笔记3.2

接笔记3的13章 集合

队列与双端队列

队列可以让人们有效的在尾部添加一个元素 在头部删除一个元素 有俩个端头的队列 即**双端队列** 可以让人们有效的在头部和尾部同时添加或删除元素
不支持在队列中间添加元素 在JAVA Se 6中引用了Deque接口 并由**ArrayDeque和LinkedList类**实现的 这俩个类都提供了双端队列 必要时可以增加队列的长度

优先级队列
优先级队列的元素可以按照任意的顺序插入 却总是按照排序的顺序进行检索 无论何时调用remove方法 都会删除掉当前优先级最小的元素

优先级队列并没有对所有的元素进行排序 如果用迭代来处理这些元素 是不需要对它们进行排序的 优先级使用的是一个**堆**的数据结构 这个是一个可以自我调整的二叉树 对树进行添加或删除操作 可以让**最小的元素移动到根** 而不必花时间对元素进行排序

与TreeSet一样 一个优先级队列即可以保存实现了Comparable接口的类对象 也可以保存在构造器中提供比较器的对象

每一个任务都有一个优先级 任务以随机顺序添加到队列中 每当**启动一个新的任务时** 都将**优先级最高**的任务从队列中**删除** （习惯是将1设为最高优先级 所以会将最小的元素删除）
下面是一个关于优先级队列的例子：

```
package com.jiheframework;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.PriorityQueue;

/**
 *优先级队列排列  这里的迭代不是按照顺序来进行访问的  删除总是删除剩余元素中优先级最小的那个
* Created by han on 2016/7/23.
 */
public class ProrityQueueTest {

public static void main(String[] args) {
        PriorityQueue<GregorianCalendar> pq = new PriorityQueue<>();
pq.add(new GregorianCalendar(1906, Calendar.DECEMBER, 9));
pq.add(new GregorianCalendar(1815, Calendar.DECEMBER, 10));
pq.add(new GregorianCalendar(1903, Calendar.DECEMBER, 3));
pq.add(new GregorianCalendar(1910, Calendar.DECEMBER, 22));

System.out.println("Iterating over elements....");
        for (GregorianCalendar date:pq)
            System.out.println(date.get(Calendar.YEAR));

System.out.println("Removing elements");

        while (!pq.isEmpty()){
            System.out.println(pq.remove().get(Calendar.YEAR));
}

    }
}
```

映射表
映射表用来存放键值对
通用有俩个实现 一个是**HashMap** 一个是**TreeMap** 这俩个类都实现了Map接口
散列表会对键进行散列 树映射表用键的整体顺序对元素进行排序 并组织成了搜索树
散列或比较函数**只能作用于键** 与键关键的值不能进行散列或比较
至于选择的是散列映射表集合还是树映射表集 这也是看情况的 一般不需要排序顺序访问 就散列

向映射表添加对象 必须同时提供键

键必须是唯一的 不能对同一个键存放俩个值 如果对同一天俩次调用put方法 第二个值就会取代第一个值

集合框架并没有将映射表本身视为一个集合 然而 可以获得映射表的视图 这是一组实现了Collection接口对象 或者它的子接口的视图

有三个视图 分别是：**键值 值集合（不是集）和键/值对集** 键与键/值对形成了一个集 映射表中一个键只能对应一个值

分为三个 Set<K> keyset Collection<K> values () Set<Map.Entry<k,v>> entrySet()

列如keyset方法
遍历下面的枚举映射表的所有键 如果想同时查看键与值就使用entrySet方法 在这里不写了

```
Set<String> keys = map.keySet();
 for (String key :keys){
     System.out.println(keys);
}
```

专用集与映射表类

弱散列映射表 WeakHashMap类
如果有一个值 如果有一个值 对应的键不再使用了

就会出现尴尬的情况 但是垃圾回收器不能删除它

至于为什么垃圾回收器不能进行回收的原因是：
垃圾回收器跟踪**活动的对象**
但是映射表对象是活动的 其中所有的桶也是活动的 所以会导致 它们不能被回收
当键的唯一引用来自散列表条目时 这一数据结构将于垃圾回收器协同工作一起删除键/值对

WeakHashMap 使用弱引用保存键 WeakHashMap对象**将引用保存到另外一个对象中** 在这里 就是散列表键
通常 **如果某个对象只能被WeakHashMap引用的话 垃圾就回收器就会回收它** 但要将引用这个对象的弱引用放入队列中
WeakHashMap定时查看是否有新添加的弱引用 一个弱引用进入队列意味这个键不再被他人使用 并且已经被收集起来 于是
WeakHashMap将删除对应的条目

链接散列集和链接映射表

LinkedHashSet 和LinkHashMap 用来记住插入元素项的顺序

链接散列映射表将用**访问顺序 而不是插入顺序** 每次调用get或put受到影响的条目将从当前的位置删除 并放到条目链表的尾部 只有条目在链表的位置受到影响 而散列表中的桶不会受到影响 一个条目总位于键散列码对应的桶中

```
package com.jiheframework;

import com.fanxin.Employee;

import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;

/**
 * Created by han on 2016/7/24.
 */
public class LinkedTest {


public static void main(String[] args) {
        Map staff = new LinkedHashMap<>();
staff.put("144-25-5464", new Employee("Amy  lee"));
staff.put("567-24-2546", new Employee("Harry Hacker"));
staff.put("157-62-7935", new Employee("Gary Cooper"));
staff.put("456-62-5527", new Employee("Francesca Gruz"));
```

```
Iterator iterator = staff.keySet().iterator();
        for (int i = 0; i < 4; i++) {
            System.out.println(iterator.next());
}

        Iterator iterator1=staff.values().iterator();
        for (int i = 0; i <4 ; i++) {
            System.out.println(iterator1.next());
}

    }
}
```

访问顺序对于实现高速缓存的 “最近最少使用” 原则十分重要 在某些情况下：比如在需要访问的时候每次都将前面几个元素删掉 留下后面最少被访问的元素 就是一个应用
我们可以让这一过程自动化 即覆盖LinkedHashMap的一个方法 构造这个 LinkedHashMap的子类

```
public class LinkedHashMapExtendsTest extends LinkedHashMap {

@Override
protected boolean removeEldestEntry(Map.Entry eldest) {
return super.removeEldestEntry(eldest);
}
}
```

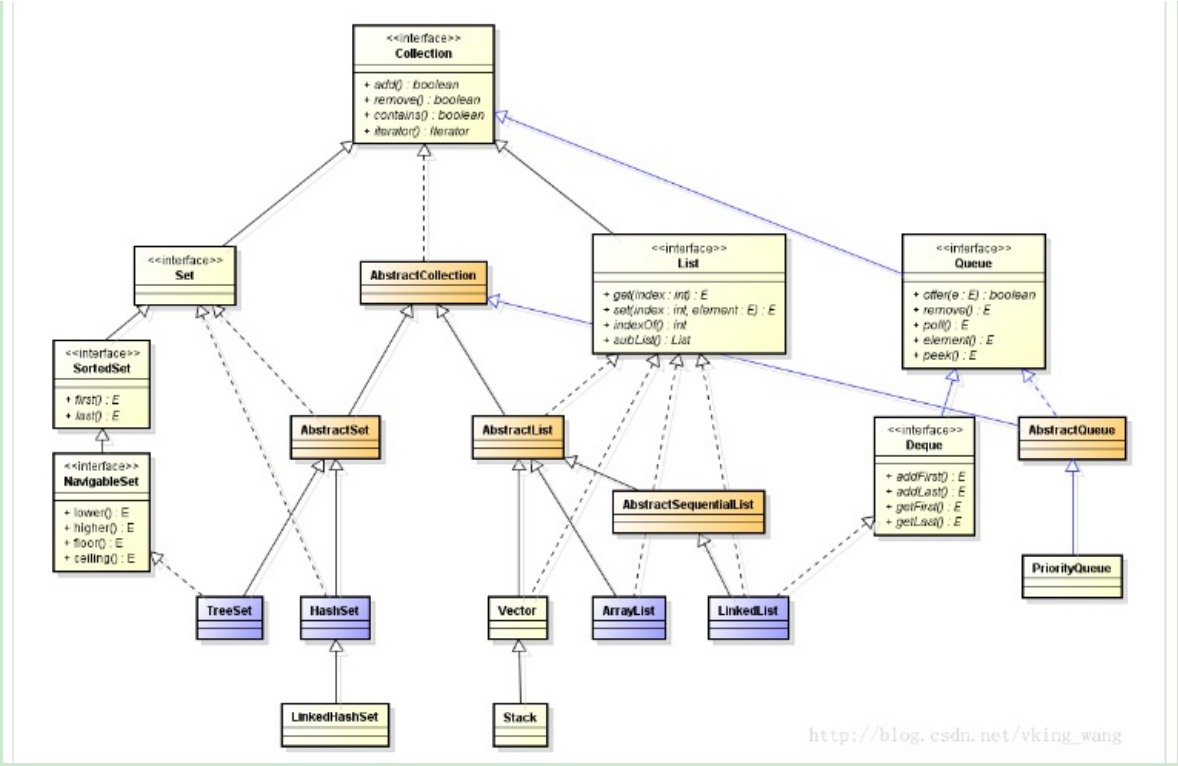
枚举集与映射表
EnumSet是一个枚举类型的元素集的高效实现
EnumSet是没有公共的构造器的 可以通过静态工厂方法构造这个集

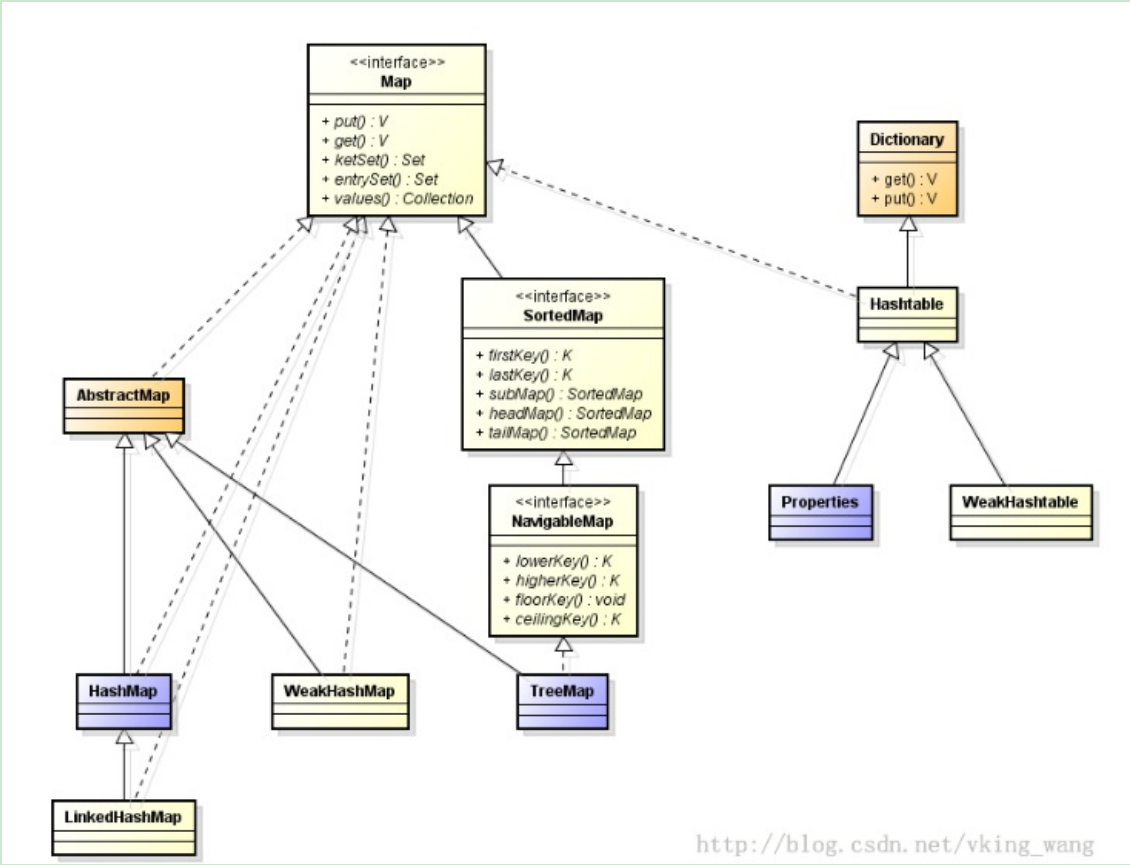
```
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> nerver = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDY, Weekday.FRIDAY);
EnumSet<Weekday> mvf = EnumSet.of(Weekday.MONDY,Weekday.WEDNESDAY,Weekday.FRIDAY);
```

可以使用set接口的方法来修改EnumSet
这个EnumSet是一个**键为枚举类型的映射表** 它可以直接并且高效的用一个值数组实现 在使用时 需要在构造器中指定键类型

标识散列映射表
IdentityHashMap 在这个类中 键的**散列值**不是由HashCode算的 **而是由System.identityHashCode方法**（这是Object.hashCode根据对象的内存地址进行计算等散列码的使用方式 对象比较时 采用的是 == 不是equals）计算的
不同的键对象 即使内容相同 也被视为不同的对象 **在实现对象遍历算法（对象序列化）时 这个类非常有用 可以用来跟踪每个对象的遍历状况**

集合框架
集合是框架是一个类的集合 是创造高级功能的基础 框架使用者创建的子类可以扩展为超类的功能 而不必重新创建这些基本的机制
java集合类库定义了集合类的框架 为集合的实现者定义了大量的接口与抽象类





有俩个基本接口 一个是Collection 一个是Map
add方法是Collection的插入元素的方法
Map因为是键值对 所以是PUT
可以在映射表中找到键对应的值 采用get方法来获取

List是一个有序集合 元素可以添加到容器某个特定的位置 添加到特定的方式有俩种 一种是使用整数索引 一种是使用列表迭代器 具体方法查API

javaSE 1.4为了检测一个特定的集合**是否支持高效的随机访问** 加入了一个接口 RandomAccess 这个接口是没有任何方法的

ListIterator 接口定义了一个方法 用于将元素添加到迭代器所处的位置的前面 add方法

Set接口与Collection接口是一样 只不过有着更严格的定义（建立Set接口后 可以让程序员编写仅接受集的方法）
Set 集合是无序不可以重复的的、**List** 集合是有序可以重复的。
Set：集
List：列表
Map：集合
最后 JAVASE 6引入了接口NavigableSet和NavigableMap 其中包含了几个**用于有序集合映射表中查找和变量的方法**（从理论上讲 这几个方法已经包含在SortedSet和SortMap中了）

前面讲过 **集合接口本身是有大量的方法的 这些方法我们可以通过更基本的方法实现 抽象类提供了许多这样的实现：**
AbstractCollection
AbstractList
AbstractSequentiallist
AbstractSet
AbstractQueue
AbstractMap

如果写了自己的集合类 就可能扩展上面的某个类 以便可以选择例行操作的实现

JAVA类库支持以下的具体类实现：
Linkedlist
ArrayList
ArrayDeque
HashSet
TreeSet
PriorityQueue
HashMap
TreeMap

还有很多在JAVA第一版遗留下来的容器 在集合框架之前就有了
是：
Vector
Stack
Hashtable
Properties
这些类现在已经被包含到集合框架中了

视图和包装器

通过使用视图可以获得其他的实现了接口和映射表接口的对象
比如KeySet方法

轻量集包装器
Arrays类的静态方法aslist返回一个包装了普通Java数组的List包装器

```
Card[] cardDeck = new Card[52];
List<Card> cardList = Arrays.asList(cardDeck);
```

在这里的asList返回的是一个视图对象 不是ArrayList 使用改变数组大小的所有方法 都会出现异常
从JAVASE 5 开始
asList方法声明了一个具有可变参数的方法

```
List list=Arrays.asList("AMY", "CESHI", "CRAL");
```

```
List<String> list1 = Collections.nCopies(100, "DEFAULT");
```

这个方法创建了100个字符串的list 每个串都被设置为"DEFAULT" 由于只存储了一次 存储代价小 这样的确可以在某些特殊的情况下省内存资源 返回了一个实现list接口的**不可修改**的对象

下面这个方法返回的是一个实现了set接口的视图对象 返回的是一个**不可修改的单元素集** 而不需要付出数据结构的开销

```
Collections.singleton(Object);
```

子范围

子范围就是在集合中截取一段子范围 我不想抄书了
就下面几个示例演示一下

```
List group2 = staff2.subList(10, 20);/*截取10 -19的元素*/
group2.clear();/*可以将操作放到整个子范围 比如：这个 清除该子范围*/
```

对于有序集和映射表 **可以使用排序规则而不是元素位置建立子范围**
有序映射表也有类似的方法
SortedSet<E> subSet(E from ,E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)
返回映射表视图
还比如在NavigableSet下也有 这个是带指定是否包括边界的 具体

不可修改的视图
Collections注意 跟Collection接口是没有关系的
http://blog.csdn.net/pacosonswjtu/article/details/50333509 这是关于集合的视图与框架的具体部分的讲解（抄书）

关于细节看书
可以使用下列6种方法获得不可修改视图：

```
1  Collections.unmodifiableCollection
2  Collections.unmodifiableList
3  Collections.unmodifiableSet
4  Collections.unmodifiableSortedSet
5  Collections.unmodifiableMap
6  Collections.unmodifiableSortedMap
```

每个方法都定义于一个接口。如， Collections.unmodifiableList 与 ArrayList、LinkedList 或者任何实现了 List接口的其他类一起协同工作；

不可修改视图并不是 集合本身不可修改

由于视图只是包装了 接口而不是 实际的集合对象， 所以只能访问 接口中定义的方法

- **W1）** unmodifiableCollection 方法将返回一个集合， 它的equals 方法不调用底层集合的equals 方法； 相反，它继承了 Object 类的 equals 方法， 该方法只是检测两个对象是否是同一个对象；
- **W2）**如果将 集 或 列表转换成集合， 就再也无法检测其内容是否相同了， 视图就是以这种方式运行的， 因为内容是否相等的检测在分层结构的这一层上没有定义妥当；
- **W3）**视图将以同样的方式处理 hashCode 方法；

同步视图

如果由**多个线程访问集合**，就必须确保集不会被意外破坏， 类库的设计者**使用视图及直接来确保常规集合的线程安全**。
例如， Collections 类的 静态 synchronizedMap 方法将任何一个映射表转换成具有同步访问方法的 Map：

```
1  Map<String, Employee> map = Collections.synchronizedMap(new HashMap<String, Employee>());
```

```
Map<String, Item> map = Collections.synchronizedMap(new HashMap<String, Item>());
```

检查视图
检测视图如下： 下面定义了一个安全列表

```
List<String> strings = new ArrayList<>();
List<String> safeStrings = Collections.checkedList(strings, String.class);
```

这样就可以在我们需要的时候正确的位置进行报告异常

关于API文档的可选操作方法是个不建议使用的东西 如果需要知道为什么就看书 P602

关于批操作
交集的使用
交集的使用主要是涉及到一个**retainAll**方法

```
List list = new ArrayList<>();
List listb = new ArrayList<>();
Set<String> result = new HashSet<>(list);
result.retainAll(listb);
```

关于交集之后进行删除的处理

这是列子：

```
Map<String,Employee>staffMap。。。。。
Set<String>terminatedIDs。。。。。。
staffMap.keySet().removeAll(terminatedIDs);
```

集合与数组进行的转换

关于数组向集合的转换只需要使用Arrays.asList 就可以了

关于集合向数组转换的话就会有一点问题
使用的toArray方法
但是这个方法有所缺陷 只返回的是Object的数组

```
Object[] values = sac.toArray();
```

我们如果想要返回跟创建的数组一样的话 就使用下面的方法

```
String[] s=  strings.toArray(new String[0]);
String[] s=  strings.toArray(new String[10]);
String[] s=  strings.toArray(new String[strings.size()]);
```

集合中的算法

下面是其中一个列子 求出最大的值算法

```
public  static  <T extends Comparable> T  max(Collection  c)

{

if (c.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = c.iterator();
T largest = iter.next();
  while (iter.hasNext()) {
T next = iter.next();
    if (largest.compareTo(next)< 0)
      largest = next;
}
return largest;
}
```

排序与混排

Collections类中的sort方法可以对List接口的集合进行排序
下面这个方法假定列表元素实现了Comparable接口 如果想采用其他方式对列表进行排序 可以将Comparator对象作为第二个参数传递给Sort方法

```
List<String>staff=new LinkedList();
Collections.sort(staff);
```

如果想按照降序进行排序 可以用一种非常方便的静态方法进行Collection.reverseOrder（） 这个方法将返回一个比较器 比较器返回b.compareTo(a)

```
ArrayList sac = (ArrayList) strings;
Collections.sort(sac,Collections.reverseOrder());
```

关于**sort方法排序**的内部实现

它直接是将所有元素转入一个数组 并使用一种**归并排序**的变体对数组进行排序 然后 再降排序后的序列复制回列表

集合类库中使用的归并排序算法比快速排序的算法慢一点 但是 归并排序算法有一个优点 就是**稳定** 即 不要交换相同的元素

比如：如果有一张表 是员工表 按照工资排序 如果工资相同的话 就按照姓名排序

什么时候列表才是可更改的？
如果列表支持set方法 则是可更改的
如果列表支持add和remove方法 则是可改变方法
Collections类有一个算法shuffle 其功能与排序相反 即随机地混合列表中元素的顺序

```
ArrayList<Card> cards = new ArrayList<>();
Collections.shuffle(cards);
```

二分查找
要在数组数组中查找一个对象 通常要依次访问数组的每个元素 直到找到匹配的元素为止 然后 如果数组是有序的 就可以直接查看位于数组中间的元素 看一看是否大于要查找的元素 如果是 就从前半部分查找 否则 就是从后半部分查找

Collections类的binarySeach方法实现了这个方法 （集合必须是排好序的 否则返回的是错误的答案）

首先 想要查找元素 必须提供集合以及查找的元素

注意 **只有采用随机访问 二分查找才有意义 如果为bingarySeach算法提供一个链表 就变成线性查找**

binarySearch

public static <T> int **binarySearch**(List<? extends Comparable<? super T>> list,
T key)

使用二分搜索法搜索指定列表，以获得指定对象。在进行此调用之前，必须根据列表元素的自然顺序对列表进行升序排序（通过 sort(List) 方法）。如果没有对列表进行排序，则结果是不确定的。如果列表包含多个等于指定对象的元素，则无法保证找到的是哪一个。

此方法对“随机访问”列表运行 log(n) 次（它提供接近固定时间的位置访问）。如果指定列表没有实现 RandomAccess 接口并且是一个大型列表，则此方法将执行基于迭代器的二分搜索，执行 O(n) 次链接遍历和 O(log n) 次元素比较。

参数：
list - 要搜索的列表。
key - 要搜索的键。
返回：
如果搜索键包含在列表中，则返回搜索键的索引；否则返回 (-(*插入点*) - 1)。*插入点* 被定义为将键插入列表的那一点：即第一个大于此键的元素索引；如果列表中的所有元素都小于指定的键，则为 list.size()。注意，这保证了当且仅当此键被找到时，返回的值将 >= 0。
抛出：
ClassCastException - 如果列表中包含不*可相互比较* 的元素（例如，字符串和整数），或者搜索键无法与列表的元素进行相互比较。

i=Collections.binarySeach(c,element);
如果集合没有采用Comparable接口的compareTo方法 就还要提供一个比较器对象
i=Collections.binarySeach(c,element,**comparator**)
插入点 是 = -i -1 ;
if(i <0){

c.add(-i -1 ,element);
} 这样就能将元素插入到正确的位置上
如果binarySeach方法返回的数值大于等于0 则表示匹配的对象的索引 也就是说 c.get(i)等于在这个比较顺序下的element
如果返回负值 则表示没有匹配的元素 但是 可以利用返回值计算将element插入到集合的哪个位置 以保持集合的有序性 插入的位置是 -i -1

只有采用随机访问 二分查找才有意义

```
int i;
i=Collections.binarySearch(list,"ce");
  if(i <0){
    list.add(-i-1,"ce");
}
```

binarySeach方法**检测列表参数是否实现了RandomAccess接口** 如果实现了这个接口 这个方法将采用二分法查找 否则 就是线性查找

从有序列表搜索一个键 如果元素**扩展了AbstractSequentialList类** 则采用线性查找 否则就是二分查找 **这个方法的时间复杂度O（a(n) log n）n是列表的长度 a(n)是访问一个元素的平均时间** 这个方法将返回这个键在列表中的索引 如果在列表中不存在这个键将返回负值 应该将这个将插入到列表索引 -i -1 以保持列表的有序性

简单算法
Collections类 包含了不少简单很有用的算法 为什么会在标准库提供这些简单的算法？ **它们可以让程序员阅读算法变成一件轻松的事**
比如Collections.max

编写自己的算法

注意：要用面向接口编程 要在参数上使用最通用的类型 也就是Collection<T>能够访问所有的元素
如果使用面向接口编程的话 就可以在日后方便改变想法 并用另一个集合实现这个方法

遗留的集合
1 Hashtable类
Hashtable与HashMap类的作用是一样的 拥有相同的接口 Hashtable与Vector类都是同步的 如果对同步与遗留代码的兼容性没有要求的话 就应该使用HashMap

2 枚举
Enumeration接口对元素序列进行遍历 Enumeration 接口有俩个方法 即basMoreElements和nextElement 这俩个方法与Iterator接口的hasnext和next都比较相似

```
Hashtable<String, String> staff = new Hashtable();
Enumeration<String> enumeration = staff.elements();
while (enumeration.hasMoreElements()) {
    String e = enumeration.nextElement();
}
```

在有的时候 会遇到遗留的代码 其参数是枚举类型的 静态方法 Collections.enumeration将产生一个枚举对象 枚举集合中的元素

```
List<InputStream> streams = new ArrayList<>();
SequenceInputStream in = new SequenceInputStream(Collections.enumeration(streams));
```

属性映射表（property map）是一个非常特殊的映射表
这就是很典型的.properties文件 就是采用这种类型
键与值都是一个字符串
表可以保存到一个文件中 也可以从文件中加载
使用一个默认的辅助表

位集
java平台的BitSet类用于存放一个位序列 如果需要高效地存储位序列比如：标识 就可以使用位集 由于将位集包装在字节里 所以 使用

位集比使用Boolean对象的Arraylist要更加高效

BitSet类提供了一些便于读取 设置 或清除各个位的接口

到此第十三章结束 如果有出错的地方 请看原书