# Метапрограммирование в Ruby

Метапрограммирование — это парадигма построения кода информационной системы с динамическим изменением поведения или структуры в зависимости от данных, действий пользователя или взаимодействия с другими системами. Задачи метапрограммирования: повышение абстракции кода и его гибкости, повторное использование, ускорение разработки, упрощение межсистемной интеграции.

Ruby содержит множество методов для динамического создания кода. Ознакомимся с ними:

### Получение, установка и удаление переменных

* Object#instance\_variable\_get
* Object#instance\_variable\_set
* Object#remove\_instance\_variable
* Module#class\_variable\_get
* Module#class\_variable\_set
* Module#remove\_class\_variable

### Получение, установка и удаление констант (и классов)

* Module#const\_get
* Module#const\_set
* Module#remove\_const

### Объявление и удаление методов

* Module#define\_method
* Module#remove\_method

### Выполнение кода, построенного «на лету»

* Object#send
* Object#instance\_eval
* Module#module\_eval (синоним Module#class\_eval)
* Kernel#eval
* Kernel#method\_missing

### Методы интроспекции (рефлексии, отражения)

Интроспекция — важная часть метапрограммирования, позволяющая изучать объекты, получая их структуру и данные

* Object#class
* Object#instance\_variables
* Object#methods
* Object#private\_methods
* Object#public\_methods
* Object#singleton\_methods
* Module#class\_variables
* Module#constants
* Module#included\_modules
* Module#instance\_methods
* Module#name
* Module#private\_instance\_methods
* Module#protected\_instance\_methods
* Module#public\_instance\_methods

## Выполнение кода в строках и блоках

Вы могли слышать о методе eval, позволяющем выполнять строку или блок как код на Ruby. Когда нужно выполнить код внутри контекста какого-то объекта, можно использовать методы instance\_eval и module\_eval (синоним class\_eval).

Метод instance\_eval выполняет код в контексте существующего объекта.

[1, 2, 3, 4].instance\_eval('size') # вернёт 4

В примере мы передали методу instance\_eval строку «size», что интерпретировалось как получение метода «size» массивом. Это эквивалентно следующему вызову:

[1, 2, 3, 4].size

Кроме того, вы можете передавать методу instance\_eval блок.

# Получаем среднее арифметическое чисел в массиве

[1, 2, 3, 4].instance\_eval { inject(:+) / size.to\_f } # вернёт 2.5

Методы inject(:+)и size.to\_f вроде бы «висят в воздухе» и не относятся ни к какому объекту, однако они выполняются в блоке, в контексте объекта, поэтому они интерпретируются как self.inject(:+) / self.size.to\_f где self — наш массив.

В то время как instance\_eval выполняет код в контексте объекта, метод module\_eval выполняет код в контексте модуля или класса.

Fixnum.module\_eval do

def to\_word

if (0..3).include? self

['ничего', 'один', 'пара', 'несколько'][self]

elsif self > 3

'много'

elsif self < 0

'отрицательно'

end

end

end

1.to\_word # вернёт 'один'

2.to\_word # вернёт 'пара'

Мы видим, как module\_eval заново открывает существующий класс Fixnum и добавляет в него новый метод. Само по себе это не очень интересно, и мы можем сделать это иначе:

class Fixnum

def to\_word

..

end

end

Лучшее применение этому методу мы найдём при динамическом генерировании кода. Добавим метод create\_multiplier, который позволит динамически генерировать методы-умножители с теми именами, которые вы захотите.

class Fixnum

def self.create\_multiplier(name, num)

module\_eval "def #{name}; self \* #{num}; end"

end

end

Fixnum.create\_multiplier('multiply\_by\_pi', Math::PI)

4.multiply\_by\_pi # вернёт 12.5663706143592

Пример выше создаёт метод класса (или «метод-синглетон») create\_multiplier, с помощью которого мы сможем создавать методы, применимые к любому объекту класса Fixnum.

## Использование send

Метод send работает почти как instance\_eval, поскольку вызывает метод объекта, переданный в качестве параметра. Он удобен, когда мы динамически получаем имя необходимого метода в виде строки или символа.

method\_name = 'size'

[1, 2, 3, 4].send(method\_name) # вернёт 4

Мы можем передать имя метода в виде строки 'size' или символа :size

Полезное свойство метода send — возможность вызывать приватные методы, как например метод Module#define\_method.

Array.define\_method(:ducky) { puts 'ducky' }

# NoMethodError: private method `define\_method' called for Array:Class

Используя send получаем:

Array.send(:define\_method, :ducky) { puts 'ducky' }

## Создание методов

Как мы видим в примере выше, мы можем создавать методы классов с помощью define\_method.

class Array

define\_method(:multiply) do |arg|

collect{|i| i \* arg}

end

end

[1, 2, 3, 4].multiply(16) # returns [16, 32, 48, 64]

## method\_missing

Метод method\_missing, объявленный в классе, вызывается в случае получения экземпляром класса несуществующего метода. Это может быть использовано для выполнения этих отсутвующих методов, вместо отправки ошибки NoMethodError.

class Fixnum

def method\_missing(meth)

method\_name = meth.id2name

if method\_name =~ /^multiply\_by\_(\d+)$/

self \* $1.to\_i

else

raise NoMethodError, "undefined method `#{method\_name}' for #{self}:#{self.class}"

end

end

end

16.multiply\_by\_64 # вернёт 1024

16.multiply\_by\_x # ошибка NoMethodError

## Как работает метод attr\_accessor?

Многие из нас используют attr\_accessor в классах, однако не все понимают, какую работу он за нас выполняет. attr\_accessor динамически генерирует getter и setter для переменной экземпляра. Давайте посмотрим на примере.

class Person

attr\_accessor :first\_name

end

john = Person.new

john.first\_name = 'John'

john.instance\_variables # вернёт ["@first\_name"]

john.methods.grep /first\_name/ # вернёт ["first\_name", "first\_name="]

Также мы видим, что attr\_accessor создаёт переменную экземпляра @first\_name и два метода: getter first\_nameи setter first\_name=

### Реализация

Все классы наследуют методы Module, поэтому мы поместим методы сюда.

class Module

# First using define\_method

def attr1(symbol)

instance\_var = ('@' + symbol.to\_s)

define\_method(symbol) { instance\_variable\_get(instance\_var) }

define\_method(symbol.to\_s + "=") { |val| instance\_variable\_set(instance\_var, val) }

end

# Second using module\_eval

def attr2(symbol)

module\_eval "def #{symbol}; @#{symbol}; end"

module\_eval "def #{symbol}=(val); @#{symbol} = val; end"

end

end

class Person

attr1 :name

attr2 :phone

end

person = Person.new

person.name = 'John Smith'

person.phone = '555-2344'

person # returns

define\_method и module\_eval оба вернут один и тот же результат.