hashmap的底层结构是一个哈希数组

hashtable的底层是一个哈希表，已经被废弃了，因为效率不好，concureentHashMap是其替代者

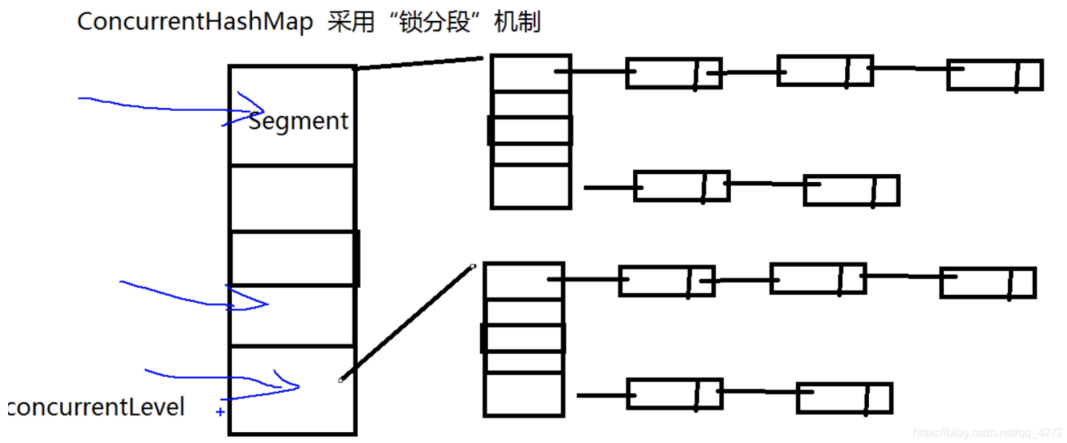
```
相比于hashmap,hashtable的区别:
    1、hashtable多提供了两个方法
        elements():Enumeration elements = hashtable.elements();//返回value的枚举
        contains(): boolean contains = hashtable.contains(new Object());//判断传入
    2、hashmap继承的是abstractMap，而hashtable继承的是Dictonnary
    3、hashMap几乎可以等价于hashtable，只是hashmap是非synchronized，而hashTable是synch
        hashtable在基本操作前面都加上了synchornized锁，也就是只要有一个线程进入了synchorni
    4、扩容不一样：hashtable默认的初始化大小是11，每次扩容变为原来的2*N+1),hashmap初始是16,
    5、计算hash的方法不同
        hashtable直接使用对象的hashcode除留余数法来获取最终的位置
        注意：hashcode是JDK根据对象的地址或者字符串或者数字计算出来的int类型的数值
        hashmap为了提高效率，将哈希表的大小固定位2的幂，这样取模运算时，不需要做除法，只需要位
```

注意:

ToString()方法默认输出的是对象的包名+类名（全类名）+此对象对应内存的首地址

hashCode方法的实现注释说是跟对象的内存地址生成的，但是对象内存地址变化了怎么办，其实也不是，他是使用的随机数生成

concurentHashMap的分段锁机制（1.8之后已经不再使用分段锁，而是转为数组的某个位置的第一个元素进行加锁，降低了锁的粒度）



其中有16个分段，每个分段都有独立的锁机制，每个独立的机制都是一张表，表的元素是一个链表，并发时对锁分段进行加锁的机制。提高了效率（这是锁的粒度减小带来的锁减少）

```
关于Segment三个参数
    1、initialCapacity：初始总容量，默认16
    2、loadFactor：加载因子，默认0.75，当数组大小大于16*0.75=12的时候就会扩容，尽量预估大小
```

```
    3、concurentLevel：并发级别，默认16，并发级别控制了segement的个数，扩容过程其实是改变的
```

```
1、Segment继承了ReetrantLock，有了锁的功能，每个锁控制一段，但是segment越来越大时，锁的粒度就
2、分段锁的优势在于操作不同map的时候可以并发的执行，操作同段map的时候，进行锁的竞争和等待，相比全
3、但是分段很多的时候回浪费内存空间，导致内存碎片化，当操作同一map的同一个分段锁的概率很小的时候
 jdk1.8的时候跟hashmap一样，列表链表变成了链表+红黑树的结构，
     1、当元素的个数超过默认的8个，数组还没超过64的时候进行简单的链表扩容，数组如果超过64此时扩
```

为什么不用reetrantLock而用synchronized?

减少内存开销，如果使用reetrantLock则需要节点继承AQS，来获得同步支持，增加内存开销

内部优化：synchornized是JVM支持的，能够在运行的时候做出相应的优化措施：锁粗话，锁消除，锁自旋

1.8的时候concurentHashMap的改动

抛弃了原有的segement分段锁，而采用cas和synchornized来保证安全性

也将1.7中存放数据的hashEntry改为node，但作用都是一样 的，其中的val，next都使用了volatile修饰，保证了可见性

HashSet

内部维护了一个hashMap，key是元素的值，value是提前初始化的一个Object

put方法：按照hashmap的方式进行插入，

hashMap的put：添加新元素，则返回null，添加了旧元素，则返回被替换的元素（表示添加失败）

hashSet的put：hashmap.put()==null

则添加新元素返回的是true，添加就元素返回的是false

关于**hashtable**

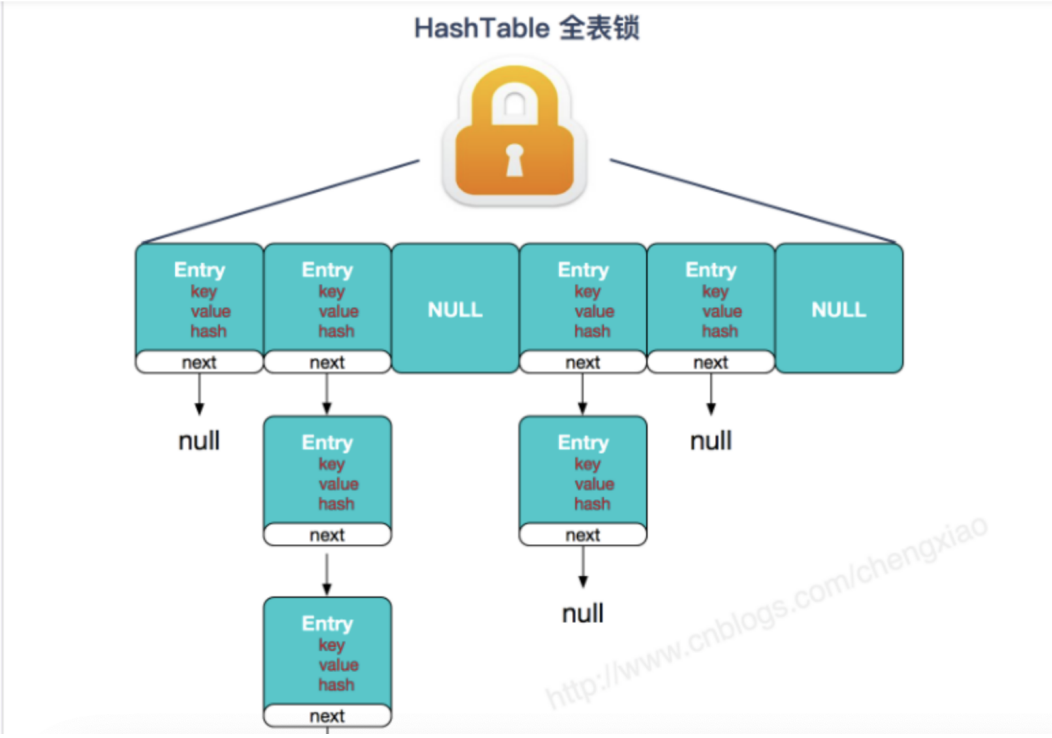
hashtable与hashmap的实现原理几乎一样，

差别：

1、hashtable不允许key与value为null

2、hashtable是线程安全的，但是安全策略的实现代价却比较大，他的get和put的相关的所有操作都是synchrnized的，这相当于给整个哈希表加了一把大锁，多线程访问的时候只要一个线程访问或操作该对象，其他线程只能阻塞

如下图通过**hash去定位bucket，之后通过hash去比较**



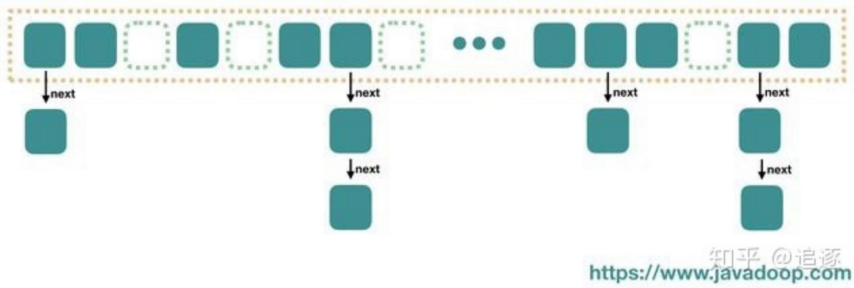
注意：

对象的默认hashCode()方法的返回值

他是一个与线程状态有关的随机数

字符串常量池（String pool）的底层数据结构

基础类型包装类的缓存池使用一个数组进行缓存，而 String 类型，JVM 内部使用 HashTable 进行缓存，我们知道，HashTable 的结构是一个数组，数组中每个元素是一个链表。和我们平时使用的 HashTable 不同，JVM 内部的这个 HashTable 是不可以动态扩容的。



字符串常量池的底层是一个hashtable，数组中的每个元素是一个链表，与平时使用的hashtable不同，JVM内部字符串常量池的hashtable是不可以动态扩容的

这个不可扩容的特性导致当hash冲突较多的时候，单个bucket中的链表很长，导致性能降低，为了降低hash冲突，将bucket的数量增加到了1009，jdk7之后固定在了60013（为什么 要选择用这些数字，就是因为这些数字是一个质数，能大大降低哈希冲突），他虽然不可扩容，但是可以rehash，当发现散列不均匀的时候会进行rehash

hashMap的扩容操作

他的扩容是通过resize方法实现的，扩容发生在putVal方法的最后，即写入元素之后才会判断是否需要扩容操作，当自增之后的size大于之前所计算好的阈值（一般为0.75，即如果大小是16，则元素数量达到12的时候就会扩容），即进行resize操作，扩容为原来的两倍，同时新的阈值也扩容为老阈值的两倍

扩容时候有三种情况

1、如果哈希桶数组中某个位置只有一个元素，即不存在哈希冲突时，则直接将该元素复制到新哈希桶数组的对应位置即可（下标是重新计算的）

2、哈希桶数组中的某个位置的节点为树节点时，则执行红黑树的扩容操作

3、哈希桶数组的某个位置节点为普通节点时候，则执行链表扩容操作，1.8中为了避免版本中并发扩容导致的死链问题，引入了高低位链表辅助进行扩容

有冲突出现的时候，采用高低位链表处理方式，通过元素的hash与就数组容量做与操作，判断落在高位还是低位，例如原有容量是8扩容之后是16，0-7是低位，8-15是高位，扩容的时候，与操作的结果在低位，那就是原位置，再用低位链将这类元素链接起来，如果在高位，则用高位低链表链接起来，最后将高低位链表的头节点分别放在扩容后的数组的执行位置，便完成了扩容

高低链的方法，降低了对共喜资源的newTab的访问频次，新组织冲突节点，最后放入newTab的指定位置，避免了每遍历一个元素就放入newTab中，从而导致并发扩容的死链问题

concurentHashMap 的set方法

1、传入k-v的时候，首先通过hash方法对key的hashCode值计算出一个hash值

2、使用一个tab的循环，防止tab初始化中、tab正在扩容、添加失败，则循环后再次put

3、 如果table是空的，则通过resize方法创建默认数组容量为16的数组

如果 通过数组最大索引与1中的hash值做与位运算，确定在数组中的位置，并判断该位置是空的，那么就创建一个node节点存在该数组中

如果该位置的Node正在扩容，则帮助进行扩容

如果 该位置有元素（对当前元素使用synchornized加锁）

判断如果当前位置是链表：对链表进行遍历，如下

判断put进来的key与当前数组Node节点的key使用相同（hash值相同，key通过equls方法判断相等），则直接返回旧的value值

如果走到了链表的末尾，即node.next==null ,则创建Node并放在这个位置

判断如果是红黑树就放在红黑树里面

添加结束之后会判断链表的长度是否大于8，如果是的话

判断数组的长度时候小于64，小于的话进行扩容，大于64的话将链表转为红黑树

关于扩容

hashmap扩容是根据加载因子即0.75*容量<元素数量 则进行扩容

concurentHashMap则是判断链表的长度，是否大于8，如果是的话，判断数组的长度时候小于64，小于的话进行扩容，大于64的话将链表转为红黑树

为什么扩容是扩容为2倍，因为这样数组的长度就是2的N次方，那么最大的下标就是2的你次方-1也就是每个位置都是1的二进制数，这样一方面在通过hash值与最大下标就可以进行与操作，充分反应hash值的特性，另一方面也方便扩容时判断元素扩容后的位置（高低位链表辅助）

参考链接:

JDK1.7 ConcurrentHashMap 底层分析

创建ConcurrentHashMap对象时:
  创建一个长度为16的大数组, 加载因子是0.75 (Segment[])
  创建一个长度为2的小数组, 将地址值赋值给0索引处, 其他索引位置都为null (HashEntry[])

添加元素时, 根据键的哈希值来计算出在大数组中的位置
  如果为null, 按照模板创建小数组
  创建完毕, 会二次哈希计算出在小数组中应存入的位置, 由于第一次都是null所以直接存入
  如果不为null, 会二次哈希, 计算出在小数组中应存入的位置
    如果小数组需要扩容, 则扩容为2倍 (存到索引1的地方)
    如果不需要扩容, 则会判断小数组当前索引位置是否为null
      如果为null代表没有元素, 直接存入
      如果不为null代表有元素, 则根据equals方法比较属性值
        一样则不存
        不一样则将老元素挂在新元素下, 形成链表 (哈希桶)

JDK1.8 ConcurrentHashMap 底层分析

ConcurrentHashMap在JDK1.8底层分析:
  结构: 哈希表 (数组 + 链表 + 红黑树)
  线程安全: CAS机制 + synchronized同步代码块

1. 如果使用空参构造创建ConcurrentHashMap对象时, 则什么都不做 (查看空参构造及父类的空参)
2. 在第一次添加元素时 (调用put方法时) 创建哈希表 (initTable方法)
   计算当前元素应存入的索引位置
     如果为null, 代表没有元素, 则通过CAS算法, 将本节点添加到数组中
     如果不为null, 代表有元素, 则利用volatile获得当前索引位置最新的节点地址, 挂在它下面, 形成链表
       当链表长度大于等于8的时候, 自动转为红黑树
3. 每次操作, 会以链表或者树的头结点为锁对象, 配合 悲观锁 (synchronized) 保证多线程操作集合时的安全问题

https://blog.csdn.net/qq_43294932/article/details/125113936