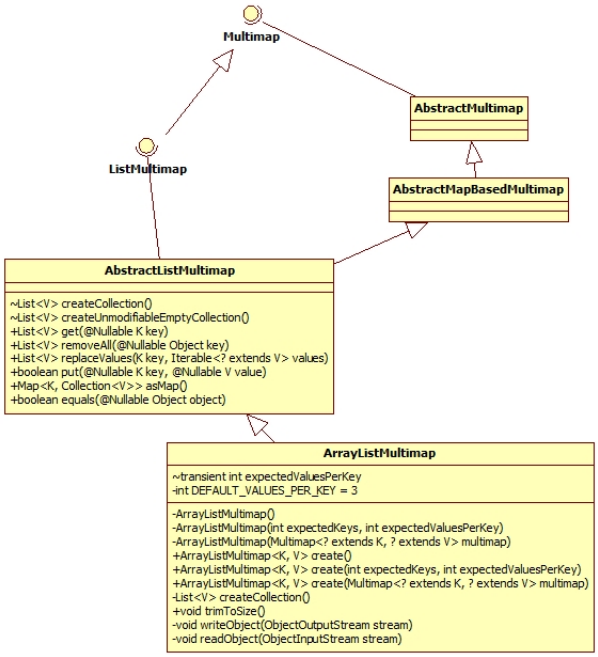
Guava学习之ArrayListMultimap

ArrayListMultimap类的继承关系如下图所示：



Guava ArrayListMultimap

ListMultimap是一个接口，继承自Multimap接口。ListMultimap接口为所有继实现自ListMultimap的子类定义了一些共有的方法签名。ListMultimap接口并没有定义自己特有的方法签名，里面所有的方法都是重写了Multimap接口中的声明，只是将Multimap接口中返回Collection类型的函数修改成返回List类型。比如Multimap接口中get函数的函数原型为Collection get(@Nullable K key);而ListMultimap接口则变成了List get(@Nullable K key);这是因为Multimap接口为多个实现类定义了其子类必须实现的方法，由于其子类（TreeMultimap、ArrayListMultimap）的实现不一样，而Multimap接口又定义了它们共有的方法，所以里面的函数原型大多数的返回类型为Collection，我们
知道List、Set都继承自Collection。ListMultimap接口的这种设计符合《里氏替换法则》。

AbstractListMultimap继承自AbstractMapBasedMultimap类，并实现了ListMultimap接口。AbstractListMultimap类中主要实现了ListMultimap接口里面的方法。同ListMultimap接口类似，AbstractListMultimap类将AbstractMapBasedMultimap类中返回Collection类型的函数修改成返回List类型。里面很多的实现都是调用了AbstractMapBasedMultimap类中相应函数的实现，仅仅简单的将返回Collection类型函数修改为List类型。

AbstractMapBasedMultimap类的介绍请
参见《Guava学习之AbstractMapBasedMultimap》。

ArrayListMul
timap类是Multimap接口的ArrayLis
t方法实现的类，在前面的《Guava学习之AbstractMapBasedMultimap》
文章中我们谈到了Multimap接口的几种实现，其中就有以ArrayList实现的。

这
里说的以
ArrayList实现或者
TreeSet实现是指key所对应的value
存放方式。详细介绍请看《Guava学习之AbstractMapBasedMultimap》

ArrayListMultimap类主要有三个构造函数，实现如下：

```
private ArrayListMultimap() {
  super(new HashMap<K, Collection<V>>());
  expectedValuesPerKey = DEFAULT_VALUES_PER_KEY;
}

private ArrayListMultimap(int expectedKeys, int expectedValuesPerKey) {
  super(Maps.<K, Collection<V>>newHashMapWithExpectedSize(expectedKeys));
  checkArgument(expectedValuesPerKey >= 0);
  this.expectedValuesPerKey = expectedValuesPerKey;
}

private ArrayListMultimap(Multimap<? extends K, ? extends V> multimap) {
  this(multimap.keySet().size(),
      (multimap instanceof ArrayListMultimap) ?
          ((ArrayListMultimap<?, ?>) multimap).expectedValuesPerKey :
          DEFAULT_VALUES_PER_KEY);
  putAll(multimap);
}
```

第一个构造函数其键存储是用HashMap来实现的；而value的存储是用ArrayList来实现的，如下所示：

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

其中的expectedValuesPerKey 表示预期一个key会有多少个value，这里直接将DEFAULT_VALUES_PER_KEY赋值给expectedValuesPerKey，通过源码我们可以得到DEFAULT_VALUES_PER_KEY=3。

第二个构造函数有两个参数int expectedKeys, int expectedValuesPerKey，其中expectedKeys表示用户预期有多少个key，expectedValuesPerKey表示用户预期一个key会有多少个value。如果你大致知道你的程序会有多少个key和value，建议用这个构造函数，这样可以省去由于空间不足而需要重新分配空间而带来的额外开销。

第三个构造函数是将另一个multimap中的所有键值元素直接复制到我们新建的multimap中，这个方式比较简单，和第一个构造函数类似，我们都不需要知道会有多少个key和value。

从上面的构造函数我们可以看出，这三个构造函数都是以private修饰的，这说明了我们不能直接调用ArrayListMultimap类的构造函数。由于此原因，ArrayListMultimap类为我们带来了以下几个用于创建ArrayListMultimap对象的静态方法：

```
public static <K, V> ArrayListMultimap<K, V> create() {
  return new ArrayListMultimap<K, V>();
}

public static <K, V> ArrayListMultimap<K, V> create(
    int expectedKeys, int expectedValuesPerKey) {
  return new ArrayListMultimap<K, V>(expectedKeys, expectedValuesPerKey);
}

public static <K, V> ArrayListMultimap<K, V> create(
    Multimap<? extends K, ? extends V> multimap) {
  return new ArrayListMultimap<K, V>(multimap);
}
```

可以看出，上面三个静态函数分别对应地调用ArrayListMultimap的三个构造函数，含义我就不说了，很简单。

如果你的ArrayListMultimap对象的value经过了各种的删除、添加操作，这时候可能会导致许多的value空间都没有用到，我们可以用下面的函数来将没用的空间去掉，使得各个key对应value空间的大小等于其对应的value个数：

```
public void trimToSize() {
  for (Collection<V> collection : backingMap().values()) {
    ArrayList<V> arrayList = (ArrayList<V>) collection;
    arrayList.trimToSize();
  }
}
```

backingMap()函数的实现：

```
Map<K, Collection<V>> backingMap() {
  return map;
}
```

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

里面的map就是构造函数super(new HashMap<k , Collection<V>>())中的HashMap。

注意，这里说的是value所用的空间，没说是key所用的空间。

（完）