Understanding Ruby's idiom: array.map(&:method)

10 Jan 2015

Ruby has some idioms that are used pretty commonly, but not very often understood. `array.map(&:method_name)` is one of them. We can see it being used everywhere to call a method on every `array` element, but why does this work? What's really happening under the hood?

In case you don't know Ruby's `map`

`map` is used to execute a block of code for each element of a given `Enumerable` object, like an `Array`. Here's an example:

```
class Foo
  def method_name
    puts "method called for #{object_id}"
  end
end

[Foo.new, Foo.new].map do |element|
  element.method_name
end

# => method called for 70339841711300
# => method called for 70339841711280
```

As we are just calling `method_name` for each element of the list, Ruby allows us to use this idiom:

```
[Foo.new, Foo.new].map(&:method_name)
```

What Ruby does when it sees `&`

The first thing that happens is that, whenever Ruby sees a `&` for a parameter, it wants this parameter to be a `Proc`. If this is not the case already, Ruby calls `#to_proc` on this object to convert it. Let's confirm this is true:

```
class MyClass
  def to_proc
    puts "trying to convert to a proc"
    Proc.new {}
  end
end
```

```
[].map(&MyClass.new)

# => trying to convert to a proc
```

> *If you don't know what a `Proc` is, you can consider it to be just like a `Lambda` or a `closure`. It's a piece of code that can be moved around and executed (by calling `call()` on it, for instance).*

As we passed a `MyClass` instance with `&` to `map`, it tried to call `to_proc` on it. This holds true for any method call, not just `map`.

Back to the previous example, we are calling `map` with `&:method_name`. So we know that Ruby will see that `&` and try to call `:method_name.to_proc`. The next step is to understand what `Symbol#to_proc` does.

Symbol's smart `to_proc` implementation

What `Symbol#to_proc` does is quite clever. It tries to calls a method with the same name (in our example, `method_name`) on the given object.

Maybe an example will make more sense:

```
:upcase.to_proc.call("string")
# => STRING
```

When we call `to_proc` on the `:upcase` symbol, it will return a `Proc` object that just call the `upcase` method for the given parameter ("string").

Implementing our own version

One of the approaches that I like to take to understand how something works is to create my own dumb implementation of it. After we understand all the building blocks that make this idiom work, this should not be that hard.

First, let's implement our own `map` method:

```
def my_map(enumerable, &block)
  result = []
  enumerable.each { |element| result << block.call(element) }
  result
end
```

We iterate over the `Enumerable` object and execute that given block. We know that `block` is going to be a `Proc`, because Ruby called `to_proc` on it, so we can just `call` it.
And this works.

```
my_map(["foo", "bar"], &:upcase)
# => ["FOO", "BAR"]
```

Now let’s implement our own `Symbol` functionality:

```
class MySymbol
  def initialize(method_name)
    @method_name = method_name
  end

  def to_proc
    Proc.new do |element|
      element.send(@method_name)
    end
  end
end
```

We know that we just need to implement the `to_proc` method that Ruby is going to call and make it return a `Proc` object.
As this is not really a `Symbol`, we will define the method to be called in the constructor. The method name is dynamic, so we need to use Ruby’s `send` to call it.
And this works.

```
my_map(["foo", "bar"], &MySymbol.new("upcase"))
# => ["FOO", "BAR"]
```

Summarizing

- Ruby instantiates a `MySymbol` object;
- Ruby checks that there is a `&` and calls `to_proc` on this object;
- `MySymbol#to_proc` returns a `Proc` object, that expects a parameter (`element`) and calls a method on it (`upcase`);
- `my_map` iterates over the received list (`['foo', 'bar']`) and calls the received `Proc` on each element, passing it as a parameter (`block.call(element)`);
- The `Proc` then executes `element.send("upcase")`, that is basically the same as `"foo".upcase`, and will return the expected result.

Written by

Share this post

Brian Storti

Follow @brianstorti

10 Comments **Brian Thomas Storti** 1 Login

Recommend 7 Tweet Share Sort by Newest

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Romenig Lima Damasio • 3 months ago

nice!

Reply • Share ›

Tomasz Drgas • 4 months ago

[].map(&MyClass.new) # why did it even do anything? Isn't .map suppose to call the (&MyClass.new) for each element of [] this array.. It is empty so I was thinking it will call (&MyClass.new) 'nil' times

Reply • Share ›

Words_Are_Wind • 8 months ago

Thanks for this. It helped a lot.

Reply • Share ›

Kanwardeep Singh Baweja • a year ago

Awesome explanation!

Reply • Share ›

Panayotis Matsinopoulos • a year ago

Can I call a method that takes arguments? "[...].each(&:method, arg1, arg2)" ?

2 • Reply • Share ›

Julien → Panayotis Matsinopoulos • 7 months ago

Curious for this too.

Reply • Share ›

Hanaa Alshareef • 2 years ago

Thanks for your explanation

Reply • Share ›

Davide Ghezzi • 2 years ago

just WOW!

1 • Reply • Share ›

Andres Cuervo • 3 years ago

Thanks, this was such a nice explanation, I needed a refresher!

1 • Reply • Share ›

Alex • 3 years ago

Amazing and simple explanation, thank you!

Reply • Share ›