**第1章JAVA\_API及object类**

**1.1equals方法**

equals方法，用于比较两个对象是否相同，它其实就是使用两个对象的内存地址在比较。Object类中的equals方法内部使用的就是==比较运算符。

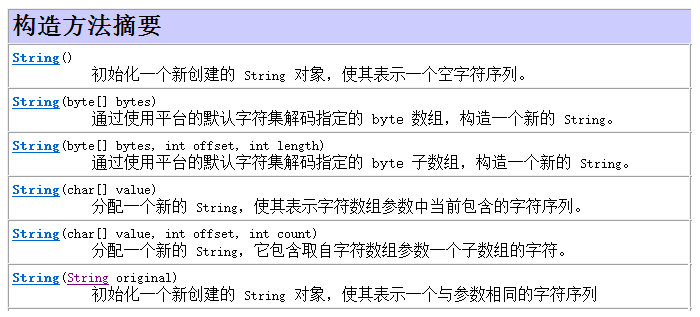
**1.2toString类**

toString方法返回该对象的字符串表示，其实该字符串内容就是对象的类型+@+内存地址值。

由于toString方法返回的结果是内存地址，而在开发中，经常需要按照对象的属性得到相应的字符串表现形式，因此也需要重写它。

**第2章String类**

**2.1String类构造方法**



String s1 = new String(); //创建String对象，字符串中没有内容

byte[] bys = new byte[]{97,98,99,100};

String s2 = new String(bys); // 创建String对象，把数组元素作为字符串的内容

String s3 = new String(bys, 1, 3); //创建String对象，把一部分数组元素作为字符串的内容，参数offset为数组元素的起始索引位置，参数length为要几个元素

char[] chs = new char[]{’a’,’b’,’c’,’d’,’e’};

String s4 = new String(chs); //创建String对象，把数组元素作为字符串的内容

String s5 = new String(chs, 0, 3);//创建String对象，把一部分数组元素作为字符串的内容，参数offset为数组元素的起始索引位置，参数count为要几个元素

String s6 = new String(“abc”); //创建String对象，字符串内容为abc

**2.2String类的方法查找**

* 字符串中有多少个字符?

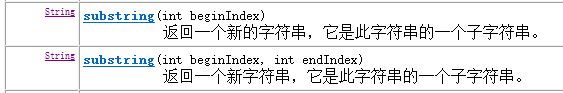


String str = "abcde";

int len = str.length();

System.out.println("len="+len);

* 获取部分字符串。



String str = "abcde";

String s1 = str.substring(1); //返回一个新字符串，内容为指定位置开始到字符串末尾的所有字符

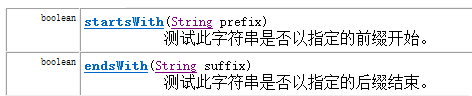
String s2 = str.substring(2, 4);//返回一个新字符串，内容为指定位置开始（包括开始）到指定位置结束（不包括结束）的所有字符

System.out.println("str="+str);

System.out.println("s1="+s1);

System.out.println("s2="+s2);

* 字符串是否以指定字符串开头。结尾同理。



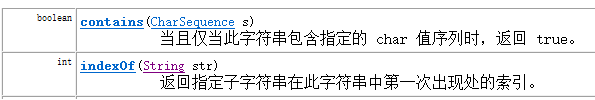
String str = "StringDemo.java";

boolean b1 = str.startsWith("Demo");//判断是否以给定字符串开头

boolean b2 = str.startsWith("String");

boolean b3 = str.endsWith("java");//判断是否以给定字符串结尾

* 字符串中是否包含另一个字符串。

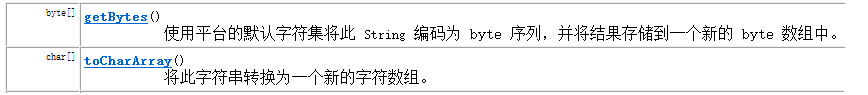


String str = "abcde";

int index = str.indexOf(“bcd”); //判断是否包含指定字符串，包含则返回第一次出现该字符串的索引，不包含则返回-1

boolean b2 = str.contains("bcd");//判断是否包含指定字符串，包含返回true，不包含返回false

* 将字符串转成一个字符数组。或者字节数组。

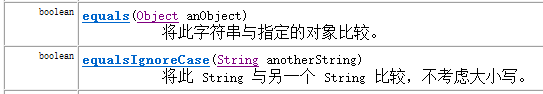


String str = "abcde";

char[] chs = str.toCharArray();

byte[] bytes = str.getBytes();

* 判断两个字符串中的内容是否相同



String str = "abcde";

String str2 = "abcde";

String str3 = "hello";

boolean b1 = str.equals(str2);

boolean b2 = str.equals(str3);

* 获取该字符串对象中的内容



String str = new String("hello");

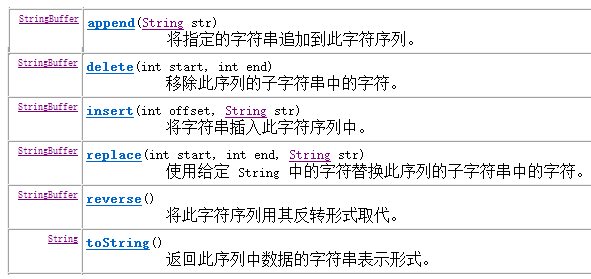
System.out.println( str.toString() );

System.out.pintln( str );

直接打印引用类型变量时，默认调用该类型进行重写后的toString方法

**第3章字符串缓冲区StringBuffer类**

**3.1 StringBuffer的方法使用**



* 代码演示：

创建一个字符串缓冲区对象。用于存储数据。

StringBuffer sb = new StringBuffer();

sb.append("haha"); //添加字符串

sb.insert(2, "it");//在指定位置插入

sb.delete(1, 4);//删除

sb.replace(1, 4, "cast");//替换指定范围内的内容

String str = sb.toString();

* 注意：append、delete、insert、replace、reverse方法调用后，返回值都是当前对象自己，所以说，StringBuffer它可以改变字符序列的长度和内容。

**3.2 StringBuilder类**

查阅API发现还有一个StringBuilder类，它也是字符串缓冲区，StringBuilder与它和StringBuffer的有什么不同呢？

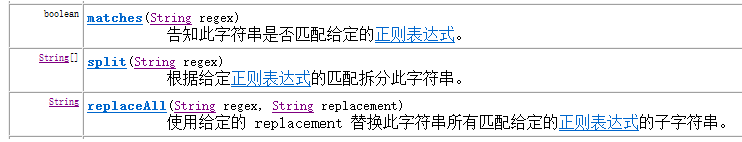
我们阅读StringBuilder的API说明发现，它也是一个可变的字符序列。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。

目前，我们还没有涉及到线程与同步，知道结论StringBuilder比StringBuffer快即可。为什么快，我们会在学习线程时讲解。

**第4章正则表达式**

**4.1正则表达式规则见“正则表达式规则表”**

**4.2字符串类中涉及正则表达式的常用方法**



* public boolean **matches**(String regex) //判断字符串是否匹配给定的规则

举例：校验qq号码.

1: 要求必须是5-15位数字

2: 0不能开头

代码演示：

String qq = "604154942";

String regex = "[1-9][0-9]{4,14}";

**boolean** flag2 = qq.matches(regex);

举例：校验手机号码

1：要求为11位数字

2：第1位为1，第2位为3、4、5、7、8中的一个，后面9位为0到9之间的任意数字。

代码演示：

String phone = "18800022116";

String regex = "1[34578][0-9]{9}";

**boolean** flag = phone.matches(regex);

* public String[] **split**(String regex) //根据给定正则表达式的匹配规则，拆分此字符串

举例：分割出字符串中的的数字

代码演示：

String s = "18-22-40-65";

String regex = "-";

String[] result = s.split(regex);

代码演示：

String s = "18 22 40 65";

String regex = " ";

String[] result = s.split(regex);

* public String **replaceAll**(String regex,String replacement) //将符合规则的字符串内容，全部替换为新字符串

举例：把文字中的数字替换成\*

代码演示：

String s = "Hello12345World6789012";

String regex = "[0-9]";

String result = s.replaceAll(regex, "\*");

**4.3正则表达式练习**

* 匹配正确的数字

匹配规则：

匹配正整数：”\\d+”

匹配正小数：”\\d+\\.\\d+”

匹配负整数：”-\\d+”

匹配负小数：”-\\d+\\.\\d+”

匹配保留两位小数的正数：”\\d+\\.\\d{2}”

匹配保留1-3位小数的正数：”\\d+\\.\\d{1,3}”

* 匹配合法的邮箱

匹配规则：

”[a-zA-Z\_0-9]+@[a-zA-Z\_0-9]+(\\.[a-zA-Z\_0-9]+)+”

”\\w+@\\w+(\\.\\w+)+”

* 获取IP地址(192.168.1.100)中的每段数字

匹配规则：

”\\.”

**第5章Date类**

**5.1Date类的概述**

类Date表示特定的瞬间，精确到毫秒。

**5.2Date类常用方法**

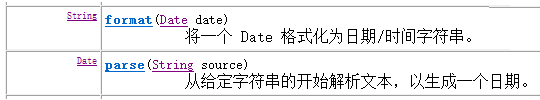
getTime():用于返回自1970年1月1日 00:00:00 GMT以来的毫秒值。

**第6章DateFormat类**

**6.1DateFormat类概述**

DateFormat 是日期/时间格式化子类的抽象类，它以与语言无关的方式格式化并解析日期或时间。

**6.2 DateFormat类常用方法**



* format方法，用来将Date对象转换成String
* parse方法，用来将String转换成Date（转换时，该String要符合指定格式，否则不能转换）。

代码演示：

练习一：把Date对象转换成String

Date date = new Date(1607616000000L);//Fri Dec 11 00:00:00 CST 2020

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

String str = df.format(date);

//str中的内容为2020年12月11日

练习二：把String转换成Date对象

String str = ”2020年12月11日”;

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

Date date = df.parse( str );

//Date对象中的内容为Fri Dec 11 00:00:00 CST 2020

**第7章Calendar**

**7.1 Calendar类概念**

Calendar是日历类，在Date后出现，替换掉了许多Date的方法。该类将所有可能用到的时间信息封装为静态成员变量，方便获取。

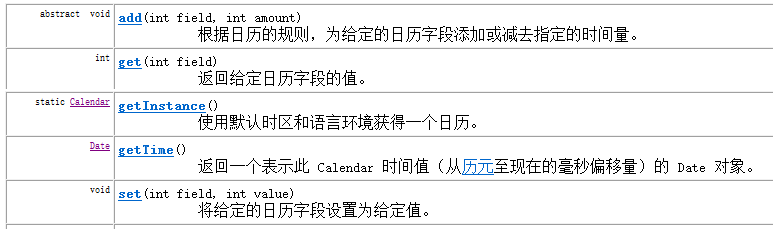
Calendar为抽象类，由于语言敏感性，Calendar类在创建对象时并非直接创建，而是通过静态方法创建，将语言敏感内容处理好，再返回子类对象，如下：

* Calendar类静态方法



Calendar c = Calendar.getInstance(); //返回当前时间

**7.2 Calendar类常用方法**



* public static Calendar **getInstance**() //获取日期对象
* public int **get**(int field) //获取时间字段值，字段参见帮助文档
  + YEAR 年
  + MONTH 月，从0开始算起，最大11；0代表1月，11代表12月。
  + DATE 天
  + HOUR 时
  + MINUTE分
  + SECOND秒

代码演示：

Calendar c = Calendar.getInstance();

int year = c.get(Calendar.YEAR);

* public void **add**(int field,int amount) //指定字段增加某值

代码演示：

Calendar c = Calendar.getInstance();

//修改当前时间为3天后

c.add(Calendar.DATE, 3);

//修改当前时间为5小时后

c.add(Calendar.HOUR, 5);

* public final void **set**(int field,int value)//设置指定字段的值

代码演示：

Calendar c = Calendar.getInstance();

//设置时间为2020年5月20日

c.set(Calendar.YEAR, 2020);

c.set(Calendar.MONTH, 4);

c.set(Calendar.DATE, 20);

* public final Date **getTime**() //获取该日历对象转成的日期对象

代码演示：

Calendar c = Calendar.getInstance();

Date d = c.getTime();

**7.3注意事项**

西方星期的开始为周日，中国为周一。

在Calendar类中，月份的表示是以0-11代表1-12月。

日期是有大小关系的，时间靠后，时间越大。

**第8章基本类型包装类**

**8.1基本数据类型对象包装类特点：用于在基本数据和字符串之间进行转换。**

* 将字符串转成基本类型：



parseXXX(String s);其中XXX表示基本类型，参数为可以转成基本类型的字符串，如果字符串无法转成基本类型，将会发生数字转换的问题 [NumberFormatException](mk://../../java/lang/NumberFormatException.html)

System.out.println(Integer.parseInt("123") + 2);

//打印结果为 125

* 将基本数值转成字符串有3种方式：
  + 基本类型直接与””相连接即可；34+""
  + 调用String的valueOf方法；**String.valueOf(34)** ；



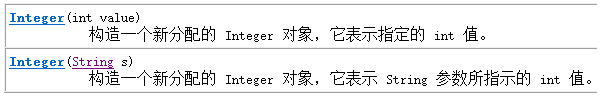
* + 调用包装类中的toString方法；**Integer.toString(34)** ；



**8.2基本类型和对象转换**

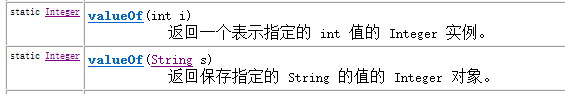
使用int类型与Integer对象转换进行演示，其他基本类型转换方式相同。

* 基本数值---->包装对象



Integer i = **new** Integer(4);//使用构造函数函数

Integer ii = **new** Integer("4");//构造函数中可以传递一个数字字符串



Integer iii = Integer.*valueOf*(4);//使用包装类中的valueOf方法

Integer iiii = Integer.*valueOf*("4");//使用包装类中的valueOf方法

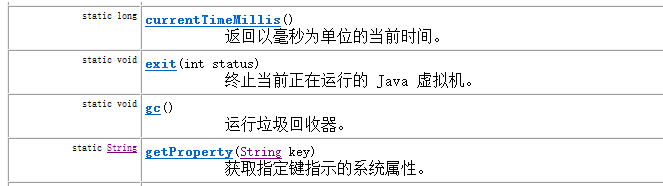
* 包装对象---->基本数值



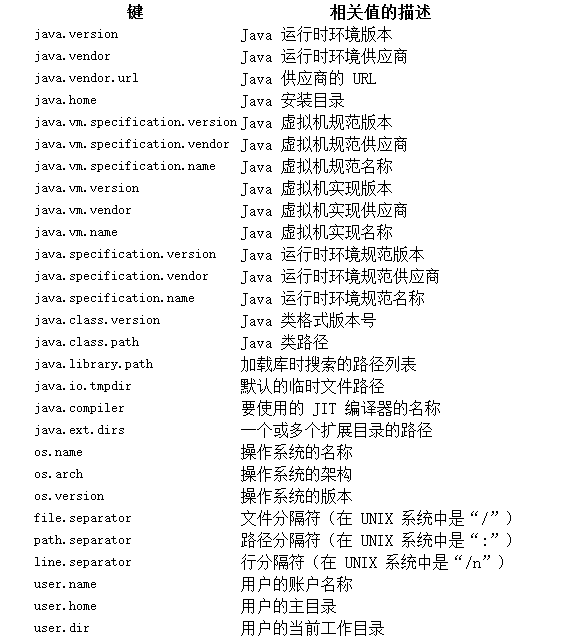
int num = i.intValue();

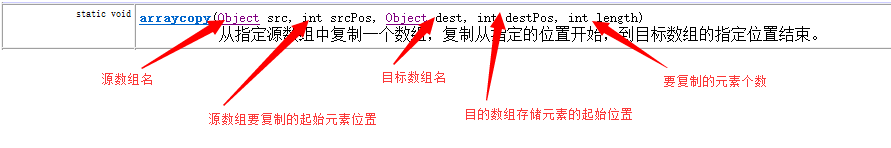
**第9章System类**

**9.1常用方法**



* **currentTimeMillis**() 获取当前系统时间与1970年01月01日00:00点之间的毫秒差值
* **exit(int status)** 用来结束正在运行的Java程序。参数传入一个数字即可。通常传入0记为正常状态，其他为异常状态
* **gc()** 用来运行JVM中的垃圾回收器，完成内存中垃圾的清除。
* **getProperty(String key)** 用来获取指定**键**(字符串名称)中所记录的系统属性信息





* **arraycopy**方法，用来实现将源数组部分元素复制到目标数组的指定位置

**第10章Math类**

**10.1常用方法**

* abs方法,结果都为正数

double d1 = Math.abs(-5); // d1的值为5

double d2 = Math.abs(5); // d2的值为5

* ceil方法，结果为比参数值大的最小整数的double值

double d1 = Math.ceil(3.3); //d1的值为 4.0

double d2 = Math.ceil(-3.3); //d2的值为 -3.0

double d3 = Math.ceil(5.1); // d3的值为 6.0

* floor方法，结果为比参数值小的最大整数的double值

double d1 = Math.floor(3.3); //d1的值为3.0

double d2 = Math.floor(-3.3); //d2的值为-4.0

double d3 = Math.floor(5.1); //d3的值为 5.0

* max方法，返回两个参数值中较大的值

double d1 = Math.max(3.3, 5.5); //d1的值为5.5

double d2 = Math.max(-3.3, -5.5); //d2的值为-3.3

* min方法，返回两个参数值中较小的值

double d1 = Math.min(3.3, 5.5); //d1的值为3.3

double d2 = Math.max(-3.3, -5.5); //d2的值为-5.5

* pow方法，返回第一个参数的第二个参数次幂的值

double d1 = Math.pow(2.0, 3.0); //d1的值为 8.0

double d2 = Math.pow(3.0, 3.0); //d2的值为27.0

* round方法，返回参数值四舍五入的结果

double d1 = Math.round(5.5); //d1的值为6.0

double d2 = Math.round(5.4); //d2的值为5.0

* random方法，产生一个大于等于0.0且小于1.0的double小数

double d1 = Math.random();

**第11章Arrays类**

**11.1常用方法**

* sort方法，用来对指定数组中的元素进行排序（元素值从小到大进行排序）

//源arr数组元素{1,5,9,3,7}, 进行排序后arr数组元素为{1,3,5,7,9}

int[] arr = {1,5,9,3,7};

Arrays.sort( arr );

* toString方法，用来返回指定数组元素内容的字符串形式

int[] arr = {1,5,9,3,7};

String str = Arrays.toString(arr); // str的值为[1, 3, 5, 7, 9]

* binarySearch方法，在指定数组中，查找给定元素值出现的位置。若没有查询到，返回位置为-1。要求该数组必须是个有序的数组。

int[] arr = {1,3,4,5,6};

int index = Arrays.binarySearch(arr, 4); //index的值为2

int index2= Arrasy.binarySearch(arr, 2); //index2的值为-1

**第12章大数据运算**

**12.1 BigInteger**

java中long型为最大整数类型,对于超过long型的数据如何去表示呢.在Java的世界中,超过long型的整数已经不能被称为整数了,它们被封装成BigInteger对象.在BigInteger类中,实现四则运算都是方法来实现,并不是采用运算符.

四则运算代码：

/

public static void main(String[] args) {

//大数据封装为BigInteger对象

BigInteger big1 = new BigInteger("12345678909876543210");

BigInteger big2 = new BigInteger("98765432101234567890");

//add实现加法运算

BigInteger bigAdd = big1.add(big2);

//subtract实现减法运算

BigInteger bigSub = big1.subtract(big2);

//multiply实现乘法运算

BigInteger bigMul = big1.multiply(big2);

//divide实现除法运算

BigInteger bigDiv = big2.divide(big1);

}

**12.2 BigDecimal**

在程序中执行下列代码,会出现什么问题?

System.*out.println(0.09 + 0.01);*

System.*out.println(1.0 - 0.32);*

System.*out.println(1.015 \* 100);*

System.*out.println(1.301 / 100);*

double和float类型在运算中很容易丢失精度,造成数据的不准确性,Java提供我们BigDecimal类可以实现浮点数据的高精度运算

实现加法减法乘法代码如下:

public static void main(String[] args) {

//大数据封装为BigDecimal对象

BigDecimal big1 = new BigDecimal("0.09");

BigDecimal big2 = new BigDecimal("0.01");

//add实现加法运算

BigDecimal bigAdd = big1.add(big2);

BigDecimal big3 = new BigDecimal("1.0");

BigDecimal big4 = new BigDecimal("0.32");

//subtract实现减法运算

BigDecimal bigSub = big3.subtract(big4);

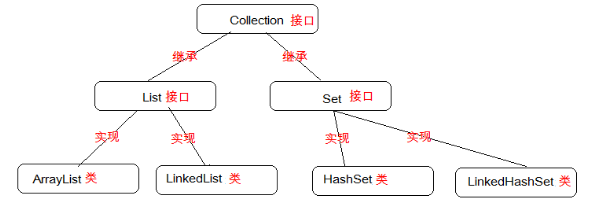
BigDecimal big5 = new BigDecimal("1.105");

BigDecimal big6 = new BigDecimal("100");

//multiply实现乘法运算

BigDecimal bigMul = big5.multiply(big6);

对于浮点数据的除法运算,和整数不同,可能出现无限不循环小数,因此需要对所需要的位数进行保留和选择舍入模式



**第13章List接口**

**13.1 List接口介绍**

用户可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。与 set 不同，列表通常允许重复的元素。

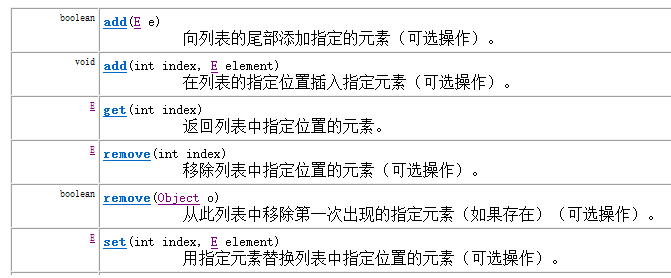
List接口：

* 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）。
* 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）。
* 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接口的常用子类有：

* ArrayList集合
* LinkedList集合

**13.2 List接口中常用的方法**



* 增加元素方法
  + add(Object e)：向集合末尾处，添加指定的元素
  + add(int index, Object e)：向集合指定索引处，添加指定的元素，原有元素依次后移
* 删除元素方法
  + remove(Object e)：将指定元素对象，从集合中删除，返回值为被删除的元素
  + remove(int index)：将指定索引处的元素，从集合中删除，返回值为被删除的元素
* 替换元素方法
  + set(int index, Object e)：将指定索引处的元素，替换成指定的元素，返回值为替换前的元素
* 查询元素方法
  + get(int index)：获取指定索引处的元素，并返回该元素

**13.2.1 Iterator的并发修改异常**

在list集合迭代元素中，对元素进行判断，一旦条件满足就添加一个新元素。代码如下

**public** **class** IteratorDemo {

//在list集合迭代元素中，对元素进行判断，一旦条件满足就添加一个新元素

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

//创建List集合

List<String> list = **new** ArrayList<String>();

//给集合中添加元素

list.add("abc1");

list.add("abc2");

list.add("abc3");

list.add("abc4");

//迭代集合，当有元素为"abc2"时，集合加入新元素"itcast"

Iterator<String> it = list.iterator();

**while**(it.hasNext()){

String str = it.next();

//判断取出的元素是否是"abc2"，是就添加一个新元素

**if**("abc2".equals(str)){

list.add("itcast");// 该操作会导致程序出错

}

}

//打印容器中的元素

System.*out*.println(list);

}

}

运行上述代码发生了错误 java.util.ConcurrentModificationException这是什么原因呢？

在迭代过程中，使用了集合的方法对元素进行操作。导致迭代器并不知道集合中的变化，容易引发数据的不确定性。

并发修改异常解决办法：在迭代时，不要使用集合的方法操作元素。

那么想要在迭代时对元素操作咋办？通过ListIterator迭代器操作元素是可以的，ListIterator的出现，解决了使用Iterator迭代过程中可能会发生的错误情况。

**13.3List集合存储数据的结构**

* 堆栈：弹夹
* 队列：安检
* 数组：查找快，增删慢
* 链表：查找慢，增删快

**13.4ArrayList集合**

ArrayList集合数据存储的结构是数组结构。元素增删慢，查找快，由于日常开发中使用最多的功能为查询数据、遍历数据，所以ArrayList是最常用的集合。

许多程序员开发时非常随意地使用ArrayList完成任何需求，并不严谨，这种用法是不提倡的。

**13.5LinkedList集合**

LinkedList集合数据存储的结构是链表结构。方便元素添加、删除的集合。实际开发中对一个集合元素的添加与删除经常涉及到首尾操作，而LinkedList提供了大量首尾操作的方法。如下图



LinkedList是List的子类，List中的方法LinkedList都是可以使用，这里就不做详细介绍，我们只需要了解LinkedList的特有方法即可。在开发时，LinkedList集合也可以作为堆栈，队列的结构使用。

**第14章Set接口**

学习Collection接口时，记得Collection中可以存放重复元素，也可以不存放重复元素，那么我们知道List中是可以存放重复元素的。那么不重复元素给哪里存放呢？那就是Set接口，它里面的集合，所存储的元素就是不重复的。

**14.1 HashSet集合介绍**

查阅HashSet集合的API介绍：此类实现Set接口，由哈希表支持（实际上是一个 HashMap集合）。HashSet集合不能保证的迭代顺序与元素存储顺序相同。

HashSet集合，采用哈希表结构存储数据，保证元素唯一性的方式依赖于：hashCode()与equals()方法。

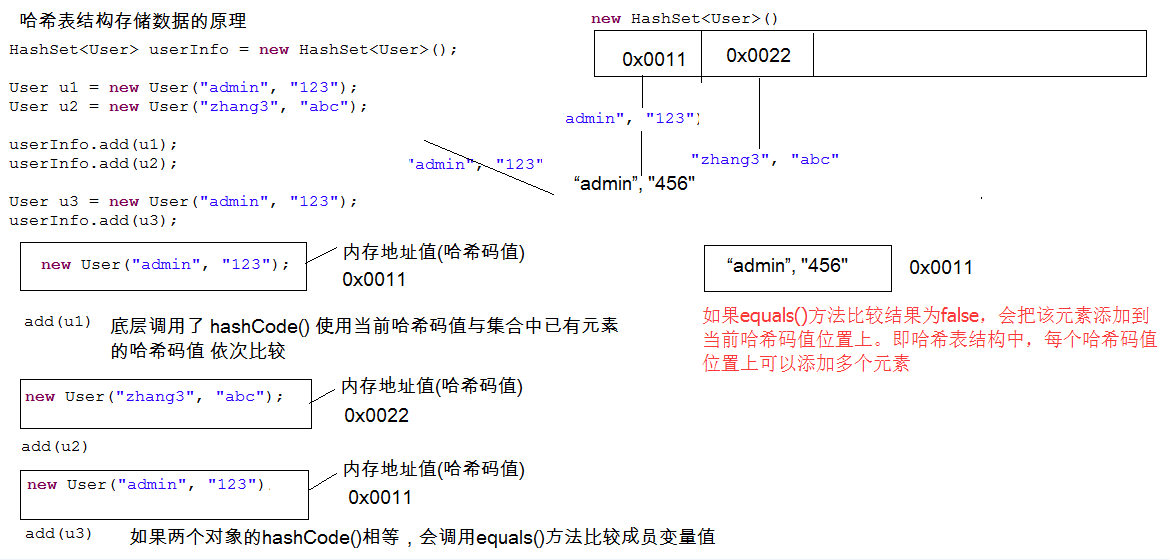
**14.2 HashSet集合存储数据的结构（哈希表）**

什么是哈希表呢？

哈希表底层使用的也是数组机制，数组中也存放对象，而这些对象往数组中存放时的位置比较特殊，当需要把这些对象给数组中存放时，那么会根据这些对象的特有数据结合相应的算法，计算出这个对象在数组中的位置，然后把这个对象存放在数组中。而这样的数组就称为哈希数组，即就是哈希表。

当向哈希表中存放元素时，需要根据元素的特有数据结合相应的算法，这个算法其实就是Object类中的hashCode方法。由于任何对象都是Object类的子类，所以任何对象有拥有这个方法。即就是在给哈希表中存放对象时，会调用对象的hashCode方法，算出对象在表中的存放位置，这里需要注意，如果两个对象hashCode方法算出结果一样，这样现象称为哈希冲突，这时会调用对象的equals方法，比较这两个对象是不是同一个对象，如果equals方法返回的是true，那么就不会把第二个对象存放在哈希表中，如果返回的是false，就会把这个值存放在哈希表中。

总结：保证HashSet集合元素的唯一，其实就是根据对象的hashCode和equals方法来决定的。如果我们往集合中存放自定义的对象，那么保证其唯一，就必须复写hashCode和equals方法建立属于当前对象的比较方式。



**14.3 HashSet存储JavaAPI中的类型元素**

给HashSet中存储JavaAPI中提供的类型元素时，不需要重写元素的hashCode和equals方法，因为这两个方法，在JavaAPI的每个类中已经重写完毕，如String类、Integer类等。

* 创建HashSet集合，存储String对象。

**public** **class** HashSetDemo {

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

//创建HashSet对象

HashSet<String> hs = **new** HashSet<String>();

//给集合中添加自定义对象

hs.add("zhangsan");

hs.add("lisi");

hs.add("wangwu");

hs.add("zhangsan");

//取出集合中的每个元素

Iterator<String> it = hs.iterator();

**while**(it.hasNext()){

String s = it.next();

System.*out*.println(s);

}

}

}

输出结果如下，说明集合中不能存储重复元素：

wangwu

lisi

zhangsan

**14.4 HashSet存储自定义类型元素**

给HashSet中存放自定义类型元素时，需要重写对象中的hashCode和equals方法，建立自己的比较方式，才能保证HashSet集合中的对象唯一

* 创建自定义对象Student

**public** **class** Student {

**private** String name;

**private** **int** age;

**public** Student(String name, **int** age) {

**super**();

**this**.name = name;

**this**.age = age;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

@Override

**public** String toString() {

**return** "Student [name=" + name + ", age=" + age + "]";

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + age;

result = prime \* result + ((name == **null**) ? 0 : name.hashCode());

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if**(!(obj **instanceof** Student)){

**System.out.println(**"类型错误"**)**;

**return** **false**;

}

Student other = (Student) obj;

**return** **this**.age == other.age && **this**.name.equals(other.name);

}

}

* 创建HashSet集合，存储Student对象。

**public** **class** HashSetDemo {

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

//创建HashSet对象

HashSet hs = **new** HashSet();

//给集合中添加自定义对象

hs.add(**new** Student("zhangsan",21));

hs.add(**new** Student("lisi",22));

hs.add(**new** Student("wangwu",23));

hs.add(**new** Student("zhangsan",21));

//取出集合中的每个元素

Iterator it = hs.iterator();

**while**(it.hasNext()){

Student s = (Student)it.next();

System.*out*.println(s);

}

}

}

输出结果如下，说明集合中不能存储重复元素：

Student [name=lisi, age=22]

Student [name=zhangsan, age=21]

Student [name=wangwu, age=23]

**14.5 LinkedHashSet介绍**

我们知道HashSet保证元素唯一，可是元素存放进去是没有顺序的，那么我们要保证有序，怎么办呢？

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

演示代码如下：

**public** **class** LinkedHashSetDemo {

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

Set<String> set = **new** LinkedHashSet<String>();

set.add("bbb");

set.add("aaa");

set.add("abc");

set.add("bbc");

Iterator it = set.iterator();

**while** (it.hasNext()) {

System.*out*.println(it.next());

}

}

}

输出结果如下，LinkedHashSet集合保证元素的存入和取出的顺序：

bbb

aaa

abc

bbc

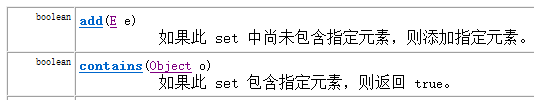
**第15章判断集合元素唯一的原理**

**15.1 ArrayList的contains方法判断元素是否重复原理**



ArrayList的contains方法会使用调用方法时，传入的元素的equals方法依次与集合中的旧元素所比较，从而根据返回的布尔值判断是否有重复元素。此时，当ArrayList存放自定义类型时，由于自定义类型在未重写equals方法前，判断是否重复的依据是地址值，所以如果想根据内容判断是否为重复元素，需要重写元素的equals方法。

**15.2 HashSet的add/contains等方法判断元素是否重复原理**



Set集合不能存放重复元素，其添加方法在添加时会判断是否有重复元素，有重复不添加，没重复则添加。

HashSet集合由于是无序的，其判断唯一的依据是元素类型的hashCode与equals方法的返回结果。规则如下：

先判断新元素与集合内已经有的旧元素的HashCode值

* 如果不同，说明是不同元素，添加到集合。
* 如果相同，再判断equals比较结果。返回true则相同元素；返回false则不同元素，添加到集合。

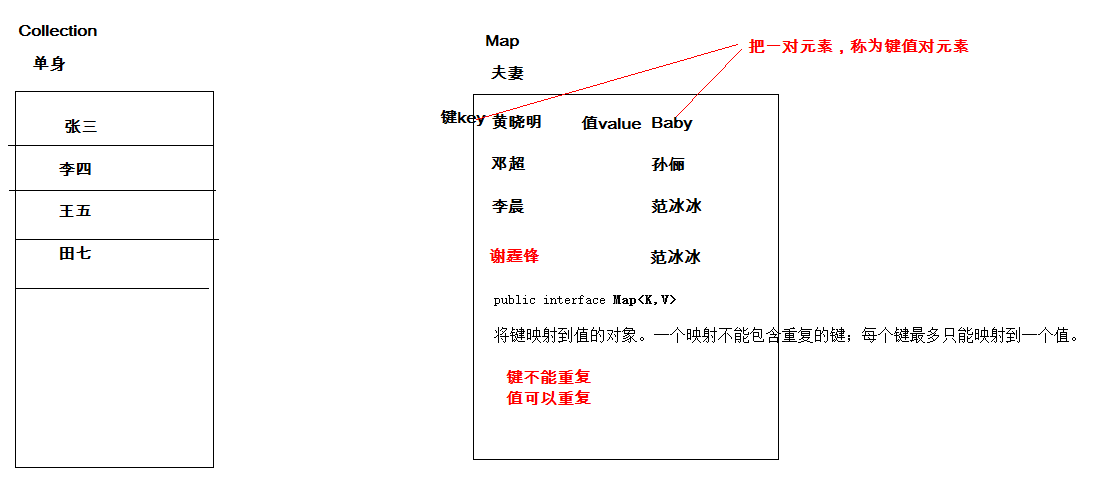
所以，使用HashSet存储自定义类型，如果没有重写该类的hashCode与equals方法，则判断重复时，使用的是地址值，如果想通过内容比较元素是否相同，需要重写该元素类的hashcode与equals方法。

**第16章Map接口**

**16.1 Map接口概述**

我们通过查看Map接口描述，发现Map接口下的集合与Collection接口下的集合，它们存储数据的形式不同，如下图。

* 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。
* Map中常用的集合为HashMap集合、LinkedHashMap集合。

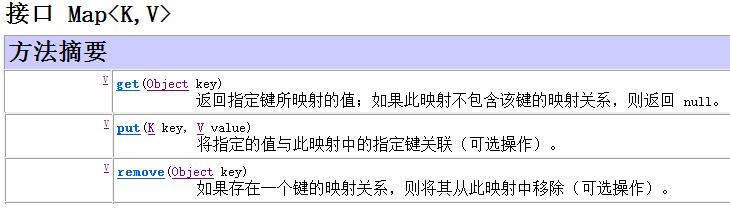


**16.2 Map接口中常用集合概述**

通过查看Map接口描述，看到Map有多个子类，这里我们主要讲解常用的HashMap集合、LinkedHashMap集合。

* **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。
* **注意**：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。

**16.3 Map接口中的常用方法**



* put方法：将指定的键与值对应起来，并添加到集合中
  + 方法返回值为键所对应的值

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；

使用put方法时，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。

* get方法：获取指定键(key)所对应的值(value)
* remove方法：根据指定的键(key)删除元素，返回被删除元素的值(value)。

Map接口的方法演示

**public** **class** MapDemo {

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

//创建Map对象

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

//给map中添加元素

map.put("星期一", "Monday");

map.put("星期日", "Sunday");

System.*out*.println(map); // {星期日=Sunday, 星期一=Monday}

//当给Map中添加元素，会返回key对应的原来的value值，若key没有对应的值，返回null

System.*out*.println(map.put("星期一", "Mon")); // Monday

System.*out*.println(map); // {星期日=Sunday, 星期一=Mon}

//根据指定的key获取对应的value

String en = map.get("星期日");

System.*out*.println(en); // Sunday

//根据key删除元素,会返回key对应的value值

String value = map.remove("星期日");

System.*out*.println(value); // Sunday

System.*out*.println(map); // {星期一=Mon}

}

}

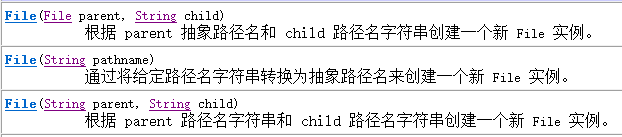
**第17章IO**

**17.1 File类**

**17.1.1字段摘要**

1. pathSeparator: 与系统有关的路径分隔符,window下是“；”。linus下是“：”。
2. separator：与系统有关的默认名称分隔符,window下是“\”。linus下是“/”。

**17.1.2构造方法**



* 通过构造方法创建File对象，我们进行演示：

**public** **class** FileDemo {

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

//File构造函数演示

String pathName = "e:\\java\_code\\day22e\\hello.java";

File f1 = **new** File(pathName);//将Test22文件封装成File对象。注意；可以封装不存在的文件或者文件夹，变成对象。

System.*out*.println(f1);

File f2 = **new** File("e:\\java\_code\\day22e","hello.java");

System.*out*.println(f2);

//将parent封装成file对象。

File dir = **new** File("e:\\java\_code\\day22e");

File f3 = **new** File(dir,"hello.java");

System.*out*.println(f3);

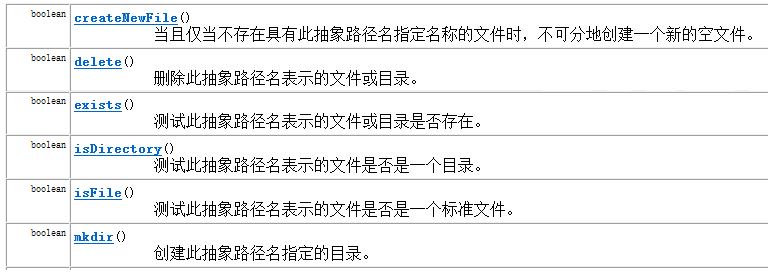
}

}

**17.1.3文件和文件夹的创建删除等**

经常上面介绍，我们知道可以通过File获取到文件名称，文件路径(目录)等信息。

接下来演示使用File类创建、删除文件等操作。



* 我们进行方法的演示

**public** **class** FileMethodDemo2 {

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

// 对文件或者文件加进行操作。

File file = **new** File("e:\\file.txt");

// 创建文件，如果文件不存在，创建 true 如果文件存在，则不创建 false。 如果路径错误，IOException。

**boolean** b1 = file.createNewFile();

System.*out*.println("b1=" + b1);

//-----------删除文件操作-------注意：不去回收站。慎用------

**boolean** b2 = file.delete();

System.*out*.println("b2="+b2);

//-----------需要判断文件是否存在------------

**boolean** b3 = file.exists();

System.*out*.println("b3="+b3);

//-----------对目录操作 创建，删除，判断------------

File dir = **new** File("e:\\abc");

//mkdir()创建单个目录。//dir.mkdirs();创建多级目录

**boolean** b4 = dir.mkdirs();

System.*out*.println("b4="+b4);

//删除目录时，如果目录中有内容，无法直接删除。

**boolean** b5 = dir.delete();

//只有将目录中的内容都删除后，保证该目录为空。这时这个目录才可以删除。

System.*out*.println("b5=" + b5);

//-----------判断文件，目录------------

File f = **new** File("e:\\javahaha");// 要判断是否是文件还是目录，必须先判断存在。

// f.mkdirs();//f.createNewFile();

System.*out*.println(f.isFile());

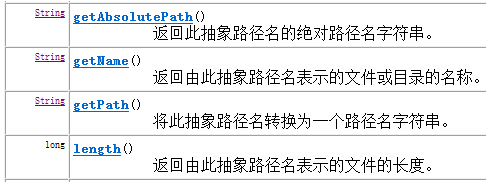
System.*out*.println(f.isDirectory());

}

}

**17.1.4 File类的获取**

创建完了File对象之后，那么File类中都有如下常用方法，可以获取文件相关信息



* 方法演示如下：

**public** **class** FileMethodDemo {

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

//创建文件对象

File file = **new** File("Test22.java");

//获取文件的绝对路径，即全路径

String absPath = file.getAbsolutePath();

//File中封装的路径是什么获取到的就是什么。

String path = file.getPath();

//获取文件名称

String filename = file.getName();

//获取文件大小

**long** size = file.length();

System.*out*.println("absPath="+absPath);

System.*out*.println("path="+path);

System.*out*.println("filename="+filename);

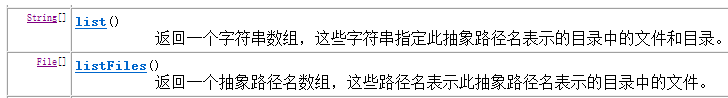
System.*out*.println("size="+size);

}

}

**17.1.5 listFiles()方法介绍**

文件都存放在目录（文件夹）中，那么如何获取一个目录中的所有文件或者目录中的文件夹呢？那么我们先想想，一个目录中可能有多个文件或者文件夹，那么如果File中有功能获取到一个目录中的所有文件和文件夹，那么功能得到的结果要么是数组，要么是集合。我们开始查阅API。



* 方法演示如下：

**public** **class** FileMethodDemo3 {

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

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

//获取的是目录下的当前的文件以及文件夹的名称。

String[] names = dir.list();

**for**(String name : names){

System.*out*.println(name);

}

//获取目录下当前文件以及文件对象，只要拿到了文件对象，那么就可以获取其中想要的信息

File[] files = dir.listFiles();

**for**(File file : files){

System.*out*.println(file);

}

}

}

注意：在获取指定目录下的文件或者文件夹时必须满足下面两个条件

1，指定的目录必须是存在的，

2，指定的必须是目录。否则容易引发返回数组为null，出现NullPointerException

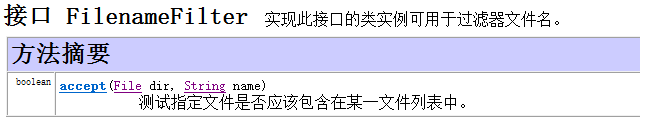
**17.1.6文件过滤器**

通过listFiles()方法，我们可以获取到一个目录下的所有文件和文件夹，但能不能对其进行过滤呢？比如我们只想要一个目录下的指定扩展名的文件，或者包含某些关键字的文件夹呢？

我们是可以先把一个目录下的所有文件和文件夹获取到，并遍历当前获取到所有内容，遍历过程中在进行筛选，但是这个动作有点麻烦，Java给我们提供相应的功能来解决这个问题。

查阅File类的API，在查阅时发现File类中重载的listFiles方法，并且接受指定的过滤器。





* 测试类

**public** **class** FileDemo2 {

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

//获取扩展名为.java所有文件

//创建File对象

File file = **new** File("E:\\code\\day11\_code");

//获取指定扩展名的文件,由于要对所有文件进行扩展名筛选，因此调用方法需要传递过滤器

File[] files = file.listFiles(**new** MyFileFilter());

//遍历获取到的所有符合条件的文件

**for** (File f : files) {

System.*out*.println(f);

}

}

}

* 自定类继承FilenameFilter过滤器接口

//定义类实现文件名称FilenameFilter过滤器

**class** MyFileFilter **implements** FilenameFilter{

**public** **boolean** accept(File dir, String name) {

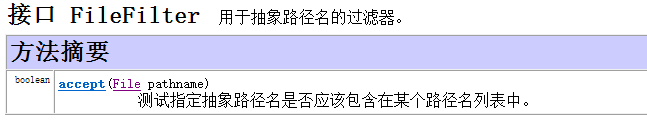
**return** name.endsWith(".java");

}

}

在查阅API时，我们发现，在listFiles(FileFilter filter) 也可以接受一个FileFilter过滤器，它和我们讲的FilenameFilter有啥区别呢？





FilenameFilter过滤器中的accept方法接受两个参数，一个当前文件或文件夹所在的路径，一个是当前文件或文件夹对象的名称。

FileFilter 过滤器中的accept方法接受一个参数，这个参数就当前文件或文件夹对象

当我们需要过滤文件名称时就可以使用FilenameFilter这个过滤器，当我们想对当前文件或文件夹进行过滤，就可以使用FileFilter ，比如需要当前目录下的所有文件夹，就可以使用FileFilter 过滤器。

* 测试类

**public** **class** FileDemo2 {

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

//获取扩展名为.java所有文件

//创建File对象

File file = **new** File("E:\\code\\day11\_code");

//获取指定目录下的文件夹

File[] files = file.listFiles(**new** FileFileterByDir());

//遍历获取到的所有符合条件的文件

**for** (File f : files) {

System.*out*.println(f);

}

}

}

* 自定义类继承FileFilter过滤器接口

//文件过滤器

**class** FileFileterByDir **implements** FileFilter{

**public** **boolean** accept(File pathname) {

**return** pathname.isDirectory();

}

}

**第18章递归**

**18.1递归的概述**

递归，指在当前方法内调用自己的这种现象

public void method(){

System.out.println(“递归的演示”);

//在当前方法内调用自己

method();

}

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。

* 递归的代码演示，计算1-n之间的和，使用递归完成

**public** **class** DiGuiDemo {

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

//计算1~num的和，使用递归完成

**int** n = 5;

**int** sum = *getSum*(n);

System.*out*.println(sum);

}

**public** **static** **int** getSum(**int** n) {

**if**(n == 1){

**return** 1;

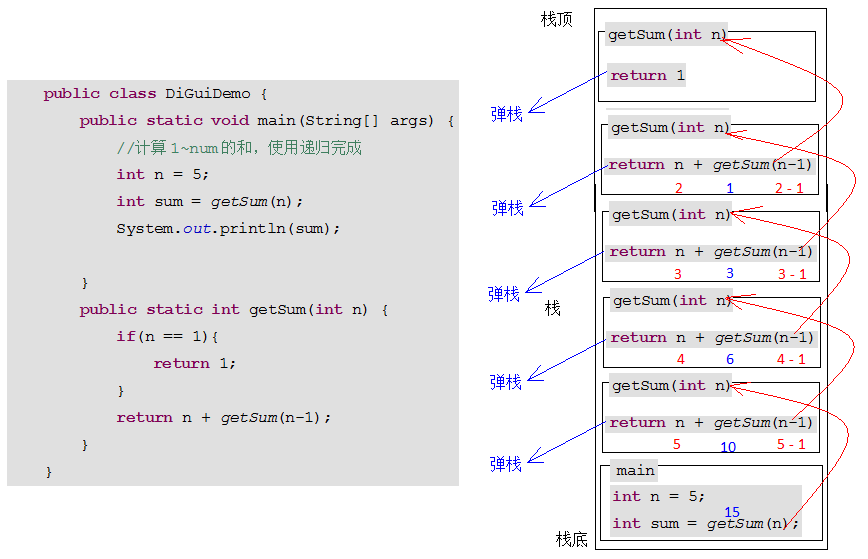
}

**return** n + *getSum*(n-1);

}

}

* 代码执行流程图解



注意：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。

**18.2递归打印所有子目录中的文件路径**

编写一个方法用来打印指定目录中的文件路径，并进行方法的调用

要求：若指定的目录有子目录，那么把子目录中的文件路径也打印出来

步骤：

1. 指定要打印的目录File对象

2. 调用getFileAll()方法

2.1 获取指定目录中的所有File对象

2.2 遍历得到每一个File对象

2.3 判断当前File 对象是否是目录

判断结果为true，说明为目录，通过递归，再次调用步骤2的getFileAll()方法

判断结果为false，说明是文件，打印文件的路径

* 代码演示

**public** **class** FileDemo2 {

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

File file = **new** File("d:\\test");

*getFileAll*(file);

}

//获取指定目录以及子目录中的所有的文件

**public** **static** **void** getFileAll(File file) {

File[] files = file.listFiles();

//遍历当前目录下的所有文件和文件夹

**for** (File f : files) {

//判断当前遍历到的是否为目录

**if**(f.isDirectory()){

//是目录，继续获取这个目录下的所有文件和文件夹

*getFileAll*(f);

}**else**{

//不是目录，说明当前f就是文件，那么就打印出来

System.*out*.println(f);

}

}

}

}

**18.3搜索指定目录中的.java文件(含子目录)**

需求：打印指定目录即所有子目录中的.java文件的文件路径

要求：编写一个方法用来打印指定目录中的.java文件路径，并进行方法的调用

若指定的目录有子目录，那么把子目录中的.java文件路径也打印出来

步骤：

1. 指定要打印的目录File对象

2. 调用getFileAll()方法，传入要打印的目录File对象

2.1 通过FilenameFilter过滤器获取指定目录中的所有.java类型的File对象

2.2 遍历得到每一个File对象

2.3 判断当前File 对象是否是目录

判断结果为true，说明为目录，通过递归，再次调用步骤2的getFileAll()方法

判断结果为false，说明是文件，打印文件的路径

**实现代码步骤**

* 测试类

**public** **class** FileDemo4 {

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

File file = **new** File("d:\\test");

*getFileAll*(file);

}

//获取指定目录以及子目录中的所有的文件

**public** **static** **void** getFileAll(File file) {

File[] files = file.listFiles(MyFileFilter());

//遍历当前目录下的所有文件和文件夹

**for** (File f : files) {

//判断当前遍历到的是否为目录

**if**(f.isDirectory()){

//是目录，继续获取这个目录下的所有文件和文件夹

*getFileAll*(f);

}**else**{

//不是目录，说明当前f就是文件，那么就打印出来

System.*out*.println(f);

}

}

}

}

* 自定类继承FilenameFilter过滤器接口

//定义类实现文件名称FilenameFilter过滤器

**class** MyFileFilter **implements** FilenameFilter{

**public** **boolean** accept(File dir, String name) {

**return** name.endsWith(".java");

}

}

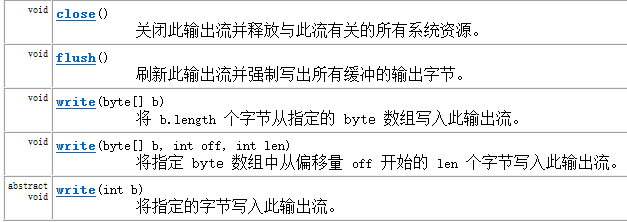
# 字节流

在前面的学习过程中，我们一直都是在操作文件或者文件夹，并没有给文件中写任何数据。现在我们就要开始给文件中写数据，或者读取文件中的数据。

## 字节输出流OutputStream

OutputStream此抽象类，是表示输出字节流的所有类的超类。操作的数据都是字节，定义了输出字节流的基本共性功能方法。

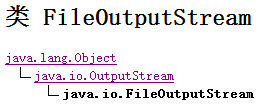
输出流中定义都是写write方法，如下图:



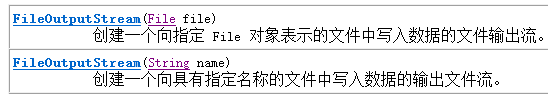
### FileOutputStream类

OutputStream有很多子类，其中子类FileOutputStream可用来写入数据到文件。

FileOutputStream类，即文件输出流，是用于将数据写入 File的输出流。



* 构造方法



### FileOutputStream类写入数据到文件中

* 将数据写到文件中，代码演示：

**public** **class** FileOutputStreamDemo {

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

//需求：将数据写入到文件中。

//创建存储数据的文件。

File file = **new** File("c:\\file.txt");

//创建一个用于操作文件的字节输出流对象。一创建就必须明确数据存储目的地。

//输出流目的是文件，会自动创建。如果文件存在，则覆盖。

FileOutputStream fos = **new** FileOutputStream(file);

//调用父类中的write方法。

**byte**[] data = "abcde".getBytes();

fos.write(data);

//关闭流资源。

fos.close();

}

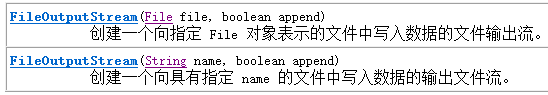
}

### 给文件中续写和换行

我们直接**new** FileOutputStream(file)这样创建对象，写入数据，会覆盖原有的文件，那么我们想在原有的文件中续写内容怎么办呢？

继续查阅FileOutputStream的API。发现在FileOutputStream的构造函数中，可以接受一个boolean类型的值，如果值true，就会在文件末位继续添加。

* 构造方法



* 给文件中续写数据和换行，代码演示：

**public** **class** FileOutputStreamDemo2 {

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

File file = **new** File("c:\\file.txt");

FileOutputStream fos = **new** FileOutputStream(file, **true**);

String str = "\r\n"+"itcast";

fos.write(str.getBytes());

fos.close();

}

}

### IO异常的处理

在前面编写代码中都发生了IO的异常。我们在实际开发中，对异常时如何处理的，我们来演示一下。

**public** **class** FileOutputStreamDemo3 {

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

File file = **new** File("c:\\file.txt");

//定义FileOutputStream的引用

FileOutputStream fos = **null**;

**try** {

//创建FileOutputStream对象

fos = **new** FileOutputStream(file);

//写出数据

fos.write("abcde".getBytes());

} **catch** (IOException e) {

System.*out*.println(e.toString() + "----");

} **finally** {

//一定要判断fos是否为null，只有不为null时，才可以关闭资源

**if** (fos != **null**) {

**try** {

fos.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException("");

}

}

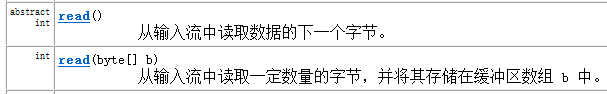
}

}

}

## 字节输入流InputStream

通过前面的学习，我们可以把内存中的数据写出到文件中，那如何想把内存中的数据读到内存中，我们通过InputStream可以实现。InputStream此抽象类，是表示字节输入流的所有类的超类。，定义了字节输入流的基本共性功能方法。



* int read():读取一个字节并返回，没有字节返回-1.
* int read(byte[]): 读取一定量的字节数，并存储到字节数组中，返回读取到的字节数。

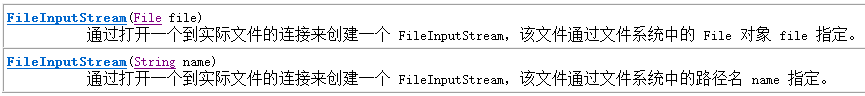
### FileInputStream类

InputStream有很多子类，其中子类FileInputStream可用来读取文件内容。

FileInputStream 从文件系统中的某个文件中获得输入字节。



* 构造方法



### FileInputStream类读取数据read方法

在读取文件中的数据时，调用read方法，实现从文件中读取数据



* 从文件中读取数据，代码演示：

**public** **class** FileInputStreamDemo {

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

File file = **new** File("c:\\file.txt");

//创建一个字节输入流对象,必须明确数据源，其实就是创建字节读取流和数据源相关联。

FileInputStream fis = **new** FileInputStream(file);

//读取数据。使用 read();一次读一个字节。

**int** ch = 0;

**while**((ch=fis.read())!=-1){

System.*out*.pr }intln("ch="+(**char**)ch);

// 关闭资源。

fis.close();

}

}

### 读取数据read(byte[])方法

在读取文件中的数据时，调用read方法，每次只能读取一个，太麻烦了，于是我们可以定义数组作为临时的存储容器，这时可以调用重载的read方法，一次可以读取多个字符。



**public** **class** FileInputStreamDemo2 {

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

/\*

\* 演示第二个读取方法， read(byte[]);

\*/

File file = **new** File("c:\\file.txt");

// 创建一个字节输入流对象,必须明确数据源，其实就是创建字节读取流和数据源相关联。

FileInputStream fis = **new** FileInputStream(file);

//创建一个字节数组。

**byte**[] buf = **new** **byte**[1024];//长度可以定义成1024的整数倍。

**int** len = 0;

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

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

}

fis.close();

}

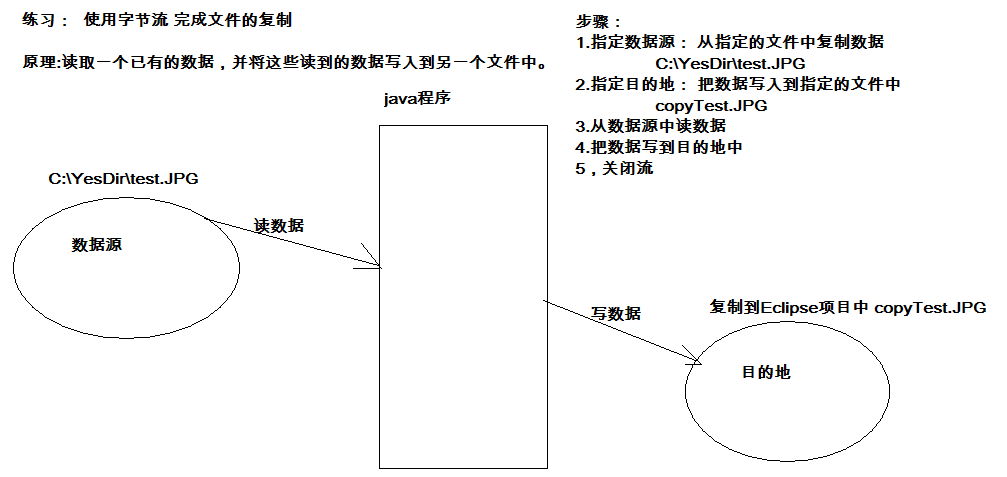
}

## 字节流练习

既然会了文件的读和写操作了，那么我们就要在这个基础上进行更为复杂的操作。使用读写操作完成文件的复制。

### 复制文件

原理；读取一个已有的数据，并将这些读到的数据写入到另一个文件中。



**public** **class** CopyFileTest {

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

//1,明确源和目的。

File srcFile = **new** File("c:\\YesDir\test.JPG");

File destFile = **new** File("copyTest.JPG");

//2,明确字节流 输入流和源相关联，输出流和目的关联。

FileInputStream fis = **new** FileInputStream(srcFile);

FileOutputStream fos = **new** FileOutputStream(destFile);

//3, 使用输入流的读取方法读取字节，并将字节写入到目的中。

**int** ch = 0;

**while**((ch=fis.read())!=-1){

fos.write(ch);

}

//4,关闭资源。

fos.close();

fis.close();

}

}

上述代码输入流和输出流之间是通过ch这个变量进行数据交换的。

上述复制文件有个问题，每次都从源文件读取一个，然后在写到指定文件，接着再读取一个字符，然后再写一个，一直这样下去。效率极低。

### 缓冲数组方式复制文件

上述代码复制文件效率太低了，并且频繁的从文件读数据，和写数据，能不能一次多把文件中多个数据都读进内容中，然后在一次写出去，这样的速度一定会比前面代码速度快。

**public** **class** CopyFileByBufferTest {

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

File srcFile = **new** File("c:\\YesDir\test.JPG");

File destFile = **new** File("copyTest.JPG");

// 明确字节流 输入流和源相关联，输出流和目的关联。

FileInputStream fis = **new** FileInputStream(srcFile);

FileOutputStream fos = **new** FileOutputStream(destFile);

//定义一个缓冲区。

**byte**[] buf = **new** **byte**[1024];

**int** len = 0;

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

fos.write(buf, 0, len);// 将数组中的指定长度的数据写入到输出流中。

}

// 关闭资源。

fos.close();

fis.close();

}

}

# 字符流

经过前面的学习，我们基本掌握的文件的读写操作，在操作过程中字节流可以操作所有数据，可是当我们操作的文件中有中文字符，并且需要对中文字符做出处理时怎么办呢？

## 字节流读取字符的问题

通过以下程序读取带有中文件的文件。

**public** **class** CharStreamDemo {

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

//给文件中写中文

*writeCNText*();

//读取文件中的中文

*readCNText*();

}

//读取中文

**public** **static** **void** readCNText() **throws** IOException {

FileInputStream fis = **new** FileInputStream("c:\\cn.txt");

**int** ch = 0;

**while**((ch = fis.read())!=-1){

System.*out*.println(ch);

}

}

//写中文

**public** **static** **void** writeCNText() **throws** IOException {

FileOutputStream fos = **new** FileOutputStream("c:\\cn.txt");

fos.write("a传智播客欢迎你".getBytes());

fos.close();

}

}

上面程序在读取含有中文的文件时，我们并没有看到具体的中文，而是看到一些数字，这是什么原因呢？既然看不到中文，那么我们如何对其中的中文做处理呢？要解决这个问题，我们必须研究下字符的编码过程。

## 字符编码表

我们知道计算机底层数据存储的都是二进制数据，而我们生活中的各种各样的数据，如何才能和计算机中存储的二进制数据对应起来呢？

这时老美他们就把每一个字符和一个整数对应起来，就形成了一张编码表，老美他们的编码表就是ASCII表。其中就是各种英文字符对应的编码。

编码表：其实就是生活中字符和计算机二进制的对应关系表。

1、ascii： 一个字节中的7位就可以表示。对应的字节都是正数。0-xxxxxxx

2、iso-8859-1:拉丁码表 latin，用了一个字节用的8位。1-xxxxxxx 负数。

3、GB2312:简体中文码表。包含6000-7000中文和符号。用两个字节表示。两个字节第一个字节是负数,第二个字节可能是正数

GBK:目前最常用的中文码表，2万的中文和符号。用两个字节表示，其中的一部分文字，第一个字节开头是1，第二字节开头是0

GB18030：最新的中文码表，目前还没有正式使用。

1. unicode：国际标准码表:无论是什么文字，都用两个字节存储。

* Java中的char类型用的就是这个码表。char c = 'a';占两个字节。
* Java中的字符串是按照系统默认码表来解析的。简体中文版 字符串默认的码表是GBK。

5、UTF-8:基于unicode，一个字节就可以存储数据，不要用两个字节存储，而且这个码表更加的标准化，在每一个字节头加入了编码信息(后期到api中查找)。

能识别中文的码表：GBK、UTF-8；正因为识别中文码表不唯一，涉及到了编码解码问题。

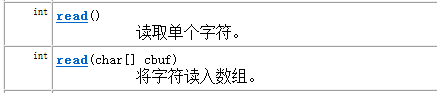
对于我们开发而言；常见的编码 GBK UTF-8 ISO-8859-1

文字--->(数字) ：编码。 “abc”.getBytes() byte[]

(数字)--->文字 : 解码。 byte[] b={97,98,99} new String(b)

## 字符输入流Reader

上述程序中我们读取拥有中文的文件时，使用的字节流在读取，那么我们读取到的都是一个一个字节。只要把这些字节去查阅对应的编码表，就能够得到与之对应的字符。API中是否给我们已经提供了读取相应字符的功能流对象，Reader，读取字符流的抽象超类。



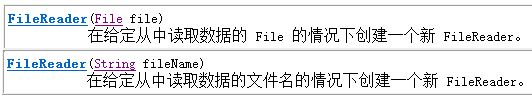
* read():读取单个字符并返回
* read(char[]):将数据读取到数组中，并返回读取的个数。

### FileReader类

查阅FileInputStream的API，发现FileInputStream 用于读取诸如图像数据之类的原始字节流。要读取字符流，请考虑使用 FileReader。

打开FileReader的API介绍。用来读取字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是适当的

* 构造方法



### FileReader读取包含中文的文件

* 使用FileReader读取包含中文的文件

**public** **class** CharStreamDemo {

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

//给文件中写中文

*writeCNText*();

//读取文件中的中文

*readCNText*();

}

//读取中文

**public** **static** **void** readCNText() **throws** IOException {

FileReader fr = **new** FileReader("D:\\test\\cn.txt");

**int** ch = 0;

**while**((ch = fr.read())!=-1){

//输出的字符对应的编码值

System.*out*.println(ch);

//输出字符本身

System.*out*.println((**char**)ch);

}

}

//写中文

**public** **static** **void** writeCNText() **throws** IOException {

FileOutputStream fos = **new** FileOutputStream("D:\\test\\cn.txt");

fos.write("a传智播客欢迎你".getBytes());

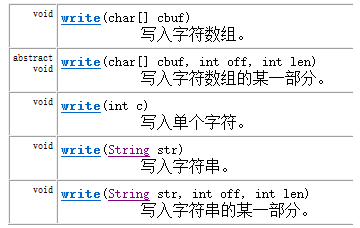
fos.close();

}

}

## 字符输出流Writer

既然有专门用于读取字符的流对象，那么肯定也有写的字符流对象，查阅API，发现有一个Writer类，Writer是写入字符流的抽象类。其中描述了相应的写的动作。



### FileWriter类

查阅FileOutputStream的API，发现FileOutputStream 用于写入诸如图像数据之类的原始字节的流。要写入字符流，请考虑使用 FileWriter。

打开FileWriter的API介绍。用来写入字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是可接受的。

* 构造方法



### FileWriter写入中文到文件中

* 写入字符到文件中，先进行流的刷新，再进行流的关闭。

**public** **class** FileWriterDemo {

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

//演示FileWriter 用于操作文件的便捷类。

FileWriter fw = **new** FileWriter("d：\\text\\fw.txt");

fw.write("你好谢谢再见");//这些文字都要先编码。都写入到了流的缓冲区中。

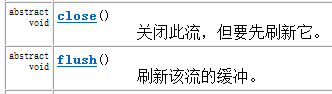
fw.flush();

fw.close();

}

}

## flush()和close()的区别？



flush():将流中的缓冲区缓冲的数据刷新到目的地中，刷新后，流还可以继续使用。

close():关闭资源，但在关闭前会将缓冲区中的数据先刷新到目的地，否则丢失数据，然后在关闭流。流不可以使用。如果写入数据多，一定要一边写一边刷新，最后一次可以不刷新，由close完成刷新并关闭。

## 字符流练习

### 复制文本文件

练习：复制文本文件。

思路：

1，既然是文本涉及编码表。需要用字符流。

2，操作的是文件。涉及硬盘。

3，有指定码表吗？没有，默认就行。

操作的是文件，使用的 默认码表。使用哪个字符流对象。直接使用字符流操作文件的便捷类。FileReader FileWriter

**public** **class** CopyTextFileTest {

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

*copyTextFile*();

}

**public** **static** **void** copyTextFile() **throws** IOException {

//1,明确源和目的。

FileReader fr = **new** FileReader("c:\\cn.txt");

FileWriter fw = **new** FileWriter("c:\\copy.txt");

//2,为了提高效率。自定义缓冲区数组。字符数组。

**char**[] buf = **new** **char**[1024];

**int** len = 0;

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

fw.write(buf,0,len);

}

/\*2,循环读写操作。效率低。

int ch = 0;

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

fw.write(ch);

}

\*/

//3,关闭资源。

fw.close();

fr.close();

}

}