<<ArrayList>>

- ArrayList内部使用的是数组存储数据的

下面是ArrayList的源码中定义的一些变量

```
public class ArrayList<E> extends AbstractList<E>
 implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
 //声明一个序列号,方便正在网络中稳定的传输
 private static final long serialVersionUID = 8683452581122892189L;
 //ArrayList的默认初始容量,也就是ArrayList初始化的底层存储Object数组的长度
 private static final int DEFAULT_CAPACITY = 10;
 //因为下面的真正用来存储的Object数组没有初始化,所以如果第一次初始化ArrayList的时候,会直接this.elementData = EMPTY_ELEMENTDATA 这样直接初始化为空数组
 private static final Object[] EMPTY_ELEMENTDATA = {};
 //这是一个空的数组,长度是0,之后判断是否是第一次初始化ArrayList的时候,会和这个进行比较,进而作出判断是否为空的未初始的新的ArrayList,如果是那么就会进行初始化
 private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
 //ArrayList真正用来存储数据的Object数组,这里刚开始声明的时候我们可以发现数组的长度是0
 transient Object[] elementData; // non-private to simplify nested class access
 //size是下一个即将存入的元素的下标,也就是目前所有的元素的个数
 //这里在每一次存入或者删除的时候,size都会立刻改变的,所以在ArrayList的size()方法的时候效率是很高的,不需要遍历,直接return size即可
 private int size;
```

这里有一个关于序列化的问题:

Q:为什么ArrayList中的数据装载数组对象elementData需要使用关键词transient修饰

A:

1. 什么是序列化

通俗点讲:它是处理对象流的一种机制,即可以很方便的保存内存中java对象的状态,同时也为了方便传输。

把对象转换为字节序列的过程称为对象的序列化。

把字节序列恢复为对象的过程称为对象的反序列化

在很多应用中，需要对某些对象进行序列化，让它们离开内存空间，入住物理硬盘，以便长期保存。比如最常见的是Web服务器中的Session对象，当有 10万用户并发访问，就有可能出现10万个Session对象，内存可能吃不消，于是Web容器就会把一些seesion先序列化到硬盘中，等要用了，再把保存在硬盘中的对象还原到内存中。

当两个进程在进行远程通信时，彼此可以发送各种类型的数据。**无论是何种类型的数据，都会以二进制序列的形式在网络上传送。发送方需要把这个Java对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为Java对象**

2. 序列化有什么作用:

- 方便传输，速度快，还很安全，被调用方序列化，调用方反序列化即可拿到传输前最原始的java对象，常用于不同进程之间的对象传输
- 方便存储，不管是存储成文件还是数据库，都行，存储为文件，下回要用可以直接反序列拿到对象

3. 为了不必要的报错麻烦： 序列化时最好是定义序列化版本id 即 public static final Long seriaVersionUID = 1L (默认) 或者 xxxxx L（自定义64位都行）.因为反序列化会判断序列化中的id和类中的id是否一样，如果不定义虽然会自动生成，但如果后面改了东西列，所以还是自觉点定义一个id，省去好多麻烦

4. transient:

transient用来表示一个域不是该对象序行化的一部分，当一个对象被序行化的时候，transient修饰的变量的值是不包括在序行化的表示中的。**但是ArrayList又是可序行化的类，elementData是ArrayList具体存放元素的成员，用transient来修饰elementData，岂不是反序列化后的ArrayList丢失了原先的元素？下面是ArrayList中的两个源码的方法**

```
/**
 * Save the state of the <tt>ArrayList</tt> instance to a stream (that
 * is, serialize it).
 *
 * @serialData The length of the array backing the <tt>ArrayList</tt>
 * instance is emitted (int), followed by all of its elements
 * (each an <tt>Object</tt>) in the proper order.
 */
private void writeObject(java.io.ObjectOutputStream s)
 throws java.io.IOException{
 // Write out element count, and any hidden stuff
 int expectedModCount = modCount;
 s.defaultWriteObject();

 // Write out size as capacity for behavioural compatibility with clone()
 s.writeInt(size);

 // Write out all elements in the proper order.
 for (int i=0; i<size; i++) {
 s.writeObject(elementData[i]);
 }

 if (modCount != expectedModCount) {
 throw new ConcurrentModificationException();
 }
}

/**
 * Reconstitute the <tt>ArrayList</tt> instance from a stream (that is,
 * deserialize it).
 */
private void readObject(java.io.ObjectInputStream s)
 throws java.io.IOException, ClassNotFoundException {
 elementData = EMPTY_ELEMENTDATA;

 // Read in size, and any hidden stuff
 s.defaultReadObject();

 // Read in capacity
 s.readInt(); // ignored

 if (size > 0) {
```

```
43  // be like clone(), allocate array based upon size not capacity
44  ensureCapacityInternal(size);
45
46  Object[] a = elementData;
47  // Read in all elements in the proper order.
48  for (int i=0; i<size; i++) {
49  a[i] = s.readObject();
50  }
51  }
52 }
```

ArrayList在序列化的时候会调用writeObject，直接将size和element写入ObjectOutputStream；反序列化时调用readObject，从ObjectInputStream获取size和element，再恢复到elementData。
为什么不直接用elementData来序列化，而采用上诉的方式来实现序列化呢？原因在于elementData是一个缓冲存储数组，它通常会预留一些容量，等容量不足时再扩充容量，那么有些空间可能就没有实际存储元素，采用上诉的方式来实现序列化时，就可以保证只序列化实际存储的那些元素，而不是整个数组，从而节省空间和时间。

- 关于ArrayList扩容:

刚开始的时候,如果我们使用的是ArrayList的无参构造初始化的ArrayList那么,ArrayList的初始容量就是10.

```
 */
public ArrayList() { this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA; }
```

当然我们可以自己指定初始化的容量大小:

```
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}
```

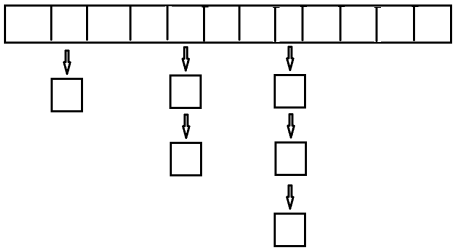
扩容时机:

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

只有在增加的元素个数超过当前容量的时候才会扩容,每次扩容会变为原来的3/2,也就是扩大0.5倍
并且,需要注意的是,每次扩容都会发生数组的copy操作;因为需要生成一个新的大小的数组.

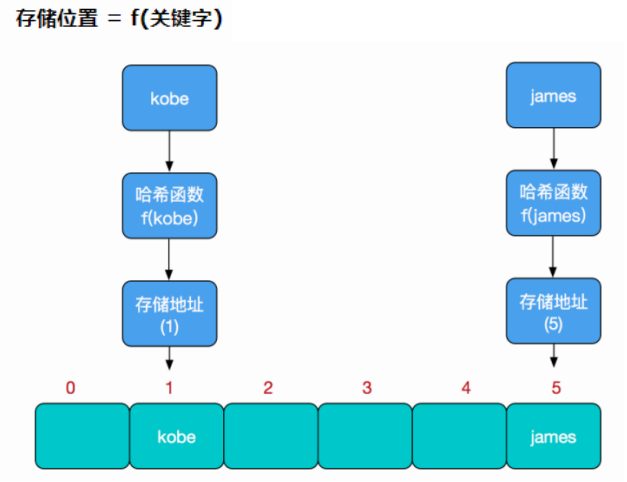
<<HashMap jdk1.7>>

- jdk1.7中,HashMap基本的存储结构是数组加单向链表



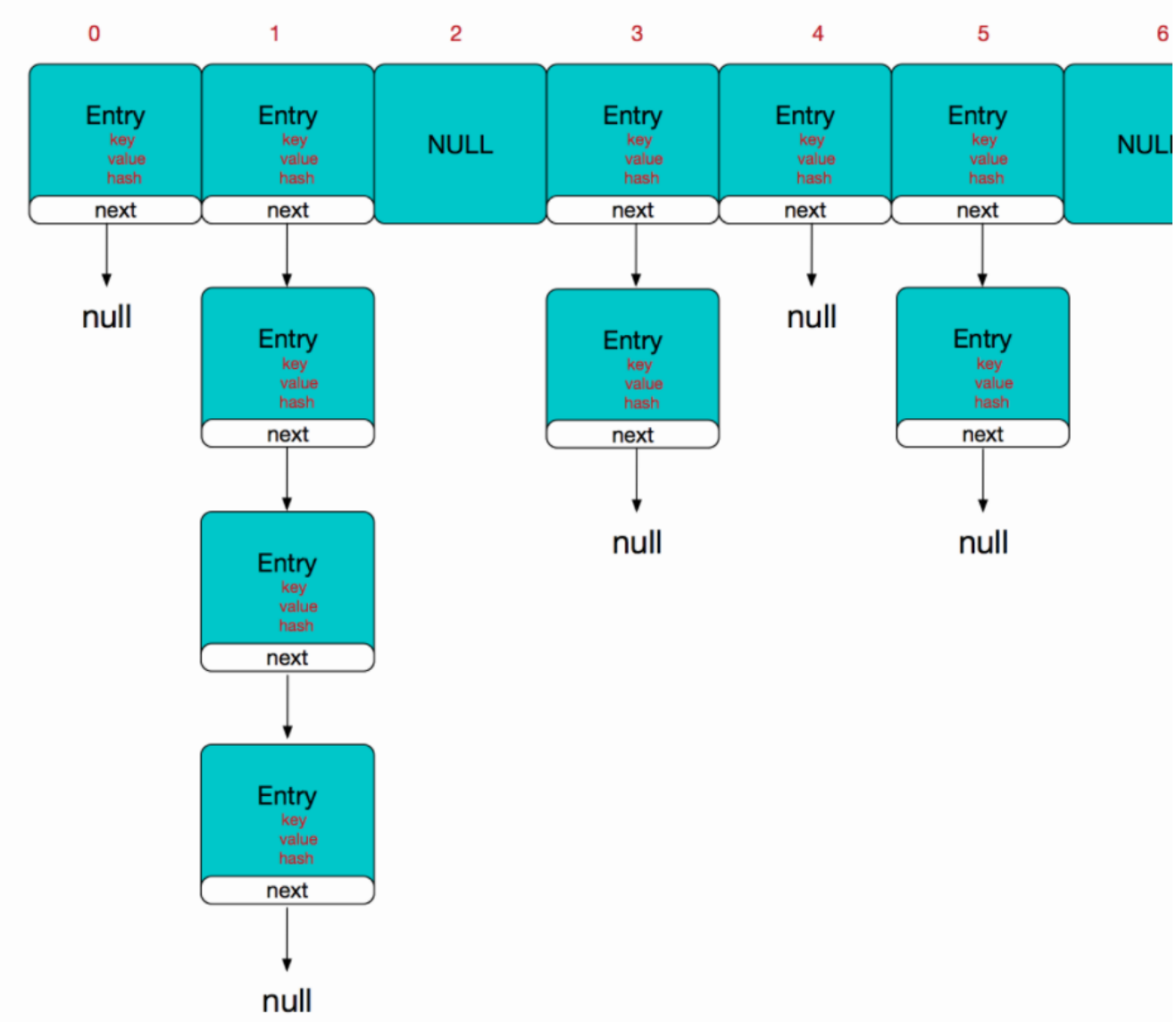
hashmap主要原理就是使用可hash表这样的数据结构,hash表和其他的,如数组,链表等在增加和删除方面有着天生的优势,他可以一次定位到需要操作的位置;主要原理是这样的:

首先,hash表的主体就是一个数组.比如我们要新增或查找某个元素，我们通过把当前元素的关键字 通过某个函数映射到数组中的某个位置，通过数组下标一次定位就可完成操作。



如果两个不同的元素，通过哈希函数得出的实际存储地址相同怎么办？也就是说，当我们对某个元素进行哈希运算，得到一个存储地址，然后要进行插入的时候，发现已经被其他元素占用了，其实这就是所谓的哈希冲突，也叫哈希碰撞。哈希冲突的解决方案有多种:开放定址法（发生冲突，继续寻找下一块未被占用的存储地址），再散列函数法，链地址法，而HashMap 1.7即是采用了链地址法，也就是数组+链表的方式

下面这张图其实就是HashMap1.7中的实际存储的描述图



下面来分析源码:

这是,HashMap1.7中的实际存储元素的hash数组,我们发现这个数组也是transient修饰的,和我们的ArrayList的思想是一样的.

我们发现这个数组存储的数据是Entry<K,V>这样的对象

```
//The table, resized as necessary. Length MUST Always be a power of two.
transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY_TABLE;
```

下面我们来看一些这个Entry对象:

```
static class Entry<K,V> implements Map.Entry<K,V> {
 final K key;
 V value;
 Entry<K,V> next;
 int hash;

 /**
 * Creates new entry.
 */
  Entry(int h, K k, V v, Entry<K,V> n) {
  value = v;
  next = n;
  key = k;
  hash = h;
```

```
15  }
16
17  public final K getKey() {
18  return key;
19  }
20
21  public final V getValue() {
22  return value;
23  }
24
25  public final V setValue(V newValue) {
26  V oldValue = value;
27  value = newValue;
28  return oldValue;
29  }
30
31  public final boolean equals(Object o) {
32  if (!(o instanceof Map.Entry))
33  return false;
34  Map.Entry e = (Map.Entry)o;
35  Object k1 = getKey();
36  Object k2 = e.getKey();
37  if (k1 == k2 || (k1 != null && k1.equals(k2))) {
38  Object v1 = getValue();
39  Object v2 = e.getValue();
40  if (v1 == v2 || (v1 != null && v1.equals(v2)))
41  return true;
42  }
43  return false;
44  }
45
46  public final int hashCode() {
47  return Objects.hashCode(getKey()) ^ Objects.hashCode(getValue());
48  }
49
50  public final String toString() {
51  return getKey() + "=" + getValue();
52  }
53
54  /**
55  * This method is invoked whenever the value in an entry is
56  * overwritten by an invocation of put(k,v) for a key k that's already
57  * in the HashMap.
58  */
59  void recordAccess(HashMap<K,V> m) {
60  }
61
62  /**
63  * This method is invoked whenever the entry is
64  * removed from the table.
65  */
66  void recordRemoval(HashMap<K,V> m) {
67  }
68  }
```

上面是Entry的源码,我们发现其有这样几个变量:

```
final K key;
V value;
Entry<K,V> next;
int hash;
```

也就是我们上面的HashMap1.7结构图的所有的属性;所以我们的HashMap的主体的这个数组结构存储的是一个个Entry对象,这个Entry对象会有一个next的Entry属性,也就是可以构成一个单向的链表;

下面我们来看一下构造方法:

```
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);

    this.loadFactor = loadFactor;
    threshold = initialCapacity;
    init();
}
```

```
public HashMap(int initialCapacity) { this(initialCapacity, DEFAULT_LOAD_FACTOR); }
public HashMap() { this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted }
```

前面的有参构造会接受程序员指定初始容量和加载因子,或者只指定一个厨师容量,最后一个无参构造那么使用的就是默认的加载因子和初始容量:

```
/**
 * The load factor used when none specified in constructor.
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;


/**
 * The default initial capacity - MUST be a power of two.
 */
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

在有参的构造方法中,我们指定了初始的容量,比如

Hash(10,0.5)

那么这个时候,我们实际在创建数组的时候,数组的容量其实不是10,是第一个比10大的2的n次方的数,也就是16,这个为什么我们后面继续讲,主要是这个方法起作用的:

```
// Find a power of 2 >= toSize
int capacity = roundUpToPowerOf2(toSize);
```

其实构造方法结束只是确定了几个主要的参数,一个是加载因子,一个是待定阈值threshold(这个待定的阈值其实就是我们指定的数组的容量

```
threshold = initialCapacity;
```

这里我们暂时使用的是threshold保存的,之后我们需要使用这个我们制定的初始容量,找到一第一个不小于这个数的2的n次方);

但是事实上,真正初始化创建主体的hash存储数组是在第一次put值的时候,我们来看一下源码:

```
1 public V put(K key, V value) {
2   如果是一个空的数组,即第一次添加元素,那么就会进入inflateTable()中,注意这个参数也就是我们指定的容量大小
3   if (table == EMPTY_TABLE) {
```

```
4   inflateTable(threshold);
5   }
6   //这里也就是说明hashmap中key是可以有null的,只允许有一个,在concurrentHashMap中的put源码中,如果key是null或者值是null的话,那么是直接排除一个异常(空指针)
7   if (key == null)
8   return putForNullKey(value);
9           // 根据key计算哈希值
10   int hash = hash(key);
11           // 根据哈希值和数据长度计算数据下标
12   int i = indexFor(hash, table.length);
13   for (Entry<K,V> e = table[i]; e != null; e = e.next) {
14   Object k;
15               // 哈希值相同再比较key是否相同，相同的话值替换，否则将这个槽转成链表
16   if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
17   V oldValue = e.value;
18   e.value = value;
19   e.recordAccess(this);
20   return oldValue;
21   }
22   }
23
24   modCount++;    // fast-fail，迭代时响应快速失败，还未添加元素就进行modCount++,将为后续留下很多隐患
25   addEntry(hash, key, value, i);    // 添加元素，注意最后一个参数i是table数组的下标
26   return null;
27   }
```

```
1
2   private void inflateTable(int toSize) {
3
4   int capacity = roundUpToPowerOf2(toSize); // 返回小于(toSize- 1) *2的最接近的2的次幂 ，如果toSize=1，则capacity=1，所以如果将initcapacity设为的话，第一次put不会扩容
5   //这个阈值也是重新计算的,通过我们新计算得到的capacity容量和我们的加载因子得到的,我们可以发现这个阈值不等于容量,而是比容量要小,因为家在因子比1小,是乘以加载因子
6   threshold = (int) Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
7   //在这里才初始化创建了这个数组,所以我们可以发现,其实这个数组的大小不是我们指定的,是一个2的n次幂
8   table = new Entry[capacity];
9   initHashSeedAsNeeded(capacity);
10   }
```

```
    int hash = hash(key);
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            V oldValue = e.value;
            e.value = value;
            e.recordAccess( m: this);
            return oldValue;
        }
    }

    modCount++;
    addEntry(hash, key, value, i);
    return null;
}
```

HashMap中几个主要的问题:

- hashmap中的modcount

每次修改hashmap(包括增加删除)这些操作的时候都会增加一次modcount的值,也就是说modcount是指hashmap被修改的次数;
这个变量和一个异常相关系;获取map的keySet, entrySet, iterator之后，都可以实现遍历,但是前两种方式在遍历时如果进行了删除操作,也就是修改了hashmap,那么会报错:java.util.ConcurrentModificationException,只有最后一个使用iterator遍历是删除元素才不会报错;

因为前面两种方式在删除的时候,使用到的remove方法可以删除任意指定的key的元素
这三种删除的内部实现分别是这样的:

1、HashMap的remove方法实现

```
public V remove(Object key) {
    Entry<K,V> e = removeEntryForKey(key);
    return (e == null ? null : e.value);
}
```

2、HashMap.KeySet的remove方法实现

```
public boolean remove(Object o) {
    return HashMap.this.removeEntryForKey(o) != null;
}
```

3、HashMap.HashIterator的remove方法实现

```
public void remove() {
   if (current == null)
        throw new IllegalStateException();
   if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
   Object k = current.key;
   current = null;
   HashMap.this.removeEntryForKey(k);
   expectedModCount = modCount;
}
```

其实这三个方法,内部删除使用的都是removeEntryForKey()
这个方法使用的时候会记录modcount的值也就是+1
但是前两个方法在删除之后没有将当前的modcount值重新赋值给exceptedModCount
只有最后一个方法做到了
所以前两个方法使用之后,exceptedModeCount和modCount肯定是不一样的,所有就会报错;

```
if (modCount != expectedModCount)
    throw new ConcurrentModificationException();
```

当然是在下个remove的时候,报错;(因为比较是否相等也是在remove等方法中)

那么现在有这样一个问题,为什么要设计这样一个modcount,从而使只有在iterater迭代器中删除的时候才不会报错.

这其实是因为,其实以上的三种方式,keySet, entrySet, iterator之后，都可以实现遍历,他们其实都是利用的迭代的方式,所以如果我们允许在迭代的时候,可以任意删除元素
就会为迭代的过程带来不可预料的后果,因为,前两种方式中,在遍历的时候,删除remove方法都是可以删除任意一个人元素,这个时候如果删除了当前迭代的下一个next元素,那么就可能出现迭代出现异常而结束,所以我们约定好(加入了modcount);在这些迭代中,不允许remove等改变modcount的操作;而第三种在iterator中遍历迭代的时候,因为其中删除只可以删除当前正在遍历的元素,所以就不会出现类似上述那样不可预料的结果;所以在这个删除中我们同步了modcount的值;进而删除不会报错;

源码学习注释文件:

HashMap　LinkedhashMap　HashTable

HashMap 是线程不安全的,jdk1.8之前底层使用的是数组加单向链表实现的,1.8之后使用的是数组+单向链表+树结构,HashMap允许一个null key 允许多个null value

LinkedHashMap 是线程不安全的,但是相对于HashMap其最大的特点就是有序性,遍历输出的时候会按照插入的顺序输出

HashTable 是线程安全的,其相对于HashMap　其中的put remove等操作都是同步方法(synchronize)

TreeMap　这个比较特殊,我们来看一下构造方法:

（1）TreeMap（ ）：构建一个空的映像树
（2）TreeMap（Map m）： 构建一个映像树，并且添加映像m中所有元素
（3）TreeMap（Comparator c）： 构建一个映像树，并且使用特定的比较器对关键字进行排序
（4）TreeMap（SortedMap s）： 构建一个映像树，添加映像树s中所有映射，并且使用与有序映像s相同的比较器排序。

可以发现,构造方法可以是空参的,这样存进来的数据是按照自然顺序存放的;
传入参数Map的那么必须是SortedMap的实现类对象,否则是会报错的;因为需要获取到SortMap中的Comparetor用来作为比较器,这个构造其实就是和最后一个预约;