集合

集合的概念

Collection表示一组对象集合体,集合体是一类特殊情形的类区间,它是属于一个类的所有对象的集合体,当该类的一个对象被创建该对象可以被插入到类区间中, 并且一旦对象被删除也可以从类区间中移除,类区间允许类能像关系一样对待,从而检查类中所有的对象;
Collection函数库是java.util包下的一些接口和类，类是用来产生对象存放数据用的，而接口是访问数据的方式;
*Collection*函数*库*和数*组*的两点不同：

1. 数据的容量有限制的,而Collection库没有这样的限制，它的容量可以自动调节;
2. Collection函数库只能用来存放对象，而数组没有这样的限制; 如下是集合框架简图:
    集合框架
    集合框架

集合的类型

Collection接口：定义了存取一组对象的方法，其子接口Set和List分别定义了存储方式;
Set中的数据对象没有顺序且不可以重复;
List 中的数据对象有顺序且可以重复;
Map 接口定义了存储"键(key)-值(value)映射对的方法"

Collection接口

接口定义	**说明**
Boolean add(Object element)	增加元素
Boolean remove(Object element)	删除元素
Boolean contains(Object element)	包含
Boolean isEmpty()	判空
void clear()	集合的删除
Iterator iterator()	
Boolean containsAll(Collection c)	
Boolean addAll(Collection c)	
Boolean removeAll(Collection c)	
Boolean retainAll(Collection c)	
Object[] toArray()	

```
\\接口练习
```

注意点

1. Collection类对象在调用remove、contains等方法时需要比较对象是否相等,这会涉及到对象类型的equals方法和hashCode方法;对于自定义的类型,需要重写equals和hashCode方法以实现自定义的对象相等规则;
2. java中规定,两个内容相同的对象应该具有相等的HashCodes

Iterator接口

所有现实了Collection接口的容器类都有一个Iterator方法用以返回一个实现了Iterator接口的对象;
Iterator对象称作迭代器,用以方便的实现对容器内元素的遍历操作:

接口	接口说明
Boolean hasNext()	判断是否有元素没有返回
Object next()	返回游标当前位置的元素并将游标移动到下一个位置
Void remove()	删除游标左边的元素，在执行完next之后,操作只能执行一次

```
\\接口练习
```

Set接口

Set接口是Collection接口的子接口,Set接口没有提供额外的方法,Set接口的特性是容器类中的元素是没有顺序的,而且不可以重复;

```
\\接口练习
```

List接口

List 容器中的元素都对应一个整数型的序记载其在容器中的位置，可以根据序号存取容器中的元素; 增加了有关序的方法

接口	说明
Void add(Object element);	
Void add(int index, Object element);	
Object get(int index);	
Object set(int index, Object element);	修改某一位置的元素
Object remove(int index);	
Int indexOf(Object o);	如果没有该数据，返回-1

\\接口练习

List常用算法

java.util.collections提供了一些静态方法实现了基于List容器的一些常用算法

方法	说明
Void sort(List)	对List容器内的元素排序，排序的规则是按照升序进行排序
Void shuffle(List)	对List容器内的元素进行随机排列
Void reverse(List)	对List容器内的元素进行逆序排列
Void fill(List, Object)	用一个特定的对象重写整个List容器
Int binarySearch(List, Object)	对于顺序的List容器，采用折半查找方法查找特定对象

Comparable接口

根据上面的算法如何确定集合中对象“大小”顺序,就需要java.lang.Comparable接口

接	说明
Public int compareTo(Object obj)	返回0表示this=obj,返回正整数表示this > obj,返回负数表示 this < obj,实现了Comparable接口的类通过实现compareTo方法从而确定该类对象的排列方式

\\接口练习

注意点

Java中的Comparable和Comparator区别比较;
个性化比较:如果实现类没有实现Comparable接口，又想对两个类进行比较（或者实现类实现了Comparable接口，但是对compareTo方法内的比较算法不满意），那么可以实现Comparator接口，自定义一个比较器，写比较算法; 解耦:实现Comparable接口的方式比实现Comparator接口的耦合性要强一些，如果要修改比较算法，要修改Comparable接口的实现类，而实现Comparator的类是在外部进行比较的，不需要对实现类有任何修改。从这个角度说，其实有些不太好，尤其在我们将实现类的.class文件打成一个.jar文件提供给开发者使用的时候

Map接口

实现Map接口的类用来存储(key-value)对; Map接口的实现类有HashMap和TreeMap等; Map中存储的键值对通过键来标识,所以键值不能重复;

接口	说明
Object put(Object key,Object value);	

接口	说明
Object get(Object key);	
Object remove(Object key);	
Boolean containsKey(Object key);	
Boolean containsValue(Object value);	
Int size();	
Boolean isEmpty();	
Void putAll(Map t);	
Void clear();	

Map的Hash值说明

Hash(哈希)

散列函数的基本特性:根据同一散列函数计算出的散列值如果不同，那么输入值肯定也不同。但是，根据同一散列函数计算出的散列值如果相同，输入值不一定相同。 两个不同的输入值，根据同一散列函数计算出的散列值相同的现象叫做碰撞
常见的hash函数如下:

名称	说明
直接定址法	直接以关键字k或者k加上某个常数（k+c）作为哈希地址。
数字分析法	提取关键字中取值比较均匀的数字作为哈希地址。
除留余数法	用关键字k除以某个不大于哈希表长度m的数p，将所得余数作为哈希表地址。
分段叠加法	按照哈希表地址位数将关键字分成位数相等的几部分，其中最后一部分可以比较短。然后将这几部分相加，舍弃最高进位后的结果就是该关键字的哈希地址。
平方取中法	如果关键字各个部分分布都不均匀的话，可以先求出它的平方值，然后按照需求取中间的几位作为哈希地址。
伪随机数法	采用一个伪随机数当作哈希函数。

HashMap的数据结构

在Java中，保存数据有两种比较简单的数据结构：数组和链表。**数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易**上面我们提到过，常用的哈希函数的冲突解决办法中有一种方法叫做链地址法，其实就是将数组和链表组合在一起，发挥了两者的优势，我们可以将其理解为链表的数组。

HashMap和HashTable的对比

1. HashMap默认的初始化大小为16，之后每次扩充为原来的2倍。
2. HashTable默认的初始大小为11，之后每次扩充为原来的2n+1。
3. 当哈希表的大小为素数时，简单的取模哈希的结果会更加均匀，所以单从这一点上看，HashTable的哈希表大小选择，似乎更高明些。因为hash结果越分散效果越好。
4. 在取模计算时，如果模数是2的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法。所以从hash计算的效率上，又是HashMap更胜一筹。
5. 但是，HashMap为了提高效率使用位运算代替哈希，这又引入了哈希分布不均匀的问题，所以HashMap为解决这问题，又对hash算法做了一些改进，进行了扰动计算。

* 集合接口线程的安全性

分类	实现	线程安全	排序	特点
List	ArrayList	否	插入排序	随机访问性能高
List	LinkedList	否	插入排序	随机访问性能低,头尾操作性能高,不占用冗余空间
List	Vector	是	插入排序	并发性能不高,线程越多性能越差
List	CopyOnWriteArrayList	是	插入排序	并发性能高
Map	HashMap	否	无序	读写性能高
Map	LinkedHashMap	否	插入排序	可按插入顺序遍历,性能与HashMap接近
Map	HashTable	是	无序	并发性能不高,线程越多性能越差
Map	ConcurrentHashMap	是	无序	并发性能比HashTable高
Map	TreeMap	否	key升序或降序	有序,读写性能O(logN)
Map	ConcurrentSkipListMap	是	key升序或降序	线程安全,性能与并发数无关,内存空间占用较大
Set	HashSet	否	无序	同hashMap
Set	LinkedHashSet	否	插入排序	同LinkedHashMap
Set	TreeSet	否	对象升序或降序	同TreeMap

分类	实现	线程安全	排序	特点
Set	ConcurrentSkipListSet	否	无序	同ConcurrentSkipListMap
Queue	ConcurrentLinkedQueue	是	插入顺序	非阻塞
Queue	LinkedBlockingQueue	是	插入排序	阻塞,无界
Queue	ArrayBlockingQueue	是	插入排序	阻塞,有界
Queue	SynchronousQueue	是		不存储任何元素,向其中插入元素的线程会阻塞,直到有另一个线程将这个元素取走,反之亦然
Queue	PriorityQueue	否	对象自然序或者自定义排序	根据元素的优先级进行排序,保证自然序或自定义序最小的对象先出列
Deque	ConcurrentLinkedDeque	是	插入排序	非阻塞
Deque	LinkedBlockingDeque	是	插入顺序	阻塞,无序

* 集合内部实现算法的比较和场景说明

* 并发编程 volatile关键字和atomic包

并发编程三个概念

在并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题。我们先看具体看一下这三个概念：

1. 原子性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

一个很经典的例子就是银行账户转账问题：

比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。

试想一下，如果这2个操作不具备原子性，会造成什么样的后果。假如从账户A减去1000元之后，操作突然中止。然后又从B取出了500元，取出500元之后，再执行 往账户B加上1000元 的操作。这样就会导致账户A虽然减去了1000元，但是账户B没有收到这个转过来的1000元。

所以这2个操作必须要具备原子性才能保证不出现一些意外的问题。

2. 可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。
举个简单的例子，看下面这段代码：

```
//线程1执行的代码
int i = 0;
i = 10;

//线程2执行的代码
j = i;
```

假若执行线程1的是CPU1，执行线程2的是CPU2。由上面的分析可知，当线程1执行 i =10这句时，会先把i的初始值加载到CPU1的高速缓存中，然后赋值为10，那么在CPU1的高速缓存当中i的值变为10了，却没有立即写入到主存当中。

此时线程2执行 j = i，它会先去主存读取i的值并加载到CPU2的缓存当中，注意此时内存当中i的值还是0，那么就会使得j的值为0，而不是10。

这就是可见性问题，线程1对变量i修改了之后，线程2没有立即看到线程1修改的值。

3. 有序性

有序性：即程序执行的顺序按照代码的先后顺序执行。举个简单的例子，看下面这段代码：

```
int i = 0;
boolean flag = false;
i = 1;                //语句1
flag = true;          //语句2
```

从代码顺序上看，语句1是在语句2前面的，那么JVM在真正执行这段代码的时候会保证语句1一定会在语句2前面执行吗？不一定，为什么呢？这里可能会发生指令重排序（Instruction Reorder）。

一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

比如上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。

但是重排序也需要遵守一定规则：

1. 重排序操作不会对存在数据依赖关系的操作进行重排序。 比如：a=1;b=a; 这个指令序列，由于第二个操作依赖于第一个操作，所以在编译时和处理器运行时这两个操作不会被重排序。
2. 重排序是为了优化性能，但是不管怎么重排序，单线程下程序的执行结果不能被改变 比如：a=1;b=2;c=a+b这三个操作，第一步（a=1)和第二步(b=2)由于不存在数据依赖关系，所以可能会发生重排序，但是c=a+b这个操作是不会被重排序的，因为需要保证最终的结果一定是c=a+b=3。

volatile关键字

volatile是Java提供的一种轻量级的同步机制。同synchronized相比（synchronized通常称为重量级锁），volatile更轻量级。

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1. 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
2. 禁止进行指令重排序。

Atomic包

在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 自增（加1操作），自减（减1操作）、以及加法操作（加一个数），减法操作（减一个数）进行了封装，保证这些操作是原子性操作。atomic是利用CAS来实现原子性操作的（Compare And Swap）