Ruby Web Service 应用 – SOAP4R Ruby 数据类型

Ruby 多线程

每个正在系统上运行的程序都是一个进程。每个进程包含一到多个线程。

线程是程序中一个单一的顺序控制流程，在单个程序中同时运行多个线程完成不同的工作,称为多线程。

Ruby 中我们可以通过 Thread 类来创建多线程，Ruby的线程是一个轻量级的，可以以高效的方式来实现并行的代码。

创建 Ruby 线程

要启动一个新的线程，只需要调用 Thread.new 即可:

```
# 线程 #1 代码部分
Thread.new {
# 线程 #2 执行代码
}
# 线程 #1 执行代码
```

实例

以下实例展示了如何在Ruby程序中使用多线程：

实例

```
#!/usr/bin/ruby
def func1
i=0
while i<=2
puts "func1 at: #{Time.now}"
sleep(2)
i=i+1
end
end
def func2
j=0
while j<=2
puts "func2 at: #{Time.now}"
sleep(1)
j=j+1
end
end
puts "Started At #{Time.now}"
t1=Thread.new{func1()}
t2=Thread.new{func2()}
t1.join
t2.join
puts "End at #{Time.now}"
```

尝试一下 »

以上代码执行结果为：

```
Started At Wed May 14 08:21:54 -0700 2014
func1 at: Wed May 14 08:21:54 -0700 2014
func2 at: Wed May 14 08:21:54 -0700 2014
func2 at: Wed May 14 08:21:55 -0700 2014
func1 at: Wed May 14 08:21:56 -0700 2014
func2 at: Wed May 14 08:21:56 -0700 2014
func1 at: Wed May 14 08:21:58 -0700 2014
End at Wed May 14 08:22:00 -0700 2014
```

线程生命周期

1、线程的创建可以使用Thread.new,同样可以以同样的语法使用Thread.start 或者Thread.fork这三个方法来创建线程。

2、创建线程后无需启动，线程会自动执行。

3、Thread 类定义了一些方法来操控线程。线程执行Thread.new中的代码块。

4、线程代码块中最后一个语句是线程的值，可以通过线程的方法来调用，如果线程执行完毕，则返回线程值，否则不返回值直到线程执行完毕。

5、Thread.current 方法返回表示当前线程的对象。 Thread.main 方法返回主线程。

6、通过 Thread.Join 方法来执行线程，这个方法会挂起主线程，直到当前线程执行完毕。

线程状态

线程有5种状态：

线程状态	返回值
可执行	run
睡眠	Sleeping
退出	aborting
正常终止	false
发生异常终止	nil

线程和异常

当某线程发生异常，且没有被rescue捕捉到时，该线程通常会被无警告地终止。但是，若有其它线程因为Thread#join的关系一直等待该线程的话，则等待的线程同样会被引发相同的异常。

```
begin
t = Thread.new do
Thread.pass # 主线程确实在等join
raise "unhandled exception"
end
t.join
rescue
```

```
p $! # => "unhandled exception"
end
```

使用下列3个方法，就可以让解释器在某个线程因异常而终止时中断运行。

- 启动脚本时指定**-d**选项，并以调试模时运行。
- 用`Thread.abort_on_exception`设置标志。
- 使用`Thread#abort_on_exception`对指定的线程设定标志。

当使用上述3种方法之一后，整个解释器就会被中断。

```
t = Thread.new { ... }
t.abort_on_exception = true
```

线程同步控制

在Ruby中，提供三种实现同步的方式，分别是：

1. 通过Mutex类实现线程同步

2. 监管数据交接的Queue类实现线程同步

3. 使用ConditionVariable实现同步控制

通过Mutex类实现线程同步

通过Mutex类实现线程同步控制，如果在多个线程钟同时需要一个程序变量，可以将这个变量部分使用lock锁定。 代码如下：

实例

```
#!/usr/bin/ruby
require "thread"
puts "Synchronize Thread"
@num=200
@mutex=Mutex.new
def buyTicket(num)
@mutex.lock
if @num>=num
@num=@num-num
puts "you have successfully bought #{num} tickets"
else
puts "sorry,no enough tickets"
end
@mutex.unlock
end
ticket1=Thread.new 10 do
10.times do |value|
ticketNum=15
buyTicket(ticketNum)
sleep 0.01
end
end
ticket2=Thread.new 10 do
10.times do |value|
```

```
ticketNum=20
buyTicket(ticketNum)
sleep 0.01
end
end
sleep 1
ticket1.join
ticket2.join
```

尝试一下 »

以上代码执行结果为：

```
Synchronize Thread
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
```

除了使用lock锁定变量，还可以使用try_lock锁定变量，还可以使用Mutex.synchronize同步对某一个变量的访问。

监管数据交接的Queue类实现线程同步

Queue类就是表示一个支持线程的队列，能够同步对队列末尾进行访问。不同的线程可以使用统一个对类，但是不用担心这个队列中的数据是否能够同步，另外使用SizedQueue类能够限制队列的长度

SizedQueue类能够非常便捷的帮助我们开发线程同步的应用程序，应为只要加入到这个队列中，就不用关心线程的同步问题。

经典的生产者消费者问题：

实例

```
#!/usr/bin/ruby
require "thread"
puts "SizedQuee Test"
```

```ruby
queue = Queue.new
producer = Thread.new do
10.times do |i|
sleep rand(i) # 让线程睡眠一段时间
queue << i
puts "#{i} produced"
end
end
consumer = Thread.new do
10.times do |i|
value = queue.pop
sleep rand(i/2)
puts "consumed #{value}"
end
end
consumer.join
```

尝试一下 »

程序的输出：

```
SizedQuee Test
0 produced
1 produced
consumed 0
2 produced
consumed 1
consumed 2
3 produced
consumed 34 produced

consumed 4
5 produced
consumed 5
6 produced
consumed 6
7 produced
consumed 7
8 produced
9 produced
consumed 8
consumed 9
```

线程变量

线程可以有其私有变量，线程的私有变量在线程创建的时候写入线程。可以被线程范围内使用，但是不能被线程外部进行共享。

但是有时候，线程的局部变量需要别别的线程或者主线程访问怎么办？ruby当中提供了允许通过名字来创建线程变量，类似的把线程看做hash式的散列表。通过[]=写入并通过[]读出数据。我们来看一下下面的代码：

实例

```
#!/usr/bin/ruby
count = 0
arr = []
10.times do |i|
arr[i] = Thread.new {
sleep(rand(0)/10.0)
Thread.current["mycount"] = count
count += 1
}
end
arr.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"
```

以上代码运行输出结果为：

```
8, 0, 3, 7, 2, 1, 6, 5, 4, 9, count = 10
```

主线程等待子线程执行完成，然后分别输出每个值。 。

线程优先级

线程的优先级是影响线程的调度的主要因素。其他因素包括占用CPU的执行时间长短，线程分组调度等等。

可以使用 Thread.priority 方法得到线程的优先级和使用 Thread.priority= 方法来调整线程的优先级。

线程的优先级默认为 0 。 优先级较高的执行的要快。

一个 Thread 可以访问自己作用域内的所有数据，但如果有需要在某个线程内访问其他线程的数据应该怎么做呢？ Thread 类提供了线程数据互相访问的方法，你可以简单的把一个线程作为一个 Hash 表，可以在任何线程内使用 []= 写入数据，使用 [] 读出数据。

```
athr = Thread.new { Thread.current["name"] = "Thread A"; Thread.stop }
bthr = Thread.new { Thread.current["name"] = "Thread B"; Thread.stop }
cthr = Thread.new { Thread.current["name"] = "Thread C"; Thread.stop }
Thread.list.each {|x| puts "#{x.inspect}: #{x["name"]}" }
```

可以看到，把线程作为一个 Hash 表，使用 [] 和 []= 方法，我们实现了线程之间的数据共享。

线程互斥

Mutex(Mutal Exclusion = 互斥锁)是一种用于多线程编程中，防止两条线程同时对同一公共资源（比如全局变量）进行读写的机制。

不使用Mutax的实例

实例

```ruby
#!/usr/bin/ruby
require 'thread'
count1 = count2 = 0
difference = 0
counter = Thread.new do
loop do
count1 += 1
count2 += 1
end
end
spy = Thread.new do
loop do
difference += (count1 - count2).abs
end
end
sleep 1
puts "count1 : #{count1}"
puts "count2 : #{count2}"
puts "difference : #{difference}"
```

以上实例运行输出结果为：

```
count1 :  9712487
count2 :  12501239
difference : 0
```

使用Mutax的实例

实例

```ruby
#!/usr/bin/ruby
require 'thread'
mutex = Mutex.new
count1 = count2 = 0
difference = 0
counter = Thread.new do
loop do
mutex.synchronize do
count1 += 1
count2 += 1
end
end
end
spy = Thread.new do
loop do
mutex.synchronize do
difference += (count1 - count2).abs
end
end
end
sleep 1
mutex.lock
```

```
puts "count1 : #{count1}"
puts "count2 : #{count2}"
puts "difference : #{difference}"
```

以上实例运行输出结果为：

```
count1 :  1336406
count2 :  1336406
difference : 0
```

死锁

两个以上的运算单元，双方都在等待对方停止运行，以获取系统资源，但是没有一方提前退出时，这种状况，就称为死锁。

例如，一个进程 p1占用了显示器，同时又必须使用打印机，而打印机被进程p2占用，p2又必须使用显示器，这样就形成了死锁。

当我们在使用 Mutex 对象时需要注意线程死锁。

实例

```
#!/usr/bin/ruby
require 'thread'
mutex = Mutex.new
cv = ConditionVariable.new
a = Thread.new {
mutex.synchronize {
puts "A: I have critical section, but will wait for cv"
cv.wait(mutex)
puts "A: I have critical section again! I rule!"
}
}
puts "(Later, back at the ranch...)"
b = Thread.new {
mutex.synchronize {
puts "B: Now I am critical, but am done with cv"
cv.signal
puts "B: I am still critical, finishing up"
}
}
a.join
b.join
```

以上实例输出结果为：

```
A: I have critical section, but will wait for cv
(Later, back at the ranch...)
B: Now I am critical, but am done with cv
B: I am still critical, finishing up
A: I have critical section again! I rule!
```

线程类方法

完整的 Thread（线程）类方法如下：

序号	方法描述
1	**Thread.abort_on_exception** 若其值为真的话，一旦某线程因异常而终止时，整个解释器就会被中断。它的默认值是假，也就是说，在通常情况下，若某线程发生异常且该异常未被Thread#join等检测到时，该线程会被无警告地终止。
2	**Thread.abort_on_exception=** 如果设置为 *true*, 一旦某线程因异常而终止时，整个解释器就会被中断。返回新的状态
3	**Thread.critical** 返回布尔值。
4	**Thread.critical=** 当其值为true时，将不会进行线程切换。若当前线程挂起(stop)或有信号(signal)干预时，其值将自动变为false。
5	**Thread.current** 返回当前运行中的线程(当前线程)。
6	**Thread.exit** 终止当前线程的运行。返回当前线程。若当前线程是唯一的一个线程时，将使用exit(0)来终止它的运行。
7	**Thread.fork { block }** 与 Thread.new 一样生成线程。
8	**Thread.kill(aThread)** 终止线程的运行.
9	**Thread.list** 返回处于运行状态或挂起状态的活线程的数组。
10	**Thread.main** 返回主线程。
11	**Thread.new([arg]*) {| args | block }** 生成线程,并开始执行。数会被原封不动地传递给块. 这就可以在启动线程的同时,将值传递给该线程所固有的局部变量。
12	**Thread.pass** 将运行权交给其他线程. 它不会改变运行中的线程的状态,而是将控制权交给其他可运行的线程(显式的线程调度)。

13	**Thread.start([args]*) {| args	block }** 生成线程,并开始执行。数会被原封不动地传递给块. 这就可以在启动线程的同时,将值传递给该线程所固有的局部变量。
14	**Thread.stop** 将当前线程挂起,直到其他线程使用run方法再次唤醒该线程。

线程实例化方法

以下实例调用了线程实例化方法 join：

实例

```
#!/usr/bin/ruby
thr = Thread.new do # 实例化
puts "In second thread"
raise "Raise exception"
end
thr.join # 调用实例化方法 join
```

以下是完整实例化方法列表：

序号	方法描述
1	**thr[name]** 取出线程内与name相对应的固有数据。 name可以是字符串或符号。 若没有与name相对应的数据时, 返回nil。
2	**thr[name] =** 设置线程内name相对应的固有数据的值， name可以是字符串或符号。 若设为nil时, 将删除该线程内对应数据。
3	**thr.abort_on_exception** 返回布尔值。
4	**thr.abort_on_exception=** 若其值为true的话，一旦某线程因异常而终止时，整个解释器就会被中断。
5	**thr.alive?** 若线程是"活"的,就返回true。
6	**thr.exit** 终止线程的运行。返回self。
7	**thr.join** 挂起当前线程,直到self线程终止运行为止. 若self因异常而终止时, 将会当前线程引发同样的异常。
8	**thr.key?** 若与name相对应的线程固有数据已经被定义的话,就返回true

9	**thr.kill** 类似于 *Thread.exit* 。
10	**thr.priority** 返回线程的优先度. 优先度的默认值为0. 该值越大则优先度越高.
11	**thr.priority=** 设定线程的优先度. 也可以将其设定为负数.
12	**thr.raise(anException)** 在该线程内强行引发异常.
13	**thr.run** 重新启动被挂起(stop)的线程. 与wakeup不同的是,它将立即进行线程的切换. 若对死进程使用该方法时, 将引发ThreadError异常.
14	**thr.safe_level** 返回self 的安全等级. 当前线程的safe_level与$SAFE相同.
15	**thr.status** 使用字符串"run"、"sleep"或"aborting" 来表示活线程的状态. 若某线程是正常终止的话,就返回false. 若因异常而终止的话,就返回nil。
16	**thr.stop?** 若线程处于终止状态(dead)或被挂起(stop)时,返回true.
17	**thr.value** 一直等到self线程终止运行(等同于join)后,返回该线程的块的返回值. 若在线程的运行过程中发生了异常, 就会再次引发该异常.
18	**thr.wakeup** 把被挂起(stop)的线程的状态改为可执行状态(run), 若对死线程执行该方法时,将会引发ThreadError异常。

← Ruby Web Service 应用 – SOAP4R

Ruby 数据类型 →

点我分享笔记