

Jay Fields' Thoughts

experiences in software development

Friday, March 21, 2008

Ruby: inject

I love inject. To be more specific, I love Enumerable#inject. I find it easy to read and easy to use. It's powerful and it lets me be more concise. Enumerable#inject is a good thing.

Of course, I didn't always love it. When I was new to Ruby I didn't understand what it was, so I found it hard to follow. However, finding it hard to understand didn't make me run from it, instead I wanted to know what all the hype was about. Enumerable#inject is an often used method by many great Rubyists, and I wanted to know what I was missing.

So, to learn about Enumerable#inject I did what I always do, I used it every possible way I could think of.

Example 1: Summing numbers

Summing numbers is the most common example for using inject. You have an array of numbers and you want the sum of those numbers.

```
[1, 2, 3, 4].inject(0) { |result, element| result + element } # => 10
```

If the example isn't straightforward, don't worry, we're going to break it down. The inject method takes an argument and a block. The block will be executed once for each element contained in the object that inject was called on ([1,2,3,4] in our example). The argument passed to inject will be yielded as the first argument to the block, the first time it's executed. The second argument yielded to the block will be the first element of the object that we called inject on.

So, the block will be executed 4 times, once for every element of our array ([1,2,3,4]). The first time the block executes the result argument will have a value of 0 (the value we passed as an argument to inject) and the element argument will have a value of 1 (the first element in our array).

You can do anything you want within the block, but the return value of the block is very important. The return value of the block will be yielded as the result argument the next time the block is executed.

In our example we add the result, 0, to the element, 1. Therefore, the return value of the block will be 0 + 1, or 1. This will result in 1 being yielded as the result argument the second time the

block is executed.

The second time the block is executed the result of the previous block execution, 1, will be yielded as the result, and the second element of the array will be yielded as the element. Again the result, 1, and the element, 2 will be added together, resulting in the return value of the block being 3.

The third time the block is executed the result of the second block execution, 3, is yielded as the result argument and the third element of the array, 3, will be yielded as the element argument. Again, the result and the element will be added, and the return value of the block for the third execution will be 6.

The fourth time will be the final time the block is executed since there are only 4 elements in our array. The result value will be 6, the result from the third execution of the block, and the element will be 4, the fourth element of the array. The block will execute, adding four plus six, and the return value of the block will be 10. On the final execution of the block the return value is used as the return value of the inject method; therefore, as the example shows, the result of executing the code above is 10.

That's the very long version of how inject works, but you could actually shortcut one of the block executions by not passing an argument to inject.

```
[1, 2, 3, 4].inject { |result, element| result + element } # => 10
```

As the example shows, the argument to inject is actually optional. If a default value is not passed in as an argument the first time the block executes the first argument (result from our example) will be set to the first element of the enumerable (1 from our example) and the second argument (element from our example) will be set to the second element of the enumerable (2 from our example).

In this case the block will only need to be executed 3 times, since the first execution will yield both the first and the second element. The first time the block executes it will add the result, 1, to the element, 2, and return a value of 3. The second time the block executes the result will be 3 and the element will also be 3. All additional steps will be the same, and the result will be 10 once again.

Summing numbers with inject is a simple example of taking an array of numbers and building a resulting sum one element at a time.

Example 2: Building a Hash

Sometimes you'll have data in one format, but you really want it in another. For example, you may have an array that contains keys and values as pairs, but it's really just an array of arrays. In that case, inject is a nice solution for quickly converting your array of arrays into a hash.

```
hash = [[:first_name, 'Shane'], [:last_name, 'Harvie']].inject({}) do |result,
```

```

element |
  result[element.first] = element.last
  result
end

hash # => {:first_name=>"Shane", :last_name=>"Harvie"}

```

As the example shows, I start with an empty hash (the argument to inject) and I iterate through each element in the array adding the key and value one at a time to the result. Also, since the result of the block is the next yielded result, I need to add to the hash, but explicitly return the result on the following line.

Ola Bini and rubikitch both pointed out that you can also create a hash from an array with the following code.

```

Hash[*[[:first_name, 'Shane'], [:last_name, 'Harvie']].flatten] # =>
{:first_name=>"Shane", :last_name=>"Harvie"}

```

Of course, I can do other things in inject also, such as converting the keys to be strings and changing the names to be lowercase.

```

hash = [[:first_name, 'Shane'], [:last_name, 'Harvie']].inject({}) do |result,
element|
  result[element.first.to_s] = element.last.downcase
  result
end

hash # => {"first_name"=>"shane", "last_name"=>"harvie"}

```

This is a central value for inject, it allows me to easily convert an enumerable into an object that is useful for the problem I'm trying to solve.

Example 3: Building an Array

Enumerable gives you many methods you need for manipulating arrays. For example, if want all the integers of an array, that are even, as strings, you can do so chaining various methods from Enumerable.

```

[1, 2, 3, 4, 5, 6].select { |element| element % 2 == 0 }.collect { |element|
element.to_s } # => ["2", "4", "6"]

```

Chaining methods of Enumerable is a solution that's very comfortable for many developers, but as the chain gets longer I prefer to use inject. The inject method allows me to handle everything I need without having to chain multiple independent methods.

The code below achieves the same thing in one method, and is just as readable, to me.

```
array = [1, 2, 3, 4, 5, 6].inject({}) do |result, element|
  result << element.to_s if element % 2 == 0
  result
end

array # => ["2", "4", "6"]
```

Of course, that example is a bit contrived; however, a realistic example is when you have an object with two different properties and you want to build an array of one, conditionally based on the other. A more concrete example is an array of test result objects that know if they've failed or succeeded and they have a failure message if they've failed. For reporting, you want all the failure messages.

You can get this with the built in methods of Enumerable.

```
TestResult = Struct.new(:status, :message)
results = [
  TestResult.new(:failed, "1 expected but was 2"),
  TestResult.new(:success),
  TestResult.new(:failed, "10 expected but was 20")
]

messages = results.select { |test_result| test_result.status == :failed }.collect {
  |test_result| test_result.message }
messages # => ["1 expected but was 2", "10 expected but was 20"]
```

But, it's not obvious what you are doing until you read the entire line. You could build the array the same way using inject and if you are comfortable with inject it reads slightly cleaner.

```
TestResult = Struct.new(:status, :message)
results = [
  TestResult.new(:failed, "1 expected but was 2"),
  TestResult.new(:success),
  TestResult.new(:failed, "10 expected but was 20")
]

messages = results.inject({}) do |messages, test_result|
  messages << test_result.message if test_result.status == :failed
  messages
end
messages # => ["1 expected but was 2", "10 expected but was 20"]
```

I prefer to build what I want using inject instead of chaining methods of Enumerable and effectively building multiple objects on the way to what I need.

Example 4: Building a Hash (again)

Building from the Test Result example you might want to group all results by their status. The inject method lets you easily do this by starting with an empty hash and defaulting each key

value to an empty array, which is then appended to with each element that has the same status.

```

TestResult = Struct.new(:status, :message)
results = [
  TestResult.new(:failed, "1 expected but was 2"),
  TestResult.new(:sucess),
  TestResult.new(:failed, "10 expected but was 20")
]

grouped_results = results.inject({}) do |grouped, test_result|
  grouped[test_result.status] = [] if grouped[test_result.status].nil?
  grouped[test_result.status] << test_result
  grouped
end

grouped_results
# => {:failed => [
#   #<struct TestResult status=:failed, message="1 expected but was 2">,
#   #<struct TestResult status=:failed, message="10 expected but was 20">],
# :sucess => [ #<struct TestResult status=:sucess, message=nil> ]
# }

```

You might be sensing a theme here.

Example 5: Building a unknown result

Usually you know what kind of result you are looking for when you use inject, but it's also possible to use inject to build an unknown object.

Consider the following Recorder class that saves any messages you send it.

```

class Recorder
  instance_methods.each do |meth|
    undef_method meth unless meth =~ /^(__|inspect|to_str)/
  end

  def method_missing(sym, *args)
    messages << [sym, args]
    self
  end

  def messages
    @messages ||= []
  end
end

Recorder.new.will.record.anything.you.want
# => #<Recorder:0x28ed8 @messages=[[:will, []], [:record, []], [:anything, []],
[:you, []], [:want, []]]>

```

By simply defining the play_for method and using inject you can replay each message on the argument and get back anything depending on how the argument responds to the recorded

methods.

```
class Recorder
  def play_for(obj)
    messages.inject(obj) do |result, message|
      result.send message.first, *message.last
    end
  end
end

recorder = Recorder.new
recorder.methods.sort
recorder.play_for(String)

# >> ["<", "<=", "<=>", "==", "===", "=~", ">", ">=", "__id__", ...]
```


Conclusion

How do I know when I want to use inject? I like to use inject anytime I am building an object a piece at a time. In the case of summing, creating a hash, or an array I'm building a result by applying changes based on the elements of the enumerable. After I'm done applying changes for each element, I have the finished object I'm looking for. The same is true of the Recorder example, I'm sending the methods one at a time until I return the result of sending all the methods.

So next time you need to build an object based on elements of an enumerable, consider using inject.

Labels: [enumerable](#), [inject](#), [ruby](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by Jay Fields : 8:12 AM 
[general focus: [Ruby](#) - [Ruby on Rails](#) - [Ruby Rake](#)]

Comments:

I love inject too!

To simply convert alist to Hash, I use Hash.[] and Array#flatten.

```
Hash[*[:first_name, 'Shane'], [:last_name, 'Harvie']].flatten # =>
{:first_name=>"Shane", :last_name=>"Harvie"}
```

And I use (obj ||= []) << elt idiom.