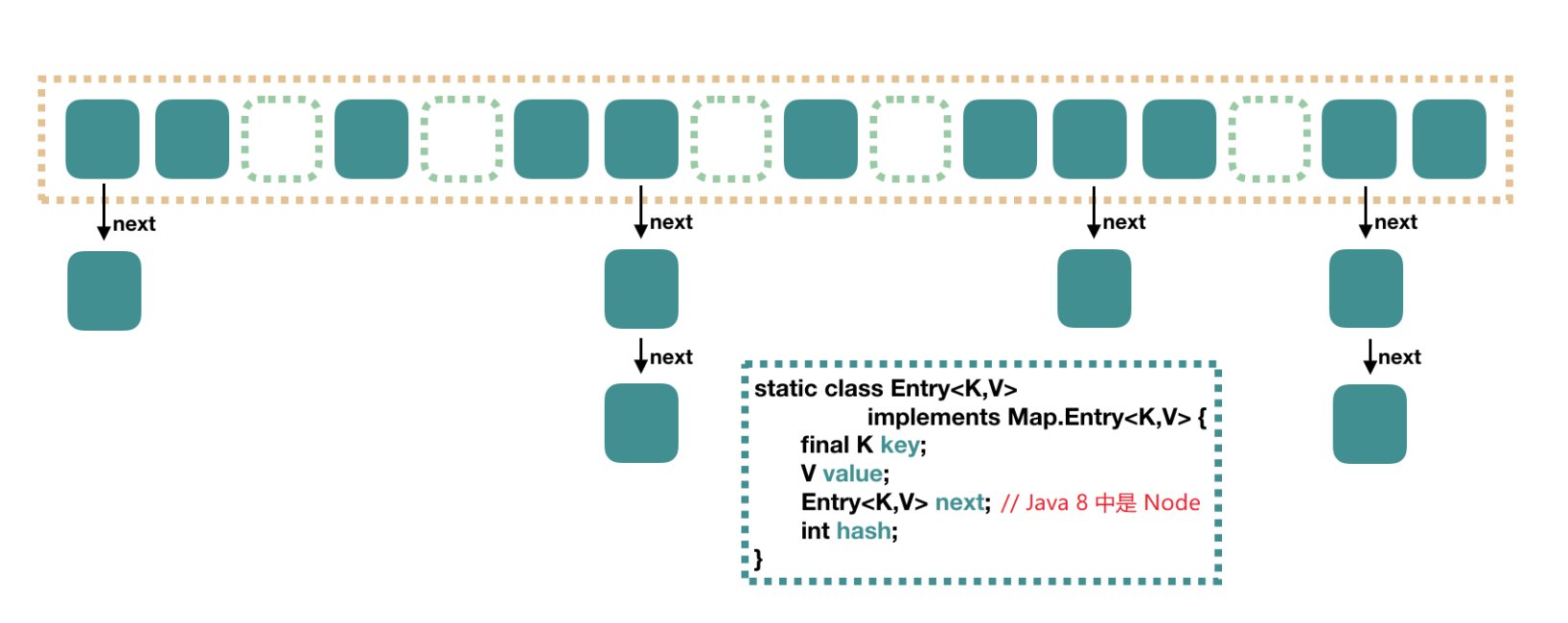
1. 数据结构



1. 构造函数

HashMap()：构造一个初始容量为16并且加载因子为0.75的容器。

HashMap(int initialCapacity)：构造一个初始容量为给定数值并且加载因子为0.75的容器。

HashMap(int initialCapacity, float loadFactor)：构造一个初始容量、加载因子为给定数值的容器。

HashMap(Map<? extends K, ? extends V> m)：构造一个与指定 Map 具有相同映射的新 HashMap。 HashMap 是使用默认加载因子 (0.75) 和足以容纳指定 Map 中的映射的初始容量创建的。

HashMap(Map<? extends K, ? extends V> m) 流程：

1. 加载因子默认为0.75；
2. putMapEntries(Map<? extends K, ? extends V> m, boolean evict)；
3. m：目标Map；
4. evict：最初构造此映射时为 false，否则为 true（中继到方法 afterNodeInsertion）。

初始化table数据结构，并调整大小。table是一个Node<K,V>[]数组，长度是2的次幂。Node是实现Node<K,V> implements Map.Entry<K,V>接口的类：

interface Entry<K, V> {

K getKey();

V getValue();

V setValue(V value);

boolean equals(Object o);

int hashCode();

<K extends Comparable<? super K>, V> Comparator<Map.Entry<K, V>> comparingByKey();

<K, V extends Comparable<? super V>> Comparator<Map.Entry<K, V>> comparingByValue();

<K, V> Comparator<Map.Entry<K, V>> comparingByKey(Comparator<? super K> cmp);

<K, V> Comparator<Map.Entry<K, V>> comparingByValue(Comparator<? super V> cmp)

}

Node是一个基本的hash节点：

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

final int hash;

final K key;

V value;

Node<K,V> next;

}

每个Node节点包括一个hash值和对应节点的key、value，并且是一个链表。

描述完对应的数据结构后，看看putMapEntries方法怎么填充Map的。首先获取传入Map的size，然后除以加载因子再加1，如果小于MAXIMUM\_CAPACITY（MAXIMUM\_CAPACITY = 1 << 30）表示可用，大于MAXIMUM\_CAPACITY则返回MAXIMUM\_CAPACITY。如果得出的容量大于threshold（容量和加载因子的乘积capacity \* load factor）。通过给定Map和加载因子得出的大小还需要进一步处理（-1 >>> Integer.numberOfLeadingZeros(cap - 1)），转换成2的幂次并且不超过MAXIMUM\_CAPACITY。如果使用的是已经有初始容量的HashMap，那么table不为空，此时如果给定的Map的size大于threshold，则调用resize()，初始化或加倍表大小。如果为空，则按照字段阈值中保存的初始容量目标进行分配。否则，因为我们使用的是二次幂扩展，每个 bin 中的元素必须保持相同的索引，或者在新表中以二次幂的偏移量移动（容量是 2 的幂时，key 的 hash 值然后 & (容量-1) 确定位置时碰撞概率会比较低，因为容量为 2 的幂时，减 1 之后的二进制数为全 1，这样与运算的结果就等于 hash值后面与 1 进行& 运算的几位。）。

然后调用putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict)将key和value存放进处理过后的table中。如果经过计算过后的位置上的数组为空，直接放进去元素。如果不为空，计算hash值、key值和equals方法（比较对应的key和value值是否相同），如果都相同直接覆盖。如果节点链表中数量超过TREEIFY\_THRESHOLD（8）个，转化为红黑树（最多64个元素）。否则就是hash值相同，但是key不同，此时使用尾插法插入最后。

最后修改modCount，它表示此 HashMap 已在结构上修改的次数 结构修改是指更改 HashMap 中的映射数量或以其他方式修改其内部结构（例如，重新散列）的那些。该字段用于使 HashMap 的 Collection-views 上的迭代器快速失败。

1. Put函数

V put(K key, V value)：将指定的值与此映射中的指定键相关联。如果映射先前包含键的映射，则替换旧值。

V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict)：实现 Map.put 和相关方法。

1、计算 key 的 hash 值。

1.计算方式是 (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

2、检查当前数组是否为空，为空需要进行初始化，初始化容量是 16 ，负载因子默认 0.75。

3、计算 key 在数组中的坐标。

1.计算方式：(容量 - 1) & hash.

2.因为容量总是 2 的次方，所以 - 1 的值的二进制总是全 1。方便与 hash 值进行 与运算。

4、如果计算出的坐标元素为空，创建节点加入，put 结束。

1.如果当前数组容量大于负载因子设置的容量，进行扩容。

5、如果计算出的坐标元素有值。

1.如果坐标上的元素值和要加入的值 key 完全一样，覆盖原有值。

2.如果坐标上的元素是红黑树，把要加入的值和 key 加入到红黑树。

3.如果坐标上的元素和要加入的元素不同（尾插法增加）。

1.如果 next 节点为空，把要加入的值和 key 加入 next 节点。

2.如果 next 节点不为空，循环查看 next 节点。

1.如果发现有 next 节点的 key 和要加入的 key 一样，对应的值替换为新值。

3.如果循环 next 节点查找超过 8 层还不为空，把这个位置元素转换为红黑树。

1. hash计算方式

key为null，放在0下标。key不为null，计算哈希值。

(key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16)

计算 hash 值 int hash = key.hashCode ()，与或上 hash 值无符号右移 16 位。

位置计算公式 index = (n - 1) & hash ，其中 n 是容量。

1. Get函数

1、计算 key 的 hash 值。

2、如果存储数组不为空，且计算得到的位置上的元素不为空。继续，否则，返回 Null。

3、如果获取到的元素的 key 值相等，说明查找到了，返回元素。

4、如果获取到的元素的 key 值不相等，查找 next 节点的元素。

1.如果元素是红黑树，在红黑树中查找。

2.不是红黑树，遍历 next 节点查找，找到则返回。

1. 快速失败

HashMap 遍历使用的是一种快速失败机制，它是 Java 非安全集合中的一种普遍机制，这种机制可以让集合在遍历时，如果有线程对集合进行了修改、删除、增加操作，会触发并发修改异常。

它的实现机制是在遍历前保存一份 modCount ，在每次获取下一个要遍历的元素时会对比当前的 modCount 和保存的 modCount 是否相等。

快速失败也可以看作是一种安全机制，这样在多线程操作不安全的集合时，由于快速失败的机制，会抛出异常。