Technical Explorations

by Keith R. Bennett

Ruby's inject/reduce and each_with_object

Nov 22, 2013 • keithrbennett

For an object oriented language, Ruby’s functional features are pretty awesome. The productivity boost of Enumerable methods was one of the most exciting things for me when I first encountered Ruby, and that has continued to be the case.

In the examples below, I’ll use trivial sum methods to illustrate. Assume the parameter passed is always an array of numbers, `[1, 2, 3]`.

`each`, `map`, and `select` were simple to understand and implement, but `inject` (`reduce`) took a little more effort. When I see code like the example below, I remember the times when I was inject-phobic:

```
def verbose_sum(numbers)
  sum = 0
  numbers.each { |n| sum += n }
  sum
end
```

This is way more verbose than it needs to be. Consider the equivalent inject method:

```
def concise_sum(numbers)
  numbers.inject(0) { |sum, n| sum += n }
end
```

…which can be reduced even further, since we’re calling a method on each object that takes no arguments (the + method):

```
def more_concise_sum(numbers)
  numbers.inject(0, :+)
end
```

One can even omit the zero and it will be inferred:

```
def even_more_concise_sum(numbers)
  numbers.inject(:+)
end
```

Nice, eh? However, let’s revisit the block variant of the method and see what happens if we add a `puts` statement at the end of such a block, and then call it:

```
def concise_sum(numbers)
  numbers.inject(0) do |sum, n|
    sum += n
    puts "sum is now #{sum}."
  end
end

# produces: NoMethodError: undefined method `+' for nil:NilClass
```

What happened? When using `inject`, the value returned by the block is the value `inject` will use as the memo for the next iteration. Since `puts` returns `nil`, and it was the last expression in the block, it was used as the memo in the next iteration, and the error occurred.

Enter `each_with_object`. Instead of using the block's return value as the memo for the next iteration, `each_with_object` unconditionally passes the object with which it was initialized. It relies on you to modify that object as per your needs in the block. So the `each_with_object` version of `sum` would look like this:

```
def ewo_sum(numbers)
  numbers.each_with_object(0) { |n, sum| sum += n }
end
```

Note that the order of the parameters is reversed, compared with inject. I remember the order by remembering that it's the same order listed in the method name itself – *each* is the object for each iteration and comes first, and *with_object* is the memo object and comes next.

When we run this code, we get…zero. WTF!?!?!?!?

Let's see if it works using a hash instead. For the example, this hash will contain each number as a key, with the key's `to_s` representation as the value:

```
def stringified_key_hash(numbers)
  numbers.each_with_object({}) do |n, hsh|
    hsh[n] = n.to_s
  end
end
```

When we run this, we get:

```
>=> {1=>"1", 2=>"2", 3=>"3"}
```

This worked! So how are the two different? As previously mentioned, the block must modify the object initially passed to the `each_with_object` method. In the case of `stringified_key_hash`, we're fine because we've passed in a `Hash` instance, and when we modify it using `[]=` in every iteration, we're always dealing with that same hash instance.

In contrast, when we used `each_with_object` in `ewo_sum`, the initial value was a `Fixnum` whose value was 0. The expression "`sum += n`" assigned and returned *a different instance* of `Fixnum`. Note that the object id's for `sum` differ before and after this expression is evaluated:

```
[21] pry(main)> sum = 0
=> 0
[22] pry(main)> sum.object_id
=> 1
[23] pry(main)> sum += 3
=> 3
[24] pry(main)> sum.object_id
=> 7
```

Since, as we said, the initial value is unconditionally passed to the block in each iteration, the revised value created in the block was discarded. So, when using `each_with_object`, be sure that the modifications are being made to the original memo instance.

Now let's go back to the earlier point about having to return the memo as the last expression of the block. Since `each_with_object` unconditionally passes the initial object, there is no need for the block to return it. If we add a puts to `stringified_key_hash`, we still get the correct result:

```
def stringified_key_hash(numbers)
  numbers.each_with_object({}) do |n, hsh|
    hsh[n] = n.to_s
    puts "Hash is now #{hsh}."
  end
end

Hash is now {1=>"1"}.
Hash is now {1=>"1", 2=>"2"}.
Hash is now {1=>"1", 2=>"2", 3=>"3"}.
=> {1=>"1", 2=>"2", 3=>"3"}
```

A minor point about my choice of `hsh` as a variable name…it's a good idea not to use `hash` as a variable name, because, in any object that is a class that includes `Kernel` in its ancestors, `hash` will be a method name:

```
>[36] pry(main)> hash
=> -1606748642386923196
[37] pry(main)> Object.new.hash
=> 4200367341767882288
```

While it's unlikely that this name collision would bite you, it's not impossible. Better to avoid the possibility altogether.

And why do I use `hsh` and not a more descriptive name like the method name `stringified_key_hash`? We already have the more descriptive method name, where it is most valuable, since that name is for the exposed API, whereas the block variable is one that API users need never see. The need for a descriptive name for the block variable is greatly reduced by its narrow scope and its proximity to the more descriptive method name.

Conclusion

One could say that `inject` and `each_with_object` are different methods that behave differently intentionally, and one should choose which one to use based on the use case. However, in my (perhaps limited) experience, I have never encountered the need to return instances different from the initial instance in a block, and I find myself *always* using `each_with_object` these days. The only reason I even discovered the `each_with_object` `Fixnum` issue was that I was involved in a discussion about `each_with_object` and wanted to produce a minimal example of it.

That said, isn't it great how many choices we have? More than any other piece of code I know of, the Enumerable (2.5, 2.0) module is a treasure trove that perpetually pleases.

Published with GitHub Pages