A "Politically Correct" Use for Native Methods – Loading Database Tables

James W. Cooper

You probably know that you can write Java programs to connect to native code. I always assumed that this feature was primarily to take advantage of platform specific features like Windows or Unix services, and sort of put the feature aside in my mind to explore later. However, there are some cases where even the die-hard "pure Java" programmer can benefit from native methods and still write relatively portable code.

Loading DB2 Tables

I came across one of these recently when I had to find a way to get a large amount of data from several huge text files into an industrial strength database. While for small projects, I have always use Microsoft Access to handle data, this project was much too large and would eventually require multi-user access. Both of these mitigated against using Access and instead I adopted IBM's DB2 Universal Database, 6.1. The version I used during development was on an NT platform, but since the product is available for a wide variety of other platforms, writing Java to carry out this data ingestion seemed very appealing.

However, the amount of data I had to load was substantial and most large databases are not optimized for fast line-by-line programmatic loading of tables. Instead, the usual approach is to build a "load file" and then tell the database system to load that table into the corresponding table in the database. The difference is really quite astonishing. While loading a database programmatically has a throughput of 10-20 lines per second, I found that using the db2load command, I could load 680,000 lines of data in 4 minutes. This corresponds to 2845 lines per second!

Sometimes, when you just need to use a system service, you can get away with using the exec method of Java's Runtime class to simply execute that program from Java. This is a good, although scarcely portable approach for many quick and dirty programs, but not so great for one you hope to pass on to other users. In addition, the db2 load command will only work from a special NT command window, which you can't instantiate from within a Java program. Further, DB2 does not provide a Java method to carry out this load directly.

However, DB2 does provide a C (and C++) interface to a number of useful methods for interacting with the database system, and, of course, one of them is the table load function. All we have to do is write a little C program and figure out how to call it from Java. You'll be pleased to know that this only took me a day or so to figure this out. And now I'll try to save you some of that time.

Writing a C Load Procedure

The C program for loading a table uses the C sqlload function and is fortunately readily available in the DB2 example program tload.sqc. The function is very simple to call once

you set up the structures accordingly. However, you can copy the code in the tload.sqc example directly. The code example in Listing 1 is the guts of the load process.

```
EXEC SQL CONNECT TO :dbase USER :userid USING :passwd;
    strcpy (checkCmd, "CONNECT TO ");
    strcat(checkCmd, dbase);
    CHECKERR (checkCmd);

/*allocate the proper amount of space for the SQL  statment */
    imp_statement = convJString(env, jInsert);

    ActionString = (struct sqlchar *)malloc(strlen(imp_statement)
                                          + sizeof (struct sqlchar));
    ActionString->length = strlen(imp_statement);
    strncpy (ActionString->data, imp_statement, strlen(imp_statement));


    /* initialize the variables for the LOAD API */
    /* the DataFileList structure */
    DataFileList.media_type = SQLU_SERVER_LOCATION;
    DataFileList.sessions = 1;
    DataFileList.target.location = (sqlu_location_entry *) malloc
                                   (sizeof(sqlu_location_entry) *
DataFileList.sessions);
    data_file = convJString(env, jDataFile);
    strcpy (DataFileList.target.location->location_entry, data_file);
    printf ("Loading the file '%s'\n", data_file);
    _flushall();
    pLobPathList = NULL;
    CallerAction = SQLU_INITIAL;

    /* the sqluload input structure */
    InputInfo.sizeOfStruct = SQLULOAD_IN_SIZE; /* should noy change */
    InputInfo.savecnt = 1;                /* frequent consistency */
    InputInfo.restartcnt = 0;             /* start at row 1 */
    InputInfo.rowcnt = 0;                 /* load all rows */
    InputInfo.warningcnt = 0;             /* don't stop for warnings */
    InputInfo.data_buffer_size = 0;       /* default buffer size */
    InputInfo.sort_buffer_size = 0;       /* warning buffer size */
    InputInfo.hold_quiesce = 0;           /* don't hold the quiesce */
    InputInfo.restartphase = ' ';         /* must be ' ',L,B,D */
    InputInfo.statsopt = SQLU_STATS_NONE;   /* don't collect them */
    InputInfo.indexing_mode =
                        SQLU_INX_AUTOSELECT; /* choose indexing mode */

    /* the sqluload output structure */
    OutputInfo.sizeOfStruct = SQLULOAD_OUT_SIZE;

    /* the CopyTargetList structure */

    pCopyTargetList = NULL;
    OutputInfo.sizeOfStruct = SQLULOAD_OUT_SIZE;

    //call the db2 load function
    sqluload (&DataFileList,
              pLobPathList,
              &DataDescriptor,
```

```
                ActionString,
                FileType,
                FileTypeMod,
                LocalMsgFileName,
                RemoteMsgFileName,
                CallerAction,
                &InputInfo,
                &OutputInfo,
                pWorkDirectoryList,
                pCopyTargetList,
                pNullIndicators,
                pReserved,
                &sqlca);
    CHECKERR ("LOADing table");
```

Listing 1 – The C code for the guts of the DB2 table loading process. Note that the top like is a macro that gets expanded by the preprocessor.

What is an Sqc file?

You probably noticed that I referred to the file we started from as tload.sqc. This file differs from a standard C file in that it utilizes some special macros that are expanded out by a preprocessor before you compile the program. You will notice one of these in the first line. In order to compile these, you need to go to a command line window and type **db2cmd** to open a special DB2 command window. This provides a special environment and a connection to DB2 for the commands that follow. In this special command line environment, commands beginning with "db2" are passed to a special processor connected to DB2. A simple batch file to preprocess and convert the tableloader.sqc file using Visual C++ is

```
rem Usage: bldmsemb prog_name [ db_name [  userid password ]]

rem connect to the specified database
db2 connect to %2 user %3 using %4

rem Precompile the program.
db2 prep %1.sqc bindfile

rem Bind the program to the database.
db2 bind %1.bnd

rem Disconnect from the database.
db2 connect reset

rem  Compile the program.
cl -Id:\jdk1.2\include -Id:\jdk1.2\include\win32 -LD %1.c  util.c
      db2api.lib –Fe%1.dll
```

Writing the Java Program

In order to connect a Java program to a native method, you start by defining the Java call you'd like to make to the native method. You have to make a declaration of the native method you are going to use, so that you can construct the correct C function. I did this by writing a little class called TableLoader that has the load method which calls the native method load_table.

```
public class TableLoader {

    private Database db;      //connection to database
//native method declaration
private native long load_Table(String dbase, String userid, String pwd,
                               String file, String mode ,  String insert);
 public TableLoader(Database datab) {
    db = datab;
    System.loadLibrary ("tableloader");   //load native library (DLL)
 }
 // loads file into specified table
 public long load(String file, Stgring colDelimiter, String insert) {

        return load_Table(db.getName(), db.getUser(), db.getPassWord(),
                      file, "fastparse coldel"+colDelimiter, insert);
 }
}
```

Then, to generate the C-header file that enables you to write the proper C code, you run the javah program against this file:

```
javah TableLoader
```

This generates the file TableLoader.h which contains the declaration:

```
JNIEXPORT jlong JNICALL Java_TableLoader_load_1Table
 (JNIEnv *, jobject, jstring, jstring, jstring,
      jstring, jstring, jstring);
```

Now all we have to do is write a C program that provides that interface.

Writing the C Program

Every variable type in the above declaration is a special Java type that is included in the file jni.h. The jlong type map directly to C longs, but you must write a simple program to convert the jstring type variables to C char* variables.

```
//function to convert a Java string to a  const char*
const char *convJString(JNIEnv *env, jstring js) {
  return (*env)->GetStringUTFChars(env, js, 0);
}
```

So our actual C program must start by converting the jstrings to C string:

```
JNIEXPORT jlong JNICALL Java_TableLoader_load_1Table
(JNIEnv *env, jobject obj, jstring jDbase,
        jstring jUserId, jstring jPwd, jstring jDataFile,
        jstring jModArgs, jstring jInsert) {
          const char *d_base;
          const char *user_id;
          const char *p_wd;
          const char *mod_args;
          const char *imp_statement;
          const char *data_file;
      //These are passed into the db2 macro
      //to connect to the database
         EXEC SQL BEGIN DECLARE SECTION;
```

```
  char userid[9];
  char passwd[19];
  char dbase[9];
 EXEC SQL END DECLARE SECTION;


  d_base = convJString(env, jDbase);
  strcpy (dbase, d_base );
  user_id = convJString(env, jUserId);
  strcpy (userid, user_id);
  p_wd = convJString(env, jPwd);
  strcpy (passwd, p_wd);

  mod_args = convJString(env, jModArgs);
//macro to connect to the specified database
  EXEC SQL CONNECT TO :dbase USER :userid USING :passwd;
//macro to check for connection error
  strcpy (checkCmd, "CONNECT TO ");
  strcat(checkCmd, dbase);
  CHECKERR (checkCmd);
```

Loading an Actual Table

Now that we've built up this infrastructure, let's see how we use it. Suppose we have a table of words and the strengths of their relations (1) obtained from some linguistic analysis of a document collection:

```
sqa review | 60 | statement of work
standalone | 50 | test management
standalone | 50 | TestDirector
steps appendix | 50 | work effort
structural function | 40 | test run
```

And, suppose you have a DB2 table TermRelations, containing the columns:

```
LeftTerm
RightTerm
Rank
```

The Java statements to insert a load table of data into that table are just

```
TableLoader tload = new TableLoader(db);
String insert =
"INSERT INTO TERMRELATIONS (LEFTTERM, RANK, RIGHTTERM)";

  rows = tload.load (fileName, "|",  insert);
```

And now you see, we have built a simple Java class which encapsulates the loading of a DB2 table.

Portability

At first you might think, "nice, but isn't this restricted to Windows systems?" Fortunately, the answer is no. This code can be compiled into the appropriate library format on any platform where DB2 is available. Since there are a wide array of such

platforms, we have written a *portable* Java class for performing the fast loading of tables. The only thing that changes is the actual compile statement for the C program on each platform: the code does not change.

Packages

One pitfall I found that took us a bit of time to work through is how to handle packages. Suppose that your TableLoader class is to be part of a package, say examples.db2.TableLoader. Then you must place it in the appropriate directory for that package and run the javah program from the root of that package structure.

```
javah examples.db2.TableLoader
```

This will produce a header file that encompasses the package structure:

```
JNIEXPORT jlong JNICALL Java_examples_db2_TableLoader_load_1Table
  (JNIEnv *, jobject, jstring, jstring,
      jstring, jstring, jstring, jstring);
```

Conclusions

We've seen that you can write a simple C library based on existing example code to perform rapid loading of DB2 database tables. Then we saw how to write the Java native interface to it and how to make a simple class to encapsulate the table loading functions. This class and the corresponding C code is completely portable between all the platforms where DB2 is available.

References

1. James W. Cooper and Roy J. Byrd, "Lexical Navigation: Visually Prompted Query Expansion and Refinement," presented at Digital Libraries, '97, Philadelphia, PA. Available on the web at http://www.acm.org/pubs/citations/proceedings/dl/23690/p23-cooper/.

James W. Cooper is a computer science researcher and the author of 13 books in the field. His latest book, *Java Design Patterns: A Tutorial*, was published by Addison-Wesley-Longman, in January, 2000.