2017.11.30

开始对JDK源码的阅读，在算法书上看到提及过这个，而且自己平时偷懒简单排一排的时候，也是使用这个方法进行排序，索性将它看上一看。

/\*\*

\* The minimum array length below which a parallel sorting

\* algorithm will not further partition the sorting task. Using

\* smaller sizes typically results in memory contention across

\* tasks that makes parallel speedups unlikely.

\*/

private static final int MIN\_ARRAY\_SORT\_GRAN = 1 << 13;

开头第一段就是这个，从字面上分析就是定义了一个常量，大小为8192，注释的意思呢就是一个数组的最小长度如果低于一个并行算法要求的值，那么它对划分排序任务没有什么帮助。使用较小的尺寸通常会导致跨任务的内存争用，这使得并行的速度不太可能得到提升。

Arrays.sort()对基本数据类型进行了重载，对于基本数据类型采用的是 Dual-Pivot Quicksort（即双基准快速排序算法/三向切分）

\* This class implements the Dual-Pivot Quicksort algorithm by

\* Vladimir Yaroslavskiy, Jon Bentley, and Josh Bloch. The 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.

\*

从注释中可以看到该快排的性能是优于其他的单基准，也就是最普通的快排方法，它在大多数情况下能保持nlogn的时间复杂度。三向切分快排相较于普通快排的最大优势是它在对很多重复主键排序时的优良性能，因为它设置了两个指针,将数组划分为了<、>、=切分元素的范围。我们知道，归并排序和快排都是分治思想的体现，它们包含着两个对自身的递归调用，而三向切分，可以让重复的主键，不用加入到递归中，因此而提高了性能。

往下则是定义了几种常量，当输入的待排序数组的大小介于某一个值的时候，采用某一种更适合排序方式的“门槛”。正如我们所知道的，快排在小型数组排序上面，相较于其他的例如：插排，归并，计数排序，并不占优。

而优先级是当数组长度小于QUICKSORT\_THRESHOLD的时候，采用三向划分快排，否则使用归并排序。

Leftmost的意思是判定是否从数组的第一位，也就是最左边开始排序。否则从中部任意位置开始。笔者试着打断点调试了一下，因为传入的left参数设置为了0的原因，如果leftmost为false，则会抛出异常。这种情况对已经有序的（包括重复主键和升序/降序）数组的效率较高，因为有个if字句，这种情况会直接return

// Inexpensive approximation of length / 7

// Inexpensive approximation of length / 7

int seventh = (length >> 3) + (length >> 6) + 1;

/\*

\* Sort five evenly spaced elements around (and including) the

\* center element in the range. These elements will be used for

\* pivot selection as described below. The choice for spacing

\* these elements was empirically determined to work well on

\* a wide variety of inputs.

\*/

int e3 = (left + right) >>> 1; // The midpoint

int e2 = e3 - seventh;

int e1 = e2 - seventh;

int e4 = e3 + seventh;

int e5 = e4 + seventh;

接下来定义了一个近似值为数组长度的1/7左右（1/8+1/64+1）,以中心点(起点+终点)/2，间距为前面定义的七分之一，均匀的分成了五份。

// Sort these elements using insertion sort

if (a[e2] < a[e1]) { int t = a[e2]; a[e2] = a[e1]; a[e1] = t; }

if (a[e3] < a[e2]) { int t = a[e3]; a[e3] = a[e2]; a[e2] = t;

if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }

}

if (a[e4] < a[e3]) { int t = a[e4]; a[e4] = a[e3]; a[e3] = t;

if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;

if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }

}

}

if (a[e5] < a[e4]) { int t = a[e5]; a[e5] = a[e4]; a[e4] = t;

if (t < a[e3]) { a[e4] = a[e3]; a[e3] = t;

if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;

if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }

}

}

}

从这里可以看到，结合前面的注释，我们知道，数组会对这几个位置上的元素采用插入排序（如：e2<e1则将两个元素交换位置等等），并且本插入排序是改良后的，并不是单纯的插入，而是尽量的将较大的元素挪到后面，这样减少了移动元素所花费的时间，因为整个数组是比较大的。（>=insertion，采用双基准快排+插入排序，在快排之前采用了插排）

// Pointers

int less = left; // The index of the first element of center part

int great = right; // The index before the first element of right part

在这定义了两个标志位，因为是双基准，所以整个数组被切分成了四份（<1切分元素,介于1,2之间的,暂时未排定的，>2切分元素） 这里定义的两个标志实际上就是中间部分的起点和终点。

如果前面a1...a5各不相等，则采用双基准快排，因为进行过前面的成对插排，所以根据经验知道a2<a4，并且应该属于数组的前1-2%大的数据

下面就是典型的快排操作，用两个while循环移动指针，使得pivot1左边的始终小于1，pivot2右边的始终大于2，并递归调用自身，这里就不再多说。

/\*

\* If center part is too large (comprises > 4/7 of the array),

\* swap internal pivot values to ends.

\*/

如果中间部分即（pivot1<&&<pivot2）大于整个数组的4/7，那么将中间的基准与最后的值交换，意在缩小中间区域的大小。接下来就是常规的两个while循环进行快排标志量的变换。

三向切分快排

接下来使用的三向切分快排，也是笔者在算法第四版上面接触到的一个快排算法的优化变种。这个算法相对于标准的快排算法最大的优势是在处理大量的重复主键上，我们知道快排的最坏时间复杂度是O(N²)，出现在数据有序或接近有序的时候。无疑，大量重复的主键有接近于最坏情况的势头。而三向切分快排，和双基准快排比较类似，只是它将数据区分成了

/\*

\* Partitioning:

\*

\* left part center part right part

\* +----------------------------------------------------------+

\* | == pivot1 | pivot1 < && < pivot2 | ? | == pivot2 |

\* +----------------------------------------------------------+

\* ^ ^ ^

\* | | |

\* less k great

\*

\* Invariants:

\*

\* all in (\*, less) == pivot1

\* pivot1 < all in [less, k) < pivot2

\* all in (great, \*) == pivot2

\*

\* Pointer k is the first index of ?-part.

\*/

等于两个切分元素的区域以及介于两个元素之间+待排元素的区域。

接下来的操作和双基准快排大同小异，就不再赘述。需要注意的一点是在后面的注释提到的东西：

if (a[great] == pivot1)

{ // a[great] < pivot2

a[k] = a[less];

/\*

\* Even though a[great] equals to pivot1, the

\* assignment a[less] = pivot1 may be incorrect,

\* if a[great] and pivot1 are floating-point zeros

\* of different signs. Therefore in float and

\* double sorting methods we have to use more

\* accurate assignment a[less] = a[great].

\*/

a[less] = pivot1;

++less;

如果great==pivot1，我们知道，这个其实就是已经将待排序区域完成排序了，这一趟的排序已经就接近了尾声。

但通过注释我们知道，这个解法并不完全正确。起初笔者也没有想到，但看到注释才恍然大悟：在浮点数排序中，我们需要确保将-0排在+0之前

而在前面的代码中我们知道，当排序完成的隐藏条件，即great==less==pivot1时，因为+0和-0在比较的过程中是相等的，而在Arrays.sort方法针对float型和double型数组的排序中，是要求将-0排在+0之前的，如下图Phase3所示：

接下来就是常规的递归调用切分方法完成整次的快排，在此不再赘述，对Arrays.sort()方法的源码分析暂时告一段落。

分析总结：

1、快排是应用最广的内部排序方法，但为了避免出现它的最坏情况，JAVA在内置的sort方法中做了很多的努力，包括分析比较排序性能，在数组较小的时候采用插入排序等排序方法提高效率，对不同类型的排序数组采用了不同的排序方法，如本次分析的Int型数组较小时采用的插入排序，而char类型的是采用的计数排序。

2、快排有几种变体，在本次的源码分析中提到了标准的快速排序（单基准快排），双基准快速排序以及三向切分快速排序。相对来说，双基准快排性能优于单基准快排，而三向切分在双基准的基础上对大量重复主键的排序做了优化。

3、快排的切分元素的选取是一个智力活，而不是一般做题的那样，就默认将数组的第一个元素作为切分元素，源码在选取切分元素上做了很大的努力。通过将枢轴上5个点（每个点间隔接近数组长度的1/7）进行类似于希尔排序，获得最优的切分元素。（快排的最优情况是每一次切分的过程中，切分都刚好落在将待排元素均匀的分成两份）