深入理解Spark 2.1 Core（十二）：TimSort 的原理与源码分析

版权声明：本文为博主原创文章，转载请附上原文地址。 http://blog.csdn.net/u011239443/article/details/57406300

http://blog.csdn.net/u011239443/article/details/57406300
在博文《深入理解Spark 2.1 Core （十）： Shuffle Map 端的原理与源码分析 》中我们提到了：

> 使用Sort等对数据进行排序，其中用到了TimSort
> 这篇博文我们就来深入理解下`TimSort`

可视化

推荐先观看下 Youtube 上关于TimSort可视化的视频。对`TimSort`有个感性的了解。

理解timsort

看完视频后也许你会发现`TimSort`和`MergeSort`非常像。没错，这里推荐先阅读关于理解timsort的博文,你就会发现它其实只是对归并排序进行了一系列的改进。其中有一些是很聪明的，而也有一些是相当简单直接的。这些大大小小的改进聚集起来使得算法的效率变得十分的吸引人。

Spark TimSort 源码分析

其实`OpenJDK`在`Java SE 7`的`Arrays`关于`Object`元素数组的`sort`也使用了`TimSort`，而`Spark`的`org.apache.spark.util.collection`包中的用`Java`编写的`TimSort`也和`Java SE 7`中的`TimSort`没有太大区别。

```
1   public void sort(Buffer a, int lo, int hi, Comparator<? super K> c) {
2     assert c != null;
3     // 未排序的数组长度
4     int nRemaining  = hi - lo;
5     // 若数组大小为 0 或者 1
6     // 那么就以及排序了
7     if (nRemaining < 2)
8       return;
9
10    // 若是小数组
11    // 则不使用归并排序
12    if (nRemaining < MIN_MERGE) {
13    // 得到递增序列的长度
14      int initRunLen = countRunAndMakeAscending(a, lo, hi, c);
15      // 二分插入排序
16      binarySort(a, lo, hi, lo + initRunLen, c);
17      return;
18    }
19    // 栈
20    SortState sortState = new SortState(a, c, hi - lo);
21    // 得到最小run长度
22    int minRun = minRunLength(nRemaining);
23    do {
24      // 得到递增序列的长度
25      int runLen = countRunAndMakeAscending(a, lo, hi, c);
26
27      // 若run太小，
28      // 使用二分插入排序
29      if (runLen < minRun) {
30        int force = nRemaining <= minRun ? nRemaining : minRun;
31        binarySort(a, lo, lo + force, lo + runLen, c);
32        runLen = force;
33      }
```

```
34
35      // 入栈
36      sortState.pushRun(lo, runLen);
37      // 可能进行归并
38      sortState.mergeCollapse();
39
40      // 查找下一run的预操作
41      lo += runLen;
42      nRemaining -= runLen;
43    } while (nRemaining != 0);
44
45    // 归并所有剩余的run，完成排序
46    assert lo == hi;
47    sortState.mergeForceCollapse();
48    assert sortState.stackSize == 1;
49  }
```

我们接下来逐个深入的讲解：

countRunAndMakeAscending

```
1   private int countRunAndMakeAscending(Buffer a, int lo, int hi, Comparator<? super K> c) {
2     assert lo < hi;
3     int runHi = lo + 1;
4     if (runHi == hi)
5       return 1;
6
7     K key0 = s.newKey();
8     K key1 = s.newKey();
9
10    // 找到run的尾部
11    if (c.compare(s.getKey(a, runHi++, key0), s.getKey(a, lo, key1)) < 0) {
12    // 若是递减的，找到尾部反转run
13      while (runHi < hi && c.compare(s.getKey(a, runHi, key0), s.getKey(a, runHi - 1, key1)) < 0)
14        runHi++;
15      reverseRange(a, lo, runHi);
16    } else {
17      while (runHi < hi && c.compare(s.getKey(a, runHi, key0), s.getKey(a, runHi - 1, key1)) >= 0)
18        runHi++;
19    }
20    // 返回run的长度
21    return runHi - lo;
22  }
```

binarySort

```
1   private void binarySort(Buffer a, int lo, int hi, int start, Comparator<? super K> c) {
2     assert lo <= start && start <= hi;
3     if (start == lo)
4       start++;
5
6     K key0 = s.newKey();
7     K key1 = s.newKey();
8
9     Buffer pivotStore = s.allocate(1);
10    // 将位置[start,hi)上的元素二分插入排序到已经有序的[lo,start)序列中
11    for ( ; start < hi; start++) {
12      s.copyElement(a, start, pivotStore, 0);
13      K pivot = s.getKey(pivotStore, 0, key0);
14
15      int left = lo;
16      int right = start;
17      assert left <= right;
```

```
      while (left < right) {
        int mid = (left + right) >>> 1;
        if (c.compare(pivot, s.getKey(a, mid, key1)) < 0)
          right = mid;
        else
          left = mid + 1;
      }
      assert left == right;

      int n = start - left;
      // 对插入做简单的优化
      switch (n) {
        case 2:  s.copyElement(a, left + 1, a, left + 2);
        case 1:  s.copyElement(a, left, a, left + 1);
          break;
        default: s.copyRange(a, left, a, left + 1, n);
      }
      s.copyElement(pivotStore, 0, a, left);
    }
  }
```

minRunLength

```
  private int minRunLength(int n) {
    assert n >= 0;
    int r = 0;
    // 这里 MIN_MERGE 为 2 的某次方
    // if n < MIN_MERGE ,
    // then 直接返回 n
    // else if n >= MIN_MERGE 且 n (>1)  为 2 的某次方,
    // then n 的二进制低位第1位 为 0, r |= (n & 1) 一直为 0 , 即返回的是  MIN_MERGE / 2
    // else  r 为之后一次循环的n的二进制低位第1位值 k , 返回的值 MIN_MERGE/2< k < MIN_MERGE
    while (n >= MIN_MERGE) {
      r |= (n & 1);
      n >>= 1;
    }
    return n + r;
  }
```

SortState.pushRun

入栈

```
    private void pushRun(int runBase, int runLen) {
      this.runBase[stackSize] = runBase;
      this.runLen[stackSize] = runLen;
      stackSize++;
    }
```

SortState.mergeCollapse

这部分代码OpenJDK中存在着bug，我们先来看一下Java SE 7是如何实现的：

```
private void mergeCollapse() {
    while (stackSize > 1) {
        int n = stackSize - 2;
        if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
            if (runLen[n - 1] < runLen[n + 1])
                n--;
            mergeAt(n);
        } else if (runLen[n] <= runLen[n + 1]) {
            mergeAt(n);
```

```
        } else {
            break;
        }
    }
}
```

我们来举个例子：
当栈中的片段长度为：

> 120, 80, 25, 20

我们插入长度的30的片段，由于`25 < 20 + 30` 并且 `25 < 30`，所以得到：

> 120, 80, 45, 30

现在，由于`80 > 45 + 30` 并且 `45 > 30`,于是合并结束。但这并不完全符合根据不变式的重存储，因为`120 < 80 + 45`!

更多细节可以参阅相关博文， Spark 也对此bug 进行了修复,修复后的代码如下：

```
    private void mergeCollapse() {
      while (stackSize > 1) {
        int n = stackSize - 2;
        if ( (n >= 1 && runLen[n-1] <= runLen[n] + runLen[n+1])
          || (n >= 2 && runLen[n-2] <= runLen[n] + runLen[n-1])) {
          if (runLen[n - 1] < runLen[n + 1])
            n--;
        } else if (runLen[n] > runLen[n + 1]) {
          break;
        }
        mergeAt(n);
      }
    }
```

SortState. mergeAt

```
    private void mergeAt(int i) {
      assert stackSize >= 2;
      assert i >= 0;
      assert i == stackSize - 2 || i == stackSize - 3;

      int base1 = runBase[i];
      int len1 = runLen[i];
      int base2 = runBase[i + 1];
      int len2 = runLen[i + 1];
      assert len1 > 0 && len2 > 0;
      assert base1 + len1 == base2;

      // 若 i 是从栈顶数第3个位置
      // 则 将栈顶元素 赋值到 从栈顶数第2个位置
      runLen[i] = len1 + len2;
      if (i == stackSize - 3) {
        runBase[i + 1] = runBase[i + 2];
        runLen[i + 1] = runLen[i + 2];
      }
      stackSize--;

      K key0 = s.newKey();

       // 从 run1 中找到 run2的第1个元素的位置
       // 在这之前的run1的元素都可以被忽略
```

```
      int k = gallopRight(s.getKey(a, base2, key0), a, base1, len1, 0, c);
      assert k >= 0;
      base1 += k;
      len1 -= k;
      if (len1 == 0)
        return;

      // 从 run2 中找到 run1的最后1个元素的位置
      // 在这之后的run2的元素都可以被忽略
      len2 = gallopLeft(s.getKey(a, base1 + len1 - 1, key0), a, base2, len2, len2 - 1, c);
      assert len2 >= 0;
      if (len2 == 0)
        return;

      // 归并run
      // 使用 min(len1, len2) 长度的临时数组
      if (len1 <= len2)
        mergeLo(base1, len1, base2, len2);
      else
        mergeHi(base1, len1, base2, len2);
    }
```

SortState. gallopRight

```
  // key: run2的第1个值
  // a: 数组
  // base:  run1的起始为位置
  // len:  run1的长度
  // hint:  从run1的hint位置开始查找，这里我们传入的值为 0
   private int gallopRight(K key, Buffer a, int base, int len, int hint, Comparator<? super K> c) {
     assert len > 0 && hint >= 0 && hint < len;

     // 对二分查找的优化:
     // 我们要从 run1中 截取出这样一段数组
     // lastOfs = k+1
     // ofs = 2×k+1
     // run1[lastOfs] <= key <= run1[ofs]
     // 即在[lastOfs,ofs],做二分查找
     int ofs = 1;
     int lastOfs = 0;
     K key1 = s.newKey();

     // 若 run2的第1个值 < run1的第1个值
     // 其实我知道，可以直接返回 0
     // 但这里还是走了完整的算法流程
     if (c.compare(key, s.getKey(a, base + hint, key1)) < 0) {
       // maxOfs = 1
       int maxOfs = hint + 1;
       // 不进入循环
       while (ofs < maxOfs && c.compare(key, s.getKey(a, base + hint - ofs, key1)) < 0) {
         lastOfs = ofs;
         ofs = (ofs << 1) + 1;
         if (ofs <= 0)
           ofs = maxOfs;
       }
       // 不进入
       if (ofs > maxOfs)
         ofs = maxOfs;

       // tmp = 0
       int tmp = lastOfs;
       // lastOfs = -1
       lastOfs = hint - ofs;
       // ofs = 0
```

```
        ofs = hint - tmp;
      } else {
      // 这种情况下，算法才会发挥真正的作用
      // maxOfs = len
        int maxOfs = len - hint;
        while (ofs < maxOfs && c.compare(key, s.getKey(a, base + hint + ofs, key1)) >= 0) {
         // 更新 lastOfs 和 ofs
          lastOfs = ofs;
          ofs = (ofs << 1) + 1;
          // 防止溢出
          if (ofs <= 0)
            ofs = maxOfs;
        }
        if (ofs > maxOfs)
          ofs = maxOfs;

        // 这里都不会变
        lastOfs += hint;
        ofs += hint;
      }
      assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;

      // 进行二分查找
      lastOfs++;
      while (lastOfs < ofs) {
        int m = lastOfs + ((ofs - lastOfs) >>> 1);

        if (c.compare(key, s.getKey(a, base + m, key1)) < 0)
        // key < a[b + m]
          ofs = m;
        else
        // a[b + m] <= key
          lastOfs = m + 1;
      }
      assert lastOfs == ofs;
      return ofs;
    }
```

`gallopLeft`和上述代码类似，就不再做讲解。

SortState. mergeLo

```
    private void mergeLo(int base1, int len1, int base2, int len2) {
      assert len1 > 0 && len2 > 0 && base1 + len1 == base2;

      // 使用 min(len1, len2) 长度的临时数组
      // 这里 len1 会较小
      Buffer a = this.a;
      Buffer tmp = ensureCapacity(len1);
      s.copyRange(a, base1, tmp, 0, len1);

     // tmp （run1） 上的指针
      int cursor1 = 0;
     // run2 上的指针
      int cursor2 = base2;
     // 合并结果 上的指针
      int dest = base1;

      // Move first element of second run and deal with degenerate cases
      // 优化：
      // 注意： run2 的第一个元素比 run1的第一个元素小
      //       run1 的最后一个元素 比 run2的最后一个元素大
      // 把 run2 的第1个 元素复制到 最终结果的第1个位置
      s.copyElement(a, cursor2++, a, dest++);
```

```
23        if (--len2 == 0) {
24        // 若 len2 为 1
25        // 直接 把 run1 拷贝到 最终结果中
26          s.copyRange(tmp, cursor1, a, dest, len1);
27          return;
28        }
29        if (len1 == 1) {
30        // 若 len1 为 1
31        // 把 run2 剩余的部分 拷贝到 最终结果中
32        // 再把 run1 拷贝到 最终结果中
33          s.copyRange(a, cursor2, a, dest, len2);
34          s.copyElement(tmp, cursor1, a, dest + len2);
35          return;
36        }
37
38        K key0 = s.newKey();
39        K key1 = s.newKey();
40
41        Comparator<? super K> c = this.c;
42        // 对归并排序的优化：
43        int minGallop = this.minGallop;
44        outer:
45        while (true) {
46        // 主要思想为 使用 count1 count2 对插入进行计数
47          int count1 = 0;
48          int count2 = 0;
49
50          do {
51          // 归并
52            assert len1 > 1 && len2 > 0;
53            if (c.compare(s.getKey(a, cursor2, key0), s.getKey(tmp, cursor1, key1)) < 0) {
54              s.copyElement(a, cursor2++, a, dest++);
55              count2++;
56              count1 = 0;
57              if (--len2 == 0)
58                break outer;
59            } else {
60              s.copyElement(tmp, cursor1++, a, dest++);
61              count1++;
62              count2 = 0;
63              if (--len1 == 1)
64                break outer;
65            }
66            // 若某个run连续拷贝的次数超过minGallop
67            // 退出循环
68          } while ((count1 | count2) < minGallop);
69
70           // 我们认为若某个run连续拷贝的次数超过minGallop，
71           // 则可能还会出现更若某个run连续拷贝的次数超过minGallop
72           // 所有需要重新进行类似于mergeAt中的操作，
73           // 截取出按“段”进行归并
74           // 直到 count1 或者 count2 < MIN_GALLOP
75          do {
76            assert len1 > 1 && len2 > 0;
77            count1 = gallopRight(s.getKey(a, cursor2, key0), tmp, cursor1, len1, 0, c);
78            if (count1 != 0) {
79              s.copyRange(tmp, cursor1, a, dest, count1);
80              dest += count1;
81              cursor1 += count1;
82              len1 -= count1;
83              if (len1 <= 1) // len1 == 1 || len1 == 0
84                break outer;
85            }
86            s.copyElement(a, cursor2++, a, dest++);
```

```
          if (--len2 == 0)
            break outer;

          count2 = gallopLeft(s.getKey(tmp, cursor1, key0), a, cursor2, len2, 0, c);
          if (count2 != 0) {
            s.copyRange(a, cursor2, a, dest, count2);
            dest += count2;
            cursor2 += count2;
            len2 -= count2;
            if (len2 == 0)
              break outer;
          }
          s.copyElement(tmp, cursor1++, a, dest++);
          if (--len1 == 1)
            break outer;
          minGallop--;
        } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
        // 调整 minGallop
        if (minGallop < 0)
          minGallop = 0;
        minGallop += 2;
      }
      // 退出 outer 循环
      this.minGallop = minGallop < 1 ? 1 : minGallop;

      // 把尾部写入最终结果
      if (len1 == 1) {
        assert len2 > 0;
        s.copyRange(a, cursor2, a, dest, len2);
        s.copyElement(tmp, cursor1, a, dest + len2);
      } else if (len1 == 0) {
        throw new IllegalArgumentException(
            "Comparison method violates its general contract!");
      } else {
        assert len2 == 0;
        assert len1 > 1;
        s.copyRange(tmp, cursor1, a, dest, len1);
      }
    }
```

mergeHi与上述类似，就不再讲解。

SortState.mergeForceCollapse

```
    private void mergeForceCollapse() {
    // 将所有的run合并
      while (stackSize > 1) {
        int n = stackSize - 2;
        // 若第3个run 长度 小于 栈顶的run
        // 先归并第2,3个 run
        if (n > 0 && runLen[n - 1] < runLen[n + 1])
          n--;
        mergeAt(n);
      }
    }
```

总结

Spark TimSort 中 对MergeSort大致有一下几点：

- 元素：不像MergeSort惰性的有原来的长度为1，再由归并自动的生成新的归并元素。TimSort是预先按连续递增（或者将连续递减的片段反转）的片段作为一个归并元素，即run。

- 插入排序：若是长度小的run， `TimSort` 会改用二分的 `InsertSort` 以及对再它进行一些小优化，而不使用 `MergeSort`
- 归并的时机： `MergeSort` 的归并时机是定死的，而 `TimSort` 中的时机
  是 `(n >= 1 && runLen[n-1] <= runLen[n] + runLen[n+1])|| (n >= 2 && runLen[n-2] <= runLen[n] + runLen[n-1])` 。以及，若从栈顶开始第3个 `run` 长度 小于 栈顶的 `run` ，先归并第2,3个 `run` 。
- 截取出需要归并的片段：run1是头部和 `run2` 的尾部都是会有可以不用进行归并的部分。 如 `TimSort` 从 `run1` 中 截取出这样一段片段： `lastOfs = k+1` ， `ofs = 2×k+1` ， `run1[lastOfs] <= key <= run1[ofs]` 。再从该片段上进行二分查找，得到 `run1` 中需要归并的起始位置
- 归并的优化：对 `run` 长度为1时，进行了小优化。实现了按单个值和按片段归并的协同。