SparseArray 的使用及实现原理

阅读 3655 收藏 133 2016-08-29
原文链接：extremej.itscoder.com

> 文章来源：itsCoder 的 WeeklyBolg 项目
>
> itsCoder主页：itscoder.com/
>
> 作者：Joe
>
> 审阅者：allenwu

序言

相信大家都用过 `HashMap` 用来存放键值对，最近在项目中使用 `HashMap` 的时候发现，有时候 IDE 会提示我这里的 `HashMap` 可以用 `SparseArray` 或者 `SparseIntArray` 等等来代替。

细心的朋友可能也发现了这个提示，并且会发现并不是所有的 `HashMap` 都会提示替换。今天就来一探究竟，到底 `SparseArray` 跟 `HaspMap` 相比有什么优缺点，又是在什么场景下来使用的呢?

如果你对于 `HashMap` 的实现原理还不是很了解，推荐你阅读allen的这篇Java 集合框架源码分析系列之 HashMap。

SparseArray 的使用

创建

```
SparseArray sparseArray = new SparseArray<>();
SparseArray sparseArray = new SparseArray<>(capacity);
```

首先来看看如何创建一个 `SparseArray`，前文说了 `SparseArray` 是用来替换 `HashMap` 的，而 `SparseArray` 只需要指定一个泛型，似乎说明 `key` 的类型在 `SparseArray` 内部已经指定了呢?

`SparseArray` 有两个构造方法，一个默认构造方法，一个传入容量。

put()

创建完 `sparseArray` 后，来看看怎么往里面存放数据吧。

```
sparseArray.put(int key,Student value);
```

噢，原来 `SparseArray` 存放的键值对中的键是 `int` 型的数据，为什么呢？后面分析源码的时候再讲。

`put()` 就跟 `HashMap` 的使用方法一样。

get()

```
sparseArray.get(int key);
sparseArray.get(int key,Student valueIfNotFound);
```

```
sparseArray.remove(int key);
```

index

前面几个都跟 `HashMap` 没有什么太大区别，而这个 `index` 就是 `SparseArray` 所特有的属性了，这里为了方便理解先提一嘴， `SparseArray` 从名字上看就能猜到跟数组有关系，事实上他底层是两条数组，一组存放 `key`，一组存放 `value`，知道了这一点应该能猜到 `index` 的作用了。

`index` — `key` 在数组中的位置。 `SparseArray` 提供了一些跟 `index` 相关的方法：

```
sparseArray.indexOfKey(int key);
sparseArray.indexOfValue(T value);
sparseArray.keyAt(int index);
sparseArray.valueAt(int index);
```

```
sparseArray.setValueAt(int index);
sparseArray.removeAt(int index);
sparseArray.removeAt(int index,int size);
```

SparseArray 实现原理

前面简单的介绍了 `SparseArray` 的使用，为了在实际工作中最合理的选用数据结构，深入的了解每种数据结构的实现原理是很有必要的，这样可以更好的理解和比较不同数据结构之间的优缺点，比死记概念要更好，甚至可以根据自己的具体需求去实现最适合需求的数据结构。

话不多说，打开源码来一探究竟。我看源码的习惯，是先看这个类文件的注释，一般能在整体上给个思路。

> SparseArrays map integers to Objects. Unlike a normal array of Objects,there can be gaps in the indices. It is intended to be more memory efficient than using a HashMap to map Integers to Objects, both because it avoids auto-boxing keys and its data structure doesn't rely on an extra entry object for each mapping.

这段注释基本解释了该类的作用：**使用 `int[]` 数组存放 `key`，避免了 `HashMap` 中基本数据类型需要装箱的步骤，其次不使用额外的结构体（Entry)，单个元素的存储成本下降。**

如果你对装箱的概念还不清楚，可以看看小黑屋的这篇文章：Java中的自动装箱与拆箱。

初始化

`SparseArray` 没有继承任何其他的数据结构，实现了 `Cloneable` 接口。

```
private int[] mKeys;
private Object[] mValues;
private int mSize;
public SparseArray() {this(10);}
public SparseArray(int initialCapacity) {
        if (initialCapacity == 0) {
            mKeys = EmptyArray.INT;
            mValues = EmptyArray.OBJECT;
        } else {
            mValues = ArrayUtils.newUnpaddedObjectArray(initialCapacity);
```

```
            mKeys = new int[mValues.length];
        mSize = 0;
```

初始化`SparseArray`只是简单的创建了两个数组。

put()

接下来就是往`SparseArray`中存放数据。

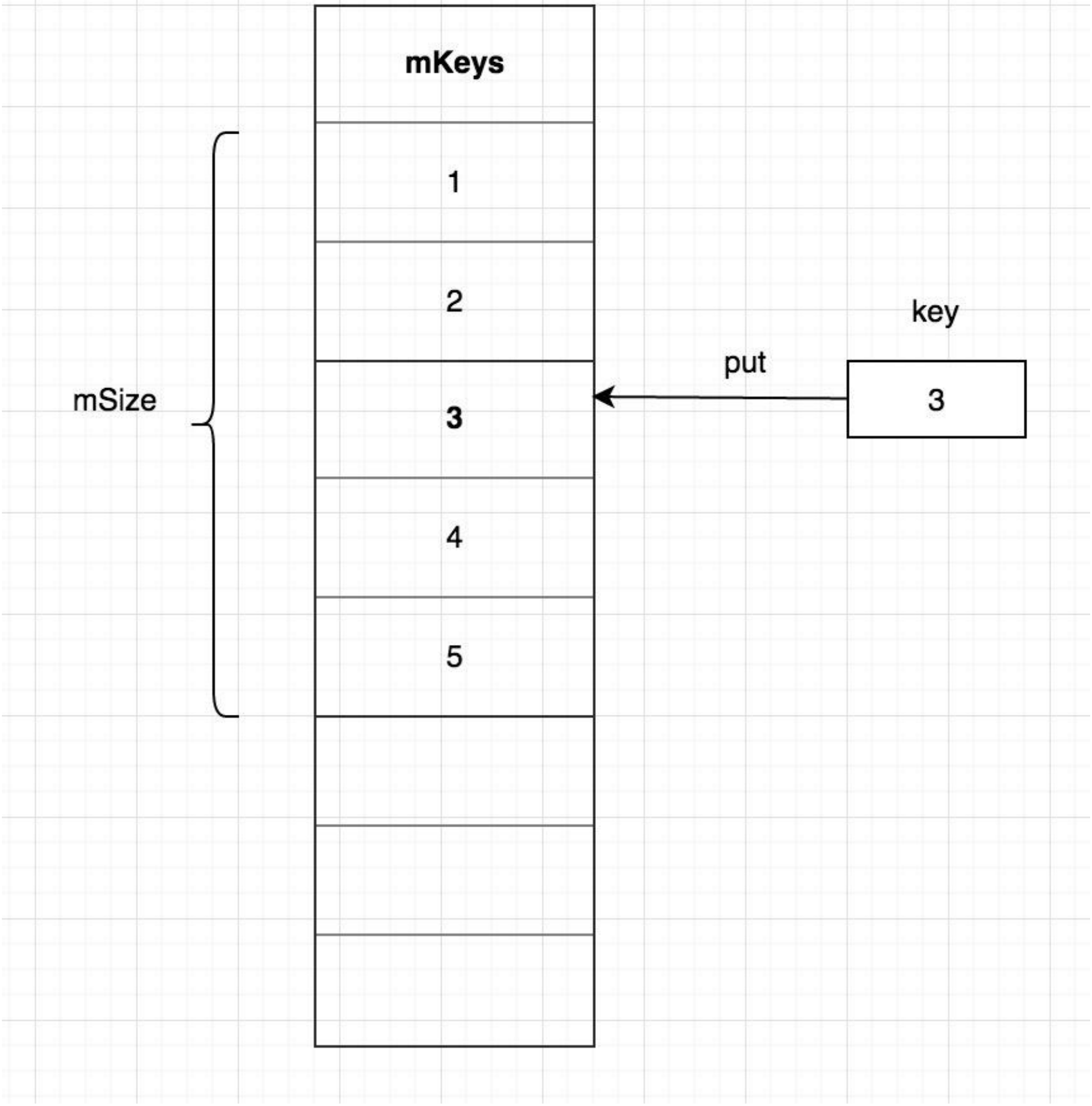
```
public void put(int key, E value) {
    int i = ContainerHelpers.binarySearch(mKeys, mSize, key);
    if (i >= 0) {
        mValues[i] = value;
    } else {
        i = ~i;
        if (i < mSize && mValues[i] == DELETED) {
            mKeys[i] = key;
            mValues[i] = value;
            return;
        if (mGarbage && mSize >= mKeys.length) {
            gc();
            i = ~ContainerHelpers.binarySearch(mKeys, mSize, key);
        mKeys = GrowingArrayUtils.insert(mKeys, mSize, i, key);
        mValues = GrowingArrayUtils.insert(mValues, mSize, i, value);
        mSize++;
```

上面这段代码是一次插入数据的操作，单看的话有些难懂，因为插入跟删除之间有一定的关系，所以要看懂这段代码，还必须搞懂删除的逻辑。在看删除之前，还是先大体梳理一下插入的几个特点：

- **存放`key`的数组是有序的（二分查找的前提条件）**
- **如果冲突，新值直接覆盖原值，并且不会返回原值（`HashMap`会返回原值）**
- **如果当前要插入的 key 的索引上的值为DELETE，直接覆盖**
- **前几步都失败了，检查是否需要`gc()`并且在该索引上插入数据**

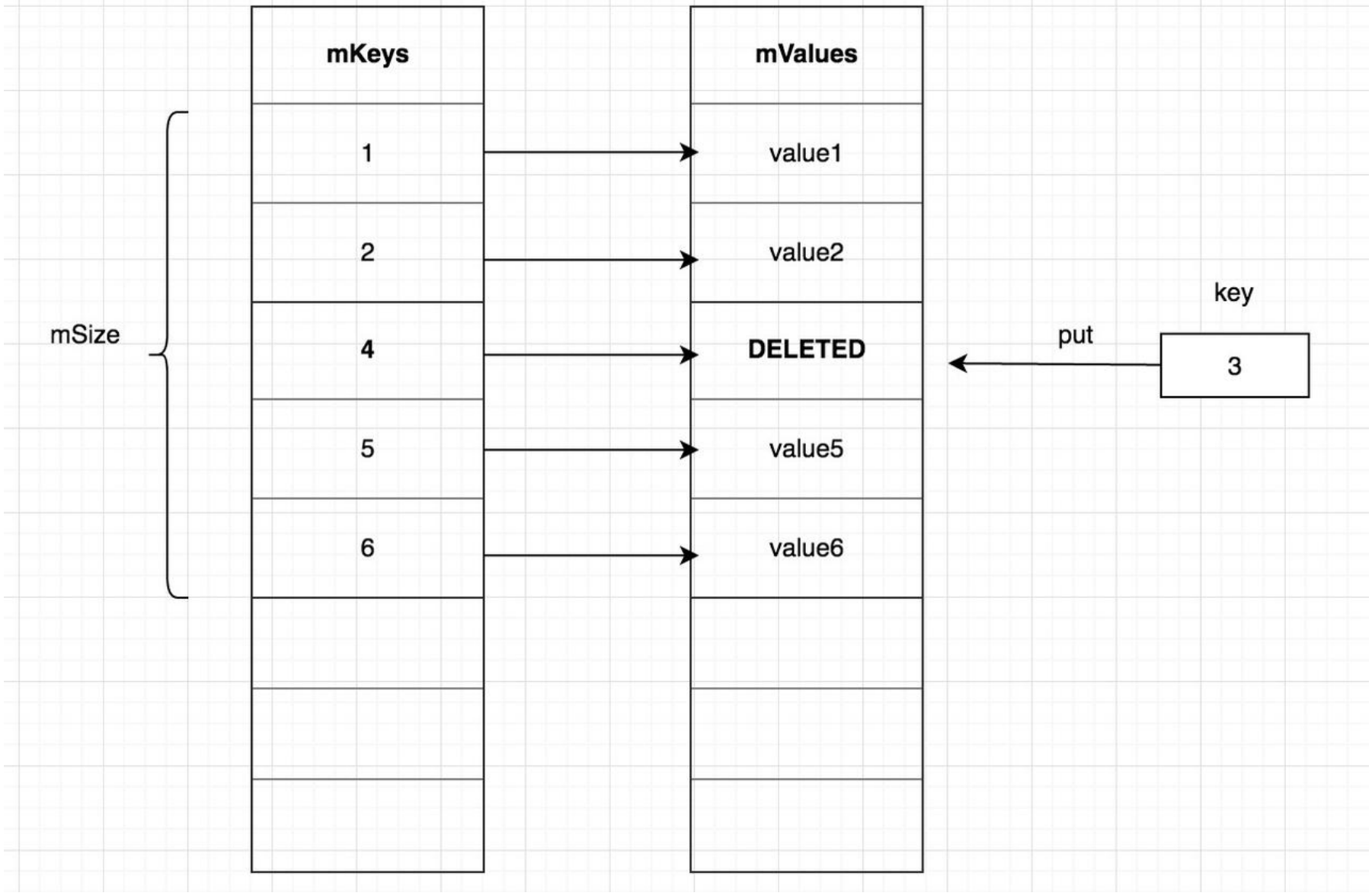
插入的逻辑大体上是这四点，理解起来可能还是有些抽象，我们来几张图：

冲突直接覆盖



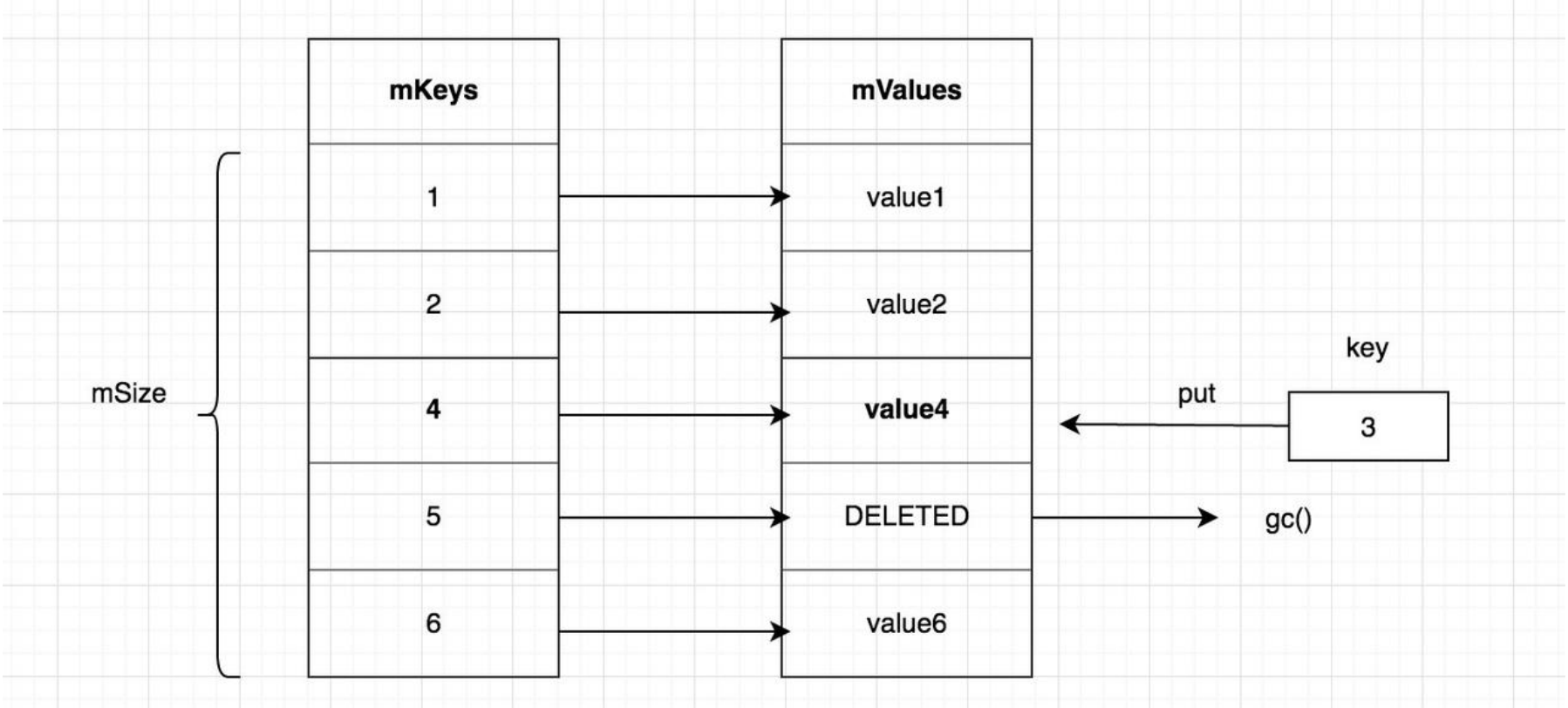
上面这个图，插入一个 `key=3` 的元素，因为在 `mKeys` 中已经存在了这个值，则直接覆盖。

插入索引上为DELETED



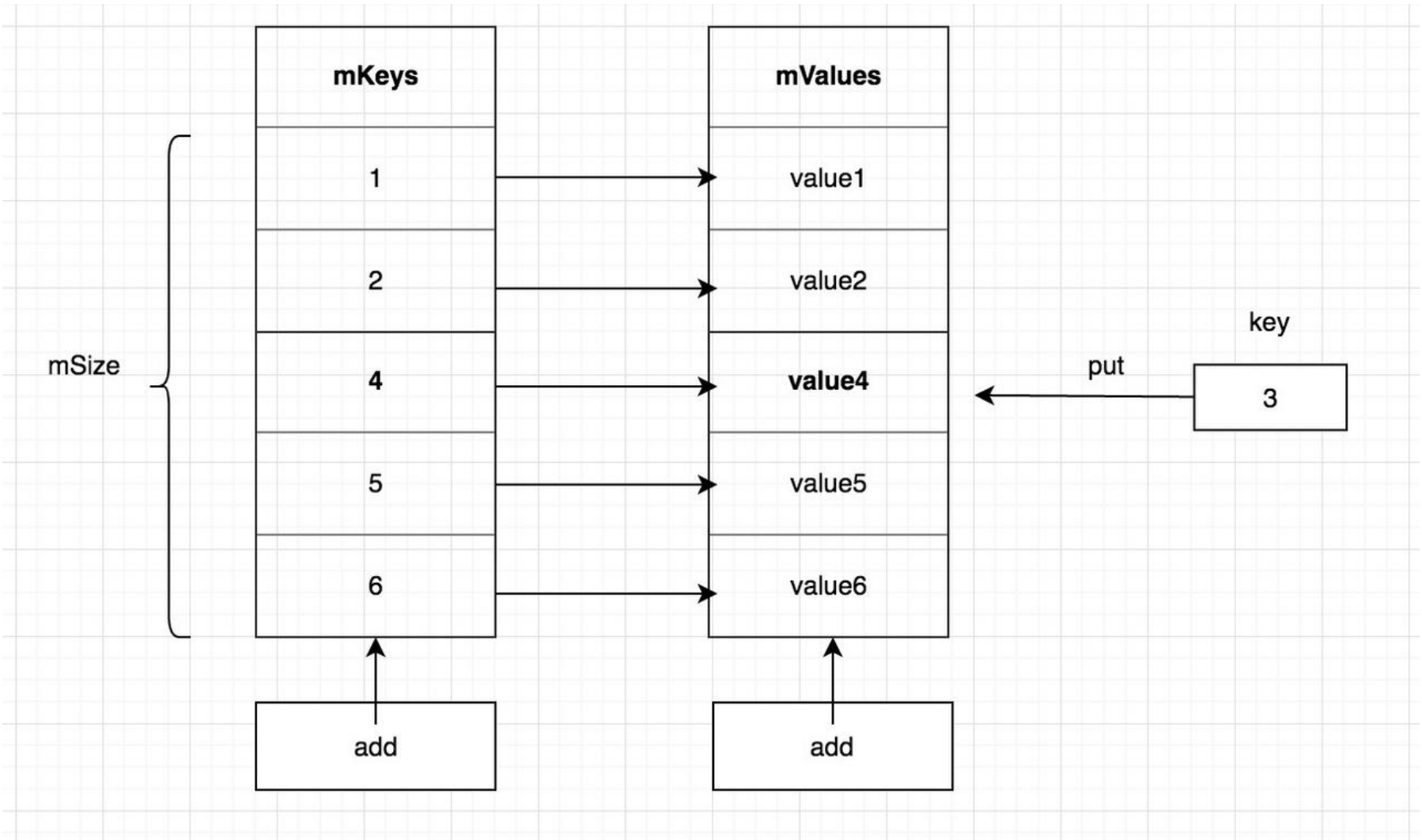
注意 `mKeys` 中并没有 3 这个值，但是通过二分查找得出来，目前应该插入的索引位置为 2，即 `key=4` 所在的位置，而当前这个位置上对应的 `value` 标记为 `DELETED` 了，所以会直接将该位置上的 `key` 赋值为 3，并且将该位置上的 `value` 赋值为 `put()` 传入的对象。

索引上有值，但是应该触发 `gc()`



注意这个图跟前面的几个又一个区别，那就是数组已经满容量了，而且 3 应该插入的位置已经有 4 了，而 5 所指向的值为 `DELETED`，**这种情况下，会先去回收 `DELETED`,重新调整数组结构，图中的例子则会回收 5 ,然后再重新计算 3 应该插入的位置**

满容且无法 `gc()`



这种情况下，就只能对数组进行扩容，然后插入数据。

结合这几个图，插入的流程应该很清晰了，但是`put()`还有几个值得我们探索的点，首先就是二分查找的算法，这是一个很普通的二分算法，注意最后一行代码，当找不到这个值的时候`return ~lo`，实际上到这一步的时候，理论上`lo==mid==hi`。所以这个位置是最适合插入数据的地方。但是为了让能让调用者既知道没有查到值，又知道索引位置，做了一个取反操作，返回一个负数。这样调用处可以首先通过正负来判断命中，之后又可以通过取反获取索引位置。

```
static int binarySearch(int[] array, int size, int value) {
    int lo = 0;
    int hi = size - 1;
    while (lo <= hi) {
        final int mid = (lo + hi) >>> 1;
        final int midVal = array[mid];
        if (midVal < value) {
            lo = mid + 1;
        } else if (midVal > value) {
            hi = mid - 1;
        } else {
            return mid;
    return ~lo;
```

第二个点就是，插入数据具体是怎么插入的。

```
mKeys = GrowingArrayUtils.insert(mKeys, mSize, i, key);
mValues = GrowingArrayUtils.insert(mValues, mSize, i, value);
```

```
public static int[] insert(int[] array, int currentSize, int index, int element) {
    assert currentSize <= array.length;
    if (currentSize + 1 <= array.length) {
        System.arraycopy(array, index, array, index + 1, currentSize - index);
        array[index] = element;
        return array;
    int[] newArray = ArrayUtils.newUnpaddedIntArray(growSize(currentSize));
    System.arraycopy(array, 0, newArray, 0, index);
        newArray[index] = element;
    System.arraycopy(array, index, newArray, index + 1, array.length - index);
    return newArray;
```

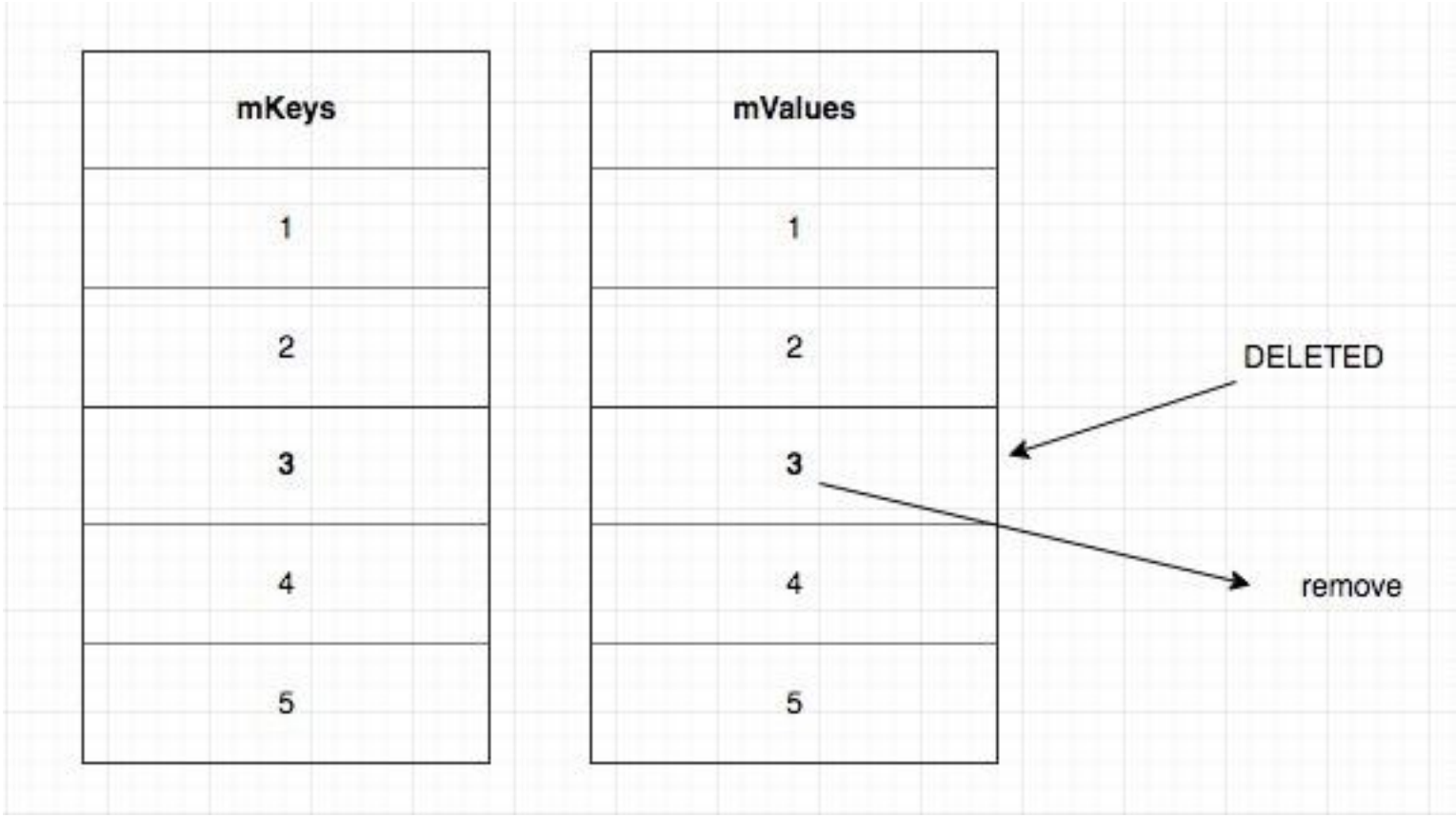
`put()`部分的代码就全部完毕了，接下来先来看看`remove()`是怎么处理的?

```
public void remove(int key) {
    delete(key);
    public void delete(int key) {
    int i = ContainerHelpers.binarySearch(mKeys, mSize, key);
    if (i >= 0) {
        if (mValues[i] != DELETED) {
            mValues[i] = DELETED;
            mGarbage = true;
            private static final Object DELETED = new Object();
```

事实上，`SparseArray` 在进行 `remove()` 操作的时候分为两个步骤：

- 删除 `value` — 在 `remove()` 中处理
- 删除 `key` — 在 `gc()` 中处理，注意这里不是系统的 GC，只是 `SparseArray` 的一个方法

`remove()` 中，将这个 `key` 指向了 `DELETED`，这时候 `value` 失去了引用，如果没有其它的引用，会在下一次系统内存回收的时候被干掉。来看一张图：



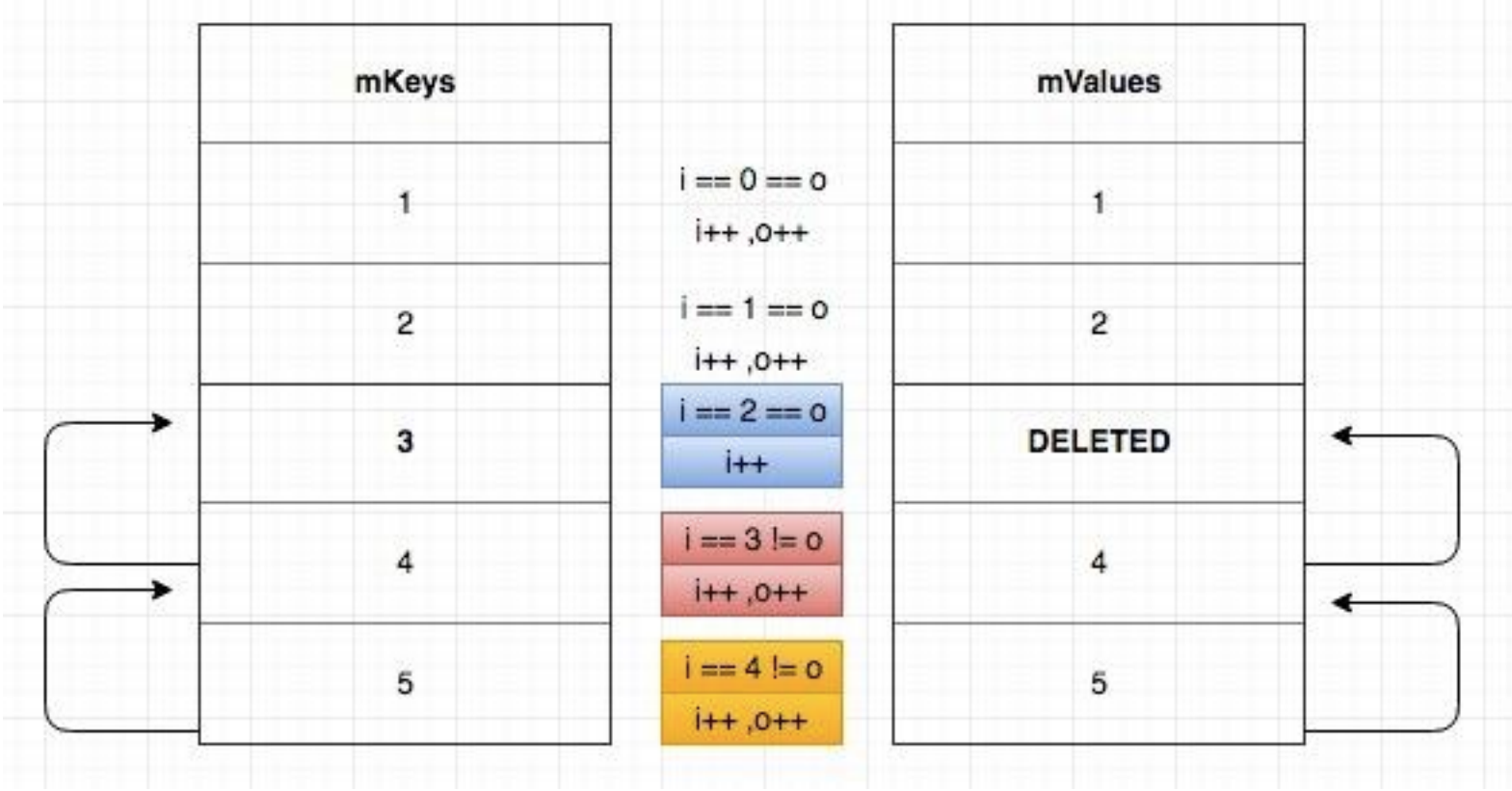
但是可以看到 `key` 仍然保存在数组中，并没有马上删除，目的应该是为了保持索引结构，同时不会频繁压缩数组，保证索引查询不会错位，那么 `key` 什么时候被删除呢？当 `SparseArray` 的 `gc()` 被调用时。

> To help with performance, the container includes an optimization when removing keys: instead of compacting its array immediately, it leaves the removed entry marked as deleted. The entry can then be re-used for the same key, or compacted later in a single garbage collection step of all removed entries. This garbage collection will need to be performed at any time the array needs to be grown or the the map size or entry values are retrieved.

gc()

```
private void gc() {
    int n = mSize;
    int o = 0;
    int[] keys = mKeys;
    Object[] values = mValues;
    for (int i = 0; i < n; i++) {
        Object val = values[i];
        if (val != DELETED) {
            if (i != o) {
                keys[o] = keys[i];
                values[o] = val;
                values[i] = null;
            o++;
    mGarbage = false;
    mSize = o;
```

上面的这段代码，直接看可能理解起来也比较困难，主要是理解 `o` 只有在值等于 `DELETED` 的时候才不会向后移，也就是说，当 `i` 向后移动一位的时候， `o` 还在值为 `DELETED` 的地方，而这时候因为 `i != o` ，就会触发第二个判断条件，将 `i` 位置的元素向前移动到 `o` 处。来看一张图：



如上图所示，在 3 之前，`i` 与 `o` 都是相等的，而到 3 的时候，因为值为 `DELETED`，所以只有 `i++`，而 `o` 的值仍然等于 2，**重点来了，到 4 的时候，发现 `i!=o` ,则会将 4 向前移动到 3，这时候 `o++` 了，但是因为 `o` 始终小于 `i` 一位（这个例子里面），因此后面的元素均会向前移动一位。**

`gc()` 的原理了解了，那么在什么情况下会触发 `gc()` 呢？上面已经知道在添加元素的时候可能会触发 `gc()`，除了添加元素，前文提到过一系列跟 `index` 有关的方法，事实上在调用这些方法的时候，都会试图去触发 `gc()`，这样可以返回给调用者一个精确的索引值。

get()

```
public E get(int key, E valueIfKeyNotFound) {
    int i = ContainerHelpers.binarySearch(mKeys, mSize, key);
    if (i < 0 || mValues[i] == DELETED) {
        return valueIfKeyNotFound;
    } else {
        return (E) mValues[i];
```

`get()` 中的代码就比较简单了，通过二分查找获取到 `key` 的索引，通过该索引来获取到 `value`

SparseArray 的系列

除了前面分析的 `SparseArray`，其实还有其它的一些类似的数据结构，它们总结起来就是用于存放基

本数据类型的键值对：

- `SparseIntArray` — int:int
- `SparseBooleanArray` — int:boolean
- `SparseLongArray` — int:long

就不一一列举了，有兴趣的可以一个一个去看看，实现原理都差不太多。

总结

了解了`SparseArray`的实现原理，就该来总结一下它与`HashMap`之间来比较的优缺点

优势：

- 避免了基本数据类型的装箱操作
- 不需要额外的结构体，单个元素的存储成本更低
- 数据量小的情况下，随机访问的效率更高

有优点就一定有缺点

- 插入操作需要复制数组，增删效率降低
- 数据量巨大时，复制数组成本巨大，`gc()`成本也巨大
- 数据量巨大时，查询效率也会明显下降

学习完了`SparseArray`，相信你对这个系列的数据结构有了更深的认识，什么时候选择什么样的数据结构，在一定程度上对于程序的运行效率会有那么一些帮助。

Android

相关热门文章

孔乙己的疑问：单例模式有几种写法

HongJay 10 2

Android LowMemoryKiller 简介

Yuloran 9

自己动手实现Android中的三级缓存框架

hankinghu 10

SocialSdk-快速接入登录分享

尚妆产品技术刊读 1

Android自定义View之实现简单炫酷的球体进度球

hankinghu 14 12

评论

输入评论...