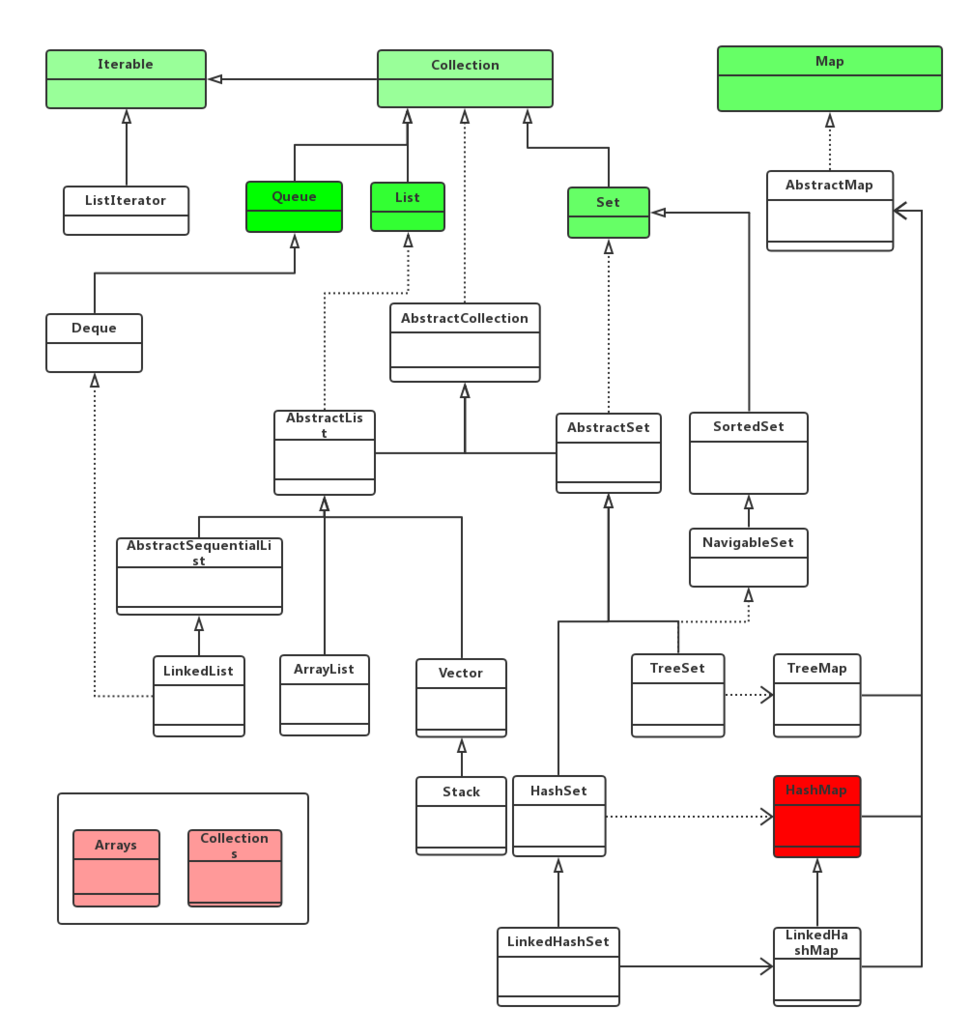
**Java中的集合主要分为两大类：**

（1）Collecton：可以理解为主要存放的是单个对象，Collection继承了Iterate接口，Iterate用于集合内迭代器抽象接口，其子类均实现接口中方法。

（2）Map：可以理解为主要存储key-value类型的对象



**一，Map**

**1.HashMap 简介**

**1.1. 关于HashMap的一些说法：**

a) HashMap的底层结构是一个数组，数组中的每一项是一条链表。

b) HashMap的实例有俩个参数影响其性能： “初始容量(默认16)” 和 装填因子()。

c) HashMap实现不同步，线程不安全。 HashTable线程安全

d) HashMap中的key-value都是存储在Entry中的。

e) HashMap可以存null键和null值，通过hashCode()方法和equals方法保证键的唯一性。

f) 解决冲突主要有三种方法：开放定址法（再散列法），拉链法，再哈希法，建立公共溢出区。 HashMap是采用拉链法解决哈希冲突的。

g）多线程的并发问题：多线程put时可能导致元素丢失；resize时可能出现环链导致死循环；fail-fast策略：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。

h）为什么String这样的wrapper类适合作为键？

因为String是不可变的，也是final的,不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，不可变性还可以保证线程安全。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。

i) JDK1.8 以后在解决哈希冲突时有了较大的变化，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。

所谓 “拉链法” 就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。

注： 链表法是将相同hash值的对象组成一个链表放在hash值对应的槽位

**hash 碰撞解决的方法：**

① 用开放定址法解决冲突的做法是：就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。

插入时可能会出现多次冲突的现象，容易产生堆积问题，不适于大规模的数据存储。

删除结点不能简单地将被删结点置为空，否则将截断在它之后填人散列表的同义词结点的查找路径。这是因为各种开放地址法中，空地址单元是查找失败的条件。因此在用开放地址法处理冲突的散列表上执行删除操作，只能在被删结点上做删除标记，而不能真正删除结点。

（1）**线性探测再散列**

di=1，2，3，…，m-1。

这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

**（2）平方探测再散列**

IMG_257

这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

② 拉链法解决冲突的做法是： 将所有关键字为同义词（hash值相同）的结点链接在同一个单链表中 。若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数组T[0..m-1]。凡是散列地址为i的结点，均插入到以T[i]为头指针的单链表中。T中各分量的初值均应为空指针。在拉链法中，装填因子α可以大于1，但一般均取α≤1。拉链法适合未规定元素的大小。

处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，**因此平均查找长度较短；**

**删除结点的操作易于实现**。只要简单地删去链表上相应的结点即可。

在 jdk1.8 版本后，java对HashMap做了改进，在链表长度大于8（且桶的数量超过64）的时候，将后面的数据存在红黑树中，以加快检索速度。

③ 再哈希法：同时构造多个不同的哈希函数，当一个冲突时使用下一个，直到不冲突为止。

**④ 建立公共溢出区：**将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

**2.类的属性：**

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable { // 序列号 private static final long serialVersionUID = 362498820763181265L; // 默认的初始容量是16 static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // 最大容量 static final int MAXIMUM\_CAPACITY = 1 << 30; // 默认的填充因子 static final float DEFAULT\_LOAD\_FACTOR = 0.75f; // 当桶(bucket)上的结点数大于这个值时会转成红黑树 static final int TREEIFY\_THRESHOLD = 8; // 当桶(bucket)上的结点数小于这个值时树转链表 static final int UNTREEIFY\_THRESHOLD = 6; // 桶中结构转化为红黑树对应的table的最小大小 static final int MIN\_TREEIFY\_CAPACITY = 64; // 存储元素的数组，总是2的幂次倍 transient Node<k,v>[] table; // 存放具体元素的集 transient Set<map.entry<k,v>> entrySet; // 存放元素的个数，注意这个不等于数组的长度。 transient int size; // 每次扩容和更改map结构的计数器 transient int modCount; // 临界值 当实际大小(容量\*填充因子)超过临界值时，会进行扩容 int threshold; // 加载因子 final float loadFactor;(默认0.75) }

* loadFactor加载因子

loadFactor加载因子是控制数组存放数据的疏密程度，loadFactor越趋近于1，那么 数组中存放的数据(entry)也就越多，也就越密，也就是会让链表的长度增加，loadFactor越小，也就是趋近于0，数组中存放的数据(entry)也就越少，也就越稀疏。

loadFactor太大导致查找元素效率低，太小导致数组的利用率低，存放的数据会很分散。loadFactor的默认值为0.75f是官方给出的一个比较好的临界值。

给定的默认容量为 16，负载因子为 0.75。Map 在使用过程中不断的往里面存放数据，当数量达到了 16 \* 0.75 = 12 就需要将当前 16 的容量进行扩容，而扩容这个过程涉及到 rehash、复制数据等操作，所以非常消耗性能。

* threshold 临界值

threshold = capacity \* loadFactor，当Size>=threshold的时候，那么就要考虑对数组的扩增了，也就是说，这个的意思就是 衡量数组是否需要扩增的一个标准。

**3.Node节点类源码:**

// 继承自 Map.Entry<K,V> static class Node<K,V> implements Map.Entry<K,V> { final int hash;// 哈希值，存放元素到hashmap中时用来与其他元素hash值比较 final K key;//键 V value;//值 // 指向下一个节点 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; } // 重写hashCode()方法 public final int hashCode() { return Objects.hashCode(key) ^ Objects.hashCode(value); } public final V setValue(V newValue) { V oldValue = value; value = newValue; return oldValue; } // 重写 equals() 方法 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; } }

树节点类源码:

static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> { TreeNode<K,V> parent; // 父 TreeNode<K,V> left; // 左 TreeNode<K,V> right; // 右 TreeNode<K,V> prev; // needed to unlink next upon deletion删除时需要取消下一个链接 boolean red; // 判断颜色 TreeNode(int hash, K key, V val, Node<K,V> next) { super(hash, key, val, next); } // 返回根节点 final TreeNode<K,V> root() { for (TreeNode<K,V> r = this, p;;) { if ((p = r.parent) == null) return r; r = p; }

**4.HashMap 和 Hashtable 的区别**

1. 继承不同

public class Hashtable extends Dictionary implements Map

public class HashMap extends AbstractMap implements Map

1. 线程是否安全： HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过synchronized 修饰。
2. 效率： 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；
3. 对Null key 和Null value的支持： HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。但是在 HashTable 中 null 不可以作为键
4. 初始容量大小和每次扩充容量大小的不同 ： ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小

**为什么 HashMap 容量一定要为2的幂呢？**

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。Hash 值的范围值大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ (n - 1) & hash”。（n代表数组长度）。这也就解释了 HashMap 的长度为什么是2的幂次方。

**这个算法应该如何设计呢？**

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是2的 n 次方；）。” 并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。

1. 底层数据结构： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，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。Hashtable 没有这样的机制。
2. hashmap中没有contains方法，hashtable中有此方法。

**HashMap 多线程操作导致死循环问题**

主要原因在于 并发下的Rehash 会造成元素之间会形成一个循环链表。不过，jdk 1.8 后解决了这个问题，但是还是不建议在多线程下使用 HashMap,因为多线程下使用 HashMap 还是会存在其他问题比如数据丢失。并发环境下推荐使用 ConcurrentHashMap 。

**5.HashMap 和 HashSet区别**

HashSet 底层就是基于 HashMap 实现的

|  |  |
| --- | --- |
| HashMap | HashSet |
| 实现了Map接口 | 实现Set接口 |
| 存储键值对 | 仅存储对象 |
| 调用 put()向map中添加元素 | 调用 add()方法向Set中添加元素 |
| HashMap使用键（Key）计算Hashcode | HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以equals()方法用来判断对象的相等性， |

**6.hashCode（）与equals（）的相关规定：**

* 如果两个对象相等，则hashcode一定也是相同的
* 两个对象有相同的hashcode值，它们也不一定是相等的
* 综上，equals方法被覆盖过，则hashCode方法也必须被覆盖
* hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

**==与equals的区别**

* ==是判断两个变量或实例是不是指向同一个内存空间 equals是判断两个变量或实例所指向的内存空间的值是不是相同
* ==是指对内存地址进行比较 equals()是对字符串的内容进行比较
* ==指引用是否相同 equals()指的是值是否相同

**7.LinkedHashMap（HashMap + 双向链表）**

LinkedHashMap维护着一个运行于所有条目的双向链表，此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序（insert-order）或者是访问顺序，其中默认的迭代访问顺序就是插入顺序，即可以按插入的顺序遍历元素，这点和HashMap有很大的不同。

所谓访问顺序(access-order)是指在迭代遍历列表中的元素时最近访问的元素会排在LinkedHashMap的尾部。

而HashMap则是按照key的顺序遍历元素的。

**8. TreeMap 和TreeSet**

**8.1 TreeMap的实现**

TreeMap是基于红黑树实现的，该集合根据其键的自然顺序进行排序。红黑树顾名思义就是节点是红色或者黑色的平衡二叉排序树，它通过颜色的约束来维持着二叉树的平衡。红黑树满足如下规则：

**1、每个节点都只能是红色或者黑色**

**2、根节点是黑色**

**3、每个叶节点（NIL节点，空节点）是黑色的。**

**4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。**

**5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。**

**红黑树的优势：**

红黑树相比普通的 avl （平衡树）树，**在检索的时候效率其实差不多**，都是通过平衡来二分查找。**但对于插入、删除等操作效率提高很多**，红黑树不像 avl 树一样追求绝对的平衡，他允许局部很少的不完全平衡，这样对于效率影响不大，但省去了很多没有必要的调平衡操作，avl 树调平衡有时候代价较大，所以效率不如红黑树。

红黑树能够以O(log2(N))的时间复杂度进行搜索、插入、删除操作。此外，任何不平衡都会在3次旋转之内解决，这一点是AVL所不具备的。

**红黑树的插入：**

将红黑树当作一颗二叉排序树，将节点插入，并将节点着为红色；然后通过旋转和重新着色等方法来修正该树，使之重新成为一颗红黑树。

**红黑树的删除：**

将红黑树当作一颗二叉排序树，将该节点从二叉排序树中删除；然后，通过"旋转和重新着色"等一系列操作来修正该树，使之重新成为一棵红黑树。

删除节点分三种情况：

（1）节点是叶节点，那么直接将该节点删除就OK了。

（2）节点只有一个孩子节点，直接删除该节点，并用该节点的唯一子节点顶替它的位置。

（3）节点有两个儿子，先找出它的后继节点，然后把“它的后继节点的内容”复制给“该节点的内容”，之后删除“它的后继节点”。

**8.2 TreeMap和TreeSet的区别和联系**

TreeSet里面绝大部分方法都是直接调用TreeMap方法来实现的。

**相同点：**

* TreeMap和TreeSet都是非同步集合，因此他们不能在多线程之间共享，不过可以使用方法Collections.synchroinzedMap()来实现同步。
* 运行速度都要比Hash集合慢，基于红黑树实现，所以他们内部对元素的操作时间复杂度为O(logN)，而HashMap/HashSet则为O(1)。
* TreeMap和TreeSet都是有序的集合，也就是说他们存储的值都是拍好序的。

**不同点：**

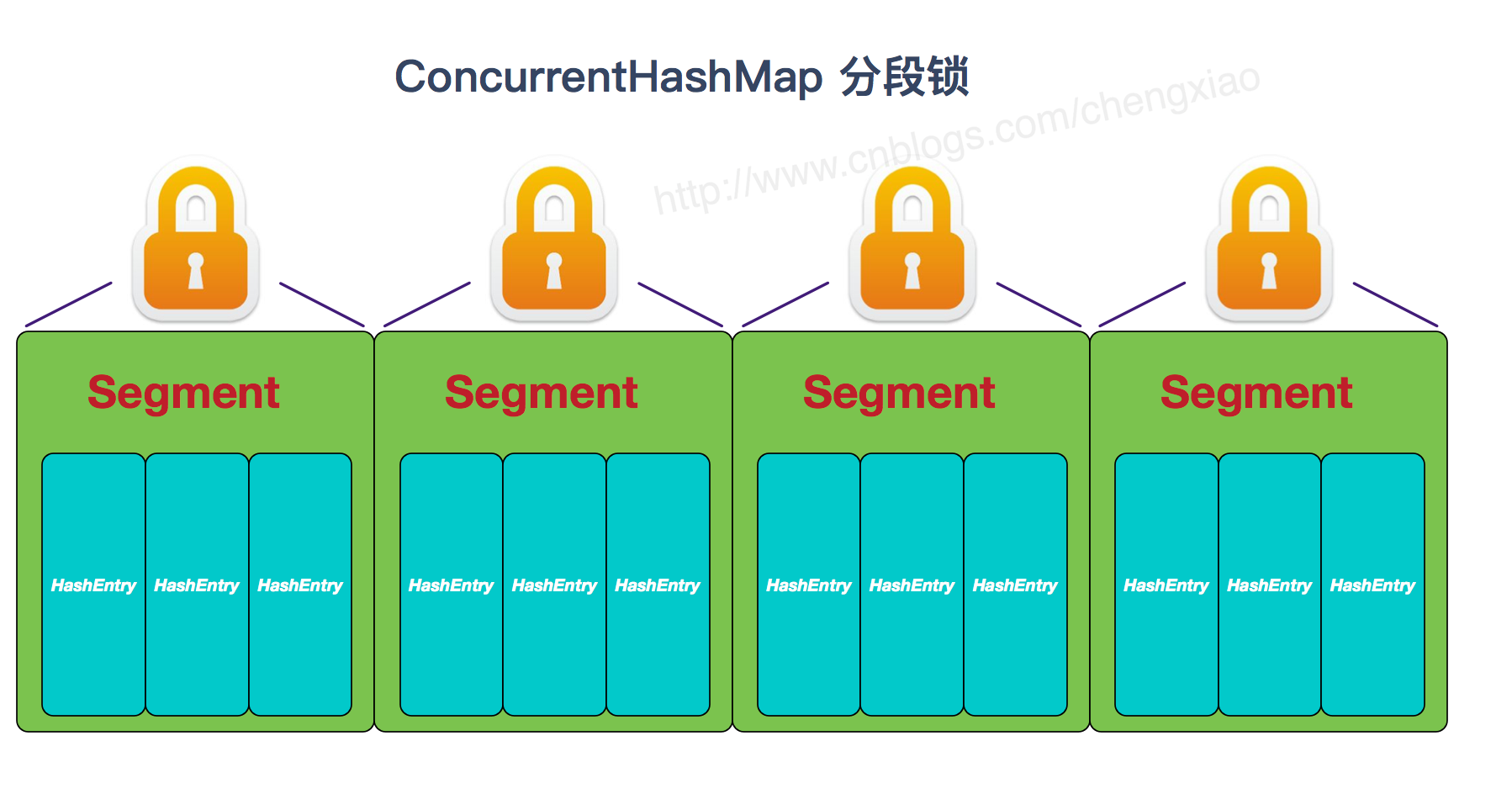
* 最主要的区别就是TreeSet和TreeMap分别实现Set和Map接口
* TreeSet只存储一个对象，而TreeMap存储两个对象Key和Value（仅仅key对象有序）。
* TreeSet中不能有重复对象，而TreeMap中可以存在重复对象。

**9.ConcurrentHashMap 和 Hashtable 的区别**

* **底层数据结构： JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；**
* **实现线程安全的方式（重要）： ① 在JDK1.7的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。② Hashtable(同一把锁) :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。**

ConcurrentHashMap线程安全的具体实现方式/底层具体实现

JDK1.7



首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。

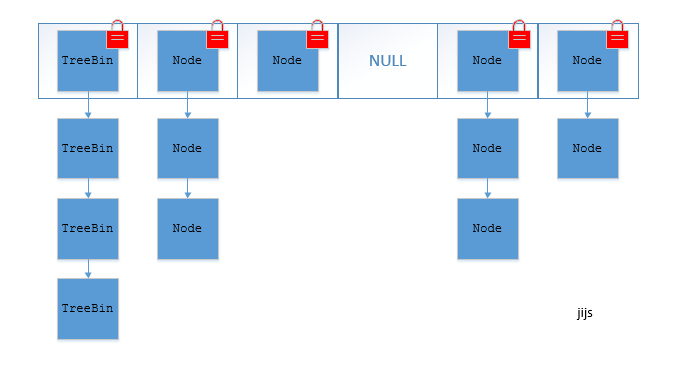
ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。

Segment 继承了 ReentrantLock,所以 Segment 是一种可重入锁，扮演锁的角色。HashEntry 用于存储键值对数据。

static class Segment<K,V> extends ReentrantLock implements Serializable { }

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和HashMap类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，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 数组的数据进行修改时，必须首先获得对应的 Segment的锁。

JDK1.8



ConcurrentHashMap取消了Segment分段锁，采用CAS(Compare-and-Swap)和synchronized来保证并发安全。数据结构跟HashMap1.8的结构类似，数组+链表/红黑二叉树。Java 8在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为O(N)）转换为红黑树（寻址时间复杂度为O(log(N))）

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。

**9、fail-fast机制**

fail-fast 是Java集合的一种错误检测机制，当多个线程对集合进行结构上的改变的时，有可能会产生fail-fast机制。

fail-fast产生的原因就在于程序在对 collection 进行迭代时，某个线程对该 collection 在结构上对其做了修改，这时迭代器就会抛出 ConcurrentModificationException 异常信息，从而产生 fail-fast。

迭代器在调用next()、remove()方法时都是调用checkForComodification()方法，该方法主要就是检测modCount （集合中定义）== expectedModCount （迭代器中定义）?是否成立，若不等则抛出ConcurrentModificationException 异常，从而产生fail-fast机制。