

使用道语言的C函数接口编程

傅利民 (phoolimin@gmail.com)

September 19, 2009

Contents

1	简介	1
2	编写一个简单的C模块	2
3	封装C/C++函数	5
4	封装C/C++数据类型	6
5	将道嵌入到其他程序里	9
6	道语言的C编程接口的高级应用 (适用道语言1.1)	12

1 简介

本文档主要介绍如何使用道语言的C函数接口编写道语言的扩展模块，或在其他应用程序中嵌入道语言虚拟机。这些函数接口都由单一头文件`dao.h`定义。这些函数即可以作为函数直接调用，也可以通过宏以函数指针的形式调用。这两种调用方式的存在使得C/C++模块或应用程序的编译比较灵活，即可以在避免编译时链接dao语言的库。要使用直接函数调用，用户必须在包含头文件`dao.h`前定义`DAO_DIRECT_API`，否则间接调用将被使用。当然在开发dao扩展或嵌入dao时，也可以使用道语言库的其他头文件。不过这有时将使得模块的维护和升级到dao的更高版本不太方便，因为那些头文件在不同的发布中可能会有所不同。而头文件`dao.h`将更稳定，不同版本间的`dao.h`会比其他头文件更一致。仅使用`dao.h`的模块或应用程序将更好维护，并更容易升级到更高的道语言版本。

2 编写一个简单的C模块

道语言的功能可以通过编写C/C++模块得到扩展，这样的模块一般都会定义一些可供道语言使用的C/C++函数或数据类型（如DaoOpenGL等模块），也可仅向添加一些特殊的全局变量（如DaoCGI等模块）。下面将通过一个极其简单的例子来介绍如何编写道语言的C模块，基于C++的模块将在后面涉及。

道语言的C模块必须定义两个必要的变量，一个是一个整数用来保存头文件*dao.h*的版本标号；另一个是一个全局变量，用来保存一组用来实现间接接口的函数指针。这两个变量可用宏*DAO_INIT_MODULE*来定义。模块还必须定义一个入口函数，这个函数将在模块被载入时被自动调用，以实现模块的初始化工作，并让模块在道虚拟机里注册它所定义的函数和数据类型。这个函数有如下形式：

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

这个函数的两个参数就是一个*DaoVmSpace*（虚拟机空间）结构，这个结构保存了当前虚拟机运行的环境信息，如当前的命名空间等，和一个命名空间结构，模块可以通过这个结构注册它所定义的函数和数据类型。目前这种函数的返回值还没被使用。下面我将介绍可被道语言调用的C函数形式，及如何将这样的函数注册到*DaoNameSpace*里。

2.1 道语言可调用的C函数

可以在道语言里调用的C函数必须有如下形式：

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

这里*DaoContext*是一个保存道语言函数的当前运行状态和环境的结构（如局部变量的值，当前虚拟机指令，当前运行函数的信息等），实际上*DaoContext*总是对应于一个道语言函数的调用，保存着这个函数调用的栈数据。而*DValue*是道语言库用来保存数据的基本结构。*ctx*是当前调用这个C函数的Dao函数的栈数据，*p*是从道运行环境传递到此C函数的一组函数调用参数，而*N*是此函数调用中出现的参数数目。*p*实际上可含有比*N*多的元素，因为使用缺省值的参数没被计算在*N*内。这个C函数直接从*p*接受从道运行环境传递来的参数，但返回到道运行环境的值必须使用相关函数设定到*ctx*里。

2.2 道语言数据的值结构DValue

下面的代码显示了*DValue* 的定义，这个结构包含了以下简单的成员域：*t* 表示此数值的类型；*sub,cst,ndef* 被道语言库内部使用，一般应设定为零；*v* 即为值，其含义可在下面的定义中看出。

```
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 一个简单而完整的例子

```
#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( nss, salute, "salute()=>string" );
    return 0;
}

```

这个例子使用了两个未曾介绍的函数。第一个函数是*DaoContext_PutMBString()*，它可以被用来将一个字符串作为返回值设定到*ctx* 里，在这个例子中*salute()* 由这个函数设定为返回”hello”。还有其他类似的函数用来设定其他类型的返回值，这些函数名均以*DaoContext_Put* 开头。请参看头文件*dao.h* 以了解有哪些这样的函数可用。

第二个函数是*DaoNameSpace_AddFunction()*，是用来注册C函数到道虚拟机里。这里*salute()* 将以”salute()=>string”为函数原型注册到道虚拟机空间*vms* 里。被注册的函数的原型必须按道语言里的函数原型的方式定义。从这个函数原型定义，道虚拟机将知道这个C函数在道语言里以”salute()”为名字，并返回一个字符串为返回值。

如果这个模块被编译为名为”mod_salute.dll”或”mod_salute.so”的动态链接库，那么这个模块可在道语言里这样用：

```

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

```

2.4 同时注册多个函数

如果一个模块定义了多个函数，这些函数可放在一个数组里，一起注册到道虚拟机里。这个数组的元素必须是*DaoFuncItem*，它的第一个成员是指向C函数的指针，第二个成员是一字符串，定义这个C函数在道语言里的函数原型。这个数组必须以空值元素结束。例如，下面的代码取自*DaoOpenGL*模块，这个*DaoFuncItem* 数组保存了那些被封装的OpenGL函数的函数指针，和它们在道语言里被使用的函数原型。

```

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 }  
};
```

然后，*DaoFuncItem* 数组可作为参数传递给 *DaoNameSpace_AddFunctions()* 函数来把数组里的C函数加到道虚拟机里：

```
DaoNameSpace_AddFunctions( ns, dao_Funcs );
```

3 封装C/C++函数

要封装一个C/C++函数，只须定义一个道语言里可调用的C函数，并在这个函数里调用那个被封装的函数即可。当然这个函数还要做些参数和返回值转换的工作，不过这样的转换很简单。

作为一个简单的例子，如果我们要封装如下的 *sin()* 函数：

```
double sin( double x );
```

sin() 函数的封装可以简单的实现为下面的代码：

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

这里双精度实数参数可由 *p[0]->v.d* 直接获得，这个参数值的类型不需要在这里作检查，因为这个函数被注册为了“*sin(x : double) => double*”，道虚拟机将在编译时和运行时作参数检查，保证这个函数以正确的参数调用。不过通常情况下，返回值的正确性必须由封装函数保证。

请注意：道语言库附带了一个半自动工具，可用来方便的封装C/C++函数和数据类型。请参看有关的文档。

4 封装C/C++数据类型

前面已经介绍了如何封装C/C++函数，那么如何才能可以在道语言里使用C/C++数据类型呢？下面将作这方面的介绍。

4.1 道语言Typer结构

实际上，一般的C/C++数据类型都可以在道语言里使用，只要有个道语言的Typer结构被关联到这个类型，并被注册到道虚拟机里。Typer结构的定义如下：

```
/* Typer结构，包含类型的基本信息 */
struct DaoTypeBase
{
    DaoTypeCore *priv; /* 内部使用成员 */
    const char *name; /* 类型名称 */
    DaoNumItem *numItems; /* 成员常数，以空元素结束数组 */
    DaoFuncItem *funcItems; /* 成员函数，以空元素结束数组 */

    /* 父类型的Typer结构，用于构建类型继承关系，主要在封装C++库时比较有用 */
    DaoTypeBase *supers[ DAO_MAX_CDATA_SUPER ];

    void* (*New)(); /* 类型的内存分配 */
    void (*Delete)( void *self ); /* 类型的内存释放 */
};
```

这个结构包含C/C++数据类型的基本属性和它的成员常数和函数等。通常应在这个结构里设定分配和释放类型内存的函数指针。但如果在成员函数里有与类型名称同名的函数，那么这样的函数（允许多个重载的函数）将被认为是类型的分配函数，*New* 所设定的内存分配函数将被忽略。如果*Delete* 所表示的内存释放函数未被设定（即被设为NULL），那么C标准库里的*free()* 函数将被使用。

如果这个结构所表示类型是从另一个类型继承而来，那么它的父类的Typer结构可放在*supers* 数组里，这个数组的最后一个元素必须是NULL。这样C/C++里的类型继承关系将继续在道语言里有效，道虚拟机将恰当的处理成员函数的调用和类型的映射。

道虚拟机将封装的C/C++数据保存在*DaoCData* 结构里，这个结构有名为*data* 成员，它保存着指向C/C++数据的指针。这个*data* 成员可由函数 *DaoCData_SetData(DaoCData *cdata, void *d)* 设定，并可由函数 *DaoCData_GetData(DaoCData *cdata)* 获取。

4.2 一个简单的例子

假设我们有个C类型”foo”和使用”foo”的函数”bar”:

```
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 基本封装

”foo”类型的基本封装如下:

```
static DaoTypeBase fooTyper =
{
    NULL,
    "foo",
    NULL, /* 没有成员常数 */
    NULL, /* 没有成员函数 */
    { NULL }, /* 没有父类型 */
    foo_new,
    NULL /* 由free()释放 */
};
```

这个Typer结构可按如下方式注册到道虚拟机空间:

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

这样在道语言里就可以使用”foo”类型了, 例如, ”foo”对象可这样直接创建:

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

而函数bar()可以这样封装后注册到道虚拟机空间:

```
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 把bar()定义为foo的成员方法?

既然"bar()"函数使用了"foo"对象，那是否可把"bar()"定义为"foo"类型的成员方法，然后使用"foo_object.bar()"调用"bar()"函数，而不需"bar(foo_object)"这样调用？

其实这可以很容易的做到，所要做的就是修改"foo"的Typer结构如下：

```
static DaoFuncItem fooMeth[] =
{
    { dao_bar, "bar( self : foo )" },
    { NULL, NULL }
};
static DaoTypeBase fooTyper =
{
    NULL,
    "foo",
    NULL, /* 没有成员常数 */
    fooMeth, /* 有一个成员方法bar() */
    { NULL }, /* 没有父类型 */
    foo_new,
    NULL /* 由free()释放 */
};
```

成员方法的函数原型的第一个参数必须名为"self"，如果"foo_object.bar()"调用时"foo_object"需要作为第一个参数传递给"bar()"的话。

有了这个修改后，下面的代码将可以在道语言里使用：

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

4.2.3 Setters和Getters

道语言的C编程接口支持给C/C++数据类型定义Setters和Getters，这样将可以更方便的访问C/C++数据的成员变量。继续前面的例子，可以给"foo"的成员变量"value"定义如下的setter和getter:

```
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 );
}
```

下面这两行必须被加到”fooMeth”里:

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

这里注册的函数原型必需和定义道类的运算符重载的函数原型一样。之后, ”foo”的成员变量”value”的值可以这样设定和获取:

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

5 将道嵌入到其他程序里

到目前为止, 一些基本的编程接口以编写道语言的扩展模块为例子做了介绍。下面将简单地介绍如何将道虚拟机嵌入到其他应用程序里。在应用程序里嵌入道将使得程序的一些功能可由道脚本设定和控制, 并允许它的功能可由道语言来扩展。

在这里, 前面介绍的接口还是有效。不过宏`DAO_INIT_MODULE`已变得不必要, 也不需要定义入口函数`DaoOnLoad()`。明显, 要嵌入道, 第一件事就是在程序里包含头文件”dao.h”, 与编写模块类似, 如果要使用直接的编程接口, 你必须在包含”dao.h”前定义”`DAO_DIRECT_API`”。如要使用间接的接口, 全局变量`__dao` (两条下划线)必须被定义:

```
DAO_DLL DaoAPI __dao;
```

要定义这个变量, 也可以还是用宏`DAO_INIT_MODULE`。

如果你不打算将程序链接到道语言库, 下面所要做的就是手动打开 道语言的库文件, 这可以使用`dao.h` 里定义的一个宏: `in dao.h`,

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

接下来必须调用`DaoInit()` 函数来初始化道语言库。这个函数需要一个`DaoAPI`结构指针为参数。如要使用直接的接口, 这个函数可被直接调用, 且参数可以

为NULL:

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

否则要以如下方式调用:

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

这里 `__dao` 的地址被作为参数传递给了 `DaoInit()` 函数。这样 `DaoInit()` 不仅执行道语言库的初始化工作, 它还将道语言的C编程接口的函数指针设定到 `__dao` 的成员里, 之后C编程接口将通过一个间接层 `__dao` 使用, 这样嵌入了道的程序在编译链接时将不需要链接到道语言库。

到了这步, 你可以用 `DaoVmSpace_RunMain()` 来执行一个脚本文件,

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

这个脚本文件将在当前路径和 `DaoVmSpace` 的搜寻路径表里查找。

或者你也可用 `DaoVmSpace_Load()` 来载入一个脚本文件并获得一个命名空间。如果该文件曾被载入过, 该文件所含代码将被重新执行, 并返回前次载入时创建的命名空间。如果不曾被载入过, 那么一个新的命名空间将被创建并返回。此脚本文件也将在当前路径和 `DaoVmSpace` 的搜寻路径表里查找。

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

你也可以用下面的函数来获得一个命名空间结构,

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

或,

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

以获得 `DaoVmSpace` 的主命名空间。

或,

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

从另一个命名空间`nameSpace`获得一给定名字`name`的命名空间，如果`nameSpace`没有名为`name`的命名空间，那么创建一个新的命名空间，加入到`nameSpace`，并赋予名字`name`，然后返回此命名空间。

如前面介绍的，封装好的C函数和类型，和常量可被加入到命名空间结构里。命名空间结构还可用来载入脚本文件，该文件里定义的函数和类以及全局常量变量都将被放在此命名空间结构里。此脚本文件将不在`DaoVmSpace`的搜寻路径表里查找。

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

为了编译和执行脚本代码，或执行道语言函数，你还需要创建类型为`DaoVmProcess`的结构。它可从 `DaoVmSpace` 结构获得或创建。

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

使用 `DaoVmSpace` 结构，可按如下方式编译或运行道语言脚本代码，或调用道语言函数，

```
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"
```

调用道语言函数:

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

给函数调用传递参数:

```
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 结构也可用来调用道语言类的成员方法。如果是一个需要类实例的成员方法，类实例可作为DaoVmProcess.Call()的第三个参数传入。

6 道语言的C编程接口的高级应用

前面已提过，在封装C/C++数据类型时可定义或保留类型的继承关系，不仅如此，在道脚本里使用这些类型时，还可从C/C++数据类型派生出道类(class)，就象从道类派生一样。甚至，C++类可按一定方式封装，使得它们的虚函数可在道派生类中被重新定义。对于C结构，如它含有函数指针作为成员，某些情况下，这样的C结构可象C++类那样封装，使得其函数指针成员的行为象C++类的虚函数一样，也就是，可以在C结构的道派生类中定义成员方法，并当C结构的成员函数指针作为函数被调用时，道派生类的成员方法将被调用。这些技术都有点复杂，这里将不作介绍。感兴趣的可以去参考一些道模块的源代码（如DaoVTK模块等），或使用半自动工具”tools/autobind.dao”来生成封装代码然后查看产生的代码。