## **说说常见的集合有哪些吧？**

《马士兵多线程与高并发》集合的那个图。二话不说，背过

# ===========HashMap=========

HashMap 线程不安全，put数据时会参数java.util.concurrentModificationException（并发修改异常），ConcurrentHashMap(分段锁)集合工具类Collections.synchronizedMap(new HashMap)

## **HashMap与HashTable的区别？**

HashMap允许K/V都为null；Hashtable 的K/V都不允许为null；

Hashtable是jdk1.0的，设计的有问题，方法都是synchronized的，后来jdk1.2加入了HashMap。

map数据结构：key无序不可重复，value无序可重复；一个key-value键值对组成一个entry对象，也是无序不可重复

底层构成：

1、jdk7 Map map = new HashMap（）；创建一个初始大小为16的entry数组，当使用map.put(key，value) 存储数据时，首先调用key所在的类中的hashcode方法，计算key的hash值，

经过某种算法，得到entry数组中的下标位置，如果该下标已存在元素，再调用key所在类中的equals方法，返回false在链表中添加，返回true更新，如果不存在元素直接添加，在不断

添加过程中，超过16大小，需要扩容，默认扩容2倍，并将原有的数据复制过来

2、jdk8 Map map = new HashMap（）创建一个空的node{}数组，首次调用put方法，会初始化node数组大小为16，当数组某一索引上的元素以链表的形式存在的数量>8且数组大小>64(小于64扩容)时，此索引上的

所有数据采用红黑树方式存储

## **简单说说 HashMap 的底层原理？**

1. 先根据 key 的 hash 值得到这个 Entry 元素在数组中的位置（即下标），
2. 然后把这个 Entry 元素放到对应的位置中，
   1. 如果这个 Entry 元素所在的位子上已经存放有其他元素就在同一个位子上的 Entry 元素以链表的形式存放，新加入的放在链头，

从 HashMap 中 get  Entry 元素时先计算 key 的 hashcode，找到数组中对应位置的某一 Entry 元素，

然后通过 key 的 equals 方法在对应位置的链表中找到需要的 Entry 元素，所以 HashMap 的数据结构是数组和链表的结合，

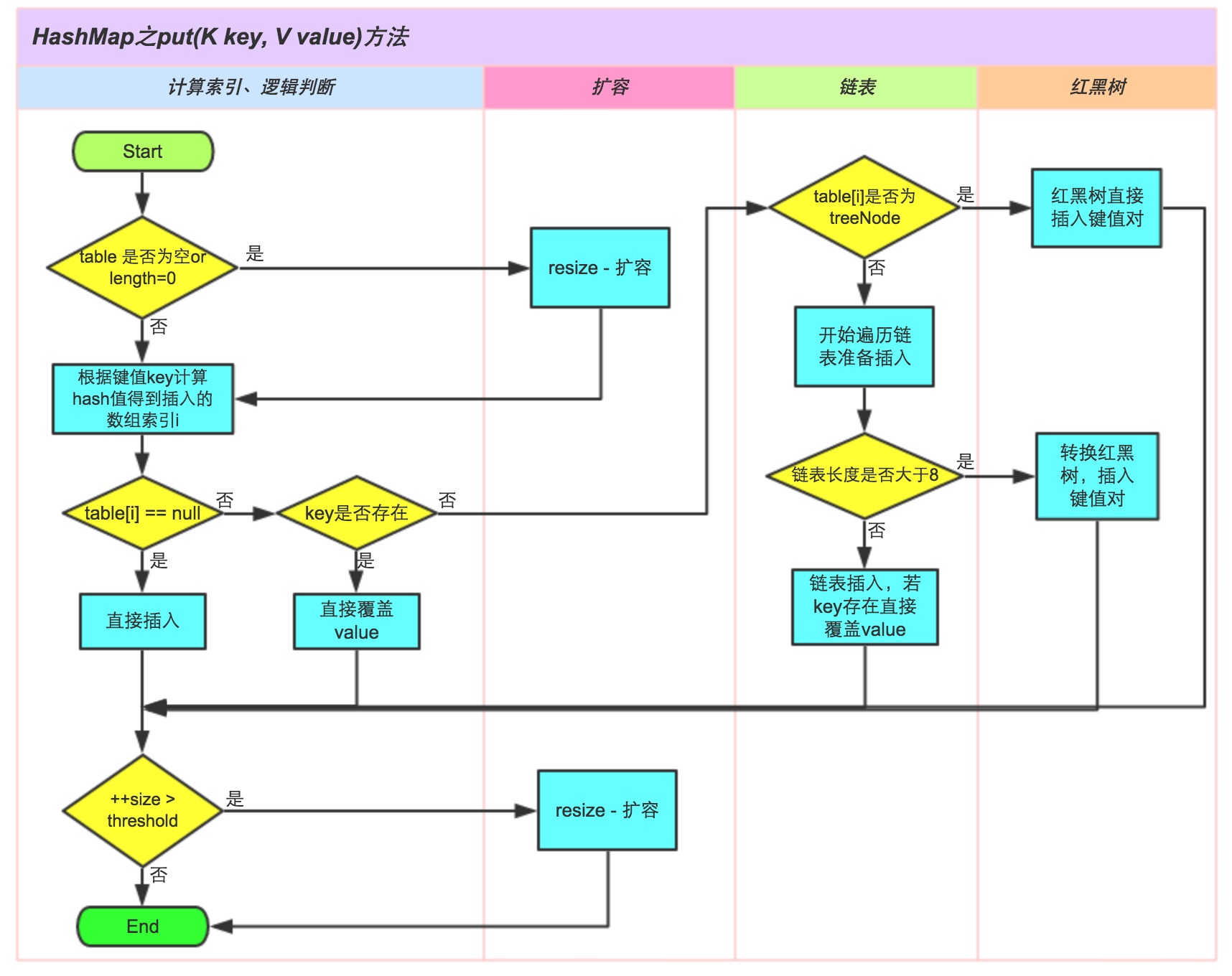
此外 HashMap 中 key 和 value 都允许为 null，key 为 null 的键值对永远都放在以 table[0] 为头结点的链表中。

对于 JDK 1.8 开始 HashMap 实现原理变成了数组+链表+红黑树的结构，数组链表部分基本不变，**红黑树是为了解决哈希碰撞后链表索引效率的问题**，所以在 JDK 1.8 中当链表的节点大于 8 个时就会将链表变为红黑树。区别是 JDK 1.8 以前碰撞节点会在链表头部插入，而 JDK 1.8 开始碰撞节点会在链表尾部插入，对于扩容操作后的节点转移 JDK 1.8 以前转移前后链表顺序会倒置，而 JDK 1.8 中依然保持原序。

## Jdk1.8中满足什么条件后将链表转化成红黑树？

很显然在putVal方法中是判断桶内的节点个数是否大于8，之后通过treeifyBin方法中判断长度是否大于最小红黑树容量64,小于则继续扩容，大于则转为红黑树。

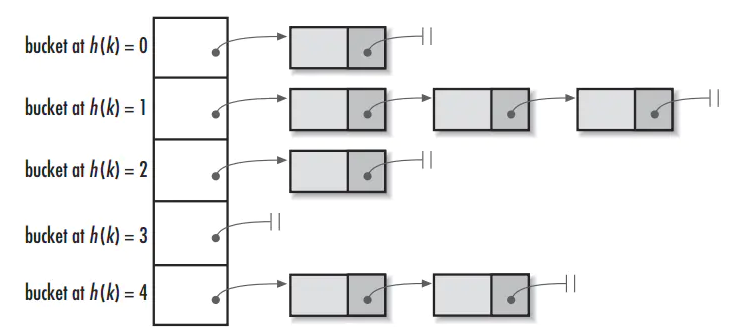
## **HashMap的put方法的具体流程？**



## **HashMap是怎么解决哈希冲突的？**

**数组+链表+扰动函数+红黑树**

1. 数组+链表解决hash冲突
2. 2次扰动函数（hash函数）来降低哈希冲突的概率，使得数据分布更平均，减少哈希碰撞
3. 引入红黑树进一步降低遍历的时间复杂度，使得遍历更快



**红黑树作用：**

但是当我们的HashMap中存在大量数据时，加入我们某个bucket下对应的链表有n个元素，那么遍历时间复杂度就为O(n)，为了针对这个问题，JDK1.8在HashMap中新增了红黑树的数据结构，进一步使得遍历复杂度降低至O(logn)；

**红黑树：**

自平衡二叉查找树，红黑树是一种特化的AVL树（[平衡二叉树](https://baike.baidu.com/item/%E5%B9%B3%E8%A1%A1%E4%BA%8C%E5%8F%89%E6%A0%91/10421057" \t "https://baike.baidu.com/item/%E7%BA%A2%E9%BB%91%E6%A0%91/_blank)），都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。

时间复杂度：O(log n)

## **简单说说 JDK 1.8 中 HashMap 是如何扩容的？与 JDK 1.7 有什么区别？**

数组默认长度：16

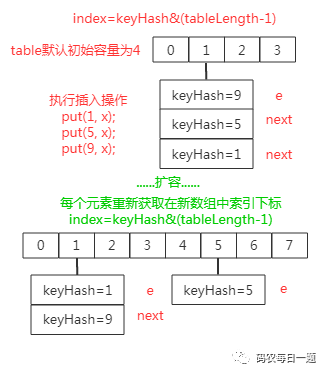
**何时扩容：元素个数 > (数组总大小length)\*阈值**

并且默认长度和扩容后的长度都必须是 2 的幂

默认填充因子0.75，提前扩容，能尽量减少链表长度

扩容2倍

### JDK1.7:

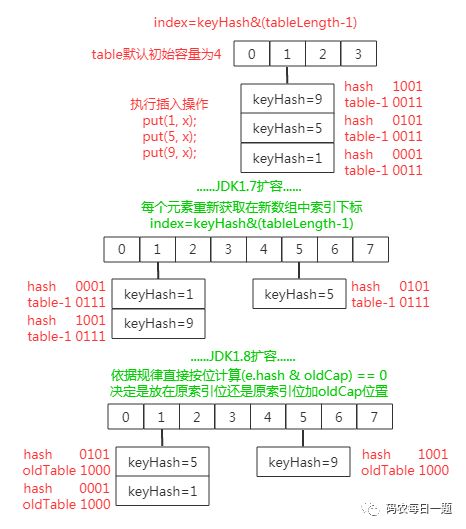


1.7 扩容过程，取出数组元素,然后遍历以该元素为头的单向链表元素，依据每个被遍历元素的 hash 值计算其在新数组中的下标然后进行交换。

链表元素会发生倒置，原来链表头扩容后变为尾巴

### JDK1.8：

而在 JDK 1.8 中，由于扩容数组的长度是 2 倍关系，所以对于假设初始 tableSize =4 要扩容到 8 来说就是 0100 到 1000 的变化（左移一位就是 2 倍），在扩容中只用判断原来的 hash 值与左移动的一位按位与操作是 0 或 1 就行，0 的话索引就不变，1 的话索引变成原索引加上扩容前数组，所以其实现如下流程图所示：



可以看见，这个设计非常赞，因为 hash 值本来就是随机性的，所以 hash 按位与上 newTable 得到的 0（扩容前的索引位置）和 1（扩容前索引位置加上扩容前数组长度的数值索引处）就是随机的，所以**扩容的过程就能把之前哈西冲突的元素再随机的分布到不同的索引去，这算是 JDK1.8 的一个优化点。**

而在 JDK1.8 中不会出现链表倒置现象

由于 JDK1.7 中发生哈西冲突时仅仅采用了链表结构存储冲突元素，所以扩容时仅仅是重新计算其存储位置而已，而 JDK1.8 中为了性能在同一索引处发生哈西冲突到一定程度时链表结构会转换为红黑数结构存储冲突元素，故在扩容时如果当前索引中元素结构是红黑树且元素个数小于链表还原阈值（哈西冲突程度常量）时**就会把树形结构缩小或直接还原为链表结构**

## **HashMap 默认的初始化长度是多少？为什么默认长度和扩容后的长度都必须是 2 的幂？**

在 JDK 中默认长度是 16，并且默认长度和扩容后的长度都必须是 2 的幂。

**当长度为 2 的幂时不同的 hash 值发生碰撞的概率比较小，这样就会使得数据在 table 数组中分布较均匀，查询速度也较快。按位操作比直接取模效率要高.**

|  |
| --- |
| 因为我们可以先看下 HashMap 的 put 方法核心，如下：    可以看到获取数组索引的计算方式为 key 的 hash 值按位与运算数组长度减一，为了说明长度尽量是 2 的幂的作用我们假设执行了 put("android", 123); 语句且 "android" 的 hash 值为 234567，二进制为 111001010001000111，然后由于 HashMap 默认长度为 16，所以减一后的二进制为 1111，接着两数做按位与操作二进制结果为 111，即十进制的 7，所以 index 为 7，可以看出这种按位操作比直接取模效率要高。  如果假设 HashMap 默认长度不是 2 的幂，譬如数组长度为 6，减一的二进制为 101，与 111001010001000111 按位与操作二进制 101，此时假设我们通过 put 再放一个 key-value 进来，其 hash 为 111001010001000101，其与 101 按位与操作后的二进制也为 101，很容易发生哈希碰撞，这就不符合 index 的均匀分布了。  通过上面两个假设例子可以看出 HashMap 的长度为 2 的幂时减一的值的二进制位数一定全为 1，这样数组下标 index 的值完全取决于 key 的 hash 值的后几位，因此只要存入 HashMap 的 Entry 的 key 的 hashCode 值分布均匀，HashMap 中数组 Entry 元素的分部也就尽可能是均匀的（也就避免了 hash 碰撞带来的性能问题），所以当长度为 2 的幂时不同的 hash 值发生碰撞的概率比较小，这样就会使得数据在 table 数组中分布较均匀，查询速度也较快。不过即使负载因子和 hash 算法设计的再合理也免不了哈希冲突碰撞的情况，一旦出现过多就会影响 HashMap 的性能，所以在 JDK 1.8 中官方对数据结构引入了红黑树，当链表长度太长（默认超过 8）时链表就转为了红黑树，而红黑树的增删改查都比较高效，从而解决了哈希碰撞带来的性能问题。 |

**为什么String, Interger这样的类适合作为键？**

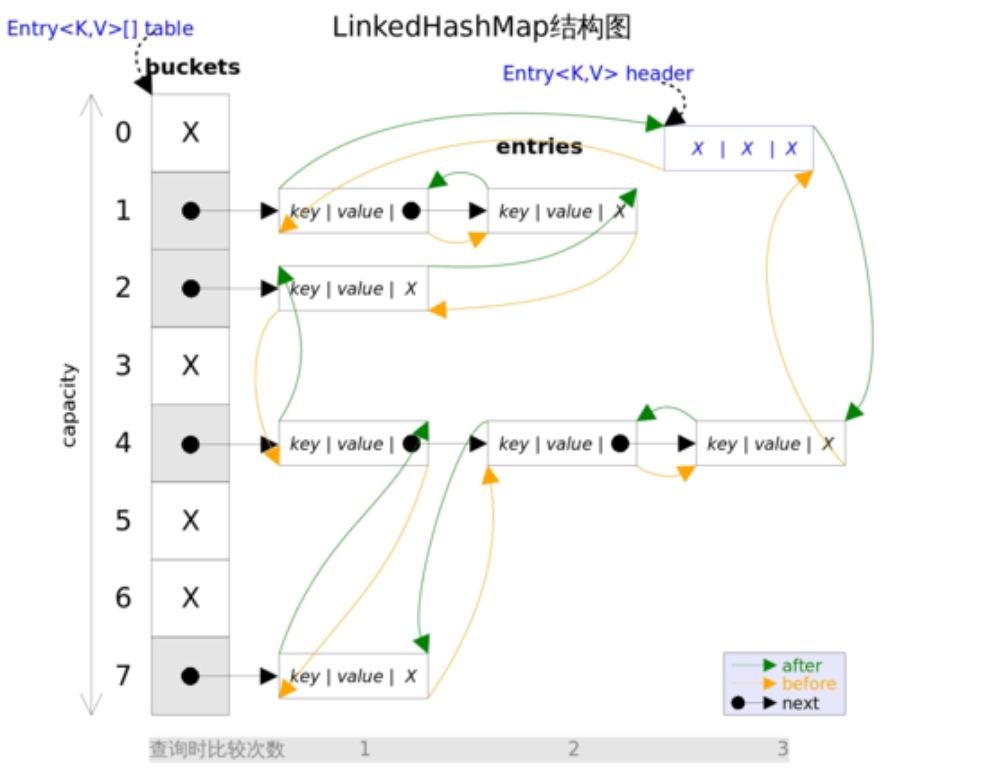
String, Interger这样的类作为HashMap的键是再适合不过了，而且String最为常用。   
　　因为String对象是不可变的，而且已经重写了equals()和hashCode()方法了。   
　　1.不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。   
　　注：String的不可变性可以看这篇文章《【java基础】浅析String》。   
　　2.因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。

## **HashMap在JDK1.7和JDK1.8中有哪些不同？**

|  |  |  |
| --- | --- | --- |
| **不同** | **JDK 1.7** | **JDK 1.8** |
| 存储结构 | 数组 + 链表 | 数组 + 链表 + 红黑树 |
| 初始化方式 | 单独函数：inflateTable() | 直接集成到了扩容函数resize()中 |
| hash值计算方式 | 扰动处理 = 9次扰动 = 4次位运算 + 5次异或运算 | 扰动处理 = 2次扰动 = 1次位运算 + 1次异或运算 |
| 存放数据的规则 | 无冲突时，存放数组；冲突时，存放链表 | 无冲突时，存放数组；冲突 & 链表长度 < 8：存放单链表；冲突 & 链表长度 > 8：树化并存放红黑树 |
| 插入数据方式 | 头插法（先讲原位置的数据移到后1位，再插入数据到该位置） | 尾插法（直接插入到链表尾部/红黑树） |
| 扩容后存储位置的计算方式 | 全部按照原来方法进行计算（即hashCode ->> 扰动函数 ->> (h&length-1)） | 按照扩容后的规律计算（即扩容后的位置=原位置 or 原位置 + 旧容量） |

treeMap 自然排序或key排序，底层使用红黑树

# ===========LinkedHashMap =========



## **HashMap和LinkedHashMap的区别**

linkedHashMap在HashMap底层的基础上增加了一个指向前后元素的指针，便于遍历；

HashMap可以允许一条键为null的键值对，允许多条值为null的键值对，其并发不安全，如果想并发安全操作可以使用Collections.synchronizedMap()方法或ConcurrentHashMap来代替。

1.LinkedHashMap 继承自 HashMap。它是线程不安全的,允许key为null,value为null

该结构由数组和链表+红黑树 。其内部还维护了一个****双向链表****，在每次****插入数据，或者访问、修改数据****时，****会增加节点、或调整链表的节点顺序****。以决定迭代时输出的顺序。默认情况，遍历时的顺序是****按照插入节点的顺序****。这也是其与HashMap最大的区别

因继承自HashMap,所以HashMap[上文](http://blog.csdn.net/zxt0601/article/details/77413921" \t "https://blog.csdn.net/zxt0601/article/details/_blank)分析的特点，除了输出无序，其他LinkedHashMap都有，比如扩容的策略，哈希桶长度一定是2的N次方等等。   
LinkedHashMap在实现时，就是重写override了几个方法。以满足其输出序列有序的需求。

# =========TreeMap =========

## TreeMap

TreeMap 是一个**有序的**key-value集合，它是通过**红黑树**实现的。

TreeMap 继承于AbstractMap，所以它是一个Map，即一个key-value集合。

TreeMap 实现了NavigableMap接口，意味着它支持一系列的导航方法。比如返回有序的key集合。

TreeMap 实现了Cloneable接口，意味着它能被克隆。

TreeMap基于红黑树（Red-Black tree）实现。该映射根据其键的自然顺序(字母排序)进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。

TreeMap的基本操作 containsKey、get、put 和 remove 的时间复杂度是 log(n) 。

**TreeMap是非线程安全的**。 它的iterator 方法返回的迭代器是fail-fast的。

## TreeMap和TreeSet在排序时如何比较元素？

   TreeSet要求存放的对象所属的类必须实现Comparable接口，该接口提供了比较元素的compareTo()方法，当插入元素时会回调该方法比较元素的大小。TreeMap要求存放的键值对映射的键必须实现Comparable接口从而根据键对元素进行排序。

## Collections工具类中的sort()方法如何比较元素？

Collections工具类的sort方法有两种重载的形式:

第一种要求传入的待排序容器中存放的对象必须实现Comparable接口以实现元素的比较；

第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是Comparator接口的子类型（需要重写compare方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java中对函数式编程的支持）

|  |
| --- |
| 例子：  第一种    测试程序：    第二种：    测试程序： |

# ========= List =========

## **ArrayList在JDK1.7、JDK1.8默认长度**

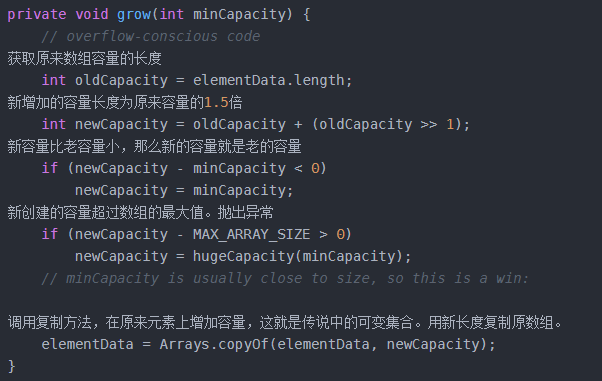
底层是数组，查询快，增删慢，线程不安全。

在jdk7：new时直接创建长度为10的数组

jdk8中：在new时初始化长度为0，只是在第一次add时会创建长度为10的数组，之后扩容按原来的1.5倍进行扩容。

## **ArrayList扩容**

每次在add()一个元素时，arraylist都需要对这个list的容量进行一个判断。如果容量够，直接添加，否则需要进行扩容。在1.8 arraylist这个类中，扩容调用的是grow()方法，通过grow()方法中调用的Arrays.copyof()方法进行对原数组的复制，在通过调用System.arraycopy()方法进行复制，达到扩容的目的



ArrayList 线程不安全，add数据时会参数java.util.concurrentModificationException（并发修改异常），可使用vector、集合工具类Collections.synchronizedList(new ArrayList) 、CopyOnWriteArrayList

## **Java集合的快速失败机制 “fail-fast”？**

例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生fail-fast机制。

**原因：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变modCount的值。每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测modCount变量是否为expectedmodCount值，是的话就返回遍历；否则抛出异常，终止遍历。**

**解决办法：**

**1. 在遍历过程中，所有涉及到改变modCount值得地方全部加上synchronized。**

**2. 使用CopyOnWriteArrayList来替换ArrayList**

## **ArrayList和LinkedList的区别？**

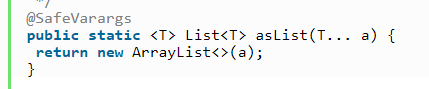
· LinkedList 实现了 List 和 Deque 接口，一般称为双向链表；ArrayList 实现了 List 接口，动态数组；

· LinkedList 在插入和删除数据时效率更高，ArrayList 在查找某个 index 的数据时效率更高；

· LinkedList 比 ArrayList 需要更多的内存；

**Arrays.asList();方法实现原理**

**Arrays工具类提供了一个方法asList, 使用该方法可以将一个变长参数或者数组转换成List 。**



原来Arrays的asList方法使用的ArrayList类是一个内部定义的类，而不是java.util.ArrayList类。

从这个内部类ArrayList的实现可以看出，它继承了抽象类java.util.AbstractList<E>, 但是没有重写add和remove方法，没有给出具体的实现。

但是，默认情况下，java.util.AbstractList类在add、set以及remove方法中，直接会抛出UnsupportedOperationException异常， 正是因为java.util.Arrays类的内部类ArrayList没有重写add和remove方法，所以，当我们调用其add方法时，其实就是调用了AbstractList类的add方法，结果就是直接抛出UnsupportedOperationException异常

从上述的例子和简单分析来看，Arrays工具类提供了一个方法asList, 使用该方法可以将一个变长参数或者数组转换成List 。

但是，生成的List的长度是固定的；能够进行修改操作（比如，修改某个位置的元素）；不能执行影响长度的操作（如add、remove等操作）。会抛出UnsupportedOperationException异常。

Arrays.asList比较适合那些已经有数组数据或者一些元素，而需要快速构建一个List，只用于读取操作，而不进行添加或删除操作的场景。

如果，想要根据已知数组数据，快速获取一个可进行增删改查的列表List，一个比较简单的方法如下：

重新使用java.util.ArrayList包装一层

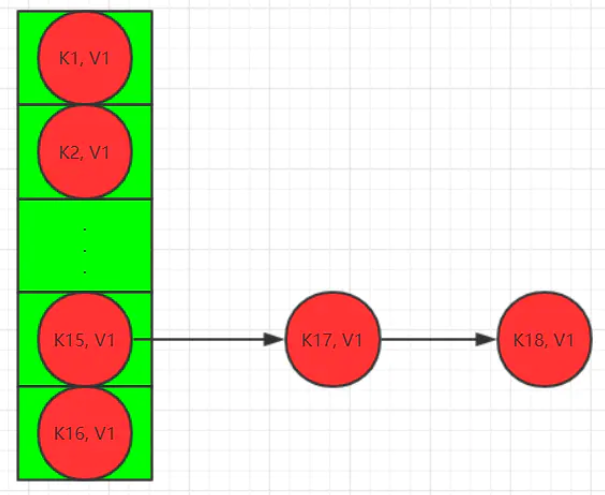
List<String> array1 = **new** ArrayList<>(Arrays.asList("Welcome", "to", "Java", "world"));

# ========= Set =========

## **HashSet**

HashSet是一个元素不会重复并且无序的容器

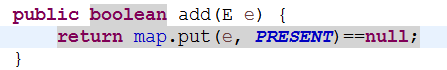
HashSet实现了Set接口，是一个不能够存放重复元素的容器，内部直接使用HashMap实现，即底层使用数组存储数据



## **HashSet是如何保证数据不可重复的？**

HashSet的底层其实就是HashMap，因HashMap的Key须唯一，只不过我们****HashSet是实现了Set接口并且把数据作为K值，而V值一直使用一个相同的虚值来保存****，我们可以看到源码：

**private** **static** **final** Object ***PRESENT*** = **new** Object();



由于HashMap的K值本身就不允许重复，并且在HashMap中如果K/V相同时，会用新的V覆盖掉旧的V，然后返回旧的V，那么在HashSet中执行这一句话始终会返回一个false，导致插入失败，这样就保证了数据的不可重复性；

HashSet:底层数组+链表，先用hashcode确定下标，再用eques确定是否一致

HashSet 线程不安全，add数据时会参数java.util.concurrentModificationException（并发修改异常），集合工具类Collections.synchronizedSet(new HashSet) 、CopyOnWriteArraySet(public CopyOnWriteArraySet() {al = new CopyOnWriteArrayList<E>();})

## **java集合中常见异常fail-fast**

今天来看java集合中一个常见的错误机制fail-fast机制。出现在这个错误机制的本质就是因为单线程和多线程的不同。下面就好好看一下这个机制是怎么是出现的。

一、认识fail-fast

今天在运行项目的时候，突然就出现了ConcurrentModificationException异常。原因是多线程中使用的，因为在多线程中使用了ArrayList，造成了这么一个异常。这是今天所讲的集合的fai-fast机制。

概念：fail-fast 机制是java集合(Collection)中的一种错误机制。当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。

从前面我们知道fail-fast是在操作迭代器时产生的。现在我们来看看ArrayList中迭代器的源代码：

从上面的源代码我们可以看出，迭代器在调用next()、remove()方法时都是调用checkForComodification()方法，它检测modCount == expectedModCount ？ 若不等则抛出ConcurrentModificationException 异常，从而产生fail-fast机制。到了这一步我们也知道了，想要弄清楚fail-fast机制，首先我们需要搞清楚modCount 和expectedModCount。

expectedModCount 是在IteratorTest中定义的：int expectedModCount = ArrayList.this.modCount;所以他的值是不可能会修改的，所以会变的就是modCount。modCount是在 AbstractList 中定义的，为全局变量：

方案一：在遍历过程中所有涉及到改变modCount值得地方全部加上synchronized或者直接使用Collections.synchronizedList（不推荐）

方案二：使用CopyOnWriteArrayList来替换ArrayList。

CopyOnWriteArrayList为什么能解决这个问题呢？CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。CopyOnWriteArrayList中add/remove等写方法是需要加锁的，目的是为了避免Copy出N个副本出来，导致并发写。但是。CopyOnWriteArrayList中的读方法是没有加锁的。

我们只需要记住一句话，那就是CopyOnWriteArrayList是线程安全的，所以我们在多线程的环境下面需要去使用这个就可以了。关于CopyOnWriteArrayList更加深入的用法，会在以后的章节中去解释说明。

三、总结

现在我们对fail-fast机制都已经有了了解了。其出现的原因是：当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。类似于我们在学习操作系统的时候出现的问题。

CurrencyHashMap使用分段锁达到线程安全并提高效率

ConcurrentHashMap中的get没有加锁，怎么保证线程安全的？

答：因为Node中的val使用volatile修饰，保证了有序性和可见性；并不是因为数组用volatile修饰了，volatile修饰数组，仅仅保证引用的可见性，并不保证内容可见性

concurrenthashmap为什么放弃分段锁：效率问题，分段锁耗费性能

8、ConcurrentHashMap中size方法，1.7和1.8实现的区别

单链表倒叙：

package com.zjf;

public class SingleLinkedNode {

private Object data;

private SingleLinkedNode next;

public SingleLinkedNode(Object data, SingleLinkedNode next) {

this.data = data;

this.next = next;

}

public static void main(String[] args) {

SingleLinkedNode n5 = new SingleLinkedNode(5,null);

SingleLinkedNode n4 = new SingleLinkedNode(4,n5);

SingleLinkedNode n3 = new SingleLinkedNode(3,n4);

SingleLinkedNode n2 = new SingleLinkedNode(2,n3);

SingleLinkedNode n1 = new SingleLinkedNode(1,n2);

System.out.println(sortNode(n1));

}

@Override

public String toString() {

String s = this.data.toString();

SingleLinkedNode next = this.next;

while (next!=null){

s+=","+next.data;

next = next.next;

}

return s;

}

private static SingleLinkedNode sortNode(SingleLinkedNode n) {

SingleLinkedNode node = new SingleLinkedNode(n.data,null);

while (n.next!=null){

SingleLinkedNode temp = n.next;

node = new SingleLinkedNode(temp.data,node);

n = temp;

}

return node;

}

}