Arrays.sort方法--JDK1.7的双轴快排原理DualPivotQuickSort

# Arrays工具类介绍

public class **Arrays** extends Object

java.lang.Object

**java.util.Arrays**

This class contains various methods for **manipulating arrays** (such as **sorting and searching**). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a NullPointerException, if the specified array reference is null, except where noted.

The documentation for the methods contained in this class includes briefs description of the implementations. **Such descriptions** should be regarded as implementation notes, rather than parts of the specification. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by sort(Object[]) does not have to be a MergeSort, but it does have to be stable.)

This class is a member of the *Java Collections Framework*.

java.util.Arrays类是一个工具类，内部提供的很多静态方法，主要包括**binarySearch、copyOf、copyOfRange、equals、fill、hashCode、parallelPrefix、parallelSort、setAll、sort、spliterator、stream、toString**。

其中使用最频繁的方法：**binarySearch二分查找、sort方法、toString方法**等。这里重点分析下sort方法。

# Arrays.sort()的简单API介绍

## 简单介绍Arrays.sort

Arrays.sort()方法存在很多重载，主要对不同的数据类型重载，可以传入的参数的数组类型为byte[]、char[]、short[]、int[]、long[]、float[]、double[]、Object[]、T[];都是升序**ascending**的。

## 以int为例，介绍

static void sort(int[] a)

Sorts the specified array into **ascending** numerical order.(升序)

static void sort(int[] a, int fromIndex, int toIndex)

对数组指定部分进行排序，范围为fromIndex---toIndex-1；不包括toIndex。

Sorts **the specified range** of the array into **ascending** order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.

Implementation note: The sorting algorithm is a **Dual-Pivot Quicksort** by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to **quadratic** performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

a - the array to be sorted

fromIndex - the index of the first element, inclusive, to be sorted

**toIndex - the index of the last element, exclusive, to be sorted**

Throws:

IllegalArgumentException - if fromIndex > toIndex

ArrayIndexOutOfBoundsException - if fromIndex < 0 or toIndex > a.length

## T[]数组，设定Comparator

对于特定类型的数组只能实现从小到大排序，但是如果需要从大到小排序，就必须自定义比较器Comparator<? super T>。

static <T> void **sort**(T[] a, Comparator<? super T> c)

Sorts the specified array of objects according to the order induced by the specified comparator.

static <T> void **sort**(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)

Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.

## 示例：

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

//注意，要想改变默认的排列顺序，不能使用基本类型（int,double, char）

//而要使用它们对应的类

Integer[] a = {9, 8, 7, 2, 3, 4, 1, 0, 6, 5};

//定义一个自定义类MyComparator的对象

Comparator cmp = new MyComparator();

**Arrays.sort(a, cmp);**

for(int i = 0; i < a.length; i ++) {

System.out.print(a[i] + " ");

}

}

}

//Comparator是一个接口，所以这里我们自己定义的类MyComparator要implents该接口 //而不是extends Comparator

class MyComparator implements Comparator<Integer>{

@Override

public int compare(Integer o1, Integer o2) {

//如果n1小于n2，我们就返回正值，如果n1大于n2我们就返回负值，

//这样颠倒一下，就可以实现反向排序了

if(o1 < o2) {

return 1;

}else if(o1 > o2) {

return -1;

}else {

return 0;

}

}

}

# Arrays.sort方法源码解读—DualPivotQuicksort

## 背景

Java1.7之前的快排

在Java1.7之前的快排只是普通的快排，跟我们今天要研究的快排不一样，性能也差了许多，但其中对快排所做的各种优化我们依然是可以学习的。

Java1.7的快排

Java1.7的快排是一种**双轴快排**，顾名思义：**双轴快排是基于两个轴来进行比较**，跟普通的选择一个点来作为轴点的快排是有很大的区别的。算法是由Vladimir Yaroslavskiy在2009年研究出来的，并在2011年发布在了Java1.7。由于Arrays.sort对于数组的排序做了各种各样的优化，并且大多数优化和我们今天要研究的双轴排序无关，所以我们暂且略过，以后有时间研究Arrays源码的时候我们再进行分析。

## 算法步骤

算法步骤

1.对于很小的数组（长度小于27），会使用**插入排序**InsertSort。

2.选择两个点P1,P2作为轴心，比如我们可以使用**第一个元素和最后一个元素**。

3.P1必须比P2要小，否则将这两个元素交换，现在将整个数组分为四部分：

（1）第一部分：比P1小的元素。

（2）第二部分：比P1大但是比P2小的元素。

（3）第三部分：比P2大的元素。

（4）第四部分：尚未比较的部分。

在开始比较前，除了轴点，其余元素几乎都在第四部分，直到比较完之后第四部分没有元素。

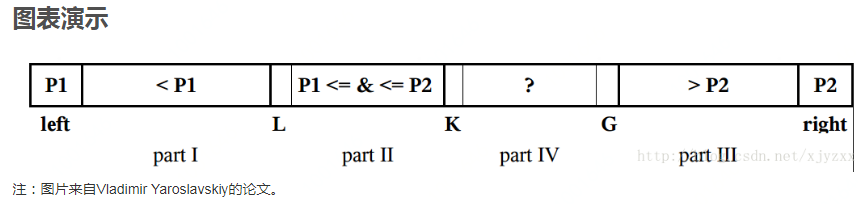
4.从第四部分选出一个元素a[K]，与两个轴心比较，然后放到第一二三部分中的一个。

5.移动L，K，G指向。

6.重复 4 5 步，直到第四部分没有元素。

7.将P1与第一部分的最后一个元素交换。将P2与第三部分的第一个元素交换。

8.递归的将第一二三部分排序。



## 源码：摘自网络

/\*\*

\* 双轴快排的具体实现

\* @param a 待排序数组

\* @param left 数组需排序上界

\* @param right 数组需排序下界

\* @param div 理解为从何位置取轴

\*/

private static void dualPivotQuicksort(int[] a, int left,int right, int div) {

int len = right - left;

//数组长度如果很小（27），则直接用插入排序对其排序

if (len < 27) {

for (int i = left + 1; i <= right; i++) {

for (int j = i; j > left && a[j] < a[j - 1]; j--) {

swap(a, j, j - 1);

}

}

return;

}

//取到位于1/div和div-1/div位置的点，并用他们来做轴

int third = len / div;

int m1 = left + third;

int m2 = right - third;

if (m1 <= left) {

m1 = left + 1;

}

if (m2 >= right) {

m2 = right - 1;

}

//确保left是小的，right是大的

if (a[m1] < a[m2]) {

swap(a, m1, left);

swap(a, m2, right);

}

else {

swap(a, m1, right);

swap(a, m2, left);

}

// 两个轴

int pivot1 = a[left];

int pivot2 = a[right];

// 代表比p1小和比p2大的两个指针

int less = left + 1;

int great = right - 1;

// 开始取出less到great之间的未知大小数据，与两个轴比较

// 并且将数据放入正确的区域后调整各个指针

for (int k = less; k <= great; k++) {

//如果取出的数比p1小，那么直接到less左侧，并且less右移

if (a[k] < pivot1) {

swap(a, k, less++);

}

//如果取出的数比p2大，那么首先确定great左侧没有比p2大的数

//然后与great位置的数字交换，great左移

//此时，great交换的数字肯定是比p2小或者相等的（首先确定过）

//那么此时再与p1相比，处理这个数的区间

else if (a[k] > pivot2) {

while (k < great && a[great] > pivot2) {

great--;

}

swap(a, k, great--);

if (a[k] < pivot1) {

swap(a, k, less++);

}

}

//如果这个数比p1大但是比p2小，则不需要交换，只需将k指针右移

}

//将p1与less左侧的第一个数交换

swap(a, less - 1, left);

//将p2与great右侧的第一个数交换

swap(a, great + 1, right);

// 计算出在两轴大小之间的个数

int dist = great - less;

//如果这个数很小（13），那么取轴的点向两边偏

if (dist < 13) {

div++;

}

// 对三个子区间分别排序，因为less-1和great+1是轴，已经排好了序

// 所以不需要比较

dualPivotQuicksort(a, left, less - 2, div);

dualPivotQuicksort(a, great + 2, right, div);

// 如果在中间区间的数字很多，那么排除掉一些相等的元素再进行排序

if (dist > len - 13 && pivot1 != pivot2) {

for (int k = less; k <= great; k++) {

if (a[k] == pivot1) {

swap(a, k, less++);

}

else if (a[k] == pivot2) {

swap(a, k, great--);

if (a[k] == pivot1) {

swap(a, k, less++);

}

}

}

}

// 对中间的区间排序

if (pivot1 < pivot2) {

dualPivotQuicksort(a, less, great, div);

}

}

总结：双轴排序利用了**区间相邻的特性**，对原本的快排进行了效率上的提高，很大程度上是利用了数学的一些特性，果然，算法跟数学还是息息相关的吖。

## DualPivotQuicksort.sort

Arrays.sort方法调用的是**DualPivotQuicksort的sort**方法。

public static void sort(int[] a) {

DualPivotQuicksort.sort(a, 0, a.length - 1, null, 0, 0);

}

## DualPivotQuicksort.sort原理

### 3种排序算法：QuickSort、MergeSort、InsertSort。

### 2个阈值threshold

/\*\*

\* If the length of an array to be sorted is **less than** this

\* constant, **Quicksort** is used in preference to merge sort.

\*/

private static final int **QUICKSORT\_THRESHOLD** = 286;

/\*\*

\* If the length of an array to be sorted is less than this

\* constant, **insertion sort(插入排序)** is used in preference to **Quicksort**.

\*/

private static final int **INSERTION\_SORT\_THRESHOLD** = 47;

说明：当所需排序范围大于286，则用归并排序；若小于286大于47，则用快排；小于47，使用插入排序。

### 源码

static void **sort**(int[] a, int left, int right,

int[] work, int workBase, int workLen) {

// Use Quicksort on small arrays

if (right - left < **QUICKSORT\_THRESHOLD**) {

sort(a, left, right, true);

return;

}

…

}