

75.31 Teoría de Language

El lenguaje de programación Ruby

28 de mayo de 2012

Diaz, Federico (83568)
Rossi, Federico Martín (92086)

1. Introducción: *Enfoque de la presentación*

La presentación, como así también el presente informe, tienen como fin mostrar a los oyentes este veterano, y a la vez moderno lenguaje de programación desde un punto de vista diferente al que generalmente se acostumbra. Es nuestro objetivo el dar a conocer las fortalezas y debilidades de éste fijando la perspectiva de la presentación en las aplicaciones cotidianas del lenguaje en el desarrollo de proyectos profesionales. Es decir, sin quitar importancia a tópicos importantes como la sintaxis, nos focalizaremos en detallar aquellas características que hacen de Ruby un lenguaje interesante y ávido de nuevos conceptos.

2. Un poco de historia

Ruby fue creado por Yukihiro “Matz” Matsumoto. Matz comenzó a trabajar en Ruby en 1993, cuando tenía 28 años, mientras trabajaba para una empresa Open source (netlab.jp). Ya por ese entonces era muy conocido en Japón por tener un alto perfil de evangelista dentro de la comunidad Open source y por trabajar en varios productos Open source. Ruby es el primer software japonés altamente conocido fuera de Japón, el cual fue presentado en 1995.

“En 1993, yo estaba hablando con un colega acerca de lenguajes de scripting. Estaba muy impresionado por su poder y sus posibilidades. Sentía que el scripting era el camino a seguir.

Me pareció que la programación orientada a objetos (POO) era muy adecuada para secuencias de comandos también (como viejo fan de la POO). Luego miré alrededor de la red. Me encontré con que Perl 5, que no se había lanzado todavía, implementaría las características de OO, pero finalmente no resultó lo que esperaba. Me di por vencido en creer en Perl como lenguaje de scripts orientado a objetos.

Entonces me encontré con Python. Se trataba de un interprete y un lenguaje orientado a objetos. Pero no lo sentía como si fuera un "script" del lenguaje. Además, se trataba de un lenguaje híbrido de la programación de procedural y la programación orientada a objetos.

Yo quería un lenguaje que fuera más poderoso que Perl, y más orientado a objetos que Python. Es por eso que me decidí a diseñar mi propio Lenguaje de programación”

Yukihiro Matsumoto (a.k.a. “Matz”)

Es así que comenzó a desarrollarlo el 24 de febrero de 1993 y en agosto desarrolló el primer Hello World con ruby. En 1994 se lanzó la primera versión alpha. Matz trabajó solo hasta el año 1996, momento en el cual se formó la primera comunidad de Ruby.

3. ¿Qué es Ruby?

Es un lenguaje escrito en C y fue diseñado teniendo en mente las capacidades de perl y python. [Colocar contenido aquí]

4. El lenguaje y sus Principios

Antes de iniciarnos en la sintaxis del lenguaje creemos necesario entender el por qué es que fue creado Ruby, es decir, entender el impacto que su creador deseaba que tuviera y las razones por las cuales conforma una forma muy particular de realizar aplicaciones.

El lenguaje de programación Ruby es un “*lenguaje de programación interpretado para una rápida y fácil programación orientada a objetos*” - Pero, ¿Qué significa esto? Veámos:

Lenguaje de programación interpretado:

- habilidad para hacer llamadas directas al sistema operativo
- poderoso manejo y operaciones con cadenas y expresiones regulares
- respuestas inmediatas durante el desarrollo

Rápido y fácil:

- innecesarias las declaraciones de variables
- variables no tipadas
- syntaxis simple y consistente
- el manejo de memoria es automático

Programación orientada a objetos:

- todo es un objeto (como SmallTalk)
- clases, métodos, herencia, etc.
- métodos singleton (métodos que pertenecen a un sólo objeto)
- funcionalidad “mixin” por módulo
- iteradores y clausuras

Además:

- números enteros de múltiple precisión
- procesamiento de excepciones
- carga dinámica
- soporte de concurrencia

Sin duda, Ruby fue diseñado para hacer a los programadores mas productivos y felices. Esto es lo que el creador de Ruby nos dice:

“Para mí, el propósito de la vida es, al menos en parte, tener alegría. Los programadores a menudo sienten alegría cuando se pueden concentrar en el aspecto creativo de la programación, por lo que Ruby está diseñado para que los programadores sean felices. Considero a un lenguaje de programación como una interfaz de usuario, por lo que deben seguir los principios de la interfaz de usuario.”

Yukihiro Matsumoto (a.k.a. “Matz”), 2000

¿Cuáles son los principios de una buena interfaz de usuario? Estos son los tres principios con citas de apoyo por parte de Matz:

Principio de concisión: *“Quiero que las computadoras sean mis siervos, no mis maestros. Por lo tanto, me gustaría darles órdenes rápidamente. Un buen siervo debe hacer un montón de trabajo con una breve orden.”*

Principio de consistencia: *“... un pequeño conjunto de normas cubre la totalidad del lenguaje Ruby. Ruby es un lenguaje relativamente sencillo, pero no es demasiado simple. He tratado de seguir el principio de sin sorpresas. Ruby no es demasiado único, por lo que un programador con conocimientos básicos de otros lenguajes de programación puede aprender muy rápidamente.”*

Principio de flexibilidad: *“Porque las lenguas son para expresar el pensamiento, una lengua no debe restringir el pensamiento humano, sino que debe ayudarlo. Ruby consiste en un núcleo pequeño inmutable (es decir, syntaxis) y bibliotecas arbitrarias de clases extensibles...”*

5. Sintaxis

Ha llegado el momento de conocer la sintaxis de Ruby. El lector será capaz de notar una simpleza extrema de esta, lo que no significa una deficiencia en su potencial. Es decir, veremos que a partir de un número muy acotado de líneas de código seremos capaces de realizar programas realmente interesantes.

5.1. Tipos

Todos los tipos de ruby heredan de la clase Object, es decir que todos los tipos en ruby son objetos. Existen clases ya construidas que representan los tipos básicos de ruby sobre los cuales se construyen los bloques de todos los programas que se desarrollan en ruby.

5.2. Tipos numéricos

Los números en ruby son representados por las clases: Fixnum (entero $-2^{30}..2^{30}$) y Bignum. Cuando se crea un objeto numérico se asigna automáticamente al tipo Fixnum, si excede el rango será un Bignum. Los números enteros son creados ingresando el número sin coma. El formato de representación particular depende de la base del sistema numérico que se utilice. Ruby soporta las bases numéricas 10, 8, 16 y 2.

Código 1: "Tipos Numéricos"

```
1 #Numeros
2 123456789 # -123456789 Fixnum
3 0d123456789 # 1234567890 Fixnum
4 1234323424231 # 1234323424231 Bignum
5 0x5C1 # 1473 Hex
6 01411 # 777 Octal
7 0b10 # 2 binario
8 1_90_33 # 19033 Fixnum, se omite el caracter _
9 1.5 # 1.5 Float
10 1.0e5 # 100000.0 Float
11
12 #Algunos Metodos
13 -14.abs # 14
14 6.zero? # false
```

En el *Código ??*, se ven los tipos numéricos básicos que encontramos en cualquier lenguaje. Debajo de estos se encuentran sus particulares al ser objetos y no ser simplemente "tipos nativos". Existen otras representaciones numéricas como los Números imaginarios (Complex). En las secciones siguientes se verá el detalle del resto de los tipos básicos de ruby: Strings y Collections.

5.3. Strings

Los strings son simples secuencias de bytes que representan una secuencia de caracteres. A continuación se muestran ejemplos de código sobre sus cualidades básicas. En la siguiente sección se muestra el poder de las expresiones regulares.

Código 2: Representacion de strings"

```
1 #Representacion en comillas simples o dobles
2 "abc"
3 'abc'
4
5 #Caracteres de escape
6 "nueva\nlinea"
7 "\tnueva parrafo"
```

Código 3: Representación de strings"

```
1 #Expresion de evaluacion
2 apellido="Matsumoto"
3 nombre="Japones"
4 "#{apellido} #{nombre}"
```

```

5
6 i=0
7 "add one: #{i+=1}"

```

Código 4: Representacion de strings"

```

1 #Concatenacion y repeticion
2 "Matsumoto" + nombre
3 apellido * 2
4
5 #Extraccion de caracteres
6 wordF00 = "foo"
7 wordF00[-1] #muestra 111 (el codigo ASCII de "o")

```

5.4. Expresiones regulares

Otro punto importante integrado en el lenguaje y siguiendo los buenos resultados que se obtiene en PERL, son las expresiones regulares. Estas permiten comprobar si una cadena satisface un patrón.

(Tabla de Caracteres especiales en expresiones regulares)

Carácter	Descripción
[]	Especificación de rango. (p.e. [a-z] representa una letra en el rango de la a a la z)
\w	Letra o dígito; es lo mismo que [0-9A-Za-z]
\W	Ni letra, ni dígito.
\s	Espacio, es lo mismo que [t n r f]
\S	No espacio
\d	Dígito; es lo mismo que [0-9]
\D	No dígito
\b	Backspace (0x08) (sólo si aparece en una especificación de rango)
\B	Límite de palabra (sólo si no aparece en una especificación de rango)
	No límite de palabra
*	Cero o más repeticiones de lo que precede
+	Una o más repeticiones de lo que precede
[m,n]	Al menos m y como máximo n de lo que precede
?	Al menos una repetición de lo que precede; es lo mismo que [0,1]
	Puede coincidir con lo que precede o con lo que sigue
()	Agrupamiento

Código 5: .Expresiones regulares"

```

1 #Expresiones regulares
2 def chab(s) # contiene la cadena un hexadecimal (entre angulos "<>")
3   (s =~ /<[Xx][\dA-Fa-f]+>/) != nil
4 end

```

5.5. Colecciones - Rangos

La colección más primitiva de todas es el rango que permite representar una colección secuencial de valores (números o letras secuenciales) Para indicar rangos se pueden utilizar los dos puntos (incluye al primer elemento y al último elemento inclusive) o tres puntos (excluye al último elemento del rango).

Código 6: Rangos"

```

1 #rangos
2 edadNinos = 4..12
3 estrellasHotel = 1...6
4 edadNinos.include?(12) #true
5 estrellasHotel.include?(6) #false

```

5.6. Colecciones - Arrays

A continuación se muestran ejemplos sobre el uso de arrays. Más adelante, cuando se vean los conceptos de bloques se verán los iteradores sobre esta y el resto de las colecciones.

Código 7: *Arrays*

```
1 #Los array pueden almacenar elementos de distintos tipos
2 ary = [1, 2, "3"]
3
4 #Al igual que las cadenas...
5 ary + ["foo", "bar"] #=> [1, 2, "3", "foo", "bar"]
6 ary * 2 #=> [1, 2, "3", 1, 2, "3"]
7 ary[-2] #=>
8
9 #Uso de indices
10 ary[0,2] #=> [1, 2]
11 ary[-2,2] #=> [2, "3"]
12 ary[-2..-1] #=> [2, "3"]
13
14 a = [ "a", "b", "c", "d" ]
15 a.map! {|x| x + "!" }
16 a #=> [ "a!", "b!", "c!", "d!" ]
17
18 a = [1, 2, 3, 4, 5, 0]
19 a.drop_while {|i| i < 3 } #=> [3, 4, 5, 0]
20
21 a = [1, 2, 3]
22 a.permutation.to_a #=> [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

5.7. Colecciones - Hash

Conocidos también como diccionarios (aunque en el mundo ruby se prefiere el nombre hash), a continuación se exponen ejemplos de uso de los mismos.

Código 8: *Hash*

```
1 h = {1 => 2, "2" => "4"} #=> {"2"=>"4", 1=>2}
2 h[1] #=>
3 h[5] #=> nil
4 h[5] = 10 # anadir un valor
5 h[1]=nil # borrar un valor
6 h #=> {5=>10, "2"=>"4", 1=>nil}
7
8 h = { "a" => 100, "b" => 200, "c" => 300 }
9 h.delete_if {|key, value| key == "b" } #=> {"a"=>100}
```

5.8. Variables

En ruby las variables son referencias a valores en memoria. Cuando se asigna una variable a otra, no se duplica el objeto. Para duplicar objetos se puede utilizar los métodos clone o dup que son de Object. Las variables pueden tener alcance global, local (dadas por cualquier bloque: bloque de una estructura de control o bloque de un método) o de clase.

Código 9: *Variables*

```
1 #En ruby las variables son referencias
2 firstVar="Dogs"
3 secondVar = firstVar
4 firstVar.chop!
5 #ambas variablas terminan
6 #con una referencia al valor "Dog"
```

5.9. Metodos

Cuando un método es llamado en Ruby, técnicamente no se esta “llamando” un método. Lo que se esta haciendo es enviando un mensaje a un objeto, como diciendo: “Hey, vos tenes este método?” si lo tiene entonces se ejecuta, si no lo tiene se lanza una exception “NoMethodError”. Los métodos siempre devuelven un valor, el que corresponde a la última sentencia que ejecuta.

Código 10: "Declaración metodos"

```
1 #Definicion de un metodo
2 def sayHello(name)
3   puts "Hello #{name}!.."
4 end
5
6 #Parametros x defecto
7 def new_method(a = "This", b = "is", c = "fun")
8   puts a + ' ' + b + ' ' + c + ' .'
9 end
10 new_method('Rails') #Rails is fun
```

Código 11: "Parametros"

```
1 #Tamano variable en la lista de parametros
2 def print_relation(relation, *names)
3   puts "My #{relation} include: #{names.join(', ')}."
4 end
5 print_relation("cousins", "Morgan", "Miles", "Lindsey")
6
7 #Se pueden omitir los parametros
8 puts "Look ma! No parentheses!"
9
10 #Permite recursividad
11 def sum_integers(a, b)
12   return 0 if a > b
13   a + sum_integers((a + 1), b)
14 end
15
16 sum_integers(1, 10) #51 Hace la progresion de sumas en el rango dado
```

Código 12: "Alto orden utilizando metodos"

```
1 #Programacion de alto orden
2 # Implementacion del sig patron:
3 # def <name>(a, b)
4 #   return 0 if a > b
5 #   <term>(a) + <name>(<next>(a), b)
6 # end
7 def sum(term, a, the_next, b)
8   return 0 if a > b
9   term.call(a) + sum(term, the_next.call(a), the_next, b)
10 end
11
12 def termino(x)
13   x
14 end
15
16 def inc(x)
17   x+=1
18 end
19
20 term = self.method(:termino).to_proc
21 the_next= self.method(:inc).to_proc
22 sum(term,1,the_next,10)
```

5.10. Bloques y Func

Los bloques son un concepto muy poderoso e importante en Ruby. En ruby los bloques son objetos que contienen código y todo el contexto necesario para ejecutarse.

Código 13: "Bloques básicos"

```
1 #Bloques Basicos
2 myarray.each {|element| print "[" + element + "... " }
3 #Se esta pasando como parametro, el
4 #contenido del bloque al metodo each
5 #para que lo ejecute
6
7 myarray.each do |element|
8   print "[" + element + "... "
9 end
```

Un objeto procedimiento nuevo se obtiene utilizando Proc. Se pueden utilizar procedimientos anónimos. Estos objetos preservan el contexto en el cual fueron creados.

Código 14: "Proc"

```
1 #Procs
2 inthandler = proc{ print "^C ha sido pulsado.\n" }
3 trap "SIGINT", inthandler
4
5 addHandlerToSIGINT = proc{|handler| trap("SIGINT", handler) }
6 addHandlerToSIGINT.call(inthandler)
```

Código 15: "Proc anónimos"

```
1 #Proc anonimos
2 trap "SIGINT", proc{ print "^C ha sido pulsado.\n" }
3 trap "SIGINT", 'print "^C ha sido pulsado.\n"' #Forma mas compacta
```

Código 16: "Proc y Bloques"

```
1 #Procs y Bloques
2 def make_signal_handler(signal)
3   Proc.new{|handler| trap(signal, handler) }
4 end
5
6 SIGINTHandlerMaker = make_signal_handler("SIGINT")
7 SIGINTHandlerMaker.call(inthandler)
8
9 #Funciones que devuelven funciones
10 def make_filter(predicate)
11   lambda do |list|
12     new_list = []
13     list.each do |element|
14       new_list << element if predicate.call(element)
15     end
16     new_list
17   end
18 end
19
20 filter_odds = make_filter( lambda{|i| i % 2 != 0} )
21 filter_odds.call(list)
```

Código 17: "Func lambda"

```

1 #funciones lambda
2 lproc = lambda {|a,b| puts "#{a + b} <- the sum"}
3 nproc = Proc.new {|a,b| puts "#{a + b} <- the sum"}
4
5 nproc.call(1, 2, 3)
6 # 3
7 lproc.call(1, 2, 3)
8 # !ArgumentError (wrong number of arguments (3 for 2))

```

5.11. Estructura de control

Código 18: *estructuras de control*

```

1 #Case
2 i=8
3 case i
4   when 1, 2..5
5     print "1..5\n"
6   when 6..10
7     print "6..10\n"
8   end
9 #6..10
10
11 case 'abcdef'
12   when 'aaa', 'bbb'
13     print "aaa o bbb\n"
14   when /def/
15     print "incluye /def/\n"
16   end
17 #incluye /def/
18
19 #While
20 i = 0
21 print "Es cero.\n" if i == 0
22 print "Es negativo\n" if i < 0
23 print "#{i+=1}\n" while i < 3
24 #Es cero.
25 #1
26 #2
27 #3
28
29 #For
30 for elm in [100,-9.6,"pickle"]
31   print "#{elm}\t#{elm.type}\n"
32 end
33 #100 (Fixnum)
34 #-9.6 (Float)
35 #pickle (String)
36
37 # For estilo c o java
38 for i in coleccion
39   ..
40 end
41
42 # For estilo smalltalk
43 coleccion.each {|i|
44   ...
45 }

```

5.12. Convención de nombres

En Ruby se añade, por convención, ! o ? al final de ciertos nombre de métodos. La marca de exclamación (!, pronunciada como un “bang!” sonoro) recalca algo potencialmente peligroso, es decir, algo que puede modificar el valor de lo que toca. El método de la clase cadena chop! afecta directamente a la cadena pero chop sin el signo de exclamación actúa sobre una copia. Los nombres de métodos que finalizan con un signo de interrogación (?), pronunciada a veces como un “huh?” sonoro) indica que el método es un “predicado”, aquel que puede devolver o true o false.

5.13. Métodos Singleton

El comportamiento de una instancia es determinado por su clase, pero hay momentos en los que es necesario que una instancia en particular posea un comportamiento especial. En la mayoría de los lenguajes deberíamos adentrarnos en la problemática de definir una nueva clase, la cual será instanciada una sola vez. En cambio, Ruby permite dar a cualquier objeto sus propios métodos.

Código 19: "Métodos Singleton ejecutado en la consola"

```
1 ruby> class SingletonTest
2   |   def size
3   |     25
4   |   end
5   | end
6   nil
7 ruby> test1 = SingletonTest.new
8   #<SingletonTest:0xbc468>
9 ruby> test2 = SingletonTest.new
10  #<SingletonTest:0xbae20>
11 ruby> def test2.size
12   |   10
13   | end
14   nil
15 ruby> test1.size
16   25
17 ruby> test2.size
18   10
```

En el Código ??, las instancias *test1* y *test2* pertenecen a la misma clase, pero en *test2* se ha redefinido el método *size* por lo que se comportará de forma diferente. Un método que pertenece sólo a un único objeto se denomina *Método Singleton*.

6. Manejo de memoria

[Colocar contenido aquí]

7. Recolección de basura

[Colocar contenido aquí]

7.1. El porvenir de Ruby 2.0: *Bitmap Marking GC*

[Colocar contenido aquí]

8. *Ruby On Rails*: un punto fuerte del lenguaje

En la actualidad, Ruby se ha popularizado en el mundo del desarrollo de las aplicaciones webs a través del framework *Ruby On Rails*, o mas comúnmente conocido como *Rails*, escrito en este mismo lenguaje.

Ruby on Rails nace como un framework de desarrollo web especialmente diseñado con un fin en particular: hacer la vida más fácil a las personas que desarrollan aplicaciones destinadas a la web. Muchas de estas personas quiza se sientan frustradas con tecnologías como PHP, Java o .NET, que si bien son buenas, conllevan problemas de carga de recursos y una complejidad innecesaria. Ruby on Rails es simplemente más sencillo.

Rails utiliza el patrón MVC para poder administrar sus recursos. Si bien Java utiliza distintos frameworks también basados en MVC, Rails lleva el concepto mucho mas allá debido a que hay un lugar específico para cada parte del código, y cada componente de nuestra aplicación funciona de manera estándar. Es decir, es como si iniciáramos una aplicación con el esqueleto previamente armado.

Todas estas potenciales características hicieron que Ruby on Rails sea una de las razones por las que el lenguaje de programación Ruby haya logrado un importante impulso, siendo cada vez mas reconocido como una buena opción a la hora de elegir un lenguaje con el que llevar a cabo un proyecto.

9. Aplicaciones

Con el paso del tiempo Ruby fue utilizado por un número cada vez mayor de desarrolladores para llevar a cabo proyectos de grande envergadura. A continuación se muestra un listado de aplicaciones¹ realizadas en este lenguaje:

Web:

- Twitter (<http://www.twitter.com>)
- Shopify (<http://www.shopify.com>)
- Groupon (<http://www.groupon.com>)
- XING (<http://www.xing.com>)

Simulaciones:

- *NASA Langley Research Center* usa Ruby para llevar a cabo simulaciones. (<http://www.larc.nasa.gov>)
- Motorola (<http://www.motorola.com>)

Modelado 3D:

- Google SketchUp (<http://sketchup.google.com>)

Telecomunicaciones:

- *Open Domain Server*: permitir a los usuarios el uso de DNS dinámicos (<http://ods.org>)
- *Lucent*: uso en producto de telefonía wireless 3G (<http://www.lucent.com>)

10. Un ejemplo práctico de aplicacion

11. Conclusión

¹Véase una lista mas completa de aplicaciones en los siguientes vínculos: <http://rubyonrails.org/applications>
<http://www.ruby-lang.org/en/documentation/success-stories>