EuroForth 2023

Accessing an Oracle database using Forth

Abstract

It is occasionally necessary to access an Oracle database from a Forth applications. This paper illustrates the techniques we have used.

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

Within our own automation applications, we use the MySQL database. In previous papers, I have described the method that we use for accessing a MySQL database from Forth, which we refer to as "Forth Query Language" or FQL.

Despite the many benefits of MySQL, there are some of our customers who insist on using an Oracle database for their office systems.

It is therefore necessary for us sometimes to access an Oracle database, for the exchange of data about such things as category information and machine efficiency.

The technique required for Oracle is quite different from that required for MySQL.

2. A brief comparison of Oracle and MySQL

	Oracle	**MySQL**
Cost	Very expensive	Free
License	Proprietary	GPL
Paid support	Yes	Available
Free support	None	Strong
Performance	4.3/5	4.4/5
Indexing	Full text, hashed, binary	Full text, hashed
Learning curve	Very steep	Modest
Almost everything else	Same	Same

3. The available Oracle APIs

For many years, there were only two possible methods for accessing an Oracle database - the "Oracle Call Interface", and ODBC.

For anyone accustomed to the "C" API for MySQL, the Oracle Call Interface feels extremely complex and very hard to master.

ODBC adds an additional layer to the interface, and in the past we have experienced problems with versioning compatibilities of libraries and drivers.

Recently, a new API has been introduced - "Oracle Database Programming Interface for C" (ODPI-C). This is essentially a simplified wrapper that sits on top of the Oracle Call Interface. It is still more complex that the MySQL interface, but much more manageable than before. This is therefore the method we have chosen.

4. Typical tasks that are required

The actual database tables are not normally exposed on an Oracle system. To retrieve data from Oracle, one would normally use a SELECT statement, applied to a specially constructed "view". To insert data into Oracle, typically a prepared statement is called. The prepared statement would carefully constrain the provided parameters, in order to stop anything nasty from being inserted.

5. Declaring the necessary library function

Given the restricted types of operations normally required, only a small subset of the available ODPI-C functions needs to be declared in Forth, in just four different classes. This is still more than the number of functions needed for a comprehensive MySQL interface, but not too onerous.

For example, in the "statement" class, just six functions are needed.

```
\ DPISTMT

extern: int dpiStmt_execute( dpiStmt * stmt, dpiExecMode mode, uint32_t *
                             numQueryColumns );
extern: int dpiStmt_getRowCount( dpiStmt * stmt, uint64_t * count );
extern: int dpiStmt_fetch( dpiStmt * stmt, int * found, uint32_t * bufferRowIndex );
extern: int dpiStmt_defineValue( dpiStmt * stmt, uint32_t pos,
                                 piOracleTypeNum oracleTypeNum,
                                 dpiNativeTypeNum nativeTypeNum,
                                 uint32_t size, int sizeIsBytes,
                                 dpiObjectType * objType );
extern: int dpiStmt_getQueryValue( dpiStmt * stmt, uint32_t pos,
                                   dpiNativeTypeNum * nativeTypeNum,
                                   dpiData * *data );
extern: int dpiStmt_release( dpiStmt * stmt );
```

It will be seen that there are a large number of bespoke data types, and in order to ensure the accuracy of the external declarations, I always define these explicitly in the usual way, for example:

```
also types definitions
: dpiContext void ;
: dpiCommonCreateParams void ;
: dpiContextCreateParams void ;
: dpiConn int ;
: dpiErrorInfo int ;
...
previous definitions
```

A small number of structures also need to be replicated in Forth, and these need to be carefully checked for offset values against the equivalent "C" structures, so as to avoid any possible alignment problems. A typical structure is:

```
STRUCT dpiErrorInfo
  INT             dei.code
  INT             dei.offset16
  POINTER         dei.message
  INT             dei.messageLength
  POINTER         dei.encoding
  POINTER         dei.fnName
  POINTER         dei.action
  POINTER         dei.sqlState
  INT             dei.isRecoverable
  INT             dei.isWarning
  INT             dei.offset
END-STRUCT
```

Note that on a 64 bit system, an INT is still 32 bits!

Finally a selection of constants is required, and in this case they can be copied directly from the "C" header file and into Forth, for example:

```
\ Native type numbers

#define DPI_NATIVE_TYPE_INT64                       3000
#define DPI_NATIVE_TYPE_UINT64                      3001
#define DPI_NATIVE_TYPE_FLOAT                       3002
#define DPI_NATIVE_TYPE_DOUBLE                      3003
#define DPI_NATIVE_TYPE_BYTES                       3004
#define DPI_NATIVE_TYPE_TIMESTAMP                   3005
...
```

6. A Forth wrapper for an Oracle query

We first made some design decisions in order to simplify the wrapper.

a) Our interface does not need to be thread safe. Only one thread will be used to access the Oracle system. Therefore all data such as handles of connections or statements can he held in Forth VALUEs. If a thread safe interface were required, these could be moved into Forth user variables.

b) Because we query the Oracle database relatively infrequently (e.g. once every 10s), and because individual connections do not appear to be very costly in terms of processing time, we opted to create a new connection for each query. This greatly simplifies the error handling.

c) The Oracle server PC and its connections are not under our control. The server may occasionally be down for maintenance. If that happens, it is important that no data is lost. We therefore maintain a queue of data to be sent to Oracle, and offer the queries one by one with automatic retry, including backoff.

```
: ORACLE-QUERY { pzquery -- f } \ True if query prepared and executed
  FALSE                                        \ Assume failed
  ORACLE-INITCONTEXT IF                        \ Initialise Oracle context
    ORACLE-RELEASE                             \ Release any previous statement
    ORACLE-DISCONNECT                          \ New connection for each query
    ORACLE-CONNECT IF                          \ Connected OK
      pzquery ORACLE-PREPARE IF                \ Statement prepared OK
        ORACLE-EXECUTE IF                      \ Statement executed OK
        ORACLEMINRETRYTIME -> ORACLERETRYTIME   \ Set error retry time to minimum
          DROP TRUE                            \ Success
        THEN
      THEN
    THEN
  THEN
  DUP IF                                       \ Success
    ORACLEMINRETRYTIME -> ORACLERETRYTIME       \ Set error retry time to minimum
  ELSE                                         \ Failure
    ORACLECREATEDCONTEXT dpiContext_destroy DROP \ Destroy context
    0 -> ORACLECONTEXT                         \ Clear context
    ORACLE-RETRYWAIT                           \ Wait before retry
  THEN
;
```

The first step is to check for a connection "context", re-creating one if necessary. Then we ensure that everything is tidy after any previous query. The connection is then established.

Unlike in MySQL, query statements in Oracle have to be explicitly prepared first before being submitted for execution.

If all the above steps are successful, we can exit with a true flag, and as we go, reset the backoff time setting.

In the event of failure, we destroy the connection context (if the failure was due to some change in the configuration of the Oracle server, then a new context will be required). We then leave a gradually increasing amount of time before trying again.

We can look at one of those steps in slightly more detail, for example:

```
: ORACLE-ERROR ( ---zperror ) \ Returns an Oracle error message
  ORACLECONTEXT ORACLEERRORINFO dpiContext_getError   \ Populate error info struct
  ORACLEERRORINFO dei.message @                       \ Return error message
;

: ORACLE-CONNECT ( ---f ) \ True if Oracle connection established
  ORACLECONTEXT ORACLE-USER ORACLE-PASSWORD ORACLE-DATABASE NULL NULL
  ADDR ORACLECONNECTION dpiConn_create DPI_FAILURE = IF
    Z" Failed to connect to Oracle, " ORACLE-ERROR Z+ ERROR
    FALSE
  ELSE
    TRUE
  THEN
;
```

The connection parameters are normally regarded as valuable information, so need to be kept somewhere secure.

Note the recovery of the address of the full text of an error, from the error information structure as shown earlier. This is then added to our own log file, so we can see in detail about anything that goes wrong.

7. A typical SQL INSERT query

Because the insert queries are normally quite short and straightforward, we have not attempted to replicate the FQL system for constructing queries. Instead, the query is simply put together using the zero-terminated string concatenator Z+, which uses a scratchpad formed by dividing PAD into three sections. This is fine when the query can be guaranteed to be shorter than /PAD 3 / but would be no use in the MySQL general case, where queries can be very long.

```
: EXAMPLE-EXPORTDATA ( ---f ) \ Send data to Oracle
  Z" CALL EXPORT_MICROSS("                \ EXPORT_MICROSS cust, …, system
  ZQ Z+ EXPORTCUSID Z+ ZQ Z+ ZCOMMA Z+    \ Customer ID
  ...
  EXPORTAREA ZFORMAT Z+                   \ Sorting area
  Z" )" Z+
  ORACLE-QUERY
;
```

Note that the SQL INSERT statement is not used directly, instead a statement is prepared by the Oracle database supervisor, which we are able to use through an SQL CALL statement.

8. A typical SQL SELECT query

Here, the SQL SELECT statement is used directly, but is applied to a view rather than to a table directly. The view is declared by the Oracle database supervisor and contains the subset of the table columns that we are permitted to see.

```
: IMPORTPRODUCTS { | pcusid[ LENCUSID 1+ ] pcatid[ LENCATID 1+ ] -- }
\ Import products from Oracle
  FALSE                                                 \ Assume fail
  Z" SELECT CUSTOMERID, PRODUCTID "
  Z" FROM IMPORTPRODUCTS" Z+
  ORACLE-QUERY IF
    BEGIN
      ORACLE-FETCH
    WHILE
      1 ORACLE-NUMBER ZFORMAT ZCOUNT pcusid[ LENCUSID ZPUT  \ Get customer ID
      2 ORACLE-STRING                pcatid[ LENCATID ZPUT  \ Get category ID
      ...
    REPEAT
    DROP TRUE                                           \ Success
  THEN
;
```

Having run the query, we now need to analyse the results. In Oracle, results are provided in a completely different way from MySQL.

Let us first look at our wrapper word that fetches one row of results.

```
: ORACLE-FETCH { | pfound -- f } \ True if row of query result fetched
  0 -> pfound
  ORACLESTATEMENT ADDR pfound ADDR ORACLEINDEX dpiStmt_fetch
  DPI_SUCCESS = pfound 0<> AND
;
```

It is worth looking at this carefully, because it illustrates a frequent source of bugs when using Forth.

Like all "C" functions, dpiStmt_fetch can return only one parameter, and this is always a code to indicate if the function worked or not (e.g. DPI_SUCCESS). This function however also needs to tells us if it actually returned a row of data or not. We therefore need to provide as a parameter an address of where to put that information. This information is needed only within the Forth word, so we use a local value "pfound" and pass ADDR pfound to our "C" function.

Assuming that dpiStmt_fetch executes OK, then it will have written either true or false into "pfound".

But here is where the problem lies. We are in 64 bit Forth, therefore "pfound" is a 64 bit value. But dpiStmt_fetch writes a boolean value, which in 64 bit Linux is represented by an INT, which is a 32 bit value. Forth local values are not initialised, so at the start of the function "pfound" is random. After the "C" function returns, "pfound" still returns nonsense.

It is essential to remember (but very easy to forget) that local values like these must be explicitly initialised in the code "0 -> pfound".

This is such a common source of errors, that I would propose that the Forth standard should be changed to require that local values are automatically zeroed.

9. Analysing the Oracle result set

In MySQL, there is only one data type in a result set. The set consists solely of an array of pointers to zero terminated strings.

In Oracle, it is much more complicated because each column data type returns a different structure, and the result set consists of pointers to structures.

```
: ORACLE-STRING { pcol | pbytes -- p$ plen } \ Get string value at column
  ORACLESTATEMENT pcol ADDR ORACLETYPE ADDR ORACLEPDATA dpiStmt_getQueryValue DROP
  ORACLEPDATA dpiData_getBytes -> pbytes
  pbytes dpiBytes.ptr @ pbytes dpiBytes.length I@
;
```

In the example above, we recover a string result. This is a three stage process.

First we have to get the data type, and a pointer to the data. For simplicity, we have not included any error checking here - we assume we have read the customer's specification of the view correctly.

Then, from the data pointer, we extract a pointer to the string structure.

Finally, from the string structure, we extract the base address and length of the string. Note that though when tested we find that the string is in fact zero terminated, the documentation does not actually say so, therefore, we need to assume that it isn't.

10. Conclusion and future work

This set of wrapper functions make it reasonably easy to make simple queries to an Oracle database. If more complex queries are needed, it would be possible to extend the FQL system to accommodate Orace as well as MySQL. A further development would be to make the word set thread safe.