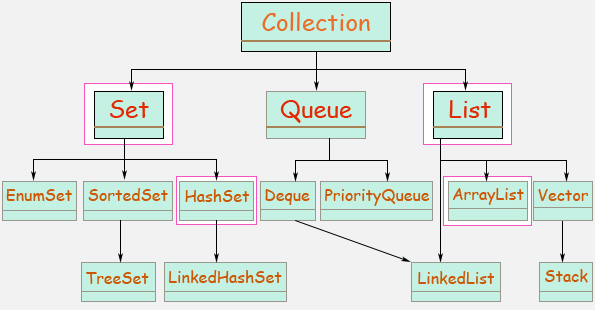
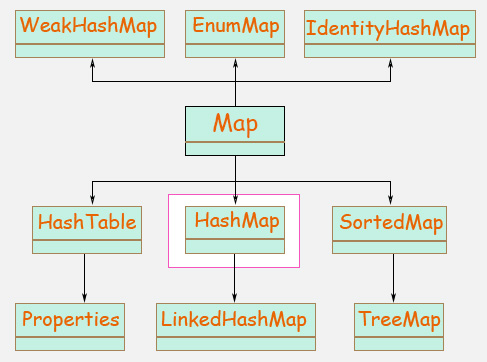
第一章 集合

一、集合概述

Java集合主要由Collection和Map两个接口派生而出，Collection和Map是java集合框架的根接口，java集合的继承树如下：



Map的继承树：



二、Collection和Iterator

1、Collection

Collection接口是Set，List，Queue接口的父接口，该接口定义的方法可用于操作Set，List，Queue集合。

2、Iterator

Iterator也是集合框架的成员，但它与Collection系列和Map系列不同，Iterator主要用于遍历Collection集合的元素，而不是存储元素。

Iterator常用的三个方法：

|  |  |
| --- | --- |
| **boolean** hasNext() | 如果被迭代的元素还没有被遍历，则返回true |
| Object next() | 返回集合下一个元素 |
| **void** remove() | 删除集合上一次next方法返回的元素 |

示例：

**public** **class** TestIterator {

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

Collection<String> books = **new** HashSet<String>();

books.add("j2EE");

books.add("dataStructure");

books.add("xml");

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

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

String book = (String) it.next();

System.*out*.println(book);

**if**(book.equals("xml")){

it.remove();

}

book = "测试字符串";//①

}

System.*out*.println(books);

}

}

上述代码①对迭代变量进行赋值，再次输出books集合时，集合里元素并没有任何改变；结论：当使用Iterator对集合元素进行迭代时，Iterator并不是把集合元素本身传给了迭代变量，而是把集合元素的值传给了迭代变量，所以修改迭代变量的值对集合元素本身没有任何改变。

三、Set接口

1、Set集合概述

Set集合同Collection集合一样，没有提供额外的方法。Set集合不允许包含相同元素，Set集合中元素之间是无序的。

Set集合通过equals判断元素对象是否相等，不使用“==”。

Set集合有三个常用的实现类：HashSet、TreeSet和EnumSet。

2、Set集合常用方法

|  |  |
| --- | --- |
| **集合元素相关的操作：添加、删除、包含、比较、清除、判断为空、集合大小** | |
| 添加 | **boolean** add(Object o), **boolean** addAll(Collection c) |
| 删除 | **boolean** remove(Object o),removeAll(Collection c) |
| 包含 | **boolean** contains(Object o),containsAll(Collection c) |
| 比较 | **boolean** equals(Object o) |
| 清除 | **void** clear(),删除集合中所有元素 |
| 判断为空 | **boolean** isEmpty() |
| 集合大小 | **int** size() |
| **集合类型转化的操作：转化成数组、转化成字符串** | |
| 转成数组 | Ojbect[] toArray(),T[] toArray(T[] s); |
| 转成字符串 | toString(); |
| **集合的访问方式：集合的遍历** | |
| 遍历 | Iterator iterator(); |

疑问：在Set集合的实现类中并没有实现toArray()方法，为何调用toArray()方法还能成功进行转换?

3、HashSet类

HashSet的特点如下：

* 不能保证元素的排列顺序，顺序有可能发生变化，即无序性；
* 不是线程同步的，即非线程安全的，涉及线程操作需用代码保证其同步；
* 集合元素的值允许为null；
* HashSet集合判断俩个元素相等的标准是两个对象通过equals方法比较相等，并且两个对象的hashCode()方法返回值也相等。

4、TreeSet类

TreeSet类是SortedSet的接口的唯一实现类，SortedSet直接实现了Set接口。

TreeSet的特点如下：

* TreeSet中的元素是有序的，TreeSet中的排序并不是按插入顺序排序，而是按元素的实际值排序；
* 向TreeSet添加的所有元素对象都必须实现Comparable接口；
* TreeSet中的元素对象都是同一个类的实例。

TreeSet特点的解释：

TreeSet中的元素排序采用了红黑树的数据结构对元素排序，TreeSet支持两种排序方法：自然排序和定制排序，默认采用自然排序；TreeSet会调用集合元素的CompareTo(Ojbect o)方法来比较元素之间的大小，然后将集合元素按升序排列，这就是自然排序。所以TreeSet中的元素对象都应该实现Comparable接口。大部分类在实现CompareTo(Ojbect obj)方法时，都需要将obj强制类型转换成相同类型，只用相同类型才有可比较性，所以TreeSet中的元素对象都会是同一个类的实例。

示例：

**class** Err{

}

**public** **class** TestTreeSet {

@SuppressWarnings({ "rawtypes", "unchecked" })

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

Set nums = **new** TreeSet();

nums.add(4);

nums.add(2);

nums.add(-5);

nums.add(8);

System.*out*.println(nums);

//引发ClassCastException，Err没有实现Comparable接口

Set ts = **new** TreeSet();

ts.add(**new** Err());

ts.add(**new** Err());

//引发ClassCastException,添加的元素不是同一个对象的实例

Set tss = **new** TreeSet();

tss.add(**new** String("XML"));

tss.add(**new** Date());

}

}

以下为控制台打印信息：

控制台信息：

[-5, 2, 4, 8]

Exception in thread "main" java.lang.ClassCastException: com.cn.Collection.Err cannot be cast to java.lang.Comparable

Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.util.Date

5、EnumSet类

EnumSet是专门为枚举类设计的集合类，EnumSet中的值都必须是指定枚举类型的枚举值，该枚举类型在创建EnumSet时显式或隐式的指定。

EnumSet集合特点如下：

* 集合元素有序，以枚举值在Enum类内的定义顺序来决定集合元素的顺序；
* 不允许加入null元素；
* 非线程安全；
* 在内部以向量形式存储，这种存储方式非常紧凑、高效，因此EnumSet对象占用内存很小，而且运行效率很高。

EnumSet类没有暴露任何构造器来创建该类的实例，只能根据其提供的静态方法来创建EnumSet对象。具体方法如下：

|  |  |
| --- | --- |
| **Static** EnumSet allOf(Class EnumType) |  |
| **Static** EnumSet complementOf(EnumSet s) |  |
| **Static** EnumSet copyOf(Collection s) |  |
| **Static** EnumSet copyOf(EnumSet s) |  |
| **Static** EnumSet noneOf(Class EnumType) |  |

实例：

**enum** Season{

*Spring*,*Summer*,*Fall*,*Winter*;

}

**public** **class** TestEnumSet {

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

//创建一个EnumSet类，将指定枚举类的所有枚举值作为集合的元素

EnumSet<Season> es1 = EnumSet.*allOf*(Season.**class**);

System.*out*.println(es1);

//创建一个EnumSet类，指定集合元素类型，但并不为集合添加任何元素

EnumSet<Season> es2 = EnumSet.*noneOf*(Season.**class**);

System.*out*.println(es2);

es2.add(Season.*Spring*);

es2.add(Season.*Summer*);

System.*out*.println(es2);

/\*创建一个EnumSet类，es3初始包含的元素是es2中没有包含的，es3+es2的元 素是指定枚举类的所有枚举值，是一个互补的关系\*/

EnumSet<Season> es3 = EnumSet.*complementOf*(es2);

System.*out*.println(es3);

//创建一个EnumSet类，将枚举类的枚举值传入作为EnumSet类的集合元素

EnumSet<Season> es4 = EnumSet.*of*(Season.*Spring*);

System.*out*.println(es4);

EnumSet<Season> es5 = EnumSet.*of*(Season.*Spring*,Season.*Winter*);

System.*out*.println(es5);

/\*创建一个EnumSet类，初始集合元素为枚举类的start枚举值到end枚举 值 的所有枚举值\*/

EnumSet<Season> es6 = EnumSet.*range*(Season.*Spring*,Season.*Fall*);

System.*out*.println(es6);

}

}

EnumSet是所有Set实现类中性能最好的，但它只能保存同一个枚举类的枚举值作为集合的元素。

四、List接口

1、List集合概述

List集合代表一个有序集合，集合中每个元素都有其对应的顺序索引。List集合允许使用重复元素，可以通过索引来访问指定位置的集合元素。

实现类有ArrayList,Vector,LinkedList。

2、List集合常用方法

|  |  |
| --- | --- |
| **集合元素相关的操作：添加、删除、包含、比较、清除、判断为空、集合大小** | |
| 添加 | **boolean** add(E e);  **void** add(**int** index, E element);  **boolean** addAll(Collection<? **extends** E> c);  **boolean** addAll(**int** index, Collection<? **extends** E> c) |
| 删除 | **boolean** remove(Object o);  **boolean** removeAll(Collection<?> c); |
| 包含 | **boolean** contains(Object o)  **boolean** containsAll(Collection<?> c);  **boolean** retainAll(Collection<?> c),是否有交集  List<E> subList(**int** fromIndex, **int** toIndex) |
| 比较 | **boolean** equals(Object o) |
| 清除 | **void** clear(),删除集合中所有元素 |
| 判断为空 | **boolean** isEmpty() |
| 集合大小 | **int** size() |
| 修改 | E set(**int** index,E element); |
| **集合类型转化的操作：转化成数组、转化成字符串** | |
| 转成数组 | Ojbect[] toArray(),T[] toArray(T[] s); |
| 转成字符串 | toString(); |
| **集合的访问方式：集合的遍历** | |
| 遍历 | Iterator iterator();  ListIterator<E> listIterator(**int** index)  E get(**int** index); |

3、ArrayList类和Vector类

ArrayList和Vector都是基于数组实现的类，封装了一个动态再分配的Object[]数组。每个ArrayList和Vector都有一个capacity属性，capacity表示它们所封装的Object[]数组长度。

像ArrayList和Vector中添加大量元素时，可以用ensureCapacity方法一次性地增加capacity。这可以减少增加重分配的次数，提高性能。

如果开始就知道ArrayList和Vector要保存多少元素，可以在创建它们的时候就指定Capacity大小。如果创建空ArrayList和Vector集合时不指定Capacity，默认值为10。

操作capacity的两个方法：

**public void** ensureCapacity(**int** minCapacity):将ArrayList和Vector的capacity增加minCapacity。

**public void** trimToSize():调整ArrayList和Vector的capacity为列表当前大小。程序可调用该方法来减少集合对象存储空间。

ArrayList和Vector最显著的区别是：ArrayList是线程不安全的。Vector是线程安全的，所以Vector的性能要比ArrayList的性能低。

五、Queue接口

1、Queue集合概述

Queue用于模拟队列这种数据结构，队列通常是指“先进先出”的容器。队列头部保存在队列中时间最长的元素，队列尾部保存在队列中时间最短的元素。新元素插入（offer）到队列尾部，访问元素（poll）操作会返回队列头部的元素。通常，队列不允许随机访问队列中的元素。

2、Queue接口中方法

|  |
| --- |
| **Queue接口中方法** |
| boolean add(E e)：将指定元素插入尾部  boolean offer(E e)：将指定元素插入尾部,队列有容量限制时，比add方法好  E remove()：返回队列首元素并删除，当队列为空时，抛出异常  E poll()：返回队列首元素并删除，当队列为空时，返回null  E element()：返回队列首元素但不删除，当队列为空时，抛出异常  E peek()：返回队列首元素但不删除，当队列为空时，返回null |

3、Deque子接口

Deque是Queue的子接口，Deque取名自“Double ended Queue”,即双端队列，可以从队列的首尾两端进行插入和删除操作。所以该接口中新增了很多方法，如下：

|  |
| --- |
| **Deque接口中新增方法** |
| void addFirst(E e)：将指定元素插入首部  void addLast(E e)：将指定元素插入尾部  boolean offerFirst(E e)：将指定元素插入首部,插入失败会抛出异常  boolean offerLast(E e)：将指定元素插入尾部,插入失败会抛出异常  E removeFirst()：返回队列首元素并删除，当队列为空时，抛出异常  E removeLast()：返回队列尾元素并删除，当队列为空时，抛出异常  E pollFirst()：返回队列首元素并删除，当队列为空时，返回null  E pollLast()：返回队列尾元素并删除，当队列为空时，返回null  E getFirst()：返回队列首元素但不删除，当队列为空时，抛出异常  E getLast()：  E peekFirst()：返回队列首元素但不删除，当队列为空时，返回null  E peekLast()：  void push(E e)：将一个元素push进该双向队列所表示的栈中（即队列头）  E pop()：pop出该双向队列所表示的栈中第一个元素  Iterator<E> iterator()：  Iterator<E> descendingIterator()：以逆序返回队列所有元素 |

4、LinkedList实现类

LinkedList是实现了Deque和List接口的集合，可以根据索引来随机访问集合中的元素当成List来使用，也可以当成双向队列来使用，还可以当成“栈”来使用。但是LinkedList的存储机制并不是数组而是采用链表的数据结构。示例：

**public** **class** TestLinkedList {

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

LinkedList<String> books = **new** LinkedList<String>();

//向队列插入元素，即向集合添加元素

books.**offer**("Struts");

System.*out*.println(books);

books.**push**("hibernate");

System.*out*.println(books);

books.**add**("mybatis");

System.*out*.println(books);

books.**offerFirst**("spring");

System.*out*.println(books);

books.**offerLast**("servelet");

System.*out*.println(books);

books.**addFirst**("jsp");

System.*out*.println(books);

books.**addLast**("springMVC");

System.*out*.println(books);

//遍历集合

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

System.*out*.println(books.get(i));

}

//迭代遍历集合

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

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

String book = it.next();

System.*out*.print(book+",");

}

//反向迭代遍历集合

Iterator<String> dit = books.descendingIterator();

**while**(dit.hasNext()){

String book = dit.next();

System.*out*.print(book+",");

}

//访问集合元素

String bk1 = books.getFirst();

String bk2 = books.peekFirst();

String bk3 = books.getLast();

String bk4 = books.peekLast();

String bk5 = books.element();

String bk6 = books.peek();

System.*out*.println(bk1+"\n"+bk2+"\n"+bk3+"\n"+bk4+"\n"+bk5+"\n"+bk6);

//删除集合元素

System.*out*.println(books.remove()+"\n"+books);

System.*out*.println(books.removeLast()+"\n"+books);

System.*out*.println(books.pop()+"\n"+books);

System.*out*.println(books.poll()+"\n"+books);

System.*out*.println(books.pollLast()+"\n"+books);

}

}

5、LinkedList，ArrayList和Vector的性能比较

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **实现机制** | **随机访问** | **迭代操作** | **插入操作** | **删除操作** |
| 数组 | 连续内存区保存元素 | 1 | 不支持 | 不支持 | 不支持 |
| ArrayList | 内部以数组保存元素 | 2 | 2 | 2 |  |
| Vector | 内部以数组保存元素 | 3 | 3 | 3 | 3 |
| LinkedList | 内部以链表保存元素 | 4 | 1 | 1 | 1 |

Vector有一个Stack子类，Stack的本质依然是一个Vector，只是增加了5个方法而已，Stack是一个线程安全的类，这也是为了让Stack和Vector保持一致--Vector也是一个线程安全的类。

从序列化机制的角度来看，ArrayList的实现比Vector的实现更安全。ArrayList使用transient修饰了elementData数组。这保证系统序列化ArrayList对象时不会直接序列化elementData数组，而是通过ArrayList提供的writeObject、readObject方法来实现定制序列化；但对于Vector而言，它没有使用transient修饰elementData数组，而且Vector只提供了一个writeObject方法，并未完全实现定制序列化。Vector唯一的好处是它是线程安全的。

ArrayList的性能总体上优于LinkedList。但如果程序经常需要添加、删除元素，尤其是经常需要调用add(E e)方法向集合中添加元素时，则应该考虑使用LinkedList集合。

六、Map接口

1、Map集合概述

Map用于保存具有映射关系的数据，因此map集合中保存着两组值，一组保存key,一组保持value,key和value可以是任何引用类型的数据。Map中的key不允许重复。

2、Map中的方法

|  |  |
| --- | --- |
| **集合元素相关的操作：添加、删除、包含、比较、清除、判断为空、集合大小** | |
| 添加 | V put(K key, V value);  **void** putAll(Map<? **extends** K, ? **extends** V> m); |
| 删除 | V remove(Object key); |
| 包含 | **boolean** containsKey(Object key)  **boolean** containsValue(Object value); |
| 比较 | **boolean** equals(Object o) |
| 清除 | **void** clear(),删除集合中所有元素 |
| 判断为空 | **boolean** isEmpty() |
| 集合大小 | **int** size() |
| **集合类型转化的操作** | |
| Key->Set | Set<K> keySet(); |
| ->entrySet | Set<Map.Entry<K, V>> entrySet(); |
| ->Collection | Collection<V> values(); |
| **集合的访问方式：集合的遍历或访问** | |
| 遍历或访问 | E get(**Object** key); |

3、实现类HashMap和HashTable

HashTable是一个线程安全的Map实现，但HashMap非线程安全，所以HashMap性能比HashTable性能高点；

HashTable不允许使用null作为Key和value,如果试图将null放进HashTable中，将引发NullPointerException;HashMap可以将null作为key和value,但最多只有一项key-value对的key为null,可以有多对key-value的value为null;

对于HashMap而言，系统将value当成key的附属，系统根据Hash算法来决定key的存储位置，这样可以保证快速存、取集合key，而value总是紧随key存储。

当程序试图将一个key-value对放入HashMap中时，首先根据该key的hashCode()返回值决定该Entry的存储位置：如果两个Entry的key的hashCode()返回值相同，那它们的存储位置相同；如果这两个Entry的key通过equals比较返回true，新添加Entry的value将覆盖集合中原有Entry的value，但key不会覆盖；如果这两个Entry的key通过equals比较返回false，新添加的Entry将与集合中原有Entry形成Entry链。当系统开始初始化HashMap时，系统会创建一个长度为capacity的Entry数组。这个数组里可以存储元素的位置被称为“桶（bucket）”，每个bucket都有其指定索引，系统可以根据其索引快速访问该bucket里存储的元素。无论何时，HashMap的每个“桶”只存储一个元素（即一个Entry）。

七、操作集合的工具类Collections

1、排序操作

Collections提供了如下方法用于对List集合元素进行排序：

|  |
| --- |
| **Queue接口中方法** |
| Static void sort(List<T> list)：对集合元素进行升序排序  void sort(List<T> list, Comparator<? super T> c)：根据Comparator产生的顺序对集合元素排序  static void reverse(List<?> list)：反转指定集合元素顺序  static void shuffle(List<?> list)：对集合元素进行随机排序  static void shuffle(List<?> list, Random rnd)：按随机度rnd进行随机排序 |

2、查找、替换操作

查找、替换操作的方法比较多，参考Collection源代码。

3、同步控制

Collections提供了多个synchronizedXXX方法，该方法返回指定集合对象对应的同步对象，从而可以解决多线程并发访问集合时的线程安全问题。

示例：

**public** **class** TestSynchornCollection {

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

Collection<String> c = Collections.*synchronizedCollection*(**new** ArrayList<String>());

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

Set<String> set = Collections.*synchronizedSet*(**new** HashSet<String>());

Map<String,Object> map = Collections.*synchronizedMap*(**new** HashMap<String,Object>());

}

}

第二章 输入输出流

一、File类

1、File概述

File类是java.io包下代表与平台无关的文件和目录，文件和目录都是通过File来操作的，File能新建、删除和重命名文件和目录，File不能访问文件内容本身。File类提供了很多操作文件和目录的方法，File通过指定的路径名来创建File实例,路径名可以是绝对路径，也可以是相对路径，默认情况下，系统总是依据用户的工作路径来解释相对路径。

2、访问文件和目录

|  |
| --- |
| **访问文件名或目录名相关方法** |
| String getName():获取文件名或目录名  String getParent():获取上一级目录名  String getPath():获取路径名  String getAbsolutePath():获取绝对路径名  File getParentFile():获取上一级目录的File对象  File getAbsoluteFile():获取绝对路径的File对象 |
| **文件检测相关方法** |
| boolean exists():文件或目录是否存在  boolean isDirectory():判断是文件目录，不是文件  boolean isFile():判读是文件，不是文件目录  boolean isHidden():判断是否是隐藏文件  boolean isAbsolute():判断是否是绝对路径  boolean canRead():判断文件可读  boolean canWrite():判断文件可写 |
| **获取常规文件信息** |
| long lastModified():最后一次修改时间  long length():文件内容长度 |
| **文件或目录操作相关方法：创建，删除，遍历…** |
| **boolean createNewFile()**:创建一个新的File  **boolean delete()**:删除当前File  **boolean mkdir()**:创建目录  **boolean mkdirs()**:创建文件当前和上一级目录  **boolean renameTo(File dest)**:重命名  **String[] list()**:列出File对象的所有子文件和目录名  **File[] listFiles()**:列出File对象的所有子文件和目录名 |

3、示例

**public** **class** TestFile {

@Test

**public** **void** TestCreatObjectFile(){

/\*1.从文件路径创建File实例，不管当前路径下的文件或文件目录是否存在，都会 创建基于该文件名或文件目录的file实例\*/

File frompath1 = **new** File("d:/iofile/myfristfile.txt");

File frompath2 = **new** File("d:\\iofile1");

File frompath3 = **new** File("relativepathfile.txt");

/\*2.File file=new File(File parent,String child);根据parent实例 所在目录，创建该目录下的child文件

注意：parent必须是文件目录，child为文件名，parent本地目录必须存在， child不能为空\*/

File fromPrtF = **new** File(**new** File("d:/iofile"),"cfpf.txt");

/\*3、File file=new File(String parent,String child);根据parent 目录名，创建该目录下的child文件\*/

File fromPrtS = **new** File("d:/iofile","comefpsfile.txt");

//4.File file=new File(URI uri);从网络地址的文件创建file实例

URI uri;

**try** {

uri = **new** URI("www.baidu.com/hello.txt");

File filefromUri = **new** File(uri);

} **catch** (URISyntaxException e1) {

e1.printStackTrace();

}

/\* 获取文件名、文件目录名，文件路径，绝对路径，上一级目录,文件最后一次 修改时间 \*/

String fileName1 = frompath1.getName();

String pathNamea = frompath1.getAbsolutePath();

String pathNamer = frompath1.getPath();

String pathNamep = frompath1.getParent();

**long** x = frompath1.lastModified();

Date date = **new** Date(frompath1.lastModified());

System.*out*.println(x+"\n"+date);

System.*out*.println("文件名："+fileName1+"绝对路径："+pathNamea+"路径："+pathNamer+"上一级目录："+pathNamep);

/\* 判断文件、目录是否存在，判断文件是否可读可写，判断是文件还是文件目录 \*/

**boolean** judgeF1 = frompath1.exists();

**boolean** judgeI1 = frompath1.isFile();

**boolean** judgeI2 = frompath1.isDirectory();

**boolean** judgeC1 = frompath1.canRead();

**boolean** judgeC2 = frompath1.canWrite();

System.*out*.println("是否存在："+judgeF1+"\n是文件："+judgeI1+"\n是目录："+judgeI2+"\n可读："+judgeC1+"\n可写："+judgeC2);

/\* 文件、文件目录的创建，删除，遍历 \*/

**boolean** cnf;

**try** {

cnf = frompath1.createNewFile();

System.*out*.println("是否创建成功："+cnf);

} **catch** (IOException e) {

e.printStackTrace();

}

**boolean** del = frompath2.delete();

System.*out*.println("是否删除成功："+del);

**boolean** mkd = frompath2.mkdir();

System.*out*.println("目录是否创建成功："+mkd);

File filemkds = **new** File("d:/iofile3/iofile1/io");

**boolean** mkds = filemkds.mkdirs();

System.*out*.println("创建文件当前和上一级目录："+mkds);

File filelist = **new** File("d:/BaiduYunDownload");

String[] lists = filelist.list();

**for**(String list: lists){

System.*out*.println(list);

}

File[] listf = filelist.listFiles();

**for**(File list: listf){

System.*out*.println("文件名："+list.getName());

System.*out*.println("路径："+list.getPath());

}

}

}

二、Java的IO流

1、流的分类

|  |
| --- |
| 输入流和输出流 |

|  |
| --- |
| 输入流：只能读取数据；  输出流：只能写出数据；  tip：上述输入输出流涉及到一个方向问题，是从程序运行所在的内存的角度来划分的， 数据保存到硬盘的过程为输出，从硬盘读取数据到程序运行内存为输入。对于网络编程来讲，从服务器通过网络流向客户端为输出，从客户端到服务器为输入。 |

|  |
| --- |
| 字节流和字符流 |
| 字节流：字节流操作数据单元为8位的字节；  字符流：字符流操作数据单元为16位的字符；  Tip：字节流由InputStream和OutputStream做基类，字符流由Reader和Writer做基类。 |
| 节点流和处理流 |
| 节点流：直接从/向一个特定的IO设备（如磁盘、网络）读/写数据的流；  处理流：通过封装节点流之后来实现数据读写的流，也称高级流； |

2、InputStream、OutputStream和Reader、Writer

InputStream和Reader是所有输入流的基类，OutPutStream和Writer是所有输出流的基类，它们是四个抽象类，本身不能创建实例;

常用方法：

|  |  |
| --- | --- |
| **InputStream** | **int** read() |
| **int** read(**byte** b[]) |
| **int** read(**byte** b[],**int** off,**int** len) |
| **Reader** | **int** read() |
| **int** read(**char** cbuf[]) |
| **int** read(**char** cbuf[],**int** off,**int** len) |
| **int** read(java.nio.CharBuffer target) |
| **OutputStream** | **void** write(**int** b) |
| **void** write(**byte** b[]) |
| **void** write(**byte** b[],**int** off,**int** len) |
| **Writer** | **void** write(**int** c) |
| **void** write(**char** cbuf[]) |
| **void** write(**char** cbuf[],**int** off,**int** len) |
| **void** write(String str) |
| **void** write(String str,**int** off,**int** len) |

3、FileInputStream,FileOutputStream,FileReader,FileWriter

这四种输入输出流是节点流，直接从物理节点读取获取流信息。这四种流也是使用最多的节点流，很多处理流都通过包装这四种节点流使操作更简捷。

使用示例：

4、处理流的使用

示例：

5、输入/输出流体系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **分类** | **字节输入流** | **字节输出流** | **字符输入流** | **字符输出流** |
| **抽象基类** | *InputStream* | *OutputStream* | *Reader* | *Writer* |
| **访问文件** | **FileInputStream** | **FileOutputStream** | **FileReader** | **FileWriter** |
| **访问数组** | ByteArrayInputStream | ByteArrayInputStream | CharArrayReader | CharArrayWriter |
| **访问管道** | PipedInputStream | PipedInputStream | PipedReader | PipedWriter |
| **访问字符串** |  |  | StringReader | StringWriter |
| **缓冲流** | **BufferedInputStream** | **BufferedOutputStream** | **BufferedReader** | **BufferedWriter** |
| **转换流** |  |  | **InputStreamReader** | **OutputStreamWriter** |
| **对象流** | **OjbectInputStream** | **ObjectOutputStream** |  |  |
| **抽象基类** | *FilterInputStream* | *FilterOutputStream* | *FilterReader* | *FilterWriter* |
| **打印流** |  | **PrintStream** |  | **PrintWriter** |
| **推回输入流** | PushbackInputStream |  | PushbackReader |  |
| **特殊流** | DataInputStream | DataOutputStream |  |  |

6、RandomAccessFile

RandomAccessFile是java输入/输出流体系中功能最丰富的文件内容访问类，它提供了众多方法来访问文件内容，它既可以读取文件内容也可以向文件输出数据。

示例：

**public** **void** TestRandomAccess(){

RandomAccessFile raf1 = **null**;

RandomAccessFile raf2 = **null**;

**try**{

raf1=**new** RandomAccessFile(**new** File("hello.txt"),"r");

raf2=**new** RandomAccessFile(**new** File("hello1.txt"),"rw");

**byte**[] b = **new** **byte**[25];

**int** len;

**while**((len = raf1.read(b))!=-1){

//1.字符读取方式

**for**(**int** i = 0;i<len;i++){

System.*out*.print((**char**) b[i]);

}

//2.字符串读取方式

System.*out*.print(**new** String(b,0,len));

raf2.write(b,0,len);

}}**catch** (IOException e) {

e.printStackTrace();

}**finally**{ **//……** }

//实现的是覆盖效果

**public** **void** TestRandomAccess1(){

String cc="i love you";

RandomAccessFile raf=**null**;

**try**{

raf = **new** RandomAccessFile(**new** File("hello.txt"),"rw");

raf.seek(5);

raf.write(cc.getBytes());

}**catch** (IOException e) {

e.printStackTrace();

}**finally**{

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

**try**{

raf.close();

}**catch**(IOException e){

e.printStackTrace();

}

}

}

}

//实现插入效果

**public** **void** TestRandomAccess2(){

String cc="i love you";

RandomAccessFile raf=**null**;

**try** {

raf = **new** RandomAccessFile(**new** File("hello.txt"),"rw");

raf.seek(5);

String str = raf.readLine();

raf.seek(5);

raf.write(cc.getBytes());

raf.write(str.getBytes());

}**catch** (IOException e) {

e.printStackTrace();

}**finally**{

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

**try** {

raf.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

//实现复杂的插入效果

**public** **void** TestRandomAccess3(){

String cc="i love you";

RandomAccessFile raf=**null**;

**try** {

raf = **new** RandomAccessFile(**new** File("hello.txt"),"rw");

raf.seek(6);

**byte**[] b =**new** **byte**[20];

**int** len;

StringBuffer sb = **new** StringBuffer();

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

sb.append(**new** String(b,0,len));

}

raf.seek(6);

raf.write(cc.getBytes());

raf.write(sb.toString().getBytes());

}**catch** (IOException e) {

e.printStackTrace();

}**finally**{

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

**try** {

raf.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

三、对象序列化

1、序列化的含有与意义

序列化机制允许将实现序列化的Java对象转换为字节序列，这些字节序列可以被保存在磁盘上，或通过网络传输，以备以后重新恢复成原来的对象。序列化机制可以使得对象可以脱离程序的运行而独立存在。

对象的序列化（Serialize）是指将一个Java对象写入IO流中，与此对应的是，对象的反序列化(Deserialize)是指从IO流中恢复该java对象。

如果需要让某个对象可以支持序列化机制，必须让它的类是可序列化的，为了让某个类是可序列化的，该类必须实现如下两个接口之一：

* Serializable
* Externalizable

Serializable是一个标记接口，实现该接口无需实现任何方法。

2、使用对象流进行序列化

* 对象流必须实现对象的序列化，如何实现，通过实现serializable 接口
* 对象的属性也必需是可序列化的
* 保证对象的反序列化，需要显示申明序列版本号，即定义private static final long serialVersionUID 为固定值
* 对象的属性或方法有static关键字标识，不能使用对象流，是非序列化的。

示例：

//定义两个可序列化类

**class** Person **implements** java.io.Serializable{

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** String name;

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

**private** Pet pet;

**public** Person(String name,**int** age,Pet pet){

**this**.name = name;

**this**.age = age;

**this**.pet = pet;

}

**public** **void** autoDescrible(){

System.*out*.println("my name is"+name+" and odd is"+age);

}

@Override

**public** String toString() {

**return** "Person [name=" + name + ", age=" + age + ", pet=" + pet + "]";

}

//省略getter，setter方法…

}

**class** Pet **implements** Serializable{

/\*\*

\* 版本序列号，显示申明，便于序列化与反序列的正确性

\*/

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** String name;

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

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

**this**.age = age;

**this**.name = name;

}

@Override

**public** String toString() {

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

}

//省略getter，setter方法…

}

**public** **class** TestSerializable{

@Test

**public** **void** TestObjOutputStream(){

Pet p1 = **new** Pet("dog",1);

Pet p2 = **new** Pet("cat",2);

Person prs1 = **new** Person("小师弟",21,p1);

Person prs2 = **new** Person("大师兄",22,p2);

ObjectOutputStream oos = **null**;

**try**{

oos = **new** ObjectOutputStream(

**new** FileOutputStream(**new** File("testobj.txt")));

oos.writeObject(prs1);oos.flush();

oos.writeObject(prs2);oos.flush();

}**catch**(IOException e){

e.printStackTrace();

}**finally**{

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

**try** {

oos.close();

} **catch** (IOException e){

e.printStackTrace();

}

}

}

}

@Test

**public** **void** TestObjectInputStream(){

ObjectInputStream ois=**null**;

**try** {

ois = **new** ObjectInputStream(

**new** FileInputStream(**new** File("testobj.txt")));

Person p1=(Person) ois.readObject();

System.*out*.println(p1);

Person p2 =(Person) ois.readObject();

System.*out*.println(p2);

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

//省略…

}

}

}

Java序列化特殊算法的内容：

* 所用保存到磁盘中的对象都有一个序列化编号；
* 当程序试图序列化一个对象时，程序将先检查该对象是否已经被序列化过，只有当该对象从未（在本次虚拟机中）被序列化过，系统才会将该对象转换成字节序列并输出；
* 如果某个对象是已经序列化过的，程序将直接只输出一个序列化编号，而不是再次重新序列化该对象。

3、自定义序列化

当对某个对象进行序列化时，系统会自动把该对象的所有属性依次进行序列化，如果某个属性引用到另一个对象，则被引用的对象也会被序列化，如果引用的对象的属性也引用了其他对象，则被引用的对象也会被序列化，这种被称为递归序列化。

在一些特殊场景下，如果某个类里包含的属性值是敏感信息，例如银行帐户，这时候不希望系统将该属性值进行序列化；通过在属性前面加上transient关键字，可指定java序列化时无需指定该属性值。使用transient修饰的属性，在反序列化恢复java对象时无法取得该属性的值。为了弥补这一缺陷，java还提供了一种自定义序列化机制。

用于实现自定义序列化的特殊方法：

|  |
| --- |
| **private** **void** writeObject(java.io.ObjectOutputStream out)**throws** IOException |
| **private** **void** readObject(java.io.ObjectInputStream in)  **throws** IOException,ClassNotFoundException |
| **private** **void** readObjectNoData() **throws** ObjectStreamException |
| **private** Object writeReplace() **throws** ObjectStreamException |
| **private** Object readResovle() **throws** ObjectStreamException |

writeObject方法负责写入特定类的实例的状态，以便相应的readOjbect方法可以恢复它。通过重写该方法，程序员可以完全获得对序列化机制的控制，程序员可以自主决定哪些属性需要序列化，需要怎样序列化。默认情况下，该方法会调用out.defaultWriteOjbect来保存j对象的属性，从而可以实现序列化java对象状态的目的。

readObject方法负责从流中读取并恢复对象属性，通过重写该方法，程序员可以完全获得对反序列化的控制，可以自主决定需要反序列化哪些属性，以及怎样反序列化。默认情况下，该方法会调用in.defaultReadObject来恢复Java对象的非静态和非瞬态属性。通常情况下，readObject方法与writeObject方法对应，如果writeObject方法中对Java对象的属性进行了一些处理，则应该在readObject方法中对其属性进行相应的操作，以便正确恢复该对象。

writeReplace方法由序列化调用，前提是该方法存在。该方法可以拥有private、protected、default等访问权限，所以其子类有可能获得该方法。Java序列化机制保证在序列化某个对象之前，先调用该对象的writePlace方法，如果该方法返回另一个对象，则系统转为序列化另一个对象。

readResolve方法会紧接这readObject方法之后调用，该方法的返回值将替代原来反序列化的对象，而原来readObject反序列化的对象将会立即丢弃。readResolve在序列化枚举类、单例类时尤其有用。注意使用jdk1.5之后的enum定义的枚举没有任何问题。所以所有单例类、枚举类实现序列化都应该提供readResolve方法，这样才可以保证反序列化的对象依然正常。

实例1，重写writeObject与readOjbect:

**class** Account **implements** java.io.Serializable{

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** String name;

**private** **double** balance;

**public** Account(String name,**double** balance){

**this**.name =name;

**this**.balance = balance;

}

**private** **void** writeObject(java.io.ObjectOutputStream out)

**throws** IOException{

out.writeObject(**new** StringBuffer(name).reverse());

/\*\*对balance不进行任何处理

out.writeObject(balance);\*/

**byte**[] b = String.*valueOf*(balance).getBytes();

//对balance进行64位加密处理

String s = **new** BASE64Encoder().encode(b);

out.writeObject(s);

**private** **void** readObject(java.io.ObjectInputStream in)

**throws** IOException,ClassNotFoundException{

/\*\*对name进行字符串反转处理

**this**.name =

((StringBuffer) in.readObject()).reverse().toString();\*/

//对name不进行任何处理

**this**.name = ((StringBuffer)in.readObject()).toString();

String bs = (String) in.readObject();

//对balance进行64位解密处理

String s =

**new** String(**new** BASE64Decoder().decodeBuffer(bs),"utf-8");

**this**.balance = Double.*parseDouble*(s);

//this.balance = (Double)in.readObject();

}

**public** String toString(){

**return** "name is "+name+",balance is "+balance;

}

}

实例2，重写writeReplace方法

**class** Account1 **implements** java.io.Serializable{

**private** String name;

**private** **double** balance;

**public** Account1(String name,**double** balance){

**this**.name =name;

**this**.balance = balance;

}

**private** Object writeReplace() **throws** ObjectStreamException{

ArrayList<Object> list = **new** ArrayList<Object>();

list.add(name);

list.add(balance);

**return** list;

}

@Override

**public** String toString(){

**return** "name is "+name+",balance is "+balance;

}

}

实例3，重写readResolve方法

**class** Account2 **implements** java.io.Serializable{

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** **static** **final** Account2 *instance* = **new** Account2();

**private** String name;

**private** **int** balance;

**private** Account2(){

}

**public** **static** Account2 getInstance(){

**return** *instance*;

}

//多用于枚举类，单例类中

**private** Object readResolve() **throws** ObjectStreamException{

**return** *instance*;

}

//getter,setter方法省略

}

自定义序列化测试实例：

**public** **class** TestAutoDefineSerial{

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

ObjectOutputStream oos = **null**;

ObjectInputStream ois = **null**;

File file = null;

**try**{

file = **new** File("ads.txt");

oos = **new** ObjectOutputStream(**new** FileOutputStream(file));

/\*\*实例1，writeObject、readObject测试

Account ac = **new** Account("Trump",567.5);

oos.writeObject(ac);

\*/

/\*\*实例2，writeReplace测试

Account1 ac = **new** Account1("Trump",555);

oos.writeObject(ac);

\*/

/\*\*实例3，readResolve测试

Account2 ac = Account2.*getInstance*();

ac.setName("Trump");

ac.setBalance(521);

oos.writeObject(ac);

\*/

/\*\*实例4，Externalizalble接口测试

Account3 ac = **new** Account3("Trump",555);

oos.writeObject(ac);

\*/

oos.flush();

ois = **new** ObjectInputStream(**new** FileInputStream("ads.txt"));

**try**{

/\*\*实例1，writeObject、readObject测试

Account gac = (Account) ois.readObject();

System.*out*.println(gac);

\*/

/\*\*实例2，writeReplace测试

ArrayList<Object> gac=(ArrayList<Object>) ois.readObject();

System.*out*.println(gac);

\*/

/\*\*实例3，writeResovle测试

Account2 gs = (Account2) ois.readObject();

System.*out*.println(gs == s);//当单例类无readResolve时返回false

\*/

/\*\*实例4，Externalizalble接口测试

Account3 gac = (Account3) ois.readObject();

System.*out*.println(gac);

\*/

}**catch**(ClassNotFoundException e){ e.printStackTrace(); }

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

**finally**{

**try**{

**if**(ois!=**null**){ ois.close(); }

**if**(oos!=**null**){ oos.close(); }

}**catch**(Exception e2){ e2.printStackTrace(); }

}

}

}

4、实现Externalizable接口实现自定义序列化

实例4：

**class** Account3 **implements** java.io.Externalizable{

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** String name;

**private** **double** balance;

/\* 无参构造器一定需要，否则在反序列化的时候会报错，说明实现Externalizable反序 列化对象是根据无参构造器来实例化对象的 \*/

**public** Account3(){}

**public** Account3(String name,**double** balance){

**this**.name =name;

**this**.balance = balance;

}

**public** String toString(){

**return** "name is "+name+",balance is "+balance;

}

@Override

**public** **void** writeExternal(ObjectOutput out) **throws** IOException{

out.writeObject(**new** StringBuffer(name).reverse());

**byte**[] b = String.*valueOf*(balance).getBytes();

String s = **new** BASE64Encoder().encode(b);

out.writeObject(s);

}

@Override

**public** **void** readExternal(ObjectInput in) **throws** IOException,

ClassNotFoundException{

**this**.name = ((StringBuffer)in.readObject()).toString();

String bs = (String) in.readObject();

String s=**new** String(**new** BASE64Decoder().decodeBuffer(bs),"utf-8");

**this**.balance = Double.*parseDouble*(s);

}

}

两种序列化的比较

|  |  |
| --- | --- |
| **实现Serializable接口** | **实现Externalizable接口** |
| 系统自动存储必要信息 | 程序员决定存储哪些信息 |
| Java内建支持，易于实现，只需实现接口，即可，无需任何代码支持 | 仅仅提供两个空方法，实现该接口必须为这两个方法提供实现 |
| 性能略差 | 性能略高 |

第三章 多线程

一、线程概述

线程是进程的组成部分，一个进程可以拥有多个线程，一个线程必须有一个父进程。线程可以拥有自己的堆栈、自己的程序计数器和自己的局部变量，但不再拥有系统资源，它与父进程的其他线程共享该进程所拥有的全部资源。

线程的执行是抢占式的，也就是说，当前运行的线程在任何时候都可能被挂起，以便另一个线程可以运行。一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。

一个程序运行后至少有一个进程，一个进程里可以包含多个线程，但至少要包含一个线程。

二、线程的创建和启动

1、继承Thread类创建线程类

Thread类创建并启动线程步骤：

①、创建线程子类（SubThread）继承Thread类；

②、子类中重写run（）方法；

③、主线程（main方法）中，实例化当前子线程类；

④、调用start（）方法，以启动子线程，并执行run()方法。

示例：

**class** SubThread **extends** Thread{

**public** **void** run(){

**for**(**int** i=1;i<=100;i++){

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

}

}

}

**public** **class** TestThread{

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

SubThread subThread = **new** SubThread();

subThread.start();

**for**(**int** i=1;i<=100;i++){

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

}

}

}

**Tip:**

main方法的方法体就是java程序主线程的线程执行体。

继承Thread类创建的线程类，多条线程之间无法共享线程类的实例变量。

* Thread.*currentThread*():返回当前正在执行的线程对象；
* getName()：返回调用该线程的名字。

2、实现Runnable接口创建线程类

实现Runnable接口创建并启动多条线程的步骤：

①、定义实现Runnable接口的实现类；

②、重写run()方法；

③、创建一个Runnable接口实现类的对象；

④、将该对象作为形参传递给Thread类的构造器中，创建Thread类的对象，该 Thread类对象即为一个线程对象；

⑤、调用start()，启动线程：执行Thread对象构造器中形参对象的run()方法。

示例：

//第二种方式实现多线程：实现Runnable 接口的方式

**class** PrintThread **implements** Runnable{

**public** **void** run(){

**for**(**int** i=1;i<=100;i++){

**if**(i%2==0){

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

}

}

}

}

**public** **class** TestThreadImp{

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

PrintThread p1 = **new** PrintThread();

Thread t1 = **new** Thread(p1);

t1.start();

}

}

3、两种方式的比较

1.联系：都必须实现Runnable接口，并实现run()方法；

2.实现方式更好：

①、规避了java单继承的局限性；

②、如果多个线程要同时操作一份资源（或数据），实现方式更优。

4、Thread类常用方法

|  |  |
| --- | --- |
| 1 | **synchronized** **void** start()：启动线程，并调用run()方法 |
| 2 | **void** run()：实现子类必须重写，所有任务执行体都在run()中 |
| 3 | **static** **native** Thread currentThread()：获取当前线程对象 |
| 4 | **final** **void** setName(String name)：设置线程名 |
| 5 | **final** String getName()：获取线程名 |
| 6 | **static** **native** **void** yield();：调用此方法的线程释放当前CPU的执行权 |
| 7 | **final** **void** join()：  在A线程中调用B线程的join(),表示：当执行此方法，A线程停止执行，直至B线程执行完毕，A线程再接着执行 |
| 8 | **final** **synchronized** **void** join(**long** millis)：  强制让调用此方法的的线程执行millis毫秒 |
| 9 | **final** **native** **boolean** isAlive()：判断线程是否还存活 |
| 10 | **static** **native** **void** sleep(**long** millis)：让当前线程睡眠ms毫秒 |
| 11 | **final** **void** setPriority(**int** newPriority)：改变线程优先级 |
| 12 | **final** **int** getPriority()：返回线程优先值 |

三、线程的生命周期

1、**new** （新创建）

当用new操作符实例化一个线程对象的时候，该线程处于该状态，该线程对象只是被实例化了，还没有运行。

2、**Runnable** （可运行）

当调用start()方法时，线程处于可运行状态，一个可运行的线程可能正在运行也可能没有运行，这取决于操作系统给线程提供运行的时间。

**Tip:**

不要对已经处于启动状态的线程再次调用start方法，否则将引发IllegalThreadStateException异常。

3、**Blocked** （被阻塞）

当一个线程试图获取一个内部的对象锁，而该锁被其他线程持有，则该线程进入阻塞状态。

4、**Waiting** （等待）

当线程等待另一个线程通知调度器一个条件时，它自己进入等待状态。调用Object.wait()和Thread.join()方法时，或者是调用java.util.concurrent库中的Lock或Condition时，就会出现这种情况。

5、**Timed\_waiting** （计时等待）

有几个方法有一个超时参数，调用它们将进入计时等待状态。这些方法为:

* **Thread.sleep(long l),**
* **Object.wait(long l),**
* **Thread.join(long l),**
* **Lock.TryLock(long l),**
* **Condition.awiat(long l)。**

6、**Terminated** （被终止）

有两个原因导致线程被终止：1、因为run方法正常退出而自然死亡；2、因为一个没有被捕获的异常终止了run方法而意外死亡。

要确定一个线程的状态，可以调用getState()方法。线程的阻塞和等待没有太大区别，只是一个是主动，一个是被动。

四、控制线程

1、join线程

在A线程中调用B线程的join方法，会让A线程等待B线程完成之后再运行。A线程进入阻塞状态，直到B线程运行完成。

2、线程睡眠：**sleep**

调用当前线程类的sleep方法会使当前线程从运行状态进入阻塞状态，在sleep时间段内，该线程不会获得执行的机会，即使系统中没有其他可运行的线程，处于sleep中的线程也不会运行，因此sleep方法常用来暂停程序的执行。

3、线程让步：**yield**

yield方法可以让当前正在执行的线程暂停，但是不会阻塞该线程，它只是将线程转入就绪状态，让系统的线程调度器重新调度一次。

当某个线程调用了yield方法后，只有优先级与当前线程相同或者优先级比当前线程更高的就绪状态的线程才会获得执行机会。

4、线程优先级

每个线程执行时都具有一定的优先级，优先级高的线程获得较多的执行机会，而优先级低的线程则获得较少的执行机会。

每个线程默认的优先级都与创建它的父线程具有相同的优先级，在默认情况下，main线程具有普通优先级。

Thread类提供了三个关于优先级静态常量，分别是：

/\* The minimum priority that a thread can have. \*/

**public** **final** **static** **int** *MIN\_PRIORITY* = 1;

/\* The default priority that is assigned to a thread. \*/

**public** **final** **static** **int** *NORM\_PRIORITY* = 5;

/\* The maximum priority that a thread can have. \*/

**public** **final** **static** **int** *MAX\_PRIORITY* = 10;

5、后台线程

有一种线程，它是在后台运行的，它的任务是为其他的线程提供服务，这种线程被称为“后台线程(Daemon Thread)，”又称“守护线程”。Jvm的垃圾回收线程就是典型的后台线程。

后台线程有个特征：如果前台线程都死亡，后台线程会自动死亡。

示例：

**public** **class** DaemonRunable **implements** Runnable {

@Override

**public** **void** run() {

**for**(**int** i=0;i<999;i++){

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

}

}

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

DaemonRunable dr = **new** DaemonRunable();

Thread tr = **new** Thread(dr);

//将此线程设置成后台线程

tr.setDaemon(**true**);

tr.start();

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

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

}

}

}

本来tr后台线程应该运行到i=999才结束，但运行时发现后台线程无法运行到999，

因为当主线程，上述程序中唯一的前台进程运行结束后，JVM会自动退出，因而后台线程也就被结束了。

**Tip:**

前台线程死亡后，JVM会通知后台线程死亡，但从它接收指令，到它做出响应，需要一定时间。setDaemon()方法必须在start之前运行，否则抛出异常。

6、join,sleep,yeild综合实例

**class** TestSleep **extends** Thread{

@Override

**public** **void** run() {

setName("Sleep");

**for**(**int** i=0;i<50;i++){

**try** {

//让当前线程睡眠100ms

TestSleep.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.*out*.println(getName()+":"+i);

}

}

}

**class** TestYield **extends** Thread{

@Override

**public** **void** run(){

setName("Yield");

**for**(**int** i=0;i<50;i++){

//暂停单前线程，让系统调度器重新调度

TestYield.*yield*();

System.*out*.println(getName()+":"+i);

}

}

}

**class** TestJoin **extends** Thread{

**private** TestSleep testSleep;

**private** TestYield testYield;

**public** TestJoin(TestSleep testSleep, TestYield testYield){

**this**.testSleep = testSleep;

**this**.testYield = testYield;

}

@Override

**public** **void** run() {

setName("Join");

**for**(**int** i=0;i<50;i++){

System.*out*.println(getName()+":"+i);

**if**(i==5){

**try** {

//暂停当前线程直至testYield运行完成

testYield.join();

} **catch** (InterruptedException e){

e.printStackTrace();

}

}**else** **if**(i==10){

**try**{

//暂停当前线程直至testSleep运行完成

testSleep.join();

}**catch**(InterruptedException e){

e.printStackTrace();

}

}

}

}

//getter,setter方法省略

}

**public** **class** TestThreadAllMeans {

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

TestSleep ts = **new** TestSleep();

TestYield ty = **new** TestYield();

TestJoin tj = **new** TestJoin(ts,ty);

ts.start();

ty.start();

tj.start();

**for**(**int** i=0;i<50;i++){

System.*out*.println(tj.getName()+":"+tj.getState());

System.*out*.println(ts.getName()+":"+ts.getState());

System.*out*.println(ty.getName()+":"+ty.getState());

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

}

}

}

五、线程同步

1、同步代码块

**synchronized**(同步监视器){

//需要被同步的代码块（即为操作共享数据的代码）

}

共享数据：多个线程共同操作同一个数据（变量）

同步监视器：由任意一个类的对象来充当，哪个线程获取该监视器，谁就执行大括号里被同步的代码。俗称：锁；

要求：所有的线程必须共用一个同步锁！

注：在实现的方式中，考虑同步的话，可以使用this来充当锁。但是在继承方式中，不能用this充当锁。

2、同步方法

将操作共享数据的的方法声明为**synchronized**。即此方法为同步方法，能够保证当其中的一个线程执行此方法时，其它线程等待。

同步方法也有同步监听器即锁，该锁为隐式的this对象。

3、同步锁（Lock）

该方式为显示定义同步锁对象来实现同步，同步锁使用Lock对象充当。Lock提供了比synchronized方法和代码块更广泛的锁定操作，支持多个相关的Condition对象。ReentrantLock锁具有重入性，ReentrantLock对象会维持一个计数器来追踪lock方法的嵌套调用，线程在每次调用lock（）加锁后，必须通过unlock()释放锁，所以一段被锁保护的代码可以调用另一被相同锁保护的方法。

**private** **final** Lock lock = **new** ReentrantLock();

//…

**public** **void** needLockMethod(){

lock.lock();

**try**{

//critical section

}**catch**(Exception e) { //… }

**finally**{

lock.unlock();

}

}

锁和条件：

* 锁用来保护代码片段，任何时刻只能有一个线程执行被保护的代码块；
* 锁可以管理试图进入被保护的代码段的线程；
* 锁可以拥有一个或多个相关的条件对象；
* 每个条件对象管理那些进入被保护的代码段但还不能运行的线程。

4、同步方法，同步代码块，同步锁实例代码

**public** **class** TestGetBanlance {

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

Account2 xm = **new** Account2("xiaoming",3000);

GetPerson gp = **new** GetPerson(xm);

Thread t1 = **new** Thread(gp);

Thread t2 = **new** Thread(gp);

Thread t3 = **new** Thread(gp);

t1.start();

t2.start();

t3.start();

}

}

**class** GetPerson **implements** Runnable{

**private** Account2 account;

**public** GetPerson(Account2 account){

**this**.account = account;

}

**public** **synchronized** **void** run() {

account.draw1(500);

account.draw2(200);

account.draw3(100);

}

//getter，setter方法省略

}

**class** Account2{

**private** String name;

**private** **int** balance;

**private** **final** Lock lock = **new** ReentrantLock();

**public** Account2(String name,**int** balance){

**this**.name = name;

**this**.balance = balance;

}

//同步锁

**public** **void** draw3(**int** aimCash){

lock.lock();

**try** {

**if**(balance<aimCash){

System.*out*.println("当前余额不足！");

}**else**{

balance=balance-aimCash;

System.*out*.println("M3当前余额："+balance);

}

}**catch**(Exception e) {

e.printStackTrace();

}**finally**{

lock.unlock();

}

}

//同步方法

**public** **synchronized** **void** draw1(**int** aimCash){

**if**(balance<aimCash){

System.*out*.println("当前余额不足！");

}**else**{

balance=balance-aimCash;

System.*out*.println("M1当前余额："+balance);

}

}

//同步代码块

**public** **void** draw2(**int** aimCash){

**synchronized**(**this**){

**if**(balance<aimCash){

System.*out*.println("当前余额不足！");

}**else**{

balance=balance-aimCash;

System.*out*.println("M2当前余额："+balance);

}

}

} }

六、线程通信

1、概述

当线程在系统内运行时，线程的调度具有一定的透明性，程序通常无法准确控制线程的轮换执行，但我们可以通过一些机制来保证线程协调运行。

2、线程协调运行

当使用同步方法和同步代码块的时候，可以使用wait(),notify(),notifyAll()使线程协调运行。

对于**synchronized**方法，同步监听器就是该类本身（**this**）,所以在同步方法中直接调用这三个方法；

对于**synchronized**代码块，同步监听器是括号内的对象，所以必须通过该对象调用这三个方法。

Java.lang.Object下的此三个方法：

**final** **void** wait():

导致当前线程等待，直到其他线程调用该同步监听器的notify()或notifyAll()方法来唤醒该线程。调用该方法的当前线程会释放对该同步监听器的锁定。

**final** **native** **void** notify():

唤醒在此同步监听器上等待的单个线程。如果所有线程都在此同步监听器上等待，则会唤醒其中一个线程。选择是任意的。只有当前线程放弃对该同步监听器的锁定后（使用wait()方法），才可以执行被唤醒的线程。

**final** **native** **void** notifyAll():

唤醒在此同步监听器上等待的所有线程。

3、条件变量控制协调

Condition使用方法如下代码所示：

**private** **final** Lock lock = **new** ReentrantLock();

**private** Condition condition = lock.newCondition();//进行线程通信

//…

**public** **void** needLockMethod(){

lock.lock();

**try** {

condition.await();

//critical section

condition.signalAll();

}**catch**(Exception e) { //… }

**finally**{

lock.unlock();

}

}

当使用同步锁（Lock）对象来保证线程同步时，Java提供了一个Condition类来保持协调。使用Condition可以让那些已经得到Lock对象却无法继续执行的线程释放Lock对象，Condition对象也可以唤醒其他处于等待的线程。

4、线程通信示例：

\*使用wait()协调线程运行：

**class** Account{

**private** String name;

**private** **double** balance;

//标识账户中是否已有存款的旗标

**private** **boolean** flag = **false**;

**public** Account(String name,**double** balance){

**this**.name = name;

**this**.balance = balance;

}

**public** **synchronized** **void** draw(**double** drawAmount){

**try**{

**if**(!flag){

**wait();**

}**else**{

balance-=drawAmount;

System.*out*.println("账户余额为："+balance);

flag = **false**;

**notifyAll();**//唤醒其他线程

}

}**catch**(InterruptedException ex){

ex.printStackTrace();

}

}

**public** **synchronized** **void** deposit(**double** depositAmount){

**try**{

**if**(flag){

**wait();**

}**else**{

balance+=depositAmount;

flag = **true**;

**notifyAll();**

}

}**catch**(InterruptedException ex){

ex.printStackTrace();

}

}

}

\*使用Condition对象控制协调

**class** Account{

**private** String name;

**private** **double** balance;

//标识账户中是否已有存款的旗标

**private** **boolean** flag = **false**;

**private** **final** Lock lock = **new** ReentrantLock();

**private** **final** Condition cdt = lock.newCondition();

**public** Account(String name,**double** balance){

**this**.name = name;

**this**.balance = balance;

}

**public** **void** draw(**double** drawAmount){

lock.lock();

**try**{

**if**(!flag){

**cdt.await();**

}**else**{

balance-=drawAmount;

System.*out*.println("账户余额为："+balance);

flag = **false**;

**cdt.signalAll();**//唤醒其他线程

}

}**catch**(InterruptedException ex){

ex.printStackTrace();

}

}

**public** **void** deposit(**double** depositAmount){

lock.lock();

**try**{

**if**(flag){

**cdt.await();**

}**else**{

balance+=depositAmount;

flag = **true**;

**cdt.signalAll();**

}

}**catch**(InterruptedException ex){

ex.printStackTrace();

}

}

}

\*线程通信两种方式测试代码：

**class** DrawThread **extends** Thread{

**private** Account account;

**private** **double** drawAmount;

**public** DrawThread(String name,Account account,**double** drawAmount){

**this**.account = account;

**this**.drawAmount = drawAmount;

}

**public** **void** run(){

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

account.draw(drawAmount);

}

}

}

**class** DepositThread **extends** Thread{

**private** Account account;

**private** **double** depositAmount;

**public** DepositThread(String name,Account account,**double** depositAmount){

**this**.account = account;

**this**.depositAmount = depositAmount;

}

**public** **void** run(){

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

account.deposit(depositAmount);

}

}

}

**public** **class** TestSynNotify {

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

Account acct = **new** Account("1234567",0);

**new** DrawThread("取钱者",acct,800).start();

**new** DepositThread("存款者甲",acct,800).start();

}

}

七、线程池

1、概述

系统启动一个新线程的成本是比较高的，所以在这种情况下，使用线程池可以大大提高性能。线程池在系统启动时即创建大量空闲的线程，将一个Runnable对象传给线程池，线程池就会启动一条线程来执行该对象的run方法，当run方法执行结束后，该线程并不会死亡，而是再次返回线程池中成为空闲状态，等待执行下个对象的run方法。

2、线程池的使用

①、调用Executors类的静态工厂方法创建一个ExecutorService对象，该对象 代表一个线程池；

②、创建Runnable实现类或Callable实现类的实例，作为线程执行任务；

③、调用ExecutorService对象的submit方法来提交Runnable实例或Callable 实例；

④、当不想提交任何任务时调用ExecutorService对象的shutdown方法来关闭 线程池。

3、示例代码

\*对线程通信实例的测试方法采用线程池的方式：

**public** **class** TestSynNotify {

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

Account acct = **new** Account("1234567",0);

//创建一个具有固定线程数的线程池

ExecutorService pool = Executors.*newFixedThreadPool*(6);

//向线程池中提交两个线程

pool.submit(**new** DrawThread1("取钱者",acct,800));

pool.submit(**new** DepositThread1("存款者",acct,800));

//关闭线程池

pool.shutdown();

}

}

八、ThreadLocal模式（以空间换时间）

**Tip:**

ThreadLoacl模式用来保证线程安全，当并非实现线程同步；ThreadLocal不是用来解决共享对象的多线程访问问题的，一般情况下，通过ThreadLocal.set()到线程中的对象是该线程自己使用的对象，其他线程是不需要访问的，也访问不到的**。**各个线程中访问的是不同的对象。

实现步骤：

①、建立一个类，并在其中封装一个静态的ThreadLocal变量，使其成为一个共享数据环境。

②、在类中实现访问静态ThreadLocal变量的静态方法（设值和取值）。

实例代码：

**class** Account5{

**private** **static** ThreadLocal<Integer> *tdll* =

**new** ThreadLocal<Integer>(){

**protected** Integer initialValue(){

**return** 2000;

}

};

**private** String name;

**public** **static** **int** getBalance(){

**return** *tdll*.get();

}

**public** **static** **void** setBalance(Integer value){

*tdll*.set(value);

}

**public** **static** **void** draw(**int** aimCash){

**if**(*tdll*.get()==0){

System.*out*.println("帐户余额为0！");

}**else** **if**(*tdll*.get()<aimCash){

System.*out*.println("帐户余额不足");

}**else**{

*tdll*.set(*tdll*.get()-aimCash);

System.*out*.println("取款金额："+aimCash);

System.*out*.println("帐户余额："+*tdll*.get());

}

}

}

其他部分无异同，在此不在赘述。

ThreadLocal并不能替代同步机制，两者面向的问题领域不同。同步机制是为了同步多个线程对相同资源的并发访问，是多个线程之间进行通信的有效方式；而ThreadLocal是隔离多个线程的数据共享，从根本上避免了多个线程之间共享资源，也就不需要对多个线程进行同步了。

通常我们认为：如果需要进行多个线程之间共享资源，以达到线程之间通信的功能，就使用同步机制，如果仅仅需要隔离多个线程之间共享冲突，可以使用ThreadLocal。