**一、ArrayList和LinkedList异同**

**1. 是否保证线程安全**

ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

**2. 底层数据结构**

Arraylist 底层使用的是Object数组；LinkedList 底层使用的是双向循环链表数据结构；

**3. 插入和删除是否受元素位置的影响**

① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 比如：执行add(E e)方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（add(int index, E element)）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ② LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。

**4. 是否支持快速随机访问**

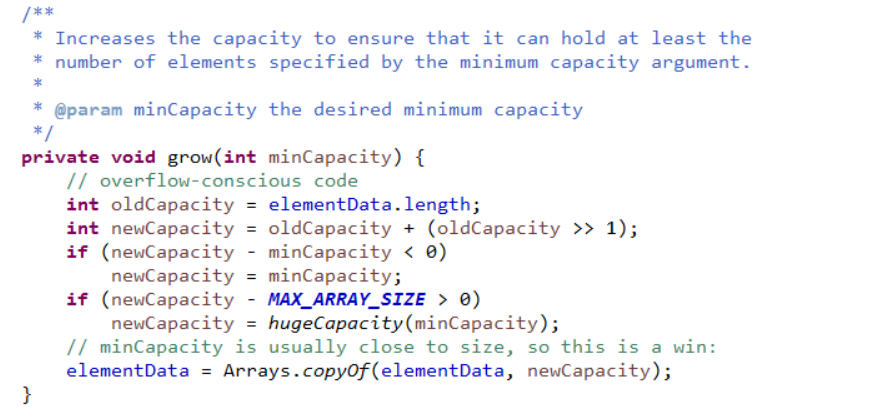
LinkedList 不支持高效的随机元素访问，而ArrayList 实现了RandmoAccess 接口，所以有随机访问功能。快速随机访问就是通过元素的序号快速获取元素对象(对应于get(int index)方法)。

**5. 内存空间占用**

ArrayList的空间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

**6.ArrayList中添加元素时数组的调整**

在创建ArrayList的时候可以不传入参数、传入Collection或者传入初始容量。当向ArrayList中添加元素时，会先调用ensureCapacityInternal(size + 1)方法，然后获得（当前的数组元素的数量+1）和DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA=10中的最大值，然后通过grow方法增加容量，oldCapacity >> 1表示的就是oldCapacity的一半。

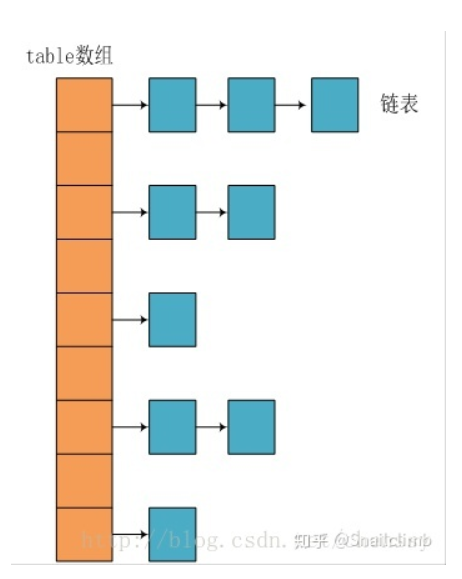


**二、HashMap底层实现**

**1.JDK1.8以前HashMap的底层实现**

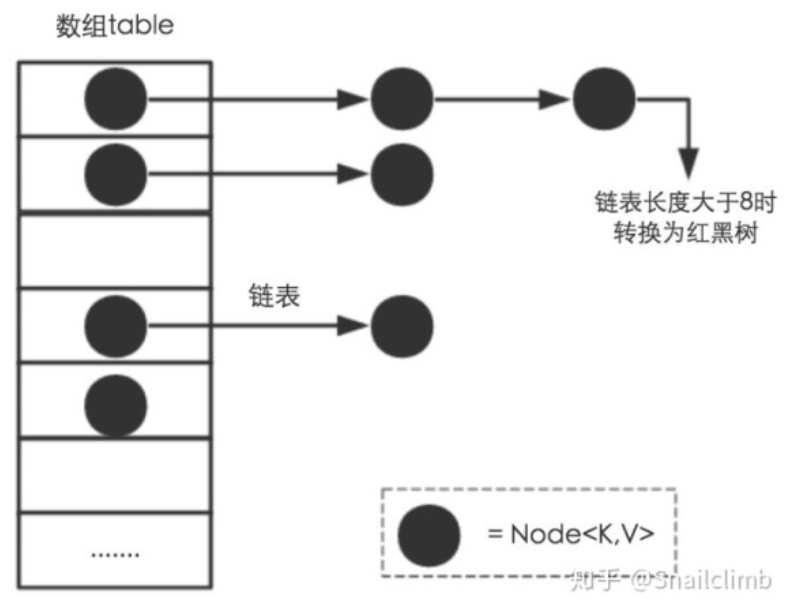
JDK1.8 之前 HashMap 由 数组+链表 组成的（“链表散列” 即数组和链表的结合体），数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（HashMap 采用 “拉链法也就是链地址法” 解决冲突），如果定位到的数组位置不含链表（当前 entry 的 next 指向 null ）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度依然为 O(1)，因为最新的 Entry 会插入链表头部，急需要简单改变引用链即可，而对于查找操作来讲，此时就需要遍历链表，然后通过 key 对象的 equals 方法逐一比对查找.

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。



**2.JDK1.8以后**

相比于之前的版本， JDK1.8之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。



TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

**三、HashMap与HashTable的区别**

**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。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

**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 put 进的键值只要有一个 null，直接抛出 NullPointerException。

**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。也就是说 HashMap 总是使用2的幂作为哈希表的大小,后面会介绍到为什么是2的幂次方。

**四、HashMap的长度为什么是2的n次方**

**1.Java中HashCode算法详解**

Java中的集合，比如HashMap/HashSet/HashTable在实现上都用到了hashCode算法，用来计算元素在数组中的位置。hashCode是Object类中的一个方法，所以，所有的Java类都有这个方法，只是一些类对这个方法进行了覆写，下面以String类的实现为例进行说明：

public int hashCode() {

int h =hash;

if (h ==0 &&value.length >0)

{

      char val[] =value;

        for (int i =0; i < value.length; i++) {

        h =31 \* h + val[i];

        }

      hash = h;

}

return h;

}

其实这个算法的实现很简单，以“hangzhou”这个字符串为例，计算过程如下：

第一步：int ‘h’

第二步：31 \* （第一步结果） + int ‘a’

第三步：31 \* （第二部结果） + int ‘n’

第四步：31 \* （第三步结果） + int ‘g’

第五步：31 \* （第四步结果） + int ‘z’

第六步：31 \* （第五步结果） + int ‘h’

第七步：31 \* （第六步结果） + int ‘o’

第八步： 31 \* （第七步结果） + int ‘u’可以得到“hangzhou”的hashcode为4740586。

**2.为什么HashMap中的&位必须位奇数（length-1）**

从key映射到HashMap数组的对应位置需要一个Hash函数：index = Hash("hangzhou")如何实现一个尽量分布均匀的hash函数呢？我们使用key的hashcode做某种运算：index = hashCode("hangzhou") & (Length - 1) 其中，Length为HashMap的长度，下面来演示整个过程：

1、“hangzhou”的hashcode为4740586，二进制表示为100 1000 0101 0101 1110 1010

2、假定HashMap的长度为默认的16，则Length - 1为15，也就是二进制的1111

3、把以上两个结果做与运算，得到的结果为1010，也就是index为10

可以说，Hash算法最终得到的index结果完全取决于hashCode的最后几位。假设，HashMap的长度为10，则Length - 1为9，也就是二进制的1001，通过Hash算法得到的最终index为8，当只有一个元素的时候这没问题。但是我们再来试一个hashCode：100 1000 0101 0101 1110 1110时，通过Hash算法得到的最终的index也是8，另外还有100 1000 0101 0101 1110 1000得到的index也是8。也就是说，即使我们把倒数第二、三位的0、1变换，得到的index仍旧是8，说明有些index结果出现的几率变大！！而有些index结果永远不会出现，比如二进制0000.

这样，显然不符合Hash算法均匀分布的要求。反观，长度16或其他2的幂次方，Length - 1的值的二进制所有的位均为1，这种情况下，Index的结果等于hashCode的最后几位。只要输入的hashCode本身符合均匀分布，Hash算法的结果就是均匀的。

一句话，HashMap的长度为2的幂次方的原因是为了减少Hash碰撞，尽量使Hash算法的结果均匀分布。

**五、HashSet和HashMap的区别**



**六、ConcurrentHashMap和HashTable的区别**

**1.底层数据结构**

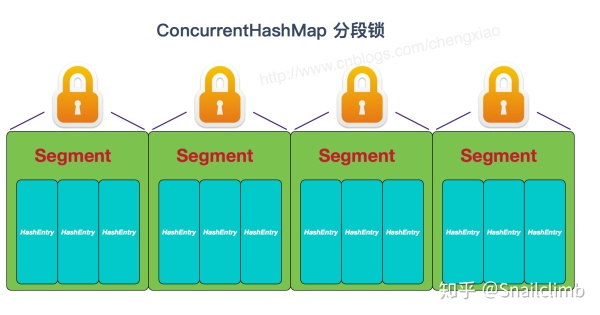
JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；

**2. 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。（默认分配16个Segment，比Hashtable效率提高16倍。） 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。（JDK1.6以后 对 synchronized锁做了很多优化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；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。

**七、ConcurrentHashMap线程安全的具体实现方式/底层具体实现**

**1.JDK1.7中**



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。

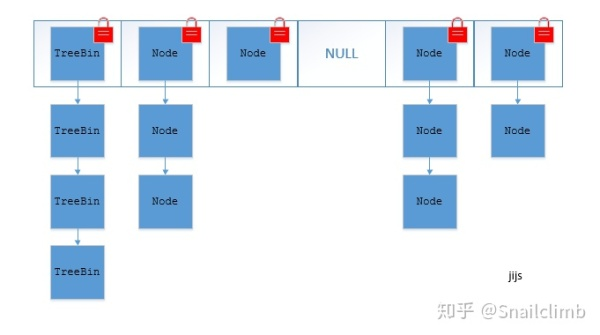
ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成

Segment 实现了 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。

**2. JDK1.8中**



JDK1.8的ConcurrentHashMap（TreeBin: 红黑二叉树节点 Node: 链表节点）：

ConcurrentHashMap取消了Segment分段锁，采用CAS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。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。