JDBC

Di Michele de Nittis
Versione 06/08/2008
www.mdenittis.it

JDBC 1
Introduzione 2
Caricamento dell'appropriato Driver JDBC 2
Classi 3
  Connection 4
  Statement 5
  PreparedStatement 6
  ResultSet 7
Alcuni *Driver Name* ed URL 9
  MS SqlServer 2000 9
  Sybase 11 10
  Mysql 5 10
Fonti Documentali 10

Introduzione

JDBC è un'API che consente l'accesso ai dati gestiti in un database management system (RDBMS) da un'applicazione java.

Per connettersi ad un database è necessario caricare il driver specifico di quel RDBMS, ovvero quel software che provvede a tradurre le invocazioni delle funzioni dell'API JDBC nell'opportuno protocollo proprietario del prodotto RDBMS usato.

Esistono quattro tipologie di driver JDBC:

- Tipo 1: **Driver Ponte JDBC-ODBC**: il driver JDBC non si connette direttamente al prodotto RDBMS ma ad un driver ODBC disponibile per quel prodotto. Il driver ODBC è, a sua volta, un software di livello intermedio che garantisce **isolamento**, cioè consente la connessione con un RDBMS sfruttando le API native del produttore ed esponendo al software di livello superione un'interfaccia omogenea e totalmente indipendente dalle caratteristiche implementative dell'RDBMS in uso;
- Tipo 2: **Driver JDBC Nativo**: il driver, in parte scritto in java, per connettersi al RDBMS, sfrutta direttamente le API native del produttore;
- Tipo 3: **Driver JDBC Java puro con connessione verso middleware**: questo driver, scritto totalmente in java, non si interfaccia direttamente con l'RDBMS ma con un livello software intermedio (*middleware*) del produttore del database che provvede a servire le richieste di connessione con l'RDBMS;
- Tipo 4: **Driver JDBC puro**: il driver di questo tipo è scritto completamente in java e si connette direttamente con l'RDBMS dialogando con esso mediante il protocollo specifico del produttore.

Caricamento dell'appropriato Driver JDBC

Per caricare il **driver** è possibile, e spesso preferibile, invocare il **caricamento dinamico** delle classi invocando il metodo statico ***Class.forName(****DriverName****)***. Questo metodo, oltre a caricare la classe del driver JDBC specificata dall'argomento *DriverName* di tipo *String*, restituisce un riferimento ad un oggetto di classe ***Class*** che descrive la classe del driver che abbiamo caricato e dovremo utilizzare.

Mediante il meccanismo detto "*riflessione*" (reflection) possiamo caricare una classe ed esplorarla a livello descrittivo attraverso un oggetto di classe ***Class***. Per ottenere un oggetto di questa classe a partire da un oggetto di classe java.myclasses.X si può invocare semplicemente il metodo metodo getClass() dell'oggetto. Se, invece, non si dispone dell'oggetto X (cioè di un'istanza della classe X) si può usare, come già detto, il metodo statico ***forName()*** della classe **Class** il quale compie tre passaggi fondamentali ed ha un *overload*:

```
public static Class  forName(String name, boolean initialize, ClassLoader loader)
throws ClassNotFoundException
```

a. **Locate and Load (individua e carica):** Il primo passaggio è quello di *Locate and Load*, cioè quello di individuare la classe attraverso la stringa identificativa *Name* e di caricarne il codice (*ByteCode*) mediante o un oggetto *ClassLoader* specifico o di sistema (se loader=null);

b. **Link (collega)**: Il secondo passaggio è quello di *Link*, cioè quello di verifica del codice e del suo successivo collegamento con il resto del programma chiamante;
c. **Initialize (inizializzazione)**: Il terzo ed ultimo passaggio è quello di *Inizialize*, cioè di inizializzazione della classe (da non confondere con l'inizializzazione dei suoi oggetti) che consiste nell'esecuzione di tutti gli inizializzatori statici e dei blocchi di inizializzazione della classe (ovviamente stiamo parlando della classe del driver JDBC). L'inilizzalizzazione viene eseguita solo se il parametro booleano **initialize** è **true**.

L'overload del metodo ***forName()*** ha la seguente firma:

```
public static Class forName(String className) throws ClassNotFoundException
```

ed è equivalente a ***Class.forName(className, true, currentLoader)***. Il metodo ***forName()*** restituisce, quindi, un riferimento ad un oggetto di classe ***Class***. Generalmente, in assenza di particolari esigenze, si è soliti usare questo *overload*.

Esempio:
Il caricamento del driver JDBC per l'RDBMS Sybase SQL Server 11 con il driver di pubblico dominio JTDS è il seguente:

```
try{
                    // caricamento della classe del driver
                    Class.forName("net.sourceforge.jtds.jdbc.Driver");
              } catch(Exception E){ . . . }
```

E' di fondamentale importanza porre attenzione sul prossimo punto, altrimenti quanto verrà detto successivamente sull'oggetto ***Connection*** rimarrà avvolto da un fastidioso alone di "mistero".

Le classi **Driver JDBC** hanno un *blocco statico di inizializzazione*, eseguito come conseguenza dell'invocazione del metodo statico ***Class.forName()***, che **registra** il driver sulla classe ***DriverManager*** impiegata, come vedremo, per creare una connessione JDBC. Registrare significa specificare alla classe ***DriverManager*** di usare l'opportuna classe **Driver** quando si vuole instaurare una connessione con uno specifico database. Sarà tutto più chiaro tra breve.

Esempio:
Il driver JDBC di MySql ha il seguente blocco statico di inizializzazione (i sorgenti di questi driver sono scaricabili gratuitamente):

```
static {
              try {
                    java.sql.DriverManager.registerDriver(new Driver());
              } catch (SQLException E) {
                    throw new RuntimeException("Can't register driver!");
              }
        }
```

Classi

L'API JDBC è costituito da diverse classi di oggetti. Le più importanti sono:

- *Connection;*
- *Statement/PreparedStatement;*
- *ResultSet.*

L'API, inoltre, mette a disposizione un particolare tipo di eccezione, ***SQLException***, che consente di avere informazioni sugli errori verificatisi durante le operazioni di interrogazione e manipolazione dei dati, secondo le specifiche del produttore dell'RDBMS usato.

Connection

Un oggetto di classe ***Connection*** rappresenta la connessione con uno specifico database management system. Per creare ed ottenere un oggetto di classe ***Connection*** si deve invocare il metodo statico ***DriverManager.getConnection(url)*** della classe ***DriverManager*** che cerca di stabilire una connessione con il database specificato dalla stringa ***url***, impiegando l'appropriato driver precedentemente caricato. Se il metodo completa correttamente la propria esecuzione, restituirà un riferimento ad un oggetto di classe **Connection** che rappresenta la voluta connessione con il database.

Esempio:

```
try{
Connection MyConn = DriverManager.getConnection("jdbc:jtds:sybase://10.16.227.196:5000/TESTBD;user=Me;password=*******");
} catch(Exception E){ . . .}
```

La domanda che sorge spontanea è: se con il metodo forName() della classe ***Class*** sono stati registrati più driver JDBC, come fa la JVM a impiegare quello opportuno per connettersi ad uno specifico database? Inoltre, come fa la classe ***DriverManager*** a usare l'appropriato oggetto driver in assenza di un suo riferimento? La risposta è nella stringa identificativa del driver, o ***url***, che viene passato per argomento al metodo.

Il metodo statico ***DriverManager.getConnection()*****,** infatti**,** seleziona il driver specificato nella ***url*** passatagli per argomento tra quelli registrati. La stringa ***url*** contiene informazioni strutturate in questo modo: *<protocol>:<driver>:<location>*

<protocol> in genere è valorizzato a "***jdbc***";

<driver> è la stringa identificativa del driver;

<location> è un insieme di informazioni strutturate che indicano, tra le altre cose, il server, la porta di connessione, il database, la login e la password.

E' proprio l'informazione contenuta nei campi <protocol>:<driver> (nel nostro esempio jdts:sybase) a consentire alla classe ***DriverManager*** di usare il giusto driver jdbc precedentemente caricato e registrato. La classe ***DriverManager***, inoltre, non ha bisogno di un riferimento ad un oggetto della classe del driver jdbc da impiegare perché questo è stato registrato.

Di seguito si riporta una possibile implementazione del metodo statico ***DriverManager.registerDriver()*** trovata all'indirizzo:

(http://www.flex-compiler.lcs.mit.edu/Harpoon/srcdoc/java/sql/DriverManager.html). E' possibile trovare sulla rete altre implementazioni del metodo (vedi anche http://kickjava.com/src/java/sql/DriverManager.java.htm).

```
191  /**
192   * A newly loaded driver class should call registerDriver to make itself
193   * known to the DriverManager.
194   *
195   * @param driver the new JDBC Driver
196   * @exception SQLException if a database-access error occurs.
197   */
```

```
198   public static synchronized void registerDriver(java.sql.Driver driver)
199               throws SQLException {
200     if (!initialized) {
201       initialize();
202     }
203     DriverInfo di = new DriverInfo();
204     di.driver = driver;
205     di.className = driver.getClass().getName();
206     // Note our current securityContext.
207     di.securityContext = getSecurityContext();
208     drivers.addElement(di);
209     println("registerDriver: " + di);
210   }
```

Registrare, dunque, può significare semplicemente memorizzare in un vettore (`private static Vector drivers`), denominato ***drivers***, un oggetto contenente due informazioni: la denominazione completa della classe del driver da registrare ed un'istanza della medesima (quella passata come argomento). Dunque il metodo ***getConnection()*** non deve fare altro che una semplice ricerca nel vettore ***drivers*** per trovare quello più opportuno.

Esempi

Creazione di una connessione con un database Sybase 11 con i driver jtds:

```
try{
      // caricamento della classe del driver
      Class.forName("net.sourceforge.jtds.jdbc.Driver");
} catch(Exception E){
      System.out.println(E.toString());
}
try{
      MyConn =
DriverManager.getConnection("jdbc:jtds:sybase://MyDb.sql.it:5000/MyDb;user=mydb;
password= "*******");
} catch(Exception E){
      System.out.println(E.toString());
}
```

Creazione di una connessione con un database MySql 5 con gli opportuni driver

```
try{
      // caricamento della classe del driver
      Class.forName("com.mysql.jdbc.Driver");
} catch(Exception E){
      System.out.println(E.toString());
}
try{
      MyConn =
DriverManager.getConnection("jdbc:mysql://MyDb.sql.it:3306/MyDb?user=mydb&passwo
rd="*******");
} catch(Exception E){
      System.out.println(E.toString());
      }
```

Creazione di una connessione con un database Ms SqlServer 2000 con gli opportuni driver del fornitore.

```
try{
      // caricamento della classe del driver
      Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
} catch(Exception E){
      System.out.println(E.toString());
}
try{
      MyConn =
DriverManager.getConnection("jdbc:microsoft:sqlserver://MyDb.sql.it:1433;
DatabaseName=MyDb;User=mydb&Password="*******");
} catch(Exception E){
      System.out.println(E.toString());
      }
```

Statement

Statement è l'oggetto usato per eseguire una query SQL (uno statement SQL, per l'appunto) ed, eventualmente, accedere ai risultati.

Non è possibile istanziare direttamente un oggetto ***Statement***, giacché ***java.sql.Statement*** è un'interfaccia, ma si deve usare un metodo factory, nella fattispecie ***Connection.createStatement()***.

Una volta ottenuto un oggetto Statement (ad esempio, per il driver jTDS la classe che implementa l'interfaccia ***Statement*** è ***net.sourceforge.jtds.jdbc.JtdsStatement***), si può procedere all'esecuzione di una query SQL, formulata all'interno di un astringa, impiegando alcuni dei suoi metodi, i più usati tra i quali sono:

- **execute**() ed i suoi overload: esegue la query SQL passata come parametro (String);
- **executeUpdate()** ed i suoi overload: esegue la query SQL passata come argomento che **non genera risultati** (ad esempio INSERT, DELETE, UPDATE, e tutti gli statement DDL);
- **executeQuery**(): come **execute()** ma restituisce i risultati in un oggetto ***ResultSet***.

PreparedStatement

Generalmente nelle applicazioni si compongono le query che dovranno essere eseguite in funzione dello stato del programma o dei dati inseriti dall'utente. Quindi, in generale, le stringhe delle query SQL vengono composte a tempo d'esecuzione in modo dinamico.

Questa modalità di composizione delle query, per quanto generale e flessibile, comporta un certo impegno nella scrittura del codice da parte del programmatore ed un notevole impiego di memoria.

Quando nelle applicazioni si devono eseguire ripetutamente le stesse query, variando solamente alcuni parametri tra un'esecuzione e la successiva, è conveniente utilizzare l'oggetto **PreparedStatement** che consente di preparare una volta per tutte una query completa in ogni sua parte e parametrica.

```
String anno=null;
String capitolo=null;
PreparedStatement PS=null;
Connection MyConn=null;
ResultSet ResSet2=null;
….
….
PS = MyConn.prepareStatement("SELECT SAN.ID, SAN.[Numero Mandato],
SAN.Capitolo, SAN.anno FROM SCHEDA_ARCHIVIO_NEW SAN WHERE SAN.ANNO=?
AND SAN.Capitolo=? AND SAN.Oscurato='N'  ORDER BY SAN.[Numero Mandato] DESC");
```

```
....
....
try{
        PS.setString(1, anno);
        PS.setString(2, capitolo);
        ResSet2 = PS.executeQuery();
        ResSet2.next(); //mi posiziono sul primo record
        System.out.println("Debug: Seconda query eseguita.");
} catch(SQLException SQLE){
....
}
```

Mediante un oggetto di classe **PreparedStatement** non ci si deve più preoccupare di comporre la query dinamicamente prima della sua esecuzione, ma è possibile scriverla in modo parametrico una sola volta nel codice sorgente ed impostare opportunamente i parametri prima di ogni esecuzione. La parametrizzazione della query avviene mediante il **carattere segnaposto "?"**. In pratica con l'oggetto **PreparedStatement** si costruisce un **template** di una query ricorrente. La sostituzione dei caratteri "?" con gli appropriati valori avviene mediante i metodi "**setter**" dell'oggetto **PreparedStatement**. Tali metodi hanno tutti una firma simile: il primo parametro è l'indice posizionale del carattere segnaposto all'interno della query ed il secondo è il valore che si vuol sostituire al carattere segnaposto.

Operativamente, come si vede dal frammento di codice sorgente sopra riportato, si istanzia un nuovo oggetto di classe **PreparedStatement** mediante il metodo factory **prepareStatement()** dell'oggetto Connection, passando come parametro una stringa che rappresenta il template di una query SQL.

ResultSet

I risultati di una query lanciata con il metodo executeQuery() di un oggetto **Statement** o **PreparedStatement** vengono restituiti al chiamante mediante un oggetto che implementa l'interfaccia **ResultSet.** Questa definiosce metodi per:

1. Portarsi sul primo o sull'ultimo record dell'insieme dei risultati;
2. Portarsi sul successivo o sul precedente record;
3. Verificare se la query ha prodotto risultati;
4. Prelevare i valori degli attributi del record corrente con opportuni metodi **getter**;
5. Aggiornare i valori degli attributi del record corrente (dalla versione JDBC 2.0);
6. Leggere il numero d'ordine del record corrente;
7. Eseguire Altre operazioni.

Si deve premettere che lo scorrimento in avanti e indietro tra i record dipende dal tipo di **ResultSet** che si è chiesto ed ottenuto. Esistono, infatti, oggetti **ResultSet** a **scorrimento monodirezionale** (TYPE_FORWARD_ONLY) ed a **scorrimento bidirezionale** (TYPE_SCROLL_SENSITIVE, TYPE_SCROLL_INSENSITIVE). La differenza tra i due tipi di ResultSet a scorrimento bidirezionale è che nel primo tipo non è possibile spostarsi sul record che si trova in una determinata posizione assoluta (quindi sono possibili solo spostamenti per scostamenti relativi alla posizione corrente), nel secondo tipo si [1].

Gli oggetti **ResultSet** di tipo TYPE_SCROLL_INSENSITIVE non sono generalmente sensibili ai cambiamenti apportati ai record da altri utenti, mentre quelli di tipo TYPE_SCROLL_SENSITIVE lo sono.

Altra caratteristica degli oggetti che implementano l'interfaccia **ResultSet** è il loro *livello di concorrenza*. I **ResultSet** di tipo CONCUR_READ_ONLY sono di *sola lettura*, mentre quelli di

tipo CONCUR_UPDATABLE *possono essere aggiornati*. In quest'ultimo caso le modifiche apportate potranno eventualmente influenzare altri **ResultSet** di tipo TYPE_SCROLL_SENSITIVE.

Il tipo di **ResultSet** deve essere specificato al momento dell'istanziamento, cioè quando si invoca il metodo *factory* **createStatement()** con la firma a due parametri interi:

```
public Statement createStatement(int resultSetType, int resultSetConcurrency) throws SQLException
```

Nel caso si usi l'overload senza parametri, viene restituito un **ResultSet** di tipo TYPE_FORWARD_ONLY, CONCUR_READ_ONLY con il quale non si possono utilizzare tutti i metodi di navigazione tra i record, pena un'eccezione.

Ad un oggetto che implementa **ResultSet** è associato un *cursore* ovvero una struttura che indica la posizione all'interno dell'insieme dei record. Inizialmente la il cursore è posizionato prima dell'eventuale primo record dell'insieme. Questo fatto può essere fonte di diversi errori di *runtime*.

Ecco un elenco dei metodi più comuni ed utili dell'interfaccia **ResultSet**. Ovviamente l'elenco esaustivo si trova nella documentazione ufficiale dell'API JDBC (vedi paragrafo sulle fonti).

boolean absolute(int row)	Sposta il cursore al record in posizione **assoluta** specificata dal parametro intero;
void afterLast()	Sposta il cursore **dopo** l'ultimo record;
`void beforeFirst()`	Sposta il cursore **prima** del primo record;
`Void close()`	Rilascia le risorse allocate per il ResultSet;
`void first()`	Sposta il cursore **al primo** record;
`Array getArray(int i)` `Array getArray(String colName)`	Restituiscono tutti i valori assunti nel ResultSet da una colonna, specificata per indice o per nome, in un Array.
`<Tipo> get<Tipo>(int i)` `<Tipo> get<Tipo>(String colName)`	**Metodi getter**: ve ne uno per ogni tipo nativo java <Tipo>, compreso Object: restituiscono il valore della colonna specificata, per indice o per nome, del record corrente.
`ResultSetMetaData getMetaData()`	Restituisce un oggetto di classe **ResultSetMetaData**, importantissimo per avere informazioni riguardanti le colonne dell'insieme di record del ResultSet. Vedi documentazione API.
`int getRow()`	Restituisce la posizione del record corrente.

`boolean isFirst()`
`boolean isBeforeFirst()`
`boolean isLast()`
`Boolean isAfterLast()`
	Metodi importantissimi nei cicli di scansione del ResultSet per comprendere rispettivamente se si è sul primo record, prima del primo record, sull'ultimo record, dopo l'ultimo record.

`boolean next()`
`boolean previous()`
`boolean relative(int Rows)`
	Metodi importantissimi nei cicli di scansione del ResultSet per muoversi in modo **relativo** rispettivamente di una posizione in avanti, indietro, di un certo numero di posizioni (l'argomento negativo implica spostamenti indietro). I metodi restituiscono **false** se il cursore non referenzia un record dell'insieme.

`void update<Tipo>(int i, <Tipo> x)` `void update<Tipo>(String colName, <Tipo> x)`	**Metodi update**: ve ne uno per ogni tipo nativo java <Tipo>, compreso Object: impostano il valore della colonna specificata, per indice o per nome, del record corrente.
`boolean wasNull()`	Indica se il valore dell'ultima colonna letta è NULL. Questo fatto implica che questo metodo deve essere invocato **dopo** l'invocazione di uno dei metodi **getter** (nota: Nei database NULL è un valore. Non si deve, quindi, confondere NULL con la parola chiave java **null**).

Una volta terminato l'uso di un oggetto che implementa l'interfaccia ResultSet, è buona norma rilasciare le risorse impiegate e ciò si può fare invocando il metodo **close()**.

Un Esempio di uso di ResultSet

```
.....
            ResultSet ResSet;
            MySql = new StringBuffer("SELECT SO.name, SO.id, SC.colid, SC.name
colonna, SC.type, SC.length dimensione, ST.name tipo, ST.type, SC.prec, SC.scale
FROM");
            MySql.append(" (syscolumns SC INNER JOIN sysobjects SO ON
SO.id=SC.id) INNER JOIN systypes ST ON ST.usertype=SC.usertype WHERE ");
            MySql.append(" SO.name='").append(tabella).append("'");
            try{
            stm= MyConn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
            ResSet = stm.executeQuery(MySql.toString());
            } catch (SQLException SQLE){
                  throw new Exception("Impossibile leggere le colonne - " +
SQLE.toString());
                  }
       … … …
try{
            if(ResSet.isBeforeFirst()==true){ //ciclo di scansione del ResultSet
                  while(ResSet.next()==true){
                        counter++;
                        tmpCol = new colonna();
                        tmpCol.tabella=tabella;
                        tmpCol.dimensione = ResSet.getInt("dimensione");
                        tmpCol.nome=ResSet.getString("colonna");
                        tmpCol.tipo=ResSet.getString("tipo");
                        tmpCol.precision=ResSet.getInt("prec");
                        tmpCol.scale=ResSet.getInt("scale");
      . . . . . .
                  }
                  ResSet.close();
            } //fine ciclo di scansione del ResultSet
            } catch (SQLException SQLE){
                  throw new Exception("Errore in getColonne()- ciclo: " +
SQLE.toString());
                  }
```

Alcuni *Driver Name* ed URL

MS SqlServer 2000

Driver *Name: com.microsoft.jdbc.sqlserver.SQLServerDriver;*
URL: *jdbc:microsoft:sqlserver*;
Porta di default: *1433*;

Sybase 11
Driver Name*: net.sourceforge.jtds.jdbc.Driver;*
URL: *jdbc:jtds:sybase*;

Mysql 5
Driver Name*:* com.mysql.jdbc.Driver*;*
URL: *jdbc:mysql*;
Porta di default: *3306*;

Fonti Documentali

[1]. API Java: http://java.sun.com/j2se/1.4.2/docs/api/overview-summary.html
[2]. Tutorial JDBC: http://java.sun.com/docs/books/tutorial/jdbc/basics/index.html

Per errori, commenti e segnalazioni: m_dennix@yahoo.com, m.dennix@tiscali.it