Android中的稀疏数组：SparseArray

30 Mar 2014

在一般情况下，使用`HashMap<K, V>`，如果`K`整数类型的话，使用`SparseArray`效率会更高。

和`HashMap`类似，`SparseArray`建立整数索引和对象的关系。和简单的对象数组相比，`SparseArray`允许索引之间有间隔。

`SparseArray`支持和`HashMap`类似的`put`和`get`方法。在其内部，维护着两个数组，一个用于存储索引，一个用于存储对象。

```
public class SparseArray<E> implements Cloneable {

    private int[] mKeys;
    private Object[] mValues;
    private int mSize;
```

整数索引被从小到大映射到`mKeys`数组中。

索引的映射

在计算整数索引映射到数组中的位置的时候，用了一个改造过的的二分搜索算法：

这个算法输入的参数是：要搜索的数组`a`，搜索的起始位置`start`，搜索的长度`len`，要检索的关键字`key`，如下：

```
private static int binarySearch(int[] a, int start, int len, int key) {
    int high = start + len, low = start - 1, guess;

    while (high - low > 1) {
        guess = (high + low) / 2;

        if (a[guess] < key)
            low = guess;
        else
            high = guess;
    }

    if (high == start + len)
        return ~(start + len);
```

```
    else if (a[high] == key)
        return high;
    else
        return ~high;
}
```

这是一个巧妙设计的算法，如果输入的`key`在区间内则返回等于关键字或者最小的大于关键字的索引。

如果关键字不在区间内，则将区间首个索引或者区间最后一个索引加1取反码，非负数的反码都是负数，因为符号位被取反了。

有一个数组：`int[] a = new int[] {2, 5, 8, 0, 0};`结构如下:

```
  0   1   2   3   4
+---+---+---+---+---+
| 2 | 5 | 8 |   |   |
+---+---+---+---+---+
```

对于`start = 0`, `len = 3`, 对于不同的关键字：

key	return	
1	-1	在区间范围最左边，返回区间最首个索引取反
2	0	关键字存在，返回对应的索引
4	1	返回最小的比关键字大的值的对应的索引, 即5的索引
5	1	关键字存在，返回对应的索引
9	-4	在区间最右边，区间最右索引加1取反`~(2+1)`

put过程

put的过程分为以下几步：

1. 计算索引映射。
2. 如果在在区间内有对应槽位，设置值，返回。
3. 如有必要，进行扩容。

容量以类似2的指数次幂增长。对象引用和和整数都占用4个字节，数组本身还需要占用3个字节。 为了内存4字节对齐，数组大小应该是：2^n - 3(n >=2)。

4. 如有必要，移动区段

   如果计算出的映射索引，在现有对象的位置上，需要移动区段。

   对于上述数组，如果要插入4：计算得的索引为1，需要将索引位置开始的所有元素后移：

```
  0   1   2   3   4
+---+---+---+---+---+
| 2 |   | 5 | 8 |   |
+---+---+---+---+---+
```

5. 最后，设置值，将数据长度加1。

   主要代码如下，省略了部分细节：

```
public void put(int key, E value) {

    // 1. 计算索引
    int i = binarySearch(mKeys, 0, mSize, key);

    // 2. key已经有对应槽位，更新值
    if (i >= 0) {
        mValues[i] = value;
    } else {
        i = ~i;

        // 3. 扩容
        if (mSize >= mKeys.length) {
        }

        // 4. 移动区段
        if (mSize - i != 0) {
            // Log.e("SparseArray", "move " + (mSize - i));
            System.arraycopy(mKeys, i, mKeys, i + 1, mSize - i);
            System.arraycopy(mValues, i, mValues, i + 1, mSize - i);
        }

        // 4. 设置值，长度加1
```

```
            mKeys[i] = key;
            mValues[i] = value;
            mSize++;
        }
    }
```

get和遍历

- 如果索引不存在，`indexOfKey(int key)`，将会返回负数值。
- 遍历需要获取数组的总的对象大小，然后用`keyAt(int index)`获取索引或者`valueAt(int index)`获取值。

```
int key = 0;
for(int i = 0; i < sparseArray.size(); i++) {
       key = sparseArray.keyAt(i);
       Object obj = sparseArray.valueAt(key);
}
```

效率的提升和使用限制

稀疏数组的使用，对于索引是整数的情景，有时能带来一些效率的提升。

1. 减少了hashCode时间消耗
2. 减小了所使用的内存大小。

和`SparseArray`类似的，有`SparseBooleanArray`，`SparseIntArray`。前者，减少了存储对象占用的空间，后者减少了类型转换。

但在所管理的对象数量很大时，效率却反而有可能更低：

1. 在插入的时候，有可能导致大段数组的复制;
2. 在删除之后，也有可能导致数组的大段元素被按个移动（不是复制数组，而是一个一个单独移动）;
3. 索引的映射，采用了二分查找，时间复杂度为O(logn)。

写完这篇文章，搜索了一下相关的内容 发现中文的文章，大部分都是重复的，几个原创的也有一些错误。特强调如下：

1. **`SparseArray`**是针对**`HashMap`**做的优化。
    1. `HashMap`内部的存储结构，导致一些内存的浪费。
    2. 在刚扩容完，`SparseArray`和`HashMap`都会存在一些没被利用的内存。

2. `SparseArray`并不是任何时候都会更快，有时反而会更慢