

Dao Library Reference

Limin Fu (phoolimin@gmail.com)

September 19, 2009

Contents

1	introduction	3
2	string	4
3	complex	8
4	list	9
5	map	13
6	numeric array	15
7	cdata	18
8	std (was stdlib)	21
9	stream and io (was stdio) library	25
10	math	28
11	reflection	32
12	coroutine	36
13	multi-threading library	38
14	thread object	40
15	mutex	41
16	condition variable	42

<i>CONTENTS</i>	2
17 semaphore	43
18 network	45
19 message passing interface	48
(For Dao 1.1)	
This document is licensed under <i>GNU Free Documentation License</i> .	

Chapter 1

introduction

This documentation introduces the standard libraries for a number of applications and the standard methods for various data types. In the method parameter lists, *par* : *XYZ* means parameter *par* should be of the type of *XYZ* ; *par* = *XYZ* means that *par* has default value *XYZ* . There are also a number of overloaded methods, which have the same name, but with different parameter lists. Some methods of one library can be imported to the current namespace by using "use library;" statement so that the methods can be used directly, e.g.,

```
use math;  
a = cos( 1.5 );
```

In the following, the prototypes of methods usually appear as for clarity:

```
type.method( parameter_list )
```

but the real prototypes are defined the following form:

```
method( self : type, parameter_list )
```

Chapter 2

string

Method list:

```
chop( )
erase( start=0, n=-1 )
find( str : string, from=0, reverse=0 ) => int
insert( str : string, index=0 )
replace( str1 : string, str2 : string, index=0 )
replace( str1 : string, table : map<string,string>, max=0 )
expand( keys :map<string,string>, spec='$', keep=1 )
expand( keys : tuple, spec='$', keep=1 )
resize( size : int )
size( ) => int
split( sep : string, quote="", rm=1 ) => list<string>
tokenize( seps : string, quotes="", backslash=0, simplify=0 ) => list<string>
tonumber( base=10 ) => double
tolower( ) => string
toupper( ) => string
encrypt( key : string, hex=0 ) => string
decrypt( key : string, hex=0 ) => string
```

There are a few other methods that use regular expression pattern for string operation, please refer to the *Dao Regex Tutorial* .

Method details:

```
string.chop( utf=0 )
```

Chop off new line symbol from the end of a string. If utf=1, chop off the characters

that do not form a valid UTF-8 encoding.

```
string.erase( start=0, n=-1 )
```

Erase n characters starting from index $start$. If $n=-1$, erase all the rest.

```
string.find( str : string, from=0, reverse=0 ) => int
```

Find substring str , starting from $from$; search backward if $reverse$ is true. Return the index of the occurrence of str , return -1 if not found.

```
string.insert( str : string, index=0 )
```

Insert substring str at $index$

```
string.replace( str1 : string, str2 : string, index=0 )
```

Replace the $index$ -th occurrence of substring $str1$ to substring $str2$; the index starts from 1, which means the first, and 0 means all.

```
string.replace( table : map<string,string>, max=0 )
```

Replace the occurrence of the keys of $table$ by the corresponding values. If max is zero, replace the shorter key first, otherwise, replace the longer first.

```
string.expand( keys : map<string,string>, spec='$', keep=1 ) => string
```

If the string contains place holders in form of $$(name)$, where $$$ is the special character passed in by $spec$, this method will expand or fill at each place holder by the value string from $keys$ with key equal to the name of the place holder.

If $keep$ is zero, place holders with names not found in $keys$ will be replaced by empty string, namely, removing the place holders; otherwise the are kept.

```
tpl = 'The quick brown $(A) jumps over the lazy $(B)';
str = tpl.expand( { 'A' => 'fox', 'B' => 'dog' } );
stdio.println( str );
```

Output

The quick brown fox jumps over the lazy dog

```
string.expand( keys : tuple, spec='$', keep=1 ) => string
```

If the string contains place holders in form of $\$(name)$, where $\$$ is the special character passed in by *spec*, this method will expand or fill at each place holder by the item value string from *keys* with item field name equal to the name of the place holder.

If *keep* is zero, place holders with names not found in *keys* will be replaced by empty string, namely, removing the place holders; otherwise the are kept.

```
tpl = 'The quick brown $(A) jumps over the lazy $(B)';
str = tpl.expand( ( A => 'fox', B => 'dog' ) );
stdio.println( str );
```

Output

```
The quick brown fox jumps over the lazy dog
```

```
string.resize( size : int )
```

Resize the string to have length *size*, and the extended part are filled with blank space.

```
string.size( ) => int
```

Return the length of a string.

```
string.split( sep : string, quote="", rm=1 ) => list<string>
```

Split the string by separator *sep*, and return the tokens as a list. Quotation symbols may also be specified to avoid breaking the string inside a pair of quotes. If the quotations symbols appear in the begin and end of a token and *rm* is true, they are removed. The separators are not included in the result list.

```
string.tokenize( seps : string, quotes="", backslash=0 ) => list<string>
```

Tokenize the string by a set of separators. Each character in *seps* is treated as a separator. Quotation symbols may also be specified to avoid breaking the string inside a pair of quotes. If *backslash* is true, the separators and quotes preceded by a backslash are considered as normal characters. The separators, quotes and backslashes are not removed from

the result tokens. If *simplify* is true, the blank spaces are removed from the beginning and end of each token, and then empty tokens are removed from the resulting list.

```
string.tonumber( base=10 ) => double
```

Convert the string to a number with base equal to *base* .

```
string.toLowerCase( ) => string
```

Convert the string to lower case. Return *self* string.

```
string.toUpperCase( ) => string
```

Convert the string to upper case. Return *self* string.

```
string.encrypt( key : string, hex=0 ) => string
```

Encrypt the string with *key* using XXTEA algorithm. If *hex* is true, store the encrypted string as hex digits. Return *self* string.

```
string.decrypt( key : string, hex=0 ) => string
```

Decrypt the string with *key* using XXTEA algorithm. If *hex* is true, the *self* string is interpreted as hex digits. Return *self* string.

Chapter 3

complex

Method list:

```
imag( v=0.00 ) => double  
real( v=0.00 ) => double
```

Method details:

```
complex.imag( v=0.00 ) => double
```

If it is called with parameters, set the imaginary part of the complex. No matter if it is called with or without parameters, return the imaginary part before modification.

```
complex.real( v=0.00 ) => double
```

If it is called with parameters, set the real part of the complex. No matter if it is called with or without parameters, return the real part before modification.

Chapter 4

list

Method list:

```
append( item : @ITEM )
clear( )
dequeue( )
enqueue( item : @ITEM )
erase( start=0, n=1 )
front( ) => @ITEM
insert( item : @ITEM, pos=0 )
max( ) => tuple<@ITEM,int>
min( ) => tuple<@ITEM,int>
pop( )
popback( )
popfront( )
push( item : @ITEM )
pushback( item : @ITEM )
pushfront( item : @ITEM )
resize( size : int )
size( ) => int
ranka( k=0 ) => list<int>
rankd( k=0 ) => list<int>
sorta( k=0 ) => list<@ITEM>
sortd( k=0 ) => list<@ITEM>
sum( ) => @ITEM
top( ) => @ITEM
```

The **sort()** method that can take an expression as parameter is now supported as a built-in method, please read *here* .

Method details:

```
list<@ITEM>.append( item : @ITEM )
```

Append *item* to the list.

Here **@ITEM** indicates the *item* parameter will have the same type as the list, so `append(item : @ITEM)` will only allow appending an item of the same type as the type of items of the list.

```
list<@ITEM>.clear()
```

Remove all items from the list.

```
list<@ITEM>.dequeue()
```

Use the list as a queue, get the first item in the queue.

```
list<@ITEM>.enqueue( item : @ITEM )
```

Use the list as a queue, push *item* in the end of the queue.

```
list<@ITEM>.erase( start=0, n=1 )
```

Erase *n* items from the list starting from index *start* .

```
list<@ITEM>.front() => @ITEM
```

Get the first item in the front.

```
list<@ITEM>.insert( item : @ITEM, pos=0 )
```

Insert *item* at position *pos* .

```
list<@ITEM>.max( ) => <@ITEM,int>
```

Return the max item and its index in the list.

```
list<@ITEM>.min( ) => <@ITEM,int>
```

Return the min item and its index in the list.

```
list<@ITEM>.pop( )
```

Pop out one item from the end of the list.

```
list<@ITEM>.popback( )
```

Pop out one item from the end of the list.

```
list<@ITEM>.popfront( )
```

Pop out one item from the begin of the list.

```
list<@ITEM>.push( item : @ITEM )
```

Push *item* to the end of the list.

```
list<@ITEM>.pushback( item : @ITEM )
```

Push *item* to the end of the list.

```
list<@ITEM>.pushfront( item : @ITEM )
```

Push *item* to the begin of the list.

```
list<@ITEM>.ranka( k=0 ) => list<int>
list<@ITEM>.rankd( k=0 ) => list<int>
```

Ascending or descending ranking of the list. If *k* is not zero, rank the smallest or largest *k* items.

```
list<@ITEM>.resize( size : int )
```

Resize the list to have size *size* . Extended part is filled with nil.

```
list<@ITEM>.size( ) => int
```

Return the size of the list.

```
list<@ITEM>.sorta( k=0 ) => list<@ITEM>  
list<@ITEM>.sortd( k=0 ) => list<@ITEM>
```

Ascending or descending sorting of the list. If k is not zero, sort the smallest or largest k items.

```
list<@ITEM>.sum( ) => @ITEM
```

Return the sum of the list. Only for lists contain numbers, strings or complex numbers.

```
list<@ITEM>.top( ) => @ITEM
```

Get the last item in the list.

Chapter 5

map

Method list:

```
clear( )  
erase( from : @KEY=nil, to : @KEY=nil )  
find( key : @KEY, type=0 ) => tuple<int,@KEY,@VALUE>  
insert( key : @KEY, value : @VALUE )  
keys() => list<@KEY>  
keys( from : @KEY ) => list<@KEY>  
keys( from : @KEY, to : @KEY ) => list<@KEY>  
size( ) => int  
values() => list<@VALUE>  
values( from : @KEY ) => list<@VALUE>  
values( from : @KEY, to : @KEY ) => list<@VALUE>
```

Method details:

```
map<@KEY,@VALUE>.clear( )
```

Erase all the element pairs from the map.

```
map<@KEY,@VALUE>.erase( from : @KEY=nil, to : @KEY=nil )
```

Erase key-value pairs between key *from* and key *to* (inclusive). If the second parameter is omitted, erase the element with key equal to *from* . If both parameters are omitted, it will erase all elements.

```
map<@KEY,@VALUE>.find( key : @KEY, type=0 ) => tuple<int,@KEY,@VALUE>
```

If $type=0$, find the key equal to key ; If $type<0$, find the maximum key lesser than or equal to key ; If $type>0$, find the minimum key greater than or equal to key ; return a tuple of an integer, the key and value of the found element, the integer is 1 if found; 0, otherwise.

```
map<@KEY,@VALUE>.insert( key : @KEY, value : @VALUE )
```

Insert a pair of element.

```
keys() => list<@KEY>
keys( from : @KEY ) => list<@KEY>
keys( from : @KEY, to : @KEY ) => list<@KEY>
```

Return all keys, or keys starting from $from$, or keys between $from$ and to .

```
map<@KEY,@VALUE>.size( ) => int
```

Return the number of elements in the map.

```
values() => list<@VALUE>
values( from : @KEY ) => list<@VALUE>
values( from : @KEY, to : @KEY ) => list<@VALUE>
```

Return all values, or values with keys starting from $from$, or with keys between $from$ and to .

Chapter 6

numeric array

Method list:

```
dim( ) => array<int>
dim( i : int ) => int
index( i : int ) => array<int>
size( ) => int
resize( dims : array<int> )
reshape( dims : array<int> )
sorta( k=0 )
sortd( k=0 )
ranka( k=0 ) => array<int>
rankd( k=0 ) => array<int>
max( slice={} ) => tuple<@ITEM,int>
min( slice={} ) => tuple<@ITEM,int>
sum( slice={} ) => @ITEM
varn( slice={} ) => double
fft( inv=-1 )
```

The **apply()** method that can take an expression as parameter is now supported as a built-in method, please read *here* . The **noapply()** method is removed.

Method details:

```
dim( ) => array<int>
dim( self : array<@ITEM> ) => array<int>
```

Return the sizes of each dimension as a vector.

```
array<@ITEM>.dim( i : int ) => int
```

Return the size of the i -th dimension of the array.

```
array<@ITEM>.index( i : int ) => array<int>
```

Convert a flat index of the array to the corresponding multi-dimensional index.

```
array<@ITEM>.size( ) => int
```

return the number of elements in the array.

```
array<@ITEM>.resize( dims : array<int> )
```

Resize the array to dimension as specified by *dims* .

```
array<@ITEM>.reshape( dims : array<int> )
```

Reshape the array to dimension as specified by *dims* .

```
array<@ITEM>.ranka( k=0 ) => array<int>
array<@ITEM>.rankd( k=0 ) => array<int>
```

Ascending or descending ranking of the array. If k is not zero, rank the smallest or largest k items.

```
array<@ITEM>.sorta( k=0 )
array<@ITEM>.sortd( k=0 )
```

Ascending or descending sorting of the array. If k is not zero, sort the smallest or largest k items.

```
array<@ITEM>.max( slice={} ) => tuple<@ITEM,int>
```

Return the max value and its flat index of the elements specified by *slice* as sub-array. If *slice* is empty or omitted, return the max value of the array.

```
mat = [ 1.5, 2.5, 3, 4; 5, 6, 7, 8; 9, 0, 1, 2 ];
stdio.println( mat.max( { 1:, 1:2 } ) );
```

Output

```
( 7.000000, 6 )
```

This example prints the maximum value and its index of the elements in a sub-matrix formed by rows starting from the second row, and the second and third columns.

```
array<@ITEM>.min( slice={} ) => tuple<@ITEM,int>
```

Return the min value and its flat index of the elements specified by *slice* . If *slice* is empty or omitted, return the min value of the array.

```
array<@ITEM>.sum( slice={} ) => @ITEM
```

Return the sum value of the elements specified by *slice* . If *slice* is empty or omitted, return the sum value of the array.

```
array<@ITEM>.varn( slice={} ) => double
```

Return the sample variance of the elements specified by *slice* . If *slice* is empty or omitted, return the sample variance of the array.

```
array<complex>.fft( inv=-1 )
```

Fast Fourier Transform.

Chapter 7

cdata

The **cdata** (previously named as **buffer**) data type represents a block of memory.

Method list:

```
cdata( size : int ) => cdata
copydata( buf : buffer )
getbyte( index : int, signed=1 ) => int
getdouble( index : int ) => double
getfloat( index : int ) => float
getint( index : int, signed=1 ) => int
getshort( index : int, signed=1 ) => int
getString( mbs=1 ) => string
resize( size : int )
setbyte( index : int, value : int, signed=1 )
setdouble( index : int, value : double )
setfloat( index : int, value : float )
setint( index : int, value : int, signed=1 )
setshort( index : int, value : int, signed=1 )
setstring( string : string )
size( ) => int
```

Method details:

```
buffer( size = 0 ) => buffer
```

Create a buffer of *size* bytes.

```
buffer.copydata( buf :buffer )
```

Copy data from buffer *buf* .

```
buffer.getbyte( index : int, signed=1 ) => int
```

Get one byte with specified *index* ; if *signed=1* , the byte is interpreted as signed.

```
buffer.getdouble( index : int ) => double
```

Get double floating number with specified *index* .

```
buffer.getfloat( index : int ) => float
```

Get single floating number with specified *index* .

```
buffer.getint( index : int, signed=1 ) => int
```

Get integer with specified *index* . if *signed=1* , the integer is interpreted as signed.

```
buffer.getshort( index : int, signed=1 ) => int
```

Get short integer with specified *index* . if *signed=1* , the short integer is interpreted as signed.

```
buffer.getstring( mbs=1 ) => string
```

Interprete the buffer as a string; Return MBS string if *mbs=1* , otherwise return WCS.

```
buffer.resize( size : int )
```

Resize the buffer to *size* bytes;

```
buffer.setbyte( index : int, value : int, signed=1 )
```

Set one byte specified *index* with *value* ; if *signed=1* , the byte is interpreted as signed.

```
buffer.setdouble( index : int, value : double, )
```

Set one double specified *index* with *value* ;

```
buffer.setfloat( index : int, value : float )
```

Set one float specified *index* with *value* ;

```
buffer.setint( index : int, value : int, signed=1 )
```

Set one integer specified *index* with *value* ; if *signed=1* , the integer is interpreted as signed.

```
buffer.setshort( index : int, value : int, signed=1 )
```

Set one short integer specified *index* with *value* ; if *signed=1* , the short integer is interpreted as signed.

```
buffer.setstring( string : string )
```

```
buffer.size( ) => int
```

Return the size of the buffer in bytes.

Chapter 8

std (was stdlib)

Method list:

```
stdlib.about( ... ) => string
stdlib.callable( object ) => int
stdlib.compile( source : string )
stdlib.copy( object : @OBJECT ) => @OBJECT
stdlib.ctimef( time=0, format='%Y-%M-%D, %H:%I:%S', namemap : map<string,list<string>>
={=>} ) => string
stdlib.ctime( time=0 ) => tuple<year:int,month:int,day:int,wday:int,hour:int,minute:int,second:int>
stdlib.debug( ... )
stdlib.error( info : string )
stdlib.eval( source : string, stream=stdio )
stdlib.exit( code=0 )
stdlib.gcmax( limit=0 ) => int
stdlib.gcmin( limit=0 ) => int
stdlib.listmeth( object )
stdlib.enable_fe( flags : int ) => int
stdlib.disable_fe( flags : int ) => int
stdlib.load( file : string )
stdlib.pack( number : int ) => string
stdlib.pack( list :list<int> ) => string
stdlib.sleep( seconds : float )
stdlib.system( command : string )
stdlib.time( ) => int
stdlib.time( tm : tuple<year:int,month:int,day:int,wday:int,hour:int,minute:int,second:int>
) => int
stdlib.tokenize( source : string ) => list<string>
stdlib.unpack( string : string ) => list<int>
stdlib.warn( info : string )
```

Method details:

```
about( ... ) => string
```

Return the type name and address of the parameters.

```
callable( object ) => int
```

Tell if the object is callable.

```
compile( source : string )
```

Compile the source code and return a routine object.

```
copy( object : @OBJECT ) => @OBJECT
```

Return a deep copy of the object. (TODO: the implementation might be incomplete.)

```
ctime( time=0 ) => tuple<year:int,month:int,day:int,wday:int,hour:int,minute:int,second:int>
```

Convert *time* returned from **stdlib.time()** into calendar time as a tuple.

```
ctimef( time=0, format='%Y-%M-%D, %H:%I:%S', namemap : map<string,list<string>> ={} ) => string
```

Convert *time* returned from **stdlib.time()** into calendar time with specified *format* ; the names for year, month and date etc. can be specified in *namemap*

```
debug( ... )
```

If called with parameters, return the type name and address of the parameters. If called without parameters, the debugging console will be prompted up. It only has effects in debugging running mode.

```
error( info : string )
```

Print *info* as error information and abort current virtual machine process.

```
eval( source : string, stream=stdio )
```

Evaluate the source, and use *stream* as the standard IO device.

```
exit( code=0 )
```

Exit with *code* .

```
gcmax( limit=0 ) => int
```

Return the current maximum GC threshold. If there is the parameter, the threshold is changed to *limit* . The maximum GC threshold is the upper bound of number of candidate garbage objects, if this limit is exceeded, all the mutator threads are blocked until they are waked up by the garbage collector after finishing to process a certain part of the garbages. Normally, the garbage collection thread runs concurrently with the mutator threads, or is blocked if there are too few garbage.

```
gcmin( limit=0 ) => int
```

Return the current minimum GC threshold. If there is the parameter, the threshold is changed to *limit* . The minimum GC threshold is the lower bound of number of candidate garbage objects for the garbage collector to wake up and start garbage collecting.

```
listmeth( object )
```

List the available methods for the object.

```
load( file : string )
```

Load and execute scripts from a file.

```
pack( number : int ) => string
```

Convert the number into a character.

```
pack( list :list<int> ) => string
```

Convert the list of numbers into a string.

```
sleep( seconds : float )
```

Let the current thread to sleep for *seconds* (may lesser than 1) seconds.

```
system( command : string )
```

Run a system command.

```
time( ) => int
```

Return current time.

```
time( tm : tuple<year:int,month:int,day:int,wday:int,hour:int,minute:int,second:int> ) =>  
int
```

Convert calendar time to time in seconds (as returned by time()).

```
tokenize( source : string ) => list<string>
```

Tokenize Dao source code.

```
unpack( string : string ) => list<int>
```

Unpack string into a list of number.

```
warn( info : string )
```

Print a warning, with file name and line number information.

Chapter 9

stream and io (was stdio) library

Method list:

```
close( )
eof( )
flush( )
getstring( )
isopen( )
name( )
open( )
open( file : string, mode : string )
popen( cmd : string, mode : string )
write( ... )
writeln( ... )
writef( format : string, ... )
print( ... )
println( ... )
printf( format : string, ... )
read( count=0 )=>string
read( file : string )=>string
seek( pos : int, from : int )
sstream( mbs=1 )
tell( )
```

Method details:

```
close( )
```

Close a file stream.

```
eof( )
```

Check if the end of the stream is reached.

```
flush( )
```

Flush the stream to cause all buffered output to be written to the device.

```
getstring( )
```

Get output string from a string stream.

```
isopen( )
```

Check if a file stream is open.

```
name( )
```

Get the file name of a file stream

```
open( )
```

Open a temporary file stream.

```
open( file : string, mode : string )
```

Open a file stream on *file* with mode *mode* . The supported mode is exactly the same as what is supported in standard C function *fopen()* .

```
popen( cmd : string, mode : string )
```

Open a pipe in mode *mode* to execute command *cmd* .

```
write( ... )  
print( ... )
```

Print to the stream.

```
writeln( ... )  
println( ... )
```

Print to the stream with a line break symbol.

```
writef( format : string, ... )
printf( format : string, ... )
```

Formatted printing, *format* can be specified in the same way as in C.

```
read( count=0 )=>string
read( file : string )=>string
```

If *count* is positive, read *count* number of bytes from the stream; if *count* is negative, read all bytes from the stream; otherwise, read one line from the stream including the line break symbol.

```
read( file : string )=>string
```

Read the whole file from *file* .

```
seek( pos : int, from : int )
```

Seek stream position with offset *pos* from *from* ; *from* can be "stdio.SEEK_CUR", "stdio.SEEK_SET" or "stdio.SEEK_END";

```
sstream( mbs=1 )
```

Create a string stream; the internal string buffer can be MBS or WCS depends on parameter *mbs* .

```
tell( )
```

Tell the current stream position.

Chapter 10

math

These math functions except **srand()** and **pow()** , they are also supported as builtin functions since Dao 1.1.

Method list:

```
math.abs( p : double )
```

```
math.abs( p : complex )
```

```
math.acos( p : double )
```

```
math.arg( p : complex )
```

```
math.asin( p : double )
```

```
math.atan( p : double )
```

```
math.ceil( p : double )
```

```
math.ceil( p : complex )
```

```
math.cos( p : double )
```

```
math.cos( p : complex )
```

```
math.cosh( p : double )
```

```
math.cosh( p : complex )
```

```
math.exp( p : double )
```

```
math.exp( p : complex )
```

```
math.floor( p : double )
```

```
math.floor( p : complex )
```

```
math.imag( p : complex )
```

```
math.log( p : double )
```

```
math.log( p : complex )
```

```
math.norm( p : complex )
```

```
math.pow( p1 : double, p2 : double )
```

```
math.pow( p1 : double, p2 : complex )
```

```
math.pow( p1 : complex, p2 : double )
```

```
math.pow( p1 : complex, p2 : complex )
```

```
math.real( p : complex )
```

```
math.sin( p : double )
```

```
math.sin( p : complex )
```

```
math.sinh( p : double )
```

```
math.sinh( p : complex )
```

```
math.sqrt( p : double )
```

```
math.sqrt( p : complex )
```

```
math.srand( p : double )
```

```
math.rand( p : double )
```

```
math.tan( p : double )
```

```
math.tan( p : complex )
```

```
math.tanh( p : double )
```

```
math.tanh( p : complex )
```

Chapter 11

reflection

Method list:

```
namespace( object=nil ) => any
name( object ) => string
type( object ) => any
base( object ) => list<any>
doc( object, newdoc='' ) => string
constant( object, restrict=0 )=>map<string,tuple<value:any,type:any> >
variable( object, restrict=0 )=>map<string,tuple<value:any,type:any> >
constant( object, name:string, value=nil )=>tuple<value:any,type:any>
variable( object, name:string, value=nil )=>tuple<value:any,type:any>
class( object ) => any
routine() => any
routine( rout : any ) => list<any>
param( rout )=>list<tuple<name:string,type:any,deft:int,value:any> >
isa( object, name : string ) => int
isa( object, type : any ) => int
self( object ) => any
argc() => int
argv() => list<any>
argv( i : int ) => any
```

Method details:

```
namespace( object=nil ) => any
```

If object is a routine or class, return the namespace where it is defined; otherwise,

return the current namespace.

```
name( object ) => string
```

Get the name of a routine, function, method or class.

```
type( object ) => any
```

return the type of the object.

```
base( object ) => list<any>
```

If *object* is a class, return its direct parent classes; If *object* is a class instance, return its direct parent instances;

```
doc( object, newdoc='' ) => string
```

return the documentation string of a routine or class; if "newdoc" is passed, set the documentation string to "newdoc";

```
constant( object, restrict=0 )=>map<string,tuple<value:any,type:any> >
```

Return the constant fields of object, if "restrict" is not zero and there are private or protected fields in "object", only return the public ones. The return value is a map, mapping from field name and field value and type.

```
variable( object, restrict=0 )=>map<string,tuple<value:any,type:any> >
```

Return the variable fields of object, if "restrict" is not zero and there are private or protected fields in "object", only return the public ones. The return value is a map, mapping from field name and field value and type.

```
constant( object, name:string, value=nil )=>tuple<value:any,type:any>
```

Return a tuple of value and type for constant field "name" in object. If "value" is passed in, set the field value to "value". All field can be accessed and modified by this function.

```
variable( object, name:string, value=nil )=>tuple<value:any,type:any>
```

Return a tuple of value and type for variable field "name" in object. If "value" is passed in, set the field value to "value". All field can be accessed and modified by this function.

```
class( object ) => any
```

If *object* is a routine, return its host class which defines this routine as a method; If *object* is a class instance, return its class prototype.

```
routine() => any
```

return the current routine that is running.

```
routine( rout : any ) => list<any>
```

return the list of overloaded routines with the same name as routine "rout".

```
param( rout )=>list<tuple<name:string,type:any,deft:int,value:any> >
```

return the parameter list of "rout". The returned list contains tuples composed of parameter name, type, and default value. The "value" field is set to nil, if the parameter has no default value. The "deft" field of the tuples indicates whether the "value" is a default value or not, in case that a nil value is set as default.

```
isa( object, name : string ) => int
```

Check if object is of type indicated by *name* . Normally *name* should be a valid type name such as **int** , **float** , **list<int>** or a class name etc. It can also be one of the name listed in the following table, to check if *object* is in a category of certain types. For example, when "class" is passed as *name* , *isa()* will return true, if *object* is a class.

name	type
class	a class
object	a class instance
routine	a Dao routine
function	a C function
namespace	a namespace
tuple	a tuple
list	a list
map	a map
array	an array

```
isa( object, type : any ) => int
```

Check if *object* is of type *type* , which is a type object that can be returned by *reflect.type()* .

```
argc() => int
```

Return the argument count of the current function call.

```
argv() => list<any>
```

Return the argument values as a list of the current function call.

```
argv( i : int ) => any
```

Return the i-th argument value of the current function call.

```
self( object )
```

Return the mapping-down class instance of the class instance *object*

```
trace( )
```

Print the trace of calling stack.

Chapter 12

coroutine

Method list:

```
coroutine.create( object, ... )  
coroutine.resume( object, ... )  
coroutine.status( object )  
coroutine.yield( ... )
```

Method details:

```
create( object, ... )
```

Create a coroutine object which is actually a virtual machine process with its own calling stack of contexts. *object* must be a routine or context (function closure); The addition parameters will be passed to the created coroutine object in the same way as normal function call. The coroutine object is created in suspending status, and should be executed by calling **coroutine.resume()**.

Create coroutine with wrong parameters will raise an exception.

```
resume( object, ... )
```

Resume the coroutine object. The additional parameters to **resume()** will become results from **yield()**.

Resuming a coroutine that has reach its end will raise an exception.

```
status( object )
```

Return the status of the coroutine object as a string, which can be one of

suspended	running	aborted	finished	not_a_coroutine
-----------	---------	---------	----------	-----------------

```
yield( ... )
```

Yield current coroutine execution. Any parameters in **yield** will become the result from **resume()**.

Chapter 13

multi-threading library

The multi-threading library in Dao is provided by the library object "mtlib".

Method list:

```
mtlib.thread( object, ... ) => thread
mtlib.mutex( ) => mutex
mtlib.condition( ) => condition
mtlib.semaphore( ) => semaphore
mtlib.exit( )
mtlib.mydata( ) => map<string,any>
mtlib.self( ) => thread
mtlib.testcancel( )
```

Method details:

```
condition( ) => condition
```

Create a condition variable.

```
thread( object, ... )
```

Create a thread object. *object* must be a Dao routine or context, which will receive parameters from the additional part of the parameters for **thread.create()** .

```
exit( )
```

Exit current thread.

```
mutex( ) => mutex
```

Create a mutex.

```
mydata( ) => map<string,any>
```

Get thread specific data as a map.

```
self( ) => thread
```

```
semaphore( ) => semaphore
```

Create a semaphore object.

```
testcancel( )
```

Create a cancel point. If **cancel()** is called before this point, calling this function will terminate the current thread.

Chapter 14

thread object

Method list:

```
cancel( )  
detach( )  
join( )  
mydata( ) => map<string,any>
```

Method details:

```
cancel( )
```

Cancel the execution of the thread object.

```
detach( )
```

Detach the execution of the thread object.

```
join( )
```

Join the execution of the thread object with the current thread.

```
mydata( ) => map<string,any>
```

Get thread specific data as a map.

Chapter 15

mutex

Method list:

```
lock( )  
trylock( )  
unlock( )
```

Method details:

```
lock( )
```

Lock the mutex.

```
trylock( )
```

Try to lock the mutex. Return true if successful, otherwise, false.

```
unlock( )
```

Unlock the mutex.

Chapter 16

condition variable

Method list:

```
broadcast( )  
signal( )  
timedwait( mtx : mutex, seconds : float )  
wait( mtx : mutex )
```

```
broadcast( )
```

Wake all threads blocked on the condition variable.

```
signal( )
```

Wake one thread(randomly) blocked on the condition variable.

```
timedwait( mtx : mutex, seconds : float )
```

Block the current thread on the condition variable with timeout, which can be a decimal number.

```
wait( mtx : mutex )
```

Block the current thread on the condition variable.

Chapter 17

semaphore

Method list:

```
getvalue( )  
post( )  
setvalue( n : int )  
wait( )
```

Method details:

```
getvalue( )
```

Get the available resource count of the semaphore.

```
post( )
```

Release one resource of the semaphore.

```
setvalue( n : int )
```

Set the total number of resource of the semaphore.

```
wait( )
```

Require one resource of the semaphore. The current thread will be blocked if the resource is not available.

Chapter 18

network

Method list:

```
network.FD_CLR( fd : int, set :fd_set )
network.FD_ISSET( fd : int, set :fd_set )
network.FD_SET( fd : int, set :fd_set )
network.FD_ZERO( set :fd_set )
network.accept( socket : int )
network.bind( port : int )
network.close( socket : int )
network.connect( host : string, port : int )
network.gethost( host : string )
network.getpeername( socket : int )
network.listen( socket : int, backlog =10 )
network.receive( socket : int, maxlen =1000 )
network.receive_dao( socket : int )
network.select( nfd : int, setr :fd_set, setr :fd_set, setr :fd_set, tv : float )
network.send( socket : int, data : string )
network.send_dao( socket : int, ... )
```

Network File Description Set Handling

File description set can be created by: *fd_set()*

```
FD_CLR( fd : int, set :fd_set )
```

Remove *fd* from *set* .

```
FD_ISSET( fd : int, set :fd_set )
```

Check if *fd* is in *set* .

```
FD_SET( fd : int, set :fd_set )
```

Add *fd* to *set* .

```
FD_ZERO( set :fd_set )
```

Clear the set.

Method details:

```
accept( socket : int )
```

Accept a connection from the socket.

```
bind( port : int )
```

Bind to the port.

```
close( socket : int )
```

```
connect( host : string, port : int )
```

Connect to *host* through *port* .

```
gethost( host : string )
```

Get host information (names and addresses).

```
getpeername( socket : int )
```

Get peer information.

```
listen( socket : int, backlog =10 )
```

Listen to the socket.

```
receive( socket : int, maxlen =1000 )
```

Receive raw data from the socket.

```
receive_dao( socket : int )
```

Receive data from the socket, and create and return proper Dao data. It should be used in pair with *send_dao()* .

```
select( nfd : int, setr :fd_set, setw :fd_set, sete :fd_set, tv : float )
```

Quoted from glibc documentation:

The select function blocks the calling process until there is activity on any of the specified sets of file descriptors, or until the timeout period has expired.

The file descriptors specified by the read-fds argument are checked to see if they are ready for reading; the write-fds file descriptors are checked to see if they are ready for writing; and the except-fds file descriptors are checked for exceptional conditions. You can pass a null pointer for any of these arguments if you are not interested in checking for that kind of condition.

A file descriptor is considered ready for reading if it is not at end of file. A server socket is considered ready for reading if there is a pending connection which can be accepted with accept; see section Accepting Connections. A client socket is ready for writing when its connection is fully established; see section Making a Connection.

```
send( socket : int, data : string )
```

Send raw data to the socket.

```
send_dao( socket : int, ... )
```

Send Dao data to the socket, with data type information attached to the network packets.

Chapter 19

message passing interface

Method list:

```
mpi.receive( timeout=-1.0 )  
mpi.receive( pid : string, timeout=-1.0 )  
mpi.send( object, ... )  
mpi.spawn( pid : string, proc : string, timeout=-1.0, ... )
```

Method details:

```
receive( timeout=-1.0 )
```

Receive message with timeout. No positive timeout means infinite waiting. Return a list of received data.

```
receive( pid : string, timeout=-1.0 )
```

Receive message from process *pid* with timeout.

```
send( object, ... )
```

If *object* is a process, send the rest of parameters as message to it; If *object* is a callable object, this object is scheduled to be called asynchronously with the rest parameters.

```
spawn( pid : string, proc : string, ... )
```

Spawn a virtual machine(VM) or operation system(OS) process. If a OS process or a VM process within another OS process is not spawned successfully within the timeout, an exception will be raised. Return a process handle if it created a local (within the same OS process) VM process. *pid* must be one of the form in the following table:

pid	proc	the rest parameters	meaning
""	routine name	parameters for the routine	VM process without name, accessible within the same OS process by its handle
"vm_proc"	as above	as above	named VM process, globally accessible by different OS processes on different computers
"@os_proc"	Dao script file name	timeout (default -1.0)	named OS process, globally accessible
"vm_proc@os_proc"	as "vm_proc"	as "vm_proc"	named VM process within a OS process named "os_proc", globally accessible
"@os_proc@@host"	Dao script file name	timeout (default -1.0)	named OS process on network "host", globally accessible
"vm_proc@os_proc@@host"	as "vm_proc"	as "vm_proc"	named VM process within a OS process named "os_proc@@host", globally accessible