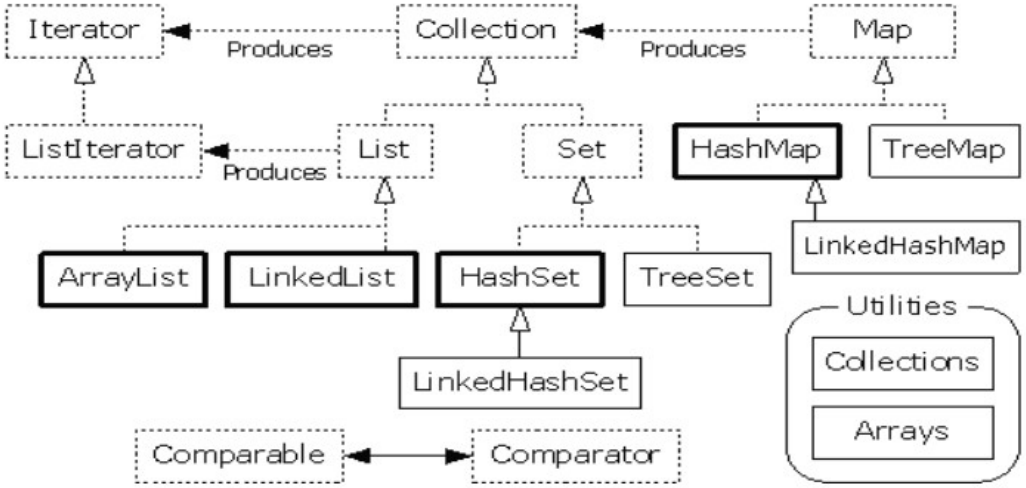
集合

简化图：



1.所有集合类都位于**java.util**包下。Java的集合类主要由两个接口派生而出：**Collection**和**Map**，Collection和Map是Java集合框架的根接口，这两个接口又包含了一些子接口或实现类。

5. Collection 接口是一组允许重复的对象。

6. Set 接口继承 Collection，集合元素不重复。

7. List 接口继承 Collection，允许重复，维护元素插入顺序。

8. Map接口是键 - 值对象，**与Collection接口没有什么关系。**

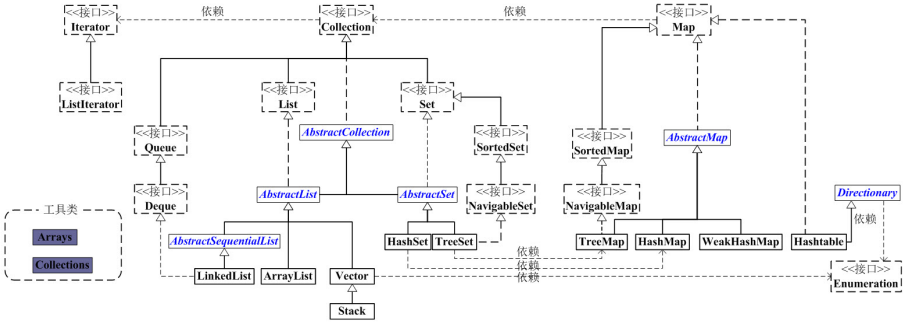
9.Set、List和Map可以看做集合的三大类：

List集合是有序集合，集合中的元素可以重复，访问集合中的元素可以根据元素的索引来访问。

Set集合是无序集合，集合中的元素不可以重复，访问集合中的元素只能根据元素本身来访问（也是集合里元素不允许重复的原因）。

Map集合中保存Key-value对形式的元素，访问时只能根据每项元素的key来访问其value。

二、总体分析



大致说明：

看上面的框架图，先抓住它的主干，即**Collection和Map。**

1、Collection是一个接口，是高度抽象出来的集合，**它包含了集合的基本操作和属性。**Collection包含了**List和Set**两大分支。

（1）List是一个有序的队列，每一个元素都有它的索引。第一个元素的索引值是0。List的实现类有LinkedList, ArrayList, Vector, Stack。

（2）Set是一个不允许有重复元素的集合。Set的实现类有HastSet和TreeSet。HashSet依赖于HashMap，它实际上是通过HashMap实现的；TreeSet依赖于TreeMap，它实际上是通过TreeMap实现的。

2、Map是一个映射接口，即key-value键值对。Map中的每一个元素包含“一个key”和“key对应的value”。AbstractMap是个抽象类，它实现了Map接口中的大部分API。而HashMap，TreeMap，WeakHashMap都是继承于AbstractMap。

Hashtable虽然继承于Dictionary，但它实现了Map接口。

3、接下来，再看Iterator。它是遍历集合的工具，即我们通常通过Iterator迭代器来遍历集合。我们说Collection依赖于Iterator，是因为Collection的实现类都要实现iterator()函数，返回一个Iterator对象。ListIterator是专门为遍历List而存在的。

5、最后，看Arrays和Collections。它们是操作数组、集合的两个工具类。

有了上面的整体框架之后，我们接下来对每个类分别进行分析。

三、Collection接口

Collection接口是处理对象集合的**根接口**，其中定义了很多对元素进行操作的方法。Collection接口有两个主要的子接口**List**和**Set**，注意**Map不是Collection的子接口，这个要牢记**。

Collection接口中的方法如下：

其中，有几个比较常用的方法，比如方法add()添加一个元素到集合中，addAll()将指定集合中的所有元素添加到集合中，contains()方法检测集合中是否包含指定的元素，toArray()方法返回一个表示集合的数组。

另外，Collection中有一个iterator()函数，它的作用是返回一个Iterator接口。通常，我们通过Iterator迭代器来遍历集合。ListIterator是List接口所特有的，在List接口中，通过ListIterator()返回一个ListIterator对象。

Collection接口有两个常用的子接口，下面详细介绍。

1.List接口

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。

List接口继承于Collection接口，它可以定义一个**允许重复**的**有序集合**。因为List中的元素是有序的，所以我们可以通过使用索引（元素在List中的位置，类似于数组下标）来访问List中的元素，这类似于Java的数组。

List接口为Collection直接接口。List所代表的是**有序的Collection**，即它用某种特定的插入顺序来维护元素顺序。用户可以对列表中每个元素的插入位置进行精确地控制，同时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。实现List接口的集合主要有：ArrayList、LinkedList、Vector、Stack。

（1）ArrayList

ArrayList是一个**动态数组**，也是我们最常用的集合。它允许任何符合规则的元素插入甚至包括null。每一个ArrayList都有一个初始容量（10），该容量代表了数组的大小。随着容器中的元素不断增加，容器的大小也会随着增加。在每次向容器中增加元素的同时都会进行容量检查，当快溢出时，就会进行扩容操作。所以如果我们明确所插入元素的多少，最好指定一个初始容量值，避免过多的进行扩容操作而浪费时间、效率。

size、isEmpty、get、set、iterator 和 listIterator 操作都以固定时间运行。add 操作以分摊的固定时间运行，也就是说，添加 n 个元素需要 O(n) 时间（由于要考虑到扩容，所以这不只是添加元素会带来分摊固定时间开销那样简单）。

ArrayList擅长于随机访问。同时ArrayList是非同步的。

（2）LinkedList

同样实现List接口的LinkedList与ArrayList不同，**ArrayList是一个动态数组，而LinkedList是一个双向链表**。所以它除了有ArrayList的基本操作方法外还额外提供了get，remove，insert方法在LinkedList的首部或尾部。

由于实现的方式不同，**LinkedList不能随机访问**，它所有的操作都是要按照双重链表的需要执行。在列表中索引的操作将从开头或结尾遍历列表（从靠近指定索引的一端）。这样做的好处就是可以通过较低的代价在List中进行插入和删除操作。

与ArrayList一样，**LinkedList也是非同步的**。如果多个线程同时访问一个List，则必须自己实现访问同步。一种解决方法是在创建List时构造一个同步的List：

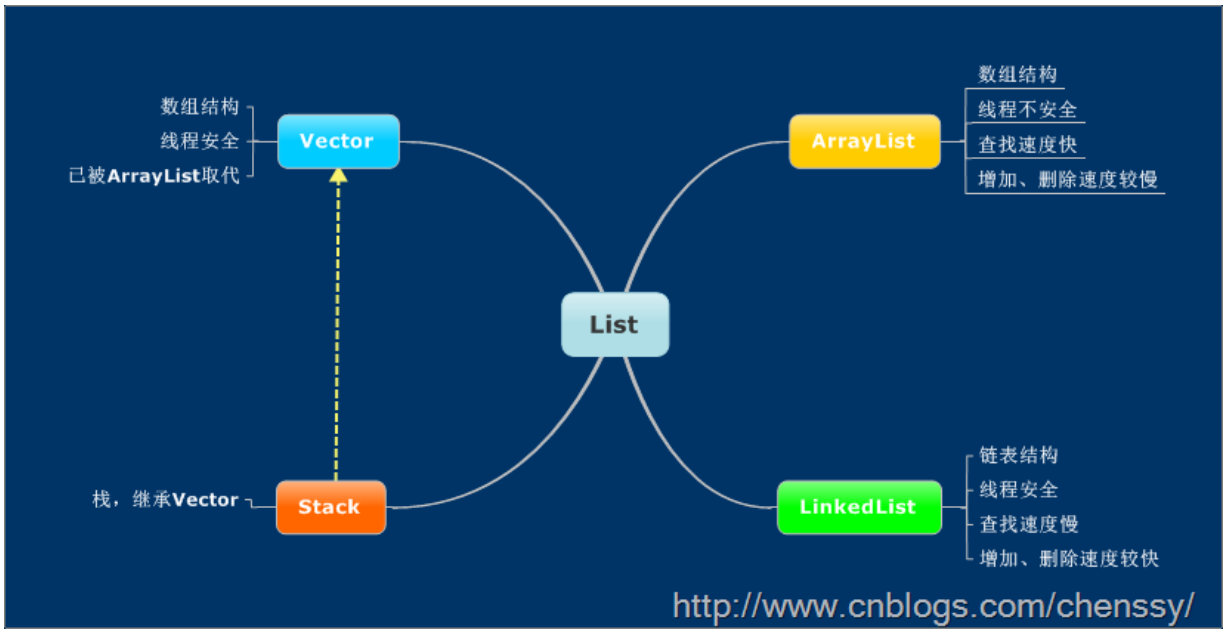
List list = Collections.synchronizedList(new LinkedList(...));

（3）Vector

与ArrayList相似，但是**Vector是同步的**。所以说**Vector是线程安全的动态数组**。它的操作与ArrayList几乎一样。

（4）Stack

Stack继承自Vector，实现一个后进先出的堆栈。Stack提供5个额外的方法使得Vector得以被当作堆栈使用。基本的push和pop 方法，还有peek方法得到栈顶的元素，empty方法测试堆栈是否为空，search方法检测一个元素在堆栈中的位置。Stack刚创建后是空栈。



2.Set接口

Set是一种**不包括重复元素的Collection**。它维持它自己的内部排序，所以随机访问没有任何意义。与List一样，它同样允许null的存在但是**仅有一个**。由于Set接口的特殊性，**所有传入Set集合中的元素都必须不同**，同时要注意任何可变对象，如果在对集合中元素进行操作时，导致e1.equals(e2)==true，则必定会产生某些问题。Set接口有三个具体实现类，分别是散列集HashSet、链式散列集LinkedHashSet和树形集TreeSet。

Set是一种不包含重复的元素的Collection，无序，即任意的两个元素e1和e2都有**e1.equals(e2)=false**，Set最多有一个null元素。需要注意的是:虽然Set中元素没有顺序，但是元素在set中的位置是由该元素的HashCode决定的，其具体位置其实是固定的。

此外需要说明一点，在set接口中的不重复是有特殊要求的。

举一个例子:对象A和对象B，本来是不同的两个对象，正常情况下它们是能够放入到Set里面的，但是如果对象A和B的都重写了hashcode和equals方法，并且重写后的hashcode和equals方法是相同的话。那么A和B是不能同时放入到Set集合中去的，也就是Set集合中的去重和hashcode与equals方法直接相关。

为了更好地理解，请看下面的例子：

复制代码

```
public class Test{
public static void main(String[] args) {
    Set<String> set=new HashSet<String>();
    set.add("Hello");
```

```
        set.add("world");
        set.add("Hello");
        System.out.println("集合的尺寸为:"+set.size());
        System.out.println("集合中的元素为:"+set.toString());
    }
}
```

复制代码

运行结果：

集合的尺寸为:2

集合中的元素为:[world, Hello]

分析：由于String类中重写了hashcode和equals方法，用来比较指向的字符串对象所存储的字符串是否相等。所以这里的第二个Hello是加不进去的。

再看一个例子：

复制代码

```
public class TestSet {

    public static void main(String[] args){

        Set<String> books = new HashSet<String>();
        //添加一个字符串对象
        books.add(new String("Struts2权威指南"));

        //再次添加一个字符串对象，
        //因为两个字符串对象通过equals方法比较相等，所以添加失败，返回false
        boolean result = books.add(new String("Struts2权威指南"));

        System.out.println(result);

        //下面输出看到集合只有一个元素
        System.out.println(books);


    }
}
```

复制代码

运行结果：

false

[Struts2权威指南]

说明：程序中，book集合两次添加的字符串对象明显不是一个对象（程序通过new关键字来创建字符串对象），当使用==运算符判断返回false，使用equals方法比较返回true，所以不能添加到Set集合中，最后只能输出一个元素。

（1）HashSet

HashSet 是一个**没有重复**元素的集合。它是由**HashMap实现的**，**不保证元素的顺序(这里所说的没有顺序是指：元素插入的顺序与输出的顺序不一致)，而且HashSet允许使用null 元素**。HashSet是**非同步的**，如果多个线程同时访问一个哈希set，而其中至少一个线程修改了该set，那么它必须保持外部同步。 HashSet按Hash算法来存储集合的元素，因此具有很好的存取和查找性能。

HashSet的实现方式大致如下，通过一个HashMap存储元素，元素是存放在HashMap的Key中，而Value统一使用一个Object对象。

HashSet使用和理解中容易出现的误区:

a.HashSet中存放null值

HashSet中是允许存入null值的，但是在HashSet中仅仅能够存入一个null值。

b.HashSet中存储元素的位置是固定的

HashSet中存储的元素的是无序的，这个没什么好说的，但是由于HashSet底层是基于Hash算法实现的，使用了hashcode，所以HashSet中相应的元素的位置是固定的。

c.必须小心操作可变对象（Mutable Object）。如果一个Set中的可变元素改变了自身状态导致Object.equals(Object)=true将导致一些问题。

（2）LinkedHashSet

LinkedHashSet继承自HashSet，其底层是**基于LinkedHashMap来实现**的，**有序，非同步**。LinkedHashSet集合同样是根据元素的hashCode值来决定元素的存储位置，但是它同时**使用链表维护元素的次序**。这样使得元素看起来像是以插入顺序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。

（3）TreeSet

TreeSet是一个**有序集合**，其底层是**基于TreeMap实现**的，**非线程安全**。TreeSet可以确保集合元素处于排序状态。TreeSet支持两种排序方式，**自然排序和定制排序**，其中自然排序为默认的排序方式。当我们构造TreeSet时，若使用不带参数的构造函数，则TreeSet的使用自然比较器；若用户需要使用自定义的比较器，则需要使用带比较器的参数。

注意：TreeSet集合不是通过hashcode和equals函数来比较元素的.它是通过compare或者comparaeTo函数来判断元素是否相等.compare函数通过判断两个对象的id，相同的id判断为重复元素，不会被加入到集合中。

四、Map接口

Map与List、Set接口不同，它是由一系列键值对组成的集合，提供了key到Value的映射。同时它也没有继承Collection。在Map中它保证了key与value之间的一一对应关系。也就是说一个key对应一个value，所以它不能存在相同的key值，当然value值可以相同。

1.HashMap

以哈希表数据结构实现，查找对象时通过哈希函数计算其位置，它是为快速查询而设计的，其内部定义了一个hash表数组（Entry[] table），元素会通过哈希转换函数将元素的哈希地址转换成数组中存放的索引，如果有冲突，则使用散列链表的形式将所有相同哈希地址的元素串起来，可能通过查看HashMap.Entry的源码它是一个单链表结构。

2.LinkedHashMap

LinkedHashMap是HashMap的一个子类，它保留**插入的顺序**，如果需要输出的顺序和输入时的相同，那么就选用LinkedHashMap。

LinkedHashMap是Map接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现提供所有可选的映射操作，并**允许使用null值和null键**。此类不保证映射的顺序，特别是它**不保证该顺序恒久不变**。

LinkedHashMap实现与HashMap的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可以是**插入顺序或者是访问顺序**。

根据链表中元素的顺序可以分为：按插入顺序的链表，和按访问顺序(调用get方法)的链表。**默认是按插入顺序排序**，如果指定按访问顺序排序，那么调用get方法后，会将这次访问的元素移至链表尾部，不断访问可以形成按访问顺序排序的链表。

注意，此实现**不是同步的**。如果多个线程同时访问链接的哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。

由于LinkedHashMap需要维护元素的插入顺序，因此性能略低于HashMap的性能，但在迭代访问Map里的全部元素时将有很好的性能，因为它以链表来维护内部顺序。

3.TreeMap

TreeMap 是一个**有序的**key-value集合，**非同步**，**基于红黑树（Red-Black tree）实现**，每一个key-value节点作为红黑树的一个节点。TreeMap存储时会进行排序的，会根据**key**来对key-value键值对进行排序，其中排序方式也是分为两种，一种是**自然排序**，一种是**定制排序**，具体取决于使用的构造方法。

自然排序：TreeMap中所有的key**必须实现Comparable接口**，并且所有的key都应该是**同一个类的对象**，否则会报ClassCastException异常。

定制排序：定义TreeMap时，创建一个**comparator对象**，该对象对所有的treeMap中所有的key值进行排序，采用定制排序的时候不需要TreeMap中所有的key必须实现Comparable接口。

TreeMap判断两个元素相等的标准：两个key通过compareTo()方法返回0，则认为这两个key相等。

如果使用自定义的类来作为TreeMap中的key值，且想让TreeMap能够良好的工作，则**必须重写自定义类中的equals()方法**，TreeMap中判断相等的标准是：两个key通过equals()方法返回为true，并且通过compareTo()方法比较应该返回为0。

五、Iterator 与 ListIterator详解

1.Iterator

Iterator的定义如下：

```
public interface Iterator<E> {}
```

Iterator是一个接口，它是集合的迭代器。集合可以通过Iterator去遍历集合中的元素。Iterator提供的API接口如下：

boolean hasNext()：判断集合里是否存在下一个元素。如果有，hasNext()方法返回 true。

Object next()：返回集合里下一个元素。

void remove()：删除集合里上一次next方法返回的元素。

使用示例：

复制代码

```
public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        a.add("aaa");
        a.add("bbb");
        a.add("ccc");
        System.out.println("Before iterate : " + a);
        Iterator<String> it = a.iterator();
        while (it.hasNext()) {
            String t = it.next();
            if ("bbb".equals(t)) {
                it.remove();
            }
        }
        System.out.println("After iterate : " + a);
```

```
    }
}
```

复制代码

输出结果如下：

```
Before iterate : [aaa, bbb, ccc]
After iterate : [aaa, ccc]
```

注意：

（1）Iterator只能单向移动。

（2）Iterator.remove()是唯一安全的方式来在迭代过程中修改集合；如果在迭代过程中以任何其它的方式修改了基本集合将会产生未知的行为。而且每调用一次next()方法，remove()方法只能被调用一次，如果违反这个规则将抛出一个异常。

2.ListIterator

ListIterator是一个功能更加强大的迭代器，它继承于Iterator接口,只能用于各种List类型的访问。可以通过调用listIterator()方法产生一个指向List开始处的ListIterator，还可以调用listIterator(n)方法创建一个一开始就指向列表索引为n的元素处的ListIterator.

ListIterator接口定义如下:

复制代码

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();

    E next();

    boolean hasPrevious();

    E previous();

    int nextIndex();

    int previousIndex();

    void remove();

    void set(E e);

    void add(E e);

}
```

复制代码

由以上定义我们可以推出ListIterator可以:

(1)双向移动（向前/向后遍历）.

(2)产生相对于迭代器在列表中指向的当前位置的前一个和后一个元素的索引.

(3)可以使用set()方法替换它访问过的最后一个元素.

(4)可以使用add()方法在next()方法返回的元素之前或previous()方法返回的元素之后插入一个元素.

使用示例：

复制代码

```
public class ListIteratorExample {

    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        a.add("aaa");
```

```
        a.add("bbb");
        a.add("ccc");
        System.out.println("Before iterate : " + a);
        ListIterator<String> it = a.listIterator();
        while (it.hasNext()) {
            System.out.println(it.next() + ", " + it.previousIndex() + ", " + it.nextIndex());
        }
        while (it.hasPrevious()) {
            System.out.print(it.previous() + " ");
        }
        System.out.println();
        it = a.listIterator(1);
        while (it.hasNext()) {
            String t = it.next();
            System.out.println(t);
            if ("ccc".equals(t)) {
                it.set("nnn");
            } else {
                it.add("kkk");
            }
        }
        System.out.println("After iterate : " + a);
    }
}
```

复制代码

输出结果如下：

复制代码

```
Before iterate : [aaa, bbb, ccc]
aaa, 0, 1
bbb, 1, 2
ccc, 2, 3
ccc bbb aaa
bbb
ccc
After iterate : [aaa, bbb, kkk, nnn]
```

复制代码

六、异同点

1.ArrayList和LinkedList

（1）ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

（2）对于随机访问get和set，ArrayList绝对优于LinkedList，因为LinkedList要移动指针。

（3）对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

这一点要看实际情况的。若只对单条数据插入或删除，ArrayList的速度反而优于LinkedList。但若是批量随机的插入删除数据，LinkedList的速度大大优于ArrayList. 因为ArrayList每插入一条数据，要移动插入点及之后的所有数据。

2.HashTable与HashMap

相同点：

（1）都实现了Map、Cloneable、java.io.Serializable接口。

（2）都是存储"键值对(key-value)"的散列表，而且都是采用拉链法实现的。

不同点：

（1）历史原因:HashTable是基于陈旧的Dictionary类的，HashMap是Java 1.2引进的Map接口的一个实现 。

（2）同步性:HashTable是线程安全的，也就是说是同步的，而HashMap是线程序不安全的，不是同步的 。

（3）对null值的处理：HashMap的key、value都可为null，HashTable的key、value都不可为null 。

（4）基类不同：HashMap继承于AbstractMap，而Hashtable继承于Dictionary。

Dictionary是一个抽象类，它直接继承于Object类，没有实现任何接口。Dictionary类是JDK 1.0的引入的。虽然Dictionary也支持“添加key-value键值对”、“获取value”、“获取大小”等基本操作，但它的API函数比Map少；而且Dictionary一般是通过Enumeration(枚举类)去遍历，Map则是通过Iterator(迭代M器)去遍历。 然而由于Hashtable也实现了Map接口，所以，它即支持Enumeration遍历，也支持Iterator遍历。

AbstractMap是一个抽象类，它实现了Map接口的绝大部分API函数；为Map的具体实现类提供了极大的便利。它是JDK 1.2新增的类。

（5）支持的遍历种类不同：HashMap只支持Iterator(迭代器)遍历。而Hashtable支持Iterator(迭代器)和Enumeration(枚举器)两种方式遍历。

3.HashMap、Hashtable、LinkedHashMap和TreeMap比较

Hashmap 是一个最常用的Map，**它根据键的HashCode 值存储数据**，根据键可以直接获取它的值，**具有很快的访问速度**。遍历时，**取得数据的顺序是完全随机的**。HashMap最多只允许**一条记录的键为Null;**允许多条记录的值为Null;HashMap**不支持线程的同步**，即任一时刻可以有多个线程同时写HashMap;可能会导致数据的不一致。如果需要同步，可以用Collections的synchronizedMap方法使HashMap具有同步的能力。

Hashtable 与 HashMap类似，不同的是:**它不允许记录的键或者值为空**;它**支持线程的同步**，即任一时刻只有一个线程能写Hashtable，因此也导致了Hashtale在写入时会比较慢。

LinkedHashMap**保存了记录的插入顺序**，在用Iterator遍历LinkedHashMap时，**先得到的记录肯定是先插入的**，也可以在构造时用带参数，按照应用次数排序。在遍历的时候会比HashMap慢，不过有种情况例外，当HashMap容量很大，实际数据较少时，遍历起来可能会比LinkedHashMap慢，因为LinkedHashMap的遍历速度只和实际数据有关，和容量无关，而HashMap的遍历速度和他的容量有关。**如果需要输出的顺序和输入的相同，那么用LinkedHashMap可以实现**，它还可以按读取顺序来排列，像连接池中可以应用。LinkedHashMap实现与HashMap的不同之处在于，后者维护着一个运行于所有条目的双重链表。此链接列表定义了迭代顺序，**该迭代顺序可以是插入顺序或者是访问顺序**。对于LinkedHashMap而言，它继承与HashMap、底层使用**哈希表与双向链表来保存所有元素**。其基本操作与父类HashMap相似，它通过重写父类相关的方法，来实现自己的链接列表特性。

TreeMap实现SortMap接口，内部实现是**红黑树**。能够把它保存的记录**根据键排序**，**默认是按键值的升序排序**，也可以指定排序的比较器，**当用Iterator 遍历TreeMap时，得到的记录是排过序的。TreeMap不允许key的值为null。非同步的。**

一般情况下，我们用的最多的是HashMap，HashMap里面存入的键值对在取出的时候是随机的，它根据键的HashCode值存储数据，根据键可以直接获取它的值，具有很快的访问速度。在Map 中插入、删除和定位元素，HashMap是最好的选择。

TreeMap取出来的是排序后的键值对。但如果您要**按自然顺序或自定义顺序遍历键**，那么TreeMap会更好。

LinkedHashMap 是HashMap的一个子类，如果需要**输出的顺序和输入的相同，**那么用LinkedHashMap可以实现，它还可以按读取顺序来排列，像连接池中可以应用。

复制代码

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.TreeMap;

public class MapTest {

    public static void main(String[] args) {

        //HashMap
        HashMap<String,String> hashMap = new HashMap();
```

```
        hashMap.put("4", "d");
        hashMap.put("3", "c");
        hashMap.put("2", "b");
        hashMap.put("1", "a");

        Iterator<String> iteratorHashMap = hashMap.keySet().iterator();

        System.out.println("HashMap-->");

        while (iteratorHashMap.hasNext()){

            Object key1 = iteratorHashMap.next();
            System.out.println(key1 + "--" + hashMap.get(key1));
        }

        //LinkedHashMap
        LinkedHashMap<String,String> linkedHashMap = new LinkedHashMap();
        linkedHashMap.put("4", "d");
        linkedHashMap.put("3", "c");
        linkedHashMap.put("2", "b");
        linkedHashMap.put("1", "a");

        Iterator<String> iteratorLinkedHashMap = linkedHashMap.keySet().iterator();

        System.out.println("LinkedHashMap-->");

        while (iteratorLinkedHashMap.hasNext()){

            Object key2 = iteratorLinkedHashMap.next();
            System.out.println(key2 + "--" + linkedHashMap.get(key2));
        }

        //TreeMap
        TreeMap<String,String> treeMap = new TreeMap();
        treeMap.put("4", "d");
        treeMap.put("3", "c");
        treeMap.put("2", "b");
        treeMap.put("1", "a");

        Iterator<String> iteratorTreeMap = treeMap.keySet().iterator();

        System.out.println("TreeMap-->");

        while (iteratorTreeMap.hasNext()){

            Object key3 = iteratorTreeMap.next();
            System.out.println(key3 + "--" + treeMap.get(key3));
        }

    }

}
```

复制代码

输出结果：

复制代码

```
HashMap-->
3--c
2--b
1--a
4--d
LinkedHashMap-->
4--d
3--c
2--b
1--a
TreeMap-->
1--a
2--b
3--c
4--d
```

复制代码

4.HashSet、LinkedHashSet、TreeSet比较

Set接口

Set**不允许包含相同的元素**，如果试图把两个相同元素加入同一个集合中，add方法返回false。

Set判断两个对象相同不是使用==运算符，而是**根据equals方法**。也就是说，只要两个对象用equals方法比较返回true，Set就不会接受这两个对象。

HashSet

HashSet有以下特点：

-> 不能保证元素的排列顺序，顺序有可能发生变化。

-> 不是同步的。

-> 集合元素可以是null，但只能放入一个null。

当向HashSet结合中存入一个元素时，HashSet会调用该对象的hashCode()方法来得到该对象的hashCode值，然后根据 hashCode值来决定该对象在HashSet中存储位置。简单的说，HashSet集合判断两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。

注意，如果要把一个对象放入HashSet中，重写该对象对应类的equals方法，也应该重写其hashCode()方法。其规则是如果两个对象通过equals方法比较返回true时，其hashCode也应该相同。另外，对象中用作equals比较标准的属性，都应该用来计算 hashCode的值。

LinkedHashSet

LinkedHashSet集合同样是**根据元素的hashCode值来决定元素的存储位置**，但是它同时使用链表维护元素的次序。这样使得元素看起来像是以插入顺序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。

LinkedHashSet在迭代访问Set中的全部元素时，**性能比HashSet好，但是插入时性能稍微逊色于HashSet。**

TreeSet类

TreeSet是SortedSet接口的唯一实现类，TreeSet可以确保集合元素处于排序状态。TreeSet支持两种排序方式，自然排序和定制排序，其中自然排序为默认的排序方式。向TreeSet中加入的应该是同一个类的对象。

TreeSet判断两个对象不相等的方式是两个对象通过equals方法返回false，或者通过CompareTo方法比较没有返回0。

自然排序

自然排序使用要排序元素的CompareTo（Object obj）方法来比较元素之间大小关系，然后将元素按照升序排列。

Java提供了一个Comparable接口，该接口里定义了一个compareTo(Object obj)方法，该方法返回一个整数值，实现了该接口的对象就可以比较大小。obj1.compareTo(obj2)方法如果返回0，则说明被比较的两个对象相等，如果返回一个正数，则表明obj1大于obj2，如果是负数，则表明obj1小于obj2。如果我们将两个对象的equals方法总是返回true，则这两个对象的compareTo方法返回应该返回0。

定制排序

自然排序是根据集合元素的大小，以升序排列，如果要定制排序，应该使用Comparator接口，实现 int compare(T o1,T o2)方法。

复制代码

```
package com.test;

import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.TreeSet;

/**
 * @description 几个set的比较
 *     HashSet：哈希表是通过使用称为散列法的机制来存储信息的，元素并没有以某种特定顺序来存放；
 *     LinkedHashSet：以元素插入的顺序来维护集合的链接表，允许以插入的顺序在集合中迭代；
 *     TreeSet：提供一个使用树结构存储Set接口的实现，对象以升序顺序存储，访问和遍历的时间很快。
 * @author Zhou-Jingxian
 *
 */
public class SetDemo {

    public static void main(String[] args) {

        HashSet<String> hs = new HashSet<String>();
        hs.add("B");
        hs.add("A");
        hs.add("D");
        hs.add("E");
        hs.add("C");
        hs.add("F");
        System.out.println("HashSet 顺序:\n"+hs);

        LinkedHashSet<String> lhs = new LinkedHashSet<String>();
        lhs.add("B");
        lhs.add("A");
        lhs.add("D");
        lhs.add("E");
        lhs.add("C");
        lhs.add("F");
        System.out.println("LinkedHashSet 顺序:\n"+lhs);

        TreeSet<String> ts = new TreeSet<String>();
        ts.add("B");
        ts.add("A");
        ts.add("D");
        ts.add("E");
        ts.add("C");
        ts.add("F");
        System.out.println("TreeSet 顺序:\n"+ts);
    }
}
```

复制代码

输出结果：

```
HashSet 顺序:[D, E, F, A, B, C]
LinkedHashSet 顺序:[B, A, D, E, C, F]
TreeSet 顺序:[A, B, C, D, E, F]
```

5、Iterator和ListIterator区别

我们在使用List，Set的时候，为了实现对其数据的遍历，我们经常使用到了Iterator(迭代器)。使用迭代器，你不需要干涉其遍历的过程，只需要每次取出一个你想要的数据进行处理就可以了。但是在使用的时候也是有不同的。List和Set都有iterator()来取得其迭代器。对List来说，你也可以通过listIterator()取得其迭代器，两种迭代器在有些时候是不能通用的，Iterator和ListIterator主要区别在以下方面：

（1）ListIterator有add()方法，可以向List中添加对象，而Iterator不能

（2）ListIterator和Iterator都有hasNext()和next()方法，可以实现顺序向后遍历，但是ListIterator有hasPrevious()和previous()方法，可以实现逆向（顺序向前）遍历。Iterator就不可以。

（3）ListIterator可以定位当前的索引位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。

（4）都可实现删除对象，但是ListIterator可以实现对象的修改，set()方法可以实现。Iierator仅能遍历，不能修改。

因为ListIterator的这些功能，可以实现对LinkedList等List数据结构的操作。其实，数组对象也可以用迭代器来实现。

6、Collection 和 Collections区别

（1）java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。

```
Collection
├List
│ ├LinkedList
│ ├ArrayList
│ └Vector
│   └Stack
└Set
```

（2）java.util.Collections 是一个包装类（工具类/帮助类）。它包含有各种有关集合操作的**静态多态方法**。此类**不能实例化**，就像一个工具类，用于对集合中元素进行排序、搜索以及线程安全等各种操作，服务于Java的Collection框架。

代码示例：

复制代码

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TestCollections {

    public static void main(String args[]) {
        //注意List是实现Collection接口的
        List list = new ArrayList();
        double array[] = { 112, 111, 23, 456, 231 };
        for (int i = 0; i < array.length; i++) {
            list.add(new Double(array[i]));
        }
        Collections.sort(list);
        for (int i = 0; i < array.length; i++) {
            System.out.println(list.get(i));
        }
        // 结果：23.0 111.0 112.0 231.0 456.0
    }
}
```