hashmap 继承自Object类

Class HashMap<K,V> K是keys，V指的是映射的值

所有已实现的接口有

Serializable接口可序列化， Cloneable借口可克隆

直接已知子类有

LinkedHashMap，PrinterStateReasons

公共类HashMap <K，V>

扩展AbstractMap <K，V>

实现Map <K，V>，Cloneable，Serializable

基于哈希表的Map接口实现。此实现提供所有可选的映射操作，并允许 空值和空键。（HashMap 类大致相当于Hashtable，除了它是不同步的并且允许空值。）这个类不保证Map的顺序; 特别是，它不保证订单会随着时间的推移保持不变。

对hashmap简单了解，和方法的使用，搜索相关资料，查询1.8中的优化

接口java.util.Map，此接口主要有四个常用的实现类，分别是：

HashMap：继承abstractMap,底层实现数组+链表+红黑树（JDK1.8增加了红黑树部分），线程不安全，kv结构，可以用 Collections 的 synchronizedMap 方法使HashMap具有线程安全的能力

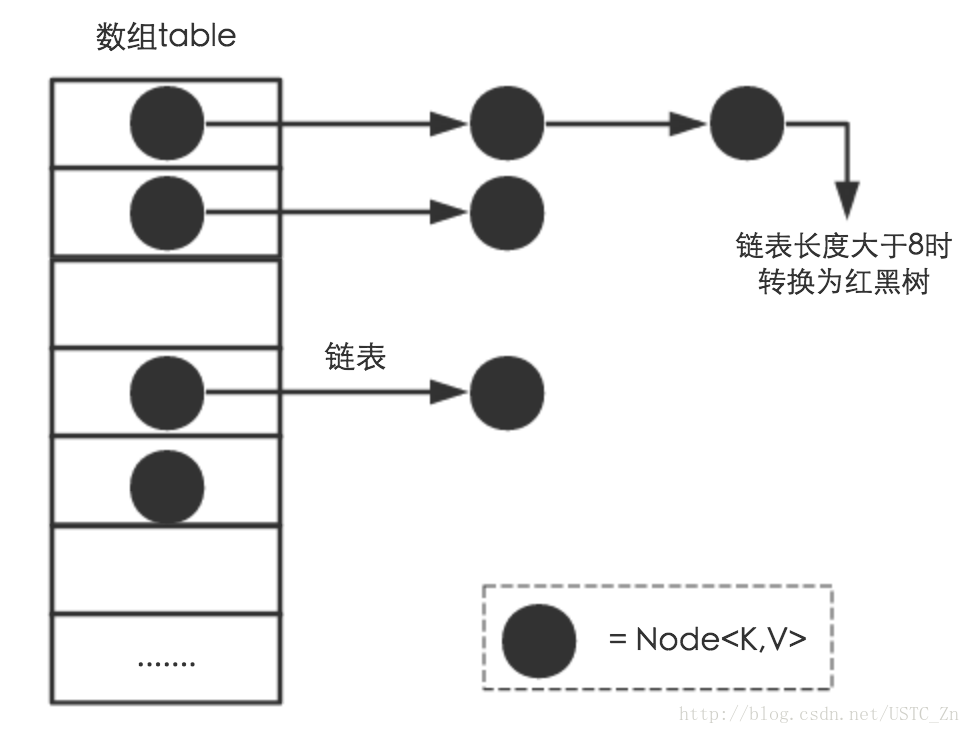
Hashtable：继承Dictionary类，是遗留类，很多映射的常用功能与HashMap类似，线程安全，但是不建议在新代码使用，不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。

ConcurrentHashMap：Hashtable并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁

LinkedHashMap：是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

TreeMap：继承abstractMap,实现SortedMap接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。如果使用排序的映射，建议使用TreeMap。在使用TreeMap时，key必须实现Comparable接口或者在构造TreeMap传入自定义的Comparator，否则会在运行时抛出java.lang.ClassCastException类型的异常。

从结构实现来讲，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，如下如所示。



数据底层具体存储的是什么？这样的存储方式有什么优点呢？

(1) 从源码可知，HashMap类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个Node的数组。

static class Node<K,V> implements Map.Entry<K,V> {

final int hash; *//用来定位数组索引位置*

final K key;

V value;

Node<K,V> next; *//链表的下一个node*

Node(int hash, K key, V value, Node<K,V> next) { ... }

public final K getKey(){ ... }

public final V getValue() { ... }

public final String toString() { ... }

public final int hashCode() { ... }

public final V setValue(V newValue) { ... }

public final boolean equals(Object o) { ... }

}

Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。上图中的每个黑色圆点就是一个Node对象。

1. HashMap就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java中HashMap采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。在空间成本和时间成本之间权衡，其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的hash算法减少Hash碰撞。那么通过什么方式来控制map使得Hash碰撞的概率又小，哈希桶数组（Node[] table）占用空间又少呢？答案就是好的Hash算法和扩容机制。

int threshold; *// 所能容纳的key-value对极限*

final float loadFactor; *// 负载因子*

int modCount;

int size;

首先，Node[] table的初始化长度length(默认值是16)，Load factor为负载因子(默认值是0.75)，threshold是HashMap所能容纳的最大数据量的Node(键值对)个数。threshold = length \* Load factor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。

在HashMap中，哈希桶数组table的长度length大小必须为2的n次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。HashMap采用这种非常规设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap定位哈希桶索引位置时，也加入了高位参与运算的过程。

这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。<https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.01.md> 了解红黑树

Jdk1.8最大的区别就在于底层的红黑树策略，采用位桶数组

美团面试题

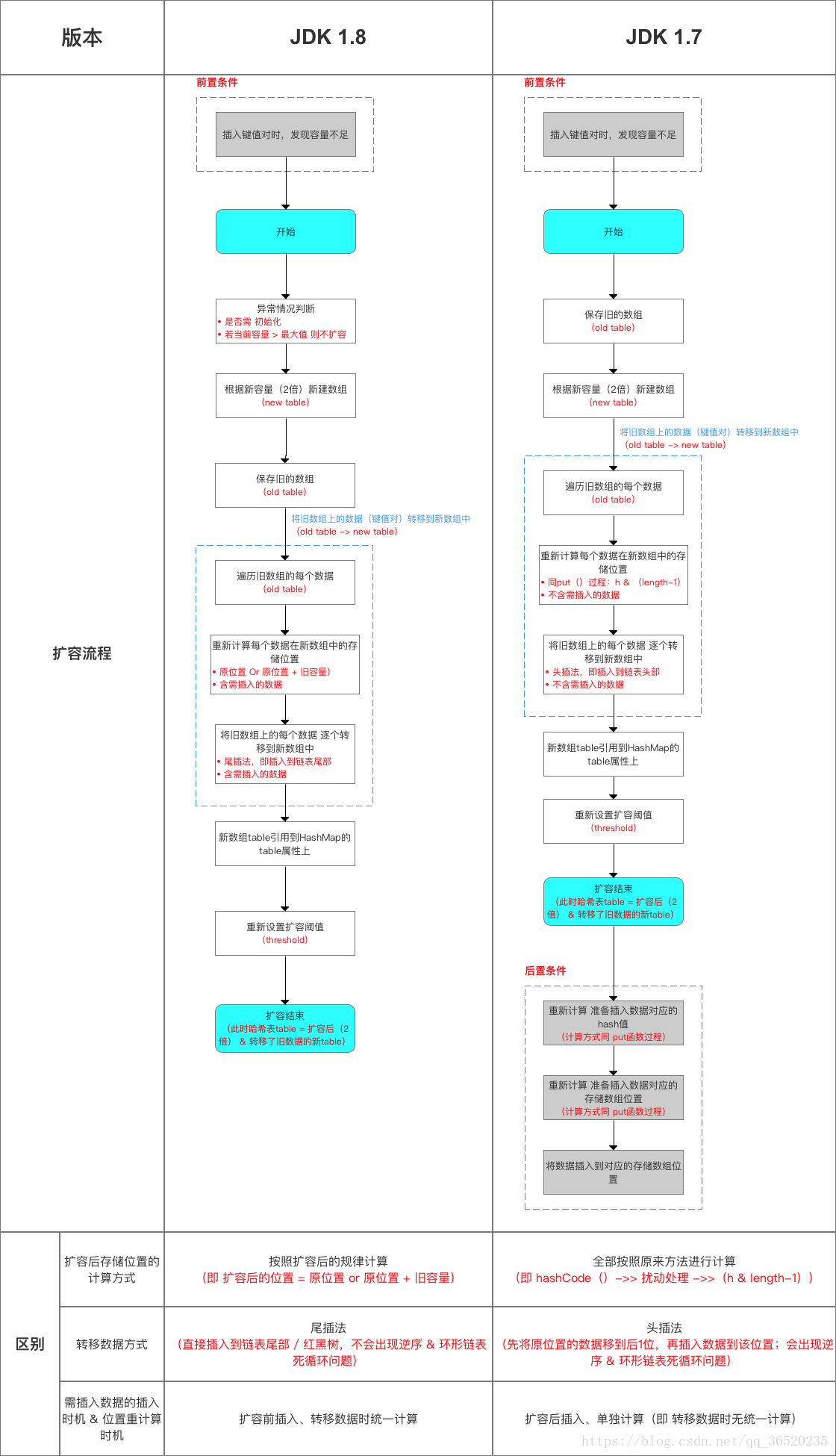
不同点

（1）JDK1.7用的是头插法，而JDK1.8及之后使用的都是尾插法，那么他们为什么要这样做呢？因为JDK1.7是用单链表进行的纵向延伸，当采用头插法时会容易出现逆序且环形链表死循环问题。但是在JDK1.8之后是因为加入了红黑树使用尾插法，能够避免出现逆序且链表死循环的问题。

（2）扩容后数据存储位置的计算方式也不一样：1. 在JDK1.7的时候是直接用hash值和需要扩容的二进制数进行&（这里就是为什么扩容的时候为啥一定必须是2的多少次幂的原因所在，因为如果只有2的n次幂的情况时最后一位二进制数才一定是1，这样能最大程度减少hash碰撞）（hash值 & length-1）

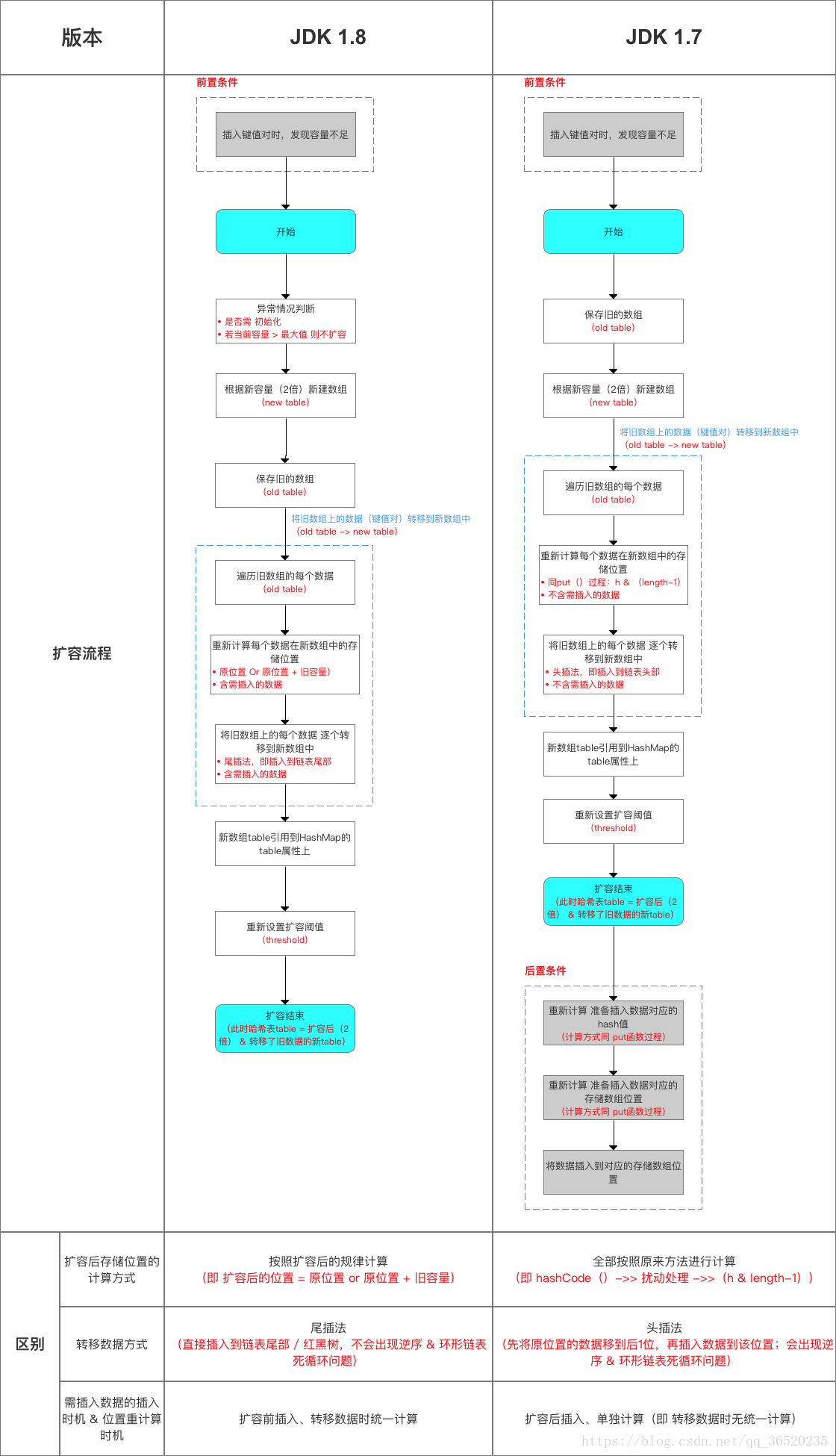
2、而在JDK1.8的时候直接用了JDK1.7的时候计算的规律，也就是扩容前的原始位置+扩容的大小值=JDK1.8的计算方式，而不再是JDK1.7的那种异或的方法。但是这种方式就相当于只需要判断Hash值的新增参与运算的位是0还是1就直接迅速计算出了扩容后的储存方式。

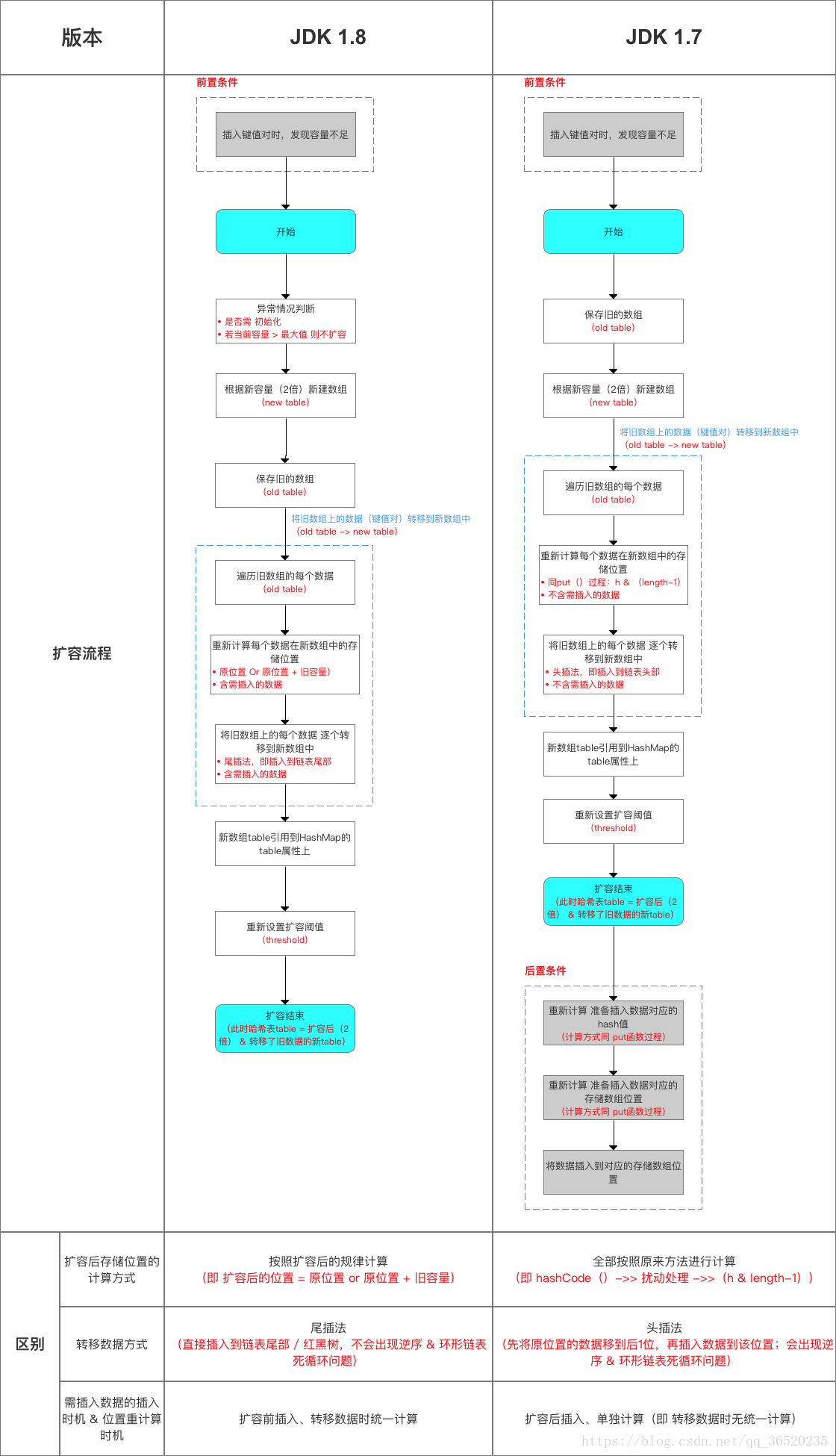
在计算hash值的时候，JDK1.7用了9次扰动处理=4次位运算+5次异或，而JDK1.8只用了2次扰动处理=1次位运算+1次异或。

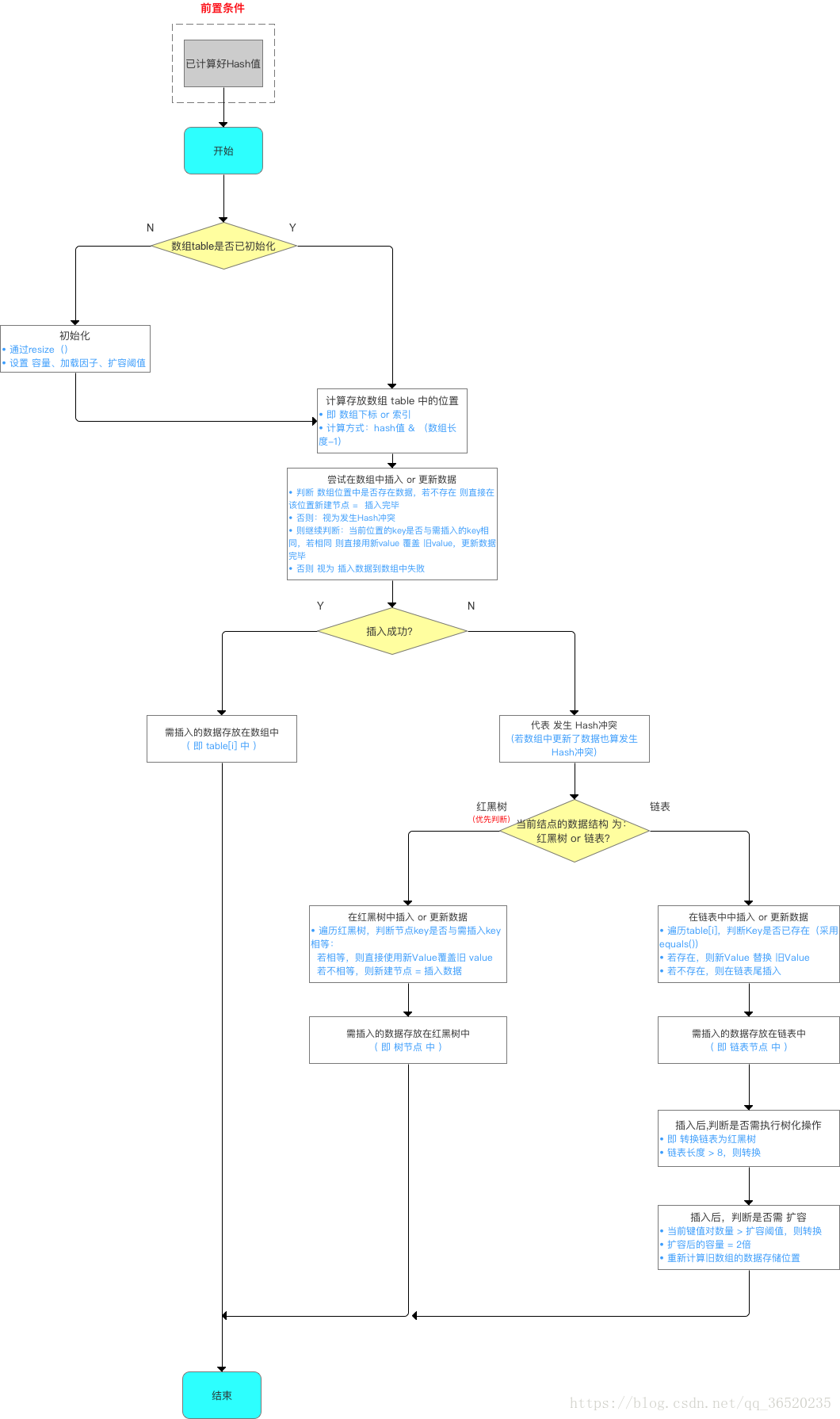


1. JDK1.7的时候使用的是数组+ 单链表的数据结构。但是在JDK1.8及之后时，使用的是数组+链表+红黑树的数据结构（当链表的深度达到8的时候，也就是默认阈值，就会自动扩容把链表转成红黑树的数据结构来把时间复杂度从O（n）变成O（logN）提高了效率）

来自<https://blog.csdn.net/qq_36520235/article/details/82417949>







ArrayList

 ArrayList是Java容器中相对比较简单的一个，它底层是采用数组实现的，包括的主要操作有remove 、set、contains、clear、add、clone、indexOf、toString等等常用的方法

简单总结下，jdk1.7版本对比1.6版本，三个改动：

1、懒初始化，因此默认构造方法创建的是空数组，不再是10个大小的数组；

2、扩容相关的完善；

3、一些方法不再直接使用父类的通用实现，改为利用数组特性的更有效率的实现。

一、基本性质

1、底层使用原生数组实现，实现RandomAccess接口，可以随机访问，随机访问指的是下标索引操作index(i)的时间复杂度是O(1)。

size、isEmpty、get、set、iterator和listIterator操作在O(1)内完成，add(e)操作平均在O(1)内完成，即添加n个元素需要O(n)时间（这个是Collection.add，是在尾部添加注意区分下List.add(index, e)）。其他操作基本都是O(n)内完成。ArrayList与LinkedList实现相比，O(n)的各个方法的时间复杂度的常数因子更小。

2、因为底层数组 elementData 的容量是不能改变的，所以容量不够时，需要把 elementData 换成一个更大的数组，这个过程叫作扩容。实际的元素的数量size，总是不会超过底层数组的容量 elementData.length，因为扩容需要申请更大的内存，并且需要原来数组的进行一次复制，所以扩容是个耗时的操作。在添加大量元素之前，使用者最好是预估一个大致的数量，手动调用ensureCapacity进行一次扩容操作，避免一个个添加导致频繁扩容影响性能。

3、ArrayList是未同步的，多线程并发读写时需要外部同步，如果不外部同步，那么可以使用Collections.synchronizedList方法对ArrayList的实例进行一次封装，或者使用Vector。

4、对存储的元素无限制，允许null元素。

5、ArrayList的iterator和listIterator方法返回的迭代器是快速失败的，也就是如果在创建迭代器之后的任何时间被结构性修改，除了通过迭代器自己的remove或add方法之外，迭代器将直接抛出一个ConcurrentModificationException，从而达到快速失败fail-fast的目的，尽量避免不确定的行为。

ArrayList的迭代器的快速失败行为不能被严格保证，并发修改时它会尽量但不100%保证抛出ConcurrentModificationException。因此，依赖于此异常的代码的正确性是没有保障的，迭代器的快速失败行为应该仅用于检测bug。

6、实现clone接口，可以调用其clone方法（虽然clone()是Object中的方法，但是它是protected，使用子类的clone()必须在子类中覆盖此方法）。clone方法复制一个ArrayList，底层数组elementData不共享，但是实际的元素还是共享的。

不过clone是ArrayList中覆盖的，不属于List中的方法，因此常见的声明形式

     List<String> strs = new ArrayList<>()；

声明出来的变量不能直接使用clone方法，本身也用得极少。

7、实现Serializable接口，可以被序列化。ArrayList"实现"了自定义序列化方法，这么做主要是为了节省空间 。对于占用空间的大头——元素list，仅仅序列化实际size大小的元素，同时不序列化对于新对象无用属性的——来自父类AbstractList的modCount。ArrayList的实际size不会超过底层数组的length，大多数情况下比底层数组length小，使用默认序列化的话，会直接序列化整个底层数组，序列化后字节流会变大，浪费空间。

原文：https://blog.csdn.net/u011392897/article/details/57105709

二、构造方法

1、默认构造方法，ArrayList()

jdk1.7的，相对1.6版本，引入了一个新的常量EMPTY\_ELEMENTDATA，它是一个空数组，因此容量为0。

jdk1.8的，相对1.7版本，又引入了一个新的常量DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA ，它也是一个空数组，因此容量也为0。

从1.7开始，无参构造方法构造的ArrayList的底层数组elementData大小默认为0。

java集合类在jdk1.7版本基本上都有一种改动：懒初始化。懒初始化指的是默认构造方法构造的集合类，占据尽可能少的内存空间（对于ArrayList来说，使用空数组来占据尽量少的空间，不使用null是为了避免null判断），在第一次进行包含有添加语义的操作时，才进行真正的初始化工作。

1.7开始的ArrayList，默认构造方法构造的实例，底层数组是空数组，容量为0，在进行第一次add/addAll等操作时才会真正给底层数组赋非empty的值。如果add/addAll添加的元素小于10，则把elementData数组扩容为10个元素大小，否则使用刚好合适的大小（例如，第一次addAll添加6个，那么扩容为10个，第一次添加大于10个的，比如24个，扩容为24个，刚好合适）；1.8版本，默认构造的实例这个行为没有改变，只是用的数组名字变了。

1. 带初始容量的构造方法，public ArrayList(int initialCapacity)

678三个版本的这个构造方法的实际行为基本一致：如果initialCapacity >= 0，把底层数组elementData赋值为一个大小为initialCapacity的数组，数组的所有元素都是默认值null。1.8稍微进行了一点优化，也是赋值为空数组，但是重用了常量对象。

jdk1.6中new ArrayList<?>()跟new ArrayList<?>(10)的行为是一模一样的，所以跟new ArrayList<?>(0)有很明显区别，这个好理解 。从1.7版本开始，new ArrayList<>()和new ArrayList<>(0)，虽然创建后底层内容和容量都一样，但是实际的行为有些细小的差别，那就是这两个在第一次自动扩容时策略不一样。不过这一点影响比较小，基本不影响使用。

1.8中使用两个空数组，正如注释所说的，是在优化（避免创建无用的空数组）的同时，保留其扩容初始策略区别。只用一个空数组就不能再优化的同时，继续保持这个小区别了。

扩容方法四个位置用到：两个add方法，两个addAll方法，一个反序列化方法，还有就是手动扩容方法ensureCapacity（称之为手动，是因为此方法是public的，可以外部手动调用）。在1.6版本是只有这个手动的方法，内部自动操作也是调用这个方法，1.7开始进行了区分，并且进一步改进了扩容操作。

1.7的和1.8的基本相同，唯一的一点区别就是1.8用两个空数组，导致这里的空数组的名字不一样，两个版本的代码可以看作是一样的。