extends Thread {

//定义指定线程名称的构造方法

public MyThread(String name) {

//调用父类的String参数的构造方法，指定线程的名称

super(name);

}

/\*\*

\* 重写run方法，完成该线程执行的逻辑

\*/

@Override

public void run() {

for (int i = 0; i < 10; i++) {

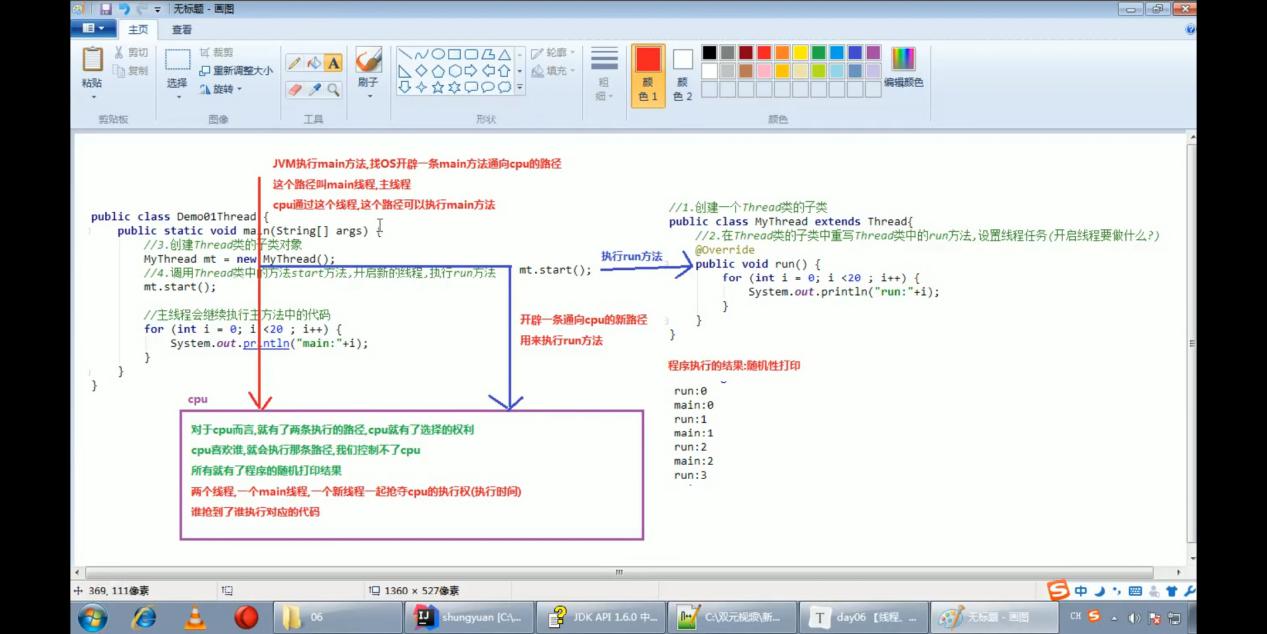
System.out.println(getName()+"：正在执行！"+i);

}

}

}

~~~



1. 线程、同步

**Thread类**

**构造方法：**public Thread() :分配一个新的线程对象。

public Thread(String name) :分配一个指定名字的新的线程对象。

public Thread(Runnable target) :分配一个带有指定目标新的线程对象。

public Thread(Runnable target,String name) :分配一个带有指定目标新的线程对象并指定名字。

**常用方法：**

public String getName() :获取当前线程名称。

public void start() :导致此线程开始执行; Java虚拟机调用此线程的run方法。

public void run() :此线程要执行的任务在此处定义代码。

public static void sleep(long millis) :使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行）。

public static Thread currentThread() :返回对当前正在执行的线程对象的引用。

**创建线程方式二**

采用 java.lang.Runnable 也是非常常见的一种，我们只需要重写run方法即可。

步骤如下：

1. 定义Runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。

2. 创建Runnable实现类的实例，并以此实例作为Thread的target来创建Thread对象，该Thread对象才是真正

的线程对象。

3. 调用线程对象的start()方法来启动线程。

代码如下：

public class MyRunnable implements Runnable{

@Override

public void run() {

for (int i = 0; i < 20; i++) {

System.out.println(Thread.currentThread().getName()+" "+i);

}

}

}

public class Demo {

public static void main(String[] args) {

//创建自定义类对象 线程任务对象

MyRunnable mr = new MyRunnable();

//创建线程对象

Thread t = new Thread(mr, "小强");

t.start();

for (int i = 0; i < 20; i++) {

System.out.println("旺财 " + i);

}

}

}

**Thread和Runnable的区别**

**总结：**

**实现Runnable接口比继承Thread类所具有的优势：**

1. 适合多个相同的程序代码的线程去共享同一个资源。

2. 可以避免java中的单继承的局限性。

3. 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立。

4. 线程池只能放入实现Runable或Callable类线程，不能直接放入继承Thread的类。

扩充：在java中，每次程序运行至少启动2个线程。一个是main线程，一个是垃圾收集线程。因为每当使用 java命令执行一个类的时候，实际上都会启动一个JVM，每一个JVM其实在就是在操作系统中启动了一个进程。

**匿名内部类方式实现线程的创建**



**线程安全**

为了保证每个线程都能正常执行原子操作,Java引入了线程同步机制。

1. 同步代码块。

2. 同步方法。

3. 锁机制。

**同步代码块**： synchronized 关键字可以用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。

格式:

synchronized(同步锁){

需要同步操作的代码

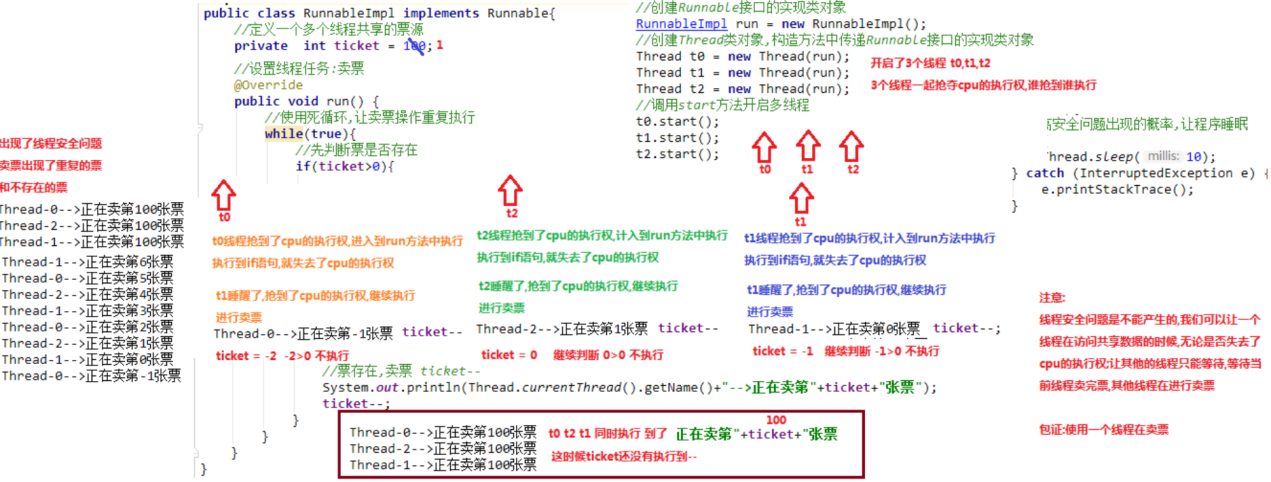
}

例如：

Object object=New Object();

Synchronized(object){

}



**同步方法**

**同步方法**:使用synchronized修饰的方法,就叫做同步方法,保证A线程执行该方法的时候,其他线程只能在方法外

等着。

格式：

public synchronized void method(){

可能会产生线程安全问题的代码

}

同步锁是谁?

对于非static方法,同步锁就是this。

对于static方法,我们使用当前方法所在类的字节码对象(类名.class)

**Lock锁**

java.util.concurrent.locks.Lock 机制提供了比**synchronized**代码块和**synchronized**方法更广泛的锁定操作,

同步代码块/同步方法具有的功能Lock都有,除此之外更强大,更体现面向对象。

Lock锁也称同步锁，加锁与释放锁方法化了，如下：

public void lock() :加同步锁。

public void unlock() :释放同步锁。

使用如下：

public class Ticket implements Runnable{

private int ticket = 100;

Lock lock = new ReentrantLock();

/\*

\* 执行卖票操作

\*/

@Override

public void run() {

//每个窗口卖票的操作

//窗口 永远开启

while(true){

lock.lock();

if(ticket>0){//有票 可以卖

//出票操作

//使用sleep模拟一下出票时间

try {

Thread.sleep(50);

} catch (InterruptedException e) {

// TODO Auto‐generated catch block

e.printStackTrace();

}

//获取当前线程对象的名字

String name = Thread.currentThread().getName();

System.out.println(name+"正在卖:"+ticket‐‐);

}

lock.unlock();

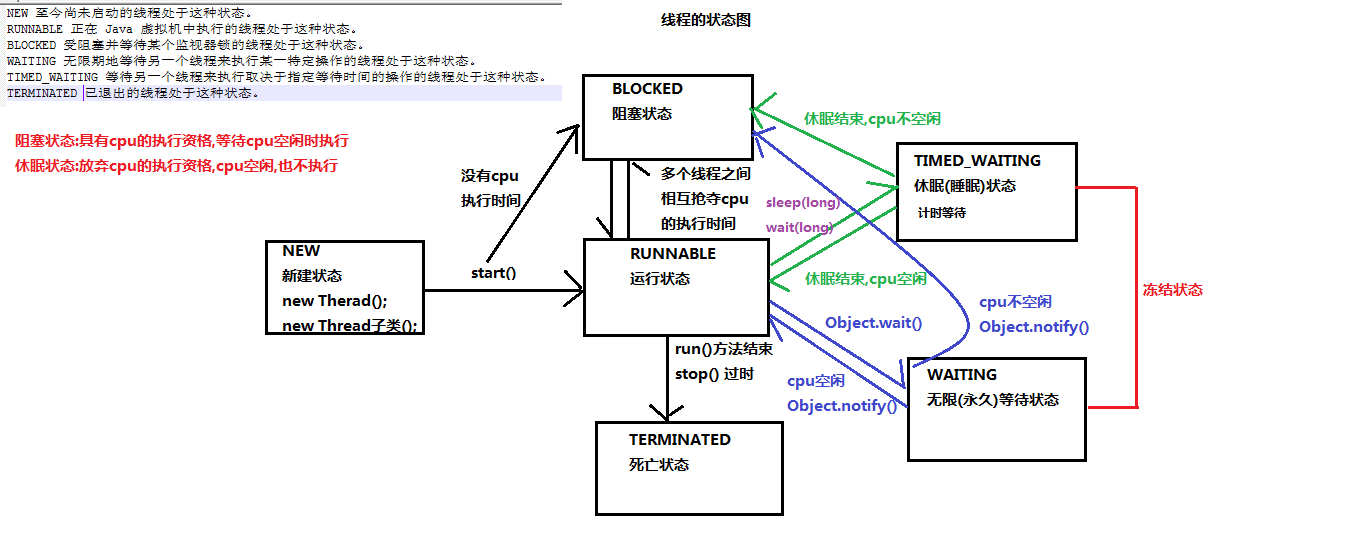
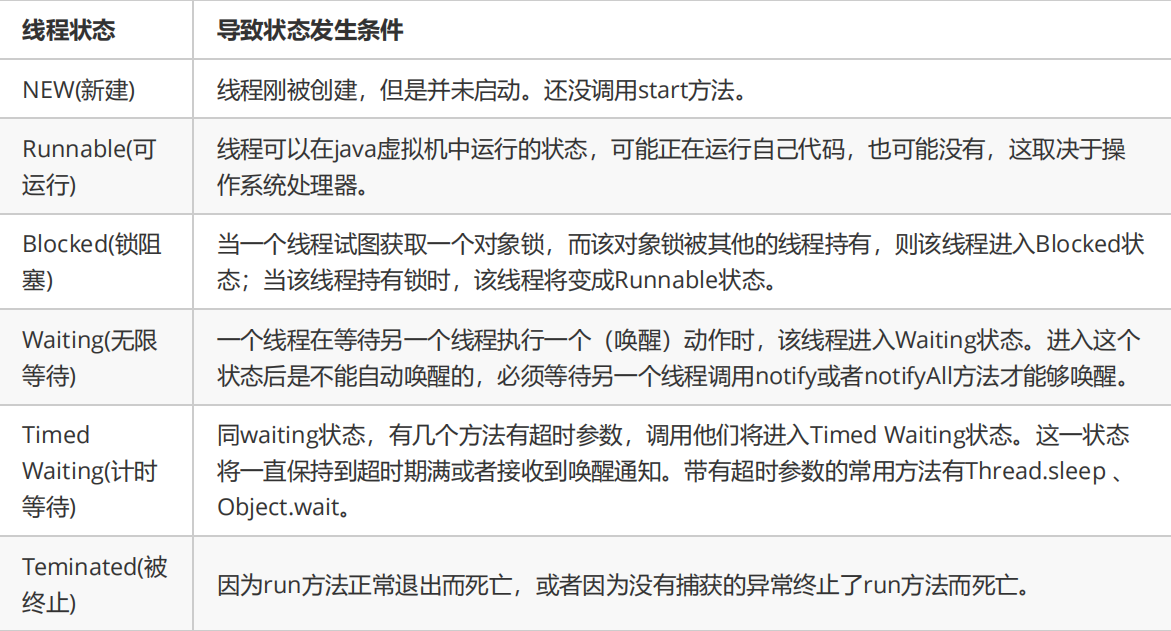
}

}

}

**线程状态**

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中， 有几种状态呢？在API中 java.lang.Thread.State 这个枚举中给出了六种线程状态：



七、线程池、Lambda表达式

合理利用线程池能够带来三个好处：

1. 降低资源消耗。减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2. 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。

3. 提高线程的可管理性。可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

使用线程池中线程对象的步骤：

1. 创建线程池对象。

2. 创建Runnable接口子类对象。(task)

3. 提交Runnable接口子类对象。(take task)

4. 关闭线程池(一般不做)。

Runnable实现类代码：

public class MyRunnable implements Runnable {

@Override

public void run() {

System.out.println("我要一个教练");

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("教练来了： " + Thread.currentThread().getName());

System.out.println("教我游泳,交完后，教练回到了游泳池");

}

}

线程池测试类：

public class ThreadPoolDemo {

public static void main(String[] args) {

// 创建线程池对象

ExecutorService service = Executors.newFixedThreadPool(2);//包含2个线程对象

// 创建Runnable实例对象

MyRunnable r = new MyRunnable();

service.submit(r);

// 再获取个线程对象，调用MyRunnable中的run()

service.submit(r);

service.submit(r);

// 注意：submit方法调用结束后，程序并不终止，是因为线程池控制了线程的关闭。

// 将使用完的线程又归还到了线程池中

// 关闭线程池

//service.shutdown();

}

}

**Lambda表达式**

借助Java 8的全新语法，上述`Runnable`接口的匿名内部类写法可以通过更简单的Lambda表达式达到等效：

```java

public class Demo02LambdaRunnable {

public static void main(String[] args) {

new Thread(() -> System.out.println("多线程任务执行！")).start(); // 启动线程

}

}

```

Lambda的使用前提

Lambda的语法非常简洁，完全没有面向对象复杂的束缚。但是使用时有几个问题需要特别注意：

1. 使用Lambda必须具有接口，且要求\*\*接口中有且仅有一个抽象方法\*\*。

无论是JDK内置的`Runnable`、`Comparator`接口还是自定义的接口，只有当接口中的抽象方法存在且唯一时，才可以使用Lambda。

2. 使用Lambda必须具有\*\*上下文推断\*\*。

也就是方法的参数或局部变量类型必须为Lambda对应的接口类型，才能使用Lambda作为该接口的实例。

> 备注：有且仅有一个抽象方法的接口，称为“\*\*函数式接口\*\*”。

八、File类、递归

# 第一章 File类

## 1.1 概述

`java.io.File` 类是文件和目录路径名的抽象表示，主要用于文件和目录的创建、查找和删除等操作。

## 1.2 构造方法

\* `public File(String pathname) ` ：通过将给定的\*\*路径名字符串\*\*转换为抽象路径名来创建新的 File实例。

\* `public File(String parent, String child) ` ：从\*\*父路径名字符串和子路径名字符串\*\*创建新的 File实例。

\* `public File(File parent, String child)` ：从\*\*父抽象路径名和子路径名字符串\*\*创建新的 File实例。

\* 构造举例，代码如下：

```java

// 文件路径名

String pathname = "D:\\aaa.txt";

File file1 = new File(pathname);

// 文件路径名

String pathname2 = "D:\\aaa\\bbb.txt";

File file2 = new File(pathname2);

// 通过父路径和子路径字符串

String parent = "d:\\aaa";

String child = "bbb.txt";

File file3 = new File(parent, child);

// 通过父级File对象和子路径字符串

File parentDir = new File("d:\\aaa");

String child = "bbb.txt";

File file4 = new File(parentDir, child);

```

> 小贴士：

>

> 1. 一个File对象代表硬盘中实际存在的一个文件或者目录。

> 2. 无论该路径下是否存在文件或者目录，都不影响File对象的创建。

## 1.3 常用方法

### 获取功能的方法

\* `public String getAbsolutePath() ` ：返回此File的绝对路径名字符串。

\* ` public String getPath() ` ：将此File转换为路径名字符串。

\* `public String getName()` ：返回由此File表示的文件或目录的名称。

\* `public long length()` ：返回由此File表示的文件的长度。

方法演示，代码如下：

```java

public class FileGet {

public static void main(String[] args) {

File f = new File("d:/aaa/bbb.java");

System.out.println("文件绝对路径:"+f.getAbsolutePath());

System.out.println("文件构造路径:"+f.getPath());

System.out.println("文件名称:"+f.getName());

System.out.println("文件长度:"+f.length()+"字节");

File f2 = new File("d:/aaa");

System.out.println("目录绝对路径:"+f2.getAbsolutePath());

System.out.println("目录构造路径:"+f2.getPath());

System.out.println("目录名称:"+f2.getName());

System.out.println("目录长度:"+f2.length());

}

}

输出结果：

文件绝对路径:d:\aaa\bbb.java

文件构造路径:d:\aaa\bbb.java

文件名称:bbb.java

文件长度:636字节

目录绝对路径:d:\aaa

目录构造路径:d:\aaa

目录名称:aaa

目录长度:4096

```

> API中说明：length()，表示文件的长度。但是File对象表示目录，则返回值未指定。

### 绝对路径和相对路径

\* \*\*绝对路径\*\*：从盘符开始的路径，这是一个完整的路径。

\* \*\*相对路径\*\*：相对于项目目录的路径，这是一个便捷的路径，开发中经常使用。

```java

public class FilePath {

public static void main(String[] args) {

// D盘下的bbb.java文件

File f = new File("D:\\bbb.java");

System.out.println(f.getAbsolutePath());

// 项目下的bbb.java文件

File f2 = new File("bbb.java");

System.out.println(f2.getAbsolutePath());

}

}

输出结果：

D:\bbb.java

D:\idea\_project\_test4\bbb.java

```

### 判断功能的方法

- `public boolean exists()` ：此File表示的文件或目录是否实际存在。

- `public boolean isDirectory()` ：此File表示的是否为目录。

- `public boolean isFile()` ：此File表示的是否为文件。

方法演示，代码如下：

```java

public class FileIs {

public static void main(String[] args) {

File f = new File("d:\\aaa\\bbb.java");

File f2 = new File("d:\\aaa");

// 判断是否存在

System.out.println("d:\\aaa\\bbb.java 是否存在:"+f.exists());

System.out.println("d:\\aaa 是否存在:"+f2.exists());

// 判断是文件还是目录

System.out.println("d:\\aaa 文件?:"+f2.isFile());

System.out.println("d:\\aaa 目录?:"+f2.isDirectory());

}

}

输出结果：

d:\aaa\bbb.java 是否存在:true

d:\aaa 是否存在:true

d:\aaa 文件?:false

d:\aaa 目录?:true

```

### 创建删除功能的方法

- `public boolean createNewFile()` ：当且仅当具有该名称的文件尚不存在时，创建一个新的空文件。

- `public boolean delete()` ：删除由此File表示的文件或目录。

- `public boolean mkdir()` ：创建由此File表示的目录。

- `public boolean mkdirs()` ：创建由此File表示的目录，包括任何必需但不存在的父目录。

方法演示，代码如下：

```java

public class FileCreateDelete {

public static void main(String[] args) throws IOException {

// 文件的创建

File f = new File("aaa.txt");

System.out.println("是否存在:"+f.exists()); // false

System.out.println("是否创建:"+f.createNewFile()); // true

System.out.println("是否存在:"+f.exists()); // true

// 目录的创建

File f2= new File("newDir");

System.out.println("是否存在:"+f2.exists());// false

System.out.println("是否创建:"+f2.mkdir()); // true

System.out.println("是否存在:"+f2.exists());// true

// 创建多级目录

File f3= new File("newDira\\newDirb");

System.out.println(f3.mkdir());// false

File f4= new File("newDira\\newDirb");

System.out.println(f4.mkdirs());// true

// 文件的删除

System.out.println(f.delete());// true

// 目录的删除

System.out.println(f2.delete());// true

System.out.println(f4.delete());// false

}

}

```

> API中说明：delete方法，如果此File表示目录，则目录必须为空才能删除。

## 1.4 目录的遍历

\* `public String[] list()` ：返回一个String数组，表示该File目录中的所有子文件或目录。

\* `public File[] listFiles()` ：返回一个File数组，表示该File目录中的所有的子文件或目录。

```java

public class FileFor {

public static void main(String[] args) {

File dir = new File("d:\\java\_code");

//获取当前目录下的文件以及文件夹的名称。

String[] names = dir.list();

for(String name : names){

System.out.println(name);

}

//获取当前目录下的文件以及文件夹对象，只要拿到了文件对象，那么就可以获取更多信息

File[] files = dir.listFiles();

for (File file : files) {

System.out.println(file);

}

}

}

```

> 小贴士：

>

> 调用listFiles方法的File对象，表示的必须是实际存在的目录，否则返回null，无法进行遍历。

# 第二章 递归

## 2.1 概述

\* \*\*递归\*\*：指在当前方法内调用自己的这种现象。

\* \*\*递归的分类:\*\*

- 递归分为两种，直接递归和间接递归。

- 直接递归称为方法自身调用自己。

- 间接递归可以A方法调用B方法，B方法调用C方法，C方法调用A方法。

\* \*\*注意事项\*\*：

- 递归一定要有条件限定，保证递归能够停止下来，否则会发生栈内存溢出。

- 在递归中虽然有限定条件，但是递归次数不能太多。否则也会发生栈内存溢出。

- 构造方法,禁止递归

```java

public class Demo01DiGui {

public static void main(String[] args) {

// a();

b(1);

}

/\*

\* 3.构造方法,禁止递归

\* 编译报错:构造方法是创建对象使用的,不能让对象一直创建下去

\*/

public Demo01DiGui() {

//Demo01DiGui();

}

/\*

\* 2.在递归中虽然有限定条件，但是递归次数不能太多。否则也会发生栈内存溢出。

\* 4993

\* Exception in thread "main" java.lang.StackOverflowError

\*/

private static void b(int i) {

System.out.println(i);

//添加一个递归结束的条件,i==5000的时候结束

if(i==5000){

return;//结束方法

}

b(++i);

}

/\*

\* 1.递归一定要有条件限定，保证递归能够停止下来，否则会发生栈内存溢出。 Exception in thread "main"

\* java.lang.StackOverflowError

\*/

private static void a() {

System.out.println("a方法");

a();

}

}

```

## 2.2 递归累加求和

### 计算1 ~ n的和

\*\*分析\*\*：num的累和 = num + (num-1)的累和，所以可以把累和的操作定义成一个方法，递归调用。

\*\*实现代码\*\*：

```java

public class DiGuiDemo {

public static void main(String[] args) {

//计算1~num的和，使用递归完成

int num = 5;

// 调用求和的方法

int sum = getSum(num);

// 输出结果

System.out.println(sum);

}

/\*

通过递归算法实现.

参数列表:int

返回值类型: int

\*/

public static int getSum(int num) {

/\*

num为1时,方法返回1,

相当于是方法的出口,num总有是1的情况

\*/

if(num == 1){

return 1;

}

/\*

num不为1时,方法返回 num +(num-1)的累和

递归调用getSum方法

\*/

return num + getSum(num-1);

}

}

```

### 代码执行图解

![](img/day08\_01\_递归累和.jpg)

> 小贴士：递归一定要有条件限定，保证递归能够停止下来，次数不要太多，否则会发生栈内存溢出。

## 2.3 递归求阶乘

\* \*\*阶乘\*\*：所有小于及等于该数的正整数的积。

```java

n的阶乘：n! = n \* (n-1) \*...\* 3 \* 2 \* 1

```

\*\*分析\*\*：这与累和类似,只不过换成了乘法运算，学员可以自己练习，需要注意阶乘值符合int类型的范围。

```

推理得出：n! = n \* (n-1)!

```

\*\*代码实现\*\*：

```java

public class DiGuiDemo {

//计算n的阶乘，使用递归完成

public static void main(String[] args) {

int n = 3;

// 调用求阶乘的方法

int value = getValue(n);

// 输出结果

System.out.println("阶乘为:"+ value);

}

/\*

通过递归算法实现.

参数列表:int

返回值类型: int

\*/

public static int getValue(int n) {

// 1的阶乘为1

if (n == 1) {

return 1;

}

/\*

n不为1时,方法返回 n! = n\*(n-1)!

递归调用getValue方法

\*/

return n \* getValue(n - 1);

}

}

```

## 2.4 递归打印多级目录

\*\*分析\*\*：多级目录的打印，就是当目录的嵌套。遍历之前，无从知道到底有多少级目录，所以我们还是要使用递归实现。

\*\*代码实现\*\*：

```java

public class DiGuiDemo2 {

public static void main(String[] args) {

// 创建File对象

File dir = new File("D:\\aaa");

// 调用打印目录方法

printDir(dir);

}

public static void printDir(File dir) {

// 获取子文件和目录

File[] files = dir.listFiles();

// 循环打印

/\*

判断:

当是文件时,打印绝对路径.

当是目录时,继续调用打印目录的方法,形成递归调用.

\*/

for (File file : files) {

// 判断

if (file.isFile()) {

// 是文件,输出文件绝对路径

System.out.println("文件名:"+ file.getAbsolutePath());

} else {

// 是目录,输出目录绝对路径

System.out.println("目录:"+file.getAbsolutePath());

// 继续遍历,调用printDir,形成递归

printDir(file);

}

}

}

}

```

1. 字节流、字符流

# 第一章 IO概述

## 1.1 什么是IO

生活中，你肯定经历过这样的场景。当你编辑一个文本文件，忘记了`ctrl+s` ，可能文件就白白编辑了。当你电脑上插入一个U盘，可以把一个视频，拷贝到你的电脑硬盘里。那么数据都是在哪些设备上的呢？键盘、内存、硬盘、外接设备等等。

我们把这种数据的传输，可以看做是一种数据的流动，按照流动的方向，以内存为基准，分为`输入input` 和`输出output` ，即流向内存是输入流，流出内存的输出流。

Java中I/O操作主要是指使用`java.io`包下的内容，进行输入、输出操作。\*\*输入\*\*也叫做\*\*读取\*\*数据，\*\*输出\*\*也叫做作\*\*写出\*\*数据。

## 1.2 IO的分类

根据数据的流向分为：\*\*输入流\*\*和\*\*输出流\*\*。

\* \*\*输入流\*\* ：把数据从`其他设备`上读取到`内存`中的流。

\* \*\*输出流\*\* ：把数据从`内存` 中写出到`其他设备`上的流。

格局数据的类型分为：\*\*字节流\*\*和\*\*字符流\*\*。

\* \*\*字节流\*\* ：以字节为单位，读写数据的流。

\* \*\*字符流\*\* ：以字符为单位，读写数据的流。

## 1.3 IO的流向说明图解

![](img/1\_io.jpg)

## 1.4 顶级父类们

| | \*\*输入流\*\* | 输出流 |

| :-----: | :------------------------: | :-------------------------: |

| \*\*字节流\*\* | 字节输入流<br />\*\*InputStream\*\* | 字节输出流<br />\*\*OutputStream\*\* |

| \*\*字符流\*\* | 字符输入流<br />\*\*Reader\*\* | 字符输出流<br />\*\*Writer\*\* |

# 第二章 字节流

## 2.1 一切皆为字节

一切文件数据(文本、图片、视频等)在存储时，都是以二进制数字的形式保存，都一个一个的字节，那么传输时一样如此。所以，字节流可以传输任意文件数据。在操作流的时候，我们要时刻明确，无论使用什么样的流对象，底层传输的始终为二进制数据。

## 2.2 字节输出流【OutputStream】

`java.io.OutputStream `抽象类是表示字节输出流的所有类的超类，将指定的字节信息写出到目的地。它定义了字节输出流的基本共性功能方法。

\* `public void close()` ：关闭此输出流并释放与此流相关联的任何系统资源。

\* `public void flush() ` ：刷新此输出流并强制任何缓冲的输出字节被写出。

\* `public void write(byte[] b)`：将 b.length字节从指定的字节数组写入此输出流。

\* `public void write(byte[] b, int off, int len)` ：从指定的字节数组写入 len字节，从偏移量 off开始输出到此输出流。

\* `public abstract void write(int b)` ：将指定的字节输出流。

> 小贴士：

>

> close方法，当完成流的操作时，必须调用此方法，释放系统资源。

## 2.3 FileOutputStream类

`OutputStream`有很多子类，我们从最简单的一个子类开始。

`java.io.FileOutputStream `类是文件输出流，用于将数据写出到文件。

### 构造方法

\* `public FileOutputStream(File file)`：创建文件输出流以写入由指定的 File对象表示的文件。

\* `public FileOutputStream(String name)`： 创建文件输出流以指定的名称写入文件。

当你创建一个流对象时，必须传入一个文件路径。该路径下，如果没有这个文件，会创建该文件。如果有这个文件，会清空这个文件的数据。

\* 构造举例，代码如下：

```java

public class FileOutputStreamConstructor throws IOException {

public static void main(String[] args) {

// 使用File对象创建流对象

File file = new File("a.txt");

FileOutputStream fos = new FileOutputStream(file);

// 使用文件名称创建流对象

FileOutputStream fos = new FileOutputStream("b.txt");

}

}

```

### 写出字节数据

1. \*\*写出字节\*\*：`write(int b)` 方法，每次可以写出一个字节数据，代码使用演示：

```java

public class FOSWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileOutputStream fos = new FileOutputStream("fos.txt");

// 写出数据

fos.write(97); // 写出第1个字节

fos.write(98); // 写出第2个字节

fos.write(99); // 写出第3个字节

// 关闭资源

fos.close();

}

}

输出结果：

abc

```

> 小贴士：

>

> 1. 虽然参数为int类型四个字节，但是只会保留一个字节的信息写出。

> 2. 流操作完毕后，必须释放系统资源，调用close方法，千万记得。

2. \*\*写出字节数组\*\*：`write(byte[] b)`，每次可以写出数组中的数据，代码使用演示：

```java

public class FOSWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileOutputStream fos = new FileOutputStream("fos.txt");

// 字符串转换为字节数组

byte[] b = "黑马程序员".getBytes();

// 写出字节数组数据

fos.write(b);

// 关闭资源

fos.close();

}

}

输出结果：

黑马程序员

```

3. \*\*写出指定长度字节数组\*\*：`write(byte[] b, int off, int len)` ,每次写出从off索引开始，len个字节，代码使用演示：

```java

public class FOSWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileOutputStream fos = new FileOutputStream("fos.txt");

// 字符串转换为字节数组

byte[] b = "abcde".getBytes();

// 写出从索引2开始，2个字节。索引2是c，两个字节，也就是cd。

fos.write(b,2,2);

// 关闭资源

fos.close();

}

}

输出结果：

cd

```

### 数据追加续写

经过以上的演示，每次程序运行，创建输出流对象，都会清空目标文件中的数据。如何保留目标文件中数据，还能继续添加新数据呢？

- `public FileOutputStream(File file, boolean append)`： 创建文件输出流以写入由指定的 File对象表示的文件。

- `public FileOutputStream(String name, boolean append)`： 创建文件输出流以指定的名称写入文件。

这两个构造方法，参数中都需要传入一个boolean类型的值，`true` 表示追加数据，`false` 表示清空原有数据。这样创建的输出流对象，就可以指定是否追加续写了，代码使用演示：

```java

public class FOSWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileOutputStream fos = new FileOutputStream("fos.txt"，true);

// 字符串转换为字节数组

byte[] b = "abcde".getBytes();

// 写出从索引2开始，2个字节。索引2是c，两个字节，也就是cd。

fos.write(b);

// 关闭资源

fos.close();

}

}

文件操作前：cd

文件操作后：cdabcde

```

### 写出换行

Windows系统里，换行符号是`\r\n` 。把

以指定是否追加续写了，代码使用演示：

```java

public class FOSWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileOutputStream fos = new FileOutputStream("fos.txt");

// 定义字节数组

byte[] words = {97,98,99,100,101};

// 遍历数组

for (int i = 0; i < words.length; i++) {

// 写出一个字节

fos.write(words[i]);

// 写出一个换行, 换行符号转成数组写出

fos.write("\r\n".getBytes());

}

// 关闭资源

fos.close();

}

}

输出结果：

a

b

c

d

e

```

> \* 回车符`\r`和换行符`\n` ：

> \* 回车符：回到一行的开头（return）。

> \* 换行符：下一行（newline）。

> \* 系统中的换行：

> \* Windows系统里，每行结尾是 `回车+换行` ，即`\r\n`；

> \* Unix系统里，每行结尾只有 `换行` ，即`\n`；

> \* Mac系统里，每行结尾是 `回车` ，即`\r`。从 Mac OS X开始与Linux统一。

## 2.4 字节输入流【InputStream】

`java.io.InputStream `抽象类是表示字节输入流的所有类的超类，可以读取字节信息到内存中。它定义了字节输入流的基本共性功能方法。

- `public void close()` ：关闭此输入流并释放与此流相关联的任何系统资源。

- `public abstract int read()`： 从输入流读取数据的下一个字节。

- `public int read(byte[] b)`： 从输入流中读取一些字节数，并将它们存储到字节数组 b中 。

> 小贴士：

>

> close方法，当完成流的操作时，必须调用此方法，释放系统资源。

## 2.5 FileInputStream类

`java.io.FileInputStream `类是文件输入流，从文件中读取字节。

### 构造方法

\* `FileInputStream(File file)`： 通过打开与实际文件的连接来创建一个 FileInputStream ，该文件由文件系统中的 File对象 file命名。

\* `FileInputStream(String name)`： 通过打开与实际文件的连接来创建一个 FileInputStream ，该文件由文件系统中的路径名 name命名。

当你创建一个流对象时，必须传入一个文件路径。该路径下，如果没有该文件,会抛出`FileNotFoundException` 。

- 构造举例，代码如下：

```java

public class FileInputStreamConstructor throws IOException{

public static void main(String[] args) {

// 使用File对象创建流对象

File file = new File("a.txt");

FileInputStream fos = new FileInputStream(file);

// 使用文件名称创建流对象

FileInputStream fos = new FileInputStream("b.txt");

}

}

```

### 读取字节数据

1. \*\*读取字节\*\*：`read`方法，每次可以读取一个字节的数据，提升为int类型，读取到文件末尾，返回`-1`，代码使用演示：

```java

public class FISRead {

public static void main(String[] args) throws IOException{

// 使用文件名称创建流对象

FileInputStream fis = new FileInputStream("read.txt");

// 读取数据，返回一个字节

int read = fis.read();

System.out.println((char) read);

read = fis.read();

System.out.println((char) read);

read = fis.read();

System.out.println((char) read);

read = fis.read();

System.out.println((char) read);

read = fis.read();

System.out.println((char) read);

// 读取到末尾,返回-1

read = fis.read();

System.out.println( read);

// 关闭资源

fis.close();

}

}

输出结果：

a

b

c

d

e

-1

```

循环改进读取方式，代码使用演示：

```java

public class FISRead {

public static void main(String[] args) throws IOException{

// 使用文件名称创建流对象

FileInputStream fis = new FileInputStream("read.txt");

// 定义变量，保存数据

int b ；

// 循环读取

while ((b = fis.read())!=-1) {

System.out.println((char)b);

}

// 关闭资源

fis.close();

}

}

输出结果：

a

b

c

d

e

```

> 小贴士：

>

> 1. 虽然读取了一个字节，但是会自动提升为int类型。

> 2. 流操作完毕后，必须释放系统资源，调用close方法，千万记得。

2. \*\*使用字节数组读取\*\*：`read(byte[] b)`，每次读取b的长度个字节到数组中，返回读取到的有效字节个数，读取到末尾时，返回`-1` ，代码使用演示：

```java

public class FISRead {

public static void main(String[] args) throws IOException{

// 使用文件名称创建流对象.

FileInputStream fis = new FileInputStream("read.txt"); // 文件中为abcde

// 定义变量，作为有效个数

int len ；

// 定义字节数组，作为装字节数据的容器

byte[] b = new byte[2];

// 循环读取

while (( len= fis.read(b))!=-1) {

// 每次读取后,把数组变成字符串打印

System.out.println(new String(b));

}

// 关闭资源

fis.close();

}

}

输出结果：

ab

cd

ed

```

错误数据`d`，是由于最后一次读取时，只读取一个字节`e`，数组中，上次读取的数据没有被完全替换，所以要通过`len` ，获取有效的字节，代码使用演示：

```java

public class FISRead {

public static void main(String[] args) throws IOException{

// 使用文件名称创建流对象.

FileInputStream fis = new FileInputStream("read.txt"); // 文件中为abcde

// 定义变量，作为有效个数

int len ；

// 定义字节数组，作为装字节数据的容器

byte[] b = new byte[2];

// 循环读取

while (( len= fis.read(b))!=-1) {

// 每次读取后,把数组的有效字节部分，变成字符串打印

System.out.println(new String(b，0，len));// len 每次读取的有效字节个数

}

// 关闭资源

fis.close();

}

}

输出结果：

ab

cd

e

```

> 小贴士：

>

> 使用数组读取，每次读取多个字节，减少了系统间的IO操作次数，从而提高了读写的效率，建议开发中使用。

## 2.6 字节流练习：图片复制

### 复制原理图解

![](img/2\_copy.jpg)

### 案例实现

复制图片文件，代码使用演示：

```java

public class Copy {

public static void main(String[] args) throws IOException {

// 1.创建流对象

// 1.1 指定数据源

FileInputStream fis = new FileInputStream("D:\\test.jpg");

// 1.2 指定目的地

FileOutputStream fos = new FileOutputStream("test\_copy.jpg");

// 2.读写数据

// 2.1 定义数组

byte[] b = new byte[1024];

// 2.2 定义长度

int len;

// 2.3 循环读取

while ((len = fis.read(b))!=-1) {

// 2.4 写出数据

fos.write(b, 0 , len);

}

// 3.关闭资源

fos.close();

fis.close();

}

}

```

> 小贴士：

>

> 流的关闭原则：先开后关，后开先关。

# 第三章 字符流

当使用字节流读取文本文件时，可能会有一个小问题。就是遇到中文字符时，可能不会显示完整的字符，那是因为一个中文字符可能占用多个字节存储。所以Java提供一些字符流类，以字符为单位读写数据，专门用于处理文本文件。

## 3.1 字符输入流【Reader】

`java.io.Reader`抽象类是表示用于读取字符流的所有类的超类，可以读取字符信息到内存中。它定义了字符输入流的基本共性功能方法。

- `public void close()` ：关闭此流并释放与此流相关联的任何系统资源。

- `public int read()`： 从输入流读取一个字符。

- `public int read(char[] cbuf)`： 从输入流中读取一些字符，并将它们存储到字符数组 cbuf中 。

## 3.2 FileReader类

`java.io.FileReader `类是读取字符文件的便利类。构造时使用系统默认的字符编码和默认字节缓冲区。

> 小贴士：

>

> 1. 字符编码：字节与字符的对应规则。Windows系统的中文编码默认是GBK编码表。

>

> idea中UTF-8

>

> 2. 字节缓冲区：一个字节数组，用来临时存储字节数据。

### 构造方法

- `FileReader(File file)`： 创建一个新的 FileReader ，给定要读取的File对象。

- `FileReader(String fileName)`： 创建一个新的 FileReader ，给定要读取的文件的名称。

当你创建一个流对象时，必须传入一个文件路径。类似于FileInputStream 。

- 构造举例，代码如下：

```java

public class FileReaderConstructor throws IOException{

public static void main(String[] args) {

// 使用File对象创建流对象

File file = new File("a.txt");

FileReader fr = new FileReader(file);

// 使用文件名称创建流对象

FileReader fr = new FileReader("b.txt");

}

}

```

### 读取字符数据

1. \*\*读取字符\*\*：`read`方法，每次可以读取一个字符的数据，提升为int类型，读取到文件末尾，返回`-1`，循环读取，代码使用演示：

```java

public class FRRead {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileReader fr = new FileReader("read.txt");

// 定义变量，保存数据

int b ；

// 循环读取

while ((b = fr.read())!=-1) {

System.out.println((char)b);

}

// 关闭资源

fr.close();

}

}

输出结果：

黑

马

程

序

员

```

> 小贴士：虽然读取了一个字符，但是会自动提升为int类型。

>

2. \*\*使用字符数组读取\*\*：`read(char[] cbuf)`，每次读取b的长度个字符到数组中，返回读取到的有效字符个数，读取到末尾时，返回`-1` ，代码使用演示：

```java

public class FRRead {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileReader fr = new FileReader("read.txt");

// 定义变量，保存有效字符个数

int len ；

// 定义字符数组，作为装字符数据的容器

char[] cbuf = new char[2];

// 循环读取

while ((len = fr.read(cbuf))!=-1) {

System.out.println(new String(cbuf));

}

// 关闭资源

fr.close();

}

}

输出结果：

黑马

程序

员序

```

获取有效的字符改进，代码使用演示：

```java

public class FISRead {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileReader fr = new FileReader("read.txt");

// 定义变量，保存有效字符个数

int len ；

// 定义字符数组，作为装字符数据的容器

char[] cbuf = new char[2];

// 循环读取

while ((len = fr.read(cbuf))!=-1) {

System.out.println(new String(cbuf,0,len));

}

// 关闭资源

fr.close();

}

}

输出结果：

黑马

程序

员

```

## 3.3 字符输出流【Writer】

`java.io.Writer `抽象类是表示用于写出字符流的所有类的超类，将指定的字符信息写出到目的地。它定义了字节输出流的基本共性功能方法。

- `void write(int c)` 写入单个字符。

- `void write(char[] cbuf) `写入字符数组。

- `abstract void write(char[] cbuf, int off, int len) `写入字符数组的某一部分,off数组的开始索引,len写的字符个数。

- `void write(String str) `写入字符串。

- `void write(String str, int off, int len)` 写入字符串的某一部分,off字符串的开始索引,len写的字符个数。

- `void flush() `刷新该流的缓冲。

- `void close()` 关闭此流，但要先刷新它。

## 3.4 FileWriter类

`java.io.FileWriter `类是写出字符到文件的便利类。构造时使用系统默认的字符编码和默认字节缓冲区。

### 构造方法

- `FileWriter(File file)`： 创建一个新的 FileWriter，给定要读取的File对象。

- `FileWriter(String fileName)`： 创建一个新的 FileWriter，给定要读取的文件的名称。

当你创建一个流对象时，必须传入一个文件路径，类似于FileOutputStream。

- 构造举例，代码如下：

```java

public class FileWriterConstructor {

public static void main(String[] args) throws IOException {

// 使用File对象创建流对象

File file = new File("a.txt");

FileWriter fw = new FileWriter(file);

// 使用文件名称创建流对象

FileWriter fw = new FileWriter("b.txt");

}

}

```

### 基本写出数据

\*\*写出字符\*\*：`write(int b)` 方法，每次可以写出一个字符数据，代码使用演示：

```java

public class FWWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileWriter fw = new FileWriter("fw.txt");

// 写出数据

fw.write(97); // 写出第1个字符

fw.write('b'); // 写出第2个字符

fw.write('C'); // 写出第3个字符

fw.write(30000); // 写出第4个字符，中文编码表中30000对应一个汉字。

/\*

【注意】关闭资源时,与FileOutputStream不同。

如果不关闭,数据只是保存到缓冲区，并未保存到文件。

\*/

// fw.close();

}

}

输出结果：

abC田

```

> 小贴士：

>

> 1. 虽然参数为int类型四个字节，但是只会保留一个字符的信息写出。

> 2. 未调用close方法，数据只是保存到了缓冲区，并未写出到文件中。

### 关闭和刷新

因为内置缓冲区的原因，如果不关闭输出流，无法写出字符到文件中。但是关闭的流对象，是无法继续写出数据的。如果我们既想写出数据，又想继续使用流，就需要`flush` 方法了。

\* `flush` ：刷新缓冲区，流对象可以继续使用。

\* `close `:先刷新缓冲区，然后通知系统释放资源。流对象不可以再被使用了。

代码使用演示：

```java

public class FWWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileWriter fw = new FileWriter("fw.txt");

// 写出数据，通过flush

fw.write('刷'); // 写出第1个字符

fw.flush();

fw.write('新'); // 继续写出第2个字符，写出成功

fw.flush();

// 写出数据，通过close

fw.write('关'); // 写出第1个字符

fw.close();

fw.write('闭'); // 继续写出第2个字符,【报错】java.io.IOException: Stream closed

fw.close();

}

}

```

> 小贴士：即便是flush方法写出了数据，操作的最后还是要调用close方法，释放系统资源。

### 写出其他数据

1. \*\*写出字符数组\*\* ：`write(char[] cbuf)` 和 `write(char[] cbuf, int off, int len)` ，每次可以写出字符数组中的数据，用法类似FileOutputStream，代码使用演示：

```java

public class FWWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileWriter fw = new FileWriter("fw.txt");

// 字符串转换为字节数组

char[] chars = "黑马程序员".toCharArray();

// 写出字符数组

fw.write(chars); // 黑马程序员

// 写出从索引2开始，2个字节。索引2是'程'，两个字节，也就是'程序'。

fw.write(b,2,2); // 程序

// 关闭资源

fos.close();

}

}

```

2. \*\*写出字符串\*\*：`write(String str)` 和 `write(String str, int off, int len)` ，每次可以写出字符串中的数据，更为方便，代码使用演示：

```java

public class FWWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象

FileWriter fw = new FileWriter("fw.txt");

// 字符串

String msg = "黑马程序员";

// 写出字符数组

fw.write(msg); //黑马程序员

// 写出从索引2开始，2个字节。索引2是'程'，两个字节，也就是'程序'。

fw.write(msg,2,2); // 程序

// 关闭资源

fos.close();

}

}

```

3. \*\*续写和换行\*\*：操作类似于FileOutputStream。

```java

public class FWWrite {

public static void main(String[] args) throws IOException {

// 使用文件名称创建流对象，可以续写数据

FileWriter fw = new FileWriter("fw.txt"，true);

// 写出字符串

fw.write("黑马");

// 写出换行

fw.write("\r\n");

// 写出字符串

fw.write("程序员");

// 关闭资源

fw.close();

}

}

输出结果:

黑马

程序员

```

> 小贴士：字符流，只能操作文本文件，不能操作图片，视频等非文本文件。

>

> 当我们单纯读或者写文本文件时 使用字符流 其他情况使用字节流

# 第五章 属性集

## 5.1 概述

`java.util.Properties ` 继承于` Hashtable` ，来表示一个持久的属性集。它使用键值结构存储数据，每个键及其对应值都是一个字符串。该类也被许多Java类使用，比如获取系统属性时，`System.getProperties` 方法就是返回一个`Properties`对象。

## 5.2 Properties类

### 构造方法

- `public Properties()` :创建一个空的属性列表。

### 基本的存储方法

- `public Object setProperty(String key, String value)` ： 保存一对属性。

- `public String getProperty(String key) ` ：使用此属性列表中指定的键搜索属性值。

- `public Set<String> stringPropertyNames() ` ：所有键的名称的集合。

```java

public class ProDemo {

public static void main(String[] args) throws FileNotFoundException {

// 创建属性集对象

Properties properties = new Properties();

// 添加键值对元素

properties.setProperty("filename", "a.txt");

properties.setProperty("length", "209385038");

properties.setProperty("location", "D:\\a.txt");

// 打印属性集对象

System.out.println(properties);

// 通过键,获取属性值

System.out.println(properties.getProperty("filename"));

System.out.println(properties.getProperty("length"));

System.out.println(properties.getProperty("location"));

// 遍历属性集,获取所有键的集合

Set<String> strings = properties.stringPropertyNames();

// 打印键值对

for (String key : strings ) {

System.out.println(key+" -- "+properties.getProperty(key));

}

}

}

输出结果：

{filename=a.txt, length=209385038, location=D:\a.txt}

a.txt

209385038

D:\a.txt

filename -- a.txt

length -- 209385038

location -- D:\a.txt

```

### 与流相关的方法

- `public void load(InputStream inStream)`： 从字节输入流中读取键值对。

参数中使用了字节输入流，通过流对象，可以关联到某文件上，这样就能够加载文本中的数据了。文本数据格式:

```

filename=a.txt

length=209385038

location=D:\a.txt

```

加载代码演示：

```java

public class ProDemo2 {

public static void main(String[] args) throws FileNotFoundException {

// 创建属性集对象

Properties pro = new Properties();

// 加载文本中信息到属性集

pro.load(new FileInputStream("read.txt"));

// 遍历集合并打印

Set<String> strings = pro.stringPropertyNames();

for (String key : strings ) {

System.out.println(key+" -- "+pro.getProperty(key));

}

}

}

输出结果：

filename -- a.txt

length -- 209385038

location -- D:\a.txt

```

> 小贴士：文本中的数据，必须是键值对形式，可以使用空格、等号、冒号等符号分隔。

1. 缓冲流、转换流、序列化流、打印流

# 第一章 缓冲流

昨天学习了基本的一些流，作为IO流的入门，今天我们要见识一些更强大的流。比如能够高效读写的缓冲流，能够转换编码的转换流，能够持久化存储对象的序列化流等等。这些功能更为强大的流，都是在基本的流对象基础之上创建而来的，就像穿上铠甲的武士一样，相当于是对基本流对象的一种增强。

## 1.1 概述

缓冲流,也叫高效流，是对4个基本的`FileXxx` 流的增强，所以也是4个流，按照数据类型分类：

\* \*\*字节缓冲流\*\*：`BufferedInputStream`，`BufferedOutputStream`

\* \*\*字符缓冲流\*\*：`BufferedReader`，`BufferedWriter`

缓冲流的基本原理，是在创建流对象时，会创建一个内置的默认大小的缓冲区数组，通过缓冲区读写，减少系统IO次数，从而提高读写的效率。

## 1.2 字节缓冲流

### 构造方法

\* `public BufferedInputStream(InputStream in)` ：创建一个 新的缓冲输入流。

\* `public BufferedOutputStream(OutputStream out)`： 创建一个新的缓冲输出流。

构造举例，代码如下：

```java

// 创建字节缓冲输入流

BufferedInputStream bis = new BufferedInputStream(new FileInputStream("bis.txt"));

// 创建字节缓冲输出流

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("bos.txt"));

```

### 效率测试

查询API，缓冲流读写方法与基本的流是一致的，我们通过复制大文件（375MB），测试它的效率。

1. 基本流，代码如下：

```java

public class BufferedDemo {

public static void main(String[] args) throws FileNotFoundException {

// 记录开始时间

long start = System.currentTimeMillis();

// 创建流对象

try (

FileInputStream fis = new FileInputStream("jdk9.exe");

FileOutputStream fos = new FileOutputStream("copy.exe")

){

// 读写数据

int b;

while ((b = fis.read()) != -1) {

fos.write(b);

}

} catch (IOException e) {

e.printStackTrace();

}

// 记录结束时间

long end = System.currentTimeMillis();

System.out.println("普通流复制时间:"+(end - start)+" 毫秒");

}

}

十几分钟过去了...

```

2. 缓冲流，代码如下：

```java

public class BufferedDemo {

public static void main(String[] args) throws FileNotFoundException {

// 记录开始时间

long start = System.currentTimeMillis();

// 创建流对象

try (

BufferedInputStream bis = new BufferedInputStream(new FileInputStream("jdk9.exe"));

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("copy.exe"));

){

// 读写数据

int b;

while ((b = bis.read()) != -1) {

bos.write(b);

}

} catch (IOException e) {

e.printStackTrace();

}

// 记录结束时间

long end = System.currentTimeMillis();

System.out.println("缓冲流复制时间:"+(end - start)+" 毫秒");

}

}

缓冲流复制时间:8016 毫秒

```

如何更快呢？

使用数组的方式，代码如下：

```java

public class BufferedDemo {

public static void main(String[] args) throws FileNotFoundException {

// 记录开始时间

long start = System.currentTimeMillis();

// 创建流对象

try (

BufferedInputStream bis = new BufferedInputStream(new FileInputStream("jdk9.exe"));

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("copy.exe"));

){

// 读写数据

int len;

byte[] bytes = new byte[8\*1024];

while ((len = bis.read(bytes)) != -1) {

bos.write(bytes, 0 , len);

}

} catch (IOException e) {

e.printStackTrace();

}

// 记录结束时间

long end = System.currentTimeMillis();

System.out.println("缓冲流使用数组复制时间:"+(end - start)+" 毫秒");

}

}

缓冲流使用数组复制时间:666 毫秒

```

## 1.3 字符缓冲流

### 构造方法

\* `public BufferedReader(Reader in)` ：创建一个 新的缓冲输入流。

\* `public BufferedWriter(Writer out)`： 创建一个新的缓冲输出流。

构造举例，代码如下：

```java

// 创建字符缓冲输入流

BufferedReader br = new BufferedReader(new FileReader("br.txt"));

// 创建字符缓冲输出流

BufferedWriter bw = new BufferedWriter(new FileWriter("bw.txt"));

```

### 特有方法

字符缓冲流的基本方法与普通字符流调用方式一致，不再阐述，我们来看它们具备的特有方法。

\* BufferedReader：`public String readLine()`: 读一行文字。

\* BufferedWriter：`public void newLine()`: 写一行行分隔符,由系统属性定义符号。

`readLine`方法演示，代码如下：

```java

public class BufferedReaderDemo {

public static void main(String[] args) throws IOException {

// 创建流对象

BufferedReader br = new BufferedReader(new FileReader("in.txt"));

// 定义字符串,保存读取的一行文字

String line = null;

// 循环读取,读取到最后返回null

while ((line = br.readLine())!=null) {

System.out.print(line);

System.out.println("------");

}

// 释放资源

br.close();

}

}

```

`newLine`方法演示，代码如下：

```java

public class BufferedWriterDemo throws IOException {

public static void main(String[] args) throws IOException {

// 创建流对象

BufferedWriter bw = new BufferedWriter(new FileWriter("out.txt"));

// 写出数据

bw.write("黑马");

// 写出换行

bw.newLine();

bw.write("程序");

bw.newLine();

bw.write("员");

bw.newLine();

// 释放资源

bw.close();

}

}

输出效果:

黑马

程序

员

```

## 1.4 练习:文本排序

请将文本信息恢复顺序。

```

3.侍中、侍郎郭攸之、费祎、董允等，此皆良实，志虑忠纯，是以先帝简拔以遗陛下。愚以为宫中之事，事无大小，悉以咨之，然后施行，必得裨补阙漏，有所广益。

8.愿陛下托臣以讨贼兴复之效，不效，则治臣之罪，以告先帝之灵。若无兴德之言，则责攸之、祎、允等之慢，以彰其咎；陛下亦宜自谋，以咨诹善道，察纳雅言，深追先帝遗诏，臣不胜受恩感激。

4.将军向宠，性行淑均，晓畅军事，试用之于昔日，先帝称之曰能，是以众议举宠为督。愚以为营中之事，悉以咨之，必能使行阵和睦，优劣得所。

2.宫中府中，俱为一体，陟罚臧否，不宜异同。若有作奸犯科及为忠善者，宜付有司论其刑赏，以昭陛下平明之理，不宜偏私，使内外异法也。

1.先帝创业未半而中道崩殂，今天下三分，益州疲弊，此诚危急存亡之秋也。然侍卫之臣不懈于内，忠志之士忘身于外者，盖追先帝之殊遇，欲报之于陛下也。诚宜开张圣听，以光先帝遗德，恢弘志士之气，不宜妄自菲薄，引喻失义，以塞忠谏之路也。

9.今当远离，临表涕零，不知所言。

6.臣本布衣，躬耕于南阳，苟全性命于乱世，不求闻达于诸侯。先帝不以臣卑鄙，猥自枉屈，三顾臣于草庐之中，咨臣以当世之事，由是感激，遂许先帝以驱驰。后值倾覆，受任于败军之际，奉命于危难之间，尔来二十有一年矣。

7.先帝知臣谨慎，故临崩寄臣以大事也。受命以来，夙夜忧叹，恐付托不效，以伤先帝之明，故五月渡泸，深入不毛。今南方已定，兵甲已足，当奖率三军，北定中原，庶竭驽钝，攘除奸凶，兴复汉室，还于旧都。此臣所以报先帝而忠陛下之职分也。至于斟酌损益，进尽忠言，则攸之、祎、允之任也。

5.亲贤臣，远小人，此先汉所以兴隆也；亲小人，远贤臣，此后汉所以倾颓也。先帝在时，每与臣论此事，未尝不叹息痛恨于桓、灵也。侍中、尚书、长史、参军，此悉贞良死节之臣，愿陛下亲之信之，则汉室之隆，可计日而待也。

```

### 案例分析

1. 逐行读取文本信息。

2. 解析文本信息到集合中。

3. 遍历集合，按顺序，写出文本信息。

### 案例实现

```java

public class BufferedTest {

public static void main(String[] args) throws IOException {

// 创建map集合,保存文本数据,键为序号,值为文字

HashMap<String, String> lineMap = new HashMap<>();

// 创建流对象

BufferedReader br = new BufferedReader(new FileReader("in.txt"));

BufferedWriter bw = new BufferedWriter(new FileWriter("out.txt"));

// 读取数据

String line = null;

while ((line = br.readLine())!=null) {

// 解析文本

String[] split = line.split("\\.");

// 保存到集合

lineMap.put(split[0],split[1]);

}

// 释放资源

br.close();

// 遍历map集合

for (int i = 1; i <= lineMap.size(); i++) {

String key = String.valueOf(i);

// 获取map中文本

String value = lineMap.get(key);

// 写出拼接文本

bw.write(key+"."+value);

// 写出换行

bw.newLine();

}

// 释放资源

bw.close();

}

}

```

# 第二章 转换流

## 2.1 字符编码和字符集

### 字符编码

计算机中储存的信息都是用二进制数表示的，而我们在屏幕上看到的数字、英文、标点符号、汉字等字符是二进制数转换之后的结果。按照某种规则，将字符存储到计算机中，称为\*\*编码\*\* 。反之，将存储在计算机中的二进制数按照某种规则解析显示出来，称为\*\*解码\*\* 。比如说，按照A规则存储，同样按照A规则解析，那么就能显示正确的文本符号。反之，按照A规则存储，再按照B规则解析，就会导致乱码现象。

编码:字符(能看懂的)--字节(看不懂的)

解码:字节(看不懂的)-->字符(能看懂的)

\* \*\*字符编码`Character Encoding`\*\* : 就是一套自然语言的字符与二进制数之间的对应规则。

编码表:生活中文字和计算机中二进制的对应规则

### 字符集

\* \*\*字符集 `Charset`\*\*：也叫编码表。是一个系统支持的所有字符的集合，包括各国家文字、标点符号、图形符号、数字等。

计算机要准确的存储和识别各种字符集符号，需要进行字符编码，一套字符集必然至少有一套字符编码。常见字符集有ASCII字符集、GBK字符集、Unicode字符集等。![](img/1\_charset.jpg)

可见，当指定了\*\*编码\*\*，它所对应的\*\*字符集\*\*自然就指定了，所以\*\*编码\*\*才是我们最终要关心的。

\* \*\*ASCII字符集\*\* ：

\* ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）是基于拉丁字母的一套电脑编码系统，用于显示现代英语，主要包括控制字符（回车键、退格、换行键等）和可显示字符（英文大小写字符、阿拉伯数字和西文符号）。

\* 基本的ASCII字符集，使用7位（bits）表示一个字符，共128字符。ASCII的扩展字符集使用8位（bits）表示一个字符，共256字符，方便支持欧洲常用字符。

\* \*\*ISO-8859-1字符集\*\*：

\* 拉丁码表，别名Latin-1，用于显示欧洲使用的语言，包括荷兰、丹麦、德语、意大利语、西班牙语等。

\* ISO-8859-1使用单字节编码，兼容ASCII编码。

\* \*\*GBxxx字符集\*\*：

\* GB就是国标的意思，是为了显示中文而设计的一套字符集。

\* \*\*GB2312\*\*：简体中文码表。一个小于127的字符的意义与原来相同。但两个大于127的字符连在一起时，就表示一个汉字，这样大约可以组合了包含7000多个简体汉字，此外数学符号、罗马希腊的字母、日文的假名们都编进去了，连在ASCII里本来就有的数字、标点、字母都统统重新编了两个字节长的编码，这就是常说的"全角"字符，而原来在127号以下的那些就叫"半角"字符了。

\* \*\*GBK\*\*：最常用的中文码表。是在GB2312标准基础上的扩展规范，使用了双字节编码方案，共收录了21003个汉字，完全兼容GB2312标准，同时支持繁体汉字以及日韩汉字等。

\* \*\*GB18030\*\*：最新的中文码表。收录汉字70244个，采用多字节编码，每个字可以由1个、2个或4个字节组成。支持中国国内少数民族的文字，同时支持繁体汉字以及日韩汉字等。

\* \*\*Unicode字符集\*\* ：

\* Unicode编码系统为表达任意语言的任意字符而设计，是业界的一种标准，也称为统一码、标准万国码。

\* 它最多使用4个字节的数字来表达每个字母、符号，或者文字。有三种编码方案，UTF-8、UTF-16和UTF-32。最为常用的UTF-8编码。

\* UTF-8编码，可以用来表示Unicode标准中任何字符，它是电子邮件、网页及其他存储或传送文字的应用中，优先采用的编码。互联网工程工作小组（IETF）要求所有互联网协议都必须支持UTF-8编码。所以，我们开发Web应用，也要使用UTF-8编码。它使用一至四个字节为每个字符编码，编码规则：

1. 128个US-ASCII字符，只需一个字节编码。

2. 拉丁文等字符，需要二个字节编码。

3. 大部分常用字（含中文），使用三个字节编码。

4. 其他极少使用的Unicode辅助字符，使用四字节编码。

## 2.2 编码引出的问题

在IDEA中，使用`FileReader` 读取项目中的文本文件。由于IDEA的设置，都是默认的`UTF-8`编码，所以没有任何问题。但是，当读取Windows系统中创建的文本文件时，由于Windows系统的默认是GBK编码，就会出现乱码。

```java

public class ReaderDemo {

public static void main(String[] args) throws IOException {

FileReader fileReader = new FileReader("E:\\File\_GBK.txt");

int read;

while ((read = fileReader.read()) != -1) {

System.out.print((char)read);

}

fileReader.close();

}

}

输出结果：

���

```

那么如何读取GBK编码的文件呢？

## 2.3 InputStreamReader类

转换流`java.io.InputStreamReader`，是Reader的子类，是从字节流到字符流的桥梁。它读取字节，并使用指定的字符集将其解码为字符。它的字符集可以由名称指定，也可以接受平台的默认字符集。

### 构造方法

\* `InputStreamReader(InputStream in)`: 创建一个使用默认字符集的字符流。

\* `InputStreamReader(InputStream in, String charsetName)`: 创建一个指定字符集的字符流。

构造举例，代码如下：

```java

InputStreamReader isr = new InputStreamReader(new FileInputStream("in.txt"));

InputStreamReader isr2 = new InputStreamReader(new FileInputStream("in.txt") , "GBK");

```

### 指定编码读取

```java

public class ReaderDemo2 {

public static void main(String[] args) throws IOException {

// 定义文件路径,文件为gbk编码

String FileName = "E:\\file\_gbk.txt";

// 创建流对象,默认UTF8编码

InputStreamReader isr = new InputStreamReader(new FileInputStream(FileName));

// 创建流对象,指定GBK编码

InputStreamReader isr2 = new InputStreamReader(new FileInputStream(FileName) , "GBK");

// 定义变量,保存字符

int read;

// 使用默认编码字符流读取,乱码

while ((read = isr.read()) != -1) {

System.out.print((char)read); // ��Һ�

}

isr.close();

// 使用指定编码字符流读取,正常解析

while ((read = isr2.read()) != -1) {

System.out.print((char)read);// 大家好

}

isr2.close();

}

}

```

## 2.4 OutputStreamWriter类

转换流`java.io.OutputStreamWriter` ，是Writer的子类，是从字符流到字节流的桥梁。使用指定的字符集将字符编码为字节。它的字符集可以由名称指定，也可以接受平台的默认字符集。

### 构造方法

- `OutputStreamWriter(OutputStream in)`: 创建一个使用默认字符集的字符流。

- `OutputStreamWriter(OutputStream in, String charsetName)`: 创建一个指定字符集的字符流。

构造举例，代码如下：

```java

OutputStreamWriter isr = new OutputStreamWriter(new FileOutputStream("out.txt"));

OutputStreamWriter isr2 = new OutputStreamWriter(new FileOutputStream("out.txt") , "GBK");

```

### 指定编码写出

```java

public class OutputDemo {

public static void main(String[] args) throws IOException {

// 定义文件路径

String FileName = "E:\\out.txt";

// 创建流对象,默认UTF8编码

OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream(FileName));

// 写出数据

osw.write("你好"); // 保存为6个字节

osw.close();

// 定义文件路径

String FileName2 = "E:\\out2.txt";

// 创建流对象,指定GBK编码

OutputStreamWriter osw2 = new OutputStreamWriter(new FileOutputStream(FileName2),"GBK");

// 写出数据

osw2.write("你好");// 保存为4个字节

osw2.close();

}

}

```

### 转换流理解图解

\*\*转换流是字节与字符间的桥梁！\*\*![](img/2\_zhuanhuan.jpg)

## 2.5 练习：转换文件编码

将GBK编码的文本文件，转换为UTF-8编码的文本文件。

### 案例分析

1. 指定GBK编码的转换流，读取文本文件。

2. 使用UTF-8编码的转换流，写出文本文件。

### 案例实现

```java

public class TransDemo {

public static void main(String[] args) {

// 1.定义文件路径

String srcFile = "file\_gbk.txt";

String destFile = "file\_utf8.txt";

// 2.创建流对象

// 2.1 转换输入流,指定GBK编码

InputStreamReader isr = new InputStreamReader(new FileInputStream(srcFile) , "GBK");

// 2.2 转换输出流,默认utf8编码

OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream(destFile));

// 3.读写数据

// 3.1 定义数组

char[] cbuf = new char[1024];

// 3.2 定义长度

int len;

// 3.3 循环读取

while ((len = isr.read(cbuf))!=-1) {

// 循环写出

osw.write(cbuf,0,len);

}

// 4.释放资源

osw.close();

isr.close();

}

}

```

# 第三章 序列化

## 3.1 概述

Java 提供了一种对象\*\*序列化\*\*的机制。用一个字节序列可以表示一个对象，该字节序列包含该`对象的数据`、`对象的类型`和`对象中存储的属性`等信息。字节序列写出到文件之后，相当于文件中\*\*持久保存\*\*了一个对象的信息。

反之，该字节序列还可以从文件中读取回来，重构对象，对它进行\*\*反序列化\*\*。`对象的数据`、`对象的类型`和`对象中存储的数据`信息，都可以用来在内存中创建对象。看图理解序列化： ![](img/3\_xuliehua.jpg)

## 3.2 ObjectOutputStream类

`java.io.ObjectOutputStream ` 类，将Java对象的原始数据类型写出到文件,实现对象的持久存储。

### 构造方法

\* `public ObjectOutputStream(OutputStream out) `： 创建一个指定OutputStream的ObjectOutputStream。

构造举例，代码如下：

```java

FileOutputStream fileOut = new FileOutputStream("employee.txt");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

```

### 序列化操作

1. 一个对象要想序列化，必须满足两个条件:

\* 该类必须实现`java.io.Serializable ` 接口，`Serializable` 是一个标记接口，不实现此接口的类将不会使任何状态序列化或反序列化，会抛出`NotSerializableException` 。

\* 该类的所有属性必须是可序列化的。如果有一个属性不需要可序列化的，则该属性必须注明是瞬态的，使用`transient` 关键字修饰。

```java

public class Employee implements java.io.Serializable {

public String name;

public String address;

public transient int age; // transient瞬态修饰成员,不会被序列化

public void addressCheck() {

System.out.println("Address check : " + name + " -- " + address);

}

}

```

2.写出对象方法

\* `public final void writeObject (Object obj)` : 将指定的对象写出。

```java

public class SerializeDemo{

public static void main(String [] args) {

Employee e = new Employee();

e.name = "zhangsan";

e.address = "beiqinglu";

e.age = 20;

try {

// 创建序列化流对象

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.txt"));

// 写出对象

out.writeObject(e);

// 释放资源

out.close();

fileOut.close();

System.out.println("Serialized data is saved"); // 姓名，地址被序列化，年龄没有被序列化。

} catch(IOException i) {

i.printStackTrace();

}

}

}

输出结果：

Serialized data is saved

```

## 3.3 ObjectInputStream类

ObjectInputStream反序列化流，将之前使用ObjectOutputStream序列化的原始数据恢复为对象。

### 构造方法

\* `public ObjectInputStream(InputStream in) `： 创建一个指定InputStream的ObjectInputStream。

### 反序列化操作1

如果能找到一个对象的class文件，我们可以进行反序列化操作，调用`ObjectInputStream`读取对象的方法：

- `public final Object readObject ()` : 读取一个对象。

```java

public class DeserializeDemo {

public static void main(String [] args) {

Employee e = null;

try {

// 创建反序列化流

FileInputStream fileIn = new FileInputStream("employee.txt");

ObjectInputStream in = new ObjectInputStream(fileIn);

// 读取一个对象

e = (Employee) in.readObject();

// 释放资源

in.close();

fileIn.close();

}catch(IOException i) {

// 捕获其他异常

i.printStackTrace();

return;

}catch(ClassNotFoundException c) {

// 捕获类找不到异常

System.out.println("Employee class not found");

c.printStackTrace();

return;

}

// 无异常,直接打印输出

System.out.println("Name: " + e.name); // zhangsan

System.out.println("Address: " + e.address); // beiqinglu

System.out.println("age: " + e.age); // 0

}

}

```

\*\*对于JVM可以反序列化对象，它必须是能够找到class文件的类。如果找不到该类的class文件，则抛出一个 `ClassNotFoundException` 异常。\*\*

### \*\*反序列化操作2\*\*

\*\*另外，当JVM反序列化对象时，能找到class文件，但是class文件在序列化对象之后发生了修改，那么反序列化操作也会失败，抛出一个`InvalidClassException`异常。\*\*发生这个异常的原因如下：

\* 该类的序列版本号与从流中读取的类描述符的版本号不匹配

\* 该类包含未知数据类型

\* 该类没有可访问的无参数构造方法

`Serializable` 接口给需要序列化的类，提供了一个序列版本号。`serialVersionUID` 该版本号的目的在于验证序列化的对象和对应类是否版本匹配。

```java

public class Employee implements java.io.Serializable {

// 加入序列版本号

private static final long serialVersionUID = 1L;

public String name;

public String address;

// 添加新的属性 ,重新编译, 可以反序列化,该属性赋为默认值.

public int eid;

public void addressCheck() {

System.out.println("Address check : " + name + " -- " + address);

}

}

```

## 3.4 练习：序列化集合

1. 将存有多个自定义对象的集合序列化操作，保存到`list.txt`文件中。

2. 反序列化`list.txt` ，并遍历集合，打印对象信息。

### 案例分析

1. 把若干学生对象 ，保存到集合中。

2. 把集合序列化。

3. 反序列化读取时，只需要读取一次，转换为集合类型。

4. 遍历集合，可以打印所有的学生信息

### 案例实现

```java

public class SerTest {

public static void main(String[] args) throws Exception {

// 创建 学生对象

Student student = new Student("老王", "laow");

Student student2 = new Student("老张", "laoz");

Student student3 = new Student("老李", "laol");

ArrayList<Student> arrayList = new ArrayList<>();

arrayList.add(student);

arrayList.add(student2);

arrayList.add(student3);

// 序列化操作

// serializ(arrayList);

// 反序列化

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("list.txt"));

// 读取对象,强转为ArrayList类型

ArrayList<Student> list = (ArrayList<Student>)ois.readObject();

for (int i = 0; i < list.size(); i++ ){

Student s = list.get(i);

System.out.println(s.getName()+"--"+ s.getPwd());

}

}

private static void serializ(ArrayList<Student> arrayList) throws Exception {

// 创建 序列化流

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("list.txt"));

// 写出对象

oos.writeObject(arrayList);

// 释放资源

oos.close();

}

}

```

# 第四章 打印流

## 4.1 概述

平时我们在控制台打印输出，是调用`print`方法和`println`方法完成的，这两个方法都来自于`java.io.PrintStream`类，该类能够方便地打印各种数据类型的值，是一种便捷的输出方式。

## 4.2 PrintStream类

### 构造方法

\* `public PrintStream(String fileName) `： 使用指定的文件名创建一个新的打印流。

构造举例，代码如下：

```java

PrintStream ps = new PrintStream("ps.txt")；

```

### 改变打印流向

`System.out`就是`PrintStream`类型的，只不过它的流向是系统规定的，打印在控制台上。不过，既然是流对象，我们就可以玩一个"小把戏"，改变它的流向。

```java

public class PrintDemo {

public static void main(String[] args) throws IOException {

// 调用系统的打印流,控制台直接输出97

System.out.println(97);

// 创建打印流,指定文件的名称

PrintStream ps = new PrintStream("ps.txt");

// 设置系统的打印流流向,输出到ps.txt

System.setOut(ps);

// 调用系统的打印流,ps.txt中输出97

System.out.println(97);

}

}

```

十一、网络编程

\*\*UDP\*\*：用户数据报协议(User Datagram Protocol)。UDP是无连接通信协议，即在数据传输时，数据的发送端和接收端不建立逻辑连接。简单来说，当一台计算机向另外一台计算机发送数据时，发送端不会确认接收端是否存在，就会发出数据，同样接收端在收到数据时，也不会向发送端反馈是否收到数据。

由于使用UDP协议消耗资源小，通信效率高，所以通常都会用于音频、视频和普通数据的传输例如视频会议都使用UDP协议，因为这种情况即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。

\*\*TCP\*\*：传输控制协议 (Transmission Control Protocol)。TCP协议是\*\*面向连接\*\*的通信协议，即传输数据之前，在发送端和接收端建立逻辑连接，然后再传输数据，它提供了两台计算机之间可靠无差错的数据传输。

在TCP连接中必须要明确客户端与服务器端，由客户端向服务端发出连接请求，每次连接的创建都需要经过“三次握手”。

- 三次握手：TCP协议中，在发送数据的准备阶段，客户端与服务器之间的三次交互，以保证连接的可靠。

- 第一次握手，客户端向服务器端发出连接请求，等待服务器确认。

- 第二次握手，服务器端向客户端回送一个响应，通知客户端收到了连接请求。

- 第三次握手，客户端再次向服务器端发送确认信息，确认连接。整个交互过程如下图所示。

2.4 简单的TCP网络程序

### TCP通信分析图解

1. 【服务端】启动,创建ServerSocket对象，等待连接。

2. 【客户端】启动,创建Socket对象，请求连接。

3. 【服务端】接收连接,调用accept方法，并返回一个Socket对象。

4. 【客户端】Socket对象，获取OutputStream，向服务端写出数据。

5. 【服务端】Scoket对象，获取InputStream，读取客户端发送的数据。

> 到此，客户端向服务端发送数据成功。

6. 【服务端】Socket对象，获取OutputStream，向客户端回写数据。

7. 【客户端】Scoket对象，获取InputStream，解析回写数据。

8. 【客户端】释放资源，断开连接。

### 客户端向服务器发送数据

\*\*服务端实现：\*\*

```java

public class ServerTCP {

public static void main(String[] args) throws IOException {

System.out.println("服务端启动 , 等待连接 .... ");

// 1.创建 ServerSocket对象，绑定端口，开始等待连接

ServerSocket ss = new ServerSocket(6666);

// 2.接收连接 accept 方法, 返回 socket 对象.

Socket server = ss.accept();

// 3.通过socket 获取输入流

InputStream is = server.getInputStream();

// 4.一次性读取数据

// 4.1 创建字节数组

byte[] b = new byte[1024];

// 4.2 据读取到字节数组中.

int len = is.read(b)；

// 4.3 解析数组,打印字符串信息

String msg = new String(b, 0, len);

System.out.println(msg);

//5.关闭资源.

is.close();

server.close();

}

}

```

\*\*客户端实现：\*\*

```java

public class ClientTCP {

public static void main(String[] args) throws Exception {

System.out.println("客户端 发送数据");

// 1.创建 Socket ( ip , port ) , 确定连接到哪里.

Socket client = new Socket("localhost", 6666);

// 2.获取流对象 . 输出流

OutputStream os = client.getOutputStream();

// 3.写出数据.

os.write("你好么? tcp ,我来了".getBytes());

// 4. 关闭资源 .

os.close();

client.close();

}

}

```

# 第三章 综合案例

## 3.1 文件上传案例

### 文件上传分析图解

1. 【客户端】输入流，从硬盘读取文件数据到程序中。

2. 【客户端】输出流，写出文件数据到服务端。

3. 【服务端】输入流，读取文件数据到服务端程序。

4. 【服务端】输出流，写出文件数据到服务器硬盘中。

![](img/6\_upload.jpg)

#### 基本实现

\*\*服务端实现：\*\*

```java

public class FileUpload\_Server {

public static void main(String[] args) throws IOException {

System.out.println("服务器 启动..... ");

// 1. 创建服务端ServerSocket

ServerSocket serverSocket = new ServerSocket(6666);

// 2. 建立连接

Socket accept = serverSocket.accept();

// 3. 创建流对象

// 3.1 获取输入流,读取文件数据

BufferedInputStream bis = new BufferedInputStream(accept.getInputStream());

// 3.2 创建输出流,保存到本地 .

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("copy.jpg"));

// 4. 读写数据

byte[] b = new byte[1024 \* 8];

int len;

while ((len = bis.read(b)) != -1) {

bos.write(b, 0, len);

}

//5. 关闭 资源

bos.close();

bis.close();

accept.close();

System.out.println("文件上传已保存");

}

}

```

\*\*客户端实现：\*\*

```java

public class FileUPload\_Client {

public static void main(String[] args) throws IOException {

// 1.创建流对象

// 1.1 创建输入流,读取本地文件

BufferedInputStream bis = new BufferedInputStream(new FileInputStream("test.jpg"));

// 1.2 创建输出流,写到服务端

Socket socket = new Socket("localhost", 6666);

BufferedOutputStream bos = new BufferedOutputStream(socket.getOutputStream());

//2.写出数据.

byte[] b = new byte[1024 \* 8 ];

int len ;

while (( len = bis.read(b))!=-1) {

bos.write(b, 0, len);

bos.flush();

}

System.out.println("文件发送完毕");

// 3.释放资源

bos.close();

socket.close();

bis.close();

System.out.println("文件上传完毕 ");

}

}

```