## **DualPivotQuicksort source code**

这个算法是Arrays.java中给基本类型的数据排序使用的具体实现。它针对每种基本类型都做了实现，实现的方式有稍微的差异，但是思路都是相同的，所以这里只挑了int类型的排序来看。

整个实现中的思路是 首先检查数组的长度，比一个阈值(286)小的时候直接使用双轴快排。其它情况下，先检查数组中数据的顺序连续性。把数组中连续升序或者连续降序的信息记录下来，顺便把连续降序的部分倒置。这样数据就被切割成一段段连续升序的数列。

如果顺序连续性好，直接使用TimSort算法。这个我们之前介绍过，TimSort算法的核心在于利用数列中的原始顺序，所以可以提高很多效率。这里的TimSort算法是之前介绍的[TimSort算法](http://www.jianshu.com/p/10aa41b780f2" \t "http://www.jianshu.com/p/_blank)的精简版，剪掉了动态阈值的那一部分。

顺序连续性不好的数组直接使用了 双轴快排 + 长度小于47时（普通插入排序 和 成对插入排序）。成对插入排序是插入排序的改进版，它采用了同时插入两个元素的方式调高效率。双轴快排是从传统的单轴快排到3-way快排演化过来的，网上之前已经有很多博客介绍这种算法。

Arrays.java是Java中用来操作数组的类。使用这个工具类可以减少平常很多的工作量。了解其实现，可以避免一些错误的用法。

它提供的操作包括：

1. 排序 sort
2. 查找 binarySearch()
3. 比较 equals
4. 填充 fill
5. 转列表 asList()
6. 哈希 Hash()
7. 转字符串 toString()

这个类的代码量很多，Java1.7中有4000多行。因为每一种基本类型都做了兼容，所以整个类真正逻辑不多。下面简单介绍一下它各个功能的实现：

## **排序**

这里的排序实现有两种

一种是为基本类型数组设计的，它的对外接口有两种，如下:

//whole arraypublic static void sort(primitive[] a);

//sub arraypublic static void sort(primitive[] a, int fromIndex, int toIndex);

它们的具体实现方式是一样的都是调用了DualPivotQuicksort.sort(...)方法。这个方法的中文含义是 ****双轴快速排序****。它在性能上优于传统的单轴快速排序。

算法的逻辑可以参考[国外一篇博客](http://rerun.me/2013/06/13/quicksorting-3-way-and-dual-pivot/" \t "http://www.jianshu.com/p/_blank)  
如果想要阅读源码可以参考我的另一篇博客[双轴快速排序源码阅读笔记](http://www.jianshu.com/p/6d26d525bb96" \t "http://www.jianshu.com/p/_blank)

它是****不稳定****的

另一种是为Object对象设计的，它要求传进来的数组对象必须实现Comparable接口。  
它提供的接口如下：

// whole array, default asecpublic static void sort(Object[] a);

// subArray, default asecpublic static void sort(Object[] a, int fromIndex, int toIndex);

还有带泛型参数的接口，它需要指定一个比较器。

// whole array with comparatorpublic static <T> void sort(T[] a, Comparator<? super T> c);

// sub array with comparatorpublic static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c);

他的实现方式如下：

// java/utils/Arrays.javastatic final class LegacyMergeSort {

private static final boolean userRequested =

java.security.AccessController.doPrivileged(

new sun.security.action.GetBooleanAction(

"java.util.Arrays.useLegacyMergeSort")).booleanValue();

}

//sort 方法的实现public static void sort(Object[] a) {

if (LegacyMergeSort.userRequested)

legacyMergeSort(a);

else

//与TimSort的逻辑是相同的

ComparableTimSort.sort(a);

}

//legacyMergeSortprivate static void legacyMergeSort(Object[] a) {

Object[] aux = a.clone();

mergeSort(aux, a, 0, a.length, 0);

}

//归并排序

private static void mergeSort(Object[] src,

Object[] dest,

int low,

int high,

int off) {

// 小数组直接进行普通插入排序

if (length < INSERTIONSORT\_THRESHOLD) {

///...

return;

}

//下面是归并排序的实现，

///...

}

从上面的逻辑可以看出来，它的实现方式分为两种，一种是通过Arrays.java中的归并排序实现的，另一种采用了TimSort算法。其中Arrays.java中的归并排序的逻辑相对简单，是一种插入排序与传统归并排序的结合。当待排序的数组小于INSERTIONSROT\_THERSHOLD的时候直接进行插入排序，不再递归。TimSort算法也是一种插入排序与归并排序结合的算法，不过它的细节优化要比Arrays.java中的算法做的多。详细介绍可以参考[维基百科](https://en.wikipedia.org/wiki/Timsort" \t "http://www.jianshu.com/p/_blank)或者我的[TimSort 源码笔记](http://www.jianshu.com/p/10aa41b780f2" \t "http://www.jianshu.com/p/_blank)。

两种算法的切换依靠运行时系统变量的设置。具体参考[StackOverFlow](http://stackoverflow.com/questions/15893487/compile-in-java-6-run-in-7-how-to-specify-uselegacymergesort" \t "http://www.jianshu.com/p/_blank)上的一篇回答。我们默认情况下是不打开这个开关的，也就是说没有特殊要求的情况下，我们默认使用TimSort算法来实现排序。从注释上来看，在未来某个版本，Arrays.java中的merge方法将会被删除掉。

这个排序方法是 ****稳定**** 的。

## **查找**

Arrays.java中只提供了二分查找。二分查找的前提就是数组是经过升序排序的，所以使用之前务必确保数组是有序的。

调用的接口比较简单：

public static int binarySearch(primative[] a, primative key);public static int binarySearch(primative[] a, int fromIndex, int toIndex, primative key);public static int binarySearch(Object[] a, Object key);public static int binarySearch(Object[] a, int fromIndex, int toIndex, Object key);public static <T> int binarySearch(T[] a, T key, Comparator< ? super T> c);public static <T> int binarySearch(T[] a, int fromIndex, int toIndex, T key,Comparator<? super T> c);

## **equals**

这个也比较简单，equals方法与==的不同大家应该都很熟悉了，这里直接贴出接口:

// 基本类型

public static boolean equals(long[] a, long[] a2) {

//...

}

// 对象

public static boolean equals(Object[] a, Object[] a2) {

//...

}

// 高维数组的equal比较，通过递归实现

// 这里没有对循环引用进行检查，如果两个如组同时存在循环引用的情况下可能出现死循环的风险。

public static boolean deepEquals(Object[] a1, Object[] a2) {

//...

}

## **fill 填充**

批量初始化的时候不要自己写for循环了，已经有人帮我们写好了。

// 基本类型批量赋值public static void fill(int[] a, int fromIndex, int toIndex, int val) {

rangeCheck(a.length, fromIndex, toIndex);

for (int i = fromIndex; i < toIndex; i++)

a[i] = val;

}// 对象批量赋值public static void fill(Object[] a, int fromIndex, int toIndex, Object val) {

rangeCheck(a.length, fromIndex, toIndex);

for (int i = fromIndex; i < toIndex; i++)

a[i] = val;

}

## **复制**

数组复制的最终实现是调用了JVM中的方法。具体没有深究，但在数据量大的时候应该能快些。

// @file Arrays.java// 基本类型的复制，从0开始到指定长度public static byte[] copyOf(byte[] original, int newLength) {

byte[] copy = new byte[newLength];

System.arraycopy(original, 0, copy, 0, Math.min(original.length, newLength));

return copy;

}// 基本类型复制，从指定起点到指定终点public static byte[] copyOfRange(byte[] original, int from, int to) {

int newLength = to - from;

if (newLength < 0)

throw new IllegalArgumentException(from + " > " + to);

byte[] copy = new byte[newLength];

System.arraycopy(original, from, copy, 0,

Math.min(original.length - from, newLength));

return copy;

}//这里是泛型数组的复制， 结合泛型进阶中的内容，这里好理解很多。public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {

T[] copy = ((Object)newType == (Object)Object[].class)

? (T[]) new Object[newLength]

: (T[]) Array.newInstance(newType.getComponentType(), newLength);

System.arraycopy(original, 0, copy, 0,

Math.min(original.length, newLength));

return copy;

}

// @file System.javapublic static native void arraycopy(Object src, int srcPos, Object dest, int destPos, int length);

//

## **转换成列表 asList**

将一组对象转换成列表，这里一定要注意返回的ArrayList并非平常用的java.util.ArrayList ，而是Arrays.java中定义的一个简单的静态内部类--ArrayList。它不支持添加和移除元素，不支持扩容。

@file java/util/Arrays.java

@SafeVarargspublic static <T> List<T> asList(T... a) {

return new ArrayList<>(a);

}

//注意，此ArrayList非平常用的ArrayList;这里的实现比较简单。//不能扩容，所以不支持add,remove等操作。private static class ArrayList<E> extends AbstractList<E>

implements RandomAccess, java.io.Serializable {

/// ...

}

## **哈希 hash**

高维数组的哈希计算，采用递归实现。同样，如果自己的某个元素直接或者间接持有自己，会出现死循环。

// 基本类型哈希

public static int hashCode(long a[]) {

// ...

}

// 对象哈希

public static int hashCode(Object a[]) {

//...

}

// 高维数组哈希，采用递归实现。如果自己的某个元素直接或者间接持有自己，会出现死讯环，

// 所以`Object[]`最好直接使用`hashcode(Object)`。

public static int deepHashCode(Object a[]) {

//...

}

## **toString**

有了这个方法，打Log的时候就不需要写循环了。  
这里的高维数组直接或者间接持有自己的情况****不会出现****死循环。因为这里做了特殊处理，用一个Set保存了打印过的数组。

// 基本类型

public static String toString(int[] a) {

// ...

}

// 对象

public static String toString(Object[] a) {

// ...

}

// 高维数组toString(). 这里处理了自我持有的问题。

public static String deepToString(Object[] a) {

// ...

deepToString(a, buf, new HashSet<Object[]>());

return buf.toString();

}

// 真正的实现方法， 递归实现。

// 使用了一个Set来存储已经打印过的类，如果再次出现这个对象，直接打印[...]

private static void deepToString(Object[] a, StringBuilder buf,

Set<Object[]> dejaVu) {

//...

}