集合关注

1、是否允许空

2、是否允许重复数据

3、是否有序，有序的意思是读取数据的顺序和存放数据的顺序是否一致

4、是否线程安全

# ArrayList

ArrayList() 默认初始容量是10 底层是默认调用ArrayList(10) 的有参构造函数 ，每次容量不够时扩充1.5倍

然后通过Arrays.copyOf将数据复制过来 插入和删除都会调用Arrays.copyOf()移动数组其他元素

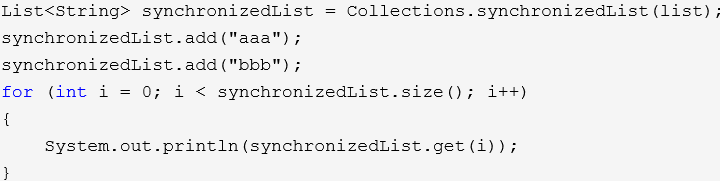
实现迭代器接口，可通过Iterator和for循环遍历

适合查找，插入删除慢

可空，可重复，有序，不安全

想线程安全怎么办

1. 用Collections.synchronizedList方法把你的ArrayList变成一个线程安全的List



2.使用vector

Vector每次自动扩容为2倍

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。

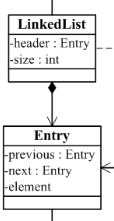
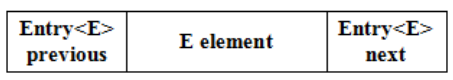
例如：当某一个线程A通过iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出ConcurrentModificationException异常，产生fail-fast事件。

volatile：volatile的本意是“易变的”。volatile关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。当要求使用volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。volatile 指出 i是随时可能发生变化的，每次使用它的时候必须从i的地址中读取。对于volatile类型的变量，系统每次用到他的时候都是直接从对应的内存当中提取，而不会利用cache当中的原有数值，以适应它的未知何时会发生的变化。

# LinkedList

本质是一个双向链表

适合插入删除，查找慢

LinkedList包含两个重要的成员：内部类Entry和 size。 Entry是双向链表节点所对应的数据结构

　　header是双向链表的表头，它是双向链表节点所对应的类Entry的实例。Entry中包含成员变量：previous, next, element。其中，previous是该节点的上一个节点，next是该节点的下一个节点，element是该节点所包含的值。

　　size是双向链表中节点的个数。

LinkedList插入删除指定索引位置的元素时，查找index位置的节点时，get(int location)，有一个加速动作：若index < 双向链表长度的1/2，则从前向后查找; 否则，从后向前查找

支持转化为数组



有些说法认为LinkedList做插入和删除更快，这种说法其实是不准确的：

（1）LinkedList做插入、删除的时候，慢在寻址，快在只需要改变前后Entry的引用地址

（2）ArrayList做插入、删除的时候，慢在数组元素的批量copy，快在寻址

所以，如果待插入、删除的元素是在数据结构的前半段尤其是非常靠前的位置的时候，LinkedList的效率将大大快过ArrayList，因为ArrayList将批量copy大量的元素；会很慢

ArrayList使用最普通的for循环遍历，LinkedList使用foreach循环比较快

因为ArrayList是实现了RandomAccess接口而LinkedList则没有实现这个接口

在编译的时候编译器会自动将对for这个关键字的使用转化为对目标的迭代器的使用，这就是foreach循环的原理，任何一个集合，无论是JDK提供的还是自己写的，只要想使用foreach循环遍历，就必须正确地实现Iterable接口

# Hashmap

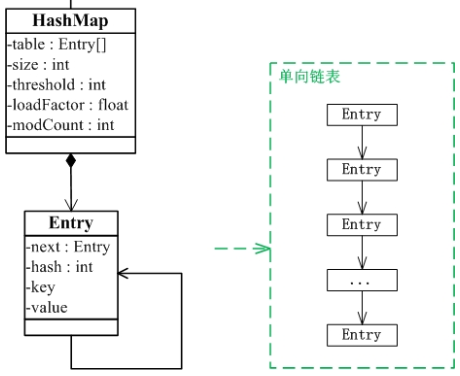
键不允许重复

无序

键和值都可以为null //hashtable键值都不可以为null，会抛出空指针异常，synchronized保证安全

线程不安全 // ConcurrentHashMap解决，取代hashtable，键值都不可以为null

使用拉链法解决哈希冲突



底层有一个entry类型的数组 初始容量16，加载因子0.75

当大于容积和因子成积时进行rehash(翻倍)操作

（新建一个更大的数组，遍历原来table中每个位置的链表，并对每个元素进行重新hash）

成员变量包括：table size threshold loadFactor modCount

table是一个Entry[]数组类型，而Entry是一个单向链表。哈希表的"key-value键值对"都存储在Entry数组中的。其中Entry为HashMap的内部类，它包含了key，value， hash值，下一节点next

size是HashMap的大小，它是HashMap保存的键值对的数量。默认16

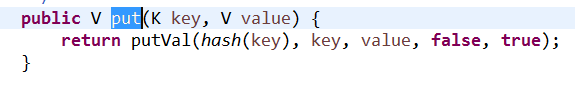
threshold是HashMap的阈值，判断是否需要翻倍HashMap的容量。threshold的值="容量\*加载因子"，当HashMap中存储数据的数量达到threshold时， HashMap的容量加倍。

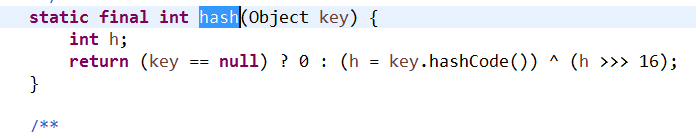
loadFactor就是加载因子---0.75

modCount是用来实现fail-fast机制的，每次修改结构自增

当我们想一个HashMap中添加一对key-value时，系统首先会计算key的hash值，然后根据hash值确认在table中存储的位置。若该位置没有元素，则直接插入。否则迭代该处元素链表并依此比较其key的hash值。如果两个hash值相等且key值相等(e.hash == hash && ((k = e.key) == key || key.equals(k))),则用新的Entry的value覆盖原来节点的value。如果两个hash值相等但key值不等 ，则将该节点插入该链表的链头(jdk1.8后改成尾部插入，防止并发get造成链表成环死循环问题)

为什么hashmap键可为null





HashMap在put的时候会调用hash()方法来计算key的hashcode值，可以从hash算法中看出当key==null时返回的值为0。因此key为null时，hash算法返回值为0，默认放在数组0位置

而Hashtable会直接抛出空指针异常，而无特殊处理

HashMap在get时 containsKey() 首先通过getEntry(key)获取key对应的Entry，然后判断该Entry是否为null

1.8版本前后优化更新

JDK1.7及之前

使用一个 Entry 数组来存储数据，用key的HashCode 取模来决定key会被放到数组里的位置，如果 HashCode相同，或者 HashCode 取模后的结果相同（ hash collision ），那么这些 key 会被定位到 Entry 数组的同一个 格子里，这些 key 会形成一个链表。

在 HashCode 特别差的情况下，比方说所有key的 HashCode 都相同，这个链表可能会很长，那么 put/get 操作 都可能需要遍历这个链表，也就是说时间复杂度在最差情况下会退化到 O(n)

JDK1.8中

使用一个 Node 数组来存储数据，但这个 Node 可能是链表结构，也可能是红黑树结构

如果插入的 key 的 HashCode 相同，那么这些key也会被定位到 Node 数组的同一个格子里。

如果同一个格子里的key不超过8个，使用链表结构存储。

如果超过了8个，那么会调用 treeifyBin 函数，将链表转换为红黑树。少于6个复原成链表

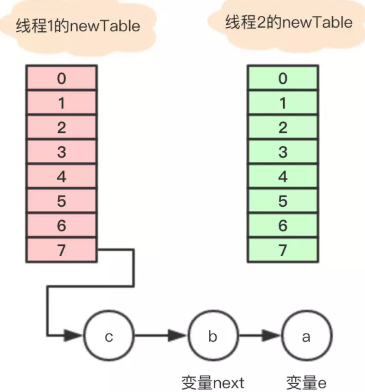
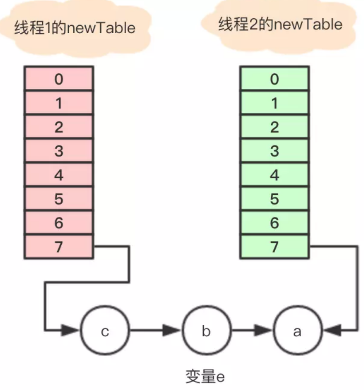
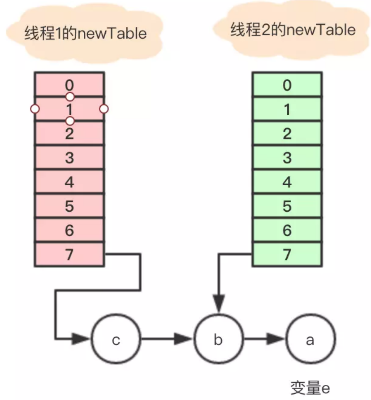
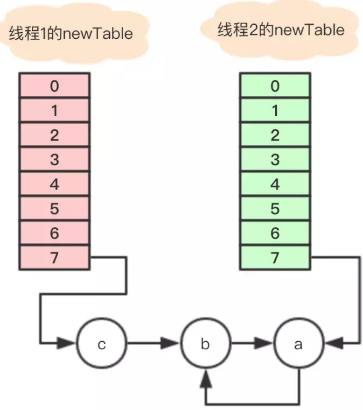
那么即使 HashCode 完全相同，由于红黑树的特点，查找某个特定元素，也只需要O(log n)的开销

也就是说put/get的操作的时间复杂度最差只有 O(log n) 听起来挺不错，但是真正想要利用它的好处有一个限制： key的对象，必须正确的实现了 Compare 接口 如果没有实现 Compare 接口，或者实现得不正确（比方说所有 Compare 方法都返回0） 那其实还是慢于 JDK1.7 的

HashMap 是线程安全的吗，为什么不是线程安全的?

1.put的时候导致的多线程数据不一致。

2.hashMap的get()方法死循环

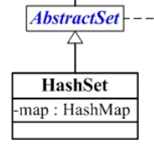
Hashmap 你传的是10，那么桶数组长度会为16(2的n次)

HashMap是懒加载，创建hashmap不会创建数组，第一次hHashMap的put方法的时候才会执行resize，进行table数组的初始化，即，初始化table数组的时候会执行resize函数



# Hashset

一个没有重复元素的集合 元素无序 允许null元素 异步



HashSet中含有一个HashMap类型的成员变量map，HashSet的操作函数实际都是通过map实现

HashSet 是如何保证不重复的

向 HashSet 中 add ()元素时，判断元素是否存在不仅要比较hash值，同时还要结合 equals 方法比较

# TreeMap

TreeMap 是一个有序的key-value集合，它是通过红黑树实现的。

该映射根据其键的自然顺序进行排序，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是异步的。 它的iterator 方法返回的迭代器是fail-fast的

# LinkedHashmap

通过维护一个运行于所有条目的双向链表，LinkedHashMap保证了元素迭代的顺序。

Hashcode

算出结果后数组有多大就对几取余，分散于整个数组

为什么重写Object的equals(Object obj)方法尽量要重写Object的hashCode()方法

若只重写equals方法而没有重写hashCode方法的话，则会违反相等的对象必须具有相等的hashCode

将对象的hashcode其所有属性的hashcode和，生成一个尽量唯一的对象hashCode，这样就保证了属性都相等的两个对象它们的HashCode是相等的

