Ruby 101

Advanced Ruby: Fibers

Advanced Ruby:

When threads are too much: Fibers

Like threads, except:

- Fibers must be controlled by the programmer
- They are never automatically scheduled
- Fibers must cooperate with scheduling

Advanced Ruby:

Creating a Fiber:

```
my_fiber = Fiber.new do

    ...

end
```

Advanced Ruby:

Fiber class methods:

`Fiber.current` => yields the current fiber (must `require 'fiber'`)

`Fiber.yield` => Yields control to the program context which called `.resume`

Fiber instance methods:

`.alive?` => true if fiber can be resumed or transferred (must `require 'fiber'`)

`.resume` *vars* => starts a fiber or resumes it

`.transfer` *vars* => transfers from current fiber to recieving fiber; fibers tranferring must have control transferred back to continue processing; can't be resumed (must `require 'fiber'`)

(variables -> passed to program block on first execution, then passed as the value of the last yield statement to be executed)

Advanced Ruby:

Fiber Example

```
count_fiber = Fiber.new do |base|
    next_resume = Fiber.yield base += 1
    next_resume += 1
end

a = count_fiber.resume 1
puts a
b = count_fiber.resume a + 2
puts b
```

--------Output------

```
2
5
```

Advanced Ruby:

Fiber Example

```
count_fiber = Fiber.new do |base|
    next_resume = Fiber.yield base += 1
    next_resume += 1
end

a = count_fiber.resume 1
puts a
b = count_fiber.resume a + 2
puts b
```

--------Output------

```
2
5
```

Advanced Ruby:

Fiber Example 2

```
require 'fiber'
count_fiber = Fiber.new do |base|
    next_resume = Fiber.yield base += 1
    next_resume += 1
end
starting_fiber = Fiber.new do
    puts "Start with 1"
    next_var = count_fiber.transfer 1
    count_fiber.transfer next_var
end
a = starting_fiber.resume
puts a
a = starting_fiber.transfer a
puts a
a = starting_fiber.transfer a
puts a
```

Advanced Ruby:

Fiber Example: Output

```
Start with 1
2
3
3
```