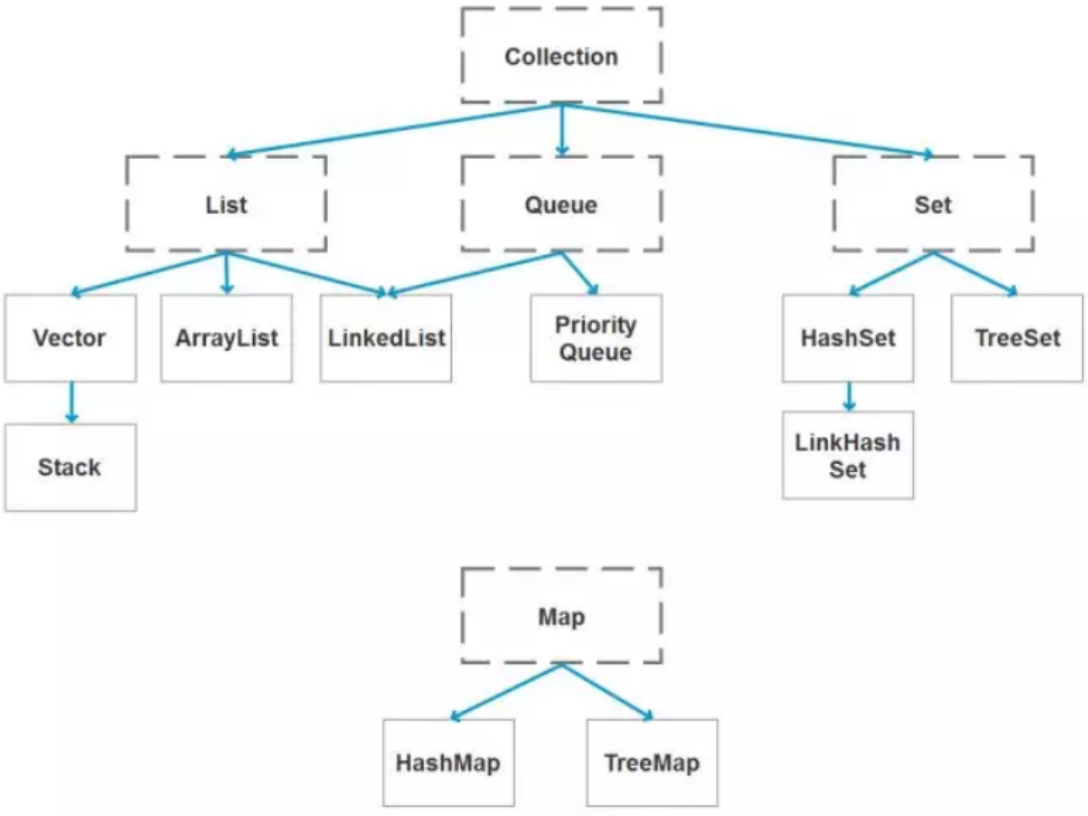
Java集合类总结

笔记本: 2-Java集合类
创建时间: 2019/9/7 21:01 **更新时间:** 2019/9/7 21:01
作者: pc941206@163.com

更多资料请关注微信公众号：**码农求职小助手**

一、集合类概览



常见容器主要包括 Collection 和 Map 两种，Collection 存储着对象的集合，而 Map 存储着键值对（两个对象）的映射表。

1、Collection

1.1 Set

1、 SetTreeSet： 基于红黑树实现，支持有序性操作，例如：根据一个范围查找元素的操作。但是查找效率不如 HashSet，HashSet 查找的时间复杂度为 O(1)，TreeSet 则为 O(logN)。

2、HashSet： 基于哈希表实现，支持快速查找，但不支持有序性操作。并且失去了元素的插入顺序信息，也就是说使用 Iterator 遍历 HashSet 得到的结果是不确定的。

3、LinkedHashSet： 具有 HashSet 的查找效率，且内部使用双向链表维护元素的插入顺序。

2、List

1、ArrayList： 基于动态数组实现，支持随机访问。

2、Vector： 和 ArrayList 类似，但它是线程安全的。

3、LinkedList： 基于双向链表实现，只能顺序访问，但是可以快速地在链表中间插入和删除元素。不仅如此，LinkedList 还可以用作栈、队列和双向队列。

3、Queue

LinkedList：可以用它来实现双向队列。

PriorityQueue：基于堆结构实现，可以用它来实现优先队列。

2、Map

1、TreeMap： 基于红黑树实现。

2、HashMap： 基于哈希表实现。

3、HashTable： 和 HashMap 类似，但它是线程安全的，这意味着同一时刻多个线程可以同时写入 HashTable 并且不会导致数据不一致。它是遗留类，不应该去使用它。现在可以使用 ConcurrentHashMap 来支持线程安全，并且 ConcurrentHashMap 的效率会更高，因为 ConcurrentHashMap 引入了分段锁。

4、LinkedHashMap： 使用双向链表来维护元素的顺序，顺序为插入顺序或者最近最少使用（LRU）顺序。

二、常见面试题

1、数组和链表的区别

数组： 是将元素在内存中连续存储的。它的优点是：内存地址连续，查找数据快；缺点：在存储之前，需要申请一块连续的内存空间，并且在编译时就必须确定好它的空间大小。在运行的时候，空间的大小是无法跟随你的需要进行增加和减少的，当数据较大的时候，有可能出现越界的情况；数据比较小的时候，又有可能会浪费内存空间。在改变数据个数时，增加、插入和删除数据的效率较低。

链表： 是动态申请内存空间的，不需要像数组那样提前申请好内存的大小，链表根据需要来动态的申请或者删除内存空间，对于数据的增加和删除以及插入操作比数组灵活。链表中的数据可以在内存中的任意位置，通过元素的指针来关联。

应用场景：

数组的应用场景：数据比较少、经常做的运算是按序号访问的数据元素、数组更容易实现、构建的线性表较稳定。

链表的应用场景：对线性表的长度或者规模难以估计；频繁做插入删除操作；构建动态性较强的线性表。

2、ArrayList 和 LinkedList 的区别?

ArrayList： 底层是基于数组实现的，查找快，增删较慢；

LinkedList： 底层是基于链表实现的。确切的说是循环双向链表（JDK1.6 之前是双向循环链表、1.7 之后取消了循环），查找慢、增删快。LinkedList 链表由一系列表项连接而成，一个表项包含 3 个部分：元素内容、前驱表和后驱表。链表内部有一个 header 表项，既是链表的开始也是链表的结尾。header 的后继表项是链表中的第一个元素，header 的前驱表项是链表中的最后一个元素。

- **ArrayList 的增删未必就是比 LinkedList 要慢**

1、**如果增删都是在末尾来操作**【每次调用的都是 remove() 和 add()】，此时 ArrayList 就不需要移动和复制数组来进行操作了。如果数据量有百万级的时，**速度是会比 LinkedList 要快的。**(我测试过)

2、**如果删除操作的位置是在中间。**由于 LinkedList 的消耗主要是在遍历上，ArrayList 的消耗主要是在移动和复制上（底层调用的是 arrayCopy() 方法，是 native 方法）。**LinkedList 的遍历速度是要慢于 ArrayList 的复制移动速度的如果数据量有百万级的时，还是 ArrayList 要快。**(我测试过)

> 补充：https://blog.csdn.net/weixin_39148512/article/details/79234817
> **ArrayList 集合实现 RandomAccess 接口有何作用？为何 LinkedList 集合却没实**

> **现这接口?**
> 1、RandomAccess 接口只是一个标志接口，只要 List 集合实现这个接口，就能支持快速随机访问。通过查看 Collections 类中的 binarySearch() 方法，可以看出，判断 List 是否实现 RandomAccess 接口来实行indexedBinarySerach(list,key) 或 iteratorBinarySerach(list,key)方法。再通过查看这两个方法的源码发现：**实现 RandomAccess 接口的 List 集合采用一般的 for 循环遍历，而未实现这接口则采用迭代器，即 ArrayList 一般采用 for 循环遍历，而 LinkedList 一般采用迭代器遍历。**
> 2、**ArrayList 用 for 循环遍历比 iterator 迭代器遍历快，LinkedList 用 iterator 迭代器遍历比 for 循环遍历快。**所以说，当我们在做项目时，应该考虑到 List 集合的不同子类采用不同的遍历方式，能够提高性能!

3、ArrayList 的扩容机制?

https://juejin.im/post/5d42ab5e5188255d691bc8d6

1、当使用 add 方法的时候首先调用 ensureCapacityInternal 方法，传入 size+1 进去，检查是否需要扩充 elementData 数组的大小；

2、newCapacity = 扩充数组为原来的 1.5 倍(不能自定义)，如果还不够，就使用它指定要扩充的大小 minCapacity ，然后判断 minCapacity 是否大于 MAX_ARRAY_SIZE(Integer.MAX_VALUE - 8) ，如果大于，就取 Integer.MAX_VALUE

3、扩容的主要方法：grow

3、ArrayList 中 copy 数组的核心就是 System.arraycopy 方法，将 original 数组的所有数据复制到 copy 数组中，这是一个本地方法。

4、Array 和 ArrayList 有何区别？什么时候更适合用Array?

1．**Array 可以容纳基本类型和对象，而 ArrayList 只能容纳对象**；

2．Array 是指定大小的，而 ArrayList 大小是固定的。

什么时候使用 Array:

1、如果列表的大小已经指定，大部分情况下是存储和遍历它们；

2、对于遍历基本数据类型，尽管 Collections 使用自动装箱来减轻编码任务，在指定大小的基本类型的列表上工作也会变得很慢；

3、如果你要使用多维数组，使用 [ ][ ] 比 List> 更容易。

5、HashMap相关【重点】

1、数据结构

Jdk1.7：Entry数组 + 链表

Jdk1.8：Node 数组 + 链表/红黑树，当链表上的元素个数超过 8 个并且数组长度 >= 64 时自动转化成红黑树，节点变成树节点，以提高搜索效率和插入效率到 O(logN)。entry 和 Node 都包含 key、value、hash、next 属性。

2、get 和 put 方法流程

（1）put方法的流程：

当我们想往一个 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次比较其 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。**

（2）get方法的流程

通过 key 的 hash 值找到在 table 数组中的索引处的 Entry，然后返回该 key 对应的 value 即可。

在这里能够根据 key 快速的取到 value 除了和 HashMap 的数据结构密不可分外，还和 Entry 有莫大的关系。**HashMap 在存储过程中并没有将 key，value 分开来存储，而是当做一个整体 key-value 来处理的，这个整体就是Entry 对象。**同时 value 也只相当于 key 的附属而已。在存储的过程中，系统根据 key 的 hashcode 来决定 Entry 在 table 数组中的存储位置，在取的过程中同样根据 key 的 hashcode 取出相对应的 Entry 对象（value 就包含在里面）。

3、resize 方法【重点】

有两种情况会调用 resize 方法：

1、第一次调用 HashMap 的 put 方法时，会调用 resize 方法对 table 数组进行初始化，如果不传入指定值，默认大小为 16。

2、扩容时会调用 resize，即 size > threshold 时，table 数组大小翻倍。

- **resize方法的逻辑：**

每次扩容之后容量都是**翻倍。**扩容后要将原数组中的所有元素找到在新数组中合适的位置。

当我们把 table[i] 位置的所有 Node 迁移到 newtab 中去的时候：这里面的 **node 要么在 newtab 的 i 位置（不变），要么在 newtab 的 i + n 位置。**也就是我们可以这样处理：**把 table[i] 这个桶中的 node 拆分为两个链表 l1 和 l2：如果 hash & n == 0，那么当前这个 node 被连接到 l1 链表；否则连接到 l2 链表。**这样下来，当遍历完 table[i] 处

的所有 node 的时候，我们得到两个链表 l1 和 l2，这时我们令 newtab[i] = l1，newtab[i + n] = l2，这就完成了 table[i] 位置所有 node 的迁移（rehash），这也是 HashMap 中容量一定的是 2 的整数次幂带来的方便之处。

4、size 必须是 2 的整数次方原因?

这样做总是能够保证 HashMap 的底层数组长度为 2 的 n 次方。当 length 为 2 的 n 次方时，h & (length - 1) 就相当于对 length 取模，而且速度比直接取模快得多，这是 HashMap 在速度上的一个优化。而且每次扩容时都是翻倍。

4、HashMap 多线程死循环问题

https://blog.csdn.net/xuefeng0707/article/details/40797085

https://blog.csdn.net/dgutliangxuan/article/details/78779448

主要是多线程同时 put 时，如果同时触发了 **rehash 操作，会导致 HashMap 中的链表中出现循环节点，进而使得后面 get 的时候，会死循环。**

5、影响 HashMap 的性能因素

key 的 hashCode 函数实现、loadFactor、初始容量

6、HashMap 中 key 的 hash 值计算方法以及原因

对于 HashMap 的 table 而言，数据分布需要均匀（最好每项都只有一个元素，这样就可以直接找到），不能太紧也不能太松，太紧会导致查询速度慢，太松则浪费空间。在 HashMap 内部计算是使用 indexFo r方法来计算 key 的 hash 值，该方法只有一条语句：**h & (length - 1)，该方法相当于 length 的取模运算，加快运算速度，并可以均匀分布 table数据和充分利用空间。**

8、table[i] 位置的链表什么时候会转变成红黑树?

在 JDK1.8 中，Node 数组中每一个桶中存储的是 Node 链表，当链表长度 >= 8 的时候并且 Node 数组的大小 >= 64，链表会变为红黑树结构【因为红黑树的增删改查复杂度是 O(logn)，链表是 O(n)，红黑树结构比链表代价更小】。

9、HashMap 主要成员属性

threshold、loadFactor、HashMap 的懒加载（第一次用到的时候才对 map进行初始化）。

10、HashMap 的 get 方法能否判断某个元素是否在 map 中?

HashMap 的 get 函数的返回值不能判断一个 key 是否包含在 map 中，因为 get 返回 null 有可能是不包含该 key，也有可能该 key 对应的 value 为 null。因为 HashMap 中允许 key 为 null，也允许 value 为 null。

11、HashMap 线程安全吗，哪些环节最有可能出问题，为什么?

我们都知道 HashMap 是线程不安全的，那么哪些环节最优可能出问题呢，及其原因：没有参照，这个问题有点不好直接回答，但是我们可以找参照啊。

参照：ConcurrentHashMap，因为大家都知道 HashMap 不是线程安全的，ConcurrentHashMap 是线程安全的。对照 ConcurrentHashMap，看看 ConcurrentHashMap 在 HashMap 的基础之上增加了哪些安全措施，这个问题就迎刃而解了。

这里先简要回答这个问题：HashMap 的 put 操作是不安全的，因为没有使用任何锁。HashMap 在多线程下最大的安全隐患发生在扩容的时候。想想一个场合：HashMap 使用默认容量 16，这时 100 个线程同时往 HashMap 中 put 元素，会发生什么？扩容混乱，因为扩容也没有任何锁来保证并发安全。

12、HashMap 的 value 允许为 null，但是 HashTable 和 ConcurrentHashMap 的 value 都不允许为 null，试分析原因?

首先要明确 ConcurrentHashMap 和 Hashtable 从技术从技术层面讲是可以允许 value 为 null 。但是它们实际上是不允许的，这肯定是为了解决一些问题，为了说明这个问题，我们看下面这个例子（这里以 ConcurrentHashMap 为例，HashTable 也是类似）。

HashMap 由于允 value 为 null，get 方法返回 null 时有可能是 map 中没有对应的 key，也有可能是该 key 对应的 value 为 null。所以 get 不能判断 map 中是否包含某个 key，只能使用 contains 判断是否包含某个 key。看下面的代码段，要求完成这个一个功能：**如果 map 中包含了某个 key ，则返回对应的 value，否则抛出异常：**

```
1 if (map.containsKey(k)) {
2    return map.get(k);
3 } else {
4    throw new KeyNotPresentException();
5 }
```

1、如果上面的 map 为 HashMap，那么没什么问题，因为 HashMap 本来就是线程不安全的，如果有并发问题应该用 ConcurrentHashMap，所以在单线程下面可以返回正确的结果。

2、如果上面的 map 为 ConcurrentHashMap，此时存在并发问题：在 map.containsKey(k) 和 map.get 之间有可能其他线程把这个 key 删除了，这时候 map.get 就会返回 null，而 ConcurrentHashMap 中不允许 value 为 null，也就是这时候返回了 null，一个根本不允许出现的值。但是因为 ConcurrentHashMap 不允许 value 为 null，所以可以通过 map.get(key) 是否为 null 来判断该 map 中是否包含该 key，这时就没有上面的并发问题了。

HashMap 在 JDK1.8 中的优化?

1、resize 时不需要重新计算 hash，只要看看原来的 hash 新增的那个 bit 是 0 还是 1。0 的话索引没变，1 的话索引变成原索引加 oldcap。找到新数组下标，确定索引位置，增加随机性；

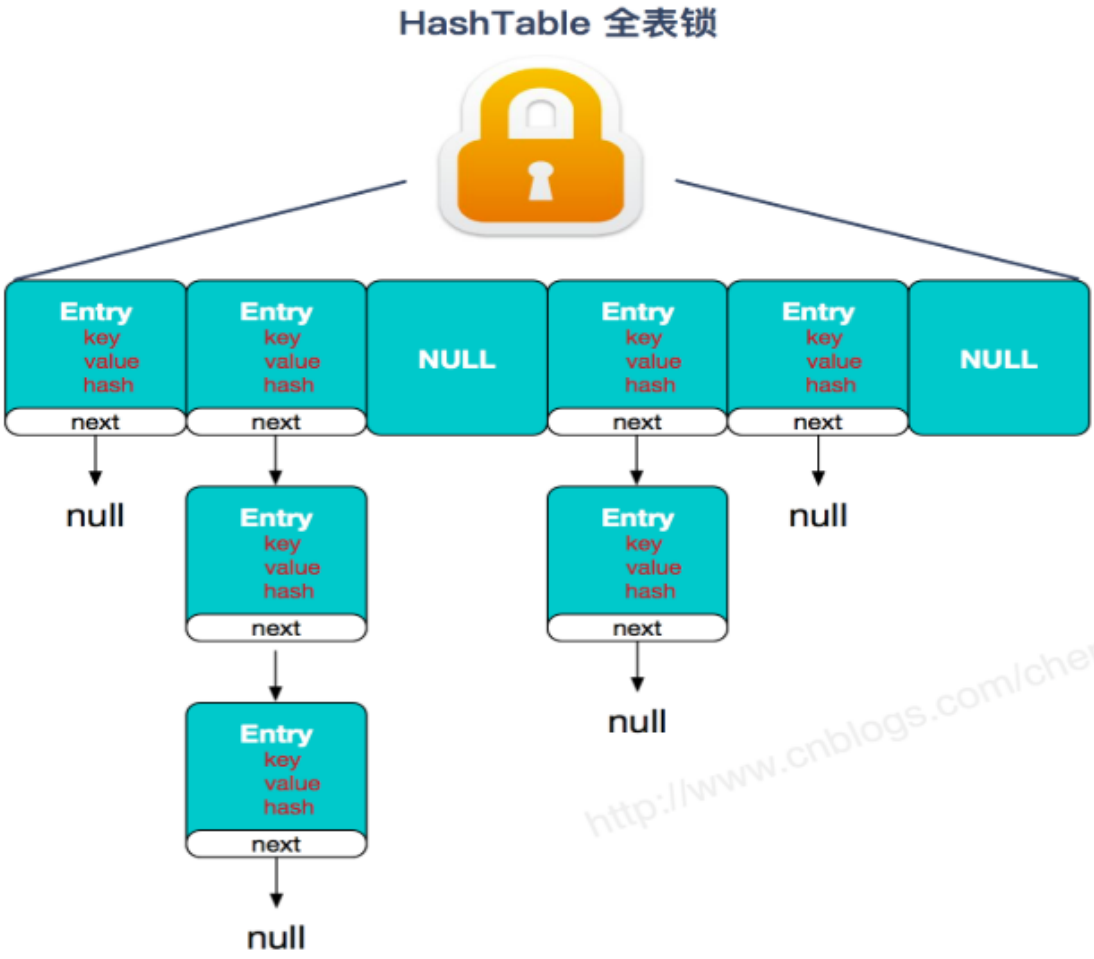
2、不会形成闭环，扩容时不再使用头插法改成尾插法。

HashTable

HashTable 和 HashMap 的实现原理几乎一样，差别无非是

1、HashTable 不允许 key 和 value 为 null；

2、HashTable 是线程安全的。但是 HashTable 线程安全的策略实现代价却太大了，简单粗暴，get/put 所有相关操作都是 synchronized 的，这相当于给整个哈希表加了一把大锁，多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作串行化，在竞争激烈的并发场景中性能就会非常差。



- **HashMap 与 HashTable 的区别是什么?**

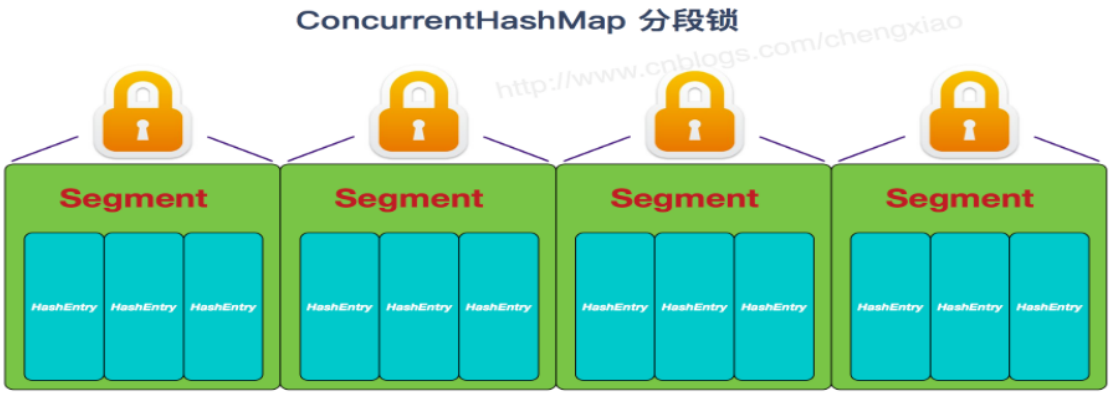
1．HashTable 基于 Dictionary 类，而 HashMap 是基于 AbstractMap。Dictionary 是任何可将键映射到相应值的类的抽象父类，而 AbstractMap 是基于 Map 接口的实现，它以最大限度地减少实现此接口所需的工作。

2．HashMap 的 key 和 value 都允许为 null，而 Hashtable 的 key 和 value 都不允许为 null。HashMap 遇到 key 为 null 的时候，调用 putForNullKey 方法进行处理，而对 value 没有处理；Hashtable 遇到 null，直接返回 NullPointerException。

3．Hashtable 是线程安全的，而 HashMap 不是线程安全的，但是我们也可以通过 Collections.synchronizedMap(hashMap)，使其实现同步。

ConcurrentHashMap

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。



- **底层数据结构实现**

JDK 7：中 ConcurrentHashMap 采用了 **数组 + Segment + 分段锁** 的方式实现。

JDK 8：中 ConcurrentHashMap 参考了 JDK 8 HashMap 的实现，采用了 **数组 + 链表 + 红黑树** 的实现方式来设计，内部大量采用 CAS 操作。

- **源码分析**

ConcurrentHashMap 采用了非常精妙的"分段锁"策略，ConcurrentHashMap 的主干是个 Segment 数组。

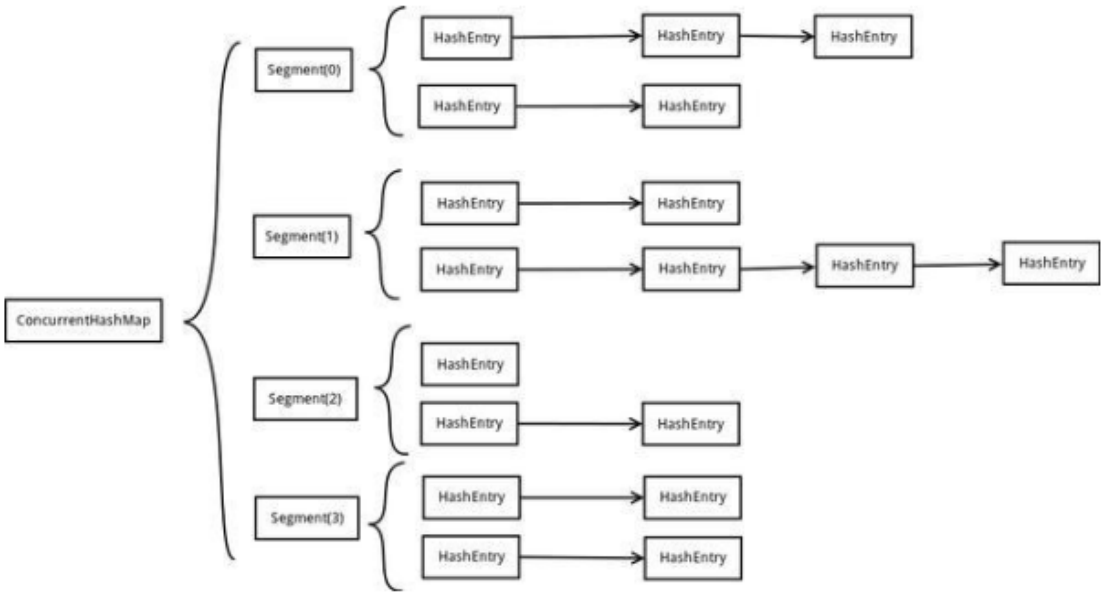
```
final  Segment<K,V>[]  segments;
```

Segment 继承了 ReentrantLock，所以它就是一种可重入锁（ReentrantLock)。在 ConcurrentHashMap，一个 Segment 就是一个子哈希表，Segment 里维护了一个 HashEntry 数组，并发环境下，对于不同 Segment 的数据进行操作是不用考虑锁竞争的。就按默认的 ConcurrentLevel 为 16 来讲，理论上就允许 16 个线程并发执行。

所以，对于同一个 Segment 的操作才需考虑线程同步，不同的 Segment 则无需考虑。Segment 类似于 HashMap，一个 Segment 维护着一个HashEntry 数组：

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

HashEntry 是目前我们提到的最小的逻辑处理单元了。一个 ConcurrentHashMap 维护一个 Segment 数组，一个 Segment 维护一个 HashEntry 数组。



从上面的结构可以发现 ConcurrentHashMap 定位一个元素的过程需要进行两次 Hash 操作。第一次 Hash 定位到 Segment，第二次 Hash 定位到元素所在的链表的头部。

- **put 方法**

1、定位 segment 并确保定位的 Segment 已初始化；

2、调用 Segment 的 put 方法。

- **get 方法**

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。

- **关于 segmentShift 和 segmentMask**

segmentShift 和 segmentMask 这两个全局变量的主要作用是用来定位 Segment：

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

segmentMask：段掩码，假如 segments 数组长度为16，则段掩码为 16-1=15；segments 长度为 32，段掩码为 32-1=31。这样得到的所有 bit 位都为 1，可以更好地保证散列的均匀性；

segmentShift：2 的 sshift 次方等于 ssize，segmentShift=32-sshift。若 segments 长度为16，segmentShift=32-4=28; 若 segments 长度为 32，segmentShift=32-5=27。而计算得出的 hash 值最大为 32 位，无符号右移 segmentShift，则意味着只保留高几位（其余位是没用的），然后与段掩码 segmentMask 位运算来定位 Segment。

7、HashSet 的底层实现是什么?

通过看源码知道 HashSet 的实现是依赖于 HashMap 的，HashSet 的值都是存储在 HashMap 中的。在 HashSet 的构造法中会初始化一个 HashMap 对象，HashSet 不允许值重复。因此，HashSet 的值是作为 HashMap 的 key 存储在 HashMap 中的，当存储的值已经存在时返回 false。

- **HashSet 怎样保证元素不重复?**

```
public boolean add(E e) {

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

元素值作为的是 map 的 key，map 的 value 则是 PRESENT 变量，这个变量只作为放入 map 时的一个占位符而存在，所以没什么实际用处。其实，这时候答案已经出来了：HashMap 的 key 是不能重复的，而这里HashSet 的元素又是作为了 map 的 key，当然也不能重复了。

若要将对象存放到 HashSet 中并保证对象不重复，应根据实际情况将对象的 hashCode 方法和 equals 方法进行重写。

8、LinkedHashMap 的实现原理?

LinkedHashMap 也是基于 HashMap 实现的，不同的是它定义了一个 Entry header，这个 header 不是放在 Table 里，它是额外独立出来的。**LinkedHashMap 通过继承 hashMap 中的 Entry，并添加两个属性 Entry before，after 和 header 结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。LinkedHashMap 定义了排序模式 accessOrder，该属性为 boolean 型变量，对于访问顺序，为 true；对于插入顺序，则为 false 。** 一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。

9、迭代器

- **Iterator 是什么?**

迭代器是一种设计模式，它是一个对象，它可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。迭代器通常被称为“轻量级”对象，因为创建它的代价小。

- **Iterator 怎么使用？有什么特点?**

Java 中的 Iterator 功能比较简单，并且只能单向移动：

(1) 使用方法 iterator() 要求容器返回一个 Iterator。第一次调用 Iterator 的 next() 方法时，它返回序列的第一个元素。注意：iterator() 方法是 java.lang.Iterable 接口，被 Collection继承。

(2) 使用 next() 获得序列中的下一个元素。

(3) 使用 hasNext() 检查序列中是否还有元素。

(4) 使用 remove() 将迭代器新返回的元素删除。

Iterator 是 Java 迭代器最简单的实现，为 List 设计的 ListIterator 具有更多的功能，它可以从两个方向遍历 List，也可以从 List 中插入和删除元素。

- **Iterator 和 ListIterator 有什么区别?**

Iterator 可用来遍历 Set 和 List 集合，但是 ListIterator 只能用来遍历 List。 Iterator 对集合只能是前向遍历，ListIterator 既可以前向也可以后向。 ListIterator 实现了 Iterator 接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引等等。

- **Iterator 和 Enumeration 接口的区别?**

以前我们只是大概知道的是：Iterator 替代了 Enumeration，Enumeration 是一个旧的迭代器了。

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。我们在做练习的时候，迭代时会不会经常出错，抛出 ConcurrentModificationException 异常，说我们在遍历的时候还在修改元素。这其实就是 fail-fast 机制。具体区别有三点：

1、Iterator 的方法名比 Enumeration 更科学；

2、Iterator 有 fail-fast 机制，比 Enumeration 更安全；

3、Iterator 能够删除元素，Enumeration 并不能删除元素。

10、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。**当检测到正在遍历的集合的结构被改变时，fail-fast 迭代器抛出ConcurrentModificationException，而 fail-safe 迭代器从不抛出 ConcurrentModificationException。**