第七章：深入集合 Collection

7.1 集合框架与 ArrayList

如图 1 所示，是 Java 常见的一些集合框架的类。最上面是 Collection，它有两个子接口，List 和 Set。这些 List 有一些类来实现这个借口，比如说 AbstractList，在 AbstractList 下面还有 Vector Stack，还有 ArrayList 等。但是对于 set 这个接口来说，下面也实现了很多类。最右端是 Map 这个借口，实现它的有 HashMap 等等。

图 1

1.ArrayList

List 借口的可变数组实现。实现了所有可选列表操作，并允许包括 null 在内的所有元素，它是非线程安全。它的底层使用的数据结构是数组，**适合查找修改，弱于删减**。

例：ArrayList 实现分析

```
//用指定的元素替代此列表中指定位置上的元素
//并返回以前位于该位置上的元素
public E set(int index, E element){
    RangeCheck(index);//检查 index 是否合法

    E oldValue = (E) elementData[index];
    elementData[index] = element;
    return oldValue;
}
//将指定的元素添加到此列表的尾部
//直接添加速度快
public boolean add(E e){
    ensureCapacity(size+1);//确保容量满足
    element[size++] = e;
    return true;
}
```

第四行是用来检查 index 是否符合范围要求。下面这一行取出 index 值保存在 oldValue 中，将下标为 index 的值设置为新的，然后把旧的元素值返回。

下面是增加元素的方法。首先确保还有一个额外容量，然后把值赋值给最后一个元素，返回 true 代表增加成功了。

```
//将指定的元素插入此列表中的指定位置。
//如果当前位置有元素，则右移当前位于该位置上的元素
//以及后续所有元素（即将索引加 1）
//涉及数组拷贝，插入速度不及 add 方法
public void add(int index, E element){
    if(index>size)||index <0){
        throw new IndexOutOfBoundsException("Index:"+index+",Size:"+size);
```

```
        }
    //如果数组长度不足，进行扩容
    ensureCapacity(size+1);
    System.arraycopy(elementData, index,elementData,index+1,size-index);
    elementData[index] = element;
    size++;
}
```

首先判断下标 index 是否大于尺寸，或者 index 小于 0，这就说明数组长度不足，需要对数组进行扩容，这个时候就抛出一个错误来。接下来确定数组的容量是能够容纳的。arraycopy 就是把列表中若干值进行拷贝，也就是说所有元素往后拷贝 1 位。然后把值复制给空的那个地方。这样的操作代价还是很大的，所以性能并不好。

```
//数组扩容，1.5 倍方式扩容。涉及数组拷贝，速度慢
public void ensureCapacity(int minCapacity){
    modCount++;
    int oldCapacity = elementData.length;
    if(minCapacity > oldCapacity){
        int newCapacity = (oldCapacity*3)/2+1;
        if(newCapacity < minCapacity)
            newCapacity = minCapacity;
    elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
```

获取旧容量大小给 oldcapacity。如果指定最小的容量比原有还要大，那么先扩大 1.5，如果扩充后这个长度比需要的最小长度还要小，那么直接设置为 mincapacity 的大小。然后把原来列表的数据直接复制给新的数组。

7.2 LinkedList

LinkedList 是 List 借口的链接列表实现，它可以实现所有的可选列表操作，并且允许所有元素（包括 null），它实现 Deque 借口，为 add、poll 提供 FIFO 队列操作以及其他堆栈和双端队列操作。该实现是非线程安全的。当我们在编程序时，如果用到这个链表，并且是多线程去访问这个链表的时候，需要自己加 XX（没听清），LinkedList 不做同步与互斥来控制，需要你自己写程序来解决这个问题。**它适合增减，弱于查改**。

例：LinkedList 数据结构

```
private transient Entry<E> header = new Entry<E>(null,null,null);
private static class Entry<E>{
    E element;
    Entry<E> next;
    Entry<E> previous;
    /***
    *构造方法：目标对象 paramE 将被防止在 paramEntry1 之前，
    *paramEntry2 之后
    */
    Entry(E paramE, Entry<E>paramEntry1,Entry<E>paramEntry2){
```

```
            this.element = paramE;
            this.next = paramEntry1;
            this.previous = paramEntry2;
        }
}
```

header 是链表的头，三个空在后面有指。这个结点中包含三个元素，element 是值，next 是当前节点的下一个节点是什么，previous 是前一根节点是什么。后面的构造方法指定了这三个元素的内容。

```
//根据序号获取 Entry 对象
private Entry<E> entry(int paramint){
    if((paramint<0)||(paramint>=this.size)){
        throw new indexOutOfBoundsException("Index:"+paramint+",size:"+this.size);
    }
    Entry localEntry = this.header;
    int i;
    //最多遍历 size/2 个元素
    if(paramint<this.size>>1){
        for(i = 0; i<=paramint;i++)
            localEntry = localEntry.next;
    }else{
        for(i=this.size;i>paramint;i--)
            localEntry = localEntry.previous;
    }
    return localEntry;
}
```

这个方法是用来根据给定序号获取这个节点的值。首先判断序号是否越界，若越界则抛出异常类，否则继续进行。然后先设置链接到链表的头部，也就是从头部开始进行遍历。然后开始判断所给定的序号比列表一半的长度小还是长度达。若比长度右移一位（this.size>>1 代表右移一位，实际上也就代表着长度除以 2，这是我们在学二进制的时候就学过的）小，那么就从表头开始，一步一步往下找，直到找到我们需要的序号的元素，然后输出值。若大于，则同理。

```
/***
*要添加的元素：paramE
*目标对象：paramEntryEntry
*特点：插入速度快
*/
private Entry<E> addBefore(E paramE, Entry<E> paramEntry){
    //要添加的对象，设置其 previous 和 next
    Entry localEntry = new Entry(paramE, paramEntry, paramEntry.previous)

    localEntry.previous.next=localEntry;
    localEntry.next.previous = localEntry;

    this.size+=1;
```

```
        this.modCount+=1;
        return localEntry;
}
```

实例化一个新的节点，这个时候我们必须指定这个结点放在哪里。

```
//指定位置添加元素，需要先找到 index 的元素，然后添加
public void add(int index, E element){
        addBefore(element,(index==size?header：entry（index）））；
}
//队首添加元素
public void addFirst(E paramE){
        addBefore(paramE,this.header.next);
}
//队尾添加元素
public void addLast(E paramE){
        addBefore(paramE, this.header);
}

public E remove()
{
        return removeFirst();
}//删除第一个
public E remove(int index){
        return remove(entry(index));
}//删除第 index 个元素
public E removeFirst(){
        return remove(header.next);
}
public E removeLast(){
        return remove(header.previous);
}//删除最后一个
```

List 的适用范围

（1）ArrayList 适用于对于数据查询修改大于数据增加删除的场合；

（2）LinkedList 适用于对于数据增删大于数据查询的场合。

7.3 HashMap 与 HashTable

1.HashMap（哈希映射）

它是基于哈希表的 Map 借口的实现，提供所有可选的映射操作，并允许使用 null 值和 null 键。它是非线性安全的。它不保证映射的顺序，特别是不保证顺序恒久不变的。HashMap 的数据结构如图 2 所示。

图 2

代码描述版：

```
transient Entry[] table;
```

```
static class Entry<K,V> implements Map.Entry<K,V>{
    final K key;
    V value;
    Entry<K,V> next;
    final int hash;
    //……剩余代码
}
```

第一行就是设置一个数组，类型是 Entry。第二行说明这个 Entry。后面说明是 Entry 是一个列表等等。

总结来说，HashMap 底层就是一个数组结构，数组中的每一项又是一个链表，先建一个的时候就会初始化这么一个数组，Entry 就是数组中的元素，那么每个 Map Entry 就是一个 K value 对，它只有一个向下的指针来指示我们下一个元素，这就构成了链表。

例：如何向 HashMap 中添加元素

```
public V put(K key, V value){
    if(key==null)//null 键视为相同的键
        return putForNullKey(value);//根据 key 的 keycode 重新计算 hash 值
    int hash = hash(key.hashCode());//搜索指定 hash 值在对应 table 中索引
    int i = indexFor(hash,table.length);
//如果 i 索引处的 Entry 不为 null，通过循环不断遍历 e 元素的下一个元素
    for(Entry<K,V>e=table[i];e!=null;e=e.next){
        Object k;//key 重复出现则更新其 value
        if(e.hash ==hash&&((k=e.key)==key||key.equals(k))){
            V oldValue = e.value;
            e.value =value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    addEntry(hash,key,value,i);
    return null;
}
```

例：hash 函数，加入高位计算，防止地位不变，高位变化时，造成 hash 冲突

代码：

```
static int hash(int h){
    h^=(h>>>20)^(h>>>12);
    return h^(h>>>7)^(h>>>4);
}

void addEntry(int hash, K key, V value, int bucketIndex){
    //获取指定 bucketIndex 索引处的 Entry
    Entry<K,V>e=table[bucketIndex];
    //将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry
```

```
        table[bucketIndex] = new Entry<K,V>(hash,key,value,e);
        //如果 Map 中的 key-value 对的数量超过了极限
        if(size++>=threshold)
            //把 table 对象的长度扩充到原来的 2 倍
            resize(2*table.length);
}
//根据 hash 值查找对应的 table 位置
static int indexFor(int h, int length){
        return h&(length-1);
}
```

例：HashMap 的 get 操作

```
public V get(Object key){
        if(key==null)
            return getForNullKey();
        int hash =hash(key.hashCode());
        for(Entry<K,V>e = table[indexFor(hash,table.length)];
        e!=null;
        e=e.next){
            Object k;
            if(e.hash==hash&&((k=e.key)==key||key.equals(k)))
                return e.value;
        }
        return null;
}
```

2.HashTable（哈希表）

HashTable 和 HashMap 采用相同的存储机制，二者的实现基本一致。但是 HashTable **不允许 NULL 存在**。HashTable 是**线程安全**的，内部的方法基本都是 synchronized。它的迭代器具有强一致性。

7.4 TreeMap 与 LinkedHashMap

1.TreeMap

它实际上是 Map 接口的树实现，不允许 null 值存在，非线性安全，键值有序。它的数据结构实际是采用了红黑二叉树，是一个自平衡二叉查找树，在进行插入和删除操作时，通过特定的操作能够保持二叉查找树的平衡，从而获得较高的查找性能。它的优点是做查找、插入和删除操作的时间是 O(logn)。

如图 3 所示，Entry 是红黑树的结点，它包含了红黑树的 6 个基本组成成分：key、value、left（左孩子）、right（右孩子）、parent、color。Entry 节点根据 key 进行排序，本身 entry 节点包含的内容为 value。红黑树排序时，根据 entry 中的 key 进行排序，entry 中的 key 比较大小是根据比较器 comparator 来进行判断的。

图 3

TreeMap 的优点：

（1）空间利用率高：HashMap 的数组大小必须为 2 的 n 次方；TreeMap 中树的每一个

节点就代表了一个元素。

（2）性能稳定：Hash 碰撞会导致 HashMap 查询开销提高；HashMap 扩容时会 rehash，开销高；TreeMap 的操作均能在 O(logn)内完成。

2.LinkedHashMap

Map 接口的哈希表和链接列表实现，提供所有可选的映射操作，并允许使用 null 值和 null 键，非线程安全，具有可预知的迭代顺序。

例：LinkedHashMap 实现分析

如图 4 所示。

图 4

HashMap 第一级是数组，第二级是单向链表；而这个第二级结构是一个双向链表。

3.Map 的适用范围

（1）HashMap 适用于一般的键值映射需求；

（2）HashTable 适用于有多线程并发的场合；

（3）TreeMap 适用于要按照键排序的迭代场合；

（4）LInkedHashMap 适用于特殊顺序的迭代场合（如 LRU 算法）。

7.5 HashSet

1.HashSet

它实现 Set 借口，由哈希表支持，**允许使用 null 元素**。非线程安全，不保证 set 的迭代顺序，特别是不保证该顺序恒久不变。

例：HashSet 的实现分析

图 5

2.Set 的特点

（1）HashSet 通过 HashMap 实现；

（2）TreeSet 通过 TreeMap 实现；

（3）LinkedHashSet 通过 LinkedHashMap 实现；

（4）Set 类与 Map 类拥有近似的使用特性。