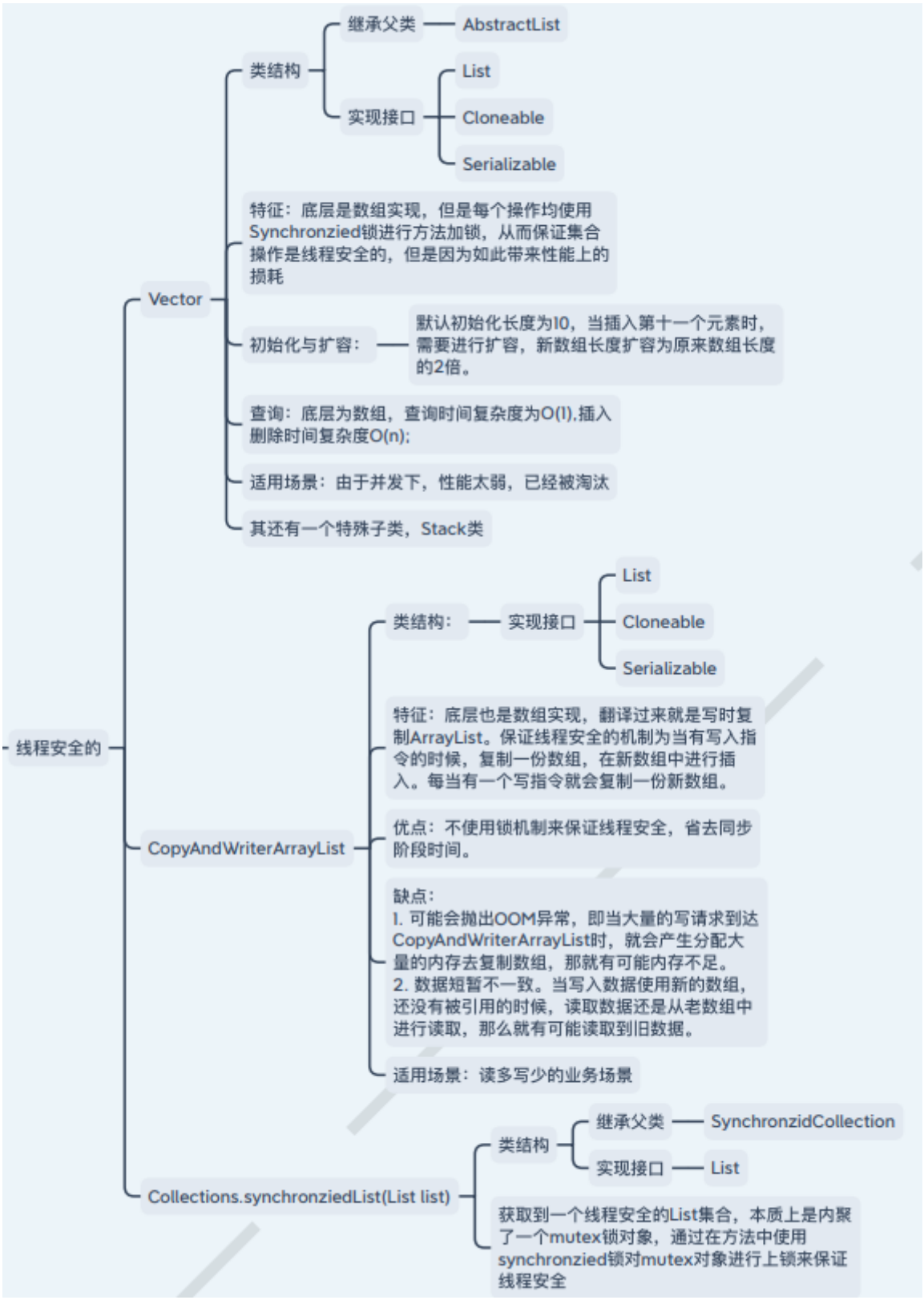
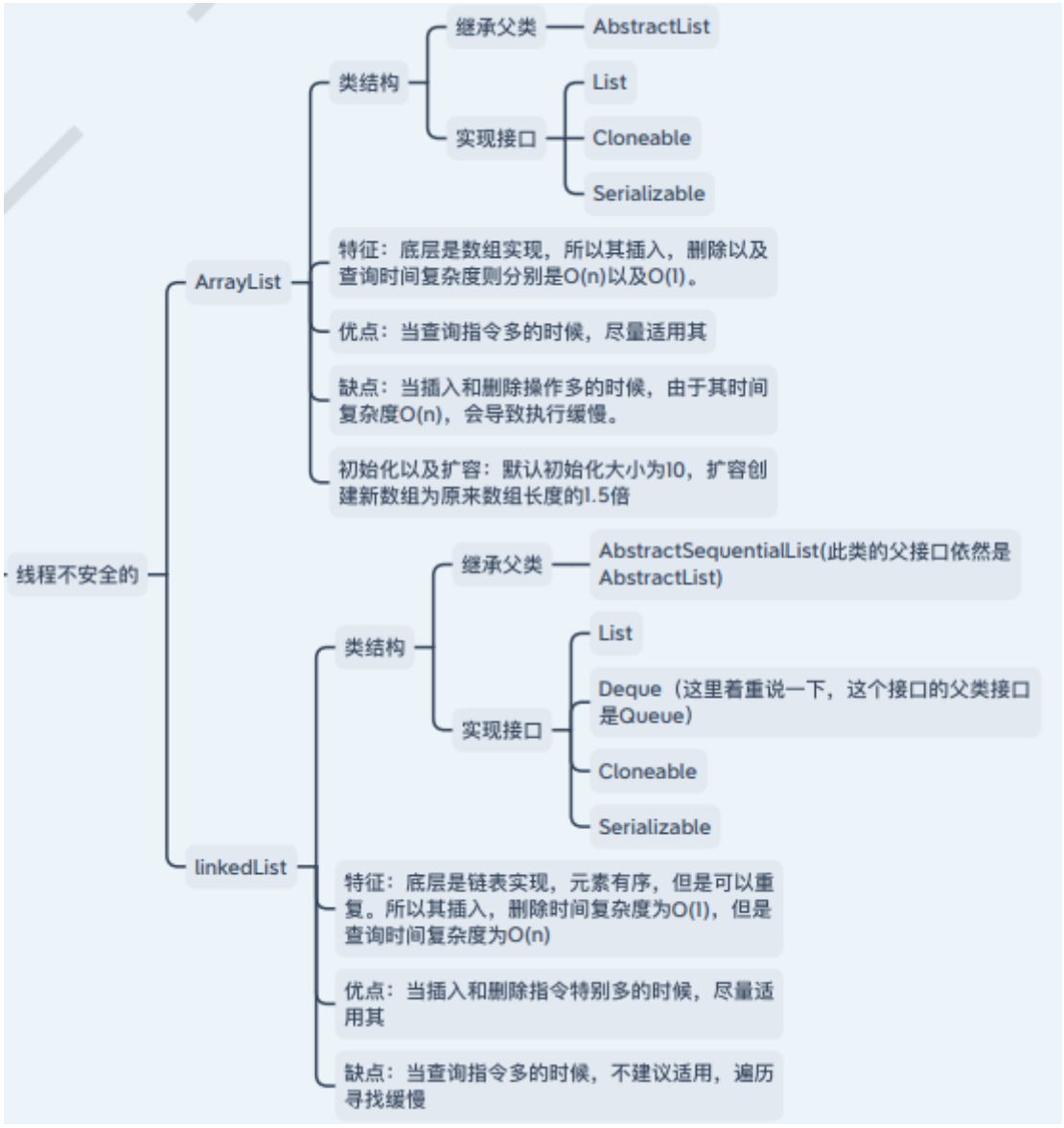
集合

Collection

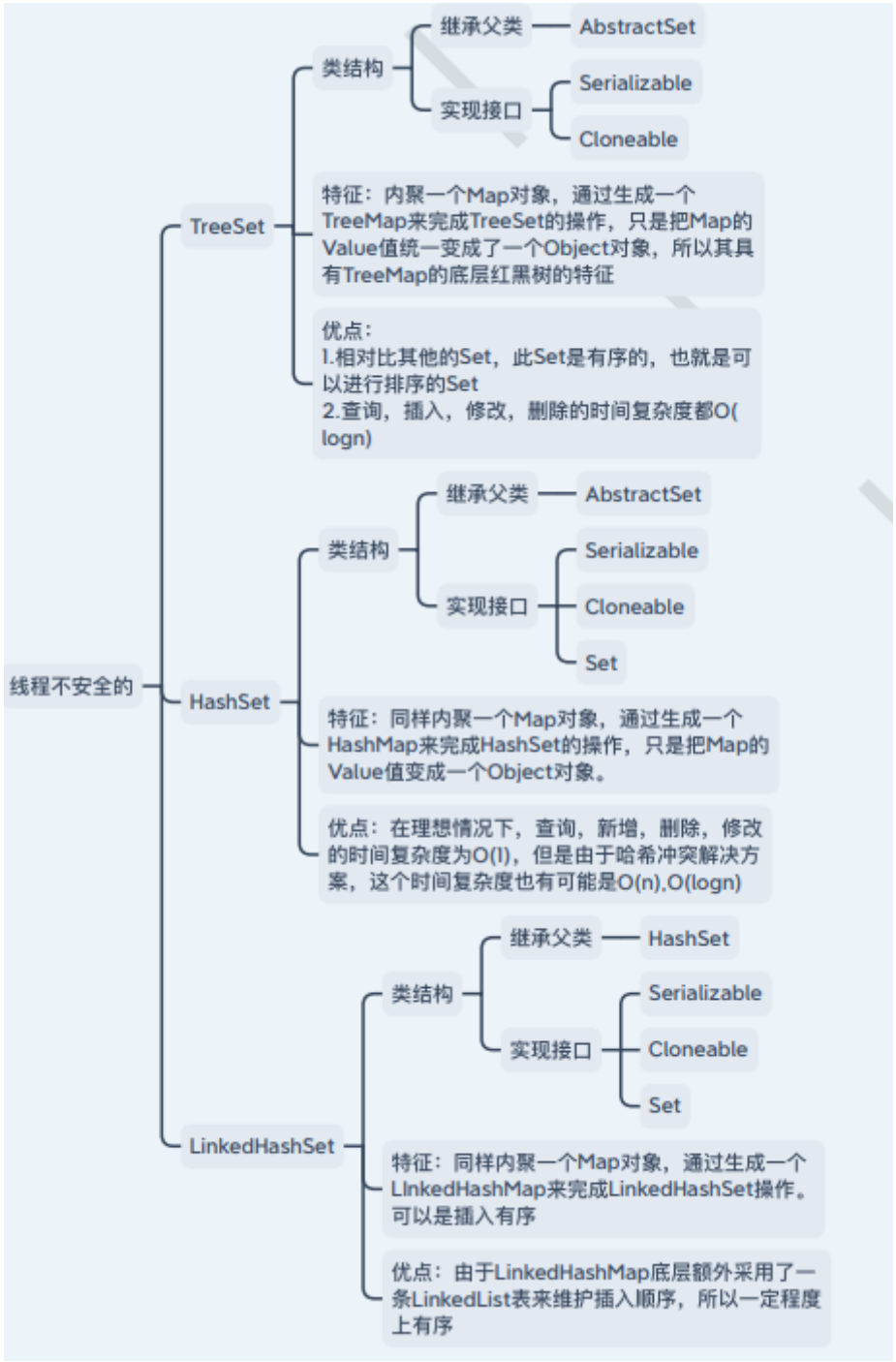
- ## List
  - 特点： 元素有序，但是可以重复。
  - 线程安全的：Vector、CopyAndWriterArrayList、Collections.synchronziedList(List list)
  - 

  - 线程不安全的：ArrayList、LinkedList



- **Set**
  - 特点 元素不能重复，而且无序
  - 线程安全的：CopyAndWriterArraySet、Collections.synchronziedSet(Set set)

  - 线程不安全的：TreeSet、HashSet、LinkedHashSet

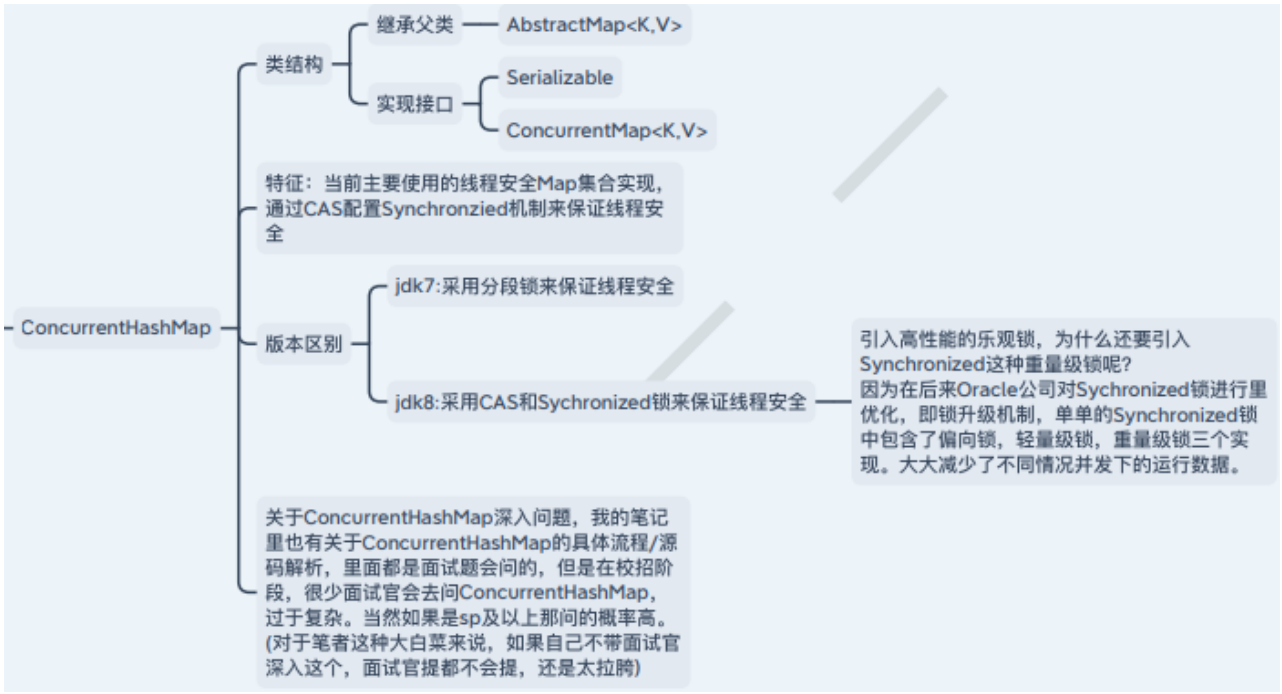
- 

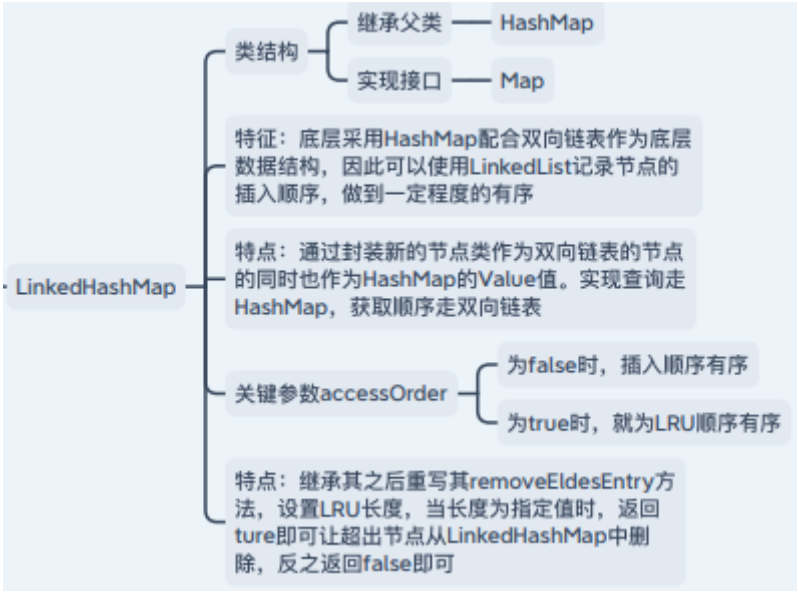
- Queue

Map

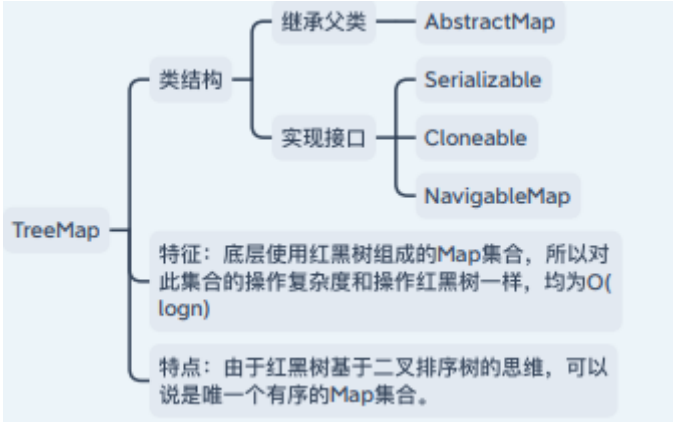
- 线程安全的：Hashtable、Collections.synchronizedMap(new HashMap<>())、ConcurrentHashMap
- 

- 

- 

- 线程不安全的：HashMap、LinkedHashMap、TreeMap
- 

-



- HashMap
  - 类结构：继承父类 AbstractMap ， 实现接口 Serializable Cloneable Map
  - 特征：底层采用哈希表配合链表，红黑树组成。 而且容量必须为2的次方数，会通过多次右移运 算符算法完成输入容量向最近的上界2的次方数 转化。初始容量为16
  - 扩容因子0.75：
    - ```
      关于扩容因子为什么规定为0.75，这其实是一个数学分析问题，也就是泊松分布

      只有当扩容因子为0.75的时候，才能保证在正常情况下桶中元素个数不会超过8个，当然极端情况是会超过8个的，所以jdk8采用红黑树进行了优化。扩容因子可以说是内存空间利用率和哈希冲突率之间的权衡，如果我们把扩容因子的值设置的很小，那么在哈希表中所含元素很少的时候，哈希表就会进行扩容，翻倍成原来的2倍长度，这样降低哈希冲突概率，但是内存空间使用率低了很多。

      反之如果扩容因子设置的很高，那么虽然空间使用率上来，但是哈希冲突率直接上升。所以太高太低是不行的，最后经过大量测试规定在了0.75

      ```
  - 容量必须为2的次方数
    - 计算桶的位置
    - ```
      计算桶的位置

      正常在哈希表理论计算的时候，我们计算桶位置首先会通过hashCode方法计算出Hash值，然后通过此Hash值与数组长度-1进行求摸运算，然后得到相应的桶位置。但是在HashMap中首先优化了求Hash值的方法，虽然还是会使用HashCode获取到哈希值，但是会通过Hash >>> 16（无符号右移）的方式，让高16位到达低16位，为什么这样做呢？主要是给我们下一步优化做铺垫，也就是关键的用&替换掉求摸运算，即hash & (length -1)；只有在哈希数组长度在2的次方数时，才能完全替换掉公式。那么为什么Hash >>> 16 是给Hash &(length - 1)做铺垫呢？ 因为如果换成&运算，那么比较的就是二进制码，当数组长度比较小的时候，高位16位均为0，低位才有变化。所以为了降低哈希冲突率，使用Hash >>> 16吧hash值的高位移动到低位
      ```
    - 扩容优化：数组翻倍和数据迁移
    - 数组翻倍：由于HashMap是线程不安全的，不需要考虑多个 线程同时参与扩容时，生成多个新数组的情况， 所以直接翻倍后引用即可。
    - 数据迁移：

- 迁移的核心算法就是：
  hash & length 的值，如果结果为1，那么就需要重新计算其Hash值对应位置，但是如果结果为0，那么其位置不用动即可。那么这个结果为什么可以做这个判断呢？这其实和length-1，length两个值的二进制码有关，我们举个例子，16的二进制码为10000，那么length-1也就是15的二进制码为1111，那么翻倍之后32的二进制码为100000，31的二进制码为11111，那么两者之间和Hash进行&运算的差别就在于，Hash值第五位二进制码到底是1还是0，如果是0的话，那么和15，31的计算结果不变，如果为1的话，那就会改变结果。而hash & length 就可以知道Hash值的第5位到底是0还是1。

- put方法具体流程:
  - 1.首先会去检测哈希数组是否初始化，没有就初 始化
  - 2.计算key值所在桶位置
  - 3.检查当前桶位置是否有元素，没有元素直接插 入
  - 4.当桶位置有元素时，判断是链表节点还是红黑树节点
  - 5.如果时链表节点，则进行遍历查询插入位置， 如果有节点和key相同，则替换value，并返回替 换的value。反之直接插入到末尾
  - 6.如果是红黑树节点，那么窒息红黑树插入流程
- 红黑树插入流程