

Programming with C interface to Dao

Limin Fu (phoolimin@gmail.com)

September 19, 2009

Contents

1	Introduction	1
2	Build A Simple Extension	2
3	Wrapping C/C++ Functions	5
4	Wrapping C/C++ Types	6
5	Embedding Dao	9
6	Advanced Using of The C Interfaces	12

(For Dao 1.1)

This document is licensed under *GNU Free Documentation License* .

1 Introduction

This document covers the methods for extending and embedding of Dao through C interfaces defined in the header file *dao.h* . The C interfaces are designed in such a way that the extension or the application that embeds Dao can be built using only the header file *dao.h* , namely, without linking against the Dao library. This is achieved though an intermediate wrapping layer, to skip this layer, one needs only to define *DAO_DIRECT_API* before including *dao.h* . However, the direct interfaces and the interfaces with a wrapping layer are identical, so it is just a matter of choice to use the direct interface or the wrapped one.

Obviously, one can also use other header files of the Dao library, to interact more directly with Dao, but the other header files may change from release to release, while

dao.h is intended to be more stable. Using only *dao.h* makes it easier for the extension or application to work with updated Dao library. Moreover, there is a tool that can wrap C/C++ libraries in a semi-automated way, and this tool is updated whenever *dao.h* is changed. So if the extension is built using this tool, whenever *dao.h* is changed, it is only necessary to run again this tool to generate the extension.

2 Build A Simple Extension

A Dao extension is a module written in C that can provide extended functionalities for Dao. A Dao extension should always define two mandatory variables and one mandatory function. The first of the two variables is an integer signature of the header file *dao.h* that is used to build the extension, and the second is a global variable as the wrapping layer for the indirect interfaces. These two variables can be defined using macro *DAO_INIT_MODULE*. The mandatory function is the function to be called when the module is loaded. This function should be named as *DaoOnLoad()* with the following form,

```
int DaoOnLoad( DaoVmSpace *vms, DaoNameSpace *ns )
{
    ...
}
```

The two parameters of this functions are a structure *DaoVmSpace* which is the virtual machine space (VM Space) that defines an environment for the virtual machine, and a structure *DaoNameSpace*, which is used to store the data and types defined by the module. In the function body of *DaoOnLoad()*, one should add the wrapped constants, functions and C/C++ types to the *DaoNameSpace*. The returned value of this function is not used yet, it is reserved for future use. Now I will first introduce the form of a C function that is callable in Dao, then I will show how to add such functions to the *DaoNameSpace*.

2.1 Callable Function for Dao

A C function that is to be called in Dao must has the following form,

```
void cfunction( DaoContext *ctx, DValue *p[], int N );
```

Where *DaoContext* is the structure for storing execution context, and *DValue* is the basic structure used in the Dao library to store data. *ctx* is the current context that invokes the C function, and *p* is an array of parameters passed to this call and *N* is the number

of parameters presented in the call. p may contain more elements than N , if the function is called without one or more parameters that have default values. While the C function receives data directly through p , the returned data for Dao must be put into ctx using some functions.

2.2 The Dao Value Structure: DValue

The *DValue* structure is shown below, and it has the follow simple fields: the t field indicates the type of the data; the $sub, cst, ndef$ fields are mainly for internal use, and should be set to zero; the v field contains the value of the data.

```
struct DValue
{
    uchar_t t; /* type */
    uchar_t sub; /* sub-type */
    uchar_t cst; /* const */
    uchar_t ndef; /* not a default parameter */
    union {
        dint      i; /* int */
        float     f; /* float */
        double    d; /* double */
        complex16 *c; /* complex */
        DString   *s; /* string */
        DLong     *l; /* big integer */
        DaoBase   *p; /* NOT one of the above data types */
        DaoArray  *array;
        DaoList   *list;
        DaoMap    *map;
        DaoPair   *pair;
        DaoTuple  *tuple;
        DaoRoutine *routine;
        DaoFunction *func;
        DaoObject *object;
        DaoClass  *klass;
        DaoCData  *cdata;
        DaoContext *context;
        DaoStream *stream;
        DaoNameSpace *ns;
        DaoVmProcess *vmp;
    } v ;
};
```

2.3 A Simple and Complete Example

```
#include"dao.h"
```

```

DAO_INIT_MODULE;
void salute( DaoContext *ctx, DValue *p[], int N )
{
    DaoContext_PutMBString( ctx, "hello" );
}
int DaoOnLoad( DaoVmSpace *vms, DaoNameSpace *ns )
{
    DaoNameSpace_AddFunction( ns, salute, "salute()=>string" );
    return 0;
}

```

In this example, there are only two functions which have not been introduced before. The first function *DaoContext_PutMBString()* is used to put a string as returned value to the execution context, in this case, *salute()* will return "hello".

There are other functions that can be used to put different types of data into an execution context, they all have "DaoContext.Put" as part of their names. Please look into the header file *dao.h* for the complete list of such functions. The second function *DaoNameSpace_AddFunction()* is used to register the *salute()* function to the namespace *ns* with function prototype defined as "salute()=>string". The function prototype should be written in the same way as a Dao routine, in this case, this prototype tells the Dao VM that this added function will have "salute()" as its name in Dao codes, and it will return a string as returned value.

After compiling this code into a "mod_salute.dll" or "mod_salute.so" file, one can use this module as the following,

```

load mod_salute;
stdio.println( salute() );

```

2.4 Adding Multiple Functions

If the extension defines multiple functions, these functions can be grouped together and added to the namespace by one call. To do this, one just has to create an array of *DaoFuncItem*, in which the first field is the function pointer of the Dao-callable C function, and the second field is the function prototype used by Dao. This array must be ended by an item containing null fields. For example, the following is the *DaoFuncItem* array for the wrapped OpenGL functions defined in the *DaoOpenGL* module,

```

static DaoFuncItem dao_Funcs[] =
{
    { dao__glClearColor, "glClearColor( c : float )" },
    { dao__glClearColor, "glClearColor( red : float, green : float, blue : float, alpha

```

```

: float )" },
  { dao__glClear, "glClear( mask : int )" },
  { dao__glIndexMask, "glIndexMask( mask : int )" },
  { dao__glColorMask, "glColorMask( red : int, green : int, blue : int, alpha : int )"
},
  { dao__glAlphaFunc, "glAlphaFunc( func : int, ref : float )" },
  { dao__glBlendFunc, "glBlendFunc( sfactor : int, dfactor : int )" },
  { dao__glLogicOp, "glLogicOp( opcode : int )" },
  .....
  { NULL, NULL }
};

```

Then the *DaoFuncItem* array can be passed to function *DaoNameSpace_AddFunctions()* to added all the functions in the array to a namespace,

```

DaoNameSpace_AddFunctions( ns, dao_Funcs );

```

3 Wrapping C/C++ Functions

To wrap a C function, one only has to create a Dao-callable C function and call the C function to be wrapped with simple conversions of parameter/returned values. For example, if we have the following *sin()* function to be wrapped,

```

double sin( double x );

```

Then the wrapping of *sin()* is as simple as the following,

```

void dao_sin( DaoContext *ctx, DValue *p[], int N )
{
  DaoContext_PutDouble( ctx, sin( p[0]->v.d ) );
}
...
int DaoOnLoad( DaoVmSpace *vms, DaoNameSpace *ns )
{
  ...
  DaoNameSpace_AddFunction( ns, dao_sin, "sin( x : double ) => double" );
  return 0;
}

```

One don't have to worry if the accessing of the parameter of a double value by *p[0]->v.d* is valid, because this functions is registered as "sin(x : double) => double", so that the Dao VM will perform parameter checking (at both compiling time and running time) and guarantee that the function will be called with a double as parameter.

Note that there is a tool that can do this type of wrapping automatically, please refer to the relevant documentation to see how this tool can be used. If used properly, this tool can also wrap C structures and C++ classes.

4 Wrapping C/C++ Types

What about using C structures or C++ classes in Dao? This section will explain the basics for wrapping C/C++ types.

4.1 Dao Typer Structure

Basically, a C/C++ type can be used in Dao as long as it is associated with a typer structure and is registered into the namespace. The typer structure is defined as the following,

```
/* Typer structure, contains type information of each Dao type: */
struct DaoTypeBase
{
    DaoTypeCore *priv; /* data used internally; */
    const char *name; /* type name; */
    DaoNumItem *numItems; /* constant number list */
    DaoFuncItem *funcItems; /* method list: should end with a null item */

    /* typer for super types, to create c type hierarchy;
     * mainly useful for wrapping c++ libraries. */
    DaoTypeBase *supers[ DAO_MAX_CDATA_SUPER ];

    void* (*New)(); /* function to allocate structure; */
    void (*Delete)( void *self ); /* function to free structure; */
};
```

The typer includes some basic properties of the C/C++ type, and its wrapped member constant numbers and methods. Normally, the function pointers to allocate and free the C/C++ type should also be supplied. If there is a method in the method list *funcItems* having the same name as the type name (*name* field), the *New* field will not be used. If the *Delete* field is missing, the type will be freed by the standard C function *free()*. Optionally, if the C/C++ type has parent type(s), the typer structures of such parent types can be filled in the *supers* field (the last item in this field must be NULL), so that the inheritance relations in C/C++ are also valid in Dao, and the method invocation of parent methods and the casting to a parent type will be handled properly by the Dao VM.

The Dao VM stores C/C++ types in wrapped form, which is defined as structure *DaoCData*. The *DaoCData* structure has a field named *data*, which is a pointer pointing to the

stored C/C++ data. The stored C/C++ data can be set using `DaoCData_SetData(DaoCData *cdata, void *d)` and retrieved using `DaoCData_GetData(DaoCData *cdata)`.

4.2 A Simple Example

Suppose we have the following simple C type name "foo" and a simple C function named "bar" that operates on "foo" type,

```
typedef struct foo { int value; } foo;
foo* foo_new()
{
    return malloc( sizeof( foo ) );
}
void bar( foo *f )
{
    printf( "bar() is called with foo object: %p\n", f );
}
```

4.2.1 Basic Wrapping

The wrapping of the "foo" is the following,

```
static DaoTypeBase fooTyper =
{
    NULL,
    "foo",
    NULL, /* no constant number */
    NULL, /* no method */
    { NULL }, /* no parent type */
    foo_new,
    NULL /* free by free() */
};
```

The this typer can be registered in a Dao VM space by,

```
DaoNameSpace_AddType( ns, &fooTyper );
```

Then the type name can be used in Dao, for example, to create a "foo" object, one can do,

```
f = foo();
```

The function *bar()* can be wrapped and add into VM space by,

```
static void dao_bar( DaoContext *ctx, DValue *p[], int N )
{
    foo *f = DaoCData_GetData( p[0]->v.cdata );
    bar( f );
}
...
DaoNameSpace_AddFunction( ns, dao_bar, "bar( f : foo )" );
...
```

4.2.2 bar() As Method of foo?

Since the "bar()" function operates on "foo" object, what about declaring "bar()" as a method of "foo" type, so that instead of calling "bar(foo_object)", one can call "foo_object.bar()"?

This can be done easily, one needs only to modify the typer of foo as the following,

```
static DaoFuncItem fooMeth[] =
{
    { dao_bar, "bar( self : foo )" },
    { NULL, NULL }
};
static DaoTypeBase fooTyper =
{
    NULL,
    "foo",
    NULL, /* no constant number */
    fooMeth, /* one method bar() */
    { NULL }, /* no parent type */
    foo_new,
    NULL /* free by free() */
};
```

The first parameter in the function prototype of the method must be named as "self", if one wants "foo_object" to be passed to "bar()" as the first parameter when "foo_object.bar()" is called.

With this modification, one can do the following in Dao,

```
f = foo();
f.bar();
```

4.2.3 Setters and Getters

Setters and getters can be easily implemented for more conveniently accessing fields of C/C++ types. Continue the previous example, the setter and getter of the "value" field of "foo" can be defined as,

```
static void foo_SETF_value( DaoContext *ctx, DValue *p[], int N )
{
    foo *f = DaoCData_GetData( p[0]->v.cdata );
    f->value = p[1]->v.i;
}
static void foo_GETF_value( DaoContext *ctx, DValue *p[], int N )
{
    foo *f = DaoCData_GetData( p[0]->v.cdata );
    DaoContext_PutInteger( ctx, f->value );
}
```

Then the following two lines should be added to the "fooMeth",

```
{ foo_SETF_value, ".value=( self : foo, v : int )" },
{ foo_GETF_value, ".value( self : foo ) => int" },
```

Please note that, their method prototypes are exactly the same as overload operators in Dao classes.

Then in Dao, the "value" of "foo" can be set and got as,

```
f = foo();
f.value = 123;
a = f.value;
```

5 Embedding Dao

So far, some basics of the C programming interfaces have been introduced in the context of extending dao, now in this section, I will give a brief introduction on how to embed the Dao virtual machine into other applications. Embedding Dao can make an application programmable, and allow some components of the application to be controlled in a programmable way, or allow the functionalities of the application to become extendable by the Dao language.

The interfaces I introduced so far for extending are also valid for embedding, the only differences are that the macro *DAO_INIT_MODULE* is no longer necessary here and there is no need to provide function *DaoOnLoad()*. To embed Dao, the very first thing is to include

the header file "dao.h", similar to extending, you should define "DAO_DIRECT_API" before including "dao.h" if the direct interfaces are to be used. If indirect interfaces are used, one global variable must be defined, which is `__dao` (two underscore symbols),

```
DAO_DLL DaoAPI __dao;
```

you can also simply use the macro `DAO_INIT_MODULE`.

If you are not going to compile Dao together with your application, then the next thing is to open the Dao library file using a macro defined in `dao.h`,

```
void *handle = DaoLoadLibrary( "/usr/local/dao/dao.so" );
```

then function `DaoInit()` must be called to initialize the Dao library, this function will return a structure of `DaoVmSpace`, which can be used to load modules or compile Dao source codes. If direct interfaces are used, this function can be called directly,

```
DaoVmSpace *vmSpace = DaoInit( NULL );
```

Otherwise, one has to do the following,

```
DaoInitFunc pfunc = (DaoInitFunc)DaoFindSymbol( handle, "DaoInit" );  
DaoVmSpace *vmSpace = (*pfunc)( & __dao );
```

In the second way, calling `DaoInit()` with the address of `__dao` as parameter will allow the function pointer fields of `__dao` to be filled, so that the interfaces can be accessed through a wrapper layer provided by `__dao`, which allows the application to be built without linking against the Dao library.

At this point, you can use `DaoVmSpace_RunMain()` to run a script file,

```
if( ! DaoVmSpace_RunMain( vmSpace, "script.dao" ) ) return;
```

The file "script.dao" will be searched in the searching paths of the VM space.

Or you can use `DaoVmSpace_Load()` to load a script file to obtain a namespace structure. If it was already loaded before, it will execute the codes in the file and return the previously created namespace structure, otherwise it will return a new one. The file will also be searched in the current fold and in the searching paths of the VM space.

```
if( ! DaoVmSpace_Load( vmSpace, "script.dao" ) ) return;
```

You may also use the following functions to obtain a namespace structure,

```
DaoNameSpace *ns = DaoNameSpace_New( vmSpace );
```

Or,

```
DaoNameSpace *ns = DaoVmSpace_MainNameSpace( vmSpace );
```

to get the main namespace of a VM space structure.

Or,

```
DaoNameSpace *ns = DaoNameSpace_GetNameSpace( nameSpace, "name" );
```

to get a namespace with given name from another namespace. If it already exists, return it, otherwise return a new one, and put it in that namespace with name "name".

As introduced before, with namespace structure, one can add types and C functions, or constants to the namespace. It can also be used to load script file that is not searched in the VM space paths, The functions, class, global constants and variables will be put into this namespace structure.

```
DaoNameSpace_Load( ns, "script.dao" );
```

To compile and execute scripts, or run Dao functions, one need to create another type of structure **DaoVmProcess**, which can be obtained from DaoVmSpace structure or create from it,

```
DaoVmProcess *vmp = DaoVmProcess_MainProcess( vmSpace );  
DaoVmProcess *vmp2 = DaoVmProcess_New( vmSpace );
```

With DaoVmProcess structure, one can compile or evaluation a script, or call Dao function

```
DString *src = DString_New(1);  
DString_SetMBS( src, "io.writeln( 'hello' )" );  
DaoVmProcess_Eval( vmp, ns, src, 0 );  
  
DString_SetMBS( src, "routine hello(){io.writeln( 'hello' ) }" );  
DaoVmProcess_Compile( vmp, ns, src, 0 );  
// routine "hello()" is in the namespace "ns"
```

To call a Dao function,

```
DValue value = DaoNameSpace_FindData( ns, "myfunc" );
if( value.t == DAO_ROUTINE ){
    DaoVmProcess_Call( vmp, value.v.routine, NULL, NULL, 0 );
}
```

To pass parameter(s) to the call,

```
DValue buf[3];
DValue *par[3];
for(i=0; i<3; i++) par[i] = & buf[i];
buf[0] = DValue_NewInteger(1);
buf[2] = DValue_NewFloat(2.0);
buf[3] = DValue_NewMBString( "abc", 3 );
DaoVmProcess_Call( vmp, func, NULL, par, 3 );
DValue_ClearAll( buf, 3 );
```

DaoVmProcess structure can also be used to run a Dao class method, in case that the method needs a class instance, the instance object should be passed to DaoVmProcess.Call() as the third parameter.

6 Advanced Using of The C Interfaces

As mentioned before, it is possible to retain or define inheritance relationships between C/C++ types. Moreover, in Dao, one can derive sub-classes from C/C++ types just in the same way as deriving from Dao classes. It is even possible to wrap a C++ class in such way that its virtual methods can be overrode by Dao routines when deriving a Dao class from the C++ class. For C structures that contain fields of function pointers, in certain cases, these C structures can be wrapped as if they are C++ class and have virtual methods of the same name as their function pointer fields. These techniques are a bit complicated, and will not explain here. Anyone who are interested can refer to the source codes for some Dao modules (e.g. module for VTK5), or can use the automated tool ("tools/autobind.dao") to generate such wrappers and then examine the generated codes.