RGSS3 小探（三）

——面向编辑器

1 引言

RPG Maker VX ACE 在工具上也做出了重大的改变，使得对于制作游戏的新手更容易上手。因此，RGSS3 系统在处理由编辑器确定的数据方面也做出了相应的改变。本文将从几个方面来探讨这些改变以及给我们带来的影响。

2 Fiber

不同于 Ruby 1.8，Ruby 1.9 引入了一个叫做**纤程**（Fiber）的概念来替代**线程**（Thread）。对比“纤”和“线”，你大概会猜到，纤程实际上是一种轻量级的线程，相较于线程，他体积更加的小，更容易维护，也有着强大的在**上下文**（Context）间切换的能力[①]。

纤程跟传统意义上的**协程**（coroutine）是同一个概念，是一种非抢占式的多线程模型。用紫苏在 Ruby/RGSS Tips 中提到的例子作喻：

合作式多任务模型，显然就是指执行任务的对象之间有团队精神，可以协同工作。第一个对象完成了一部分工作，就可以把成果交给第二个对象，让他在成果的基础上继续工作，并等待他完成。所以，我们使用纤程的主要目的，就是实现这种所谓的“协程”。比如：动物园的售票系统模型，观光者需要售票员处理售票相关的事项，然后把票递给自己；而售票员需要观光者把钱递给自己。

并援引 Ruby 帮助文档中的例子作释，#=> 符号后则是输出：

代码片段 2.1 Ruby 1.9.2-p180::Fiber

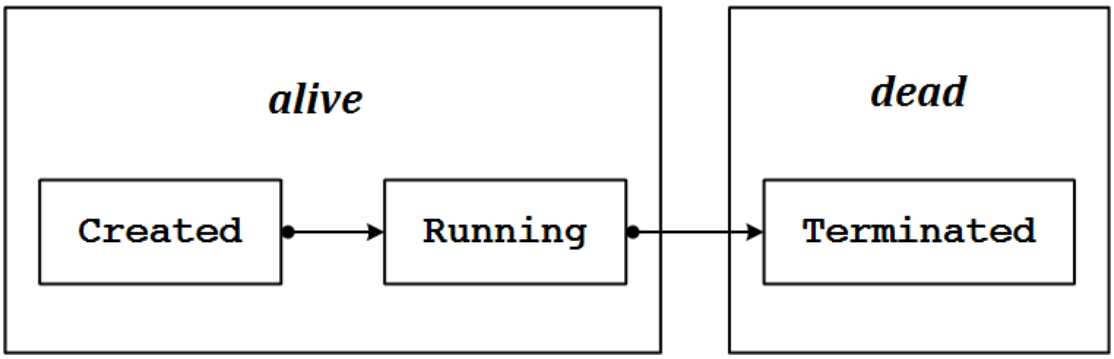
```
fiber = Fiber.new do
  Fiber.yield 1
  2
end

puts fiber.resume #=> 1
puts fiber.resume #=> 2
puts fiber.resume #=> FiberError: dead fiber called
```

首先要阐明的是 Fiber 的建立：Ruby 中内置了 Fiber 类[②]，建立一个有效的 Fiber

仅需要标准化的调用 Fiber.new 了。纤程也是程序，因此你要给他指明需要执行的代码，因此需要在建立的时候传递一个**代码块**（Code Block）。用 do...end 或{}将其括住皆可，但我们一般使用 do...end。

Fiber 有三种状态：Created、Running、Terminated，当 Fiber 处于前两种状态时，他就是**有效的**（alive），当他处于 Terminated 的时候，他即是**失效的**（dead）。Created 态表明 Fiber 刚刚被创立，还没有开始执行，而 Running 态则表示 Fiber 正在执行中。每一个 Fiber 在被创建后都处于 Created 态，即他们不会立即执行，需要使用 resume 方法使他们转移到 Running 态。而当 Fiber 执行完后，即会转移到 Terminated 态，此时 Fiber 即是 dead。区分一个 Fiber 是 alive 还是 dead 很重要[3]，因为对一个失效的 Fiber 调用 resume 方法会产生 FiberError！



好了，让我们回头看看代码吧，我们首先创立了一个 Fiber，然后调用 resume 方法启动他。请注意，小写的 fiber 是指我们创建的纤程实例，而大写的 Fiber 则是特指 Fiber 类：

```
fiber 开始执行...
代码是 Fiber.yield(1)...
fiber 挂起，返回 1...
挂起中...
```

这又是怎么回事？这一切并不显得那么不可思议，Fiber 实现的是异步 IO，因此可以实现不同上下文间的交互。当我们调用 fiber.resume，就告知 fiber：“你可以开始运行了”，我们就把控制权交给了 fiber。我们想要重新获得控制权，就得等 fiber 死去，即当他执行到代码尾并失效。但这不是唯一的方法，类方法 Fiber.yield 就可以暂时挂起 fiber 的执行，并把控制权交还给调用者（我们）。如果 Fiber.yield 后面还跟有参数的话，那么我们在重新获得控制权的同时，也会得到这个参数，这就实现了信息的交互。

当我们下一次调用 resume 时，fiber 即从上次被挂起的地方重新开始执行，直到又遇到 Fiber.yield 或者执行到代码尾。

事实上，resume 方法也可以带上一个参数，来实现更复杂的信息交互。我们再使用一个来自于 Ruby 帮助文档的例子：

代码片段 2.2 Ruby1.9.2-p180::Fiber

```
fiber = Fiber.new do |first|
  second = Fiber.yield first + 2
end

puts fiber.resume 10 #=> 12
puts fiber.resume 14 #=> 14
puts fiber.resume 18 #=> FiberError: dead fiber called
```

这个例子在为Fiber创建Block的时候强调了此Block会有一个参数first，因此，在我们第一次调用resume的时候（注意此时fiber由Created态转为Running态），first将会被赋值，即为 resume 的参数。我会用第一人称来描述整个过程：

（我们）调用 puts fiber.resume(10)
（我们）求值 fiber.resume(10)
　　（fiber） fiber 开始执行
　　（fiber）哦，有参数传递过来，first 应该为 10
　　（fiber）执行代码 second = Fiber.yield(first + 2)
　　（fiber）求值 Fiber.yield(first + 2)
　　（fiber） first + 2 是 12
　　（fiber）哦，Fiber.yield，我得停下来
　　（fiber）记录好我执行到哪里了
　　（fiber）好吧，我返回 first + 2，就是 12
（我们） fiber 返回 12，那么我应该执行 puts 12
（我们）调用 puts fiber.resume(14)
（我们）求值 fiber.resume(14)
　　（fiber）哦！我又被唤醒了！
　　（fiber）老大哥[④]给我说是 14，我执行到哪里了？
　　（fiber）不，老大哥叫我把 Fiber.yield(12)看做 14
　　（fiber）好吧，second = 14
　　（fiber）我的工作全部完成了，我最后一次返回的是 14
　　（fiber）本人已死，有事烧纸~~我返回 14
（我们） fiber 返回 14，那么我应该执行 puts 14
（我们）调用 puts fiber.resume(18)
（我们）求值 fiber.resume(18)
　　（Ruby 解释器） FiberError!!!fiber 已死，有事烧纸

对，整个过程就是这样。

3 RGSS3 中的 Game_Interpreter

在彻底的谈 RGSS3 中引入 Fiber 后的应用之前，我们有必要来看看 RGSS3 中一些并

不算是革命性的改变。其中 execute_command 方法的修改，使得 RGSS3 与 RGSS2 相比 Interpreter 部分节省了不少的代码。

代码片段 3.1 RGSS3::Game_Interpreter

```
#--------------------------------------------------------------
# ● 执行事件指令
#--------------------------------------------------------------
def execute_command
  command = @list[@index]
  @params = command.parameters
  @indent = command.indent
  method_name = "command_#{command.code}"
  send(method_name) if respond_to?(method_name)
end
```

而 RGSS2 中，execute_command 的代码则从 198 行写到了 393 行。RGSS2 使用了人工用 case 关键字来判断事件指定的 code 值，因此显得十分笨拙。而 RGSS3 利用 Ruby 元编程的特性，巧妙构造了几个语句，的确少做了许多无用功。而关于实际用到的元编程技术，并不在本文的探讨范围之内，请在[参考文献]一节中查找相关资料。

在 RGSS3 的 Game_Interpreter 创建了一个 Fiber 实例：

代码片段 3.2 RGSS3::Game_Interpreter

```
#--------------------------------------------------------------
# ● 生成纤程
#--------------------------------------------------------------
def create_fiber
  @fiber = Fiber.new { run } if @list
end
```

这个纤程就负责处理、解释种种事件指令。还有三个很有趣的方法，定义如下：

代码片段 3.3 RGSS3::Game_Interpreter

```
#--------------------------------------------------------------
# ● 判定是否执行中
#--------------------------------------------------------------
def running?
  @fiber != nil
end
#--------------------------------------------------------------
# ● 更新画面
#--------------------------------------------------------------
def update
```

```
    @fiber.resume if @fiber
  end
  #--------------------------------------------------------------
  # ● 等待
  #--------------------------------------------------------------
  def wait(duration)
    duration.times { Fiber.yield }
  end
```

事件在执行中吗？当然我们得看@fiber 的死活，可是我们又没有 fiber 库，不能使用 Fiber#alive?方法,因此,RGSS3 就聪明的通过检验@fiber 是否为 nil 来做出判断。那么 RGSS3 中的“等待”又是怎样实现的呢？

哈哈，看看 wait 方法吧：他调用了制定帧数次的 Fiber.yield。后果又怎样呢？好戏就上演了呗！

(@fiber) 我不干了！我挂起 [在 wait 方法]
(老大哥) 你还没死啊，给我继续干活 [在 update 方法]
(@fiber) 我不干了！我挂起 [在 wait 方法]
(老大哥) 你还没死啊，给我继续干活 [在 update 方法]
(@fiber) 我不干了！我挂起 [在 wait 方法]
(老大哥) 你还没死啊，给我继续干活 [在 update 方法]
...数帧后...
(@fiber) 工作、工作、该死的地主 [在某个方法中]

那么老大哥叫@fiber 去执行 run，他要怎么 run 呢？

代码片段 3.4 RGSS3::Game_Interpreter

```
  #--------------------------------------------------------------
  # ● 执行
  #--------------------------------------------------------------
  def run
    wait_for_message
    while @list[@index] do
      execute_command
      @index += 1
    end
    Fiber.yield
    @fiber = nil
  end
```

如果能获取需要执行的指令，那么就调用 execute_command 来执行，并递推索引。这些执行完后再调用一次 Fiber.yield（请注意，由 execute_command 方法执行的指

令也可能会调用 Fiber.yield)，往后再调用@fiber.resume 的话则会将@fiber 赋值为 nil。

n 参考文献

此处列出了可供参考的文献资料，以方便读者检索：

[1]:Ruby/RGSS Tips，紫苏关于 Fiber 的发言：
http://bbs.66rpg.com/forum.php?mod=redirect&goto=findpost&ptid=154785&pid=1545398
[2]:庄周梦蝶，Ruby Fiber 指南（一）：
http://www.blogjava.net/killme2008/archive/2010/03/11/315158.html
[3]:David，Coroutines (via fibers) in Ruby 1.9：
http://www.davidflanagan.com/2007/08/coroutines-via-fibers-in-ruby-19.html
[4]:**协程**（Coroutine）：http://zh.wikipedia.org/wiki/%E5%8D%8F%E7%A8%8B
[5]: 芝哲史、笹田耕一， Ruby 1.9 での高速な Fiber の実装：
http://www.atdot.net/~ko1/activities/shiba-prosym2010-paper.pdf

① 本段参考了诸多资料，请在[参考文献]一节中查找。
② Ruby 里面有一个叫做 fiber 的标准库，是对 Fiber 的加强。
③ fiber 库中提供了 Fiber#alive?方法检测 Fiber，返回 true 或 false。
④ 参考英国作家 *George Orwell* 的著作 *Nineteen Eighty-four*。