

道语言的并行计算

傅利民(phoolimin@gmail.com)

September 19, 2009

1 使用协程

道语言支持协程`coroutine`实现协作式多线程。道语言里一个协程对象实际上就是一个虚拟机进程，可被暂停或继续其运行状态。道语言有两种方式支持协程：第一种方式跟Lua里的协程很相似，使用`coroutine`库里的`create()`,`yield()`,`resume()`等方法来创建和使用协程；另一种方式是基于道语言对它的语法支持，这种方式用起来更方便，并且可以有效利用道语言的类型系统进行类型检查。

这里首先介绍Lua相似的协程使用方式（可参考*Lua*文档以了解更多）。但这种方式跟Lua的并不完全一致，请参看道语言的标准库文档以了解有关区别。

下面这个例子修改自Lua文档里的协程例子：

```
routine foo( a )
{
    stdout.println( "foo: ", a );
    return coroutine.yield( 2 * a );
}

routine bar( a, b )
{
    stdout.println( "co-body: ", a, "\t", b );
    r = foo( a + 1 );
    ( r, s ) = coroutine.yield( a+b, a-b );
    stdout.println( "co-body: ", r, "\t", s );
    return b, "end";
}

co = coroutine.create( bar, 1, 10 )

stdout.println( "main", coroutine.resume( co ) )
stdout.println( "main", coroutine.resume( co, "r" ) )
stdout.println( "main", coroutine.resume( co, "x", "y" ) )
# 最后这个调用将导致异常,因为协程已运行到了末尾:
stdout.println( "main", coroutine.resume( co, "x", "y" ) )
```

这里`create()`函数创建一个协程对象，其第一个参数必须是一个函数或函数名。额外的参数将被传递给被创建的协程，如果参数类型不符合`object`的参数类型，将抛出一个异常。协程的初始状态是暂停，必须使用`coroutine.resume()`使它继续运行。额外的`resume()`参数将变成`yield()`的返回值。在协程运行时，可用`yield()`暂停当前协程的运行，将执行权返回给其调用者。`yield()`的参数将变成`resume()`的返回值。

使用协程的另一种方法是使用道语言对其的语法支持。当一个函数被调用时，如果它函数名前有`@`符号，此调用将不执行该函数，而是返回一个发生器或协程。在这样的函数里，可使用`yield`语句来暂停当前运行的协程并向其调用者返回值。当调用者再次重新执行此协程时，协程将从暂停处继续执行。`yield`语句执行完后，它将会象函数调用一样返回值，被返回的值就是其调用者作为参数传入的值。当函数运行至函数末尾或`return`语句后，函数将终止运行，并且不可续。

```
routine foo( a = 0, b = '' )
{
    stdout.println( 'foo:', a );
    return yield( 2 * a, 'by foo()' );
}

routine bar( a = 0, b = '' )
{
    stdout.println( 'bar:', a, b );
    ( r, s ) = foo( a + 1, b );
    stdout.println( 'bar:', r, s );
    ( r, s ) = yield( a + 100, b );
    stdout.println( 'bar:', r, s );
    return a, 'ended';
}

co = @bar( 1, "a" );

stdout.println( 'main: ', co() );
stdout.println( 'main: ', co( 1, 'x' ) );
stdout.println( 'main: ', co( 2, 'y' ) );
# 协程已运行完,再调用将产生异常:
stdout.println( 'main: ', co( 3, 'z' ) );
```

2 使用系统线程

道语言对多线程编程有内在的支持。道语言里对多线程的操作主要是通过`mtlib`，它可用来创建线程，互斥子，条件变量和信号标等，也可以通过它来进行其他线程操作。

2.1 创建线程

在道语言里创建线程是件极其简单的事，只要这样即可，

```
thread_id = mtlib.thread( function / object.method, p1, p2, ... )
thread_id = mtlib.thread( function_curry )
```

这里`mtlib.thread()`的第一个参数必须是一个函数或方法，其余的参数将被传递给其函数参数。如果`function`函数或`object.method`方法有重载的版本，`mtlib.thread()`将根据其额外参数的类型确定具体的哪个函数将被使用。

```
routine MyRout( name : string )
{
    stdio.println( "string parameter = ", name )
}
routine MyRout( index : int )
{
    stdio.println( "int parameter = ", index )
}

class MyType
{
    public

    routine show( name : string ){ stdio.println( "show string: ", name ); }
    routine show( index : int ){ stdio.println( "show number: ", index ); }
}

t1 = mtlib.thread( MyRout, "DAO" )
# or: t1 = mtlib.thread( MyRout{"DAO" } )
t2 = mtlib.thread( MyRout, 111 )
# or: t2 = mtlib.thread( MyRout{111 } )
t1.join();
t2.join();

mt = MyType();
t1 = mtlib.thread( mt.show{ "DAO" } )
t2 = mtlib.thread( mt.show{ 111 } )
t1.join();
t2.join();
```

2.2 异步处理

如果一个程序在没有异步处理下运行多个线程，其执行的结果可能是不确定的。为了让多线程程序给出可预测和合乎需要的结果，程序员可用互斥子，条件变量和信号标等来进行适当的异步处理。

2.2.1 互斥子

互斥子可被用来协调多个线程对一共享数据的读取与修改。它总处于两种状态：已锁和未锁。互斥子只能被一个线程上锁，只能被上锁的线程来解锁，且只能解一次，多次解锁将导致互斥子不可用。当一个线程试图去给一个已锁的互斥子上锁时，该线程将被阻塞（进入等待状态），直到该互斥子被另一线程解锁。

互斥子可由库对象`mtlib`创建：

```
mutex = mtlib.mutex();
```

然后可由下面的方法上锁和解锁，

```
mutex.lock();  
    mutex.unlock();  
    mutex.trylock();
```

2.2.2 条件变量

条件变量是一种可以根据某个条件来进行异步处理的装置。它允许一个线程在一个条件未满足时进入等待状态，直到条件满足时被另一线程唤醒。其基本的操作有：等待（当条件未满足时）和释放信号（当条件满足时，用于唤醒一个或多个在此条件变量上等待的线程）。

```
condvar = mtlib.condition();
```

条件变量总应该和一互斥子在一起使用。在条件变量上等待可用，

```
mtx.lock()  
    condvar.wait( mtx );  
    mtx.unlock();
```

如果要给等待限时可用，

```
mtx.lock()  
    condvar.timedwait( mtx, seconds );  
    mtx.unlock();
```

其中`seconds`可以是小数，如`condvar.timedwait(mtx,0.005)`将最多等待5毫秒。

2.2.3 信号标

信号标可用来限制资源的使用。它有一个计数器，当它的计数大于0时，它允许一个线程来递减它的计数，并让此线程（获取一份信号标所保护的资源）继续执行。如果信号标的计数已为0，任何试图递减其计数以获取资源的线程将阻塞，直到信号标的计数变为大于0。如果一线程递减了一信号标的计数，当它完成相关操作时，它应递增该信号标的计数，以释放资源。创建信号标时需要给出初始计数，

```
sema = mtlb.semaphore( count );
```

To access a resource guarded by a semaphore, use, 要获取信号标资源可用，

```
sema.wait()
```

当信号标资源被成功获取时，其计算将减一。

To release the resource, use 要释放资源可用，

```
sema.post()
```

此方法的调用将使信号标计算增一。

2.3 取消线程

道语言里线程都是可被取消的，要取消一线程可用，

```
thread.cancel()
```

这里thread是要被取消的线程对象。

在目前的道语言实现里,只有两个函数condition.wait()和condition.timedwait() 以及其他某些跟系统相关的函数调用是线程取消点。其他地方的线程取消点必须由以下方法显式插入，

```
mtlib.testcancel()
```

来保证线程可被取消。线程取消后的清理工作完全由道解释器完成。

2.4 线程专有数据

线程库对象mtlib为每个线程创建了一个线程专有哈希表，此哈希表可由mtlib.mydata()获

得，它也可由线程对象的方法 `thread.mydata()` 获得：

```
mtlib.mydata()[ key ] = data;
mtlib.self().mydata()[ key ] = data;
```

例子，

```
routine MyRout( name : string )
{
  stdio.println( "string parameter = ", name )
  mtlib.mydata()["data"] = name;
}

t1 = mtlib.thread( MyRout, "DAO" )
t2 = mtlib.thread( MyRout, "LANGUAGE" )
t1.join();
t2.join();

stdio.println( t1.mydata()["data"] );
stdio.println( t2.mydata()["data"] );
```

3 3. With Asynchronous Function Calls

Probably, the simplest way to create multi-threaded programs in Dao is to use asynchronous function calls (AFC). The way of using AFC is almost identical to that of normal function calls, with the exception that the keyword **async** must follow the call.

```
myfunc( myparams ) async;
myobj.mymeth( myparams ) async;
```

Any functions or methods can be invoked in such asynchronous way!

Normally AFC is executed in a separated native thread, which can be either an idle thread available from the thread pool, or a new thread created on the fly. There is a limit on the size of the thread pool. If this limit is reached, and there is no idle thread available from the pool, the AFCs are scheduled to run when some threads become idle.

Though any functions or methods can be invoked asynchronously, this does not mean they can really be running concurrently in the same time. If multiple AFCs are invoked for the methods of the same class instance, the Dao VM will schedule them to be run sequentially, to guarantee that the members of the class instance will be accessed and/or modified sequentially. In fact, the AFC mechanism is implemented on the top of the Actor Model (see below). In this case, the functions, class instances, C objects and virtual machine process

etc. are the actors. As an intrinsic synchronization mechanism, one principle of the Actor Model is that an actor must process/respond sequentially (to) the messages sent to itself.

However, the Dao VM does not have internal synchronization for data shared in other ways such as parameter passing or global variables etc. For such shared data, mutex, condition variable and semaphore can be used for synchronization. If the current thread has to wait for the completion of all the AFCs invoked by itself before preceeding, keyword **join** can be used together with keyword **async** in the last AFC to join all the AFCs with the current thread.

There is another keyword that can also be used with **async** , that is, **hurry** . Note that, although the garbage collector (GC) of the Dao VM runs concurrently with the program threads, there is a limit on the number of garbage candidates that can be handled by the collector. Once this limit is reached, the program threads are blocked until the collector finishes processing garbages, with the exception of the main thread that is never blocked. If an AFC is used to handle "urgent" task, **hurry** can be used to indicate that the thread running this AFC should not be blocked by the GC.

The return value of AFC is a future value, which is a class instance of a Dao class named **FutureValue** . When the AFC is finished, the **Value** field of this future class instance is set with the returned values of the AFC.

Example,

```
routine test( name : string )
{
    i = 0;
    while( i < 5 ){
        stdio.println( name, " : ", i );
        stdlib.sleep( 1 );
        i ++ ;
    }
    return "future_value_"+name;
}

a = test( "AAA" ) async;
b = test( "BBB" ) async;

list = {};
list.append( test( "CCC" ) async join );

stdio.println( a );
stdio.println( b );
stdio.println( list );

stdio.println( a.Value );
stdio.println( b.Value );
stdio.println( list[0].Value );
```

There following is two simple implementations of parallel merge sort algorithm:

```
routine merge( list1, list2 )
{
    list = {};
    N = list1.size();
    M = list2.size();
    i = 0;
    j = 0;
    while( i < N && j < M ){
        if( list1[i] < list2[j] ){
            list.append( list1[i] )
            i ++;
        }else{
            list.append( list2[j] )
            j ++;
        }
    }
    if( i < N ) list += list1[ i : ];
    if( j < M ) list += list2[ j : ];
    return list;
}

routine merge_front( list1, list2, front, back )
{
    N = list1.size();
    M = list2.size();
    S = N+M;
    i = 0;
    j = 0;
    while( i < N && j < M && front.size()+back.size()<S ){
        if( list1[i] < list2[j] ){
            front.append( list1[i] )
            i ++;
        }else{
            front.append( list2[j] )
            j ++;
        }
    }
}

routine merge_back( list1, list2, front, back )
{
    N = list1.size();
    M = list2.size();
    S = N+M;
    i = N-1;
    j = M-1;
    while( i >=0 && j >=0 && front.size()+back.size()<S ){
        if( list1[i] > list2[j] ){
            back.pushfront( list1[i] )
            i --;
        }else{
            back.pushfront( list2[j] )
            j --;
        }
    }
}
```

```

    }
  }
}

routine sort( list )
{
  N = list.size()
  if( N <= 1 ) return list;

  sublist1 = sort( list[ : N/2-1 ] ) async;
  sublist2 = sort( list[ N/2 : ] ) async join;

#   return merge( sublist1.Value, sublist2.Value );
  front = {};
  back = {};
  merge_front( sublist1.Value, sublist2.Value, front, back ) async;
  merge_back( sublist1.Value, sublist2.Value, front, back ) async join;
  return front + back[ front.size()+back.size()-N : ];
}

list = { 213,1,35,27,49,55,63,75,87,99,115,103 };
sorted = sort( list );

stdio.println( sorted );

```

4 4. With Message Passing Mechanism Based The Actor Model

(实验阶段)

With Message Passing Interface APIs provided in library *MPI*—<http://xdao.org/weblet.dao?WIKI>, one can easily do concurrent and distributed programming in Dao. In the *MPI* library, there are 3 principle functions: `spawn()`, `send()` and `receive()`. With `spawn()`, one can create lightweighted virtual machine processes or real operation system processes, in the local or remove computers; and with `send()` and `receive()`, a process can send message to or receive message from other process.

The Dao *MPI* uses network sockets to do all the inter- OS process and inter-computer communications. When a Dao program is started with a process name specified with “-pNAME” or “-proc-name=NAME” in the command shell, it will bind to a port for accepting communications from other processes. By default, it will try to bind to port number 4115 (D:4, A:1, O:15), if this port is already used, it will try other port to bind. In each computer, the process that binds to port 4115 is the master process, which must be running if Dao program from other computer need to spawn a VM/OS process on this computer.

Each named OS process running Dao program is identified by the host name and

the port it binds to. VM processes within an OS process are identified by names. In general, to be usable by the Dao MPI, the process name (actor address) has the form of "vm_proc@os_proc@@host" ("os_proc" is mapped to the port number), which is called "pid" (process identifier) for simplicity. A pid does not need to have the full form, some parts can be omitted. See also *MPI*—<http://xdao.org/weblet.dao?WIKI> . The following figure displays the relationships between process at different scope.

Each VM process will be running or schedule to run in a seperated thread drawing from a thread pool or created on the fly, and return the thread to the pool when `mpi.receive()` is called. Each OS thread may run different VM processes at different time.

Message Passing Interfaces are available in library **mpi** .

4.0.1 Spawn Process

The prototype of **mpi.spawn()** is

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

If *pid* is of form "", "vm_proc", "vm_proc@os_proc", "vm_proc@os_proc@@host", a VM process will be spawned, and *proc* should be the name of the routine to be spawned. The rest of the parameters will be passed to the routine when the process is created (currently passing additional parameter is not supported for pid form "vm_proc@os_proc", "vm_proc@os_proc@@host"). If the pid is an empty string, then spawned VM process has no pid, and must be accessed by the process handle returned by **mpi.spawn()** .

If *pid* is of form "@os_proc", "@os_proc@@host", an OS process will be spawn in the current computer or a host in the network, and *proc* should be the name of the Dao script file to be executed. The rest parameter is the timeout for spawning. If the OS process does not spawn successfully within the timeout, an exception will be raised. The default value of the timeout is -1. No positive timeout means infinite waiting.

4.0.2 Send Message

The prototype of **mpi.send()** is

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

If *object* is a process identifier, 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.

Note, message is sent out asynchronously, and only number, string, complex number and numeric array can sent through this interface.

If the process identified by *object* , can not be found, an exception will be raised. But if the process is already dead, no exception will be raised, it is up to the user to implement such checking.

4.0.3 Receive Message

The prototype of **mpi.receive()** is

```
mpi.receive( timeout=-1 )
mpi.receive( pid :string, timeout=-1 )
```

If **mpi.receive()** is called without parameters, the calling process will wait for messages from any process, and will be paused definitely until a message arrives. The received message including the pid of the sender will be packed into a list and returned by this API. To avoid it waiting for infinite long time, a timeout can be passed as parameter to **mpi.receive()** , in this case, if no message arrive within the timeout, **mpi.receive()** will return with an empty list.

If one want to receive message from a specific process, the pid of that process can be passed to **mpi.receive()** as the first parameter. Messages from other process will be stored in the "mailbox", until **mpi.receive()** has received and processed a message from that specific process. A timeout can also be specified as the second parameter. By passing the pid of the expected process to **mpi.receive()** , synchronous communication can be realized.

4.0.4 Example

File *mpi_spawn.dao* ,

```
stdio.println( "=====" );
mpi.spawn( "@pid", "mpi_script.dao" );
mpi.spawn( "vmp@pid", "test" );

mpi.send( "@pid", "TO MAIN" );
mpi.send( "@pid", "TO MAIN" );

stdio.println( mpi.receive() );

stdio.println( "=====" );
mpi.spawn( "@pid1", "mpi_script.dao" );
mpi.spawn( "vmp@pid", "test" );
mpi.send( "vmp@pid", "MESSAGE" );
```

```

mpi.spawn( "@pid2@localhost", "mpi_script.dao" );
#mpi.spawn( "@@pid2@pid", "mpi_script.dao" );
stdio.println( "here!!!!!!!!!!!!!!!!!!!!!!!!!!!!" );
mpi.spawn( "vmp@pid2@localhost", "test" );
mpi.send( "vmp@pid2@localhost", "ANOTHER", 123.456 );

```

File *mpi_script.dao* ,

```

stdio.println( "mpi_script.dao spawned" );

routine test()
{
    stdio.println( "start test" );
    i = 0;
    while( i < 10 ){
        msg = mpi.receive();
        stdio.println( "test() ",msg );
        mpi.send( msg[0], "FROM test()" );
        i ++;
    }
}

msg = mpi.receive();
stdio.println( msg );
mpi.send( msg[0], "CONFIRMED" );
mpi.send( "main", "message from main" );
mpi.send( "main", "message from main" );
stdio.println( mpi.receive() );
stdio.println( mpi.receive() );
stdio.println( mpi.receive() );
stdio.println( "xxxxxxxxxxxxxxxxxxxx" );

while( 1 ){ stdlib.sleep(10) }

```

File *mpi_send.dao* ,

```

mpi.send( "vmp@pid", "FROM ANOTHER OS PROCESS" );
stdio.println( mpi.receive() );

```

Run *dao -pmaster mpi_spawn.dao* first, then run *dao -ptest mpi_send.dao* .