Using CachedRowSet to Transfer JDBC Query Results Between Classes

by Sean Eidemiller
06/23/2004

The Java Database Connectivity (JDBC) API provides developers with an interface to a SQL database server, such as MySQL or Oracle. Central to any JDBC application is the `java.sql.ResultSet` interface, which provides access to query results. It is a venerable interface, to be sure, but it does have one unfortunate shortcoming. As it stands, a `ResultSet` object "loses" its data when the connection to the database is terminated. Well, "loses" is perhaps not the best way to describe it, but this will be discussed later. For now, all you need to know is that the data is no longer accessible. So long, data! Dasvedanya! Arrivederci! And so forth.

Oftentimes it is not desirable to maintain a persistent connection while extracting query data from a `ResultSet` object. In the case of a production database that serves millions of queries a day, it would be more responsible to create a connection, execute a query, quickly grab the results, and then immediately close the connection, leaving the data processing itself to a middle-tier application. Or maybe the developer simply wants to create a single class for database interaction, and then delegate data processing to a separate class. In both of the preceding examples, it would definitely be cool to be able to return a `ResultSet` object to a different class or thin client *without* losing the results of a query.

Enter `CachedRowSet`, Java's answer to a "disconnected" `ResultSet`. In this article, you'll learn how to create a `CachedRowSet` object, populate it with data from a `ResultSet`, and return it to other classes that need to use and/or modify the data. Specifically, I'll show you how to write a common SQL interface class to service queries and updates for an entire application, thereby minimizing code duplication and redundancy. Also, you'll learn how to make changes to the data in a `CachedRowSet`, and then propagate those changes back to the database.

Obtaining a `ResultSet`

In a traditional JDBC application, a `java.sql.Connection` object is obtained from the driver manager and a `java.sql.Statement` object is created to execute queries on behalf of the user. A `ResultSet` is then used to access query results returned from `Statement.executeQuery()`, and is usually obtained in the following manner.

Note: The examples in this article use the MySQL Connector/J driver for JDBC, but you can use whichever driver works with your database.

```
import java.sql.*;

public class ResultSetExample {
   public static void main(String[] args) {

      Statement stmt = null;
      ResultSet resultSet = null;

      try {
         Class.forName(
            "com.mysql.jdbc.Driver").newInstance();

         Connection conn = DriverManager.getConnection(
            "jdbc:mysql://localhost/catsdb?" +
```

```
            "user=test&password=test");

         stmt = conn.createStatement();
         resultSet = stmt.executeQuery(
            "SELECT id, name FROM cats");

         // do something with resultSet
         ...

         // close the connection
         conn.close();

      } catch (SQLException se) {
         ...
      } catch (Exception e) {
         ...
      }
   }
}
```

To understand how a `ResultSet` works, think of it as a sort of pointer. The data returned from a call to `Statement.executeQuery()` is not stored in its entirety within a local context. Instead, a `ResultSet` provides a cursor that initially "points" to the first row in a query result table (Figure 1). Subsequent calls to `ResultSet.next()` will move the pointer to the next row (if it exists), which allows the `ResultSet` object to play nicely with iterative `while` loops.

```
resultSet = stmt.executeQuery("SELECT id, name FROM cats");

               ----------------------
               | id   | name        |
               ----------------------
resultSet ->   | 0    | Hobbes      |
               ----------------------
               | 1    | Bucky       |
               ----------------------
               | 2    | Bill        |
               ----------------------
```

Creating a `CachedRowSet`

This is all well and good, but unfortunately, the `ResultSet` is closed as soon as the `Statement` itself is closed, or used to retrieve another `ResultSet`. Therefore, the pointer that once provided a handle to the query results is no longer valid, and this is where `CachedRowSet` comes in. Unlike a `ResultSet`, a `CachedRowSet` stores all rows obtained from a query in local memory, which can be extremely convenient at times. Before jumping into the code, however, a short introduction is in order.

The `CachedRowSet` interface is part of the JDBC 2.0 API specification and will be included with the next J2SE release (J2SE 1.5, AKA "Tiger"). It implements both `javax.sql.RowSet` and `ResultSet`, so experienced JDBC developers should already be familiar with many of its methods. In fact, the differences between a traditional `ResultSet` and a `CachedRowSet` are largely transparent (for the purposes of this article, in any case).

Implementations of `CachedRowSet` will most likely be provided by third-party database vendors that require specific `RowSet` functionality. For now, however, you can download a generic reference implementation from the Sun Developer Network as an early access release. Incidentally, the reference implementation will work with J2SE 1.4.2, as well as the 1.5 beta. Either way, the file to be concerned with is *rowset.jar*. Simply place it in your classpath and you're ready to start coding.

One way to create a `CachedRowSet`, and perhaps the most convenient, is to populate it with data obtained from a `ResultSet` (which we already have). The reference implementation is called `CachedRowSetImpl`, and is initialized as follows:

```
CachedRowSetImpl crs = new CachedRowSetImpl();
crs.populate(resultSet);
```

Unfortunately, this method may not always provide an ideal solution. Say, for example, that a particular query produced several thousand (or even several million) rows as a result. The amount of data that can be stored in a `CachedRowSet` is of course limited by the amount of memory available to the JVM. Moreover, storing such a large amount of data can be a time-consuming and expensive operation. One way to deal with this problem is to initially populate the `CachedRowSet` with only a portion (or "page") of the results, and then store the entire set of rows at a later time, if necessary. For more information, refer to section 10.0 of the `CachedRowSet` API documentation.

Either way, you should now have a `CachedRowSet` object full of query data, with which you can do all sorts of cool things.

Returning a `CachedRowSet` to a Different Class

In the interest of code reduction, let's create a single SQL interface class to service `SELECT` queries for an entire application, and then return the results in the form of a `CachedRowSet`. Later, I'll show you how to use a `CachedRowSet` to propagate changes back to the database, but for now, we'll just stick to queries.

```
package jdbcapp.sql;

import com.sun.rowset.CachedRowSetImpl;
import java.sql.*;

public class SQLInterface {

   private CachedRowSetImpl crs;
   private String url;

   public SQLInterface(String username, String password) {
      url = "jdbc:mysql://localhost/catsdb?user=" +
      username + "&password=" + password;
   }

   public boolean execQuery(String query) {
      Statement stmt = null;
      Connection conn = null;
      ResultSet resultSet = null;

      try {
         Class.forName(
            "com.mysql.jdbc.Driver").newInstance();

         conn = DriverManager.getConnection(url);

         stmt = conn.createStatement();
         resultSet = stmt.executeQuery(query);

         // create CachedRowSet and populate
         crs = new CachedRowSetImpl();
         crs.populate(resultSet);

         // note that the connection is being closed
```

```
            conn.close();

            return true;

         } catch (SQLException se) {
            ...
            return false;

         } catch (Exception e) {
            ...
            return false;
         }
      }

      public CachedRowSetImpl getRowSet() {
         return crs;
      }
}
```

Perhaps the most convenient aspect of returning a `CachedRowSet` to the caller, rather than an array or a `Vector`, is that the code that interacts with the database doesn't need to know what sort of query results to expect, such as the number of columns or the data types. Thus, the developer is spared from the hassle of creating a `ResultSetMetaData` object to determine such things. The code example above simply executes the query, stores the data, and returns it without modification when the `getRowSet()` method is called. It will work for any `SELECT` query, regardless of the amount or type of data returned. `CachedRowSet` is dynamic by nature, and is smart enough to size itself appropriately.

Extracting Data from a `CachedRowSet`

Once the `CachedRowSet` is returned, it is obviously up to the caller to do something with it. Extracting the data stored within a `CachedRowSet` is just as easy as extracting data from a `ResultSet`. In fact, `CachedRowSet` implements the various `get` methods used by the `ResultSet` interface, such as `getInt()` and `getString()`, among others.

To demonstrate, the following class instantiates a `SQLInterface` object, performs a query, and displays the data returned in a `CachedRowSet`.

```
package jdbcapp;

import com.sun.rowset.CachedRowSetImpl;
import java.sql.SQLException;

public class CachedRowSetExample {
   public static void main(String args[]) {

      // create SQLInterface object
      SQLInterface iface =
         new SQLInterface("test", "test");

      // execute query
      if (!iface.execQuery(
         "SELECT id, name FROM cats")) {

         // exception caught, halt execution
         System.exit(1);
      }

      // create CachedRowSet and output
      // data to terminal
```

```
      try {

         CachedRowSetImpl crs =
            new CachedRowSetImpl();
         crs = iface.getRowSet();

      while (crs.next()) {
         System.out.println("ID: " + crs.getInt(1) +
            ",  Name: " + crs.getString(2));
      }

      } catch (SQLException se) {
         ...
      }
   }
}
```

The `next()`, `getInt()`, and `getString()` methods throw a `SQLException` if something goes wrong, but that's about the only thing you need to be aware of. Other important `get` methods include `getFloat()`, `getData()`, and `getObject()`. There are quite a few more, of course, and you should take a minute to familiarize yourself with them by perusing the `ResultSet` API documentation.

Making Changes to `CachedRowSet` Data

Not only do external classes have read access to the data returned in a `CachedRowSet`, they are able to propagate changes back to the database, as well. The `ResultSet` interface provides several update methods (`updateInt()`, `updateString()`, etc.) that are implemented by `CachedRowSet` and can be used to update query results data while disconnected. After making the desired modification offline, a connection is re-established, the `acceptChanges()` method is called, and the underlying database is updated accordingly. This makes `CachedRowSet` very efficient, because it effectively reduces the number of individual SQL statements that are needed, thereby reducing the load on the server itself. That being the case, a `CachedRowSet` may also be worth considering even when it is not necessary to drop the connection to the database, given the efficiency advantage.

In order to implement this functionality in the `SQLInterface` class, we need to add a `commitToDatabase()` method as follows:

```
public boolean commitToDatabase(CachedRowSetImpl crs) {
   Connection conn = null;

   try {
      Class.forName(
         "com.mysql.jdbc.Driver").newInstance();
      conn = DriverManager.getConnection(url);

      // propagate changes and close connection
      crs.acceptChanges(conn);
      conn.close();

      return true;

   } catch (SQLException se) {
      ...
      return false;

   } catch (Exception e) {
      ...
```

```
        return false;
    }
}
```

Updating a row in a `CachedRowSet` requires three distinct steps. First, assuming that the cursor is pointing to the row to be modified, a call is made to one of the aforementioned update methods (`updateString()`, for example) for each column that is to be changed. Second, the `updateRow()` method must be called *before* moving the cursor to a different row in the `CachedRowSet`. Finally, the connection is re-established and the `acceptChanges()` method is called as described earlier (in our case, the `SQLInterface` class takes care of this).

The `updateString` method in the example below accepts two parameters: the column index and the new value. Keep in mind that the column indices in a `CachedRowSet` are 1-based, as opposed to arrays, for example, which are 0-based.

```
crs.updateString(2, "Bagheera");
crs.updateRow();

iface.commitToDatabase(crs);
```

Conclusion

In this article, you've learned how to harness the power of the `CachedRowSet` by creating a common SQL interface to pass query data between classes, and propagate changes back to the database. In many ways more robust and elegant than a traditional "connected" `ResultSet`, the `CachedRowSet` provides increased functionality, to boot. In fact, we've barely scratched the surface. If you're interested in doing some further reading, be sure to check out the `RowSet` chapter in the Sun Developer Network JDBC tutorial. They cover two additional implementations of `RowSet` (`JDBCRowSet` and `WebRowSet`) that could prove useful in your next JDBC application, as well.

Resources

- Sample code for this article

Sean Eidemiller is a computer science major at Millersville University in Pennsylvania, and works as a part-time research assistant for KRvW Associates, LLC