ArrayMap是Android专门针对内存优化而设计的，用于取代Java API中的HashMap数据结构。为了更进一步优化key是int类型的Map，Android再次提供效率更高的数据结构SparseArray，可避免自动装箱过程。对于key为其他类型则可使用ArrayMap。HashMap的查找和插入时间复杂度为O(1)的代价是牺牲大量的内存来实现的，而SparseArray和ArrayMap性能略逊于HashMap，但更节省内存。

public final class ArrayMap<K, V> implements Map<K, V> {

private static final boolean CONCURRENT\_MODIFICATION\_EXCEPTIONS = true;

private static final int BASE\_SIZE = 4; // 容量增量的最小值

private static final int CACHE\_SIZE = 10; // 缓存数组的上限

static Object[] mBaseCache; //用于缓存大小为4的ArrayMap

static int mBaseCacheSize;

static Object[] mTwiceBaseCache; //用于缓存大小为8的ArrayMap

static int mTwiceBaseCacheSize;

final boolean mIdentityHashCode;

int[] mHashes; //由key的hashcode所组成的数组

Object[] mArray; //由key-value对所组成的数组，是mHashes大小的2倍

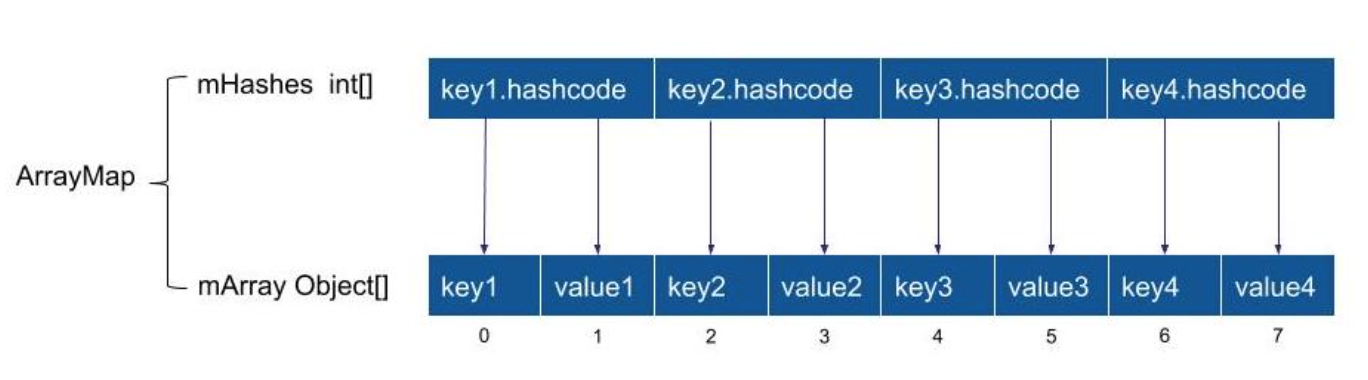
int mSize; //成员变量的个数

}

ArrayMap对象的数据储存格式如图所示：

mHashes是一个记录所有key的hashcode值组成的数组，是从小到大的排序方式；

mArray是一个记录着key-value键值对所组成的数组，是mHashes大小的2倍；



其中mSize记录着该ArrayMap对象中有多少对数据，执行put()或者append()操作，则mSize会加1，执行remove()，则mSize会减1。mSize往往小于mHashes.length，如果mSize大于或等于mHashes.length，则说明mHashes和mArray需要扩容。

ArrayMap类有两个非常重要的静态成员变量mBaseCache和mTwiceBaseCacheSize，用于ArrayMap所在进程的全局缓存功能：

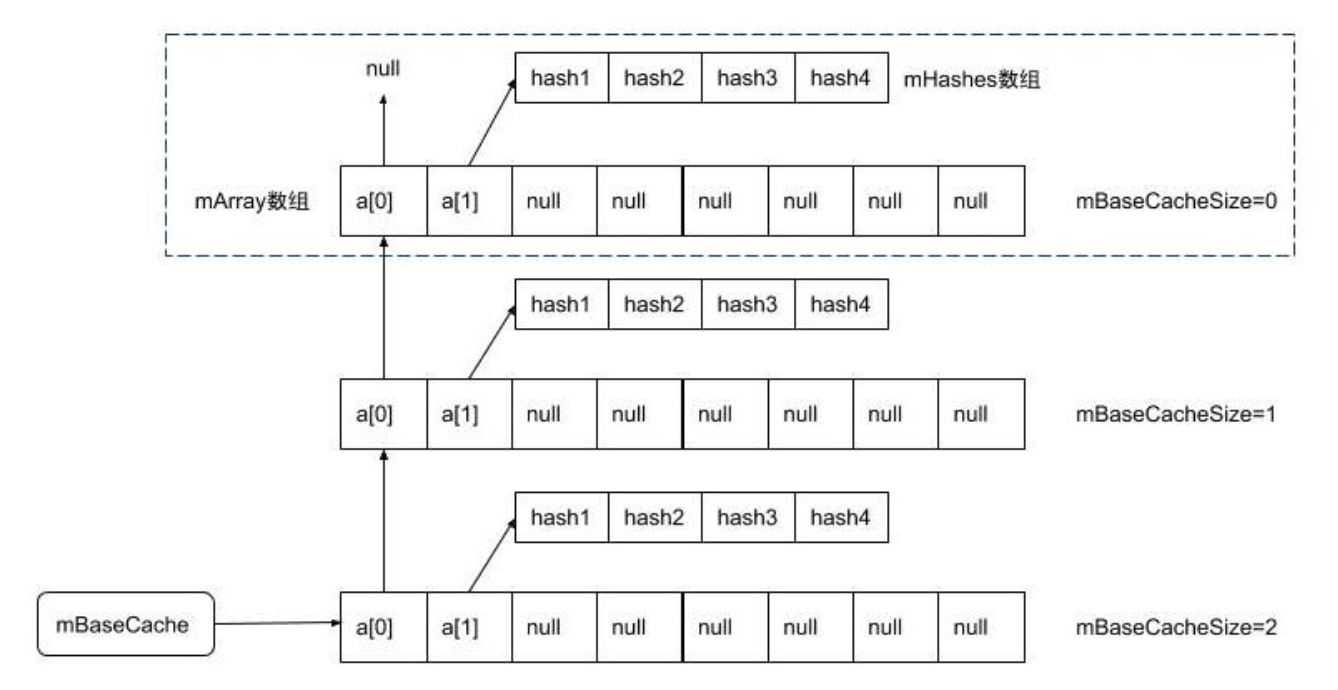
mBaseCache：用于缓存大小为4的ArrayMap，mBaseCacheSize记录着当前已缓存的数量，超过10个则不再缓存；

mTwiceBaseCacheSize：用于缓存大小为8的ArrayMap，mTwiceBaseCacheSize记录着

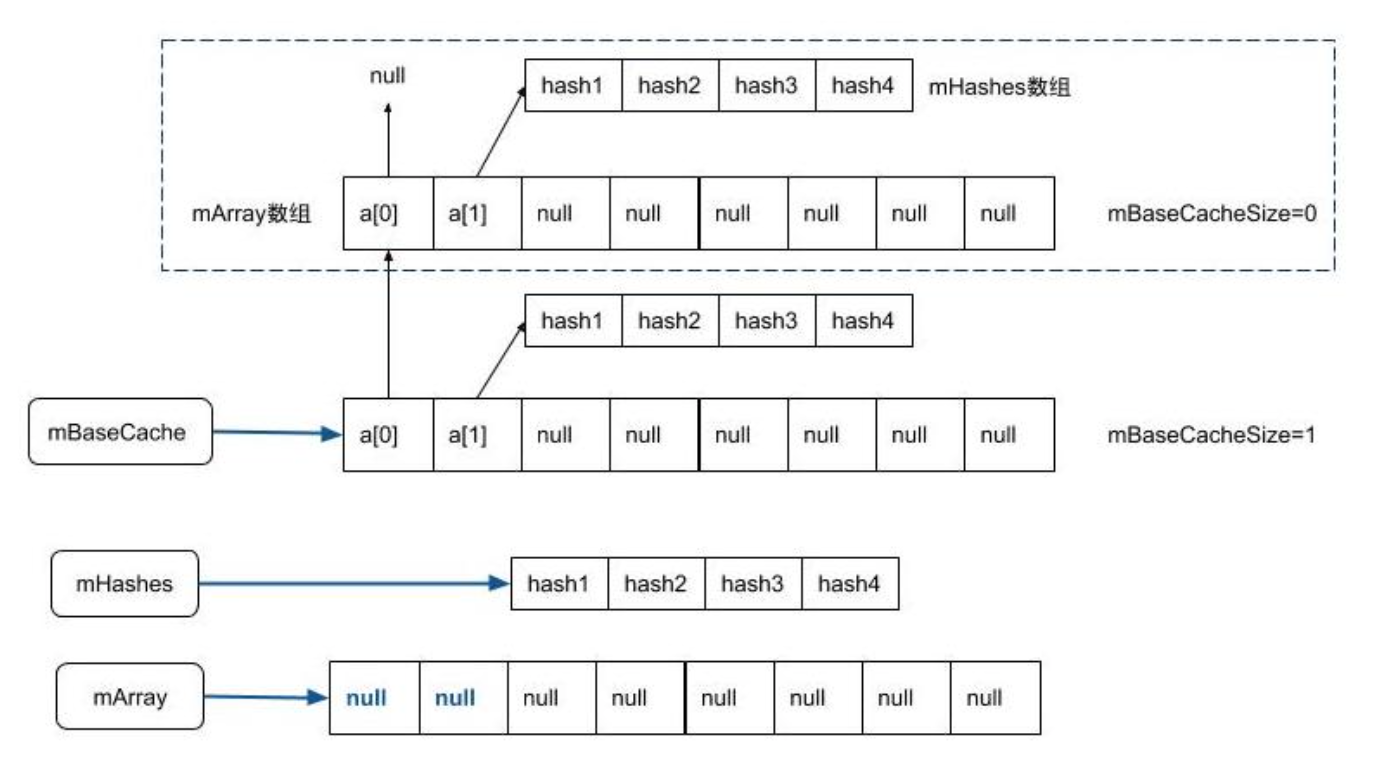
当前已缓存的数量，超过10个则不再缓存。

为了减少频繁地创建和回收Map对象，ArrayMap采用了两个大小为10的缓存队列来分别保存大小为4和8的Map对象。

最初mTwiceBaseCache和mBaseCache缓存池中都没有数据，在freeArrays释放内存时，如果同时满足释放的array大小等于4或者8，且相对应的缓冲池个数未达上限，则会把该arrya加入到缓存池中。加入的方式是将数组array的第0个元素指向原有的缓存池，第1个元素指向hashes数组的地址，第2个元素以后的数据全部置为null。再把缓存池的头部指向最新的array的位置，并将该缓存池大小执行加1操作。具体如下所示。



当allocArrays分配内存时，如果所需要分配的大小等于4或者8，且相对应的缓冲池不为空，则会从相应缓存池中取出缓存的mArray和mHashes。从缓存池取出缓存的方式是将当前缓存池赋值给mArray，将缓存池指向上一条缓存地址，将缓存池的第1个元素赋值为mHashes，再把mArray的第0和第1个位置的数据置为null，并将该缓存池大小执行减1操作，具体如下所示。



这里需要注意的是只有大小为4或者8的内存分配才有可能从缓存池取数据，因为freeArrays过程放入缓存池的大小只有4或8，对于其他大小的内存分配则需要创建新的数组。 优化小技巧，对于分配数据不超过8的对象的情况下，一定要创建4或者8大小，否则浪费了缓存机制。比如ArrayMap[7]就是不友好的写法，建议写成ArrayMap[8]。

当mSize大于或等于mHashes数组长度时则扩容，完成扩容后需要将老的数组拷贝到新分配的数组，并释放老的内存。

当map个数满足条件 osize<4时，则扩容后的大小为4；

当map个数满足条件 4<= osize < 8时，则扩容后的大小为8；

当map个数满足条件 osize>=8时，则扩容后的大小为原来的1.5倍；

可见ArrayMap大小在不断增加的过程，size的取值一般情况依次会是4，8，12，18，27，40，60，…

当数组内存的大小大于8，且已存储数据的个数mSize小于数组空间大小的1/3的情况下，需要收紧数据的内容容量，分配新的数组，老的内存靠虚拟机自动回收。

如果mSize<=8，则设置新大小为8；

如果mSize> 8，则设置新大小为mSize的1.5倍。

也就是说在数据较大的情况下，当内存使用量不足1/3的情况下，内存数组会收紧50%。

put()设计巧妙地将修改已有数据对(key-value) 和插入新的数据对合二为一个方法，主要是依赖indexOf()过程中采用的二分查找法， 当找到相应key时则返回正值，但找不到key则返回负值，按位取反所对应的值代表的是需要插入的位置index。

put()在插入时，如果当前数组内容已填充满时，则会先进行扩容，再通过System.arraycopy来进行数据拷贝，最后在相应位置写入数据。

append()过程跟put()很相似，append的差异在于该方法不会去做扩容的操作，是一个轻量级的插入方法。 那么什么场景适合使用append()方法呢？答应就是对于明确知道肯定会插入队尾的情况下使用append()性能更好，因为put()上来先做binarySearchHashes()二分查找，时间复杂度为O(logN)，而append()的时间复杂度为O(1)。

remove()过程：通过二分查找key的index，再根据index来选择移除动作；当被移除的是ArrayMap的最后一个元素，则释放该内存，否则只做移除操作，这时会根据容量收紧原则来决定是否要收紧，当需要收紧时会创建一个更小内存的容量。

clear()清理操作会执行freeArrays()方法来回收内存，而类似的方法erase()则只会清空数组内的数据，并不会回收内存。