

Database Handling by DaoDataModel

Limin Fu (phoolimin@gmail.com)

September 19, 2009

DaoDataModel is a module to map Dao classes to database tables (currently works only with MySQL). With such mapping, the handling of database with Dao becomes very simple.

For example, there is such class,

```
load DaoDataModel;
# class that can be mapped to a database table
class Gene
{
  my id : INT_PRIMARY_KEY_AUTO_INCREMENT;
  my name : VARCHAR100;
  my seq : TEXT;
}
```

Here, the type **INT_PRIMARY_KEY_AUTO_INCREMENT**, **VARCHAR100**, **TEXT** are defined by the DaoDataModel module, to allow the module interpret properly the class fields and corresponding database record fields. Those types provided by DaoDataModel are defined as aliases of the built-in data types,

- **int** type as integer fields:

```
INT
TINYINT
SMALLINT
MEDIUMINT
INT_PRIMARY_KEY
INT_PRIMARY_KEY_AUTO_INCREMENT
```

- **string** type as character fields:

```
CHAR10
```

```
CHAR20
CHAR50
CHAR100
CHAR200
VARCHAR10
VARCHAR20
VARCHAR50
VARCHAR100
VARCHAR200
```

- **string** type as text and blob fields:

```
TEXT
MEDIUMTEXT
LONGTEXT
BLOB
MEDIUMBLOB
LONGBLOB
```

1 Connect Database

For MySQL database, one can connect to the database by,

```
# connect database
model = DataModel( 'dbname', 'host', 'user', 'password' );
```

The prototype of DataModel() is,

```
DataModel( name : string, host='', user='', pwd='' )=>DataModel
```

2 Create Table

If the data table corresponding to class **Gene** does not exist yet, one can create it by,

```
# create a table for class Gene
model.CreateTable( Gene );
```

A table named **Gene** will be created.

Prototype of the method,

```
DataModel.CreateTable( klass )
```

If a class has a special constant string field named `__TABLE_NAME__`, the name of database table corresponding to the class will be the value of `__TABLE_NAME__`. A class may also contain another constant string field named `__TABLE_PROPERTY__`, which indicates the property of the table or contains extra constraints on some fields.

3 Insert Record

Then one can insert to the database an instance of class **Gene** as a record by,

```
gene = Gene{ 0, 'RAS', '...AATCCG...' };  
# insert a record into the table  
model.Insert( gene );
```

The module knows which table to insert. After inserting, if the class has a field with type **INT_PRIMARY_KEY_AUTO_INCREMENT**, that instance field (in this case it is *gene.id*) will be filled with id of the inserted record.

One can also insert multiple records at once by pass a list of class instances to the **Insert()** method. This method will return a database handler, with which one can insert more records to avoid repeatedly compile SQL statements internally.

Similarly there is `Delete()`,

```
DataModel.Insert( object ) => Handler  
DataModel.Delete( object ) => Handler
```

4 Query Database

Then one can perform querying by,

```
# SELECT * FROM Gene WHERE name = 'RAS';  
hd = model.Select( Gene ).Where().EQ( 'name', 'RAS' );  
# query and store the result into 'gene' object:  
hd.QueryOnce( gene );
```

this module uses a series of method calls to construct a SQL statement and then compile it. The following two methods can be used to create a handler for these two types of query,

```
DataModel.Select( object, ... ) => Handler  
DataModel.Update( object, ... ) => Handler
```

These two methods can take a class or a list of classes as parameters, which indicate which database tables to be queried. And after each of the class parameter, there can follow an integer, which indicates how many fields to be queried for that class/table. For example,

```
hd = model.Select( Gene, 2 ).Where().EQ( 'name', 'RAS' );
```

this will generate such SQL statement,

```
# SELECT id,name FROM Gene WHERE name = 'RAS';
```

Then the handler can call the following methods to prepare the SQL statement for a query,

```
# WHERE  
Handler.Where( ) => Handler  
# SET field=value, or, SET field=?  
Handler.Set( field : string, value=nil ) => Handler  
# SET field=field+value, or, SET field=field+?  
Handler.Add( field : string, value=nil ) => Handler  
# field=value, or, field=?  
Handler.EQ( field : string, value=nil ) => Handler  
# field!=value, or, field!=?  
Handler.NE( field : string, value=nil ) => Handler  
Handler.GT( field : string, value=nil ) => Handler  
Handler.GE( field : string, value=nil ) => Handler  
Handler.LT( field : string, value=nil ) => Handler  
Handler.LE( field : string, value=nil ) => Handler  
# SET table.field=value, or, SET table.field=?  
Handler.Set( table, field : string, value=nil ) => Handler  
Handler.Add( table, field : string, value=nil ) => Handler  
Handler.EQ( table, field : string, value=nil ) => Handler  
Handler.NE( table, field : string, value=nil ) => Handler  
Handler.GT( table, field : string, value=nil ) => Handler  
Handler.GE( table, field : string, value=nil ) => Handler  
Handler.LT( table, field : string, value=nil ) => Handler  
Handler.LE( table, field : string, value=nil ) => Handler  
# field IN ( values ), or, field IN ?  
Handler.In( field : string, values={} ) => Handler  
Handler.In( table, field : string, values={} ) => Handler  
# OR
```

```

Handler.Or( ) => Handler
Handler.And( ) => Handler
Handler.Not( ) => Handler
# (
Handler.LBrace( ) => Handler
# )
Handler.RBrace( ) => Handler
# table1.field1=table2.field2
Handler.Match( table1, table2, field1='', field2='' ) => Handler
# ORDER BY field ASC/DESC
Handler.Sort( field : string, desc=0 ) => Handler
# ORDER BY table.field ASC/DESC
Handler.Sort( table, field : string, desc=0 ) => Handler
# LIMIT limit, or, LIMIT limit OFFSET offset
Handler.Range( limit : int, offset=0 ) => Handler

```

For the methods that take an optional parameter *value* , if it is omitted, a place holder variable will be used, then data can be bind to these variables by,

```

Handler.Bind( value, index=0 ) => Handler

```

A index can be given in the parameter list to indicate which to bind, if there is multiple place-holder variables. If no index parameter is given, the binds will be done sequentially.

At last these two methods can be called to perform the query,

```

Handler.Query( ... ) => int
Handler.QueryOnce( ... ) => int

```

These two method will take class instance(s) as parameter, and store the resulting data in the members of the class instances, if the query is successful. These method will return 1 when query is successful, otherwise return 0. If the query hits multiple records, Handler.Query() can be called repeatedly to get the result. After calling Handler.Query(), it is necessary to call Handler.Done() to reset the model. If Handler.QueryOnce() is called, there will be no such necessary.

5 Other Methods

```

DataModel.Query( sql : string ) => int

```

Perform a arbitrary query, return the status of the query.

```
Handler.sqlstring( ) => string
```

Return the SQL statement as a string.

```
Handler.Insert( object ) => int
```

Use the handler created by *DataModel.Insert()* to insert more records.