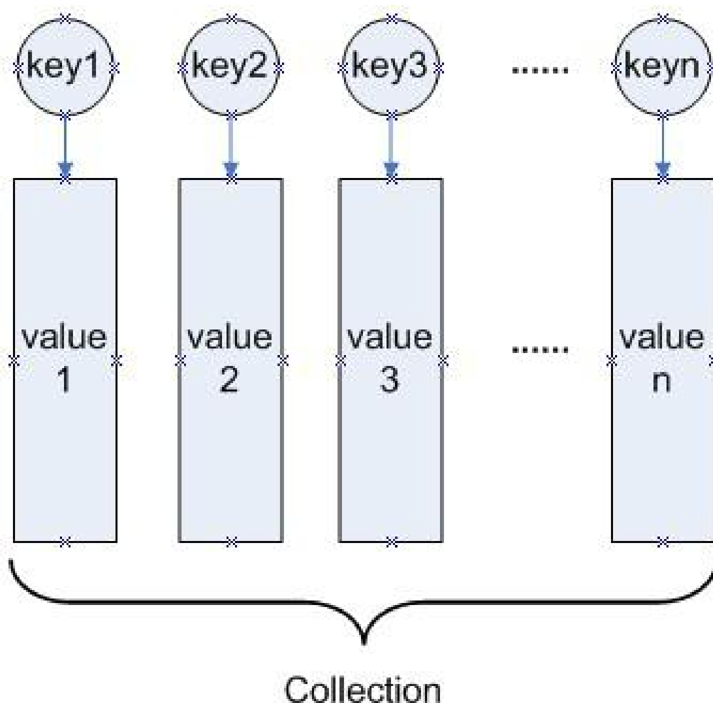
Guava学习之AbstractMapBasedMultimap

AbstractMapBasedMultimap源码分析：AbstractMapBasedMultimap是Multimap接口的基础实现类，实现了Multimap中的绝大部分方法，其中有许多的方法还是靠实现类的具体实现，比如size()方法，其计算方法在不同实现是不一样的。同时，AbstractMapBasedMultimap类也定义了自己的一些方法，比如createCollection()。AbstractMapBasedMultimap类中主要存在以下两个成员变量

```
private transient Map<K, Collection<V>> map;
private transient int totalSize;
```

map是存放所有的键值对；totalSize是存放所有值的数量。

AbstractMapBasedMultimap类将multimap转换为一个map，所有的元素都是存放在Map map数据结构中，如下图所示：



Guava中AbstractMapBasedMultimap源码

所有实现Multimap接口的子类必须实现createCollection()方法，用于存放value的值。当需要向map中插入一对key→value键值对，而这个key不存在于multimap中，此时就需要调用creat

eCollection()方法来创建一个新的Collection以便存放这个key对应的value。根据createCollection()方法的不同实现，使得同一个key中的value可以相同或者不同，比如ArrayListMultimap类重写createCollection()方法如下：

```
 @Override List<V> createCollection() {
   return new ArrayList<V>(expectedValuesPerKey);
 }
```

TreeMultimap类重写createCollection()方法如下：

```
 @Override SortedSet<V> createCollection() {
   return new TreeSet<V>(valueComparator);
 }
```

所以，下面代码输出的结果是不一样的：

```
TreeMultimap<Comparable, Comparable> comparableComparableTreeMultimap =
   TreeMultimap.create();
comparableComparableTreeMultimap.put("wyp", "xxx");
comparableComparableTreeMultimap.put("wyp", "xxx");
System.out.println(comparableComparableTreeMultimap);

ArrayListMultimap<Object, Object> objectObjectArrayListMultimap =
   ArrayListMultimap.create();
objectObjectArrayListMultimap.put("wyp", "xxx");
objectObjectArrayListMultimap.put("wyp", "xxx");
System.out.println(objectObjectArrayListMultimap);
```

输出结果分别为：

```
{wyp=[xxx]}
{wyp=[xxx, xxx]}
```

　　因为ArrayListMultimap类中存储value是用ArrayList实现的，所以ArrayListMultimap中同一

个key中可以存放相同的value；而TreeMultimap类中存储value是用TreeSet实现的，我们知道，TreeSet是不同存放相同的值，所以导致了TreeMultimap中同一个key不能存放相同的value，所以才会有上面的输出。

上面提到，AbstractMapBasedMultimap类中主要有map和totalSize两个成员变量，这里需要对totalSize做个说明，如下程序：

```
ArrayListMultimap<Object, Object> objectObjectArrayListMultimap =
    ArrayListMultimap.create();
objectObjectArrayListMultimap.put("a", "1");
objectObjectArrayListMultimap.put("a", "2");
objectObjectArrayListMultimap.put("b", "3");

System.out.println(objectObjectArrayListMultimap);
System.out.println(objectObjectArrayListMultimap.size());
```

输出的结果为：

```
{b=[3], a=[1, 2]}
3
```

是不是很奇怪？objectObjectArrayListMultimap.size()为什么输出的是3，而不是2？先看看size()函数的实现：

```
@Override
  public int size() {
    return totalSize;
  }
```

而totalSize的值计算可以用下面程序来说明

```
    totalSize = 0;
    for (Collection<V> values : map.values()) {
      checkArgument(!values.isEmpty());
      totalSize += values.size();
    }
```

看到这里应该很明白，为什么上面的结果返回的是3，而不是2。因为totalSize计算的是所有value的个数，而不是所有key的个数。从代码中可以看出，所有对map中元素的添加和删除都会对totalSize作出相应的加减处理，而这些加减是针对value数量的变化。

本博客文章除特别声明，全部都是原创！
原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接:【】（）