[**Java集合类操作优化经验总结**](http://www.codeceo.com/article/java-collection-class.html)

2016-03-13    分类：[JAVA开发](http://www.codeceo.com/article/category/develop/java)、[编程开发](http://www.codeceo.com/article/category/develop)、[首页精华](http://www.codeceo.com/article/category/pick)[暂无人评论](http://www.codeceo.com/article/java-collection-class.html#comments)     来源：[developerworks](http://www.ibm.com/developerworks/cn/java/j-lo-set-operation/index.html)

分享到：更多10

本文首先针对 Java 集合接口进行了一些介绍，并对这些接口的实现类进行详细描述，包括 LinkedList、ArrayList、Vector、Stack、Hashtable、HashMap、[WeakHashMap](http://www.codeceo.com/article/java-weakhashmap-source.html) 等，然后对一些实现类的实现方式和使用经验进行讲解，同时重点介绍 WeakHashMap。希望通过本文介绍，可以让读者对集合的操作方式、注意事项等有一些了解。

在实际的项目开发中会有很多的对象，如何高效、方便地管理对象，成为影响程序性能与可维护性的重要环节。Java 提供了集合框架来解决此类问题，线性表、链表、哈希表等是常用的数据结构，在进行 Java 开发时，JDK 已经为我们提供了一系列相应的类来实现基本的数据结构，所有类都在 java.util 这个包里，清单 1 描述了集合类的关系。

**清单 1.集合类之间关系**

Collection

├List

│├LinkedList

│├ArrayList

│└Vector

│　└Stack

└Set

Map

├Hashtable

├HashMap

└WeakHashMap

本文讲的就是集合框架的使用经验总结，注意，本文所有代码基于 JDK7。

**集合接口**

**Collection 接口**

Collection 是最基本的集合接口，一个 Collection 代表一组 Object，即 Collection 的元素（Elements）。一些 Collection 允许相同的元素、支持对元素进行排序，另一些则不行。JDK 不提供直接继承自 Collection 的类，JDK 提供的类都是继承自 Collection 的子接口，如 List 和 Set。所有实现 Collection 接口的类都必须提供两个标准的构造函数，无参数的构造函数用于创建一个空的 Collection，有一个 Collection 参数的构造函数用于创建一个新的 Collection，这个新的 Collection 与传入的 Collection 有相同的元素，后一个构造函数允许用户复制一个 Collection。

**如何遍历 Collection 中的每一个元素？**

不论 Collection 的实际类型如何，它都支持一个 iterator() 的方法，该方法返回一个迭代子，使用该迭代子即可逐一访问 Collection 中每一个元素。典型的用法如下：

Iterator it = collection.iterator(); // 获得一个迭代子

while(it.hasNext()){

Object obj = it.next(); // 得到下一个元素

}

Collection 接口派生的两个接口是 List 和 Set。

Collection 接口提供的主要方法：

1. boolean add(Object o) 添加对象到集合；
2. boolean remove(Object o) 删除指定的对象；
3. int size() 返回当前集合中元素的数量；
4. boolean contains(Object o) 查找集合中是否有指定的对象；
5. boolean isEmpty() 判断集合是否为空；
6. Iterator iterator() 返回一个迭代器；
7. boolean containsAll(Collection c) 查找集合中是否有集合 C 中的元素；
8. boolean addAll(Collection c) 将集合 C 中所有的元素添加给该集合；
9. void clear() 删除集合中所有元素；
10. void removeAll(Collection c) 从集合中删除 C 集合中也有的元素；
11. void retainAll(Collection c) 从集合中删除集合 C 中不包含的元素。

**List 接口**

List 是有序的 Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（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。和下文要提到的 Set 不同，List 允许有相同的元素。

除了具有 Collection 接口必备的 iterator() 方法外，List 还提供一个 listIterator() 方法，返回一个 ListIterator 接口。和标准的 Iterator 接口相比，ListIterator 多了一些 add() 之类的方法，允许添加、删除、设定元素、向前或向后遍历等功能。实现 List 接口的常用类有 LinkedList，ArrayList，Vector 和 Stack 等。

List 接口提供的主要方法：

1. void add(int index,Object element) 在指定位置上添加一个对象；
2. boolean addAll(int index,Collection c) 将集合 C 的元素添加到指定的位置；
3. Object get(int index) 返回 List 中指定位置的元素；
4. int indexOf(Object o) 返回第一个出现元素 O 的位置；
5. Object removeint(int index) 删除指定位置的元素；
6. Object set(int index,Object element) 用元素 element 取代位置 index 上的元素, 返回被取代的元素。

**Map 接口**

Map 没有继承 Collection 接口。Map 提供 Key 到 Value 的映射，一个 Map 中不能包含相同的 Key，每个 Key 只能映射一个 Value。Map 接口提供 3 种集合的视图，Map 的内容可以被当作一组 Key 集合，一组 Value 集合，或者一组 Key-Value 映射。

Map 提供的主要方法：

1. boolean equals(Object o) 比较对象；
2. boolean remove(Object o) 删除一个对象；
3. put(Object key,Object value) 添加 key 和 value。

**RandomAccess 接口**

RandomAccess 接口是一个标志接口，本身并没有提供任何方法，任务凡是通过调用 RandomAccess 接口的对象都可以认为是支持快速随机访问的对象。此接口的主要目的是标识那些可支持快速随机访问的 List 实现。任何一个基于数组的 List 实现都实现了 RaodomAccess 接口，而基于链表的实现则都没有。因为只有数组能够进行快速的随机访问，而对链表的随机访问需要进行链表的遍历。因此，此接口的好处是，可以在应用程序中知道正在处理的 List 对象是否可以进行快速随机访问，从而针对不同的 List 进行不同的操作，以提高程序的性能。

**集合类介绍**

**LinkedList 类**

LinkedList 实现了 List 接口，允许 Null 元素。此外 LinkedList 提供额外的 Get、Remove、Insert 等方法在 LinkedList 的首部或尾部操作数据。这些操作使得 LinkedList 可被用作堆栈（Stack）、队列（Queue）或双向队列（Deque）。请注意 LinkedList 没有同步方法，它不是线程同步的，即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，方法如

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

**ArrayList 类**

ArrayList 实现了可变大小的数组。它允许所有元素，包括 Null。Size、IsEmpty、Get、Set 等方法的运行时间为常数，但是 Add 方法开销为分摊的常数，添加 N 个元素需要 O(N) 的时间，其他的方法运行时间为线性。

每个 ArrayList 实例都有一个容量（Capacity），用于存储元素的数组的大小，这个容量可随着不断添加新元素而自动增加。当需要插入大量元素时，在插入前可以调用 ensureCapacity 方法来增加 ArrayList 的容量以提高插入效率。和 LinkedList 一样，ArrayList 也是线程非同步的（unsynchronized）。

ArrayList 提供的主要方法：

1. Boolean add(Object o) 将指定元素添加到列表的末尾；
2. Boolean add(int index,Object element) 在列表中指定位置加入指定元素；
3. Boolean addAll(Collection c) 将指定集合添加到列表末尾；
4. Boolean addAll(int index,Collection c) 在列表中指定位置加入指定集合；
5. Boolean clear() 删除列表中所有元素；
6. Boolean clone() 返回该列表实例的一个拷贝；
7. Boolean contains(Object o) 判断列表中是否包含元素；
8. Boolean ensureCapacity(int m) 增加列表的容量，如果必须，该列表能够容纳 m 个元素；
9. Object get(int index) 返回列表中指定位置的元素；
10. Int indexOf(Object elem) 在列表中查找指定元素的下标；
11. Int size() 返回当前列表的元素个数。

**Vector 类**

Vector 非常类似于 ArrayList，区别是 Vector 是线程同步的。由 Vector 创建的 Iterator，虽然和 ArrayList 创建的 Iterator 是同一接口，但是，因为 Vector 是同步的，当一个 Iterator 被创建而且正在被使用，另一个线程改变了 Vector 的状态（例如，添加或删除了一些元素），这时调用 Iterator 的方法时将抛出 ConcurrentModificationException，因此必须捕获该异常。

**Stack 类**

Stack 继承自 Vector，实现了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。除了基本的 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。注意，Stack 刚创建后是空栈。

**Set 类**

Set 是一种不包含重复的元素的 Collection，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。很明显，Set 的构造函数有一个约束条件，传入的 Collection 参数不能包含重复的元素。请注意，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，这可能会导致一些问题。

**Hashtable 类**

Hashtable 继承 Map 接口，实现了一个基于 Key-Value 映射的哈希表。任何非空（non-null）的对象都可作为 Key 或者 Value。添加数据使用 Put(Key，Value)，取出数据使用 Get(Key)，这两个基本操作的时间开销为常数。

Hashtable 通过 Initial Capacity 和 Load Factor 两个参数调整性能。通常缺省的 Load Factor 0.75 较好地实现了时间和空间的均衡。增大 Load Factor 可以节省空间但相应的查找时间将增大，会影响像 Get 和 Put 这样的操作。使用 Hashtable 的简单示例，将 1、2、3 这三个数字放到 Hashtable 里面，他们的 Key 分别是”one”、”two”、”three”，代码如清单 2 所示。

**清单 2 .Hashtable 示例**

Hashtable numbers = new Hashtable();

numbers.put(“one”, new Integer(1));

numbers.put(“two”, new Integer(2));

numbers.put(“three”, new Integer(3));

如果我们需要取出一个数，比如 2，可以用相应的 key 来取出，代码如清单 3 所示。

**清单 3.从 Hastable 读取数据**

Integer n = (Integer)numbers.get(“two”);

System.out.println(“two =”+ n);

由于作为 Key 的对象将通过计算其散列函数来确定与之对应的 Value 的位置，因此任何作为 key 的对象都必须实现 HashCode 和 Equals 方法。HashCode 和 Equals 方法继承自根类 Object，如果你用自定义的类当作 Key 的话，要相当小心，按照散列函数的定义，如果两个对象相同，即 obj1.equals(obj2)=true，则它们的 HashCode 必须相同，但如果两个对象不同，则它们的 HashCode 不一定不同，如果两个不同对象的 HashCode 相同，这种现象称为冲突，冲突会导致操作哈希表的时间开销增大，所以尽量定义好的 HashCode() 方法，能加快哈希表的操作。

如果相同的对象有不同的 HashCode，对哈希表的操作会出现意想不到的结果（期待的 Get 方法返回 Null），要避免这种问题，最好同时复写 Equals 方法和 HashCode 方法，而不要只写其中一个。

**HashMap 类**

HashMap 和 Hashtable 类似，不同之处在于 HashMap 是线程非同步的，并且允许 Null，即 Null Value 和 Null Key。但是将 HashMap 视为 Collection 时（values() 方法可返回 Collection），其迭代子操作时间开销和 HashMap 的容量成比例。因此，如果迭代操作的性能相当重要的话，不要将 HashMap 的初始化容量设得过高，或者 Load Factor 参数设置过低。

**WeakHashMap 类**

WeakHashMap 是一种改进的 HashMap，它对 Key 实行“弱引用”，如果一个 Key 不再被外部所引用，那么该 Key 可以被 GC 回收。

**集合类实践**

ArrayList、Vector、LinkedList 均来自 AbstractList 的实现，而 AbstractList 直接实现了 List 接口，并扩展自 AbstarctCollection。ArrayList 和 Vector 使用了数组实现，ArrayList 没有对任何一个方法提供线程同步，因此不是线程安全的，Vector 中绝大部分方法都做了线程同步，是一种线程安全的实现。LinkedList 使用了循环双向链表数据结构，由一系列表项连接而成，一个表项总是包含 3 个部分，元素内容、前驱表项和后驱表项。

当 ArrayList 对容量的需求超过当前数组的大小时，需要进行扩容。扩容过程中，会进行大量的数组复制操作，而数组复制时，最终将调用 System.arraycopy() 方法。LinkedList 由于使用了链表的结构，因此不需要维护容量的大小，然而每次的元素增加都需要新建一个 Entry 对象，并进行更多的赋值操作，在频繁的系统调用下，对性能会产生一定的影响，在不间断地生成新的对象还是占用了一定的资源。而因为数组的连续性，因此总是在尾端增加元素时，只有在空间不足时才产生数组扩容和数组复制。

ArrayList 是基于数组实现的，而数组是一块连续的内存空间，如果在数组的任意位置插入元素，必然导致在该位置后的所有元素需要重新排列，因此其效率较差，尽可能将数据插入到尾部。LinkedList 不会因为插入数据导致性能下降。

ArrayList 的每一次有效的元素删除操作后都要进行数组的重组，并且删除的元素位置越靠前，数组重组时的开销越大，要删除的元素位置越靠后，开销越小。LinkedList 要移除中间的数据需要便利完半个 List。

**清单 4. ArrayList 和 LinkedList 使用代码**

import java.util.ArrayList;

import java.util.LinkedList;

public class ArrayListandLinkedList {

public static void main(String[] args){

long start = System.currentTimeMillis();

ArrayList list = new ArrayList();

Object obj = new Object();

for(int i=0;i<5000000;i++){

list.add(obj);

}

long end = System.currentTimeMillis();

System.out.println(end-start);

start = System.currentTimeMillis();

LinkedList list1 = new LinkedList();

Object obj1 = new Object();

for(int i=0;i<5000000;i++){

list1.add(obj1);

}

end = System.currentTimeMillis();

System.out.println(end-start);

start = System.currentTimeMillis();

Object obj2 = new Object();

for(int i=0;i<1000;i++){

list.add(0,obj2);

}

end = System.currentTimeMillis();

System.out.println(end-start);

start = System.currentTimeMillis();

Object obj3 = new Object();

for(int i=0;i<1000;i++){

list1.add(obj1);

}

end = System.currentTimeMillis();

System.out.println(end-start);

start = System.currentTimeMillis();

list.remove(0);

end = System.currentTimeMillis();

System.out.println(end-start);

start = System.currentTimeMillis();

list1.remove(250000);

end = System.currentTimeMillis();

System.out.println(end-start);

}

}

**清单 5. 运行输出**

639

1296

6969

0

0

15

HashMap 是将 Key 做 Hash 算法，然后将 Hash 值映射到内存地址，直接取得 Key 所对应的数据。在 HashMap 中，底层数据结构使用的是数组，所谓的内存地址即数组的下标索引。HashMap 的高性能需要保证以下几点：

1. Hash 算法必须是高效的；
2. Hash 值到内存地址 (数组索引) 的算法是快速的；
3. 根据内存地址 (数组索引) 可以直接取得对应的值。

HashMap 实际上是一个链表的数组。前面已经介绍过，基于 HashMap 的链表方式实现机制，只要 HashCode() 和 Hash() 方法实现得足够好，能够尽可能地减少冲突的产生，那么对 HashMap 的操作几乎等价于对数组的随机访问操作，具有很好的性能。但是，如果 HashCode() 或者 Hash() 方法实现较差，在大量冲突产生的情况下，HashMap 事实上就退化为几个链表，对 HashMap 的操作等价于遍历链表，此时性能很差。

HashMap 的一个功能缺点是它的无序性，被存入到 HashMap 中的元素，在遍历 HashMap 时，其输出是无序的。如果希望元素保持输入的顺序，可以使用 LinkedHashMap 替代。

LinkedHashMap 继承自 HashMap，具有高效性，同时在 HashMap 的基础上，又在内部增加了一个链表，用以存放元素的顺序。

HashMap 通过 hash 算法可以最快速地进行 Put() 和 Get() 操作。TreeMap 则提供了一种完全不同的 Map 实现。从功能上讲，TreeMap 有着比 HashMap 更为强大的功能，它实现了 SortedMap 接口，这意味着它可以对元素进行排序。TreeMap 的性能略微低于 HashMap。如果在开发中需要对元素进行排序，那么使用 HashMap 便无法实现这种功能，使用 TreeMap 的迭代输出将会以元素顺序进行。LinkedHashMap 是基于元素进入集合的顺序或者被访问的先后顺序排序，TreeMap 则是基于元素的固有顺序 (由 Comparator 或者 Comparable 确定)。

LinkedHashMap 是根据元素增加或者访问的先后顺序进行排序，而 TreeMap 则根据元素的 Key 进行排序。

清单 6 所示代码演示了使用 TreeMap 实现业务逻辑的排序。

**清单 6. TreeMap 实现排序**

import java.util.Iterator;

import java.util.Map;

import java.util.TreeMap;

public class Student implements Comparable<Student>{

public String name;

public int score;

public Student(String name,int score){

this.name = name;

this.score = score;

}

@Override

//告诉 TreeMap 如何排序

public int compareTo(Student o) {

// TODO Auto-generated method stub

if(o.score<this.score){

return 1;

}else if(o.score>this.score){

return -1;

}

return 0;

}

@Override

public String toString(){

StringBuffer sb = new StringBuffer();

sb.append("name:");

sb.append(name);

sb.append(" ");

sb.append("score:");

sb.append(score);

return sb.toString();

}

public static void main(String[] args){

TreeMap map = new TreeMap();

Student s1 = new Student("1",100);

Student s2 = new Student("2",99);

Student s3 = new Student("3",97);

Student s4 = new Student("4",91);

map.put(s1, new StudentDetailInfo(s1));

map.put(s2, new StudentDetailInfo(s2));

map.put(s3, new StudentDetailInfo(s3));

map.put(s4, new StudentDetailInfo(s4));

//打印分数位于 S4 和 S2 之间的人

Map map1=((TreeMap)map).subMap(s4, s2);

for(Iterator iterator=map1.keySet().iterator();iterator.hasNext();){

Student key = (Student)iterator.next();

System.out.println(key+"->"+map.get(key));

}

System.out.println("subMap end");

//打印分数比 s1 低的人

map1=((TreeMap)map).headMap(s1);

for(Iterator iterator=map1.keySet().iterator();iterator.hasNext();){

Student key = (Student)iterator.next();

System.out.println(key+"->"+map.get(key));

}

System.out.println("subMap end");

//打印分数比 s1 高的人

map1=((TreeMap)map).tailMap(s1);

for(Iterator iterator=map1.keySet().iterator();iterator.hasNext();){

Student key = (Student)iterator.next();

System.out.println(key+"->"+map.get(key));

}

System.out.println("subMap end");

}

}

class StudentDetailInfo{

Student s;

public StudentDetailInfo(Student s){

this.s = s;

}

@Override

public String toString(){

return s.name + "'s detail information";

}

}

**清单 7 .运行输出**

name:4 score:91->4's detail information

name:3 score:97->3's detail information

subMap end

name:4 score:91->4's detail information

name:3 score:97->3's detail information

name:2 score:99->2's detail information

subMap end

name:1 score:100->1's detail information

subMap end

WeakHashMap 特点是当除了自身有对 Key 的引用外，如果此 Key 没有其他引用，那么此 Map 会自动丢弃该值。如清单 8 所示代码声明了两个 Map 对象，一个是 HashMap，一个是 WeakHashMap，同时向两个 map 中放入 A、B 两个对象，当 HashMap 删除 A，并且 A、B 都指向 Null 时，WeakHashMap 中的 A 将自动被回收掉。出现这个状况的原因是，对于 A 对象而言，当 HashMap 删除并且将 A 指向 Null 后，除了 WeakHashMap 中还保存 A 外已经没有指向 A 的指针了，所以 WeakHashMap 会自动舍弃掉 a，而对于 B 对象虽然指向了 null，但 HashMap 中还有指向 B 的指针，所以 WeakHashMap 将会保留 B 对象。

**清单 8.WeakHashMap 示例代码**

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import java.util.WeakHashMap;

public class WeakHashMapTest {

public static void main(String[] args) throws Exception {

String a = new String("a");

String b = new String("b");

Map weakmap = new WeakHashMap();

Map map = new HashMap();

map.put(a, "aaa");

map.put(b, "bbb");

weakmap.put(a, "aaa");

weakmap.put(b, "bbb");

map.remove(a);

a=null;

b=null;

System.gc();

Iterator i = map.entrySet().iterator();

while (i.hasNext()) {

Map.Entry en = (Map.Entry)i.next();

System.out.println("map:"+en.getKey()+":"+en.getValue());

}

Iterator j = weakmap.entrySet().iterator();

while (j.hasNext()) {

Map.Entry en = (Map.Entry)j.next();

System.out.println("weakmap:"+en.getKey()+":"+en.getValue());

}

}

}

**清单 9 .运行输出**

map:b:bbb

weakmap:b:bbb

WeakHashMap 主要通过 expungeStaleEntries 这个函数来实现移除其内部不用的条目，从而达到自动释放内存的目的。基本上只要对 WeakHashMap 的内容进行访问就会调用这个函数，从而达到清除其内部不再为外部引用的条目。但是如果预先生成了 WeakHashMap，而在 GC 以前又不曾访问该 WeakHashMap, 那不是就不能释放内存了吗？

**清单 10. WeakHashMapTest1**

import java.util.ArrayList;

import java.util.List;

import java.util.WeakHashMap;

public class WeakHashMapTest1 {

public static void main(String[] args) throws Exception {

List<WeakHashMap<byte[][], byte[][]>> maps = new ArrayList<WeakHashMap<byte[][], byte[][]>>();

for (int i = 0; i < 1000; i++) {

WeakHashMap<byte[][], byte[][]> d = new WeakHashMap<byte[][], byte[][]>();

d.put(new byte[1000][1000], new byte[1000][1000]);

maps.add(d);

System.gc();

System.err.println(i);

}

}

}

不改变任何 JVM 参数的情况运行清单 10 所示代码，由于 Java 默认内存是 64M，抛出内存溢出了错误。

**清单 11. 运行输出**

241

242

243

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

at WeakHashMapTest1.main(WeakHashMapTest1.java:10)

果不其然，WeakHashMap 这个时候并没有自动帮我们释放不用的内存。清单 12 所示代码不会出现内存溢出问题。

**清单 12. WeakHashMapTest2**

import java.util.ArrayList;

import java.util.List;

import java.util.WeakHashMap;

public class WeakHashMapTest2 {

public static void main(String[] args) throws Exception {

List<WeakHashMap<byte[][], byte[][]>> maps = new ArrayList<WeakHashMap<byte[][], byte[][]>>();

for (int i = 0; i < 1000; i++) {

WeakHashMap<byte[][], byte[][]> d = new WeakHashMap<byte[][], byte[][]>();

d.put(new byte[1000][1000], new byte[1000][1000]);

maps.add(d);

System.gc();

System.err.println(i);

for (int j = 0; j < i; j++) {

System.err.println(j + " size" + maps.get(j).size());

}

}

}

}

运行结果发现这次测试输出正常, 不再出现内存溢出问题。

总的来说，WeakHashMap 并不是你什么也干它就能自动释放内部不用的对象的，而是在你访问它的内容的时候释放内部不用的对象。

WeakHashMap 实现弱引用，是因为它的 Entry<K,V>是继承自 WeakReference<K>的，

在 WeakHashMap$Entry<K,V>的类定义及构造函数里面如清单 13 所示。

**清单 13. WeakHashMap 类定义**

private static class Entry<K,V> extends WeakReference<K>

implements Map.Entry<K,V> Entry(K key, V value, ReferenceQueue<K> queue,int hash, Entry<K,V> next) {

super(key, queue);

this.value = value;

this.hash = hash;

this.next = next;

}

请注意它构造父类的语句：“super(key, queue);”，传入的是 Key，因此 Key 才是进行弱引用的，Value 是直接强引用关联在 this.value 之中。在 System.gc() 时，Key 中的 Byte 数组进行了回收，而 Value 依然保持 (Value 被强关联到 Entry 上，Entry 又关联在 Map 中，Map 关联在 ArrayList 中)。

For 循环中每次都 New 一个新的 WeakHashMap，在 Put 操作后，虽然 GC 将 WeakReference 的 Key 中的 Byte 数组回收了，并将事件通知到了 ReferenceQueue，但后续却没有相应的动作去触发 WeakHashMap 去处理 ReferenceQueue，所以 WeakReference 包装 Key 依然存在于 WeakHashMap 中，其对应的 value 也当然存在。

那 value 是何时被清除的呢? 对清单 10 和清单 11 两个示例程序进行分析可知，清单 11 的 maps.get(j).size() 触发了 Value 的回收，那又如何触发的呢？查看 WeakHashMap 源码可知,Size 方法调用了 expungeStaleEntries 方法，该方法对 JVM 要回收的的 Entry(Quene 中) 进行遍历，并将 Entry 的 Value 置空，回收了内存。所以效果是 Key 在 GC 的时候被清除，Value 在 Key 清除后访问 WeakHashMap 被清除。

WeakHashMap 类是线程不同步的，可以使用 Collections.synchronizedMap 方法来构造同步的 WeakHashMap, 每个键对象间接地存储为一个弱引用的指示对象。因此，不管是在映射内还是在映射之外，只有在垃圾回收器清除某个键的弱引用之后，该键才会自动移除。需要注意的是，WeakHashMap 中的值对象由普通的强引用保持。因此应该小心谨慎，确保值对象不会直接或间接地强引用其自身的键，因为这会阻止键的丢弃。注意，值对象可以通过 WeakHashMap 本身间接引用其对应的键，这就是说，某个值对象可能强引用某个其他的键对象，而与该键对象相关联的值对象转而强引用第一个值对象的键。

处理此问题的一种方法是，在插入前将值自身包装在 WeakReferences 中，如：m.put(key, new WeakReference(value))，然后，分别用 get 进行解包，该类所有“collection 视图方法”返回的迭代器均是快速失败的，在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器自身的 Remove 或 Add 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就完全失败，而不是冒着在将来不确定的时间任意发生不确定行为的风险。

注意，我们不能确保迭代器不失败，一般来说，存在不同步的并发修改时，不可能做出任何完全确定的保证。

**总结**

综合前面的介绍和实例代码，我们可以知道，如果涉及到堆栈、队列等操作，应该考虑用 List。对于需要快速插入、删除元素等操作，应该使用 LinkedList。如果需要快速随机访问元素，应该使用 ArrayList。如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高。如果多个线程可能同时操作一个类，应该使用同步的类。要特别注意对哈希表的操作，作为 Key 的对象要正确复写 Equals 和 HashCode 方法。尽量返回接口而非实际的类型，如返回 List 而非 ArrayList，这样如果以后需要将 ArrayList 换成 LinkedList 时，客户端代码不用改变，这就是针对抽象进行编程思想。

本文只是针对应用层面的分享，后续文章会针对具体源代码级别的实现进行深入介绍，也会对具体实现所基于的算法进行深入介绍，请有需要的读者关注后续文章。