# HashMap详解

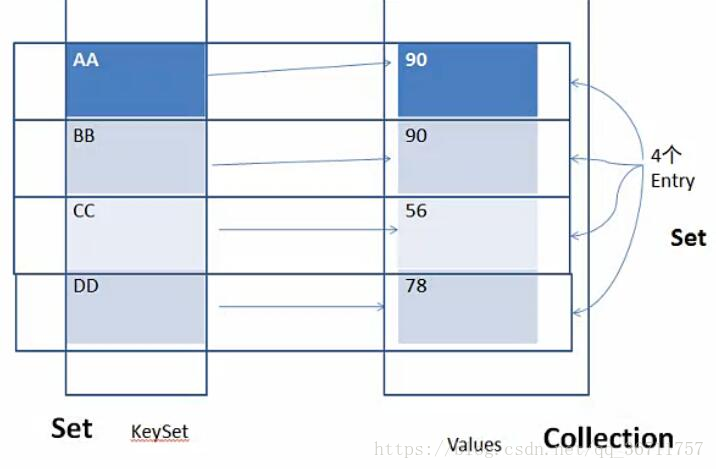
## Map讲解

### 1.1 什么是Map？

Map接口定义了一个保存key-value的对象，该对象中key值是不存在重复的，每个key值至多对应一个value。

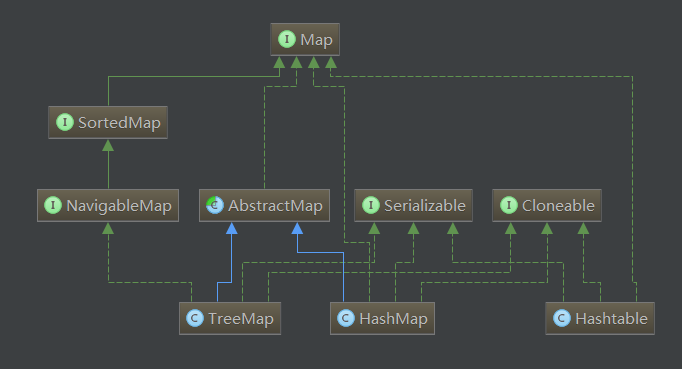
### Map的存储结构是什么？

Map的存储结构：



上边这个图就是map的存储结构，没错，看起来就像是数据库中的关系表，有两个字段（或者说属性），keyset（键的集合）和values（值的集合），每一条记录都是一个entry（一个键值对）。

### Map的类图结构是什么？



（图片和说明来源：<https://www.jianshu.com/p/38ba8d5200da>）

如上图所示是实现Map接口的类图结构，主要包含了如下的类与接口：

* **Map接口**： 定义将键值映射到值的对象，Map规定不能包含重复的键值，每个键最多可以映射一个值,这个接口是用来替换Dictionary类。
* **SortedMap接口**:定义按照key排序的Map结构，规定key-value是根据键key值的自然排序进行排序的，或者根据构造key-value时设定的构造器进行排序。
* **NavigableMap接口**： 是SortedMap接口的子接口，在其基础上扩展了针对搜索目标返回最近匹配项的导航方法，例如方法lowEntry、floorEntry、ceilingEntry等，如果不存在这样的键，则返回null
* **AbstractMap类**：提供了一个Map骨架的实现,尽量减少了实现Map接口所需要的工作量
* **HashMap类**：HashMap是实现了Map接口的key-value集合，实现了所有map的操作，允许key和value为null,它相当于Hashtable，与之存在的区别是hashMap不是线程安全的，HashMap允许null值。
* **TreeMap类**：TreeMap是基于红黑树的实现，也是记录了key-value的映射关系，该映射根据key的自然排序进行排序或者根据构造方法中传入的比较器进行排序，也就是说TreeMap是有序的key-value集合
* **Hashtable类**：它是类似与HashMap的key-value的哈希表，不允许key-value为NULL值，另外一点值得注意的是Hashtable是线程安全的
* **Serializable接口**：实现了该接口标识了类可以被序列化和反序列化，具体的 查询序列化详解
* **Cloneable接口**：实现了该接口的类可以显示的调用Object.clone()方法，合法的对该类实例进行字段复制，如果没有实现Cloneable接口的实例上调用Obejct.clone()方法，会抛出CloneNotSupportException异常。正常情况下，实现了Cloneable接口的类会以公共方法重写Object.clone()

### Map的特点是什么？

（1）没有重复的 key（一方面，key用set保存，所以key必须是唯一，无序的；另一方面，map的取值基本上是通过key来获取value，如果有两个相同的key，计算机将不知道到底获取哪个对应值；这时候有可能会问，那为什么我编程时候可以用put（）方法传入两个key值相同的键值对？那是因为源码中，传入key值相同的键值对，将作为覆盖处理）

（2）每个 key 只能对应一个 value, 多个 key 可以对应一个 value（这就是映射的概念，最经典的例子就是射箭，一排射手，一排箭靶，一个射手只能射中一个箭靶，而每个箭靶可能被不同射手射中。这里每个射手只有一根箭，不存在三箭齐发还都中靶这种骚操作。将射手和射中的靶子连线，这根线加射手加靶子就是一个映射）

（3）key,value 都可以是任何引用类型（包括 null）的数据（只能是引用类型）

（4）Map 取代了古老的 Dictionary 抽象类（知道就行，可以忽略）

### 哈希全家桶

把任意长度的输入（输入叫做预映射，知道就行），通过一种函数（hashCode() 方法），变换成固定长度的输出，该输出就是**哈希值（hashCode）**，这种函数就叫做**哈希函数**，而计算哈希值的过程就叫做**哈希**。哈希的主要应用是哈希表和分布式缓存。

这里有个问题，哈希算法和哈希函数不是一个东西，哈希函数是哈希算法的一种实现，以后面试就说哈希函数就行。

在将键值对存入数组之前，将key通过哈希算法计算出哈希值，把哈希值作为数组下标，把该下标对应的位置作为键值对的存储位置，通过该方法建立的数组就叫做**哈希表**，而这个存储位置就叫做**桶（bucket）**。数组是通过整数下标直接访问元素，哈希表是通过字符串key直接访问元素，也就说哈希表是一种特殊的数组（关联数组），哈希表广泛应用于实现数据的快速查找（在map的key集合中，一旦存储的key的数量特别多，那么在要查找某个key的时候就会变得很麻烦，数组中的key需要挨个比较，哈希的出现，使得这样的比较次数大大减少。）

哈希表选用哈希函数计算哈希值时，可能不同的 key 会得到相同的结果，一个地址怎么存放多个数据呢？这就是**哈希冲突（碰撞）**。解决哈希冲突有两种方法，**拉链法（链接法）**和**开放定址法**（这种没用过）。拉链法：将键值对对象封装为一个node结点，新增了next指向，这样就可以将碰撞的结点链接成一条单链表，保存在该地址（数组位置）中。

原文：<https://blog.csdn.net/qq_36711757/article/details/80394272>

## HashMap

### 2.1 HashMap简介

#### 2.1.1 HashMap是什么

解释一：

HashMap是一个实现了Map接口的基于哈希表的类 。

也就是说，HashMap既有map的键值对特点，也有哈希表的特点

简单点说，利用HashMap类：

查找时，给出一个关键字key，我们可以根据hash算法计算出key-value的存储位置然后取出value

存储时，我们根据哈希算法计算出该键值对应该存储的位置，将其存进去。

也就是说，当没有冲突时，HashMap存取的时间复杂度为O(1)

这是HashMap类的部分代码(部分数据域和构造函数)

解释二：

HashMap 是一个散列表，它存储的内容是键值对(key-value)映射。  
HashMap 继承于AbstractMap，实现了Map、Cloneable、java.io.Serializable接口。

HashMap 的实现不是同步的，这意味着它不是线程安全的。它的key、value都可以为null。此外，HashMap中的映射不是有序的。

HashMap 的实例有两个参数影响其性能：“**初始容量**” 和 “**加载因子**”。容量 是哈希表中桶的数量，初始容量 只是哈希表在创建时的容量。加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。

通常，**默认加载因子是 0.75**, 这是在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本（在大多数 HashMap 类的操作中，包括 get 和 put 操作，都反映了这一点）。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少 rehash 操作次数。如果初始容量大于最大条目数除以加载因子，则不会发生 rehash 操作。

**HashMap的构造函数**

HashMap共有**4个构造函数**,如下：

[复制代码](javascript:void(0);)

// 默认构造函数。

HashMap()

// 指定“容量大小”的构造函数

HashMap(int capacity)

// 指定“容量大小”和“加载因子”的构造函数

HashMap(int capacity, float loadFactor)

// 包含“子Map”的构造函数

HashMap(Map<? extends K, ? extends V> map)

[复制代码](javascript:void(0);)

**HashMap的API**

[复制代码](javascript:void(0);)

void clear()

Object clone()

boolean containsKey(Object key)

boolean containsValue(Object value)

Set<Entry<K, V>> entrySet()

V get(Object key)

boolean isEmpty()

Set<K> keySet()

V put(K key, V value)

void putAll(Map<? extends K, ? extends V> map)

V remove(Object key)

int size()

Collection<V> values()

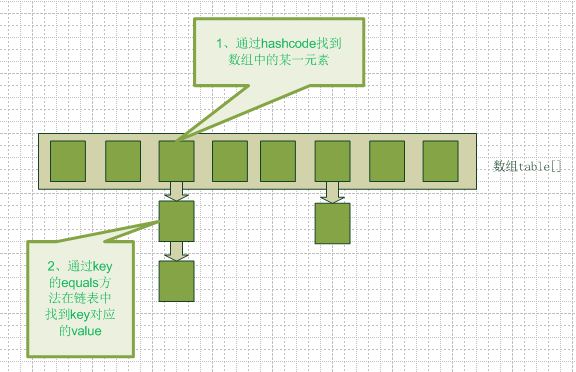
#### 2.1.2动机

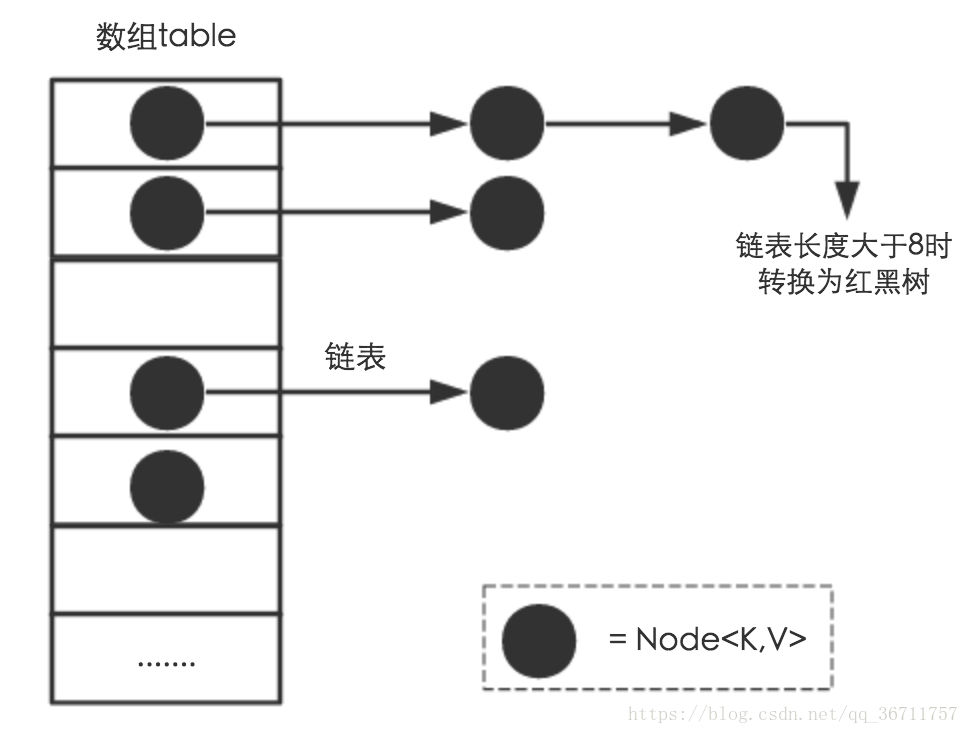
为了实现非常快速和高效地地根据键值key进行数据的存取。

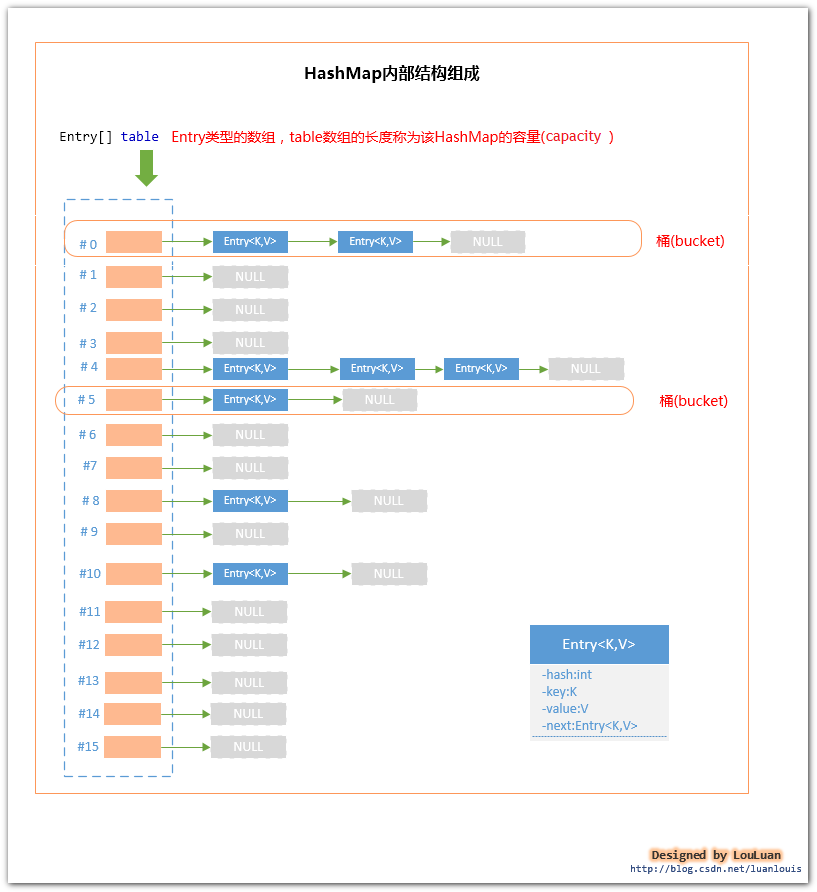
### HashMap的存储结构

#### 2.2.1 hashmap的数据结构

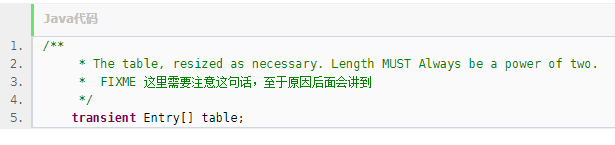
要知道hashmap是什么，首先要搞清楚它的数据结构，在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，hashmap也不例外。Hashmap实际上是一个数组和链表的结合体（在数据结构中，一般称之为“链表散列“），请看下图（横排表示数组，纵排表示数组元素【实际上是一个链表】）。

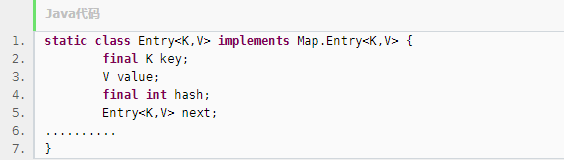






从图中我们可以看到一个hashmap就是一个数组结构，当新建一个hashmap的时候，就会初始化一个数组。我们来看看java代码：





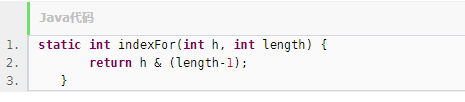
上面的Entry就是数组中的元素，它持有一个指向下一个元素的引用，这就构成了链表。

当我们往hashmap中put元素的时候，先根据key的hash值得到这个元素在数组中的位置（即下标），然后就可以把这个元素放到对应的位置中了。如果这个元素所在的位子上已经存放有其他元素了，那么在同一个位子上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。从hashmap中get元素时，首先计算key的hashcode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。从这里我们可以想象得到，如果每个位置上的链表只有一个元素，那么hashmap的get效率将是最高的，但是理想总是美好的，现实总是有困难需要我们去克服，哈哈~

#### 2.2.2 hash算法

我们可以看到在hashmap中要找到某个元素，需要根据key的hash值来求得对应数组中的位置。如何计算这个位置就是hash算法。前面说过hashmap的数据结构是数组和链表的结合，所以我们当然希望这个hashmap里面的元素位置尽量的分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，而不用再去遍历链表。

所以我们首先想到的就是把hashcode对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，“模”运算的消耗还是比较大的，能不能找一种更快速，消耗更小的方式那？java中时这样做的：



#### 2.2.3 hashmap的resize

当hashmap中的元素越来越多的时候，碰撞的几率也就越来越高（因为数组的长度是固定的），所以为了提高查询的效率，就要对hashmap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，所以这是一个通用的操作，很多人对它的性能表示过怀疑，不过想想我们的“均摊”原理，就释然了，而在hashmap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。   
        那么hashmap什么时候进行扩容呢？当hashmap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，也就是说，默认情况下，数组大小为16，那么当hashmap中元素个数超过16\*0.75=12的时候，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。比如说，我们有1000个元素new HashMap(1000), 但是理论上来讲new HashMap(1024)更合适，不过上面annegu已经说过，即使是1000，hashmap也自动会将其设置为1024。 但是new HashMap(1024)还不是更合适的，因为0.75\*1000 < 1000, 也就是说为了让0.75 \* size > 1000, 我们必须这样new HashMap(2048)才最合适，既考虑了&的问题，也避免了resize的问题。

#### 2.2.4 key的hashcode与equals方法改写

在第一部分hashmap的数据结构中，annegu就写了get方法的过程：首先计算key的hashcode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。所以，hashcode与equals方法对于找到对应元素是两个关键方法。 

Hashmap的key可以是任何类型的对象，例如User这种对象，为了保证两个具有相同属性的user的hashcode相同，我们就需要改写hashcode方法，比方把hashcode值的计算与User对象的id关联起来，那么只要user对象拥有相同id，那么他们的hashcode也能保持一致了，这样就可以找到在hashmap数组中的位置了。如果这个位置上有多个元素，还需要用key的equals方法在对应位置的链表中找到需要的元素，所以只改写了hashcode方法是不够的，equals方法也是需要改写滴~当然啦，按正常思维逻辑，equals方法一般都会根据实际的业务内容来定义，例如根据user对象的id来判断两个user是否相等。   
在改写equals方法的时候，需要满足以下三点：   
(1) 自反性：就是说a.equals(a)必须为true。   
(2) 对称性：就是说a.equals(b)=true的话，b.equals(a)也必须为true。   
(3) 传递性：就是说a.equals(b)=true，并且b.equals(c)=true的话，a.equals(c)也必须为true。   
通过改写key对象的equals和hashcode方法，我们可以将任意的业务对象作为map的key(前提是你确实有这样的需要)。 

总结：

        本文主要描述了HashMap的结构，和hashmap中hash函数的实现，以及该实现的特性，同时描述了hashmap中resize带来性能消耗的根本原因，以及将普通的域模型对象作为key的基本要求。尤其是hash函数的实现，可以说是整个HashMap的精髓所在，只有真正理解了这个hash函数，才可以说对HashMap有了一定的理解。

### 2.3 HashMap的特点

• 底层实现是链表数组，JDK8后又加了红黑树

• 实现Map全部的方法

• key 用 Set 存放，所以想做到 key 不允许重复，key 对应的类（一般是String）需要重写 hashCode 和 equals 方法

• 允许空键和空值（但空键只有一个，且放在第一位，知道就行）

• 元素是无序的，而且顺序会不定时改变（每次扩容后，都会重新哈希，也就是key通过哈希函数计算后会得出与之前不同的哈希值，这就导致哈希表里的元素是没有顺序，会随时变化的，这是因为哈希函数与桶数组容量有关，每次结点到了临界值后，就会自动扩容，扩容后桶数组容量都会乘二，而key不变，那么哈希值一定会变）

• 插入、获取的时间复杂度基本是 O(1)（前提是有适当的哈希函数，让元素分布在均匀的位置）

• 遍历整个 Map 需要的时间与数组的长度成正比（因此初始化时 HashMap 的容量不宜太大）

• 两个关键因子：初始容量、加载因子

• HashMap不是同步，HashTable是同步的，但HashTable已经弃用，如果需要线程安全，可以用synchronizedMap，例如            Map m = Collections.synchronizedMap(new HashMap(...));

### 2.4 HashMap的线程安全问题

#### 2.4.1 为什么线程不安全？

HashMap 在并发时可能出现的问题主要体现在以下几个方面：

1. 首先如果多个线程同时使用put方法添加元素，而且假设正好存在两个 put 的 key 发生了碰撞(根据 hash 值计算的 bucket 一样)，那么根据 HashMap 的实现，这两个 key 会添加到数组的同一个位置，这样最终就会发生其中一个线程的 put 的数据被覆盖。
2. 第二就是如果多个线程同时检测到元素个数超过数组大小\* loadFactor ，这样就会发生多个线程同时对 Node 数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给 table，也就是说其他线程的都会丢失，并且各自线程 put 的数据也丢失。
3. 多个线程操作HashMap在rehash过程中容易发生死循环。
4. 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

具体分析过程参见：

<https://blog.csdn.net/qq_33275597/article/details/79692056>

<http://www.importnew.com/22011.html>

#### 2.4.2 如何避免或解决HashMap的线程安全问题

三种解决方案

1、**Hashtable替换HashMap**

Hashtable 是同步的，但由迭代器返回的 Iterator 和由所有 Hashtable 的“collection 视图方法”返回的 Collection 的 listIterator 方法都是快速失败的：在创建 Iterator 之后，如果从结构上对 Hashtable 进行修改，除非通过 Iterator 自身的移除或添加方法，否则在任何时间以任何方式对其进行修改，Iterator 都将抛出 ConcurrentModificationException。因此，面对并发的修改，Iterator 很快就会完全失败，而不冒在将来某个不确定的时间发生任意不确定行为的风险。由 Hashtable 的键和值方法返回的 Enumeration 不是快速失败的。

注意，迭代器的快速失败行为无法得到保证，因为一般来说，不可能对是否出现不同步并发修改做出任何硬性保证。快速失败迭代器会尽最大努力抛出 ConcurrentModificationException。因此，为提高这类迭代器的正确性而编写一个依赖于此异常的程序是错误做法：迭代器的快速失败行为应该仅用于检测程序错误。

**2、Collections.synchronizedMap将HashMap包装起来**

返回由指定映射支持的同步（线程安全的）映射。为了保证按顺序访问，必须通过返回的映射完成对底层映射的所有访问。在返回的映射或其任意 collection 视图上进行迭代时，强制用户手工在返回的映射上进行同步：

Map m = Collections.synchronizedMap(new HashMap());

...

Set s = m.keySet(); // Needn't be in synchronized block

...

synchronized(m) { // Synchronizing on m, not s!

Iterator i = s.iterator(); // Must be in synchronized block

while (i.hasNext())

foo(i.next());

}

不遵从此建议将导致无法确定的行为。如果指定映射是可序列化的，则返回的映射也将是可序列化的。

**3、ConcurrentHashMap替换HashMap**

支持检索的完全并发和更新的所期望可调整并发的哈希表。此类遵守与 Hashtable 相同的功能规范，并且包括对应于 Hashtable 的每个方法的方法版本。不过，尽管所有操作都是线程安全的，但检索操作不必锁定，并且不支持以某种防止所有访问的方式锁定整个表。此类可以通过程序完全与 Hashtable 进行互操作，这取决于其线程安全，而与其同步细节无关。

检索操作（包括 get）通常不会受阻塞，因此，可能与更新操作交迭（包括 put 和 remove）。检索会影响最近完成的更新操作的结果。对于一些聚合操作，比如 putAll 和 clear，并发检索可能只影响某些条目的插入和移除。类似地，在创建迭代器/枚举时或自此之后，Iterators 和 Enumerations 返回在某一时间点上影响哈希表状态的元素。它们不会抛出 ConcurrentModificationException。不过，迭代器被设计成每次仅由一个线程使用。

## HashTable

### 3.1 HashTable介绍

和[HashMap](http://www.cnblogs.com/skywang12345/p/3310835.html)一样，Hashtable 也是一个**散列表**，它存储的内容是**键值对(key-value)映射**。  
Hashtable **继承于Dictionary**，实现了Map、Cloneable、java.io.Serializable接口。  
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。此外，Hashtable中的映射不是有序的。

Hashtable 的实例有两个参数影响其性能：**初始容量** 和 **加载因子**。容量 是哈希表中桶 的数量，初始容量 就是哈希表创建时的容量。注意，哈希表的状态为 open：在发生“哈希冲突”的情况下，单个桶会存储多个条目，这些条目必须按顺序搜索。加载因子 是对哈希表在其容量自动增加之前可以达到多满的一个尺度。初始容量和加载因子这两个参数只是对该实现的提示。关于何时以及是否调用 rehash 方法的具体细节则依赖于该实现。  
通常，**默认加载因子是 0.75**, 这是在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查找某个条目的时间（在大多数 Hashtable 操作中，包括 get 和 put 操作，都反映了这一点）。

**Hashtable的构造函数**

[复制代码](javascript:void(0);)

// 默认构造函数。

public Hashtable()

// 指定“容量大小”的构造函数

public Hashtable(int initialCapacity)

// 指定“容量大小”和“加载因子”的构造函数

public Hashtable(int initialCapacity, float loadFactor)

// 包含“子Map”的构造函数

public Hashtable(Map<? extends K, ? extends V> t)

[复制代码](javascript:void(0);)

**Hashtable的API**

[复制代码](javascript:void(0);)

synchronized void clear()

synchronized Object clone()

boolean contains(Object value)

synchronized boolean containsKey(Object key)

synchronized boolean containsValue(Object value)

synchronized Enumeration<V> elements()

synchronized Set<Entry<K, V>> entrySet()

synchronized boolean equals(Object object)

synchronized V get(Object key)

synchronized int hashCode()

synchronized boolean isEmpty()

synchronized Set<K> keySet()

synchronized Enumeration<K> keys()

synchronized V put(K key, V value)

synchronized void putAll(Map<? extends K, ? extends V> map)

synchronized V remove(Object key)

synchronized int size()

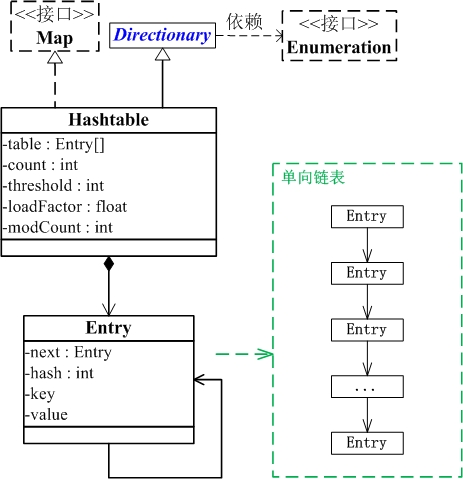
synchronized String toString()

synchronized Collection<V> values()

[复制代码](javascript:void(0);)

### 3.2 HashTable的数据结构

**Hashtable与Map关系如下图：**



从图中可以看出：

(01) Hashtable继承于Dictionary类，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。   
(02) Hashtable是通过"拉链法"实现的哈希表。它包括几个重要的成员变量：table, count, threshold, loadFactor, modCount。  
　　table是一个Entry[]数组类型，而Entry实际上就是一个单向链表。哈希表的"key-value键值对"都是存储在Entry数组中的。   
　　count是Hashtable的大小，它是Hashtable保存的键值对的数量。   
　　threshold是Hashtable的阈值，用于判断是否需要调整Hashtable的容量。threshold的值="容量\*加载因子"。

loadFactor就是加载因子。

modCount是用来实现fail-fast机制的

其实，HashTable的存储结构和HashMap是一样的，值不过是HashTable是线程安全的，而HashMap不是线程安全的。因为，在每个HashTable的公开的方法比如get都使用了synchronized描述符。而遍历视图比如keySet都使用了Collections.synchronizedXXX进行了同步包装。

### 3.3 HashTable和HashMap的区别

区别：

1. 两个类的继承体系有些不同。虽然都实现了Map、Cloneable、Serializable三个接口。但是HashMap继承自抽象类AbstractMap，而HashTable继承自抽象类Dictionary。其中Dictionary类是一个已经被废弃的类。所以，在 新的jdk中，HashTable不再继承Dictionary。
2. HashTable比HashMap多了两个公开方法。一个是elements，这来自于抽象类Dictionary，鉴于该类已经废弃，所以这个方法也就没什么用处了。另一个多出来的方法是contains，这个多出来的方法也没什么用，因为它跟containsValue方法功能是一样的。
3. HashMap是支持null键和null值的，而HashTable在遇到null时，会抛出NullPointerException异常。这并不是因为HashTable有什么特殊的实现层面的原因导致不能支持null键和null值，这仅仅是因为HashMap在实现时对null做了特殊处理，将null的hashCode值定为了0，从而将其存放在哈希表的第0个bucket中。
4. 初始化大小不同，扩充机制不同。

HashTable默认的初始大小为11，之后每次扩充为原来的2n+1。HashMap默认的初始化大小为16，之后每次扩充为原来的2倍。如果在创建时给定了初始化大小，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。

也就是说HashTable会尽量使用素数、奇数。而HashMap则总是使用2的幂作为哈希表的大小。我们知道当哈希表的大小为素数时，简单的取模哈希的结果会更加均匀（具体证明，见这篇文章），所以单从这一点上看，HashTable的哈希表大小选择，似乎更高明些。但另一方面我们又知道，在取模计算时，如果模数是2的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法。所以从hash计算的效率上，又是HashMap更胜一筹。

所以，事实就是HashMap为了加快hash的速度，将哈希表的大小固定为了2的幂。当然这引入了哈希分布不均匀的问题，所以HashMap为解决这问题，又对hash算法做了一些改动。具体我们来看看，在获取了key对象的hashCode之后，HashTable和HashMap分别是怎样将他们hash到确定的哈希桶（Entry数组位置）中的。

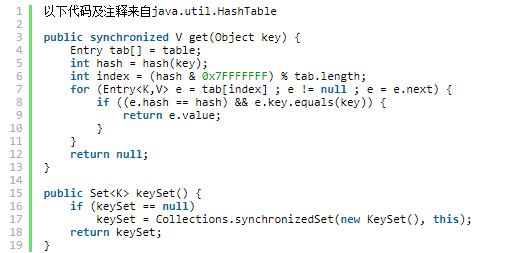


正如我们所言，HashMap由于使用了2的幂次方，所以在取模运算时不需要做除法，只需要位的与运算就可以了。但是由于引入的hash冲突加剧问题，HashMap在调用了对象的hashCode方法之后，又做了一些位运算在打散数据。关于这些位计算为什么可以打散数据的问题，本文不再展开了。感兴趣的可以看这里。

如果你有细心读代码，还可以发现一点，就是HashMap和HashTable在计算hash时都用到了一个叫hashSeed的变量。这是因为映射到同一个hash桶内的Entry对象，是以链表的形式存在的，而链表的查询效率比较低，所以HashMap/HashTable的效率对哈希冲突非常敏感，所以可以额外开启一个可选hash（hashSeed），从而减少哈希冲突。因为这是两个类相同的一点，所以本文不再展开了，感兴趣的看这里。事实上，这个优化在JDK 1.8中已经去掉了，因为JDK 1.8中，映射到同一个哈希桶（数组位置）的Entry对象，使用了红黑树来存储，从而大大加速了其查找效率。

1. 线程安全：HashTable是线程安全的，而HashMap不是线程安全的。

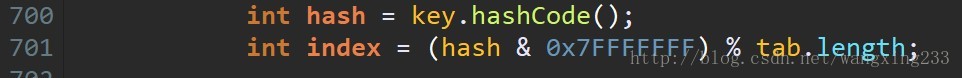
我们说HashTable是同步的，HashMap不是，也就是说HashTable在多线程使用的情况下，不需要做额外的同步，而HashMap则不行。那么HashTable是怎么做到的呢？



可以看到，也比较简单，就是公开的方法比如get都使用了synchronized描述符。而遍历视图比如keySet都使用了Collections.synchronizedXXX进行了同步包装。

1. 计算hash值的方法不同

为了得到元素的位置，首先需要根据元素的 KEY计算出一个hash值，然后再用这个hash值来计算得到最终的位置。



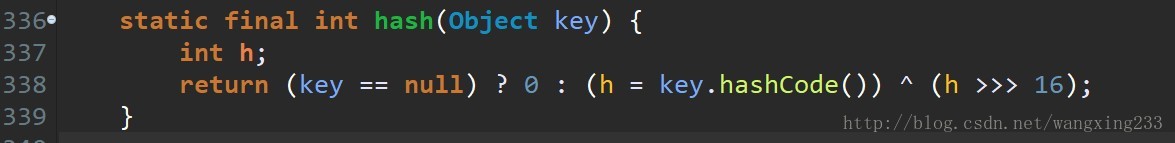
Hashtable直接使用对象的hashCode。hashCode是JDK根据对象的地址或者字符串或者数字算出来的int类型的数值。然后再使用除留余数发来获得最终的位置。

Hashtable在计算元素的位置时需要进行一次除法运算，而除法运算是比较耗时的。

HashMap为了提高计算效率，将哈希表的大小固定为了2的幂，这样在取模预算时，不需要做除法，只需要做位运算。位运算比除法的效率要高很多。

HashMap的效率虽然提高了，但是hash冲突却也增加了。因为它得出的hash值的低位相同的概率比较高，而计算位运算

为了解决这个问题，HashMap重新根据hashcode计算hash值后，又对hash值做了一些运算来打散数据。使得取得的位置更加分散，从而减少了hash冲突。当然了，为了高效，HashMap只做了一些简单的位处理。从而不至于把使用2 的幂次方带来的效率提升给抵消掉。



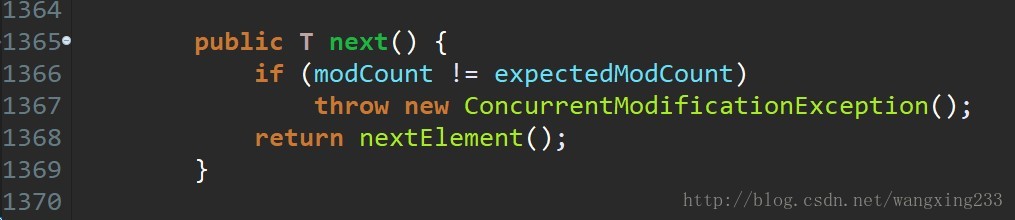
原文：<https://blog.csdn.net/wangxing233/article/details/79452946>

1. 遍历方式的内部实现上不同

Hashtable、HashMap都使用了 Iterator。而由于历史原因，Hashtable还使用了Enumeration的方式 。

HashMap的Iterator是fail-fast迭代器。当有其它线程改变了HashMap的结构（增加，删除，修改元素），将会抛出ConcurrentModificationException。不过，通过Iterator的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。

JDK8之前的版本中，Hashtable是没有fast-fail机制的。在JDK8及以后的版本中 ，HashTable也是使用fast-fail的， 源码如下：



modCount的使用类似于并发编程中的CAS（Compare and Swap）技术。我们可以看到这个方法中，每次在发生增删改的时候都会出现modCount++的动作。而modcount可以理解为是当前hashtable的状态。每发生一次操作，状态就向前走一步。设置这个状态，主要是由于hashtable等容器类在迭代时，判断数据是否过时时使用的。尽管hashtable采用了原生的同步锁来保护数据安全。但是在出现迭代数据的时候，则无法保证边迭代，边正确操作。于是使用这个值来标记状态。一旦在迭代的过程中状态发生了改变，则会快速抛出一个异常，终止迭代行为。

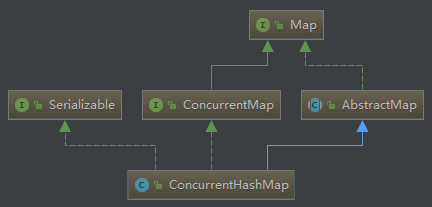
注：HashTable已经被淘汰，涉及到线程安全的地方需用ConcurrentHashMap。

## ConcurrentHashMap

### 4.1 为什么使用ConcurrentHashMap？

我们知道，HashMap是非线程安全的，也就是说，在多线程操作下，HashMap在put和resize过程中，会导致死循环，使得CPU的占用率接近100%。HashTable虽然是线程安全的，但因为采用同步锁，导致效率低下。为了解决这两个问题，即要求既快又安全，所以产生了ConcurrentHashMap。

### 4.2 ConcurrentHashMap的类图



### 4.3 ConcurrentHashMap的实现原理

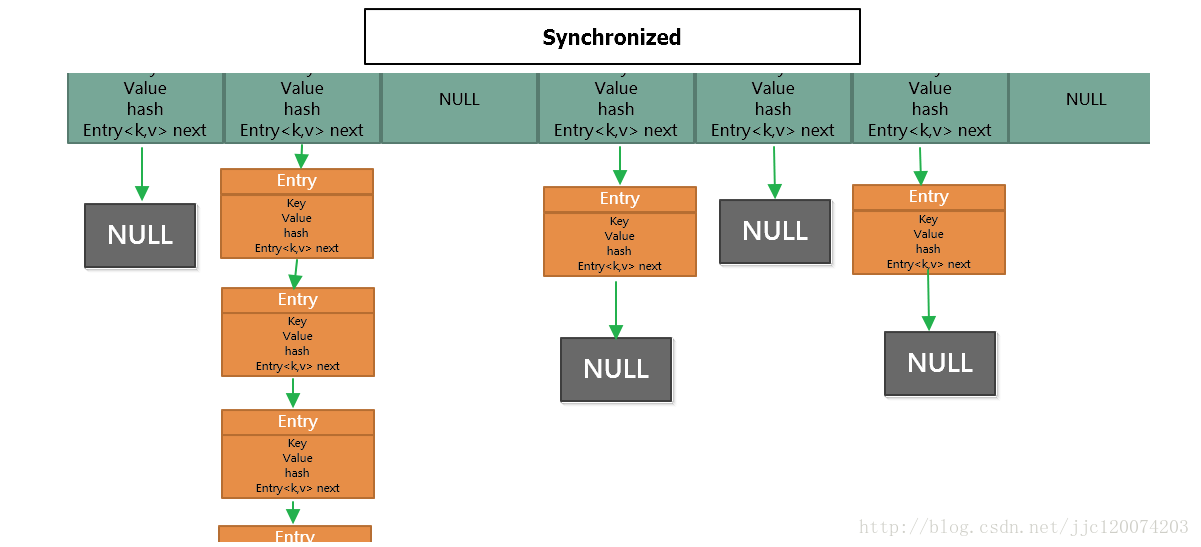
#### 4.3.1 锁分段技术

原文：<https://blog.csdn.net/jjc120074203/article/details/78625433>

**HashTable**put()源代码



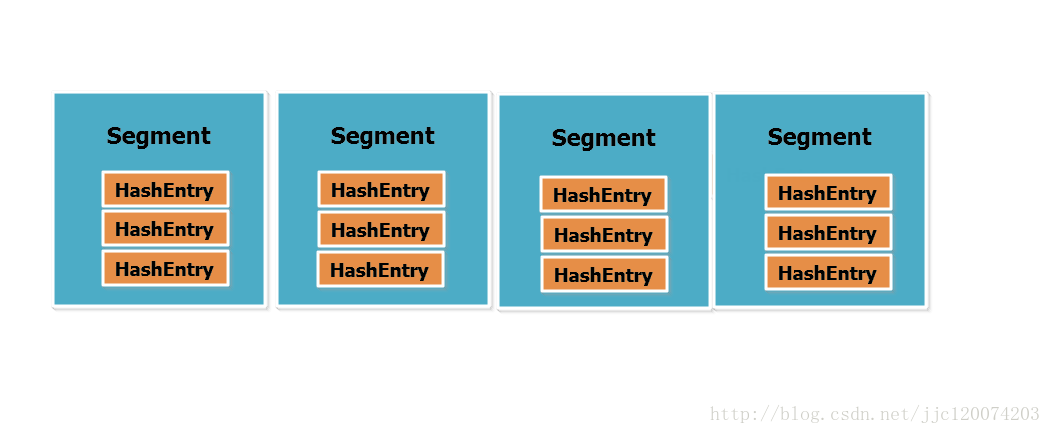
从代码可以看出来在所有put 的操作的时候 都需要用 synchronized 关键字进行同步。并且key 不能为空。

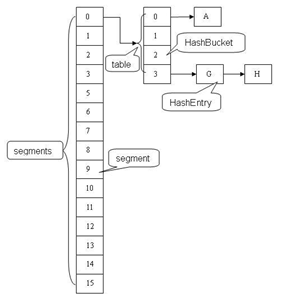


这样相当于每次进行put 的时候都会进行同步 当10个线程同步进行操作的时候，就会发现当第一个线程进去 其他线程必须等待第一个线程执行完成，才可以进行下去。性能特别差。

CurrentHashMap

分段锁技术：ConcurrentHashMap相比 HashTable而言解决的问题就是 的 它不是锁全部数据，而是锁一部分数据，这样多个线程访问的时候就不会出现竞争关系。不需要排队等待了。



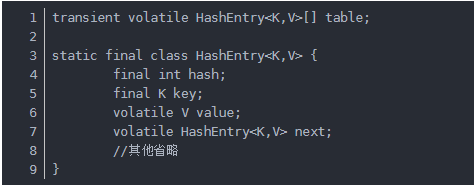


从图中可以看出来ConcurrentHashMap的主干是个Segment数组。

这就是为什么ConcurrentHashMap支持允许多个修改同时并发进行，原因就是采用的Segment分段锁功能，每一个Segment 都想的于小的hash table并且都有自己锁，只要修改不再同一个段上就不会引起并发问题。

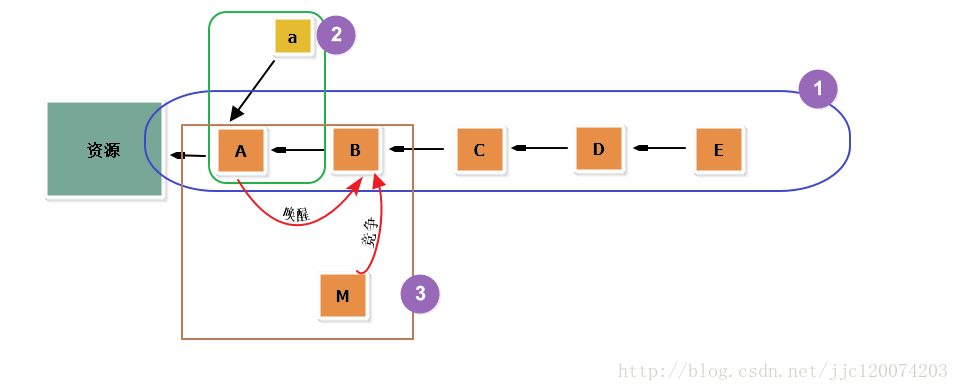


使用ConConcurrentHashMap时候 有时候会遇到跨段的问题，跨段的时候【size()、 containsValue()】，可能需要锁定部分段或者全段，当操作结束之后，又回按照 顺序 进行 释放 每一段的锁。注意是按照顺序解锁的。，每个Segment又包含了多个HashEntry。



需要注意的是 Segment 是一种可重入锁（继承ReentrantLock)。

#### 4.3.2 ReentrantLock 与synchronized



（1）synchronized 是一个同步锁 synchronized （this）

同步锁 当一个线程A 访问 【资源】的代码同步块的时候，A线程就会持续持有当前锁的状态，如果其他线程B-E 也要访问【资源】的代码同步块的时候将会收到阻塞，因此需要排队等待A线程释放锁的状态。（如图情况1）但是注意的是，当一个线程B-E 只是不能方法 A线程 【资源】的代码同步块，仍然可以访问其他的非资源同步块。

（2）ReentrantLock 可重入锁 通常两类：公平性、非公平性

公平性：根据线程请求锁的顺序依次获取锁，当一个线程A 访问 【资源】的期间，线程A 获取锁资源，此时内部存在一个计数器num+1，在访问期间，线程B、C请求 资源时，发现A 线程在持有当前资源，因此在后面生成节点排队（B 处于待唤醒状态），假如此时a线程再次请求资源时，不需要再次排队，可以直接再次获取当前资源 （内部计数器+1 num=2） ，当A线程释放所有锁的时候（num=0），此时会唤醒B线程进行获取锁的操作，其他C-E线程就同理。（情况2）

非公平性：当A线程已经释放所之后，准备唤醒线程B获取资源的时候，此时线程M 获取请求，此时会出现竞争，线程B 没有竞争过M线程，测试M获取的线程因此，M会有限获得资源，B继续睡眠。（情况2）

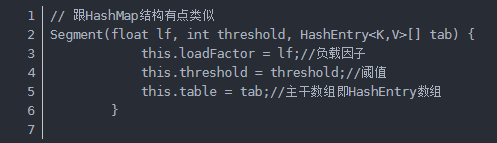
synchronized 是一个非公平性锁。 非公平性 会比公平性锁的效率要搞很多原因，不需要通知等待。

ReentrantLock 提供了 new Condition可以获得多个Condition对象,可以简单的实现比较复杂的线程同步的功能.通过await(),signal()以实现。

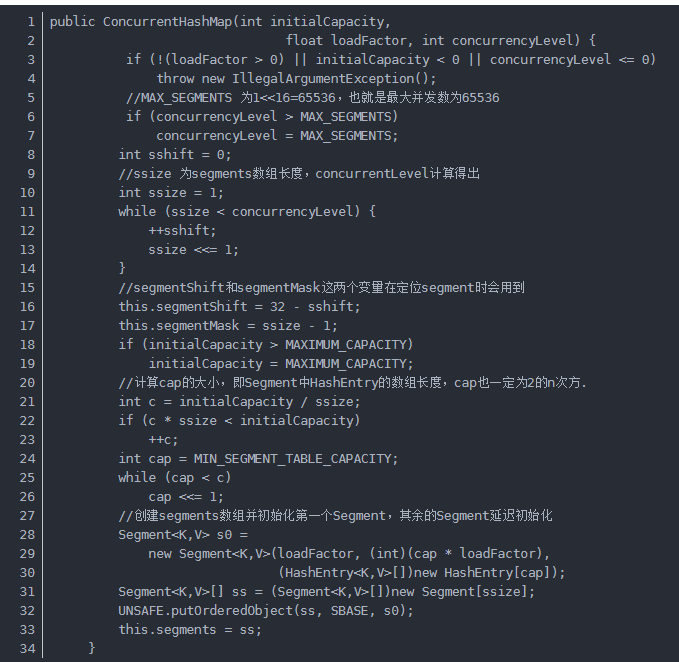
ReentrantLock 提供可以中断锁的一个方法lock.lockInterruptibly()方法。

Jdk 1.8 synchronized和 ReentrantLock 比较的话，官方比较建议用synchronized。

在了解Segment 机制之后我们继续看一下ConcurrentHashMap核心构造方法代码。



构造方法



从以上代码可以看出ConcurrentHashMap有比较重要的三个参数：

1、loadFactor 负载因子 0.75

2、threshold 初始 容量 16

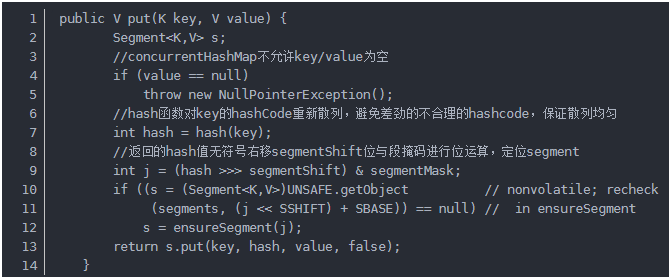
3、concurrencyLevel 实际上是Segment的实际数量。

#### 4.3.3 ****ConcurrentHashMap如何发生ReHash****

ConcurrentLevel 一旦设定的话，就不会改变。ConcurrentHashMap当元素个数大于临界值的时候，就会发生扩容。但是ConcurrentHashMap与其他的HashMap不同的是，它不会对Segmengt 数量增大，只会增加Segmengt 后面的链表容量的大小。即对每个Segmengt 的元素进行的ReHash操作。

#### 4.3.4 put、get和remove

我们再看一下核心的ConcurrentHashMapput ()方法：



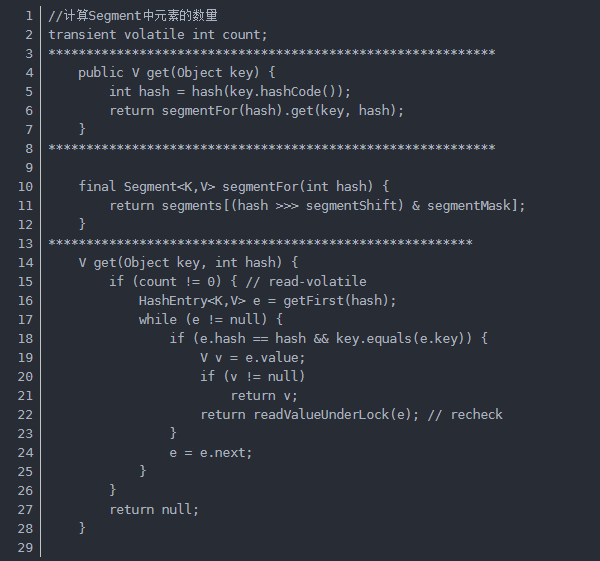
主要注意的是 当前put 方法 当前key 为空的时候 ，代码报错。  
这个代码主要是把Key 通过Hash函数计算出hash值 **现计算出当前key属于那个Segment** 调用**Segment**.put 分段方法Segment.put()。

ConcurrentHashMap的put方法代码省略……

1、Put的时候 ，通过Hash函数将即将要put 的元素均匀的放到所需要的Segment 段中，调用Segment的put 方法进行数据。

2、Segment的put 是加锁中完成的。如果当前元素数大于最大临界值的的话将会产生rehash. 先通过 getFirst 找到链表的表头部分，然后遍历链表，调用equals 比配是否存在相同的key ,如果找到的话，则将最新的Key 对应value值。如果没有找到，新增一个HashEntry 它加到整个Segment的头部。

我们先看一下Get 方法的源码：



1.读取的时候 传递Key值，通过Hash函数计算出 对应Segment 的位置。

2.调用segmentFor（int hash） 方法，用于确定操作应该在哪一个segment中进行 ，通过 右无符号位运算 右移segmentShift位在与运算 segmentMask【偏移码】 获得需要操作的Segment

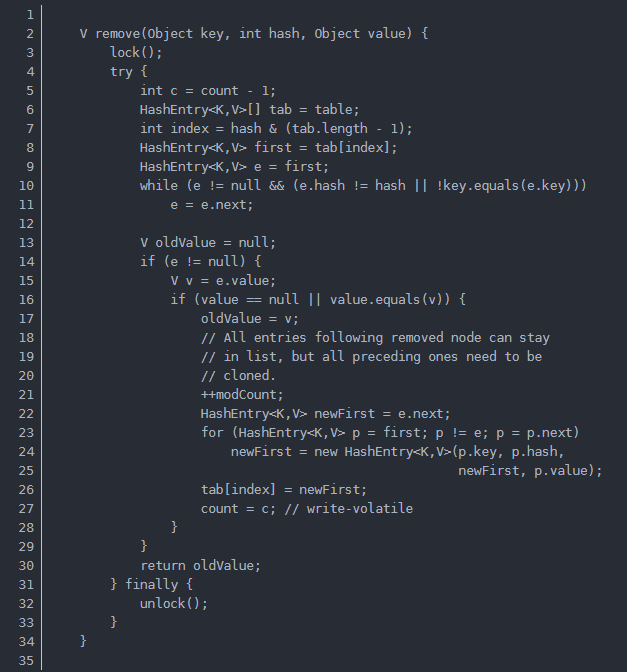
确定了需要操作的Segment 再调用 get 方法获取对应的值。通过count 值先判断当前值是否为空。在调用getFirst（）获取头节点，然后遍历列表通过equals对比的方式进行比对返回值。

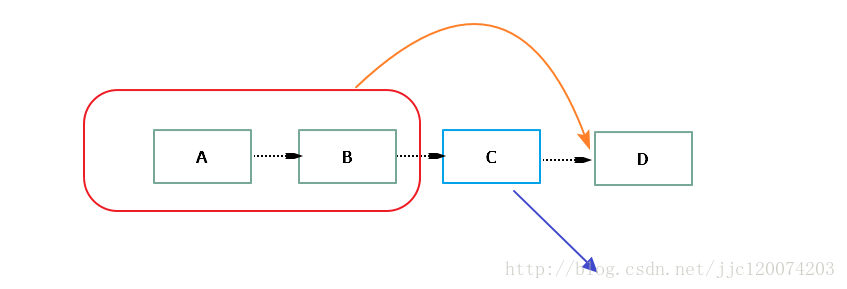
**ConcurrentHashMap为什么读的时候不加锁？**

ConcurrentHashMap是分段并发分段进行读取数据的。

Segment 里面有一个Count 字段，用来表示当前Segment中元素的个数 它的类型是volatile变量。所有的操作到最后都会 在最后一部更新count 这个变量，由于volatile变量 happer-before的特性。导致get 方法能够几乎准确的获取最新的结构更新。

再看一下ConcurrentHashMapRemove()方法：





1. 调用Segment 的remove 方法，先定位当前要删除的元素C，此时需要把A、B元素全部复制一遍，一个一个接入到D上。
2. remove 也是在加锁的情况下进行的。

#### 4.3.5 volatile 变量

我们发现 对于CurrentHashMap而言的话，源码里面又很多地方都用到了这个变量。比如HashEntry 、value 、Segment元素个数Count。

volatile 属于JMM 模型中的一个词语。首先先简单说一下 Java内存模型中的 几个概念：

**原子性**：保证 Java内存模型中原子变量内存操作的。通常有 read、write、load、use、assign、store、lock、unlock等这些。

**可见性**：就是当一个线程对一个变量进行了修改，其他线程即可立即得到这个变量最新的修改数据。

**有序性**：如果在本线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。

**先行发生**：happen-before 先行发生原则是指Java内存模型中定义的两项操作之间的依序关系，如果说操作A先行发生于操作B，其实就是说发生操作B之前.

**传递性**

**volatile 变量 与普通变量的不同之处？**

1、volatile 是有可见性，一定程度的有序性。

2、volatile 赋值的时候新值能够立即刷新到主内存中去，每次使用的时候能够立刻从内存中刷新。

## TreeMap

### 5.1 TreeMap简介

TreeMap 是一个**有序的key-value集合**，它是通过[红黑树](http://www.cnblogs.com/skywang12345/p/3245399.html)实现的。

TreeMap **继承于AbstractMap**，所以它是一个Map，即一个key-value集合。

TreeMap 实现了NavigableMap接口，意味着它**支持一系列的导航方法。**比如返回有序的key集合。

TreeMap 实现了Cloneable接口，意味着**它能被克隆**。

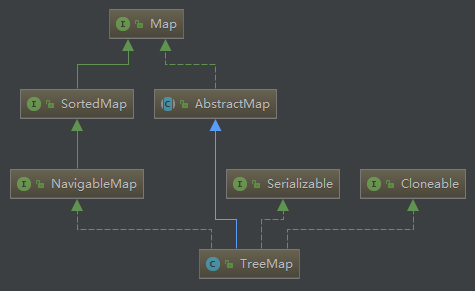
TreeMap 实现了java.io.Serializable接口，意味着**它支持序列化**。

TreeMap基于**红黑树（Red-Black tree）实现**。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的基本操作 containsKey、get、put 和 remove 的时间复杂度是 log(n) 。

另外，TreeMap是**非同步**的。 它的iterator 方法返回的**迭代器是fail-fastl**的。

### 5.2 TreeMap类图



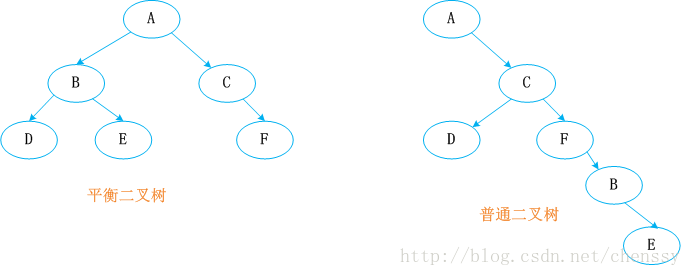
### 5.3 红黑树

#### 5.3.1 红黑树简介

红黑树又称红-黑二叉树，它首先是一颗二叉树，它具体二叉树所有的特性。同时红黑树更是一颗自平衡的排序二叉树。

       我们知道一颗基本的二叉树他们都需要满足一个基本性质--即树中的任何节点的值大于它的左子节点，且小于它的右子节点。按照这个基本性质使得树的检索效率大大提高。我们知道在生成二叉树的过程是非常容易失衡的，最坏的情况就是一边倒（只有右/左子树），这样势必会导致二叉树的检索效率大大降低（O(n)），所以为了维持二叉树的平衡，大牛们提出了各种实现的算法，如：[AVL](http://baike.baidu.com/view/414610.htm)，[SBT](http://baike.baidu.com/view/2957252.htm)，[伸展树](http://baike.baidu.com/view/1118088.htm)，[TREAP](http://baike.baidu.com/view/956602.htm) ，[红黑树](http://baike.baidu.com/view/133754.htm?fr=aladdin#1_1)等等。

       平衡二叉树必须具备如下特性：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。也就是说该二叉树的任何一个等等子节点，其左右子树的高度都相近。



       红黑树顾名思义就是节点是红色或者黑色的平衡二叉树，它通过颜色的约束来维持着二叉树的平衡。对于一棵有效的红黑树二叉树而言我们必须增加如下规则：

**1、每个节点都只能是红色或者黑色**

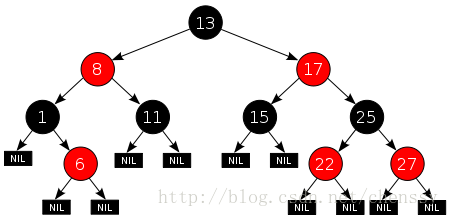
**2、根节点是黑色**

**3、每个叶节点（NIL节点，空节点）是黑色的。**

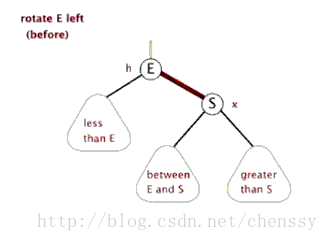
**4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。**

**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。**

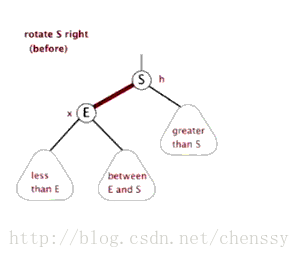
这些约束强制了红黑树的关键性质: 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这棵树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。所以红黑树它是复杂而高效的，其检索效率O(log *n*)。下图为一颗典型的红黑二叉树。



对于红黑二叉树而言它主要包括三大基本操作：左旋、右旋、着色。



左旋

右旋

关于TreeMap和红黑树可以仔细阅读以下两篇文章：

<https://blog.csdn.net/see__you__again/article/details/51713307>

<http://www.cnblogs.com/skywang12345/p/3245399.html>

## fail-fastl

文章：

<https://www.cnblogs.com/ccgjava/p/6347425.html?utm_source=itdadao&utm_medium=referral>

## HashSet