# 集合的数据结构

https://segmentfault.com/a/1190000016127895 --面试题(未整理)

## List介绍

-1）ArrayList：线程不安全。底层采用Object数组

-2）Vector：线程安全。底层采用Object数组

-3）LinkedList：双向循环链表

## Set介绍

-1）HashSet：无序，唯一。底层采用HashMap来实现

-2）LinkedHashSet：继承HashSet类，并且其内部是通过LinkedHashMap来实现的

-3）TreeSet：有序，唯一。红黑树（平衡的排序二叉树）

## Map介绍

-1）HashMap：

JDK1.8之前由数组+链表组成，数组是HashMap的主体，链表是为了解决哈希冲突（“拉链法”解决冲突）。

JDK1.8之后由数组+链表/红黑树组成，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。

-2）LinkedHashMap：

继承HashMap类，由数组+链表/红黑树组成。它在HashMap基础上，添加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。

https://www.cnblogs.com/xiaoxi/p/6170590.html --未整理

-3）HashTable：数组+链表组成

-4）TreeMap：红黑树（平衡的排序二叉树）

会根据key进行排序，能保证所有的key-value按顺序排序。

key排序分类：

1、自然排序：key必须实现Comparable接口，否则将会抛出ClassCastException。

2、自定义排序：创建TreeMap时，可传入一个Comparator对象，该对象负责对TreeMap中所有key进行排序。

-5）ConcurrentHashMap

JDK1.7版：

引入了“分段锁”的概念，将数据分段，每一段数据配一把锁。只有在同一个分段内才存在竞争关系，不同的分段锁之间没有锁竞争。当一个线程占用锁时，其它线程不能在访问该段数据，但其它段的数据可以被访问。

JDK1.8版：

取消了Segment分段锁，采用CAS和synchronized来保证并发安全。数据结构由数组+链表/红黑二叉树组成。

synchronized只锁定当前链表或红黑二叉树的**首节点**，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

HashTable、HashMap、TreeMap区别：

-1）HashTable是线程安全的，键值对不允许null值，其hash数组默认大小为11，增加方式为old\*2+1；

-2）而HashMap是非线程安全的，键值对允许null值，其hash数组默认大小为16，增长方式为2次幂；

-3）TreeMap能够把它保存的记录根据键排序，默认是升序排序。

## 数据结构介绍

-1）数组：

对于指定下标的查找，时间复杂度为O(1)；通过给定值进行查找，需要遍历数组，逐一比较给定关键字和数组元素，时间复杂度为O(n)。对于有序数组，可采用二分查找，可将查找复杂度提高为O(logn)；对于一般的插入删除操作，涉及到数组元素的移动，其平均复杂度为O(n)。

-2）线性链表：

对于链表的新增，删除等操作（在找到指定操作位置后），仅需处理结点间的引用即可，时间复杂度为O(1)，而查找操作需要遍历链表逐一进行比对，复杂度为O(n)。

-3）哈希表：

在哈希表中进行添加，删除，查找等操作，性能十分之高，不考虑哈希冲突的情况下，仅需一次定位即可完成，时间复杂度为O(1)。

哈希冲突：

概念：当我们对某个元素进行哈希运算后的得到了存储位置，在进行插入时发现该位置已被占用。

解决：开放定址法（发生冲突，继续寻找下一块未被占用的存储地址）、再散列函数法、链地址法。

HashMap采用链地址法，即数组+链表的方式。

# HashMap介绍(JDK1.7版)

## 概述

HashMap继承了AbstractMap类，实现了Map、Cloneable、Serializable接口。

key和value都可以为null，映射不是有序的。

HashMap不是线程安全的。需:Map map = Collections.synchronizedMap(new HashMap())

特点：

-1）若定位到的数组位置不含链表（当前entry的next指向null），那么对于查找、添加等操作的时间为O(1);

-2）若定位到的数组位置包含链表，对于添加操作，其时间复杂度也为O(1)，即最新的Entry会插入到链表头部；

-3）若定位到的数组位置包含链表，对于查找操作，此时就需要遍历链表，然后通过 key 对象的 equals 方法逐一比对查找。

>>总觉得2、3有问题!!

//存储时

int hash = hash(key.hashCode);

int index = hash % table.length;（代码中为hash&(table.leng-1)）

table[index] = value;

//取值时

int hash = hash(key.hashCode);

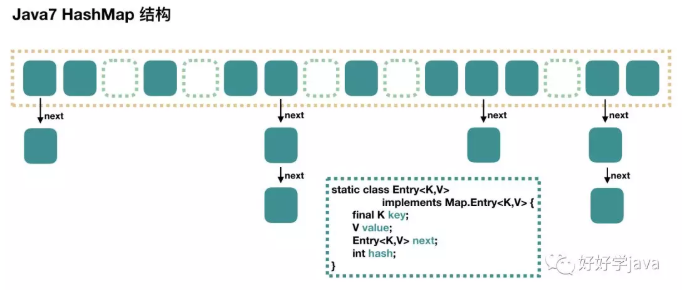
int index = hash % table.length;

return table[index];

## 结构

JDK1.7中的HashMap由“数组+链表”组成。数组是HashMap的主干，数组的每个元素是可构成一条单向链表；链表是为了解决哈希冲突而存在的。

通过key的hashCode计算hash值，再计算数组下标：hash&(length-1)。若多个key对应同一下标则用链表串起来，最新值插到最前面，然后next指向之前的链表。



## 参数说明

-1）**table**：数组主体

transient Entry[] table; //长度为2的n次幂

-2）**Entry**：节点

包含4个属性：key、value、hash值、next指针。

若两个key的hash值相等，它们会以链表的形式存储，Entry为了解决哈希冲突而存在。



Next属性说明：

作用是指向下一个Entry。例如，第一个键值对A进来，通过计算得到index=1，记做:Entry[1]=A;第二个键值对B进来，通过计算其index=1，则:B.next = A;Entry[1]=B;第三个C进来，index=1，则C.next = B, Entry[1] = C;这样我们发现index=1的地方其实存取了C,B,A三个键值对，它们通过next这个属性链接在一起。**也就是说数组中存储的是最后插入的元素。**

-3）**capacity**：容量

即当前数组长度(table.length)，扩容后为当前的2倍（大小始终为2^n）。

-4）初始容量：即哈希表在创建时的容量

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; //16初始容量

static final int MAXIMUM\_CAPACITY = 1 << 30; //最大容量(2^30)

int capacity = 1;

while(capacity < initialCapacity) {

capacity <<= 1;

}

-5）**size**：实际大小

transient int size; //实际存储的Entry个数

-6）**loadFactor**：负载因子，默认0.75

当size已经达到了阈值（threshold），且要插入的位置已经存在元素（这一点在JDK1.8中不需要），那么会触发扩容，扩容后数组大小为原来的2倍(capacity <<= 1)。

特点：加载因子越大，填满的元素越多，利用率提高，但冲突率也相应加大；反之，加载因子越小，填满的元素越少，冲突率减少了，但利用率也不高。

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

-7）**threshold**：阈值

int threshold; //用于判断调整HashMap的容量。threshold=容量\*负载因子

-8）**modCount**：修改次数

transient volatile int modCount; // fail-fast策略：在HashMap迭代时对其进行put、remove等操作会发生异常。

## put过程分析

源码分析：



具体步骤：

-1）对key做null检查。如果key是null，则存储到table[0]，因为null的hash值总是0

-2）根据key的hashCode重新计算hash值，再根据hash值计算数组对应的下标

-3）若数组该位置已存在元素，则遍历该位置元素，若存在相等key则替换旧值后返回，若不存在则往下执行步骤4

-4）将新元素放到该数组的该位置的链头，其新元素的next指向原来的链表（该位置已存在元素），或者next为null（该位置不存在元素）。

实例：

Map<String, String> testMap = new HashMap<String, String>();

testMap.put("Aa", "111");

testMap.put("BB", "222");

testMap.put("Aa", "222");

实例说明：（Aa与BB的hashcode值相同）

-1）第一次放入Aa=1111，在Entry数组下标为4的位置存入；

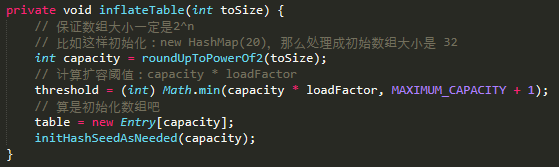
-2）之后放入BB=2222时，由于Aa与BB的hashcode相同，得到Entry数组下标也是4，因此在该位置插入BB=2222，并将BB=2222的next指向Aa=1111，此时形成了一条链表；

-3）最后再放入Aa=2222，又是数组下标为4，循环该位置的Entry链表，发现为Aa=1111，于是将Aa=1111更改为2222。

## 初始化数组（inflateTable方法）

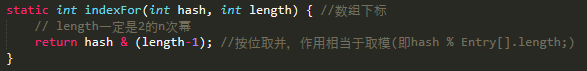
在第一个元素插入HashMap时做一次数组的初始化。

源码分析：



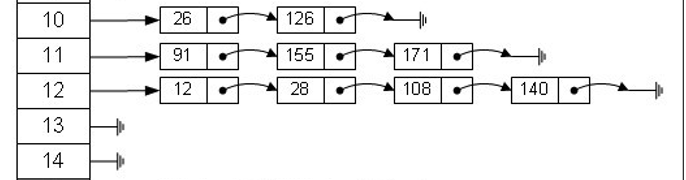
## 计算数组下标（indexFor方法）

源码分析：



该方法通过h&(table.length-1)来得到数组的下标，而HashMap底层数组的长度总是2的n次方，这是HashMap在速度上的优化。

例如在哈希表中，12%16=12,28%16=12,108%16=12,140%16=12。所以12、28、108以及140都存储在数组下标为12的位置。



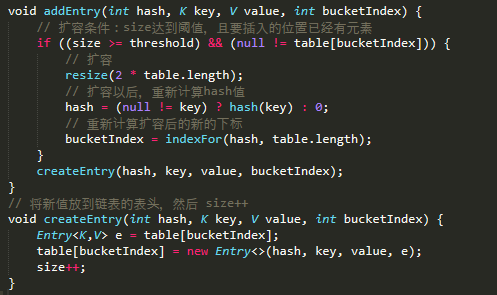
## 添加节点到链表（addEntry方法）

思路：

先判断是否需要扩容，需要的话先扩容，然后再将这个新的数据插入到扩容后的数组的相应位置处的链表的表头。

源码分析：

--貌似扩容只有新增链表的节点时才会触发！！



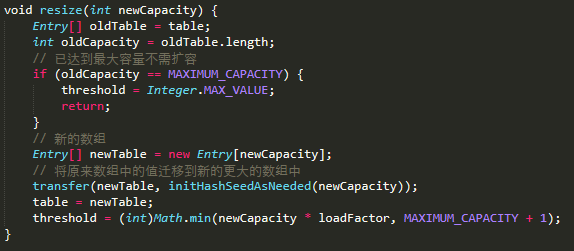
## 数组扩容（resize方法）

hashMap的数组table长度是固定的，当（元素个数size超过了阈值(threshold)，且插入的位置已经存在元素），就会进行数组扩容(resize)，扩容后的数组大小为原来的2倍(capacity <<= 1)。

初始容量(initialCapacity)为16；加载因子(loadFactor)默认为0.75，这是一个折中的值；阈值(threshold)=容量(capacity)\*加载因子(loadFactor)。

默认情况下，数组大小为16，那么当HashMap元素个数超过了16\*0.75=12时，就把数组的大小扩展为2\*16=32。即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

源码分析：



迁移机制：

在迁移过程中，会将原来table[i]中的链表的所有节点，分拆到新的数组newTable[i]和newTable[i+oldLength]位置上。例如：原来数组长度是 16，那么扩容后，原来 table[0] 处的链表中的所有元素会被分配到新数组中 newTable[0] 和 newTable[16] 这两个位置上。

## get过程分析

具体流程：

-1）根据key计算hash值，进而计算数组下标：hash&(length-1)

-2）遍历该数组下标处的链表，返回相等key的值

源码分析：



## 相关问题

1、Fail-Fast机制

由于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，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

该策略主要通过modCount属性(意思为修改次数)实现，对HashMap修改会增加这个值，在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。

transient volatile int modCount;

2、HasnMap遍历

通过Map.entrySet遍历key和value（推荐）

for(Map.Entry<String, String> entry : map.entrySet()){

System.out.println("key= "+entry.getKey()+" and value= "+entry.getValue());

}

3、key类型为何是String、Integer等wrapper类？

答：String是不可变的，也重写了equals和hashCode方法。不可变是很必要的，因为要计算hashCode()，就要防止键值改变。若键值在放入或获取时返回不同的hashCode值，就不能从HashMap中找到你想要的对象。

4、多线程环境中使用 HashMap 会出现什么问题？

答：put() 方法可能会触发它内部的数组扩容（resize），这个过程可能会造成无限循环，这就是为什么你应该换用 Hashtable 或 ConcurrentHashMap（后者更佳）。

5、覆写HashMap中的 hashCode() 方法会不会对性能有影响？

答：据我所知，hashCode() 方法实现的不好，可能会导致 HashMap 频繁出现冲突，增加了将对象放入 HashMap 的耗时。

不过从 Java 8 开始这种情况有所改善，因为冲突到达一定级别后，HashMap 会改用二叉树而不是链表来保存内容，这样时间复杂度就从 O(N) 降低到了 O(log N)。

6、HashMap的长度为什么是2的n次方？

这样使得转二进制全部为111…的形式！！！

答：在indexFor(int hash)中，(length-1)转化为二进制必定为1111…的形式，而数组下标计算为：hash&(length-1)，因此能保证数组的每个位置都能存值。

若length=15，则length-1为1110，在与hash&(length-1)时，最后一位都为0，而像0001，0011这些位置永远都不能存放元素，从而造成空间浪费。

# HashMap介绍(JDK1.8版)

微信公众号：好好学java

## 简介

JDK1.8中的HashMap由“数组+链表/红黑树”组成。

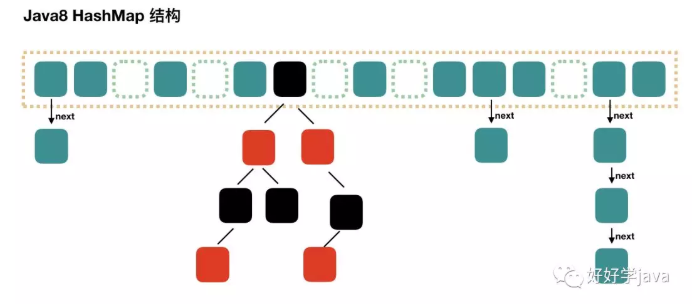
问题：在JDK1.7中，查找某元素时，根据hash值能快速定位到数据的具体下标，但之后要顺着链表一个个比较才能找到该元素，其时间复杂度为O(n)。

解决：为了降低这部分开销，JDK1.8中，HashMap在链表中元素个数超过8个后，会将链表转换为红黑树，其时间复杂度为O(logN)。

TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

如图所示：

注：图中的元素不会有这么多！！（因为这么多元素时早就扩容了）



节点描述：

Jdk1.7中使用Entry来代表每个HashMap中的数据节点；jdk1.8中使用Node；

它们基本无区别，都是key、value、hash和next四个属性。

Node只能用于链表情况，红黑树情况需使用TreeNode。

在数组中，根据第一个节点数据类型（Node或TreeNode）来判断该位置是链表还是红黑树。

## put过程分析

源码分析：

public V put(K key, V value) {

return putVal(hash(key), key, value, false, true);

}

// onlyIfAbsent 为true，则只有不存在key时才会进行put操作

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,

boolean evict) {

Node<K,V>[] tab; Node<K,V> p; int n, i;

// 第一次put时，会触发resize()进行扩容

// 第一次resize和后续的扩容不同，因为这次是数组从null初始化为默认的16或自定义的初始容量

if ((tab = table) == null || (n = tab.length) == 0)

n = (tab = resize()).length;

// 1.计算具体的数组下标，若该位置无节点，则直接创建一个Node

if ((p = tab[i = (n - 1) & hash]) == null)

tab[i] = newNode(hash, key, value, null);

// 2.数组的该位置有节点

else {

Node<K,V> e; K k;

// 2.1 判断该位置的第一个数据和我们要插入的数据是否相等，是则取出该节点

if (p.hash == hash &&

((k = p.key) == key || (key != null && key.equals(k))))

e = p;

// 2.2 与第一个数据不相等，且该节点类型是TreeNode时，调用红黑树的插值方法

else if (p instanceof TreeNode)

e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

// 2.3 否则该节点类型是Node，说明数组的该位置是一个链表

else {

for (int binCount = 0; ; ++binCount) {

// 2.3.1 当链表循环到最后一个节点是，将需插入元素插入到链表在最后(jdk1.7是插入到链表最前面)

if ((e = p.next) == null) {

p.next = newNode(hash, key, value, null);

// 当链表的节点大于8时，将链表转换为红黑树

if (binCount >= TREEIFY\_THRESHOLD - 1)

treeifyBin(tab, hash);

break;

}

// 2.3.2 当链表中找到了相等的元素

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

// 跳出循环，e为链表中与需插入元素相等的节点

break;

p = e;

}

}

// e!=null 说明存在旧值的key与插入值的key相等

// 覆盖旧值后再返回旧值

if (e != null) {

V oldValue = e.value;

if (!onlyIfAbsent || oldValue == null)

e.value = value;

afterNodeAccess(e);

return oldValue;

}

}

++modCount;

// 所有元素个数size大于阈值threshold时触发扩容

if (++size > threshold)

resize();

afterNodeInsertion(evict);

return null;

}

## 扩容机制

resize()方法用于初始化数组、或数组扩容。每次扩容后，容量为原来的2倍，并进行数据迁移。

源码分析：

final Node<K,V>[] resize() {

Node<K,V>[] oldTab = table;

int oldCap = (oldTab == null) ? 0 : oldTab.length; // 旧容量(数组长度)

int oldThr = threshold; // 旧阈值

int newCap, newThr = 0; // 新容量，新阈值

if (oldCap > 0) {

// 旧容量已达到最大

if (oldCap >= MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return oldTab;

}

// 将数组大小扩大一倍

else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

// 将阈值扩大一倍

newThr = oldThr << 1;

}

// 对应使用new HashMap(int initialCapacity)初始化后，第一次put的时候

else if (oldThr > 0)

newCap = oldThr;

// 对应使用new HashMap()初始化后，第一次put的时候

else {

newCap = DEFAULT\_INITIAL\_CAPACITY;

newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

}

if (newThr == 0) {

float ft = (float)newCap \* loadFactor;

newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?

(int)ft : Integer.MAX\_VALUE);

}

threshold = newThr;

// 用新的数组大小初始化新的数组

Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];

table = newTab; // 如果是初始化数组，到这里就结束了，返回 newTab 即可

if (oldTab != null) {

// 开始遍历原数组，进行数据迁移。

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

Node<K,V> e;

if ((e = oldTab[j]) != null) {

oldTab[j] = null;

// 如果该数组位置上只有单个元素，那就简单了，简单迁移这个元素就可以了

if (e.next == null)

newTab[e.hash & (newCap - 1)] = e;

// 如果是红黑树，具体我们就不展开了

else if (e instanceof TreeNode)

((TreeNode<K,V>)e).split(this, newTab, j, oldCap);

else {

// 这块是处理链表的情况，

// 需要将此链表拆成两个链表，放到新的数组中，并且保留原来的先后顺序

// loHead、loTail 对应一条链表，hiHead、hiTail 对应另一条链表，代码还是比较简单的

Node<K,V> loHead = null, loTail = null;

Node<K,V> hiHead = null, hiTail = null;

Node<K,V> next;

do {

next = e.next;

if ((e.hash & oldCap) == 0) {

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

}

else {

if (hiTail == null)

hiHead = e;

else

hiTail.next = e;

hiTail = e;

}

} while ((e = next) != null);

if (loTail != null) {

loTail.next = null;

// 第一条链表

newTab[j] = loHead;

}

if (hiTail != null) {

hiTail.next = null;

// 第二条链表的新的位置是 j + oldCap，这个很好理解

newTab[j + oldCap] = hiHead;

}

}

}

}

}

return newTab;

}

## get过程分析

具体流程：

-1）计算hey的hash值，根据hash值找到对应数组下标：hash&(length-1)

-2）判断数组该位置的元素是否是我们要找的，若不是走则第三步

-3）判断该元素类型是否为TreeNode，若是则使用红黑树的方法取数据，若不是则走第四步

-4）遍历链表，直到找到相等的key

源码分析：

public V get(Object key) {

Node<K,V> e;

return (e = getNode(hash(key), key)) == null ? null : e.value;

}

final Node<K,V> getNode(int hash, Object key) {

Node<K,V>[] tab; Node<K,V> first, e; int n; K k;

if ((tab = table) != null && (n = tab.length) > 0 &&

(first = tab[(n - 1) & hash]) != null) {

// 判断第一个节点是不是就是需要的

if (first.hash == hash && // always check first node

((k = first.key) == key || (key != null && key.equals(k))))

return first;

if ((e = first.next) != null) {

// 判断是否是红黑树

if (first instanceof TreeNode)

return ((TreeNode<K,V>)first).getTreeNode(hash, key);

// 链表遍历

do {

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

return e;

} while ((e = e.next) != null);

}

}

return null;

}

# ConcurrentHashMap介绍（JDK1.7版）

## 概念

JDK1.7版：

引入了“分段锁”的概念，将数据分段，每一段数据配一把锁。只有在同一个分段内才存在竞争关系，不同的分段锁之间没有锁竞争。当一个线程占用锁时，其它线程不能在访问该段数据，但其它段的数据可以被访问。

JDK1.8版：

取消了Segment分段锁，采用CAS和synchronized来保证并发安全。数据结构由数组+链表/红黑二叉树组成。

synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

## 特点

-1）ConcurrentHashMap不允许使用null键和null值；

-2）ConcurrentHashMap的key是唯一的，value值可重复；

-3）ConcurrentHashMap是无序的。

为什么使用ConcurrentHashMap？

HashMap是非线程安全的，当涉及到多线程，并且有读写的过程时，HashMap就会fail-fast。

解决HashMap的同步问题：1）HashTable 2）Collections.synchronizedMap(hashMap)

上面两种方式都是对整个hash表加上同步锁，在锁表的期间，别的线程就需等待其解锁后才能继续操作，这样性能不高，因此引入了ConcurrentHashMap。只有在同一个分段内才存在竞争关系，不同的分段锁之间没有锁竞争。

## 数据结构

ConcurrentHashMap是一个Segment数组；Segment描述为“分段锁”，通过继承ReentranLock来进行加锁，因此每次加锁的操作只锁住一个segment。一个Segment包含一个HashEntry数组，每个HashEntry是一个链表结构的元素。当对HashEntry数组的数据修改时，必须先获取对应Segment锁。

ConcurrentHashMap是由Segment数组和HashEntry数组组成。Segment是一种可重入锁，在ConcurrentHashMap里扮演锁的角色；HashEntry则用于存键值对数据。

每个HashEntry是一个链表结构的元素，每个Segment守护着一个hashEntry数组里的元素，当对HashEntry数组的数据修改时，必须首先获得它对应的Segment锁。

结构如下：

1.一个ConcurrentHashMap里包含一个Segment数组

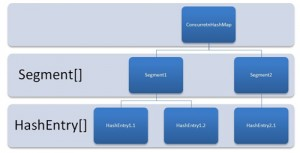
2.一个Segment里包含了一个HashEntry数组，Segment由数组+链表组成，并继承了可重入锁ReentranLock

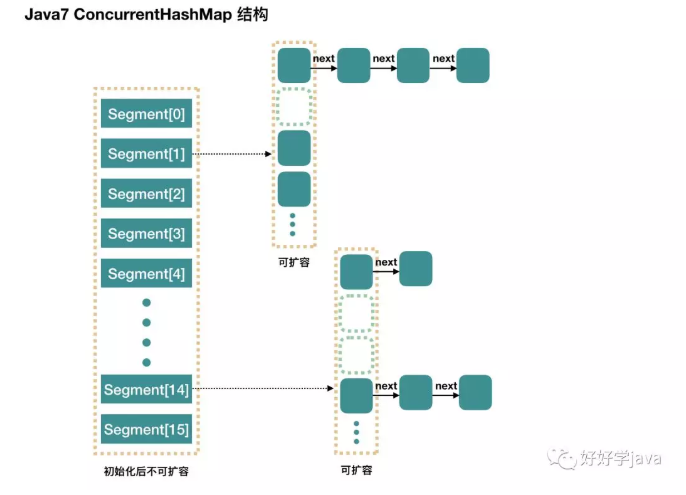
class ConcurrentHashMap<K, V>: //1

final Segment<K,V>[] segments;

class Segment<K,V> extends ReentrantLock: //2

transient volatile HashEntry<K,V>[] table;





## 初始化

初始化参数：

-1）initialCapacity：初始容量

指整个ConcurrentHashMap的初始容量，实际操作时需平均分给每个Segment。

-2）loadFactor：负载因子

Segment数组不可以扩容，所以这个负载因子是给每个Segment内部使用的。

-3）concurrencyLevel：并行级别、并发数、Segment数

默认是16，即ConcurrentHashMap有16个Segment。所以理论上最多可以同时支持16个线程并发写，只要它们的操作分别分布在不同的Segment上。该参数一旦初始化后不可以再次扩容。

源码分析：（看不懂！！）

public ConcurrentHashMap(int initialCapacity,

float loadFactor, int concurrencyLevel) {

if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)

throw new IllegalArgumentException();

if (concurrencyLevel > MAX\_SEGMENTS)

concurrencyLevel = MAX\_SEGMENTS;

// Find power-of-two sizes best matching arguments

int sshift = 0;

int ssize = 1;

// 计算并行级别 ssize，因为要保持并行级别是 2 的 n 次方

while (ssize < concurrencyLevel) {

++sshift;

ssize <<= 1;

}

// 我们这里先不要那么烧脑，用默认值，concurrencyLevel 为 16，sshift 为 4

// 那么计算出 segmentShift 为 28，segmentMask 为 15，后面会用到这两个值

this.segmentShift = 32 - sshift;

this.segmentMask = ssize - 1;

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

// initialCapacity 是设置整个 map 初始的大小，

// 这里根据 initialCapacity 计算 Segment 数组中每个位置可以分到的大小

// 如 initialCapacity 为 64，那么每个 Segment 或称之为"槽"可以分到 4 个

int c = initialCapacity / ssize;

if (c \* ssize < initialCapacity)

++c;

// 默认 MIN\_SEGMENT\_TABLE\_CAPACITY 是 2，这个值也是有讲究的，因为这样的话，对于具体的槽上，

// 插入一个元素不至于扩容，插入第二个的时候才会扩容

int cap = MIN\_SEGMENT\_TABLE\_CAPACITY;

while (cap < c)

cap <<= 1;

// 创建 Segment 数组，

// 并创建数组的第一个元素 segment[0]

Segment<K,V> s0 =

new Segment<K,V>(loadFactor, (int)(cap \* loadFactor),

(HashEntry<K,V>[])new HashEntry[cap]);

Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];

// 往数组写入 segment[0]

UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]

this.segments = ss;

}

初始化完成后，我们得到一个Segment数组。

new ConcurrentHashMap()无参构造器初始化：

-1）Segment数组长度为16，不可以扩容；

-2）Segment[i]的默认大小为2，负载因子是0.75，得出阈值为1.5。因此以后插入第一个元素不会触发扩容，插入第二个元素会进行第一次扩容；

-3）这里初始化了segment[0]，其它位置还是null

-4）当前移位数(segmentShift)的值为32-4=28，掩码(segmentMask)为16-1=15。

## put过程分析

源码分析：

public V put(K key, V value) {

Segment<K,V> s;

if (value == null)

throw new NullPointerException();

// 1. 计算key的hash值

int hash = hash(key);

// 2. 根据 hash 值找到 Segment 数组中的位置 j

// hash 是 32 位，无符号右移 segmentShift(28) 位，剩下低 4 位，

// 然后和 segmentMask(15) 做一次与操作，也就是说 j 是 hash 值的最后 4 位，也就是槽的数组下标

int j = (hash >>> segmentShift) & segmentMask;

// 刚刚说了，初始化的时候初始化了 segment[0]，但是其他位置还是 null，

// ensureSegment(j) 对 segment[j] 进行初始化

if ((s = (Segment<K,V>)UNSAFE.getObject // nonvolatile; recheck

(segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment

s = ensureSegment(j);

// 3. 插入新值到 槽 s 中

return s.put(key, hash, value, false);

}

<https://mp.weixin.qq.com/s?__biz=MzI2OTQ4OTQ1NQ==&mid=2247484007&idx=1&sn=b8bb6c49e604ee593f3fb9b87ba3d0ae&chksm=eadec627dda94f316e8b16b34c0c8f9951ec8ed0f3826a84e7e8ddac00160aa6e985c260427f&mpshare=1&scene=23&srcid=1026IkUKkqAJVVBhUIyBFXhU#rd>

--未完成！！！

# HashSet介绍

## 概述

基于HashMap实现，只使用了HashMap的key来实现各种特征。map是整个HashSet的核心，而PRESENT则是用来造一个假的value来用的。

其结构如下：

class HashSet<E>：

private transient HashMap<E,Object> map;

private static final Object PRESENT = new Object();

## 源码分析

public boolean add(E e) { //添加

return map.put(e, PRESENT) == null;

}

public boolean remove(Object o) { //移除

return map.remove(0) == PRESENT;

}

public boolean contains(Object o) { //包含

return map.containsKey(o);

}

public int size() { //集合大小

return map.size();

}

# ArrayList介绍

## 概述

以数组实现。第一次默认创建大小为10的数组。超出容量时会以1.5倍增长，用Systemt.arraycopy()复制到新的数组。

按数组下标访问元素：get(i)/set(i, e)的性能很高，这是数组的基本优势。

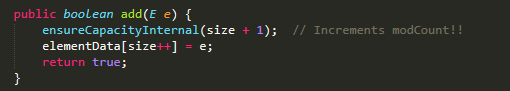
直接在数组末尾加入元素：add(e)的性能页很高。

但按下标插入、删除元素：add(i, e)/remove(i)/remove(e)，则要用System.arraycopy()来移动部分受影响的元素，因此性能会变差。

## 源码分析

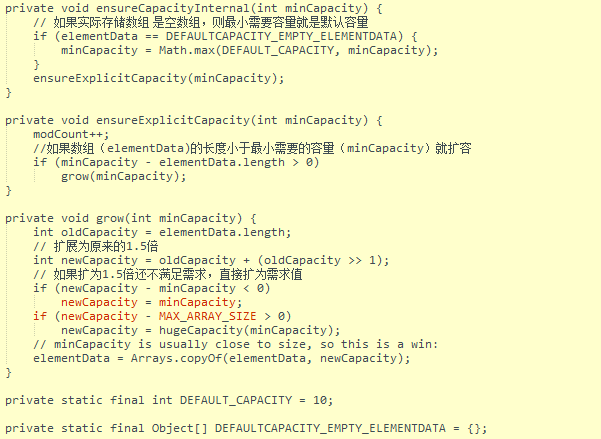
### add(e)方法

--将元素放到末尾



ensureCapacityInternal函数：

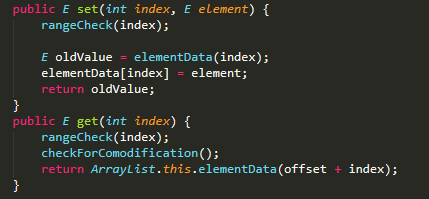
--自动扩容机制，即确保内部容量。



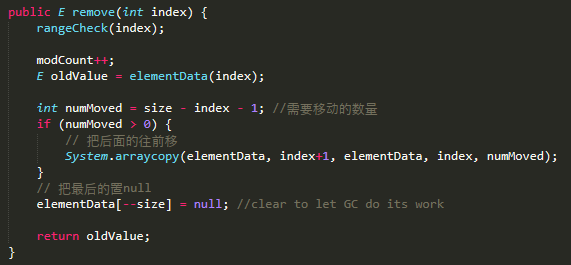
>>位运算，右移动一位。整体相当于newCapacity = oldCapacity + 0.5\*oldCapacity;

假如有20个数据需要添加，那么会分别在第一次的时候，将ArrayList的容量变为10；之后扩容会按照1.5倍增长。也就是当添加第11个数据的时候，Arraylist继续扩容变为10\*1.5=15；当添加第16个数据时，继续扩容变为15 \* 1.5 =22个。

### set和get函数



### remove函数



## ArrayList与LinkedList异同

-1）ArrayList和LinkedList都是线程不安全的；

-2）ArrayList底层使用的是Object数组，LinkedList底层使用双向链表数据结构；

-3）ArrayList采用数组存储，所以插入和删除的时间复杂度受元素位置的影响；LinkedList采用链表存储，所以插入、删除元素的复杂度不受元素位置的影响，都是O(1)；

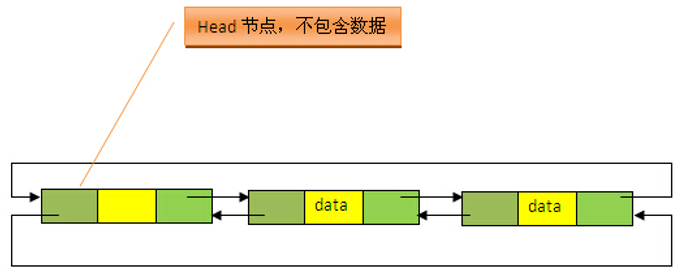
-4）LinkedList不支持高效的随机元素访问，而ArrayList实现了RandomAccess接口，所以有随机访问功能。

# LinkedList介绍

## 概述

LinkedList是基于双向链表实现。插入和删除操作比ArrayList更高效，但随机访问的效率要比ArrayList的差。

LinkedList数据结构原理：基于双向循环链表，且头结点不存放数据。



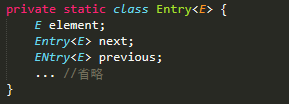
两个私有属性：

private transient Entry<E> header = new Entry<E>(null, null, null); //它是双向链表的头结点

private transient int size = 0; //双向链表中结点实例的个数

Entry的数据结构：

静态内部类，包含成员变量：previous，next，element。其中，element是该结点的值；previous指向该结点的上一结点；next指向下一结点。



# 未整理内容

## 线程不安全的HashMap

HashMap的线程不安全主要体现在resize时的死循环及使用迭代器时的fast-fail上。

附：源代码基于JDK1.7.0\_67

### HashMap数据结构

主要有数组+链表。数组存储区间连续，占用内存较多，寻址容易，插入和删除需要移动元素。链表存储区间离散，占用内存较少，寻址困难，插入和删除容易。

### 寻址方式