RMS – RECORD MANAGEMENT SYSTEM

By Fabrizio Russo
25/11/2002

www.frusso.it/j2me

Ogni applicazione che si rispetti, e le MIDlet non fanno eccezione, hanno bisogno di poter conservare i propri dati. Purtroppo l' API MIDP 1.0 non consente di accedere a risorse quali i file system, quindi, per poter memorizzare le proprie informazioni bisogna ricorrere a qualcos'altro.

Per risolvere questo tipo di problema è stato introdotto l' RMS, cioè la gestione dei record a livello di MIDP.

L'API RMS consente la gestione di record sequenziali. E' un API in grado di fornire accesso a singoli record e di effettuare anche delle ricerche in base a filtri e a ordinamenti, ma non ha niente a che vedere con JDBC, infatti mentre JDBC ha l'obiettivo di consentire un collegamento a tutti i tipi di database relazionali, gestendo concorrenza e transazionalità, RMS ha come unico obiettivo quello di fornire un sistema di persistenza a livello di Suite Midlet.

L'RMS utilizza la memoria non volatile del dispositivo per memorizzare le informazioni. Le informazioni vengono memorizzate in una sorta di database "flat" paragonabile ad un foglio a righe numerate dove ad ogni riga è associato un identificativo unico gestito dal sistema paragonabile ad una "chiave primaria".

ID	Array di byte
1	array di byte del Record 1
2	array di byte del Record 2
...	...
N	array di byte del Record N

Contrariamente a JDBC non è possibile definire tipi di campi per il record ma si ha a disposizione un array di byte su cui memorizzare qualsiasi tipo di informazione.

Questo database "flat" viene chiamato `RecordStore`. Ogni Suite di MIDlet può creare zero o più record store ed ogni record store può contenere zero o più record. Teoricamente non ci sono limiti alle dimensioni di questi "database" tranne, ovviamente, la memoria volatile messa a disposizione dal terminale MIDP (telefonino).
La vita dei record store è legata alla suite a cui appartiene. Ogni volta che viene cancellata una suite tutti i record store (indipendentemente dalla MIDlet che li ha creati) vengono perduti.

I MIDlet che fanno parte di una Suite possono accedere a tutti i recordStore creati dalle MIDlet facenti parti della Suite. Questo significa che il recordStore è un ottimo candidato per memorizzare le informazioni comuni alle applicazioni e per poter scambiare informazioni tra MIDlet. Ogni recordStore è identificato all'interno della suite da un nome univoco che può essere lungo al più 32 caratteri (16 caratteri Unicode) ed inoltre il nome è sensibile alle maiuscole/minuscole

RecordStore

L'oggetto `javax.microedition.rms.RecordStore` **rappresenta il fulcro su cui ruota tutta l'architettura RMS (In effetti è l'unica classe del package** `javax.microedition.rms`**). Questa classe metta a disposizione un'insieme di metodi per poter operare sui singoli record. La classe** `javax.microedition.rms.RecordStore` **non ha un costruttore e quindi per poter ottenere un oggetto di tipo RecordStore è necessario invocare un metodo statico della classe RecordStore.**

```
public static RecordStore openRecordStore(String recordStoreName, boolean
createIfNecessary)
```

Questo metodo apre (e se è il caso, crea) un record store associato con la Suite a cui appartiene la MIDlet. Il nome del recordStore da aprire viene passato come primo argomento, mentre il secondo argomento indica l'azione da intraprendere nel caso in cui il recordStore che si intende aprire non esista.

Altri metodi della classe RecordStore sono:

Classe javax.microedition.rms.RecordStore

`void closeRecordStore()`
Chiude il record Store

`static void deleteRecordStore(String recordStoreName)`
Cancella il recordStore

`static String[] listRecordStores()`
Elenco dei recordStore presenti all'interno della MIDlet

`int addRecord(byte data[], int offset, int numBytes)`
Aggiunge un record in coda al recordStore. E' necessario specificare l'array di bytes da memorizzare, indice di partenza all'interno dell'array e numero di byte da memorizzare

`void deleteRecord(int recordID)`
Cancella l'i-esimo record dal recordStore

`byte[] getRecord(int recordID)`
Ottiene l'array di byte dell'i-esimo record

`int getRecord(int recordID, byte buffer[], int offset)`
Ottiene l'array di byte dell'i-esimo record. L'array di byte viene copiato nel buffer di byte passato come argomento che deve essere grande abbastanza per poterlo contenere.

`int getRecordSize(int recordID)`
Restituisce la dimensione dell'array memorizzato nell'i-esima posizione

`int getNextRecordID()`
Restituisce l'ID del prossimo record

`int getNumRecord()`
Numero di record presenti nel recordStore

`long getLastModified()`
Data (in millisecondi) dell'ultima modifica apportata al recordStore

`int getVersion()`
Numero di versione del recordStore

`String getName()`
Nome del recordStore

`int getSize()`
Restituisce lo spazio occupato, in bytes, del recordStore

`int getSizeAvailable()`
Restituisce lo spazio (in bytes) ancora disponibile per il recordStore
`void addRecordListener(RecordListener listener)`
Aggiunge un ascoltatore di eventi sul recordStore
`void removeRecordListener(RecordListener listener)`
Cancella un ascoltatore di eventi sul recordStore

Una volta aperto un recordStrore è possbile leggere un record attraverso il metodo

```
public int addRecord(byte data[], int offset, int numBytes)
```

che aggiunge una array di byte (*data*) in coda al recordStore (una sorta di append). Gli altri due parametri servono ad indicare il numero di byte (*numBytes*) che si intende copiare nel recordStore a partire della posizione indicata in offset. Dato che l'operazione tipica è quella di memorizzare una Stringa lo stralcio di codice necessario è il seguente:

```
RecordStore rs = RecordStore.openRecordStore("MioDB", true);
String s = "Record da salvare";
int pos = rs.addRecord(s.getBytes(), 0, s.length());
```

E' necessario ricordarsi sempre che prima di poter inserire un record è necessario aprire il recordStore su cui si vuole inserire (pena il sollevamento di un'eccezione). Inoltre, dato che si vuole inserire una Stringa, il metodo `getBytes()` restituisce l'array di byte di cui è formata la stringa (sembra messo a posta).

L'intero restituito dal metodo `addRecord` è molto importante, quell'intero rappresenta l' ID assegnato al record appena inserito. E' molto importante conservare quel valore visto che l'unico modo per poter operare su quel record è attraveso il suo ID.

Supponiamo infatti di voler modificare il valore del record appena inserito. Il metodo per poter modificare il valore di un record è il seguente:

```
public void setRecord(inr recordID, byte newData[], int offset, int numBytes);
```

Questo metodo ha quasi gli stessi parametri del metodo addRecord in più è necessario l'ID del record su cui effettuare la modifica.

Una volta aperto un recordStore, inserito dei dati è necessario poterlo chiudere (ed eventualmente cancellare). Il metodo necessario alla chiusura di un recordStore è il metodo `closeRecordStore()`, mentre il metodo per poter rimuovere definitivamente un recordStore è il metodo `deleteRecordStore()`.
Mentre il metodo `closeRecordStore` non è statico e quindi va invocato sul recordStore da chiudere, il metodo `deleteRecordStore` lo è e va quindi indicato il nome del recordStore da cancellare. In effetti il metodo di rimozione non poteva che essere statico, altrimenti, dopo la rimozione, in che stato avremmo trovato la variabile assegnata al recordStore ?

Riepilogando vediamo uno stalcio di codice che apre un recordStore, inserisce qualche record, chiude e cancella il recordStore

```
String recordStoreName = "RecStore1";

RecordStore rs = null;
rs = RecordStore.openRecordStore(recordStoreName, true);

// Inserimento di record
String s = null;
s = "Iscrivetivi a java2me.org"; rs.addRecord(s, 0, s.length();
s = "E' bello partecipare"; rs.addRecord(s, 0, s.length();

// Chiusura del recordStore
rs.closeRecordStore();

// Cancellazione del recordStore
RecordStore.deleteRecordStore(recordStoreName);
```

In questo esempio è stata (ancora una volta) omessa tutta la gestione delle eccezioni. Basti ricordare che ogni metodo della classe `RecordStore` **può sollevare più eccezioni e che per operare su un recordStore (lettura, scrittura, modifica e cancellazione di record) il recordStore deve essere prima aperto (pena il sollevamento dell'eccezione** `RecordStoreNotOpenException`**) .**
Inoltre, nell'esempio, si è preceduto alla cancellazione fisica dell'intero recordStore. Non è assolutamente necessario farlo ogni volta altrimento che persistenza si otterrebbe ?

Passiamo ora alla lettura dei record presenti in un recordStore. E' chiaro che prima di poter leggere dei record è necessario che il recordStore sia aperto regolarmente e che ci siano record al suo interno :-)

Per poter leggere un record in una data posizione del recordStore, l' API RMS mette a disposizione il metodo

```
public int getRecord(int recordID, byte buffer[], int offset);
```

dove:

- **recordID rappresenta l' ID del record**
- **buffer rappresenta l'array di byte che andranno a contenere il record letto**
- **offset rappresenta la posizione da dove (all'interno di buffer) cominiciare a copiare il record**

E' chiaro che la lunghezza del buffer che ospiterà il record deve essere abbastanza grande da contenere il record stesso, altrimenti verrà sollevata un'eccezione di tipo `ArrayIndexOutOfBoundException`**. Il numero restituito dal metodo** `addRecord()` **rappresenta il numero di byte effettivamente letti dal recordStore.**

Vediamo un esempio di come leggere tutti i record presenti in un recordStore utilizzando anche un piccolo trucco per evitare di cadere nell'errore di allocare un buffer non grande a sufficienza per contenere tutti i record. Per semplicità verrà tralasciata la gestione delle eccezioni.

```
byte buffer = new byte [50];

for (int i=1; i <= rs.getNumRecords(); i++) {
    // Verifico se la lunghezza del buffer è sufficiente
   if (rs.getRecordSize(i) > buffer.length) buffer = new byte[rs.getRecordSize(i)];
   len = rs.getRecord(i, buffer, 0);
   String recordValue = new String(buffer, 0, len);
   System.out.println("Record num. " + i + "= " + recordValue);
}
```

Questo esempio di codice merita un pò di attenzione.

Per prima cosa è stato allocato un buffer di lunghezza 50 necessario per contenere i vari record del recordStore che di volta in volta verranno letti. Dato che non si può sapere a priori la lunghezza massima del più grande record memorizzato, il primo `if` **del ciclo** `for` **serve proprio per verificare se la lunghezza occupato dall'i-esimo record è maggiore della lunghezza del buffer. In caso affermativo viene ri-allocato un nuovo buffer capace di memorizzare il record. Con questo metodo si è mostrato anche l'utilizzo del metodo** `getRecordSize(...)`

Il ciclo for inizia da uno ed itera fino a che non si raggiunge (notare il <=) il numero di record memorizzato nel recordStore. Il metodo `getNumRecords()` **restituisce infatti il numero di record memorizzati nel recordStore. I record memorizzati hanno un indice che inizia da uno.**

Il metodo `getRecord(...)` **interno al ciclo for ha un indice che indica la posizione del record da leggere (che si è detto deve partire da uno), il buffer dove memorizzare il record (che grazie alla** `if` **precedente è grande abbastanza per contenere il record) e l' offset (interno a buffer) da dove iniziare a copiare il record. Il valore restituito rappresenta il numero di byte letti (cioè la lunghezza del record) e viene utilizzato per costruire una stringa (lunga da zero a len) che servirà per stampare il contenuto valore del record.**

A questo punto una domanda sorge spontanea. Ma nel recordStore si possono memorizzare solo Stringhe? La risposta è sicuramente no, ma tutto ciò che è possibile registrare sotto forma di array di byte.
Se necessario, in un prossimo articolo, mostrerò una tecnica per memorizzare strutture dati più complesse (tecnica che ha a che fare più con J2SE che con J2ME), dipende da quanto interesse susciterà quest'articolo.

L'esempio mostrato per leggere tutti i record presenti in un recordStore (tramite il metodo `getRecord`**) va bene per letture semplici. L' API RMS mette a disposizione degli strumenti più potenti per poter iterare sui record, ordinarli e filtrarli.**

La classe `RecordEnumeration` **fornisce i metodi per potersi muovere avanti ed indietro all'interno di un recordStore. L'impostazione di un enumeratore richiede solo qualche riga di codice, infatti l'esempio precedente, utilizzando l'enumeratore diventa:**

```
RecordEnumeration re = rs.enumerateRecord(null, null, false);
while (re.hasNextElement()) {
   String recordValue = new String(re.nextRecord());
   System.out.println("Record num. " + i + "= " + recordValue);
}
```

La classe `RecordEnumeration` **mette a disposizione i seguenti metodi**

Interfaccia RecordEnumeration

`int numRecords()`
Numero di record nell'enumerazione (set di risultati)

`void destroy()`
Libera le risorse interne usate dal RecordEnumeration

`boolean hasPreviousElement()`
Restituisce true se esesistono elementi nella direzione precedente

`boolean hasNextElement()`
Restitituisce true se esistono elementi nella direzione successiva

`boolean isKeptUpdated()`
Restitituisce true se l'enumeration sarà reindicizzata al momento della modifica del recordStore

`void keepUpdated(boolean keepUpdated)`
Imposta la funzione di reindicizzazione al momento della modifica del recordStore. Utilizzato per indicare se l'enumeration dovrà aggiornare il suo contatore interno a seguito di inserimento, cancellazione ed aggiornamento di record.

`byte[] nextRecord()`
Restituisce una copia del prossimo record nell'enumeration. Il prossimo record dipende dal filtro e dal comparatore impostati.

`int nextRecordId()`
Restituisce il recordID del prossimo record nell'enumeration.

`int previousRecordId()`
Restituisce il recordID per precedente record nell'enumeration

`byte[] previousRecord()`
Restituisce una copia del precedente record nell'enumeration.

`void rebuild()`
Richiede un aggiornamento dell'enumeration in modo da riflettere i cambiamenti

`void reset()`
Riporta lo stato dell'enumeration allo stesso stato di quando è stata creata

L'enumeratore conserva un indice interno del recordStore però è necessario porre attenzione al fatto che se una volta ottenuto un enumeratore, il recordStore viene modificato, l' enumeratore potrevve restituire risultati scorretti. Per risolvere questo problema la classe `RecordEnumerator` mette a disposizione un metodo per poter "reindicizzare" il record in modo da tener conto delle modifiche apportate ma questa non è la sola possibilità messa a disposizione dall' RMS. Infatti per poter reindecizzare è possibile:

1. **Utilizzare il metodo `rebuild()` della classe `RecordEnumeration` ogni volta che viene aggiornato, cancellato o aggiunto un record. Questo metodo funziona solo si è molto precisi e non vengono lasciati buchi**
2. **Configurare un listener dei record che ci avverta delle modifiche al recordStore.**

Per poter configurare un listener è necessario creare un oggetto che implementi l'interfaccia `RecordListener` ed implementare i suoi metodi. L'interfaccia `RecordListener` espone i seguenti metodi:

Interfaccia RecordLister

`recordAdded(RecordStore recStore, int recordID)`
Chiamato quando un record è stato aggiunto al recordStore.

`recordChanged(RecordStore recStore, int recordID)`
Chiamato dopo che un record nel recordStore è stato cambiato

`recordDeleted(RecordStore recStore, int recordID)`
Chiamato dopo che un record è stato cancellato dal recordStore

Se si osserva la firma del metodo `enumerateRecord` **della classe** `RecordStore`

```
public RecordEnumeration enumerateRecords(RecordFilter filter, RecordComparator
comparator, boolean keepUpdated)
```

si osserva che questi prende in input tre parametri. In particolare

filter **: E' un oggetto che implementa l'interfaccia RecordFilter e serve ad indicare quali record (secondo un filtro) devono far parte dell'enumeration**

comparator **: E' un oggetto che implementa l'interfaccia RecordComparator e server per poter definire una regola per ordinare i record**

keepUpdated **: Se true, l'enumeratore manterrà aggiornato l'enumeration in modo da riflettere ogni modifica che riguardi il recordStore. E' da usare con cautela dato che può impattare molto sulle performance. Se false l'enumeration non sarà aggiornata e potrebbe quindi restituire anche record che intanto sono stati cancellati o che sono stati aggiunti successivamente alla chiamata della creazione dell' enumeration.**

Molto importanti sono i primi due parametri, infatti, grazie ad essi è possibile creare enumeration di record molto particolari. L'interfaccia `RecordFilter` **serve per poter specificare dei criteri di ricerca all'interno del recordStore, verranno mostrati solo quei record che soddisfano il filtro. L'interfaccia** `RecordComparator` **serve ad ordinare i record in base a qualche criterio.**
Vediamo alcuni esempi.

Supponiamo di avere un recordStore che contenga Stringhe e che si voglia ottenere tutti i record che inizino per la Stringa J2ME, il filtro necessario allo scopo sarà:

```
class StartWithFilter implements RecordFilter {

   private String pattern = null;

   public StartWithFilter(String str) { pattern = str; }
   public boolean matches(byte[] arg0) {
      String str = new String(arg0);
      if (str.startsWith(pattern)) return true;
      else return false;
   }
}
```

Questa classe deve essere prima inizializzata con un patter che verrà utilizzato per verificare se la stringa ricevuta in input al metodo matches() soddisfa o meno il filtro. Il codice per poter eseguire una ricerca con questo filtro è il seguente:

```
try {
   StartWithFilter filter = new StartWithFilter("J2ME");
   RecordEnumeration enum = rs.enumerateRecords(filter, null, false);
   while (enum.hasNextElement()) {
      String str = new String(enum.nextRecord());
      System.out.println("Record: " + str);
   }
} catch (Exception e) { System.err.println(e); }
```

ESEMPIO 1

Supponiamo ora di voler ordinare le stringhe presenti nel database in ordine alfabetico. Per fare questo bisogna creare una classe che implementi l'interfaccia `RecordComparator` **e che sia in gradi di effetuare l'ordinamento tra due stringhe. La classe che segue fa al nostro caso:**

```
class Comparator implements RecordComparator {
    public int compare(byte[] arg0, byte[] arg1) {
      String str1 = new String(arg0);
      String str2 = new String(arg1);
      int result = str1.compareTo(str2);
      if (result == 0) return RecordComparator.EQUIVALENT;
      else if (result < 0) return RecordComparator.PRECEDES;
      else return RecordComparator.FOLLOWS;
   }
}
```

Questa classe implementa l'unico metodo dell'interfaccia `RecordComparator` **e si occupa di convertire i due array di byte in stringhe e di effettuare poi un confronto tra le due strighe per poter decidere quale venga prima e quale venga dopo. Sarà l'implmentazione stessa del metodo** `enumerateRecords` **a richiamare più volte questa classe per verificare (secondo un algoritmo di ordinamento interno) quale record vada prima dell'altro. Un esempio di codice che esegua un ordinamento utilizzando questa classe è il seguente:**

```
try {
   Comparator comp = new Comparator();
   RecordEnumeration enum = rs.enumerateRecords(null, comp, false);
   while (enum.hasNextElement()) {
      String str = new String(enum.nextRecord());
      System.out.println("Record: " + str);
   }
} catch (Exception e) { System.err.println(e); }
```

ESEMPIO 2

Notate come il codice di ESEMPIO 2 sia quasi identico a quello di ESEMPIO 1. In pratica quello che cambiano sono solo i parametri dati in pasto al metodo `enumerateRecords`**.**

Chiaramente la fantasia non ha limiti e quindi provate ad immaginare una combinazione delle due cose, cioè una chiamata a metodo `enumerateRecords` **nella quale indicate sia un filtro che un ordinamento …**

La possibilità di poter effettuare degli ordinamenti e di poter impostare dei filtri è molto potente però è anche molto dispendiosa in termini di tempo ed il tempo è prezioso (specialmente per applicazioni che girano su un telefonino). Quindi il mio consiglio è quello di non abusarne e (proprio quando non se ne puo fare a meno) implmentare degli algoritmi "furbi". Facciamo un esempio.

Supponiamo di voler ordinare le stringhe di un recordStore in base … alla lunghezza (visto che stiamo ordinando, inventiamoci algoritmi strani)

Vi propongo due implementazioni di una ipotetica classe `LenghtComparator`**.**

```
class LengthComparator implements RecordComparator {
    public int compare(byte[] arg0, byte[] arg1) {
      String str1 = new String(arg0);
      String str2 = new String(arg1);
      int result = str1.compareTo(str2);
      if (result == 0) return RecordComparator.EQUIVALENT;
      else if (result < 0) return RecordComparator.PRECEDES;
      else return RecordComparator.FOLLOWS;
   }
}
```

ESEMPIO 3A

```
class LenghtComparator implements RecordComparator {
    public int compare(byte[] arg0, byte[] arg1) {
      int diff = arg0.length - arg1.length;
      if (diff == 0) return RecordComparator.EQUIVALENT;
      else if (diff< 0) return RecordComparator.PRECEDES;
      else return RecordComparator.FOLLOWS;
   }
}
```

ESEMPIO 3B

La differenza tra le due implementazioni è che se devo verificare la lunghezza di due record non è necessario convertire l'array di byte in Stringa e poi successivamente confrontare la lunghezza delle due stringhe. La creazione di una stringa in java è un'operazione molto dispendiosa. Provate a verificare i tempi necessari ad ordinare una decina di record utilizzando queste due implementazioni e vedrete quale delle due è più performante.