笔记3

第 13章 集合

集合的接口与实现分离 **队列接口** 指出可以在队列的尾部添加元素 按照先进先出的原则检索 对象时就应该使用队列 一种使用循环数组 一种是使用链表

这下面的代码就是一个队列的代码实现

```
public interface Queue<E>  extends Collection<E>{

void add(E element);
E remove();
    int Size();


}
```

如果需要一个循环数组队列 就可以使用**ArrayDeque类**

如果需要一个链表队列 就直接使用**LinkedList类** 这个类实现了Queue接口

循环数组是一个有界集合 即容量有限 如果程序中要收集的对象**没有上限** 就最好用**链表**来进行实现

一旦构建集合了集合就不需要知道究竟使用了哪种实现 因此 只有在构建集合对象时 使用具体的类才有意义

```
Collection<String> c=new ArrayList<String>();
```

集合的类的基本接口是Cllection类

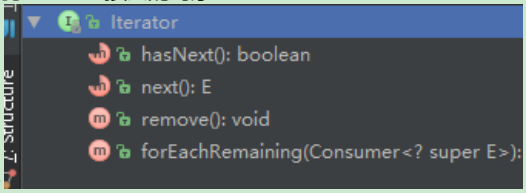
这个Collection有俩个基本方法

```
boolean add(E e);
```

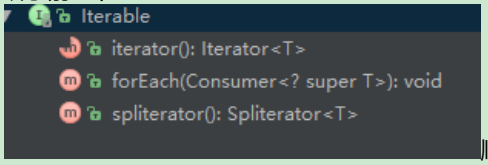
```
Iterator<E> iterator();
```

add是向集合中添加元素

Iterator方法用于返回一个实现了Iterator接口的对象

在Iterator接口中包含3个方法 

其实是4个 但是第4个应该是新的JDK加上的 这里就讲了前三个



顺便一提 Iterable是三个方法

for each 循环可以与任何实现了Iterable接口的对象一起工作 Iterable这本书这里又说只有一个iterator方法 但是其实已经有三个了 collection接口扩展了Iterable接口 因此 所有的集合都可以使用foreach循环

元素访问的顺序取决于集合的类型 在Arraylist中 遍历迭代器与HashSet遍历时不一样的 一个是顺序的取值 一个是随机的取值 虽然 迭代器都能取出所有的值

java中 在迭代器中是查找元素与位置变更是紧密相连的 **查找一个元素的唯一方法是调用next** 而在执行查找操作的同时 迭代器的位置随之向前移动

java的迭代器是位于俩个元素之间的 当调用next时 迭代器就越过下一个元素 并返回刚刚越过的那个元素的引用

如果想**删除指定位置的元素 仍然需要越过**这个元素

列如：
Iterator<String>it=c.iterator();
it.next()
it.remove();

对next方法与remove方法是具有相互依赖性的 如果在调用remove方法之前没有调用next方法就是会出错的 会抛出异常

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

    Collection<String> c=new ArrayList<String>();
c.add("c");
c.add("s");
Iterator<String> it =c.iterator() ;
it.next();
it.remove();
       for (String c2:c){
           System.out.println(c2);
}
}
```

这是具体的写法
当然如果
想像下面这样删除俩个相邻 的元素也是非法的 必须调用next越过要删除的元素

```
   it.remove();
it.remove();
```

由于Collection与Iterator都是泛型接口 **可以操作任何类型**的的实用方法

Collection接口中声明了很多有用的的方法 所有的实现类都必须提供这些方法
当然 如果实现Collection接口的每一个类都要提供如此多的例行方法将是一件很烦人的事情 为了能够实现者**更容易的实现这个接口 Java类库提供了一个抽象类AbstractCollection** 它将基础方法size与iterator抽象化了 但是提供了例行方法

这样的话 可以由**具体的我们设置的集合类提供iterator方法 而contians方法已经由AbstractCollection超类提供了** 然而 如果子类有更加有效的方式实现contains方法 也可以由子类提供 就这点而言 没有任何限制

集合类的用户可以使用泛型接口中一组更加丰富的方法 而实际数据结构实现者并没有需要实现所有例行方法的负担

链表
数组和数组列表都有一个重大的缺陷 这就是从数组的中间位置删除一个元素要付出很大的代价 其原因是数组中处于被删除元素之后的所有元素都要向数组的前端移动

链表就能解决这个问题 在java中 ：
所有链表实际上都是双向链表的 即每个阶段还存放着指向前驱结点的引用
java就提供了一个LinkedList类
这是

`List` 接口的链接列表实现

```
List<String> staff=new LinkedList<String>();
staff.add("amy");
staff.add("bob");
staff.add("Caral");
Iterator<String> iterator=staff.iterator();
String first=iterator.next();
```

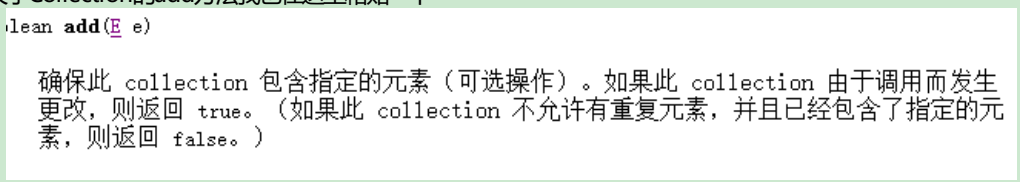
```
String second=iterator.next();
iterator.remove();
```

上面展现了一些关于LinkedList基本用法

链表和泛型集合之间有一个重要的区别 链表是一个**有序集合 每个对象的位置十分重要**
LinkedList.add方法是将每个元素添加到链表的尾部 但是 常常需要将元素添加到链表的中间
由于迭代器是描述集合中位置的 所以这种依赖于位置的add方法将由迭代器负责

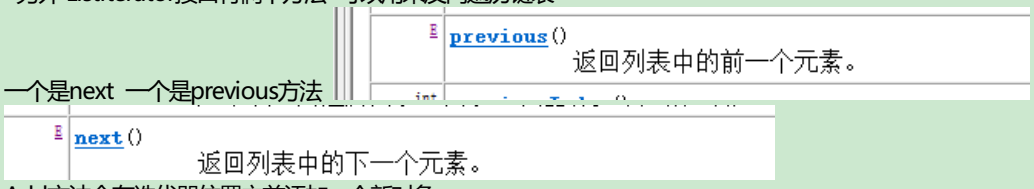
只有对自然有序的集合使用迭代器添加元素才有实际意义
而无序的类型set类型 其中的元素是无序的
因此 在Iterator接口中没有add方法 相反的 集合类库提供了子接口ListIterator 其中包含了add方法

与Collection.add不同 这个方法不返回boolean类型的值 它假定添加操作总会改变链表
关于Collection的add方法我也在这里粘贴一下



另外 ListIterator接口有俩个方法 可以用来反向遍历链表

一个是next 一个是previous方法



Add方法会在迭代器位置之前添加一个新对象
列如：下面的代码将越过链表中的第一个元素 并在第二个元素之前添加Juliet

```
List<String> staff=new LinkedList<>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter=staff.listIterator();
iter.next();
iter.add("Juliet");
```

当迭代器越过链表的最后一个元素时 添加的元素将变成列表的新队尾 如果链表有n个元素 有n+1个位置可以添加新元素

在使用**“光标”**类比时要格外小心 **add方法只依赖迭代器的位置 而remove方法依赖于迭代器的状态**

set方法用一个新元素取代调用next 或previous方法返回的上一个元素
列如 下面的代码将用一个新值取代链表的第一个元素

```
List<String> list=new LinkedList<String>();
ListIterator<String> iter=list.listIterator();
String newValue="ceshi";
String oldValue=iter.next();
iter.set(newValue);
```

在代码中如果有俩个迭代器 其中**一个迭代器进行修改集合** 另外**一个迭代器进行遍历集合 一定会出现混乱的情况**
列如：一个迭代器指向另一个迭代器刚刚删除的元素前面，现在这个迭代器是无效的 并且不应该再使用

```
List<String> list=new LinkedList<String>();
ListIterator<String> iter1=list.listIterator();
ListIterator<String> iter2=list.listIterator();
iter1.next();
iter1.remove();
iter2.next();//ERROR
```

如下代码就会抛出一个异常 Concurrent ModificationException
这是迭代器本身的设计使它能够检测到这种修改
如果迭代器发现它的集合被另一个迭代器修改了 或是该集合自身的方法修改了 就会抛出一个Concurrent ModificationException异常

为了避免发生并发修改的异常 遵守以下原则： 可以根据需要给容器附加许多的迭代器 但是这些迭代器只能读取列表 另外 再单独附加一个可以读又可以写的迭代器

有一种简单的办法可以检测并发修改的问题 集合可以跟踪改写操作的次数 每个迭代器都维护一个独立的计数值 在每个迭代器 方法的开始处都检查自己改写的操作的计数值是否与集合的改写操作计数值一致 如果不一致 就抛出一个 Concurrent ModificationException

对于并发修改列表的检测有一个奇怪的例外
链表只负责跟踪对列表的结构性修改 列如 添加元素 删除元素。**而set方法不被视为结构性修改 可以将多个迭代器附加给一个链表 所有的迭代器都调用 set方法对现有结点的内容进行修改**

可以使用ListIterator类从前后俩个方向遍历链表的元素 并可以添加 删除元素

链表不支持快速访问 如果想查看链表的第n个元素的话 就必须从头开始 越过n-1个元素
所以说 在程序需要采用整数索引访问元素时 程序员通常不选用链表

在LinkedList类中提供了一个用来访问特定元素的方法get

```
LinkedList<String> list=new LinkedList<String>();
list.add("ce");
list.add("s");
list.get(1);
```

这个方法的效率低下

尤其非常需要注意到的是不可以采用下面的方法来进行对链表来进行取值 效率真的是极其低下

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

每次查找一个元素都是从头部重新开始搜索的
get方法做了一点小小的优化 一旦是总长度/2的后面的 就从尾部开始搜索（二分查找）

列表迭代接口有一个方法可以告诉当前位置的索引 java迭代器指向俩个元素之间的位置 所以可以同时产生俩个索引 nextIndex方法返回下一个调用next方法时的整数索引 previousIndex方法返回下一次调用previous方法时返回调用元素的整数索引
这俩个方法效率非常高 这是因为迭代器保持着当前位置的计数值 最后需要说明一下 如果有一个N list.listIterator(N)将返回一个迭代器 这个迭代器指向索引为N的元素的前面 也就是说 调用next与list.get(N)会产生同一个元素 只是获得这个迭代器的效率比较低

如果链表只有很少的几个元素 就完全没有必要为get和set方法的开销烦恼 但是 需要注意的是
使用链表的**唯一理由是尽可能地减少在列表中间插入或删除元素所付出的代价** 如果列表只有少数几个元素 就完全可以使用Arraylist
建议 避免使用以整数索引表示链表中位置的所有方法 如果需要对集合进行随机访问 就使用数组或Arraylist 而不要使用链表

下面有一个关于俩个迭代器的列子

```
import java.util.*;
import java.util.ListIterator;

/**
 * 这是一个关于迭代器具体使用的列子
* 将俩个迭代器合并在一起
* 然后从二个链表中每间隔一个元素删除一个元素  最后测试removeAll方法
* Created by han on 2016/7/22.
 */
public class Test04 {

public static void main(String[] args) {
List<String> a = new LinkedList<>();
a.add("Amy");
a.add("Carl");
a.add("Eruca");

List<String> b = new LinkedList<>();
b.add("Bob");
b.add("Doug");
b.add("Frances");
b.add("Gloria");
```

```
ListIterator<String> aIter = a.listIterator();
Iterator<String> bIter = b.iterator();

while (bIter.hasNext()) {
if (aIter.hasNext()) aIter.next();
aIter.add(bIter.next());//这里的Biter进行了next
}

System.out.println(a);
bIter = b.iterator();//所以这里需要重新声明 位置归零
while (bIter.hasNext()) {
bIter.next();
if (bIter.hasNext()) {
bIter.next();
bIter.remove();
            }

        }
System.out.println(b);


a.removeAll(b);

System.out.println(a);
    }

}
```

总结 上面介绍了List接口与这个接口的LinkedList类 List接口用于描述一个有序集合 并且集合每个元素的位置十分重要 有俩种访问元素的协议 一种是迭代器 另外一种是get 和set方法随机访问每个元素 后者不适合用于链表 但是对数组有用 集合类库提供了一个大家熟悉的ArrayList类 这个类也实现了List接口 **ArrayList封装了一个动态再分配的对象数组**
注释：Vector类 与Arraylist类的使用区别 是否线程安全 但是Vector同步需要花费时间多 不需要同步使用的时候建议使用Arraylist

链表和数组可以按照人们的意愿排列元素的次序 如果想要查看某个指定的元素 却又忘记了它的位置 就需要访问所有的元素 但是这样会消耗很多时间 可以有几种能够快速查找元素的数据结构 但是无法控制元素出现的次序 它们安装**有利于其操作目的的原则组织数据**

有一种很好的数据结构 可以**方便的进行查询所需要的对象** 这就是**散列表**
散列表为每个对象计算一个整数 称为**散列码（hash code）** 散列码（是由hashcode方法得到的）是由对象的实例域产生的一个整数（**不同的数据域的对象将产生的是不同的散列码**）
如果自定义类 就需要**注意俩个方法** 一个是hashcode 一个是equals方法 这俩个方法按理说是**需要重写**的（如果是自定义类的话 而且这俩个方法必须要兼容）
注意这里的hashcode方法应该与equals方法兼容
散列表是由**链表数组实现的** 每个列表被称为**桶**

如果想**查找表中对象**的位置的话 很简单
先计算它的散列码 然后与桶的总数取余 所得到的结果就是保存这个桶的索引
列如：如果某个对象的散列码是76268 有128个桶 对应的对象应该保存在108号桶（76268/128）
在这个桶中没有元素的话 就直接插入 如果桶被占满了 这就叫**散列冲突** 需要插入的新对象对桶中的对象进行比较 来看这个元素是否已经存在
如果散列码合理 桶的数目比较大 就会比较的次数比较少

如果想更多的控制散列表的运行性能 就**直接自己指定一个初始的桶数**
桶数是指用于收集具有**相同散列值**的数目
如果要插入到散列表的元素太多 就会增加冲突的可能 降低运行性能

如果大致知道有多少元素插入散列表中 就可以设置桶数 将桶数设置为预计元素的 75% --100%
而且最好将桶数设计为素数 以防键的集聚

如果散列表太满 就需要**再散列** 如果对散列表进行再散列 就需要创建一个桶数更多的表 并将所有的元素插入这个新表中 然后丢弃原来的表 **装填因子** 决定何时对散列表进行再散列
对于大多数应用程序来说 装填因子为0.75是比较合理的（0.75的装填因子就是到表中 一但表中位置超过75%被填入就进行双倍的叠加桶数然后自动进行再散列）

在散列表中可以实现几个重要的数据结构 其中最简单的是**set类型** set是**没有重复的元素的元素集合**（set是很重要的集合接口）
set的add方法首先在集中**查找**添加的对象 如果**不存在** 就将这个对象添加进去

在java类库集合中提供一个类 这就是**HashSet** 这个是基于散列表的集 其中的contains方法已经被重新定制 **快速查找某个元素是否已经**

出现在集合中 它只在某个桶查找元素 而不必查看集合中的所有元素
散列集迭代器是将依次访问 所有的桶 但是这个是**散落在表的各个地方的** 访问的**顺序随机** 只有对集合中元素的顺序不关心的时候才使用
Hashset

implements Set<E>, Cloneable, Serializable

此类实现 Set 接口，由哈希表（实际上是一个 HashMap 实例）支持。它不保证 set 的迭代顺序；特别是它不保证该顺序恒久不变。此类允许使用 null 元素。

此类为基本操作提供了稳定性能，这些基本操作包括 add、remove、contains 和 size，假定哈希函数将这些元素正确地分布在桶中。对此 set 进行迭代所需的时间与 HashSet 实例的大小（元素的数量）和底层 HashMap 实例（桶的数量）的“容量”的和成比例。因此，如果迭代性能很重要，则不要将初始容量设置得太高（或将加载因子设置得太低）。

这里是从System.in来读取单词 添加到集合 然后再打印出来

需要在shell下（win中的cmd）运行 java < ceshi.txt 就可以读取输入的单词与 并且将他们添加到散列集中 列子：单词以随机的顺序出现 原因在前面有说

```
importjava.util.HashSet;
importjava.util.Iterator;
importjava.util.Scanner;
importjava.util.Set;

/**
 * 这里是从System.in来读取单词  添加到集合 然后再打印出来 遍历散列集中的不同单词
* Created by han on 2016/7/22.
 */
publicclassSetTest {

publicstaticvoidmain(String[] args) {
Set<String> words = newHashSet<>();
longtotaltime = 0;
Scanner in = newScanner(System.in);

while(in.hasNext()) {

String word = in.next();
longcalltime = System.currentTimeMillis();
words.add(word);
calltime = System.currentTimeMillis() - calltime;

totaltime += calltime;
        }

Iterator<String> iterator = words.iterator();
for(inti = 0; i < 20 && iterator.hasNext(); i++)
System.out.println(iterator.next());//next方法是返回迭代的下一个元素

System.out.println(".........");
System.out.println(words.size() + " distinct words " + totaltime + " millseconds");

    }
}
```

树集
TreeSet类与散列集非常相似 树集是一个**有序集合**
可以将任意顺序将元素插入到集合 在对集合遍历时 每个值就自动地按照**排序**后的顺序呈现
下面是个列子

```
SortedSet<String> sorter=new TreeSet<>();
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
```

```
for (String s:sorter) System.out.println(s);
```

这是结果：
Amy
Bob
Carl

其中的排序是用树结构实现的 这里面是采用具体的数据结构的是红黑树 每次将一个元素添加到树中 都被放置到正确的排序位置上 **迭代器总是以排好序的顺序访问每个元素**
将一个元素添加到树中比**添加到散列表慢** 但是与将**元素添加到数组或链表要快多了** 如果树包含了n个元素 **查找新元素的正确位置平均需要log2 n次比较**

TreeSet可以自动的对元素进行排序

构造方法摘要

TreeSet() 构造一个新的空 set，该 set 根据其元素的自然顺序进行排序。
TreeSet(Collection<? extends E> c) 构造一个包含指定 collection 元素的新 TreeSet，它按照其元素的*自然顺序*进行排序。
TreeSet(Comparator<? super E> comparator) 构造一个新的空 TreeSet，它根据指定比较器进行排序。
TreeSet(SortedSet<E> s) 构造一个与指定有序 set 具有相同映射关系和相同排序的新 TreeSet。

对象的比较
TreeSet是如何知道关于元素是希望如何排序的呢。
TreeSet 在**默认情况下假定插入的元素实现了Comparable接口** 这个接口定义了一个方法

```
@Override
public int compareTo(T o) {
return 0;
}
```

这是一个进行比较的方法 假如有 a ，b 如果相同 就返回0 如果排序a位于b之前返回负值 如果之后 返回正值
有的java平台类实现了comparable 比如String类

如果插入自定义的对象 就必须通过实现Comparable接口自定义排列顺序
列如 下面就实现了用部件编号给Item对象进行排序

```
class Item implements Comparable<Item>
{


@Override
public int compareTo(Item o) {
return partNumber - o.partNumber;
}
}
```

这里就是可以进行比较 注意 只有**整数在一个足够小的范围内** 才可以使用这个技巧 太大就会溢出
但是 对一个给定的类 只能实现这个接口一次 如果有**多种需求需要通用** 比如说：如果在一个集合中需要按照部件编号进行排序 另一个集合中却要按照描述信息进行排序 的情况下 就不能满足 可以通过将**Comparator**对象传递给TreeSet构造器告诉树集使用不同的比较方法
Comparator接口声明了一个带有俩显示参数的compare方法 如下：

```
class ItemComparator implements Comparator<Item>
{


@Override
public int compare(Item o1, Item o2) {
return 0;
}
}
```

我们就可以通过俩个参数来进行自定义的比较 然后将这个类的对象传递给TreeSet的构造器

```
ItemComparator itemComparator=new ItemComparator();
SortedSet<Item> sortedSet=new TreeSet<>(itemComparator);
```

这样就构建了一个带有**特殊比较器的树** 就可以在需要**比较俩个元素时使用这个对象**

有时将这种对象叫**函数对象** 函数对象通常有**动态**定义 即**定义为匿名内部类的实例**

```
ItemComparator itemComparator=new ItemComparator();
SortedSet<Item> sortedSet=new TreeSet<>(new Comparator() {
@Override
public int compare(Object o1, Object o2) {
return 0;
}
  });
```

注释：在Comparator<T>接口声明了俩个方法 compare和equals API文档说 不需要覆盖equals方法 但这样做在某些情况下可以带来性能的提升

在这里学习了俩种集合 一个是散列集 一个是树集 到底使用哪种是看情况的 取决于需要收集的数据 是否需要排序 不需要就使用散列表 更重要的是对某些数据来说 对其排序比散列函数更加困难 尤其注意 树的排序是**全序** 也就是说 任意俩个元素必须是可比的 比如有一个经典的问题 就是不可比的 ：矩形 具体原因翻书
下面有一个列子 是关于实现TreeSet的方法的 其中实现了自定义排序规则 关于Item是需要实现一个Comparable接口（并且自定义排序规则）的才能加入到这个TreeSet中

```
package com.jiheframework;

import java.util.*;

/**
 * Created by han on 2016/7/23.
 */
public class TreeSetTest {

public static void main(String[] args) {
        SortedSet<Item> parts = new TreeSet<>();
parts.add(new Item("Toaster", 1234));
parts.add(new Item("widget", 4562));
parts.add(new Item("Midem", 9912));
System.out.println(parts);

SortedSet<Item> sortedSet=new TreeSet<>(new Comparator<Item>() {
@Override
public int compare(Item a, Item b) {
                String descA=a.getDescription();
                String descB=b.getDescription();
                return -1;//
}
        });
sortedSet.addAll(parts);
System.out.println(sortedSet);
         for (Item item:parts) System.out.println(item.getCeshi()+""+item.getCeshi2());
         for (Item item:sortedSet) System.out.println(item.getCeshi()+""+item.getCeshi2());

}
}
```