

淘宝**JAVA**中间件团队博客

致力于成为中国最强大的**JAVA**技术团队

快速排序及优化

Posted by boyan on 2010-09-08

Go to comments

Leave a comment (1)

quicksort可以说是应用最广泛的排序算法之一，它的基本思想是分治法，选择一个pivot(中轴点)，将小于pivot放在左边，将大于 pivot放在右边，针对左右两个子序列重复此过程，直到序列为空或者只有一个元素。这篇blog主要目的是关注quicksort可能的改进方法，并对 这些改进方法做评测。其目的是为了理解Arrays.sort(int [ ]a)的实现。实现本身有paper介绍。

quicksort一个教科书式的简单实现，采用左端点做pivot(《算法导论》上伪代码是以右端点做pivot)：

```
public void qsort1(int[] a, int p, int r) {
    // 0个或1个元素, 返回
    if (p >= r)
        return;
    // 选择左端点为pivot
    int x = a[p];
    int j = p;
    for (int i = p + 1; i <= r; i++) {
        // 小于pivot的放到左边
        if (a[i] < x) {
            swap(a, ++j, i);
        }
    }
    // 交换左端点和pivot位置
    swap(a, p, j);
    // 递归子序列
    qsort1(a, p, j - 1);
    qsort1(a, j + 1, r);
}
```

quicksort的最佳情况下的时间复杂度O(n logn)，最坏情况下的时间复杂度是O(n^2)，退化到插入排序的最坏情况，平均情况下的平均复杂度接近于最佳情况也就是O(nlog n)，这也是基于比较的排序算法的比较次数下限。

为了对排序算法的性能改进有个直观的对比，我们建立一个测试基准，分别测试随机数组的排序、升序数组的排序、降序数组的排序以及重复元素的数组排序。首先 使用java.util.Arrays.sort建立一个评测基准，注意这里的时间单位是秒，这些绝对时间没有意义，我们关注的是相对值，因此这里我不会 列出详细的评测程序：

算法	随机数组	升序数组	降序数组	重复数组
Arrays.sort	136.293	0.548	0.524	26.822

qsort1对于输入做了假设，假设输入是随机的数组，如果排序已经排序的数组，qsort1马上退化到O(n^2)的复杂度，这是由于选定的pivot每次都会跟剩余的所有元素做比较。它跟Arrays.sort的比较：

算法	随机数组	升序数组	降序数组	重复数组
Arrays.sort	136.293	0.548	0.524	26.822
qsort1	134.475	48.498	141.968	45.244

果然，在排序已经排序的数组的时候，qsort的性能跟Arrays.sort的差距太大了。那么我们能做的第一个优化是什么？答案是将pivot的选择随机化，不再是固定选择左端点，而是利用随机数产生器选择一个有效的位置作为pivot，这就是qsort2：

```
public void qsort2(int[] a, int p, int r) {
    // 0个或1个元素, 返回
    if (p >= r)
        return;
    // 随机选择pivot
    int i = p + rand.nextInt(r - p + 1);
    // 交换pivot和左端点
    swap(a, p, i);
    // 划分算法不变
```



最新日志

- [Java跨语言调用实现方案](#)
- [Selector.wakeup实现笔记](#)
- [JavaOne美国之行-硅谷公司交流篇](#)
- [JavaOne美国之行-大会组织篇](#)
- [JavaOne美国之行-Session篇](#)

分类

- [java基础 \(1\)](#)
- [jvm和java底层 \(1\)](#)
- [产品和系列专题 \(4\)](#)
- [协程 \(1\)](#)
- [团队活动和event \(6\)](#)
- [实习和新生生活 \(1\)](#)
- [未分类 \(1\)](#)
- [模块化应用 \(1\)](#)
- [淘宝开源 \(1\)](#)
- [算法 \(2\)](#)
- [网络编程 \(1\)](#)

最新评论

- [linxuan](#) 发表于 《Java跨语言调用实现方案》
- [benjiam](#) 发表于 《Java跨语言调用实现方案》
- [shiyang](#) 发表于 《关于我们》
- [Longyuliang](#) 发表于 《Selector.wakeup实现笔记》
- [boyan](#) 发表于 《Selector.wakeup实现笔记》

标签

实习 actor aviator guice jamwiki kilim LinkedList osgi peaberry

链接

- [淘宝DBA团队](#)
- [淘宝QA团队](#)
- [淘宝UED团队](#)
- [淘宝开放平台团队](#)
- [淘宝核心系统团队](#)

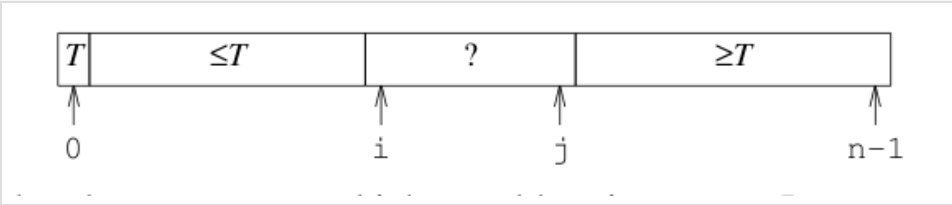
```
int x = a[p];
int j = p;
for (i = p + 1; i <= r; i++) {
    // 小于pivot的放到左边
    if (a[i] < x) {
        swap(a, ++j, i);
    }
}
// 交换左端点和pivot位置
swap(a, p, j);
// 递归子序列
qsort2(a, p, j - 1);
qsort2(a, j + 1, r);
}
```

再次进行测试，查看qsort1和qsort2的对比：

算法	随机数组	升序数组	降序数组	重复数组
qsort1	134.475	48.498	141.968	45.244
qsort2	227.87	19.009	18.597	74.639

从随机数组的排序来看，qsort2比之qsort1还有所下降，这主要是随机数产生器带来的消耗，但是在已经排序数组的排序上面，qsort2有很大进步，在有大量随机重复元素的数组排序上，qsort2却有所下降，主要消耗也是来自随机数产生器的影响。

更进一步的优化是在划分算法上，现在的划分算法只使用了一个索引i，i从左向右扫描，遇到比pivot小的，就跟从p+1开始的位置（由j索引进行递增标志）进行交换，最终的划分点落在了j，然后将pivot调换到j上，再递归排序左右两边子序列。一个更高效的划分过程是使用两个索引i和j，分别从左右两端进行扫描，i扫描到大于等于pivot的元素就停止，j扫描到小于等于pivot的元素也停止，交换两个元素，持续这个过程直到两个索引相遇，此时的pivot的位置就落在了j，然后交换pivot和j的位置，后续的工作没有不同,示意图



改进后的qsort3代码如下：

```
public void qsort3(int[] a, int p, int r) {
    if (p >= r)
        return;

    // 随机选
    int i = p + rand.nextInt(r - p + 1);
    swap(a, p, i);

    // 左索引i指向左端点
    i = p;
    // 右索引j初始指向右端点
    int j = r + 1;
    int x = a[p];
    while (true) {
        // 查找比x大于等于的位置
        do {
            i++;
        } while (i <= r && a[i] < x);
        // 查找比x小于等于的位置
        do {
            j--;
        } while (a[j] > x);
        if (j < i)
            break;
        // 交换a[i]和a[j]
        swap(a, i, j);
    }
    swap(a, p, j);
    qsort3(a, p, j - 1);
    qsort3(a, j + 1, r);
}
```

这里要用do.....while是因为i索引的初始位置是pivot值存储的左端点，而j所在初始位置是右端点之外，因此都需要先移动一个位置才是合法的。查看下qsort2和qsort3的基准测试对比：

算法	随机数组	升序数组	降序数组	重复数组
qsort2	227.87	19.009	18.597	74.639
qsort3	229.44	18.696	18.507	43.428

可以看到qsort3的改进主要体现在了大量重复元素的数组的排序上，这是因为qsort3在遇到跟pivot相等的元素的时候，还是进行停止并交换，而 非跳过；假设遇到相等的元素你不停止，那么这些相等的元素在下次划分的时候需要再次进行比较，比较次数退化到最差情况的O(n^2)，而通过在遇到相等元素的时候停止并交换，尽管增加了交换的次数，但是却避免了所有元素相同情况下最差情况的发生。

改进到这里，回头看看我们做了什么，首先是使用随机挑选pivot替代固定选择，其次是改进了划分算法，从两端进行扫描替代单向查找，并仔细处理元素相同的情况。

插入排序的时间复杂度是O(N^2)，但是在已经排序好的数组上面，插入排序的最佳情况是O(n)，插入排序在小数组的排序上是非常高效的，这给我们一个 提示，在快速排序递归的子序列，如果序列规模足够小，可以使用插入排序替代快速排序，因此可以在快排之前判断数组大小，如果小于一个阈值就使用插入排序， 这就是qsort4:

```
public void qsort4(int[] a, int p, int r) {
    if (p >= r)
        return;

    // 在数组大小小于7的情况下使用快速排序
    if (r - p + 1 < 7) {
        for (int i = p; i <= r; i++) {
            for (int j = i; j > p && a[j - 1] > a[j]; j--) {
                swap(a, j, j - 1);
            }
        }
        return;
    }

    int i = p + rand.nextInt(r - p + 1);
    swap(a, p, i);

    i = p;
    int j = r + 1;
    int x = a[p];
    while (true) {
        do {
            i++;
        } while (i <= r && a[i] < x);
        do {
            j--;
        } while (a[j] > x);
        if (j < i)
            break;
        swap(a, i, j);
    }
    swap(a, p, j);
    qsort4(a, p, j - 1);
    qsort4(a, j + 1, r);
}
```

如果数组大小小于7就使用插入排序，7这个数字完全是经验值。查看qsort3和qsort4的测试比较：

算法	随机数组	升序数组	降序数组	重复数组
qsort3	229.44	18.696	18.507	43.428
qsort4	173.201	7.436	7.477	32.195

qsort4改进的效果非常明显，所有基准测试的结果都取得了明显的进步。qsort4还有一种变形，现在是在每个递归的子序列上进行插入排序，也可以换一种形式，当小于某个特定阈值的时候直接返回不进行任何排序，在递归返回之后，对整个数组进行一次插入排序，这个时候整个数组是由一个一个没有排序的子序列按照顺序组成的，因此插入排序可以很快地将整个数组排序，这个变形的qsort5跟qsort4没有本质上的不同：

```
public void qsort5(int[] a, int p, int r) {
    if (p >= r)
        return;

    // 递归子序列，并且数组大小小于7，直接返回
    if (p != 0 && r!=(a.length-1) && r - p + 1 < 7)
        return;

    // 随机选
    int i = p + rand.nextInt(r - p + 1);
    swap(a, p, i);
```

```

    i = p;
    int j = r + 1;
    int x = a[p];
    while (true) {
        do {
            i++;
        } while (i <= r && a[i] < x);
        do {
            j--;
        } while (a[j] > x);
        if (j < i)
            break;
        swap(a, i, j);
    }
    swap(a, p, j);
    qsort5(a, p, j - 1);
    qsort5(a, j + 1, r);

    // 最后对整个数组进行插入排序
    if (p == 0 && r==a.length-1) {
        for (i = 0; i <= r; i++) {
            for (j = i; j > 0 && a[j - 1] > a[j]; j--) {
                swap(a, j, j - 1);
            }
        }
        return;
    }
}
```

基准测试的结果也证明了qsort4和qsort5是一样的：

算法	随机数组	升序数组	降序数组	重复数组
qsort4	173.201	7.436	7.477	32.195
qsort5	175.031	7.324	7.453	32.322

现在，最大的开销还是随机数产生器选择pivot带来的开销，我们用随机数产生器来选择pivot，是希望pivot能尽量将数组划分得均匀一些，可以选 择一个替代方案来替代随机数产生器来选择pivot，比如三数取中，通过对序列的first、middle和last做比较，选择三个数的中间大小的那一个做pivot，从概率上可以将比较次数下降到12/7 ln(n)。

median-of-three对小数组来说有很大的概率选择一个比较好的pivot，但是对于大数组来说就不足以保证能够选择一个好的pivot， 因此还有个办法是所谓median-of-nine，这个怎么做呢？它是先从数组中分三次取样，每次取三个数，三个样品各取出中数，然后从这三个中数当中 再取出一个中数作为pivot，也就是median-of-medians。取样也不是乱来，分别是在左端点、中点和右端点取样。什么时候采用 median-of-nine去选择pivot，这里也有个数组大小的阈值，这个值也完全是经验值，设定在40，大小大于40的数组使用median-of-nine选择pivot，大小在7到40之间的数组使用three-of-median选择pivot,大小等于7的数组直接选择中数作为pivot，大小小于7的数组则直接使用插入排序，这就是改进后的qsort6，已经非常接近于Arrays.sort的实现：

```

public void qsort6(int[] a, int p, int r) {
    if (p >= r)
        return;

    // 在数组大小小于7的情况下使用快速排序
    if (r - p + 1 < 7) {
        for (int i = p; i <= r; i++) {
            for (int j = i; j > p && a[j - 1] > a[j]; j--) {
                swap(a, j, j - 1);
            }
        }
        return;
    }

    // 计算数组长度
    int len = r - p + 1;
    // 求出中点，大小等于7的数组选择pivot
    int m = p + (len >> 1);
    // 大小大于7
    if (len > 7) {
        int l = p;
        int n = p + len - 1;
        if (len > 40) { // 大数组，采用median-of-nine选择
            int s = len / 8;
            l = med3(a, l, l + s, l + 2 * s); // 取样左端点3个数并得出中数
            m = med3(a, m - s, m, m + s); // 取样中点3个数并得出中数
            n = med3(a, n - 2 * s, n - s, n); // 取样右端点3个数并得出中数
        }
    }
}
```

```
    }
    m = med3(a, l, m, n); // 取中数中的中数
}
// 交换pivot到左端点，后面的操作与qsort4相同
swap(a, p, m);

m = p;
int j = r + 1;
int x = a[p];
while (true) {
    do {
        m++;
    } while (m <= r && a[m] < x);
    do {
        j--;
    } while (a[j] > x);
    if (j < m)
        break;
    swap(a, m, j);
}
swap(a, p, j);
qsort6(a, p, j - 1);
qsort6(a, j + 1, r);
}
```

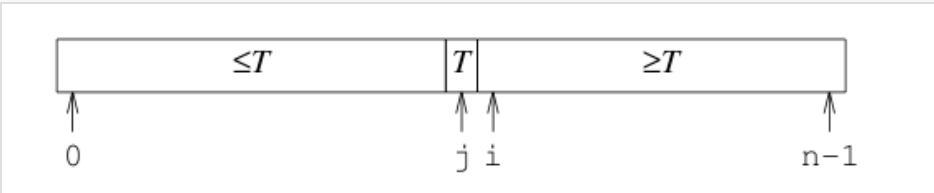
其中的med3函数用于取三个数的中数：

```
private static int med3(int x[], int a, int b, int c) {
    return x[a] < x[b] ? (x[b] < x[c] ? b : x[a] < x[c] ? c : a)
        : x[b] > x[c] ? b : x[a] > x[c] ? c : a;
}
```

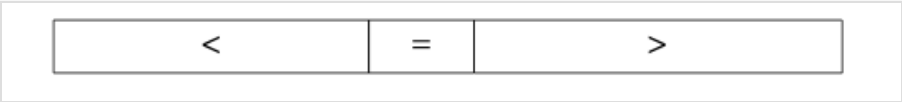
运行基准测试跟qsort4进行比较：

算法	随机数组	升序数组	降序数组	重复数组
qsort4	173.201	7.436	7.477	32.195
qsort6	143.264	0.54	0.836	27.311

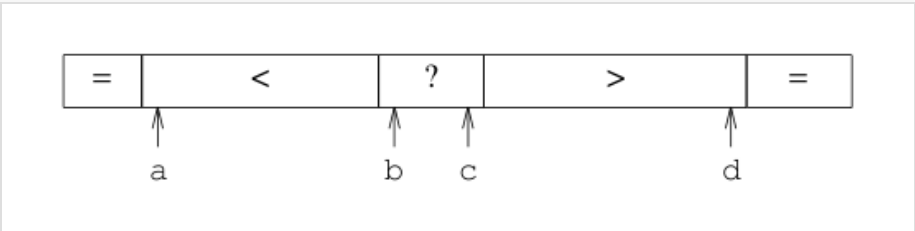
观察到qsort6的改进也非常明显，消除了随机产生器带来的开销，取中数的时间复杂度在O(1)。此时qsort6跟Arrays.sort的差距已经 非常小了。Array.sort所做的最后一个改进是针对划分算法，采用了所谓“split-end”的划分算法，这主要是为了针对equals的元素， 降低equals元素参与递归的开销。我们原来的划分算法是分为两个区域加上一个pivot：



跟pivot equals的元素分散在左右两个子序列里，继续参与递归调用。当数组里的相同元素很多的时候，这个开销是不可忽视的，因此一个方案是把这些相同的元素集中存放到中间这个地方，不参与后续的递归处理，这就是“fat partition”，此时是将数组划分为3个区域：小于pivot，等于pivot以及大于pivot：



但是Arrays.sort采用的却不是“fat partition”，这是因为fat partition的实现比较复杂并且低效，Arrays.sort是将与pivot相同的元素划分到两端，也就是将数组分为了4个区域：



这就是split-end名称的由来，这个算法的实现可以跟qsort3的改进结合起来，同样是进行两端扫描，但是遇到equals的元素不是进行互换，而是各自交换到两端。当扫描结束，还要将两端这些跟pivot equals的元素交换到中间位置，不相同的元素交换到两端，左边仍然是比pivot小的，右边是比pivot大的，分别进行递归的快速排序处理，这样改进后的算法我们成为qsort7：

```
public void qsort7(int[] x, int p, int r) {
    if (p >= r)
        return;

    // 在数组大小小于7的情况下使用快速排序
    if (r - p + 1 < 7) {
        for (int i = p; i <= r; i++) {
            for (int j = i; j > p && x[j - 1] > x[j]; j--) {
                swap(x, j, j - 1);
            }
        }
        return;
    }

    // 选择中数，与qsort6相同。
    int len = r - p + 1;
    int m = p + (len >> 1);
    if (len > 7) {
        int l = p;
        int n = p + len - 1;
        if (len > 40) {
            int s = len / 8;
            l = med3(x, l, l + s, l + 2 * s);
            m = med3(x, m - s, m, m + s);
            n = med3(x, n - 2 * s, n - s, n);
        }
        m = med3(x, l, m, n);
    }

    int v = x[m];

    // a,b进行左端扫描, c,d进行右端扫描
    int a = p, b = a, c = p + len - 1, d = c;
    while (true) {
        // 尝试找到大于pivot的元素
        while (b <= c && x[b] <= v) {
            // 与pivot相同的交换到左端
            if (x[b] == v)
                swap(x, a++, b);
            b++;
        }
        // 尝试找到小于pivot的元素
        while (c >= b && x[c] >= v) {
            // 与pivot相同的交换到右端
            if (x[c] == v)
                swap(x, c, d--);
            c--;
        }
        if (b > c)
            break;
        // 交换找到的元素
        swap(x, b++, c--);
    }

    // 将相同的元素交换到中间
    int s, n = p + len;
    s = Math.min(a - p, b - a);
    vecswap(x, p, b - s, s);
    s = Math.min(d - c, n - d - 1);
    vecswap(x, b, n - s, s);

    // 递归调用子序列
    if ((s = b - a) > 1)
        qsort7(x, p, s + p - 1);
    if ((s = d - c) > 1)
        qsort7(x, n - s, n - 1);
}
```

其中用到了vecswap方法用于批量交换一批数据，将a位置（包括a）之后n个元素与b位置(包括b)之后n个元素进行交换：

```
private static void vecswap(int x[], int a, int b, int n) {
    for (int i = 0; i < n; i++, a++, b++)
        swap(x, a, b);
}
```

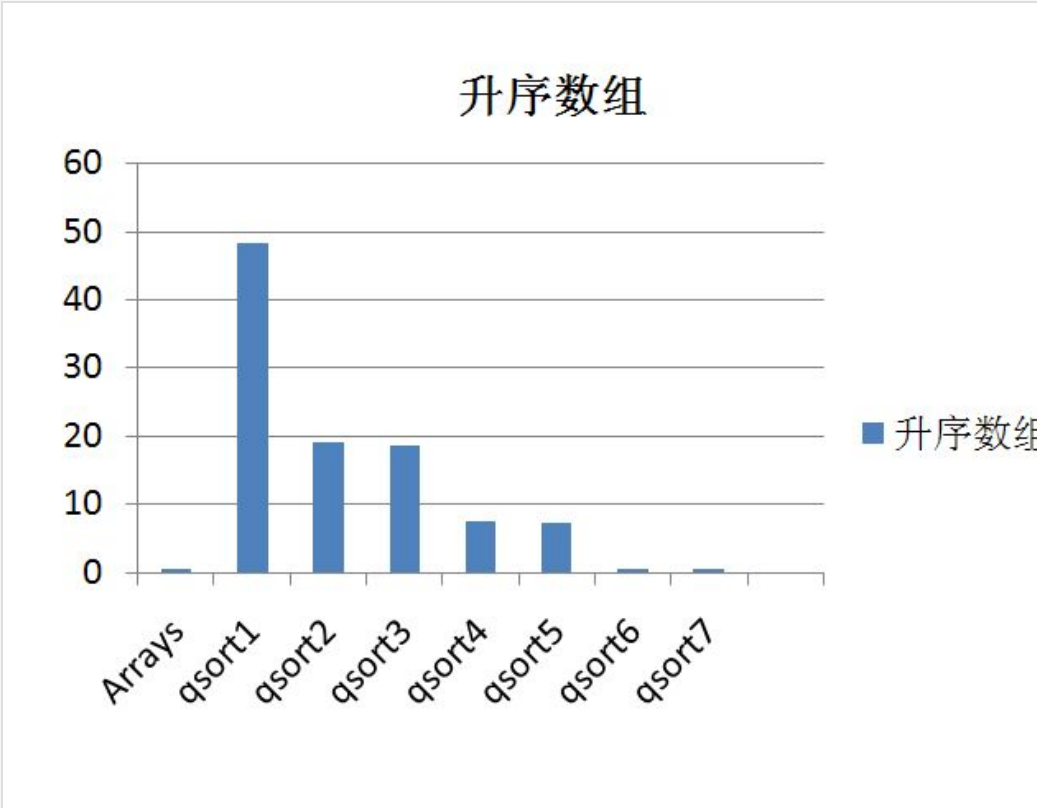
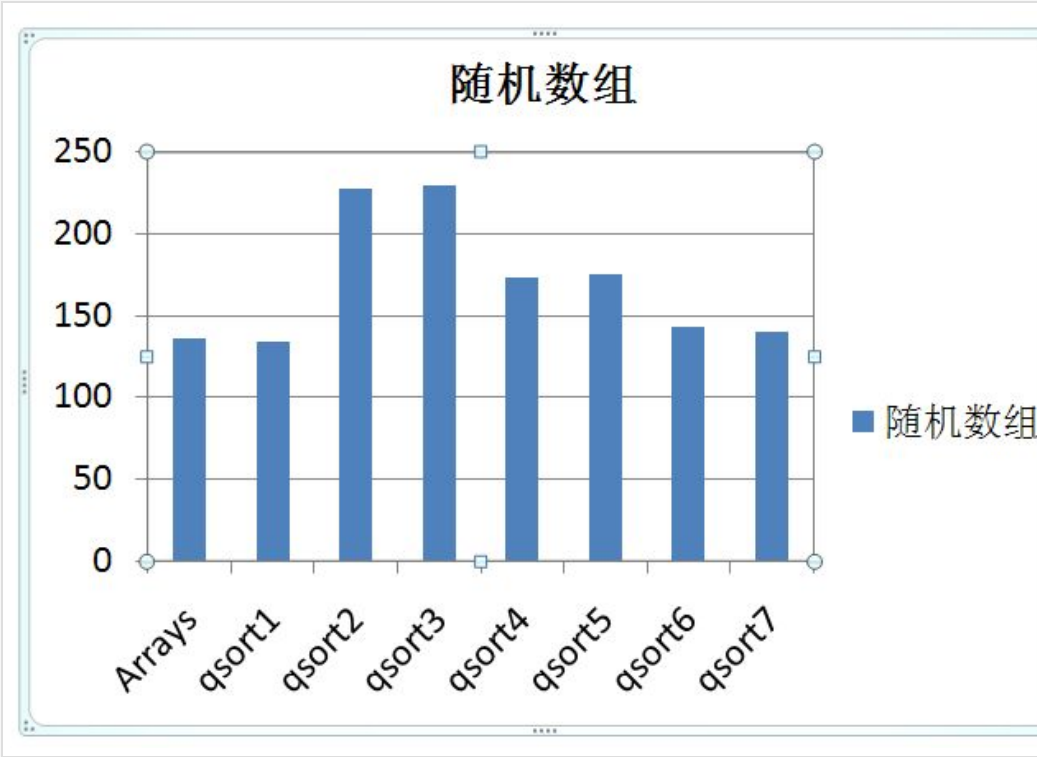
主要是用于划分后将两端与pivot相同的元素交换到中间来。qsort7的实现跟Arrays.sort的实现是一样的，看看split-end划分带来的改进跟qsort6的对比：

算法	随机数组	升序数组	降序数组	重复数组
----	------	------	------	------

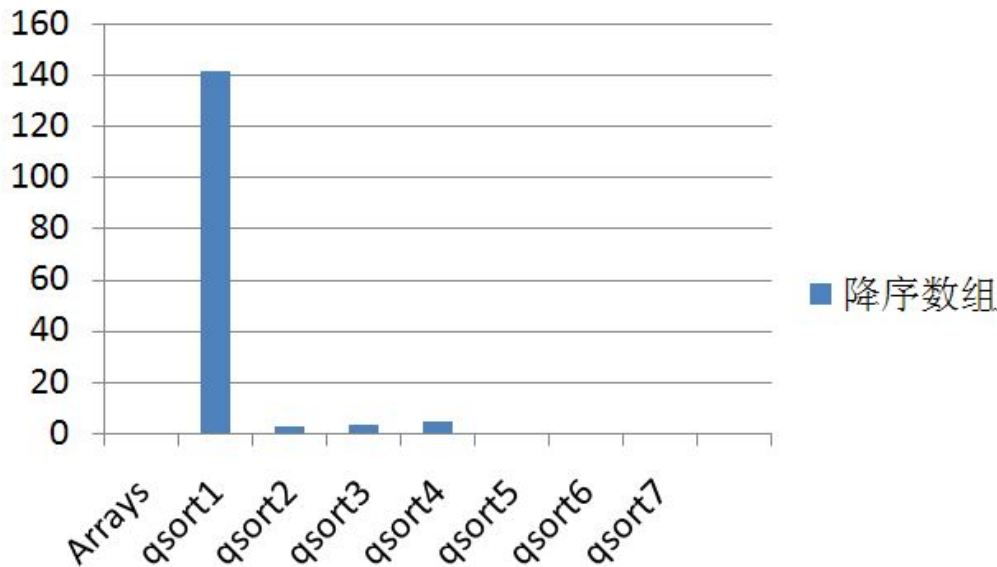
qsort6	143.264	0.54	0.836	27.311
qsort6	140.468	0.491	0.506	26.639

这个结果跟Arrays.sort保持一致。

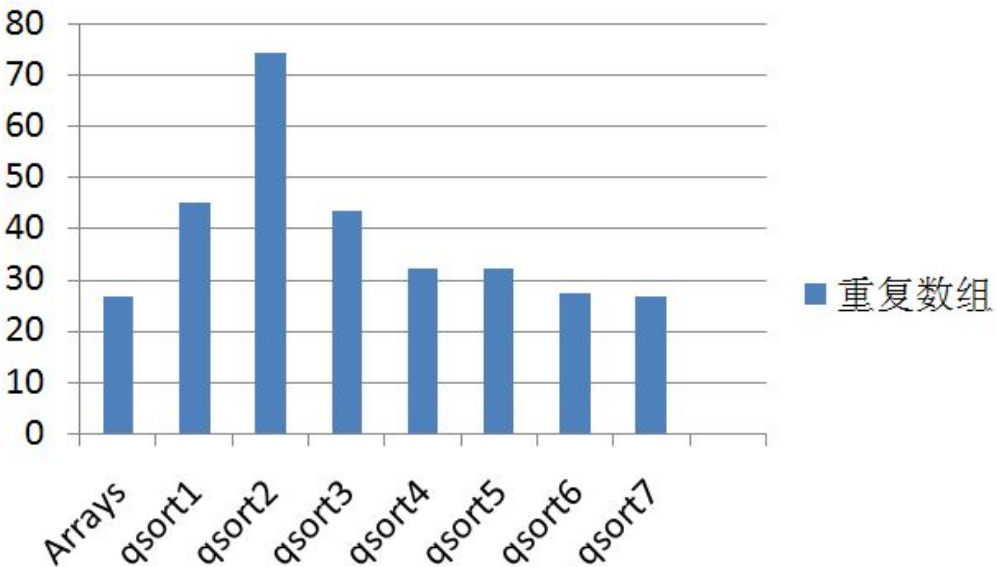
最后给几张7个快排程序的在4种测试中的对比图



降序数组



重复数组



还可以做的优化：

- 1、内联swap和vecswap函数，循环中的swap调用可以展开。
- 2、改进插入排序，这是《编程珠玑》里提到的，减少取值次数。

```
for (int i = p; i <= r; i++) {
    int t = x[i];
    int j = i;
    for (; j > p && x[j - 1] > t; j--) {
        x[j] = x[j - 1];
    }
    x[j] = t;
}
```


- 3、递归可以展开为循环，最后一个递归调用是尾递归调用，很容易展开为循环，左子序列的递归调用就没那么容易展开了。



- 4、尝试更多取样算法来提高选择好的pivot的概率。
- 5、递归处理并行化

📁 算法 📝

← [Aviator](#)（表达式执行引擎）发布1.0.1 厦门之南普陀寺 →

 [Leave a comment](#) 1 Comments.



**wangbing7928** 2010-09-21 @ 20:52 回复

前段时间刚好也研究了下arrays.sort(),发现选取最左(右)边元素，三点取中法除了效率不够高外，当排序大量元素时，这三个办法都可能使得递归深度超出系统的限制。

## Leave a Reply



[ Ctrl + Enter ]