

Web Programming Using Dao

Limin Fu (phoolimin@gmail.com)

September 19, 2009

This document will not explain how to do web programming, instead, it will just show how to use Dao as a server side scripting language. Server side web scripts usually interact with a web server (such as Apache) through the Common Gateway Interface (CGI) protocol. Web scripts can access the information of a HTTP request using CGI protocol, and decide how to respond to the request. Normally, web scripts also have to query a database, and present the results in a good way to the client browser.

1 Using CGI Module

DaoCGI is a module to parse the information of a HTTP request, and presents the information in a structured way (as Dao variables) to Dao web scripts.

The CGI module can be loaded by

```
load DaoCGI;
```

After loading, there will be several global variables imported into the current namespace:

| Name | Type | Use |
|-------------|---------------------------|--|
| HTTP_ENV | map<string,string> | environment information about the HTTP request |
| HTTP_GET | map<string,string> | variables for a GET request |
| HTTP_POST | map<string,string> | variables for a POST request |
| HTTP_GETS | map<string,list<string> > | variables for a GET request |
| HTTP_POSTS | map<string,list<string> > | variables for a POST request |
| HTTP_COOKIE | map<string,string> | cookies for the request |
| HTTP_FILE | map<string,stream> | uploaded files for the request |

For *HTTP_GETS*, *HTTP_POSTS* , there could be multiple values for the same key. For *HTTP_FILE* , the key is the upload name for the file:

```
<input type="file" name="upload_name" id="filename"/>
```

and the value is an IO stream containing the uploaded file.

A simple CGI program using Dao could be,

```
load DaoCGI;
stdio.println( 'content-type: text/plain\n' );
stdio.println( 'You are from:', HTTP_ENV[ 'REMOTE_ADDR' ] );
stdio.println( 'Welcome to visit:', HTTP_ENV[ 'REQUEST_URI' ] );
```

2 Database Handling

Currently Dao only has a module to handle MYSQL database. The way this module handles database is by mapping Dao classes to database tables (class members to table fields), and by using class instances to operate on database tables. For the details, please refer to the *Database Handling by DaoDataModel* .

3 Simple Way to Separate Model and View

It is normally the responsibility of server side web scripts to generate HTML codes to present some information to a web browser. Such information or data are often stored in files or databases, and have to be processed in certain way (model). If the information is going to be presented to a web browser, it is preferable to achieve certain visual appearance by generating appropriate HTML codes (view).

Obviously, it is better to separate model and view, so that it becomes much easier to modify one of them without affecting the other, resulting web applications that are easy to maintain or upgrade. The DaoDataModel suit well the model part of a web application. While for the view part, several string methods are available in Dao to make thing much easier. In the following, I will use an example to explain this.

Suppose there is a database table named *Friend* for an address book with the following information:

```
id : INTEGER PRIMARY KEY AUTOINCREMENT
name : CHAR(50)
```

```
phone : CHAR(20)
city : CHAR(50)
street : VARCHAR(100)
```

and the corresponding Dao class would be:

```
typedef tuple<id:string,name:string,phone:string,city:string,street:string> friend_t;

class Friend
{
    my id : INT_PRIMARY_KEY_AUTOINCREMENT;
    my name : CHAR50;
    my phone : CHAR20;
    my city : CHAR50;
    my street : CHAR100;

    routine AsTuple() => friend_t {
        return ( (string) id, name, phone, city, street );
    }
}
```

Here a tuple type (friend_t) with named items is also defined to be the basic data structure to interface between the model and view part of the application.

Now suppose again, the database is open as,

```
global model = DataModel( 'dbname', 'host', 'user', 'password' );
```

Now in the model part, querying the database table is extremely simple. For example, one would like to find all friends in certain city, this can be done as,

```
routine FriendInCity( city : string ) => list<friend_t>
{
    friends = {};
    friend = Friend();
    hd = model.Select( Friend ).Where().EQ( 'city', city );
    while( hd.Query( friend ) ) friends.append( friend.AsTuple() );
    hd.Done();
    return friends;
}
```

Obviously this function can also be defined as a member method of class **Friend** .

Creating view can also be done in a simple way by using template HTML codes. For example, if the queried friends need to be displayed in a table, one can define the following template,

```
<tr class="myrowstyle"><td class="mycellstyle">@(id)</td>
<td>@(name)</td><td>@(phone)</td>
<td>@(city)</td><td>@(street)</td></tr>
```

The template will define which fields to be displayed, and how is the layout. The color and font information can also be included in the template, but it is easier to set them by using Cascade Style Sheet (CSS).

The generation of view would be,

```
routine ViewFriends( friends : list<friend.t>, rowtpl = '' )
{
    html = '<table class="mytablestyle">\n';
    for( friend in friends ) html += rowtpl.expand( friend, '@' );
    html += '</table>';
    return html;
}
```

Here the method *rowtpl.expand()* will expand the *rowtpl* string and substitute the placeholders by the tuple items of *friend* with corresponding item names. A map (associative array) could also be used in the place of the tuple parameter of **expand()** .

So to query the friends and view them, one could use,

```
row = '<tr class="myrowstyle"><td class="mycellstyle">@(id)</td>
<td>@(name)</td><td>@(phone)</td>
<td>@(city)</td><td>@(street)</td></tr>';

friends = FriendInCity( 'Beijing' );
view = ViewFriends( friends, row );
```

By changing the template html codes (as well as CSS), the appearance of the view can be easily changed.