Metaprogramming in JS and Ruby: Javascript Proxy and Ruby Method Missing

Abram

June 3, 2021

Contents

1 MetaProgramming in JS and Ruby: Javascript 'Proxy' and Ruby 'method$_{\text{missing}}$'

1.1 Metaprogramming

- meta is "referring to itself" or "Self-referential"
  - thus meta-programming is programming referring to itself.
- Programming about Programming
- Treating programs as data to be processed
- Macros are an example, generate code for code with code.
- Code that makes code
- Code that acts like the language would
- Code about programming

1.1.1 To quote Mark

- "CODE IS DATA, DATA IS CODE" – Mark Bennet, 2021

1.2 Goal for today

- Allow objects in Javascript and Ruby to handle complex messages/method calls to them that might not be defined.
- Learn about Proxy in Javascript
- Learn about method$_{\text{missing}}$ in Ruby

1.3 Concepts

Javascript and Ruby are both OO dynamically typed languages. They have many similarities and differences.

- Object: an object has attributes or properties, it can handle method calls or messages sent to it. It often inherits methods from a class. Objects know themselves and have identity.
  - Objects represent things and individuals.

  - Objects often have a class.

- Class: the structure (methods/messages and attributes/properties) that acts as factory for objects.
  - Classes create objects.
- Instance: an object made from a class.
- Attribute/Property/Instance Variables: a value or variable associated with an object, encapsulated by the object.
- Method: code that is bound to an object that runs in the context of the object, accessing its attributes/properties.
- Method Call / Message: a function called within the context of an object, that has access to the object and its attributes.
- Delegate: an object you delegate to, an object your object has, and defers operations to (methods or messages or methods to).
- to proxy/Decorate/Wrap/Adapt: programmer slang for receiving messages, doing a little more work, and then passing off reponsibility to another object or function or method.
  - before methods
  - after methods
  - filters
  - Decorators, Adapters, Wrappers are the objects/classes who do this stuff.
- Reflection: allowing an object system to ask questions about its objects.
  - What class are you representing?
  - Do you have a constructor?
  - Can I get a new instance?
  - What properties/attributes do you have.

1.4 Proxy in Javascript JS

Javascript allows some metaprogramming through Reflect and Proxy. Proxy allows intercepting and wrapping of method calls.

Good documentation on Proxy is found here: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`

The actual specification of Proxy is found here: `https://tc39.es/ecma262/#sec-proxy-objects`

Many things you want to do with proxy will require reflection. The Reflect object is described in the spec here: `https://tc39.es/ecma262/#sec-reflect-object`

The Proxy wraps or decorates another object and intercepts calls to it. You can intercept property accesses, you can intercept property gets, set, deletions, definitions. You can even intercept playing with prototypes. You can override 'has'.

Came late because Javascript was already prototype driven.

1.4.1 The proxy handler is poorly documented. JS

I found most documentation within the spec itself and it was put in different sections and quite confusing.

Handler is an object with methods that the proxy will call. You can use an kind of object to be the handler. If the handler is undefined, the default action will happen.

`https://tc39.es/ecma262/#sec-proxy-object-internal-methods-and-internal-slots`

- get
- set
- has
- deleteProperty
- defineProperty
- getOwnPropertyDescriptor
- preventExtensions
- isExtensible
- getPrototypeOf

- setPrototypeOf
- ownKeys
- apply
- construct

1.4.2 Example intercept 'get' JS

We're going to intercept the access of properties in Javascript.

First we need a function to handle the access 'handlerGet'. Then we need to make a structure (called the Handler) that is an object with a method that addresses the operation we are wrapping. In this case the operation is 'get'.

```
function handlerGet(target, prop, receiver) {
  if (prop in target) {
    return target[prop];
  } else {
    return `Prop: ${prop} not found`;
  }
}
function makeHandler() {
        return {
                get: handlerGet
        }
}
var obj = {};
var po = new Proxy(obj,makeHandler());
po.fruit = "durian";
console.log("po.fruit:\t",po.fruit);
console.log("po.salad:\t",po.salad);
console.log("obj.salad\t",obj.salad);

po.fruit:  durian
po.salad:  Prop: salad not found
obj.salad  undefined
undefined
```

Let's be honest, why are you returning a string on a miss?

1.4.3 Default Return Value JS

```
function makeHandler( defaultValue = false ) {
  return {
    get: function(target, prop, receiver) {
      if (prop in target) {
        return target[prop]; //Reflect.get(...arguments);
      } else {
        return defaultValue;
      }
    }
  }
}
var obj = {};
var po = new Proxy(obj,makeHandler());
po.fruit = "durian";
console.log("po.fruit:\t", po.fruit);
console.log("po.salad:\t", po.salad);
console.log("obj.salad:\t",obj.salad);

po.fruit:  durian
po.salad:  false
obj.salad:  undefined
undefined
```

1.4.4 Proxy Summary

You can use Proxy in Javascript to decorate other objects in order to intercept calls.

You NEED to wrap those objects with Proxy for it to work.

1.5 Ruby Method Missing RUBY

Ruby's method is to allow any object to respond to messages to itself via $\text{method}_{\text{missing}}$.

When you call a method, if the method does not exist in the current object's class or it's superclasses then '$\text{method}_{\text{missing}}$' is called and handled by the lowest level class to handle it.

The block, the method arguments and the method's name ('$\text{method}_{\text{name}}$') (as a symbol) will be passed to the method $\text{method}_{\text{missing}}$.

This is not a decorator, this is a method that is called on the object when is there is no method to be called for ‘method$_{\text{name}}$‘

```
def method_missing(method_name, *method_arguments, &block)
  nil
end
```

1.5.1 Example of method missing responding to missing methods

RUBY

```
class Example
  def initialize()
  end
  def method_missing(m, *args, &block)
    @last_call = [:method_missing, m, *args, block]
  end
end
example = Example.new()
lc1 = example.eat("Pie")
lc2 = example.block(:Arg1) { |x|  x }
{:lc1=>lc1,:lc2=>lc2}
```

```
{:lc1=>[:method_missing, :eat, "Pie", nil], :lc2=>[:method_missing, :block, :Arg1, #<P
```

1.5.2 Delegation is easy

RUBY

```
Object.send(:remove_const,:Dog) # Make sure there's no Dog class
class Dog
  def initialize()
  end
  def speak()
    "Ruff"
  end
  def legs()
    4
  end
end

:legs

Object.send(:remove_const,:Cat) # Make sure there's no Cat class
```

```
class Cat
  def initialize()
    @delegate = Dog.new()
  end
  def method_missing(m, *args, &block)
    @delegate.send(m, *args, &block)
  end
end

:method_missing

cat = Cat.new()
cat.speak()

Ruff

class Cat
  def speak
     "meow"
  end
end
cat = Cat.new()
[cat.speak(), cat.legs()]

["meow", 4]

cat = Cat.new()
cat.consume_the_sun()

#<NoMethodError: undefined method `consume_the_sun' for #<Dog:0x000055e466a78d20>>
```

1.6 Examples

1.6.1 Autovivify JS

Autovivify means to auto-enliven something (this is from perl). So make something exist by checking in on it.

In perl if you access chains of objects it can create objects along the way.

1. Autovivify JS

```
function makeHandler( defaultValue = false ) {
  return {
    get: function(target, prop, receiver) {
      if (!(prop in target)) {
        target[prop] = defaultValue;
      }
      return target[prop]; //Reflect.get(...arguments);

    }
  }
}
obj = {};
po = new Proxy(obj,makeHandler());
po.fruit = "durian";
console.log("Fruit:\t",po.fruit);
console.log("Salad:\t",po.salad);
console.log("obj.salad\t",obj.salad);

Fruit:  durian
Salad:  false
obj.salad  false
undefined
```

2. Autovivify RUBY

Ruby's instance variables are not available unless exposed with getters or setters.

```
class Viva
  def initialize(default)
    @default = default
  end
  def write_to_instance(m,*args)
    instance_name = ("@"+m.to_s)[0..-2] # get rid of =
    instance_variable_set(instance_name, args[0])
  end
  def read_from_instance(m)
    instance_name = "@"+m.to_s
    if not instance_variable_defined?(instance_name)
      instance_variable_set(instance_name, @default)
```

```
      end
      return instance_variable_get(instance_name)
    end
    def method_missing(m, *args, &block)
      ms = m.to_s
      if ms[-1] == "="
        self.write_to_instance(m,*args)
      else
        self.read_from_instance(m)
      end
    end
  end
  v = Viva.new(:undefined)
  # this actually calls v.c=("x")
  v.c = "x"
  [v.a, v.b, v.c, v.inspect]

  [:undefined, :undefined, "x", "#<Viva:0x000055e466a1e938 @default=:undefined, @c="
```

1.6.2 Autovivify Perl Style

In perl if you access chains of objects it can create objects along the way.
‘$a = {}; $a->{b}->{c}‘ would cause ‘$a == {"b"=>{}}‘ to be made.

```
use Data::Dumper;
my $a = {};
$a->{b}->{c};
$abefore = Dumper($a);
$a->{b}->{c} = 1;
$aafter = Dumper($a);
[$abefore,$aafter]

$VAR1 = {
          'b' => {}
        };

$VAR1 = {
          'b' => {
                    'c' => 1
                  }
```

```
};
```

1. Autovivify Perl Style with Javascript JS

```
function makeVivifyHandler() {
  return {
    get: function(target, prop, receiver) {
      if (!(prop in target)) {
        target[prop] = new Proxy({},makeVivifyHandler());
      }
      return target[prop]; //Reflect.get(...arguments);
    }
  }
}
var obj = {};
var po = new Proxy(obj,makeVivifyHandler());
po.fruit = "durian";
console.log("Fruit:\t",po.fruit);
console.log("Salad:\t",po.salad);
console.log("obj.salad\t",obj.salad);
console.log("obj\t",obj);
console.log("obj.salad.what\t",obj.salad.what);
console.log("obj\t",obj);
console.log("po.salad.what\t",po.salad.what);
console.log("obj\t",obj);
console.log("po.salad.what.huh.zuh\t", po.salad.what.huh.zuh);
console.log("obj\t",obj);
console.log("salad" in po);
console.log("dessert" in po);

Fruit:  durian
Salad:  {}
obj.salad  {}
obj  { fruit: 'durian', salad: {} }
obj.salad.what  {}
obj  { fruit: 'durian', salad: { what: {} } }
po.salad.what  {}
obj  { fruit: 'durian', salad: { what: {} } }
po.salad.what.huh.zuh  {}
```

```
obj  { fruit: 'durian', salad: { what: { huh: [Object] } } }
true
false
undefined
```

2. Autovivify Perl Style with Ruby RUBY

I copied and pasted Viva, changed its name to AutoViva, removed its default parameter and just had it create itself instead.

```
class AutoViva
  def initialize()
  end
  def write_to_instance(m,*args)
    instance_name = ("@"+m.to_s)[0..-2] # get rid of =
    instance_variable_set(instance_name, args[0])
  end
  def read_from_instance(m)
    instance_name = "@"+m.to_s
    if not instance_variable_defined?(instance_name)
      instance_variable_set(instance_name, AutoViva.new())
    end
    return instance_variable_get(instance_name)
  end
  def method_missing(m, *args, &block)
    ms = m.to_s
    if ms[-1] == "="
      self.write_to_instance(m,*args)
    else
      self.read_from_instance(m)
    end
  end
end
v = AutoViva.new()
# this actually calls v.c=("x")
v.c = "x"
v.a.b.c.d
v.a.b.c.e = :liskovviolation
v2 = AutoViva.new()
v2.a = :instance
[v2.inspect, v.inspect]
```

```
["#<AutoViva:0x000055e466a73d20 @a=:instance>", "#<AutoViva:0x000055e466a7fa30 @c=
```

1.6.3 Multimethods

Multimethods inspect the arguments and delegate based on them.

1. Multimethods Javascript JS

```
class Example {
    a$number$number(x,y) {
        return x + y
    }
    a$string$number(s,n) {
        return [...Array(n)].map(() => s).join("")
    }
}
function makeSimpleMultiMethod(prefix) {
    return function() {
                let sig = prefix + "$" + [...arguments].map(x => typeof(x)).join('
        if (sig in this) {
            return this[sig](...arguments);
        } else {
            throw (sig + ' not in ' + this)
        }
    }
}
function handlerGet(target, prop, receiver) {
    props = prop.toString();
    if (props in target) {
        return Reflect.get(target,prop,receiver);
    }
    target[props] = makeSimpleMultiMethod(prop);
    return Reflect.get(target, prop, receiver);
}
function makeHandler() {
    return {
        get: handlerGet
    }
}
// We can wrap it with proxy
```

```
pe = new Proxy(new Example(),makeHandler());
console.log( "pe.a(1,2)    "     ,       pe.a(1,2)    );
console.log( "pe.a(\"x\",5)  "   ,       pe.a("x",5)  );
try {
    console.log( "pe.a(\"x\",\"x\")" ,       pe.a("x","x"));
} catch (err) {
    console.log("Error "+err);
}

pe.a(1,2)     3
pe.a("x",5)   xxxxx
Error a$string$string not in [object Object]
undefined
```

2. But we don't actually need proxy to do this!

   Since we can make a multimethod on the fly, we can also just make multimethods intentionally.

```
class Example {
    a$number$number(x,y) {
        return x + y
    }
    a$string$number(s,n) {
        return [...Array(n)].map(() => s).join("")
    }
}
function makeSimpleMultiMethod(prefix) {
    return function() {
                let sig = prefix + "$" + [...arguments].map(x => typeof(x)).join('
        if (sig in this) {
            return this[sig](...arguments);
        } else {
            throw (sig + ' not in ' + this)
        }
    }
}

// We can forget proxy and just make the method a multi method by default
Example.prototype.a = makeSimpleMultiMethod("a");
```

```
e = new Example();
console.log( "e.a(1,2)    "     ,       e.a(1,2)    );
console.log( "e.a(\"x\",5)  "   ,       e.a("x",5)  );
try {
    console.log( "e.a(\"x\",\"x\")" ,       e.a("x","x"));
} catch (err) {
    console.log("Error "+err);
}

e.a(1,2)     3
e.a("x",5)   xxxxx
Error a$string$string not in [object Object]
undefined
```

3. Multimethods Ruby RUBY

   For ruby we're going to make a utility method that inspects an object and suggests a method to call instead.

```
def resolve_simple_multi_method(obj,method,args)
  sig = (method.to_s + "__" + args.map {|x| x.class.to_s }.join("__")).to_sym
  if obj.methods.include? sig
    return sig
  else
    raise (sig.to_s + " not found")
  end
end

class MMExample
  def a__Integer__Integer(x,y)
    x + y
  end
  def a__String__Integer(s,n)
    s * n
  end
  def method_missing(method_name, *method_arguments, &block)
    new_method = resolve_simple_multi_method(self, method_name, method_arguments)
    self.send( new_method, *method_arguments, &block)
```

```
  end
end

mme = MMExample.new
[mme.a(1,2), mme.a("x",7)]

[3, "xxxxxxx"]
```

What about something now yet defined?

```
mme.a("x","x")

#<RuntimeError: a__String__String not found>
```

And being ruby we can open it for extension any time.

```
class MMExample
  def a__Integer__Integer__Integer(x,y,z)
    x ** y ** z
  end
end

mme.a(2,2,5)

4294967296
```

Of course there are many multidispatch and multimethod gems for ruby. It's just good to understand how they would work

1.7 Dangers of meta-programming

- It's confusing
- You can't grep for function definitions
- It surprises people
- You need a running system to test semantics :(
- Often you need to use a debugger or debugger tools to figure out what is being called.
  - You can always throw an exception and/or force a backtrace.
    - ‘console.trace()‘ :js:
    - ‘exception.backtrace‘ :rb:

1.8 Conclusion

- Reflect and Proxy are your buddies for Metaprogramming in Javascript
- Method Missing, and Object are your buddies for Metaprogramming in Ruby
- Reflection and message passing all for powerful constructs that will confuse just about anyone reading your code.

1.9 Copyright Statement

1.10 Init ORG-MODE

I use this section to ensure I can run the examples. You might not need this, but I eval the following elisp before I start the presentation

```
;; I need this for org-mode to work well
;; If we have a new org-mode use ob-shell
;; otherwise use ob-sh --- but not both!
;;(require 'ob-ruby)
;;(require 'inf-ruby)
;;(require 'enh-ruby-mode)

(if (require 'ob-shell nil 'noerror)
  (progn
    (org-babel-do-load-languages 'org-babel-load-languages '((shell . t))))
  (progn
    (require 'ob-sh)
    (org-babel-do-load-languages 'org-babel-load-languages '((sh . t)))))
(org-babel-do-load-languages 'org-babel-load-languages '((C . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((ruby . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((js . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((perl . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((python . t)))
(setq org-babel-js-function-wrapper
      "process.stdout.write(require('util').inspect(function(){\n%s\n}(), { maxArrayLe
(setq org-src-fontify-natively t)
```

```
(setq org-confirm-babel-evaluate nil) ;; danger!
(custom-set-faces
 '(org-block ((t (:inherit shadow :foreground "black"))))
 '(org-code ((t (:inherit shadow :foreground "black")))))
;(setq inf-ruby-default-implementation "ruby")
```

1.10.1 Org export

```
(org-html-export-to-html)
(org-latex-export-to-pdf)
(org-ascii-export-to-ascii)
```

1.10.2 Org Template

Copy and paste this to demo C

```
#include <stdio.h>

int main(int argc, char**argv) {
    return 0;
}
```

```
var util = require("util");
console.log("this is a Node.js test program!");
console.log("this is a Node.js test program! again");
```

```
this is a Node.js test program!
this is a Node.js test program! again
undefined
```

```
[12,"ruby"]
```

```
[12, "ruby"]
```

```
class XYZ
end
XYZ.new()
```

```
#<XYZ:0x000055e466a41aa0>
```

```
use Data::Dumper;
$a = {};
$a->{b}->{c};
Dumper($a);
```

```
$VAR1 = {
          'b' => {}
        };
```