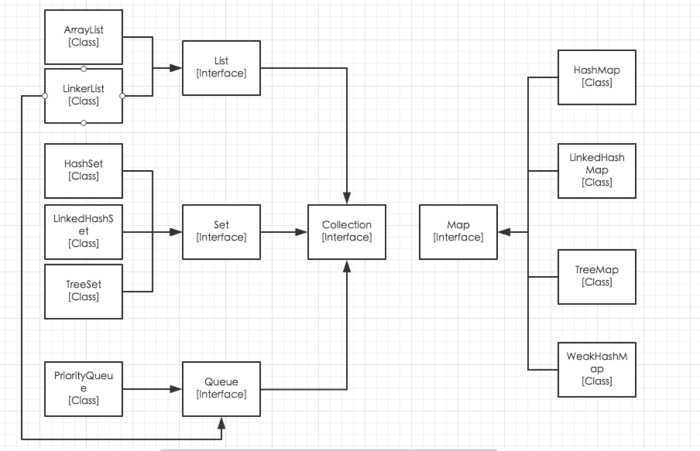
先弄份jdk源码：

我下的1.8版本的，OpenJDK，在Eclipse中想查看一些类的源码。步骤如下：  
1. 从[http://download.java.net/openjdk/jdk8/](http://download.java.net/openjdk/jdk6/" \t "https://zhidao.baidu.com/question/_blank)下载OpenJDK源码；  
2. 解压缩源码；  
3. Eclipse--->Window--->Preferences--->Java--->Installed JREs--->选择jdk8（我用的版本-\_-|||...）--->Edit--->JRE system libraries:--->rt.jar--->Source Attachment--->External Folder...；  
4. 选择源码解压所在目录/jdk/src/share/classes/;  
5. F5刷一下。

java容器中两大顶层集合接口类：Map Collection



# 1 Collection

单个元素对象的序列

**public** **interface** Collection<E> **extends** Iterable<E> {

...

}

泛型<E> Collection元素对象的类型，继承的Iterable是一个元素遍历的接口，采用hasNext next() 进行变量。

常用接口方法：

**boolean** add(E e);

**boolean** removeAll(Collection<?> c);

**boolean** addAll(Collection<? **extends** E> c);

**boolean** containsAll(Collection<?> c); isEmpty()

Iterator<E> iterator();

**boolean** remove(Object o);

**boolean** retainAll(Collection<?> c);

（\*\*ps:这个平时倒是没注意，感觉挺好用的接口，保留指定的集合\*\*）

**int** size();

Object[] toArray();转为数组

## 1.1 List (可重复有序)

List提供了一种特殊的iterator遍历器——ListIterator，继承Iterator。这种遍历器允许遍历时插入，替换，删除，双向访问。 并且还有一个重载方法允许从一个指定位置开始遍历。

ListIterator:

**public** **interface** ListIterator<E> **extends** Iterator<E> {

**boolean** hasNext();

E next();

**boolean** hasPrevious();

E previous();

**int** nextIndex();

**int** previousIndex();

**void** remove();

**void** set(E e);

**void** add(E e);

}

区别iterator接口，多了很多新方法，List是一个有序集合，所以在这实现了一个ListIteractor，可以在遍历过程中进行元素操作，并且可以双向访问。

### 1.1.1 ArrayList

可以插入null 可变数组 时间复杂度O(n) 非线程安全的

所有元素都是保存在一个Object数组中，然后通过size控制长度。

transient Object[] elementData;（transient：关键字有提到：非序列化）

private int size;

看下add 源码：

**public** **boolean** add(E e) {

ensureCapacityInternal(size + 1);

elementData[size++] = e;

**return** **true**;

}

**private** **void** ensureCapacityInternal(**int** minCapacity) {

**if** (elementData == ***EMPTY\_ELEMENTDATA***) {

minCapacity = Math.*max*(***DEFAULT\_CAPACITY***, minCapacity);

}

ensureExplicitCapacity(minCapacity);

}

**private** **void** ensureExplicitCapacity(**int** minCapacity) {

modCount++;

**if** (minCapacity - elementData.length > 0)

grow(minCapacity);

}

**private** **void** grow(**int** minCapacity) {

// overflow-conscious code

**int** oldCapacity = elementData.length;

**int** newCapacity = oldCapacity + (oldCapacity >> 1);

**if** (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

**if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0)

newCapacity = *hugeCapacity*(minCapacity);

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.*copyOf*(elementData, newCapacity);

}

主要思路：每次添加新数据时都会判断长度，不够的话会调用Arrays.copyOf把之前数据复制到新的扩容后的数组里

Remove:

**public** E remove(**int** index) {

rangeCheck(index);

modCount++;

E oldValue = elementData(index);

**int** numMoved = size - index - 1;

**if** (numMoved > 0)

System.*arraycopy*(elementData, index+1, elementData, index,numMoved);

elementData[--size] = **null**; // clear to let GC do its work

**return** oldValue;

}

其实就是直接使用System.arraycopy把需要删除index后面的都往前移一位然后再把最后一个去掉。

### 1.1.2 LinkedList

LinkedList是一个链表维护的序列容器。和ArrayList都是序列容器，一个使用数组存储，一个使用链表存储。

数组（ArrayList） 链表(LinkedList)

查找 效率更高，有索引 从头开始查询

插入、删除 效率低，需要数据整体 只需改变在插入位置改变前后

迁移 数据链表头、尾链接

内存申请 数组在超出长度时，需 动态创建

申请更大数组迁移

首先是链表的节点的定义,非常简单的一个双向链表.

**private** **static** **class** Node<E> {

E item;

Node<E> next;

Node<E> prev;

Node(Node<E> prev, E element, Node<E> next) {

**this**.item = element;

**this**.next = next;

**this**.prev = prev;

}

}

**transient** **int** size = 0;

**transient** Node<E> first; //头节点

**transient** Node<E> last; //尾节点

Add方法：

**public** **boolean** add(E e) {

linkLast(e);

**return** **true**;

}

**void** linkLast(E e) {

**final** Node<E> l = last;

**final** Node<E> newNode = **new** Node<>(l, e, **null**);

last = newNode;

**if** (l == **null**)

first = newNode;

**else**

l.next = newNode;

size++;

modCount++;

}

很简单，每次新增一个数据都会先初始化一个节点，判断当前的头节点是否有值，在安排新加入的节点位置。

Remove:

**public** E remove(**int** index) {

checkElementIndex(index);

**return** unlink(node(index));

}

**private** **void** checkElementIndex(**int** index) {//检验index

**if** (!isElementIndex(index))

**throw** **new** IndexOutOfBoundsException(outOfBoundsMsg(index));

}

Node<E> node(**int** index) {//从链表中取到要移除的节点node

**if** (index < (size >> 1)) {

Node<E> x = first;

**for** (**int** i = 0; i < index; i++)

x = x.next;

**return** x;

} **else** {

Node<E> x = last;

**for** (**int** i = size - 1; i > index; i--)

x = x.prev;

**return** x;

}

}

E unlink(Node<E> x) {//把移除节点的前节点prev.next关联到next,把后节点的next.prev关联到prev

// assert x != null;

**final** E element = x.item;

**final** Node<E> next = x.next;

**final** Node<E> prev = x.prev;

**if** (prev == **null**) {

first = next;

} **else** {

prev.next = next;

x.prev = **null**;

}

**if** (next == **null**) {

last = prev;

} **else** {

next.prev = prev;

x.next = **null**;

}

x.item = **null**;

size--;

modCount++;

**return** element;

}

代码不难，细看一遍就懂了，size >> 1是/2 size<<1是 \*2（移位操作 比java 快很多）。同时LinkedList还实现了Deque接口，Deque接口是继承Queue的。所以LinkedList还支持队列的pop，push，peek操作。

总结

| List实现 | 使用场景 | 数据结构 |
| --- | --- | --- |
| ArrayList | 数组形式访问List链式集合数据，元素可重复，访问元素较快 | 数组 |
| LinkedList | 链表方式的List链式集合，元素可重复，元素的插入删除较快 | 双向链表 |

## 1.2 Set (不重复)

### 1.2.1 HashSet

HashSet实现了Set接口，基于HashMap进行存储。遍历时不保证顺序，并且不保证下次遍历的顺序和之前一样。HashSet中允许null元素。

**private** **transient** HashMap<E,Object> map;

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

HashSet的集合其实就是HashMap的key的集合，然后HashMap的val默认都是PRESENT。HashMap的定义即是key不重复的集合。使用HashMap实现，这样HashSet就不需要再实现一遍。

### 1.2.2 TreeSet

TreeSet即是一组有次序的集合，如果没有指定排序规则Comparator，则会按照自然排序。（自然排序即e1.compareTo(e2) == 0作为比较）注意：TreeSet内的元素必须实现Comparable接口。TreeSet源码的算法即基于TreeMap，具体算法在说明TreeMap的时候进行解释。

### 1.2.3 LinkedHashSet

LinkedHashSet的核心概念相对于HashSet来说就是一个可以保持顺序的Set集合。继承于HashSet，HashSet是无序的，LinkedHashSet会根据add，remove这些操作的顺序在遍历时返回固定的集合顺序。这个顺序不是元素的大小顺序，而是可以保证2次遍历的顺序是一样的。类似HashSet基于HashMap的源码实现，LinkedHashSet的数据结构是基于LinkedHashMap。过多的就不说了。

总结

| Set实现 | 使用场景 | 数据结构 |
| --- | --- | --- |
| HashSet | 无序的、无重复的数据集合 | 基于HashMap |
| LinkedSet | 维护次序的HashSet | 基于LinkedHashMap |
| TreeSet | 保持元素大小次序的集合，元素需要实现Comparable接口 | 基于TreeMap |

### 1.2.4 Quene

很早之前用过，很少公司项目会用到

# 2 Map

键值对对象的集合

Key-value key值不重复(这个接口替代了原来的一个抽象类Dictionary)

**public** **interface** Map<K,V> {

**boolean** containsKey(Object key);

Int size();

V get(Object key);

V put(K key, V value);

...

Set<K> keySet(); key值集合①

Collection<V> values();values集合②

Set<Map.Entry<K, V>> entrySet(); key-value集合③

interface Entry<K,V> {

K getKey();

V getValue();

...

}

}

泛型<K,V> 是对应key和value类型，还有一个内部接口Entry,每一个键值对都是一个Entry的实例关系对象，Map可以说是Entyr的集合，entry内部包含key、value。Entry关系到HashMap的存储数据模式——哈希散列表。

Map提供了3种遍历的方法：

①通过获取key的set集合，因为key值不重复，所以是set而不是list。因为Collection继承了Iteralbe，通过迭代遍历所有key，再通过get方法获取value

Map<String,Object> map = **new** HashMap<String,Object>();

map.put("1", "a");

Set<String> setKey = map.keySet();

Iterator iter = setKey.iterator();

**while**(iter.hasNext()) {

String key = (String)iter.next();

System.***out***.println(key);

System.***out***.print(map.get(key));

}

②获取value的集合，只能获取value值，不能取到对应的key值。

Map<String,Object> map = **new** HashMap<String,Object>();

map.put("1", "a");

Collection<Object> cvalues = map.values();

System.***out***.println(cvalues);

③获取Entry的set集合，遍历集合获取单个Entry，再得到key和value

Map<String,Object> map = **new** HashMap<String,Object>();

map.put("1", "a");

Set<Entry<String, Object>> setM = map.entrySet();

Iterator ite = setM.iterator();

**while**(ite.hasNext()) {

Entry entry = (Entry) ite.next();

System.***out***.println(entry.getKey());

System.***out***.println(entry.getValue());

}

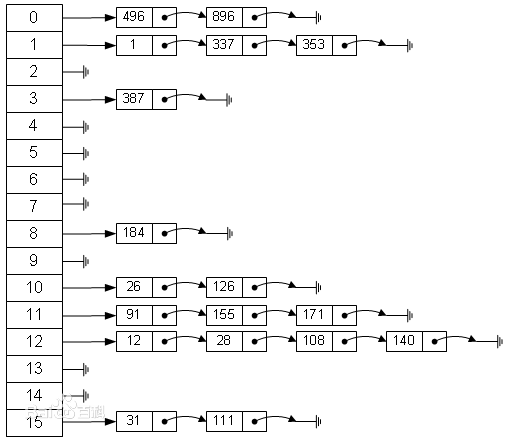
效率上③>①，②用的很少。

## 2.1 HashMap

HashMap就是最基础最常用的一种Map，它无序，以散列表的方式进行存储。之前提到过，HashSet就是基于HashMap，只使用了HashMap的key作为单个元素存储。

HashMap的访问方式就是继承于Map的最基础的3种方式，详细见上。在这里我具体分析一下HashMap的底层数据结构的实现。

在看HashMap源码前，先理解一下他的存储方式-散列表（哈希表）。像之前提到过的用数组存储，用链表存储。哈希表是使用数组和链表的组合的方式进行存储。(具体哈希表的概念自行搜索)如下图就是HashMap采用的存储方法。



hash得到数值，放到数组table中，如果遇到冲突则以链表方式挂在对应元素的next链表下方。

内部源码：

**transient** Node<K,V>[] table;

**static** **class** Node<K,V> **implements** Map.Entry<K,V> {

**final** **int** hash;

**final** K key;

V value;

Node<K,V> next;

Node(**int** hash, K key, V value, Node<K,V> next) {

**this**.hash = hash;

**this**.key = key;

**this**.value = value;

**this**.next = next;

}

**public** **final** K getKey() { **return** key; }

**public** **final** V getValue() { **return** value; }

**public** **final** String toString() { **return** key + "=" + value; }

**public** **final** **int** hashCode() {

**return** Objects.*hashCode*(key) ^ Objects.*hashCode*(value);

}

**public** **final** V setValue(V newValue) {

V oldValue = value;

value = newValue;

**return** oldValue;

}

**public** **final** **boolean** equals(Object o) {

**if** (o == **this**)

**return** **true**;

**if** (o **instanceof** Map.Entry) {

Map.Entry<?,?> e = (Map.Entry<?,?>)o;

**if** (Objects.*equals*(key, e.getKey()) &&

Objects.*equals*(value, e.getValue()))

**return** **true**;

}

**return** **false**;

}

}

Get方法源码：

**public** V get(Object key) {

Node<K,V> e;

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

}

//hash key的hash值

**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**) {

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

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

**return** first;

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

**if** (first **instanceof** TreeNode)

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

**do** {

**if** (e.hash == hash &&

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

**return** e;

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

}

}

**return** **null**;

}

上面代码中看出先根据hash值和数组长度作且运算得出下标索引。如果存在判断hash值是否完全一致，如果不完全一致则next链表向下找一致的hash值。

Put源码：

**public** V put(K key, V value) {

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

}

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

**boolean** evict) {

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

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

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

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

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

**else** {

Node<K,V> e; K k;

**if** (p.hash == hash &&

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

e = p;

**else** **if** (p **instanceof** TreeNode)

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

**else** {

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

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

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

**if** (binCount >= ***TREEIFY\_THRESHOLD*** - 1) // -1 for 1st

treeifyBin(tab, hash);

**break**;

}

**if** (e.hash == hash &&

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

**break**;

p = e;

}

}

**if** (e != **null**) { // existing mapping for key

V oldValue = e.value;

**if** (!onlyIfAbsent || oldValue == **null**)

e.value = value;

afterNodeAccess(e);

**return** oldValue;

}

}

++modCount;

**if** (++size > threshold)

resize();

afterNodeInsertion(evict);

**return** **null**;

}

上面是put的核心源码，即查找hash值所在索引是否有元素，没有的话new一个Node直接放在table中。如果已经有Node了，就遍历该Node的next，将新元素放到最后。

HashMap的遍历，是从数组遍历第一个非空的元素，然后再根据这个元素访问其next下的所有Node。因为第一个元素不是一定从数组的0开始，所以HashMap是无序遍历。

## 2.2 LinkedHashMap

LinkedHashMap相对于HashMap来说区别是，LinkedHashMap遍历的时候具有顺序，可以保存插入的顺序，（还可以设置最近访问的元素也放在前面，即LRU）。其实LinkedHashMap的存储还是跟HashMap一样，采用哈希表方法存储，只不过LinkedHashMap多维护了一份head，tail链表。

**transient** LinkedHashMap.Entry<K,V> head;

**transient** LinkedHashMap.Entry<K,V> tail;

即在创建新Node的时候将新Node放到最后，这样遍历的时候不再像HashMap一样，从数组开始判断第一个非空元素，而是直接从表头进行遍历。这样即满足有序遍历。

## 2.3 TreeMap

TreeMap平时用的不多，TreeMap会实现SortMap接口，定义一个排序规则，这样当遍历TreeMap的时候，会根据规定的排序规则返回元素。

## 2.4 Hashtable

（1）Hashtable 是一个散列表，它存储的内容是键值对(key-value)映射。

（2）Hashtable 继承于Dictionary，实现了Map、Cloneable、java.io.Serializable接口。

（3）Hashtable 的函数都是同步的，这意味着它是线程安全的。它的key、value都不可以为null。

Hashtable 和 HashMap 不同：

1 线程安全 HashTable线程安全(get/put所有相关操作都是synchronized)但是性能差 HashMap 线程不安全

2 key value 是否为null hashtable 可以 value 不能为null，HashMap 可以为null

3 继承不同 hashtable 继承dictionary HashMap继承AbastractMap

4 容量的初始值 和 增加方式都不一样

(https://www.cnblogs.com/skywang12345/p/3311126.html 比较全)

与HashMap不同的是Hashtable是继承Dictionary，实现了Map接口。Map是"key-value键值对"接口，Dictionary是声明了操作"键值对"函数接口的抽象类。

HashMap默认的容量大小是16；增加容量时，每次将容量变为“原始容量x2”。

Hashtable默认的容量大小是11；增加容量时，每次将容量变为“原始容量x2 + 1”。

HashMap默认的“加载因子”是0.75, 默认的容量大小是16。

// 默认的初始容量是16，必须是2的幂。

static final int DEFAULT\_INITIAL\_CAPACITY = 16;

// 默认加载因子

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

// 指定“容量大小”的构造函数

public HashMap(int initialCapacity) {

this(initialCapacity, DEFAULT\_LOAD\_FACTOR);

}

当HashMap的 “实际容量” >= “阈值”时，(阈值 = 总的容量 \* 加载因子)，就将HashMap的容量翻倍。

Hashtable默认的“加载因子”是0.75, 默认的容量大小是11。

当Hashtable的 “实际容量” >= “阈值”时，(阈值 = 总的容量 x 加载因子)，就将变为“原始容量x2 + 1”。

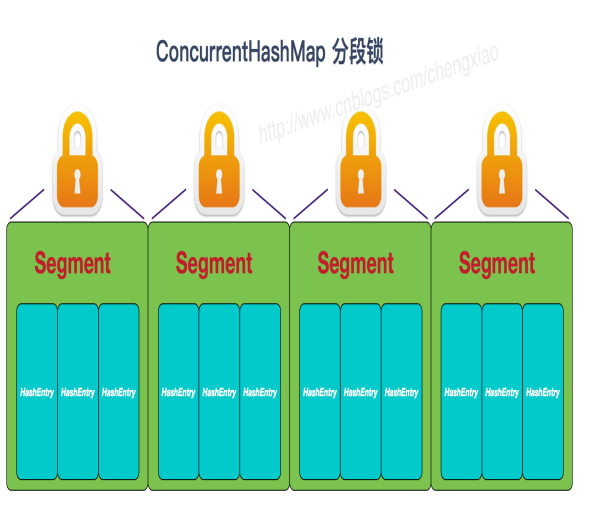
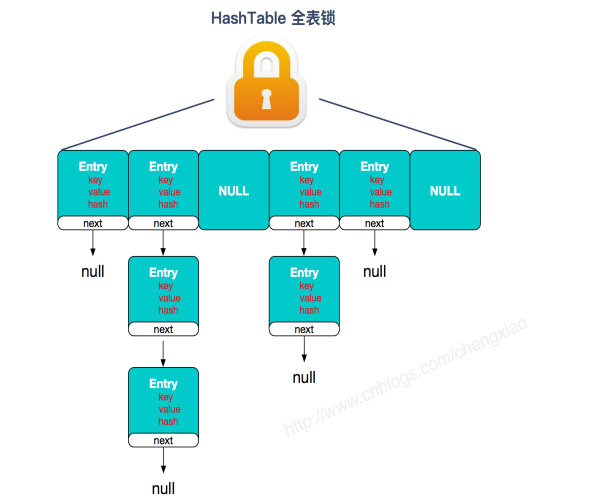
总结

| Map实现 | 使用场景 | 数据结构 |
| --- | --- | --- |
| HashMap | 哈希表存储键值对，key不重复，无序 | 哈希散列表 |
| LinkedHashMap | 是一个可以记录插入顺序和访问顺序的HashMap | 存储方式是哈希散列表，但是维护了头尾指针用来记录顺序 |
| TreeMap | 具有元素排序功能 | 红黑树 |
| WeakHashMap | 弱键映射，映射之外无引用的键，可以被垃圾回收 | 哈希散列表 |

# 3 ConcurrentHashMap

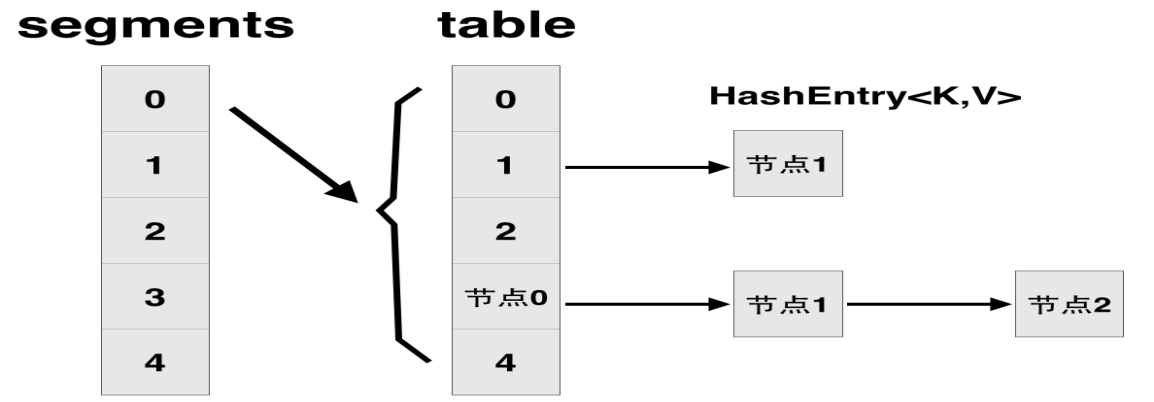
和HashMap一样继承AbstractMap实现Map接口，线程安全的。

HashTable性能差主要是由于所有操作需要竞争同一把锁，而如果容器中有多把锁，每一把锁锁一段数据，这样在多线程访问时不同段的数据时，就不会存在锁竞争了，这样便可以有效地提高并发效率。这就是ConcurrentHashMap所采用的"**分段锁**"思想。



## 3.1 1.7版本和1.8版本有所区别

1.7：



每一个segment都是一个HashEntry<K,V>[] table， table中的每一个元素本质上都是一个HashEntry的单向队列。比如table[3]为首节点，table[3]->next为节点1，之后为节点2，依次类推。

1.8做了优化：

改进一：取消segments字段，直接采用transient volatile Node<K,V>[] table(普通map声明：transient Node<K,V>[] table)保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。

改进二：将原先table数组＋单向链表的数据结构，变更为table数组＋单向链表＋红黑树的结构。对于hash表来说，最核心的能力在于将key hash之后能均匀的分布在数组中。如果hash之后散列的很均匀，那么table数组中的每个队列长度主要为0或者1。但实际情况并非总是如此理想，虽然ConcurrentHashMap类默认的加载因子为0.75，但是在数据量过大或者运气不佳的情况下，还是会存在一些队列长度过长的情况，如果还是采用单向列表方式，那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能。

## 3.2 常见问题

conCurrentHashMap 读是否加锁？

get方法无需加锁，由于其中涉及到的共享变量都使用volatile修饰，volatile可以保证内存可见性，所以不会读取到过期数据。

ConcurrentHashMap的迭代器是强一致性的迭代器还是弱一致性的迭代器？

是弱一致性的迭代器、

A B 同时访问一个数据。当A遍历数组时，刚遍历第三个数据，但B已经把数组的第四个数据删除，当A遍历到第四个数据时，迭代器不会抛出ConcurrentModificationException异常。

如果未遍历的数组上的内容发生了变化，则有可能反映到迭代过程中。这就是ConcurrentHashMap迭代器弱一致的表现。ConcurrentHashMap的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与Hashtable和同步的HashMap一样了。