# 关于ArrayList，LinkList以及vector的区别以及底层的实现方式：

ArrayList底层使用的是数组数据结构，因为数组有索引（角标）所以ArrayList的查询速度快，而添加删除元素速度稍慢。因为，你每删除或者添加一个元素，你都要移动所添加或删除元素后面的所有数据，该集合是线程不同步的  
LinkedList底层使用的是链表数据结构，链表数据结构是没有索引的，当前元素只和他的前一个和后一个元素有关联就像一串珠子一样，该数据结构的特点是，增加删除快，而查询比较慢，因为增加删除只需要找到当前元素，然后断掉当前元素与它前一个和后一个元素的关联即可，和数组比，链表不用重复大部分的数据移动工作，但是因为没有索引所以链表数据结构要一个一个的查询数据，所以LinkedList的查询速度稍慢  
Vector：底层使用的是数组数据结构，和ArrayList的功能相当，但是Vector是线程同步的，JDK已经对Vector的底层源码中可能出现同步的操作加上了同步锁，但是线程同步之后有弊端，那就是每当操作Vector集合，都会去判断同步锁，较为浪费系统资源。而我们完全可以使用ArrayList来代替Vector，如果需要同步代码，我们也可以自己手动加上同步锁；、

# HashMap、HashTable、CurrentHashMap的区别于实现原理

**HashTable ：**

底层数组+链表实现，无论key还是value都不能为null，线程安全，实现线程安全的方式是在修改数据时锁住整个HashTable，效率低，ConcurrentHashMap做了相关优化

初始size为11，扩容：newsize = olesize\*2+1

计算index的方法：index = (hash & 0x7FFFFFFF) % tab.length

**HashMap ：**

底层数组+链表实现，可以存储null键和null值，线程不安全

初始size为16，扩容：newsize = oldsize\*2，size一定为2的n次幂

扩容针对整个Map，每次扩容时，原来数组中的元素依次重新计算存放位置，并重新插入

插入元素后才判断该不该扩容，有可能无效扩容（插入后如果扩容，如果没有再次插入，就会产生无效扩容）

当Map中元素总数超过Entry数组的75%，触发扩容操作，为了减少链表长度，元素分配更均匀

计算index方法：index = hash & (tab.length – 1)

HashMap的初始值还要考虑加载因子:

哈希冲突：若干Key的哈希值按数组大小取模后，如果落在同一个数组下标上，将组成一条Entry链，对Key的查找需要遍历Entry链上的每个元素执行equals()比较。

加载因子：为了降低哈希冲突的概率，默认当HashMap中的键值对达到数组大小的75%时，即会触发扩容。因此，如果预估容量是100，即需要设定100/0.75＝134的数组大小。

空间换时间：如果希望加快Key查找的时间，还可以进一步降低加载因子，加大初始大小，以降低哈希冲突的概率。

HashMap和Hashtable都是用hash算法来决定其元素的存储，因此HashMap和Hashtable的hash表包含如下属性：

容量（capacity）：hash表中桶的数量

初始化容量（initial capacity）：创建hash表时桶的数量，HashMap允许在构造器中指定初始化容量

尺寸（size）：当前hash表中记录的数量

负载因子（load factor）：负载因子等于“size/capacity”。负载因子为0，表示空的hash表，0.5表示半满的散列表，依此类推。轻负载的散列表具有冲突少、适宜插入与查询的特点（但是使用Iterator迭代元素时比较慢）

除此之外，hash表里还有一个“负载极限”，“负载极限”是一个0～1的数值，“负载极限”决定了hash表的最大填满程度。当hash表中的负载因子达到指定的“负载极限”时，hash表会自动成倍地增加容量（桶的数量），并将原有的对象重新分配，放入新的桶内，这称为rehashing。

HashMap和Hashtable的构造器允许指定一个负载极限，HashMap和Hashtable默认的“负载极限”为0.75，这表明当该hash表的3/4已经被填满时，hash表会发生rehashing

“负载极限”的默认值（0.75）是时间和空间成本上的一种折中：

较高的“负载极限”可以降低hash表所占用的内存空间，但会增加查询数据的时间开销，而查询是最频繁的操作（HashMap的get()与put()方法都要用到查询）

较低的“负载极限”会提高查询数据的性能，但会增加hash表所占用的内存开销

可以根据实际情况来调整“负载极限”值。

**ConcurrentHashMap ：**

底层采用分段的数组+链表实现，线程安全

通过把整个Map分为N个Segment，可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。(读操作不加锁，由于HashEntry的value变量是 volatile的，也能保证读取到最新的值。)

Hashtable的synchronized是针对整张Hash表的，即每次锁住整张表让线程独占，ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁

扩容：段内扩容（段内元素超过该段对应Entry数组长度的75%触发扩容，不会对整个Map进行扩容），插入前检测需不需要扩容，有效避免无效扩容

Hashtable和HashMap都实现了Map接口，但是Hashtable的实现是基于Dictionary抽象类的。Java5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。

HashMap基于哈希思想，实现对数据的读写。当我们将键值对传递给put()方法时，它调用键对象的hashCode()方法来计算hashcode，然后找到bucket位置来存储值对象。当获取对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。HashMap使用链表来解决碰撞问题，当发生碰撞时，对象将会储存在链表的下一个节点中。HashMap在每个链表节点中储存键值对对象。当两个不同的键对象的hashcode相同时，它们会储存在同一个bucket位置的链表中，可通过键对象的equals()方法来找到键值对。如果链表大小超过阈值（TREEIFY\_THRESHOLD,8），链表就会被改造为树形结构。

在HashMap中，null可以作为键，这样的键只有一个，但可以有一个或多个键所对应的值为null。当get()方法返回null值时，即可以表示HashMap中没有该key，也可以表示该key所对应的value为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个key，应该用containsKey()方法来判断。而在Hashtable中，无论是key还是value都不能为null。

Hashtable是线程安全的，它的方法是同步的，可以直接用在多线程环境中。而HashMap则不是线程安全的，在多线程环境中，需要手动实现同步机制。

Hashtable与HashMap另一个区别是HashMap的迭代器（Iterator）是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的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。

ConcurrentHashMap是使用了锁分段技术来保证线程安全的。

锁分段技术：首先将数据分成一段一段的存储，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的时候，其他段的数据也能被其他线程访问。

ConcurrentHashMap提供了与Hashtable和SynchronizedMap不同的锁机制。Hashtable中采用的锁机制是一次锁住整个hash表，从而在同一时刻只能由一个线程对其进行操作；而ConcurrentHashMap中则是一次锁住一个桶。

ConcurrentHashMap默认将hash表分为16个桶，诸如get、put、remove等常用操作只锁住当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有16个写线程执行，并发性能的提升是显而易见的。

JAVA集合接口及类：

