MOTOROLA

Optimizing a Java ME Application Part II

- RMS Sorting

October 26, 2006

Technical Article

Optimizing a Java ME Application

Part 2: RMS Sorting

By

MOTODEV Staff

Sorting is very important in J2ME game development, especially for those games that need to assign score rank. In our former technical articles, we have examined Record Management Systems (RMS), which provides persistent storage for mobile applications. And in this document, we are going to discuss how can improve the performance of sorting in RMS.

First, we will introduce the `RecordEnumeration` class and `RecordComparator`, which is an interface for sorting provided in RMS. Then, we will compare the performance of three different sorting methods to improve the performance of sorting in RMS.

The `RecordEnumeration` Class

Before we talk about how to sort in RMS, we should introduce the `RecordEnumeration` Class. This class is very similar to the `java.util.Enumeration` in J2SE. It offers developers an easy way to access the records in the RecordStore.

```
public RecordEnumeration enumerateRecords (
                                            RecordFilter filter,
                                            RecordComparator,
                                            Boolean keepUpdated
                                        )
```

The first parameter is `RecordFilter`. If it is not null, a subset of the records can be chosen that match the supplied filter, so it can be used for providing search capabilities. The second parameter is `RecordComparator`, allowing the enumerator to index the records in an order determined by the comparator. We will give more details about it in the next section. The last parameter allows the developer to decide whether to update the `RecordEnumeration`

interface with the RecordStore. Please note, though, that setting this value as true may have serious performance impacts on your MIDlet. The performance hit occurs because `RecordEnumeration` does not have a copy of the RecordStore. Instead, it gets the data from RecordStore whenever we use it.

This interface has several methods, and one of the most important ones is `nextRecord()`. This method returns a copy of the next record in this enumeration where next is defined by the comparator and/or filters supplied in the constructor of this enumerator. With it, we can sort the data in RecordStores very easily.

The `RecordComparator` Interface

MIDlets implement the `RecordComparator` interface to define a comparator, which can compare two records and sort them in a specified order. For example, we implement the `compare (byte[] rec1, byte[] rec2)` method here to sort the records:

```
public int compare(byte[] data1, byte[] data2) {
	try {
		int iScore1 = (data1[0]&0x0000ff);
		iScore1 = (iScore1<<8)+(data1[1]&0x0000ff);
		int iScore2 = (data2[0]&0x0000ff);
		iScore2 = (iScore2<<8)+(data2[1]&0x0000ff);

		if (iScore1 > iScore2) {
			return RecordComparator.FOLLOWS;
		}else if (iScore1 < iScore2) {
			return RecordComparator.PRECEDES;
		}else {
			return RecordComparator.EQUIVALENT;
		}
	} catch (Exception e) {
		return 99;
	}
}
```

Note:

It seems many developers like to implement the `RecordComparator` interface with an inner class. However, if you do not maintain any state information in your inner class, it is expensive to have inner classes that implement only one interface. So, try to avoid using inner classes in your code.

How about the speed?

In the following sample code, we will create a RecordStore and sort the records in it.

To get started, we create three methods for sorting. `Method sortWithComp( )` uses the `RecordEnumeration` and `RecordComparator` interfaces to implement the sorting. Then, the method `sortWithBubble( )` uses `RecordEnumeration` and bubble sorting and the method `sortWithBubble2( )` uses `RecordStore.getRecord(int ID)` and bubble sorting.

Code:

1. `sortWithComp()`

```
private void sortWithComp(){

	display.setCurrent(f);
	f.removeCommand(cmd_sort);

	startTime_Comp = System.currentTimeMillis();
	comparator = new RMSComparator();
	try {
		recordEnumComp = rsScores.enumerateRecords(null, comparator, false);
	} catch (RecordStoreNotOpenException e1) {
		e1.printStackTrace();
	}

	while (recordEnumComp.hasNextElement()) {
		try {
			byte[] tmpdata = recordEnumComp.nextRecord();
			int idata = (tmpdata[0]&0x0000ff);
			idata = (idata<<8)+(tmpdata[1]&0x0000ff);
```

```
            } catch (InvalidRecordIDException e) {
                  e.printStackTrace();
            } catch (RecordStoreException e) {
                  e.printStackTrace();
            }
      }


       endTime_Comp = System.currentTimeMillis();
       System.out.println("With Comparator: Pass " + (endTime_Comp - startTime_Comp)
+ " Milliseconds" );
       f.append("With Comparator: Pass " + (endTime_Comp - startTime_Comp) + "
Milliseconds" );
}
```

2. sortWithBubble()

```
private void sortWithBubble(){

       display.setCurrent(f);

       startTime_Bubble = System.currentTimeMillis();
       try {
             recordEnum = rsScores.enumerateRecords(null, null, false);
       } catch (RecordStoreNotOpenException e1) {
             e1.printStackTrace();
       }

       //get record data
       int[] iSortedData = new int[recordEnum.numRecords()];
       int index = 0;

       startTime_Bubble = System.currentTimeMillis();
       while (recordEnum.hasNextElement()) {
             try {
```

```
                    byte[] tmpdata = recordEnum.nextRecord();
                    iSortedData[index] = (tmpdata[0] & 0x0000ff);
                    iSortedData[index] = (iSortedData[index] << 8) + (tmpdata[1] &
0x0000ff);
                    index+=1;
             } catch (InvalidRecordIDException e) {
                    e.printStackTrace();
             } catch (RecordStoreException e) {
                    e.printStackTrace();
             }
       }

       //Sorting
       for (int i = 0; i < iSortedData.length - 1; i++) {
             for (int j = i + 1; j < (iSortedData.length - 1); j++) {
                    if (iSortedData[i] > iSortedData[j]) {
                           // swap
                           int temp = iSortedData[i];
                           iSortedData[i] = iSortedData[j];
                           iSortedData[j] = temp;
                    }
             }
       }


       endTime_Bubble = System.currentTimeMillis();
       System.out.println("With Bubble: Pass " + (endTime_Bubble - startTime_Bubble)
+ " Milliseconds" );
       f.append("With Bubble: Pass " + (endTime_Bubble - startTime_Bubble) + "
Milliseconds" );
}
```

3. sortWithBubble2()

```
private void sortWithBubble2(){
	display.setCurrent(f);

	startTime_Bubble2 = System.currentTimeMillis();
	int lastID=0;
	try {
		lastID = rsScores.getNextRecordID()-1;
	} catch (RecordStoreNotOpenException e1) {
		e1.printStackTrace();
	} catch (RecordStoreException e1) {
		e1.printStackTrace();
	}

	//get record data
	int[] iSortedData = new int[lastID];
	int index = 0;

	for (int i=0; i<lastID; i++) {
		try {
			byte[] tmpdata = rsScores.getRecord(i+1);
			iSortedData[index] = (tmpdata[0] & 0x0000ff);
			iSortedData[index] = (iSortedData[index] << 8) + (tmpdata[1] &
0x0000ff);
			index+=1;
		} catch (InvalidRecordIDException e) {
			e.printStackTrace();
		} catch (RecordStoreException e) {
			e.printStackTrace();
		}
	}

	//Sorting
	for (int i = 0; i < iSortedData.length - 1; i++) {
		for (int j = i + 1; j < (iSortedData.length - 1); j++) {
			if (iSortedData[i] > iSortedData[j]) {
				// swap
```

```
                        int temp = iSortedData[i];
                        iSortedData[i] = iSortedData[j];
                        iSortedData[j] = temp;
                    }
                }
            }

        endTime_Bubble2 = System.currentTimeMillis();
        System.out.println("With Bubble2: Pass " + (endTime_Bubble2 -
startTime_Bubble2) + " Milliseconds" );
        f.append("With Bubble2: Pass " + (endTime_Bubble2 - startTime_Bubble2) + "
Milliseconds" );
}
```

In order to test the speed in different conditions, we create several groups of records (scores ranging from 0~99) with different numbers in the RecordStore. The elapsed time of these three methods is shown below:

Table 1: Elapsed times on the Motorola E680_ E680i emulator

RecordNumber	sortWithComp( )	sortWithBubble( )	sortWithBubble2( )
50	200	40	20
100	981	120	111
200	4096	541	451
300	9914	1112	1071
400	30693	3104	2925
500	31164	3114	2985

Table 2: Elapsed times on the Motorola E680i handset

RecordNumber	sortWithComp( )	sortWithBubble( )	sortWithBubble2( )
50	877	193	100
100	3906	221	210
200	15801	571	550
300	39742	1212	1171
400	120936	7609	7377

500	116719	7404	7184

Table 3: Elapsed times on the Motorola ROKR E2 handset

RecordNumber	sortWithComp( )	sortWithBubble( )	sortWithBubble2( )
50	240	22	22
100	510	66	55
200	1020	124	122
300	1796	200	207
400	2279	227	299
500	3305	382	391

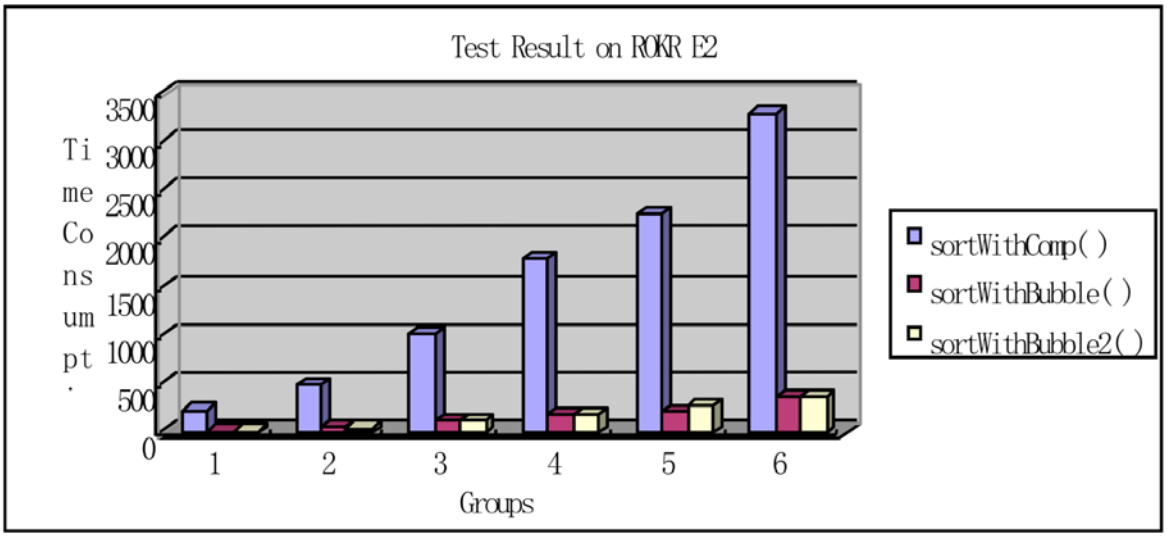

Figure 1: Test results on the Motorola ROKR E2 handset

Conclusion

From the data reflected in Figure 1, we find that using the RecordComparator interface really consumes time, though it is easy to use and makes code simple. And the speed of the method `sortWithBubble()` and `sortWithBubble2()` are almost the same. So, you may only want to implement these interfaces when the comparison is complex, or implement the sorting with your own method to improve the speed of your code.

Reference

[1] Programming Wireless Devices with the Java 2 Platform Micro Edition, Second Edition (Book).

[2] MIDP Database Programming Using RMS: a Persistent Storage for MIDlets, By Qusay Mahmoud, http://developers.sun.com/techtopics/mobility/midp/articles/persist/

[3] Working With The RMS, by John Muchow, http://www.microjava.com/articles/techtalk/rms

[4] J2ME record management store, by Soma Ghosh, Entigo http://www-128.ibm.com/developerworks/wireless/library/wi-rms/