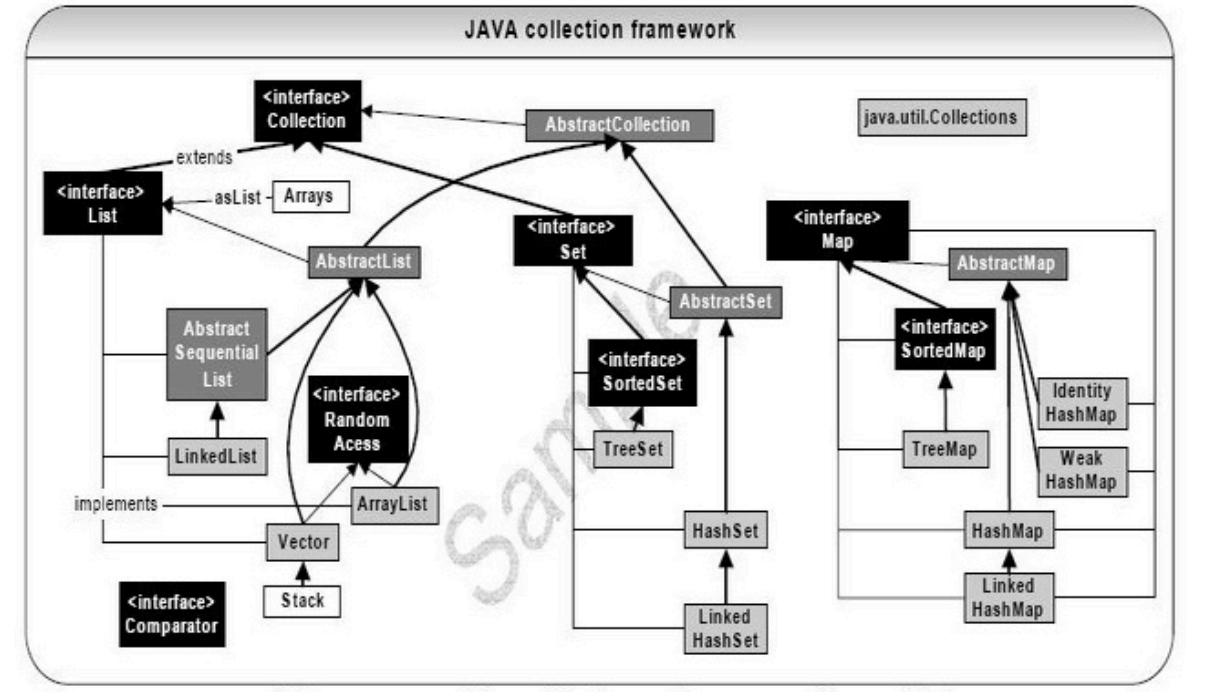
03_Java集合面试题

Java集合类关系图



1.Java的HashMap是如何工作的?

HashMap是一个针对数据结构的键值，每个键都会有相应的值，关键是识别这样的值。

HashMap 基于 hashing 原理，我们通过 put ()和 get ()方法储存和获取对象。当我们将键值对传递给 put ()方法时，它调用键对象的 hashCode ()方法来计算 hashcode，让后找到 bucket 位置来储存值对象。当获取对象时，通过键对象的 equals ()方法找到正确的键值对，然后返回值对象。HashMap 使用 LinkedList 来解决碰撞问题，当发生碰撞了，对象将会储存在 LinkedList 的下一个节点中。HashMap 在每个 LinkedList 节点中储存键值对对象。

2.什么是快速失败的故障安全迭代器?

快速失败的Java迭代器可能会引发ConcurrentModifcationException在底层集合迭代过程中被修改。故障安全作为发生在实例中的一个副本迭代是不会抛出任何

异常的。快速失败的故障安全范例定义了当遭遇故障时系统是如何反应的。例如，用于失败的快速迭代器ArrayList和用于故障安全的迭代器ConcurrentHashMap。

3.Java BlockingQueue是什么?

Java BlockingQueue是一个并发集合util包的一部分。BlockingQueue队列是一种支持操作，它等待元素变得可用时来检索，同样等待空间可用时来存储元素。

4.什么时候使用ConcurrentHashMap?

在问题2中我们看到ConcurrentHashMap被作为故障安全迭代器的一个实例，它允许完整的并发检索和更新。当有大量的并发更新时，ConcurrentHashMap此时可以被使用。这非常类似于Hashtable，但ConcurrentHashMap不锁定整个表来提供并发，所以从这点上ConcurrentHashMap的性能似乎更好一些。所以当有大量更新时ConcurrentHashMap应该被使用。

5.哪一个List实现了最快插入?

LinkedList和ArrayList是另个不同变量列表的实现。ArrayList的优势在于动态的增长数组，非常适合初始时总长度未知的情况下使用。LinkedList的优势在于在中间位置插入和删除操作，速度是最快的。

LinkedList实现了List接口，允许null元素。此外LinkedList提供额外的get，remove，insert方法在LinkedList的首部或尾部。这些操作使LinkedList可被用作堆栈（stack），队列（queue）或双向队列（deque）。

ArrayList实现了可变大小的数组。它允许所有元素，包括null。 每个ArrayList实例都有一个容量（Capacity），即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，但是增长算法并没有定义。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。

6.Iterator和ListIterator的区别?

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

2️⃣.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。

3️⃣.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。

4️⃣.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。lierator仅能遍历，不能修改。

7.什么是CopyOnWriteArrayList，它与ArrayList有何不同?

CopyOnWriteArrayList是ArrayList的一个线程安全的变体，其中所有可变操作（add、set等等）都是通过对底层数组进行一次新的复制来实现的。相比较于ArrayList它的写操作要慢一些，因为它需要实例的快照。

CopyOnWriteArrayList中写操作需要大面积复制数组，所以性能肯定很差，但是读操作因为操作的对象和写操作不是同一个对象，读之间也不需要加锁，读和写之间的同步处理只是在写完后通过一个简单的"="将引用指向新的数组对象上来，这个几乎不需要时间，这样读操作就很快很安全，适合在多线程里使用，绝对不会发生ConcurrentModificationException ，因此CopyOnWriteArrayList适合使用在读操作远远大于写操作的场景里，比如缓存。

8.迭代器和枚举之间的区别?

如果面试官问这个问题，那么他的意图一定是让你区分Iterator不同于Enumeration的两个方面：

1️⃣.Iterator允许移除从底层集合的元素。

2️⃣.Iterator的方法名是标准化的。

9.Hashmap如何同步?

当我们需要一个同步的HashMap时，有两种选择：

1️⃣.使用Collections.synchronizedMap（..）来同步HashMap。

2️⃣.使用ConcurrentHashMap的.

这两个选项之间的首选是使用ConcurrentHashMap，这是因为我们不需要锁定整

个对象，以及通过ConcurrentHashMap分区地图来获得锁。

10.IdentityHashMap和HashMap的区别?

IdentityHashMap是Map接口的实现。不同于HashMap的，这里采用参考平等。

1️⃣.在HashMap中如果两个元素是相等的，则key1.equals(key2);

2️⃣.在IdentityHashMap中如果两个元素是相等的，则key1 == key2.

11.HashMap原理,与HashTable区别?

Java中的HashMap是以键值对(key-value)的形式存储元素的。HashMap需要一个hash函数，它使用hashCode()和equals()方法来向集合/从集合添加和检索元素。当调用put()方法的时候，HashMap会计算key的hash值，然后把键值对存储在集合中合适的索引上。如果key已经存在了，value会被更新成新值。HashMap的一些重要的特性是它的容量(capacity)，负载因子(load factor)和扩容极限(threshold resizing)。

put()方法的源码：

```
public V put(K key, V value) {
    // HashMap允许存放null键和null值。
    // 当key为null时，调用putForNullKey方法，将value放置在数组第一个位置
。
    if (key == null)
        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中的索引。
    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;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
{
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
```

```
    }

    // 如果i索引处的Entry为null，表明此处还没有Entry。
    modCount++;      //这个mod是用于线程安全的，下文有讲述
    // 将key、value添加到i索引处。
    addEntry(hash, key, value, i);
    return null;
}
```

addEntry:

```
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];
    // <strong><span style="color:#ff0000;">将新创建的 Entry 放入 buck
etIndex 索引处，并让新的 Entry 指向原来的 Entry  </span></strong>
    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);
}
```

更详细的原理请看: http://zhangshixi.iteye.com/blog/672697

区别:

http://blog.csdn.net/shohokuf/article/details/3932967

HashMap允许键和值是null，而Hashtable不允许键或者值是null。
Hashtable是同步的，而HashMap不是。因此，HashMap更适合于单线程环境，而Hashtable适合于多线程环境。
HashMap提供了可供应用迭代的键的集合，因此，HashMap是快速失败（具体看下文)的。另一方面，Hashtable提供了对键的列举(Enumeration)。
一般认为Hashtable是一个遗留的类。

12.让hashmap变成线程安全的两种方法.

1️⃣.方法一:通过Collections.synchronizedMap()返回一个新的Map,这个新的map就 是线程安全的. 这个要求大家习惯基于接口编程,因为返回的并不是HashMap,而是 一个Map的实现.

```
Map map = Collections.synchronizedMap(new HashMap());
```

2️⃣.方法二:使用ConcurrentHashMap

```
Map<String, Integer> concurrentHashMap = new ConcurrentHashMap<String
, Integer>();
```

13.ArrayList也是非线程安全的.

一个 ArrayList 类，在添加一个元素的时候，它可能会有两步来完成：1. 在 Items[Size] 的位置存放此元素；2. 增大 Size 的值。
在单线程运行的情况下，如果 Size = 0，添加一个元素后，此元素在位置 0，而且 Size=1；
而如果是在多线程情况下，比如有两个线程，线程 A 先将元素存放在位置 0。但是此时 CPU 调度线程A暂停，线程 B 得到运行的机会。线程B也将元素放在位置 0，（因为size还未增长），完了之后，两个线程都是size++，结果size变成2，而只有 items[0]有元素。
util.concurrent包也提供了一个线程安全的ArrayList替代者 CopyOnWriteArrayList。

14. Hashset原理是什么?

基于HashMap实现的，HashSet底层使用HashMap来保存所有元素（看了源码之后我发现就是用hashmap的keyset来保存的），因此HashSet 的实现比较简单，相关HashSet的操作，基本上都是直接调用底层HashMap的相关方法来完成， HashSet的源代码如下：

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable{
    static final long serialVersionUID = -5024744406713321676L;

    // 底层使用HashMap来保存HashSet中所有元素。
    private transient HashMap<E,Object> map;

 //定义一个虚拟的Object对象作为HashMap的value,将此对象定义为static final
。
    private static final Object PRESENT = new Object();

    /**
```

```
     * 默认的无参构造器，构造一个空的HashSet。
     *
     *实际底层会初始化一个空的HashMap,并使用默认初始容量为16和加载因子    *
 0.75。
     */
    public HashSet() {
    map = new HashMap<E,Object>();
    }

    /**
     * 构造一个包含指定collection中的元素的新set。
     *
     * 实际底层使用默认的加载因子0.75和足以包含指定
     * collection中所有元素的初始容量来创建一个HashMap。
     * @param c 其中的元素将存放在此set中的collection。
     */
    public HashSet(Collection<? extends E> c) {
    map = new HashMap<E,Object>(Math.max((int) (c.size()/.75f) + 1, 1
6));
    addAll(c);
    }

    /**
     * 以指定的initialCapacity和loadFactor构造一个空的HashSet。
     *
     * 实际底层以相应的参数构造一个空的HashMap。
     * @param initialCapacity 初始容量。
     * @param loadFactor 加载因子。
     */
    public HashSet(int initialCapacity, float loadFactor) {
    map = new HashMap<E,Object>(initialCapacity, loadFactor);
    }

    /**
     * 以指定的initialCapacity构造一个空的HashSet。
     *
     * 实际底层以相应的参数及加载因子loadFactor为0.75构造一个空的HashMap
。
     * @param initialCapacity 初始容量。
     */
    public HashSet(int initialCapacity) {
    map = new HashMap<E,Object>(initialCapacity);
    }

    /**
     * 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。
     * 此构造函数为包访问权限，不对外公开，实际只是是对LinkedHashSet的支
```

```
   * 持。
     *
     * 实际底层会以指定的参数构造一个空LinkedHashMap实例来实现。
     * @param initialCapacity 初始容量。
     * @param loadFactor 加载因子。
     * @param dummy 标记。
     */
    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);
    }

    /**
     * 返回对此set中元素进行迭代的迭代器。返回元素的顺序并不是特定的。
     *
     * 底层实际调用底层HashMap的keySet来返回所有的key。
     * 可见HashSet中的元素，只是存放在了底层HashMap的key上，
     * value使用一个static final的Object对象标识。
     * @return 对此set中元素进行迭代的Iterator。
     */
    public Iterator<E> iterator() {
    return map.keySet().iterator();
    }

    /**
     * 返回此set中的元素的数量（set的容量）。
     *
     * 底层实际调用HashMap的size()方法返回Entry的数量，就得到该Set中元素
的   * 个数。
     * @return 此set中的元素的数量（set的容量）。
     */
    public int size() {
      return map.size();
    }

    /**
     * 如果此set不包含任何元素，则返回true。
     *
     * 底层实际调用HashMap的isEmpty()判断该HashSet是否为空。
     * @return 如果此set不包含任何元素，则返回true。
     */
    public boolean isEmpty() {
          return map.isEmpty();
    }

    /**
     * 如果此set包含指定元素，则返回true。
     * 更确切地讲，当且仅当此set包含一个满足(o==null ? e==null :       *
```

```
     o.equals(e))
     * 的e元素时，返回true。
     *
     * 底层实际调用HashMap的containsKey判断是否包含指定key。
     * @param o 在此set中的存在已得到测试的元素。
     * @return 如果此set包含指定元素，则返回true。
     */
    public boolean contains(Object o) {
          return map.containsKey(o);
    }

    /**
     * 如果此set中尚未包含指定元素，则添加指定元素。
     * 更确切地讲，如果此 set 没有包含满足(e==null ? e2==null :      *
   e.equals(e2))
     * 的元素e2，则向此set 添加指定的元素e。
     * 如果此set已包含该元素，则该调用不更改set并返回false。
     *
     * 底层实际将将该元素作为key放入HashMap。
     * 由于HashMap的put()方法添加key-value对时，当新放入HashMap的Entry中
      * key
     * 与集合中原有Entry的key相同（hashCode()返回值相等，通过equals比较也
      *     返回true），
     * 新添加的Entry的value会将覆盖原来Entry的value，但key不会有任何改变
,
     * 因此如果向HashSet中添加一个已经存在的元素时，新添加的集合元素将不会
      * 被放入HashMap中，
     * 原来的元素也不会有任何改变，这也就满足了Set中元素不重复的特性。
     * @param e 将添加到此set中的元素。
     * @return 如果此set尚未包含指定元素，则返回true。
     */
    public boolean add(E e) {
    return map.put(e, PRESENT)==null;
    }

    /**
     * 如果指定元素存在于此set中，则将其移除。
     * 更确切地讲，如果此set包含一个满足(o==null ? e==null : o.equals(e)
)     *     的元素e，
     * 则将其移除。如果此set已包含该元素，则返回true
     * （或者：如果此set因调用而发生更改，则返回true）。（一旦调用返回，则
      *     此set不再包含该元素）。
     *
     * 底层实际调用HashMap的remove方法删除指定Entry。
     * @param o 如果存在于此set中则需要将其移除的对象。
     * @return 如果set包含指定元素，则返回true。
     */
```

```
    public boolean remove(Object o) {
    return map.remove(o)==PRESENT;
    }

    /**
     * 从此set中移除所有元素。此调用返回后，该set将为空。
     *
     * 底层实际调用HashMap的clear方法清空Entry中所有元素。
     */
    public void clear() {
        map.clear();
    }

    /**
     * 返回此HashSet实例的浅表副本：并没有复制这些元素本身。
     *
     * 底层实际调用HashMap的clone()方法，获取HashMap的浅表副本，并设置到
   *     HashSet中。
     */
    public Object clone() {
        try {
            HashSet<E> newSet = (HashSet<E>) super.clone();
            newSet.map = (HashMap<E, Object>) map.clone();
            return newSet;
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
```

15.ArrayList,Vector, LinkedList的存储性能和特性

ArrayList 和Vector都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，Vector由于使用了synchronized方法（线程安全），通常性能上较ArrayList差，而LinkedList使用双向链表实现存储，按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。

16.快速失败(fail-fast)和安全失败(fail-safe)的区别?

Fail-Fast机制：

我们知道java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了map，那么将抛ConcurrentModificationException，这就是所谓fail-fast策略。

这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容的修改都将增加这个值，那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。

```
HashIterator() {
    expectedModCount = modCount;
    if (size > 0) { // advance to first entry
    Entry[] t = table;
    while (index < t.length && (next = t[index++]) == null)
        ;
    }
}
```

在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map：

注意到modCount声明为volatile，保证线程之间修改的可见性。

```
final Entry<K,V> nextEntry() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
```

在HashMap的API中指出：

由所有HashMap类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 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，迭代器很快就会完全失败，而不冒在将来不确定的时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

Fail-Safe机制：

Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。java.util包下面的所有的集合类都是快速失败（一般的集合类)的，而java.util.concurrent包下面的所有的类(比如CopyOnWriteArrayList，ConcurrentHashMap ）都是安全失败的。快速失败的迭代器会抛出ConcurrentModificationException异常，而安全失败的迭代器永远不会抛出这样的异常。

17.传递一个集合作为参数给函数时，我们如何能确保函数将无法对其进行修改?

我们可以创建一个只读集合，使用Collections.unmodifiableCollection作为参数传递给使用它的方法，这将确保任何改变集合的操作将抛出UnsupportedOperationException。

18.Collections类的方法有哪些?

上面说到了很多了collections的方法，我们来深究一下这个类

Collections则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，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。

1️⃣.排序(Sort)

使用sort方法可以根据元素的自然顺序 对指定列表按升序进行排序。列表中的所有元素都必须实现 Comparable接口。此列表内的所有元素都必须是使用指定比较器可相互比较的

可以直接Collections.sort(...)

或者可以指定一个比较器，让这个列表遵照在比较器当中所设定的排序方式进行排序，这就提供了更大的灵活性

public static void sort(List l, Comparatorc)

这个Comparator同样是一个在java.util包中的接口。这个接口中有两个方法：int compare(T o1, T o2)和boolean equals(Object obj)

2️⃣.很多常用的，没必要多讲的方法

shuffle(Collection)： 对集合进行随机排序

binarySearch(Collection,Object)方法的使用(含义：查找指定集合中的元素，返回所查找元素的索引)

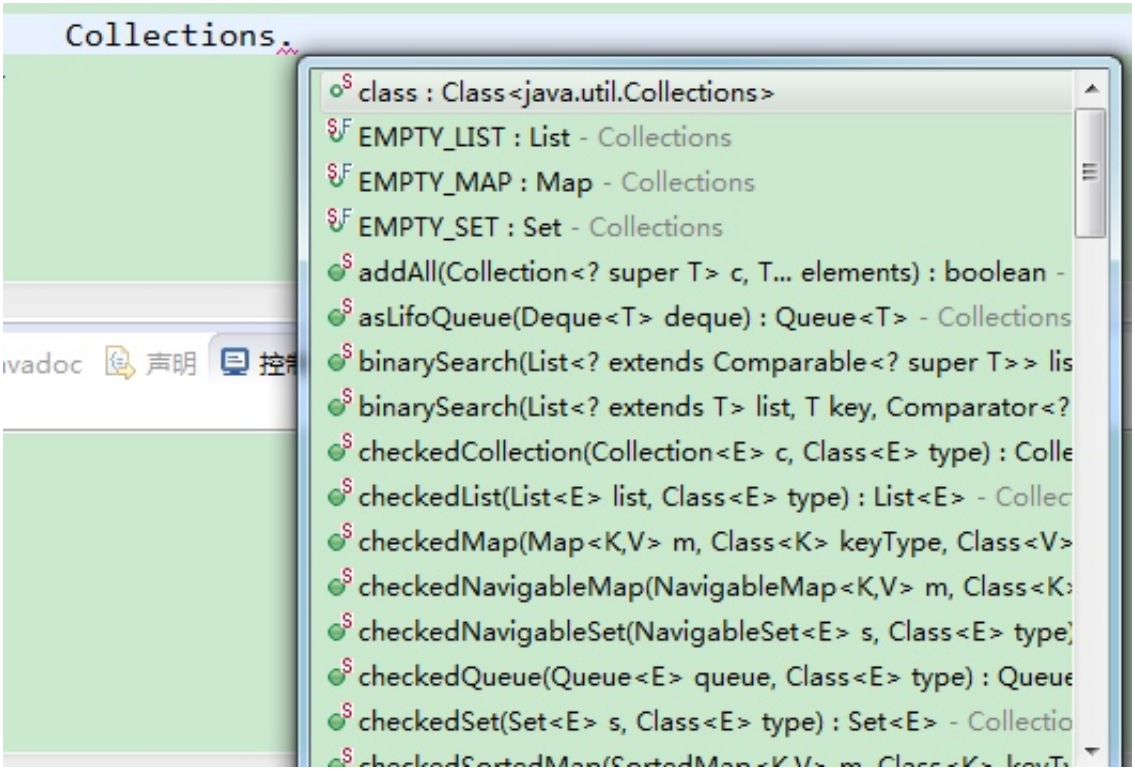
max(Collection),max(Collection,Comparator)方法的使用(前者采用Collection内含自然比较法，后者采用Comparator进行比较)

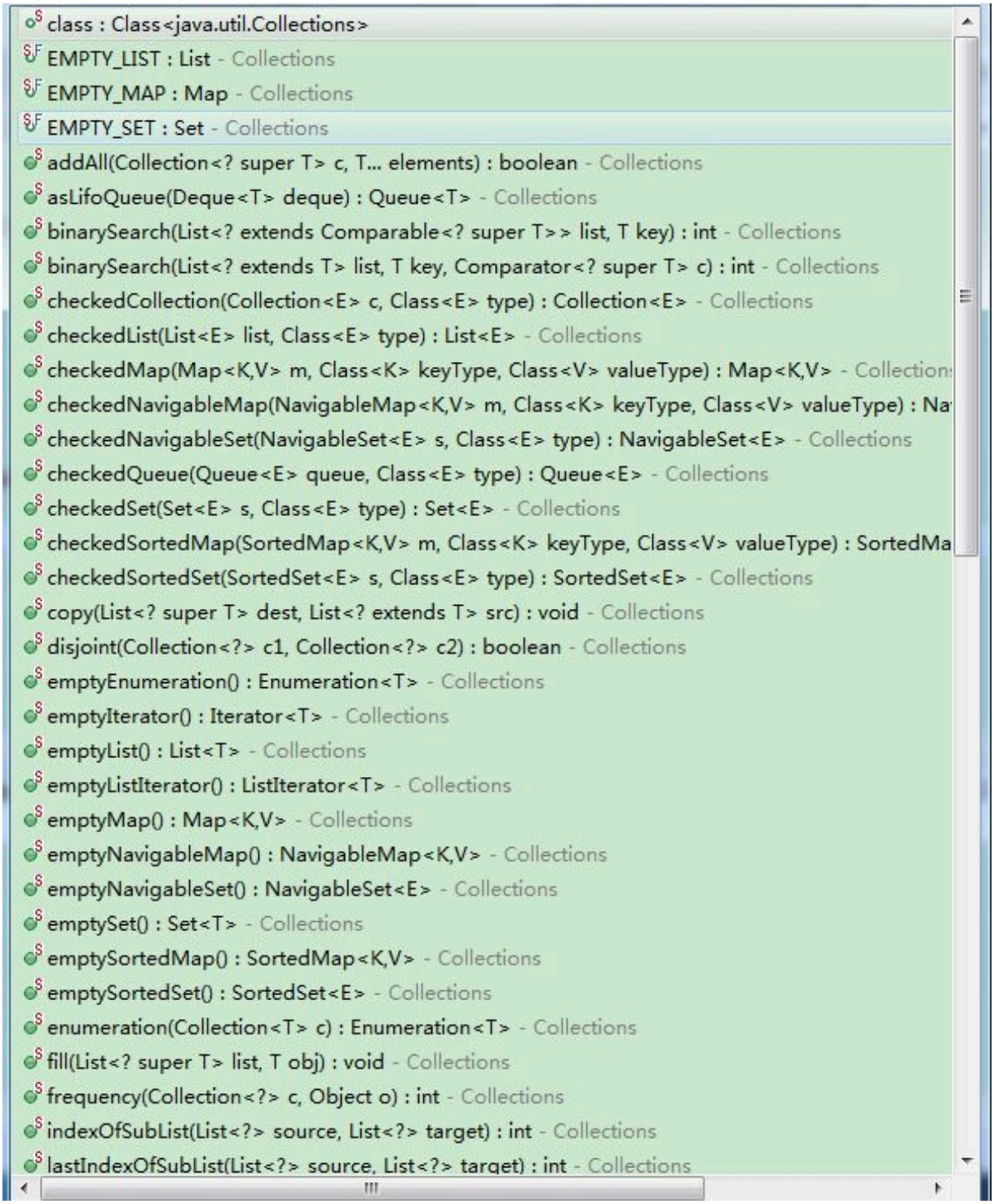
min(Collection),min(Collection,Comparator)方法的使用(前者采用Collection内含自然比较法，后者采用Comparator进行比较)。
indexOfSubList(List list,List subList)方法的使用(含义：查找subList在list中首次出现位置的索引)。

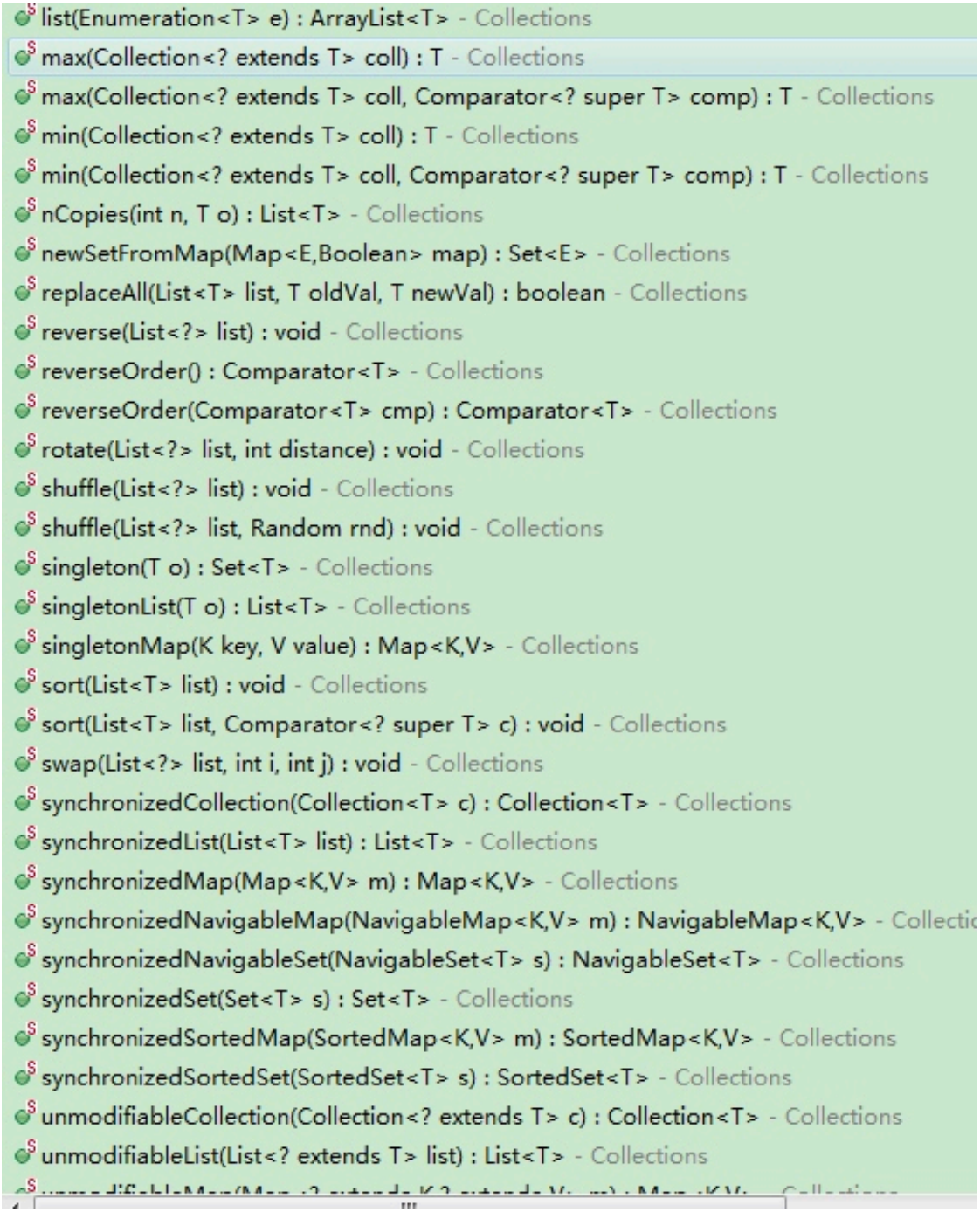
lastIndexOfSubList(List source,List target)方法的使用与上例方法的使用相同，在此就不做介绍了。

replaceAll(List list,Object old,Object new)方法的使用(含义：替换批定元素为某元素,若要替换的值存在刚返回true,反之返回false)。

3️⃣.（这一段可以直接参考JAVA API说明 http://www.apihome.cn/api/java/Collections.html ）




class : Class<java.util.Collections>
EMPTY_LIST : List - Collections
EMPTY_MAP : Map - Collections
EMPTY_SET : Set - Collections
addAll(Collection<? super T> c, T... elements) : boolean - Collections
asLifoQueue(Deque<T> deque) : Queue<T> - Collections
binarySearch(List<? extends Comparable<? super T>> list, T key) : int - Collections
binarySearch(List<? extends T> list, T key, Comparator<? super T> c) : int - Collections
checkedCollection(Collection<E> c, Class<E> type) : Collection<E> - Collections
checkedList(List<E> list, Class<E> type) : List<E> - Collections
checkedMap(Map<K,V> m, Class<K> keyType, Class<V> valueType) : Map<K,V> - Collection
checkedNavigableMap(NavigableMap<K,V> m, Class<K> keyType, Class<V> valueType) : Na
checkedNavigableSet(NavigableSet<E> s, Class<E> type) : NavigableSet<E> - Collections
checkedQueue(Queue<E> queue, Class<E> type) : Queue<E> - Collections
checkedSet(Set<E> s, Class<E> type) : Set<E> - Collections
checkedSortedMap(SortedMap<K,V> m, Class<K> keyType, Class<V> valueType) : SortedMa
checkedSortedSet(SortedSet<E> s, Class<E> type) : SortedSet<E> - Collections
copy(List<? super T> dest, List<? extends T> src) : void - Collections
disjoint(Collection<?> c1, Collection<?> c2) : boolean - Collections
emptyEnumeration() : Enumeration<T> - Collections
emptyIterator() : Iterator<T> - Collections
emptyList() : List<T> - Collections
emptyListIterator() : ListIterator<T> - Collections
emptyMap() : Map<K,V> - Collections
emptyNavigableMap() : NavigableMap<K,V> - Collections
emptyNavigableSet() : NavigableSet<E> - Collections
emptySet() : Set<T> - Collections
emptySortedMap() : SortedMap<K,V> - Collections
emptySortedSet() : SortedSet<E> - Collections
enumeration(Collection<T> c) : Enumeration<T> - Collections
fill(List<? super T> list, T obj) : void - Collections
frequency(Collection<?> c, Object o) : int - Collections
indexOfSubList(List<?> source, List<?> target) : int - Collections
lastIndexOfSubList(List<?> source, List<?> target) : int - Collections



除了2️⃣.中讲到的一些零碎的，可以看到还分成了checked , empty , singleton, synchronized unmodifiable这几类。

checked()：2个用途：

返回指定 collection 的一个动态类型安全视图。试图插入一个错误类型的元素将导致立即抛出 ClassCastException。假设在生成动态类型安全视图之前，collection 不包含任何类型不正确的元素，并且所有对该 collection 的后续访问都通过该视图进行，则可以保证 该 collection 不包含类型不正确的元素。

一般的编程语言机制中都提供了编译时（静态）类型检查，但是一些未经检查的强制转换可能会使此机制无效。通常这不是一个问题，因为编译器会在所有这类

未经检查的操作上发出警告。但有的时候，只进行单独的静态类型检查并不够。例如，假设将 collection 传递给一个第三方库，则库代码不能通过插入一个错误类型的元素来毁坏 collection。

动态类型安全视图的另一个用途是调试。假设某个程序运行失败并抛出 ClassCastException，这指示一个类型不正确的元素被放入已参数化 collection 中。不幸的是，该异常可以发生在插入错误元素之后的任何时间，因此，这通常只能提供很少或无法提供任何关于问题真正来源的信息。如果问题是可再现的，那么可以暂时修改程序，使用一个动态类型安全视图来包装该 collection，通过这种方式可快速确定问题的来源。

unmodifiable():

返回指定 集合的不可修改视图。此方法允许模块为用户提供对内部 集合的“只读”访问。在返回的 集合 上执行的查询操作将“读完”指定的集合。试图修改返回的集合（不管是直接修改还是通过其迭代器进行修改）将导致抛出 UnsupportedOperationException。

synchronized():

`public static <T> Collection<T> synchronizedCollection(Collection<T> c)`
返回指定 collection 支持的同步（线程安全的）collection。为了保证按顺序访问，必须通过返回的 collection 完成所有对底层实现 collection 的访问。
在返回的 collection 上进行迭代时，用户必须手工在返回的 collection 上进行同步：

```
Collection c = Collections.synchronizedCollection(myCollection);
     ...
  synchronized(c) {
      Iterator i = c.iterator(); // Must be in the synchronized block
      while (i.hasNext())
         foo(i.next());
  }
```

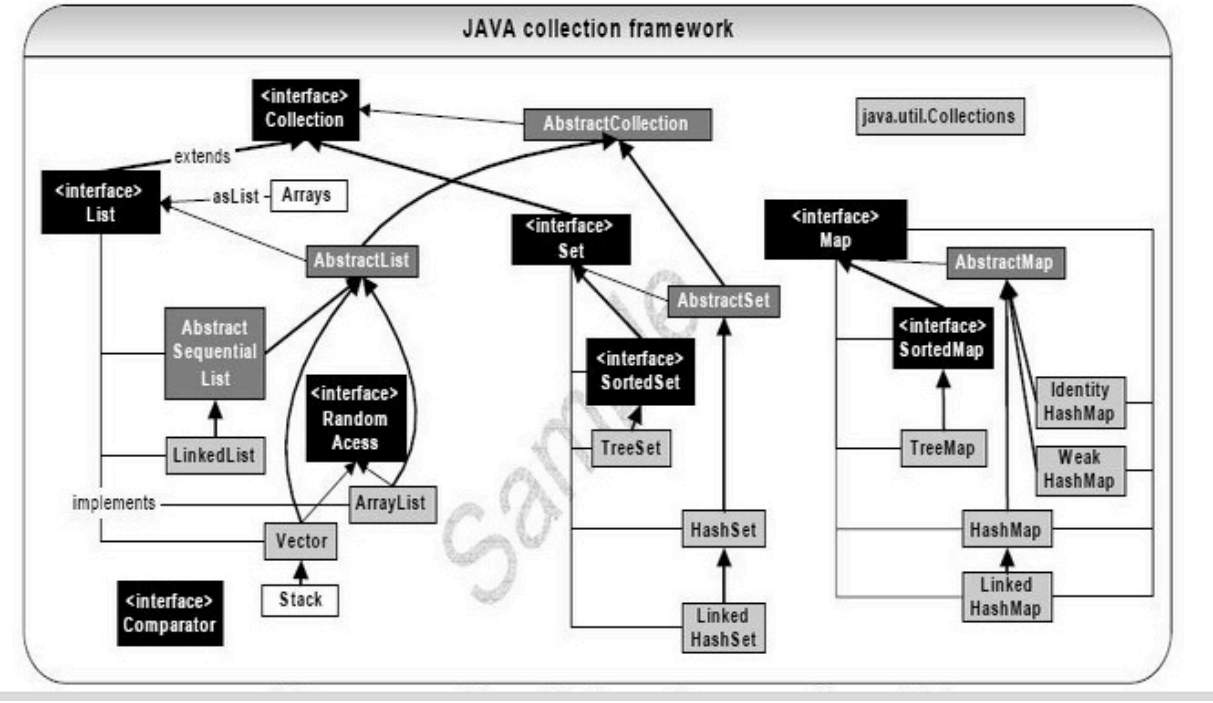
empty(): (以set 为例，我没看懂到底是干嘛的。。）

`public static final <T> Set<T> emptySet()`
返回空的 set（不可变的）。此 set 是可序列化的。与 like-named（找不到关于这个东西的资料。。） 字段不同，此方法是参数化的。
以下示例演示了获得空 set 的类型安全 (type-safe) 方式：

```
Set<String> s = Collections.emptySet();
```

19.Tree, Hash ,Linked

再看看这个图。



发现set和map的实现分成了 Tree,Hash,和Linked。

以map为例，来看看这三者的区别.

1️⃣.TreeMap用红黑树实现，能够把它保存的记录根据键排序，默认是按升序排序， 也可以指定排序的比较器。当用Iteraor遍历TreeMap时，得到的记录是排过序 的。TreeMap的键和值都不能为空。

2️⃣.HashMap上文有说。

3️⃣.LinkedHashmap：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。put方法没有重写，重写了addEntry()。（因为加入的时候要维护 好一个双向链表的结构）LinkedHashMap重写了父类HashMap的get方法，实际在调 用父类getEntry()方法取得查找的元素后，再判断当排序模式accessOrder为true 时，记录访问顺序，将最新访问的元素添加到双向链表的表头，并从原来的位置 删除。由于的链表的增加、删除操作是常量级的，故并不会带来性能的损失 (accessOrder是LinkedHashmap中的一个属性，用来判断是否要根据读取顺序来重 写调整结构。如果为false，就按照插入的顺序排序，否则按

照最新访问的放在链 表前面的顺序，以提高性能）。
LinedHashMap的作用就是在让经常访问的元素更快的被访问到。用双向链表可以 方便地执行链表中元素的插入删除操作。

20.为何Collection不从Cloneable和Serializable接口继承?

Collection接口指定一组对象，对象即为它的元素。如何维护这些元素由Collection的具体实现决定。例如，一些如List的Collection实现允许重复的元素，而其它的如Set就不允许。很多Collection实现有一个公有的clone方法。然而，把它放到集合的所有实现中也是没有意义的。这是因为Collection是一个抽象表现。重要的是实现。

当与具体实现打交道的时候，克隆或序列化的语义和含义才发挥作用。所以，具体实现应该决定如何对它进行克隆或序列化，或它是否可以被克隆或序列化。

在所有的实现中授权克隆和序列化，最终导致更少的灵活性和更多的限制。特定的实现应该决定它是否可以被克隆和序列化。

21.为何Map接口不继承Collection接口?

尽管Map接口和它的实现也是集合框架的一部分，但Map不是集合，集合也不是Map。因此，Map继承Collection毫无意义，反之亦然。

如果Map继承Collection接口，那么元素去哪儿？Map包含key-value对，它提供抽取key或value列表集合的方法，但是它不适合“一组对象”规范。

22.Enumeration和Iterator接口的区别?

Enumeration的速度是Iterator的两倍，也使用更少的内存。Enumeration是非常基础的，也满足了基础的需要。但是，与Enumeration相比，Iterator更加安全，因为当一个集合正在被遍历的时候，它会阻止其它线程去修改集合。

迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者从集合中移除元素，而Enumeration不能做到。为了使它的功能更加清晰，迭代器方法名已经经过改善。

23.为何没有像Iterator.add()这样的方法，向集合中添

加元素?

语义不明，已知的是，Iterator的协议不能确保迭代的次序。然而要注意，ListIterator没有提供一个add操作，它要确保迭代的顺序。

24.Iterater和ListIterator之间有什么区别?

1️⃣.我们可以使用Iterator来遍历Set和List集合，而ListIterator只能遍历List。

2️⃣.Iterator只可以向前遍历，而LIstIterator可以双向遍历。

3️⃣.ListIterator从Iterator接口继承，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。

25.在迭代一个集合的时候，如何避免ConcurrentModificationException?

在遍历一个集合的时候，我们可以使用并发集合类来避免ConcurrentModificationException，比如使用CopyOnWriteArrayList，而不是ArrayList。

26.hashCode()和equals()方法有何重要性?

HashMap使用Key对象的hashCode()和equals()方法去决定key-value对的索引。当我们试着从HashMap中获取值的时候，这些方法也会被用到。如果这些方法没有被正确地实现，在这种情况下，两个不同Key也许会产生相同的hashCode()和equals()输出，HashMap将会认为它们是相同的，然后覆盖它们，而非把它们存储到不同的地方。同样的，所有不允许存储重复数据的集合类都使用hashCode()和equals()去查找重复，所以正确实现它们非常重要。equals()和hashCode()的实现应该遵循以下规则：
1️⃣.如果o1.equals(o2)，那么o1.hashCode() == o2.hashCode()总是为true的。

2️⃣.如果o1.hashCode() == o2.hashCode()，并不意味着o1.equals(o2)会为true。

27.HashMap和HashTable有何不同?

1️⃣.HashMap允许key和value为null，而HashTable不允许。

2️⃣.HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，

HashTable 适合多线程环境。

3️⃣.在Java1.4中引入了LinkedHashMap，HashMap的一个子类，假如你想要遍历顺 序，你很容易从HashMap转向LinkedHashMap，但是HashTable不是这样的，它的顺 序是不可预知的。

4️⃣.HashMap提供对key的Set进行遍历，因此它是fail-fast的，但HashTable提供对 key的Enumeration进行遍历，它不支持fail-fast。

5️⃣.HashTable被认为是个遗留的类，如果你寻求在迭代的时候修改Map，你应该使用 CocurrentHashMap。

40个Java集合面试问题和答案
2015/05/19 | 分类： 基础技术, 职业生涯 | 1 条评论 | 标签： JAVA, 面试

分享到： 146
译文出处： Sanesee 原文出处：javacodegeeks
1.Java集合框架是什么？说出一些集合框架的优点?

每种编程语言中都有集合，最初的Java版本包含几种集合类：Vector、Stack、HashTable和Array。随着集合的广泛使用，Java1.2提出了囊括所有集合接口、实现和算法的集合框架。在保证线程安全的情况下使用泛型和并发集合类，Java已经经历了很久。它还包括在Java并发包中，阻塞接口以及它们的实现。集合框架的部分优点如下：

（1）使用核心集合类降低开发成本，而非实现我们自己的集合类。

（2）随着使用经过严格测试的集合框架类，代码质量会得到提高。

（3）通过使用JDK附带的集合类，可以降低代码维护成本。

（4）复用性和可操作性。

2.集合框架中的泛型有什么优点?

Java1.5引入了泛型，所有的集合接口和实现都大量地使用它。泛型允许我们为集合提供一个可以容纳的对象类型，因此，如果你添加其它类型的任何元素，它会在编译时报错。这避免了在运行时出现ClassCastException，因为你将会在编译时得到报错信息。泛型也使得代码整洁，我们不需要使用显式转换和instanceOf操作符。它也给运行时带来好处，因为不会产生类型检查的字节码指令。

3.Java集合框架的基础接口有哪些?

Collection为集合层级的根接口。一个集合代表一组对象，这些对象即为它的元素。Java平台不提供这个接口任何直接的实现。

Set是一个不能包含重复元素的集合。这个接口对数学集合抽象进行建模，被用来代表集合，就如一副牌。

List是一个有序集合，可以包含重复元素。你可以通过它的索引来访问任何元素。List更像长度动态变换的数组。

Map是一个将key映射到value的对象.一个Map不能包含重复的key：每个key最多只能映射一个value。

一些其它的接口有Queue、Dequeue、SortedSet、SortedMap和ListIterator。

4.为何Collection不从Cloneable和Serializable接口继承?

Collection接口指定一组对象，对象即为它的元素。如何维护这些元素由Collection的具体实现决定。例如，一些如List的Collection实现允许重复的元素，而其它的如Set就不允许。很多Collection实现有一个公有的clone方法。然而，把它放到集合的所有实现中也是没有意义的。这是因为Collection是一个抽象表现。重要的是实现。

当与具体实现打交道的时候，克隆或序列化的语义和含义才发挥作用。所以，具体实现应该决定如何对它进行克隆或序列化，或它是否可以被克隆或序列化。

在所有的实现中授权克隆和序列化，最终导致更少的灵活性和更多的限制。特定的实现应该决定它是否可以被克隆和序列化。

5.为何Map接口不继承Collection接口?

尽管Map接口和它的实现也是集合框架的一部分，但Map不是集合，集合也不是Map。因此，Map继承Collection毫无意义，反之亦然。

如果Map继承Collection接口，那么元素去哪儿？Map包含key-value对，它提供抽取key或value列表集合的方法，但是它不适合“一组对象”规范。

6.Iterator是什么?

Iterator接口提供遍历任何Collection的接口。我们可以从一个Collection中使用迭代器方法来获取迭代器实例。迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者在迭代过程中移除元素。

7.Enumeration和Iterator接口的区别?

Enumeration的速度是Iterator的两倍，也使用更少的内存。Enumeration是非常基础的，也满足了基础的需要。但是，与Enumeration相比，Iterator更加安全，因为当一个集合正在被遍历的时候，它会阻止其它线程去修改集合。

迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者从集合中移除元素，而Enumeration不能做到。为了使它的功能更加清晰，迭代器方法名已经经过改善。

8.为何没有像Iterator.add()这样的方法，向集合中添加元素?

语义不明，已知的是，Iterator的协议不能确保迭代的次序。然而要注意，ListIterator没有提供一个add操作，它要确保迭代的顺序。

9.为何迭代器没有一个方法可以直接获取下一个元素，而不需要移动游标?

它可以在当前Iterator的顶层实现，但是它用得很少，如果将它加到接口中，每个继承都要去实现它，这没有意义。

10.Iterater和ListIterator之间有什么区别?

（1）我们可以使用Iterator来遍历Set和List集合，而ListIterator只能遍历List。

（2）Iterator只可以向前遍历，而LIstIterator可以双向遍历。

（3）ListIterator从Iterator接口继承，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。

11.遍历一个List有哪些不同的方式?

1
2
3
4
5
6
7
8
9
10
11

```
List strList = new ArrayList<>();
//使用for-each循环
for(String obj : strList){
System.out.println(obj);
}
//using iterator
Iterator it = strList.iterator();
while(it.hasNext()){
String obj = it.next();
System.out.println(obj);
}
```

使用迭代器更加线程安全，因为它可以确保，在当前遍历的集合元素被更改的时候，它会抛出ConcurrentModificationException。

12.通过迭代器fail-fast属性，你明白了什么?

每次我们尝试获取下一个元素的时候，Iterator fail-fast属性检查当前集合结构里的任何改动。如果发现任何改动，它抛出ConcurrentModificationException。Collection中所有Iterator的实现都是按fail-fast来设计的（ConcurrentHashMap和CopyOnWriteArrayList这类并发集合类除外）。

13.fail-fast与fail-safe有什么区别?

Iterator的fail-fast属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。Java.util包中的所有集合类都被设计为fail-fast的，而java.util.concurrent中的集合类都为fail-safe的。Fail-fast迭代器抛出ConcurrentModificationException，而fail-safe迭代器从不抛出ConcurrentModificationException。

14.在迭代一个集合的时候，如何避免ConcurrentModificationException?

在遍历一个集合的时候，我们可以使用并发集合类来避免ConcurrentModificationException，比如使用CopyOnWriteArrayList，而不是ArrayList。

15.为何Iterator接口没有具体的实现?

Iterator接口定义了遍历集合的方法，但它的实现则是集合实现类的责任。每个能够返回用于遍历的Iterator的集合类都有它自己的Iterator实现内部类。

这就允许集合类去选择迭代器是fail-fast还是fail-safe的。比如，ArrayList迭代器

是fail-fast的，而CopyOnWriteArrayList迭代器是fail-safe的。

16.UnsupportedOperationException是什么?

UnsupportedOperationException是用于表明操作不支持的异常。在JDK类中已被大量运用，在集合框架java.util.Collections.UnmodifiableCollection将会在所有add和remove操作中抛出这个异常。

17.在Java中，HashMap是如何工作的?

HashMap在Map.Entry静态内部类实现中存储key-value对。HashMap使用哈希算法，在put和get方法中，它使用hashCode()和equals()方法。当我们通过传递key-value对调用put方法的时候，HashMap使用Key hashCode()和哈希算法来找出存储key-value对的索引。Entry存储在LinkedList中，所以如果存在entry，它使用equals()方法来检查传递的key是否已经存在，如果存在，它会覆盖value，如果不存在，它会创建一个新的entry然后保存。当我们通过传递key调用get方法时，它再次使用hashCode()来找到数组中的索引，然后使用equals()方法找出正确的Entry，然后返回它的值。下面的图片解释了详细内容。

其它关于HashMap比较重要的问题是容量、负荷系数和阀值调整。HashMap默认的初始容量是32，负荷系数是0.75。阀值是为负荷系数乘以容量，无论何时我们尝试添加一个entry，如果map的大小比阀值大的时候，HashMap会对map的内容进行重新哈希，且使用更大的容量。容量总是2的幂，所以如果你知道你需要存储大量的key-value对，比如缓存从数据库里面拉取的数据，使用正确的容量和负荷系数对HashMap进行初始化是个不错的做法。

18.hashCode()和equals()方法有何重要性?

HashMap使用Key对象的hashCode()和equals()方法去决定key-value对的索引。当我们试着从HashMap中获取值的时候，这些方法也会被用到。如果这些方法没有被正确地实现，在这种情况下，两个不同Key也许会产生相同的hashCode()和equals()输出，HashMap将会认为它们是相同的，然后覆盖它们，而非把它们存储到不同的地方。同样的，所有不允许存储重复数据的集合类都使用hashCode()和equals()去查找重复，所以正确实现它们非常重要。equals()和hashCode()的实现应该遵循以下规则：

（1）如果o1.equals(o2)，那么o1.hashCode() == o2.hashCode()总是为true的。

（2）如果o1.hashCode() == o2.hashCode()，并不意味着o1.equals(o2)会为true。

19.我们能否使用任何类作为Map的key?

我们可以使用任何类作为Map的key，然而在使用它们之前，需要考虑以下几点：

（1）如果类重写了equals()方法，它也应该重写hashCode()方法。

（2）类的所有实例需要遵循与equals()和hashCode()相关的规则。请参考之前提到的这些规则。

（3）如果一个类没有使用equals()，你不应该在hashCode()中使用它。

（4）用户自定义key类的最佳实践是使之为不可变的，这样，hashCode()值可以被缓存起来，拥有更好的性能。不可变的类也可以确保hashCode()和equals()在未来不会改变，这样就会解决与可变相关的问题了。

比如，我有一个类MyKey，在HashMap中使用它。

```
1
2
3
4
5
6
7
//传递给MyKey的name参数被用于equals()和hashCode()中
MyKey key = new MyKey('Pankaj'); //assume hashCode=1234
myHashMap.put(key, 'Value');
// 以下的代码会改变key的hashCode()和equals()值
key.setName('Amit'); //assume new hashCode=7890
//下面会返回null，因为HashMap会尝试查找存储同样索引的key，而key已被改变了，匹配失败，返回null
myHashMap.get(new MyKey('Pankaj'));
```

那就是为何String和Integer被作为HashMap的key大量使用。

20.Map接口提供了哪些不同的集合视图?

Map接口提供三个集合视图：

（1）Set keyset()：返回map中包含的所有key的一个Set视图。集合是受map支持的，map的变化会在集合中反映出来，反之亦然。当一个迭代器正在遍历一个

集合时，若map被修改了（除迭代器自身的移除操作以外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

（2）Collection values()：返回一个map中包含的所有value的一个Collection视图。这个collection受map支持的，map的变化会在collection中反映出来，反之亦然。当一个迭代器正在遍历一个collection时，若map被修改了（除迭代器自身的移除操作以外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

（3）Set> entrySet()：返回一个map钟包含的所有映射的一个集合视图。这个集合受map支持的，map的变化会在collection中反映出来，反之亦然。当一个迭代器正在遍历一个集合时，若map被修改了（除迭代器自身的移除操作，以及对迭代器返回的entry进行setValue外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

21.HashMap和HashTable有何不同?

（1）HashMap允许key和value为null，而HashTable不允许。

（2）HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程环境。

（3）在Java1.4中引入了LinkedHashMap，HashMap的一个子类，假如你想要遍历顺序，你很容易从HashMap转向LinkedHashMap，但是HashTable不是这样的，它的顺序是不可预知的。

（4）HashMap提供对key的Set进行遍历，因此它是fail-fast的，但HashTable提供对key的Enumeration进行遍历，它不支持fail-fast。

（5）HashTable被认为是个遗留的类，如果你寻求在迭代的时候修改Map，你应该使用CocurrentHashMap。

28.如何决定选用HashMap还是TreeMap?

对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。然而，假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的

collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

29.ArrayList和Vector有何异同点?

ArrayList和Vector在很多时候都很类似。
1️⃣.两者都是基于索引的，内部由一个数组支持。
2️⃣.两者维护插入的顺序，我们可以根据插入顺序来获取元素。
3️⃣.ArrayList和Vector的迭代器实现都是fail-fast的。
4️⃣.ArrayList和Vector两者允许null值，也可以使用索引值对元素进行随机访问。

以下是ArrayList和Vector的不同点。

1️⃣.Vector是同步的，而ArrayList不是。然而，如果你寻求在迭代的时候对列表进行改变，你应该使用CopyOnWriteArrayList。
2️⃣.ArrayList比Vector快，它因为有同步，不会过载。
3️⃣.ArrayList更加通用，因为我们可以使用Collections工具类轻易地获取同步列表和只读列表。

30.ArrayList和LinkedList有何区别?

ArrayList和LinkedList两者都实现了List接口，但是它们之间有些不同。

1️⃣.ArrayList是由Array所支持的基于一个索引的数据结构，所以它提供对元素的随 机访问，复杂度为O(1)，但LinkedList存储一系列的节点数据，每个节点都与前 一个和下一个节点相连接。所以，尽管有使用索引获取元素的方法，内部实现是 从起始点开始遍历，遍历到索引的节点然后返回元素，时间复杂度为O(n)，比ArrayList要慢。

2️⃣.与ArrayList相比，在LinkedList中插入、添加和删除一个元素会更快，因为在一个元素被插入到中间的时候，不会涉及改变数组的大小，或更新索引。

3️⃣.LinkedList比ArrayList消耗更多的内存，因为LinkedList中的每个节点存储了前后节点的引用。

31.队列和栈是什么，列出它们的区别?

栈和队列两者都被用来预存储数据。java.util.Queue是一个接口，它的实现类在Java并发包中。队列允许先进先出（FIFO）检索元素，但并非总是这样。Deque

接口允许从两端检索元素。

栈与队列很相似，但它允许对元素进行后进先出（LIFO）进行检索。

Stack是一个扩展自Vector的类，而Queue是一个接口。