Appunti di programmazione in Ruby

Massimo Maria Ghisalberti - pragmas.org

01/11/2018

Versione in pdf

Questo documento è disponibile in formato pdf.
Questa presentazione ed il file pdf sono prodotti da un unico sorgente in org-mode.

Ruby

Il linguaggio di programmazione **Ruby** è stato progettato da *Matsumoto Yukihiro* e la prima versione considerata stabile risale al 1996.
Ruby è un linguaggio generico *interpretato*, *dinamico*, *riflessivo* e *multi-paradigma*.
Matsumoto afferma di essere stato influenzato dalle idee del *Perl*, *Smalltalk* e *Lisp* di cui quindi coesistono le caratteristiche. Un modello ad oggetti *puro* (ogni elemento è un oggetto), caratteristiche funzionali e praticità imperativa. Lo spirito che avrebbe spinto *Matsumoto* nell'ideazione di **Ruby** si può assimilare al *principio della minima sorpresa* anche se come da lui affermato era *il principio della sua minima sorpresa*.

Alcune caratteristiche.

- Sintassi semplificata e flessibile
- Linguaggio ad oggetti *puro* con *ereditarietà singola*, *mixin* e *metaclassi*
- i costrutti sono *espressioni*
- Tipizzazione dinamica, *duck typing*
- Riflessione dinamica e *metaprogrammazione*
- Closure, iteratori e generatori
- Argomenti predefiniti nei metodi
- Interpolazione delle stringhe
- Sintassi letterale per le strutture dati
- Thread e fiber
- Gestione delle eccezioni
- REPL
- API per estensioni in C
- Gestione centralizzata delle librerie (RubyGems)
- multipiattaforma
- *Batterie incluse*
- estensione del file sorgente: *rb*

Sintassi e semantica.

La sintassi è simile a quella del *Perl* e del *Python*.

Le *classi* ed i *metodi* sono dichiarati con delle parole chiave, mentre i blocchi di codice ne hanno una doppia possono essere racchiusi tra parentesi o parole chiave.

Le variabili hanno quattro possibili aree di visibilità:

- globale, prefisse con `$`: `$variabile`
- classe, prefisse con `@@`: `@@variabile`
- istanza, prefisse con `@`: `@variabile`
- locale, senza prefisso: `variabile`

Le *variabili di istanza* e le *variabili di classe* sono completamente private ed hanno bisogno di *getter* e *setter* ma il linguaggio offre dei metodi di convenienza per la generazione dinamica di questi metodi:

- attr_accessor: *getter* e *setter*
- attr_reader: *getter*
- attr_writer: *setter*

Spazi ed indentazione non sono significanti come nel Python.

Ruby è un linguaggio *orientato agli oggetti puro* ed ogni valore, comprese le primitive ed il *nil*, è un oggetto.

Il codice è valutato ed *istanziato* alla *lettura* per cui la stessa dichiarazione di *classe* è di per sé un oggetto.

Le *funzioni* sono *metodi* nel senso comune dei linguaggi orientati agli oggetti e sono invocate sempre su un oggetto. Quelle dichiarate al di fuori di classi o moduli entrano a far parte del genitore principale (`Object`) diventando di fatto *globali*. I metodi (come gli altri blocchi esecutivi) restituiscono l'ultimo valore assegnato relegando la parola chiave `return` ad utilizzi specifici e limitati.

L'ereditarietà è singola ma ogni classe può importare più *moduli* con un meccanismo di *mixin*.

Il concetto di tipo segue la regola del *duck typing* soffermandosi più sul cosa fa un oggetto che piuttosto sul cosa è. Di fatto un oggetto è i suoi metodi e la risoluzione si limita alla domanda: *sei in grado di rispondere a questo messaggio?* In caso affermativo la computazione procederà, altrimenti verrà emessa una eccezione `NoMethodError` a meno che non sia implementato un *proxy* attraverso `method_missing` che è invocato nel caso la risoluzione non vada a buon fine.

```ruby
class Pippo

  def method_missing(name, *args, &block)
    puts "method: #{name} arguments: #{args.join(',')}"
  end

end

pippo = Pippo.new

pippo.m(1, 2, 3)
```

sorgente

Ruby è anche un *linguaggio funzionale* seppure *impuro* considerando che:

- ha *funzioni anonime* sotto forma di *lambda* e *Proc*, le *closure* normalmente dette *blocks*
- le funzioni sono quindi *oggetti di prima classe* (*first-class*)
- le istruzioni sono a tutti gli effetti delle *espressioni* ed hanno un *valore*
- le funzioni restituiscono l'ultimo valore della loro computazione

```ruby
somma = lambda { | a, b | a + b }

def applica_la_funzione_alla_lista(f, lista)
  lista.reduce(&f)
end

puts applica_la_funzione_alla_lista(somma, [1, 2, 3, 4, 5]) #=> 15
```

sorgente

Equivalente a questo sorgente:

```ruby
somma = ->(a, b) { a + b } #Nuova sintassi per le lambda

def applica_la_funzione_alla_lista(f, lista)
  lista.reduce(&f)
end

puts applica_la_funzione_alla_lista(somma, [1, 2, 3, 4, 5]) #=> 15
```

sorgente

Interprete e REPL

Esistono diverse implementazioni del linguaggio Ruby:

- Ruby MRI, quella ufficiale scritta in C (https://www.ruby-lang.org)
- JRuby, una implementazione in Java (https://www.jruby.org/)
- TruffleRuby, la versione all'interno di GraalVM (https://www.graalvm.org/)
- Rubinius (https://rubinius.com/about/)
- MagLev (http://maglev.github.io/)
- Crystal, che non garantisce la compatibilità e compila nativamente (https://crystal-lang.org/)
- Per le altre consultate https://github.com/cogitator/ruby-implementations/wiki/List-of-Ruby-implemen

Installazione

Installare **Ruby** non è cosa complessa, sulle maggiori distribuzioni Gnu/Linux è presente nei repository ufficiale (anche se spesso non nell'ultima versione) e va fatto riferimento al gestore di pacchetti della distribuzione specifica.

Nello sviluppo ma soprattutto nella *distribuzione* di applicazioni **Ruby** sono utili dei veri e propri manager. I più usati sono:

- rvm (https://rvm.io/)
- chruby (https://github.com/postmodern/chruby)
- rbenv (https://github.com/rbenv/rbenv)
- uru (https://bitbucket.org/jonforums/uru)

Rvm è il più completo ma anche il più invasivo. *Uru* il più semplice e più *multipiattaforma*.

- ruby-build (https://github.com/rbenv/ruby-build) è una linea di comando per compilarlo dai sorgenti in maniera facilitata.R

Informazioni sull'installazione: https://www.ruby-lang.org/en/documentation/installation/

Interprete (Ruby MRI)

L'interprete **Ruby** è un eseguibile dal nome *ruby* il cui uso è:

`ruby [switches] [--] [programfile] [arguments]`

Per evidenziare la lista completa delle opzioni da linea di comando: `ruby --help`.

L'interprete mette a disposizione una serie di *variabili predefinite* tra cui:

Alcune variabili predefinite (possono essere mutate)

```ruby
puts "Nome delle script: #{$0}"
puts "Argomenti della linea di comando: #{$*}"
puts "Il percorso di caricamento per gli script: #{$:}"
puts "Il percorso di caricamento per gli script: #{$LOAD_PATH}"
```

sorgente

Alcune costanti predefinite

```ruby
puts "La versione di Ruby: #{RUBY_VERSION }"
puts "Data di rilascio: #{RUBY_RELEASE_DATE }"
puts "La piattaforma: #{RUBY_PLATFORM }"
puts "Un dizionario che contiene le variabili di ambiente: #{ENV.keys.join(',')}"
```

sorgente

Per consultare la lista completa: https://ruby-doc.org/core-2.5.3/doc/globals_rdoc.html

Interactive Ruby Shell

La *shell interattiva* di **Ruby** è una REPL (read-eval-print loop) il cui nome eseguibile è *irb*.
Permette di valutare immediatamente espressioni e comandi nel linguaggio permettendo di sperimentare in real-time. È comunque possibile utilizzare direttamente l'interprete al *prompt* dei comandi:

```
nissl:code ⛩ ruby

def stampa(t)
  puts t
end

stampa(”hello world”)

^D

hello world

nissl:code ⛩
```

Irb ha capacità di mantenere una storia dei comandi dati e una rudimentale capacità di editing delle linee:

```
nissl:code ⛩ irb
irb(main):001:0> def stampa(t)
irb(main):002:1> puts t
irb(main):003:1> end
=> :stampa
irb(main):004:0> stampa(”hello irb”)
hello irb
=> nil
irb(main):005:0>
```

Una *REPL* più avanzata di `irb` è `pry` (http://pryrepl.org/) che oltre alla normale funzionalità offerta da quella *standard* dispone alcune caratteristiche interessanti tra cui:

- sintassi colorata
- una architettura a plugin
- funzioni di documentazione
- navigazione nei moduli con `ls`, `cd` e altri comandi
- integrazione con vari editor

```
nissl:code ⛩ pry
[1] pry(main)> def stampa(t)
[1] pry(main)*   puts t
[1] pry(main)* end
=> :stampa
[2] pry(main)> stampa(”hello world”)
hello world
=> nil
```

```
[3] pry(main)> cd Hash
[4] pry(Hash):1> ls
Object.methods: yaml_tag
Hash.methods: []  try_convert
Hash#methods:
  <      clear                delete      fetch         invert   pretty_print        shift    transform_keys
  <=     compact              delete_if   fetch_values  keep_if  pretty_print_cycle  size     transform_keys!
  ==     compact!             dig         flatten       key      rassoc              slice    transform_values
  >      compare_by_identity  each        has_key?      key?     rehash              store    transform_values!
  >=     compare_by_identity? each_key    has_value?    keys     reject              to_a     update
  []     default              each_pair   hash          length   reject!             to_h     value?
  []=    default=             each_value  include?     member?  replace             to_hash  values
  any?   default_proc         empty?      index         merge    select              to_proc  values_at
  assoc  default_proc=        eql?        inspect       merge!   select!             to_s
locals: _  __  _dir_  _ex_  _file_  _in_  _out_  _pry_
[5] pry(Hash):1> show-method each

From: hash.c (C Method):
Owner: Hash
Visibility: public
Number of lines: 10

static VALUE
rb_hash_each_pair(VALUE hash)
{
```

```
26      RETURN_SIZED_ENUMERATOR(hash, 0, 0, hash_enum_size);
27      if (rb_block_arity() > 1)
28    rb_hash_foreach(hash, each_pair_i_fast, 0);
29      else
30    rb_hash_foreach(hash, each_pair_i, 0);
31      return hash;
32  }
33  [6] pry(Hash):1>
```

Elementi base del linguaggio

Parole chiave

`__ENCODING__ __LINE__ __FILE__ BEGIN END alias and begin break case class def defined? do else elsif end ensure false for if in module next nil not or redo rescue retry return self super then true undef unless until when while yield`

Alcune parole chiave hanno dei corrispondenti operatori con precedenza più elevata:

- `&& > and`
- `|| > or`
- `! > not`

Letterali

Le espressioni letterali creano od inizializzano oggetti e coprono i vari tipi di base:
`nil, Booleani, Numeri, Stringhe, Simboli, Arrays, Hashes, Serie, Espressioni regolari, Procs`

nil e Booleani

Sia `nil` che `false` vengono ambedue valutati come `false` nei test. Il valore `nil` equivale al `null` di altri linguaggi di programmazione ma in sé è una istanza della classe `NilClass`. In **Ruby** tutto è un oggetto e non va dimenticato.
Ogni oggetto che non sia `nil` o `false` è valutato `true` nelle espressioni condizionali.

```
1  nil.class
2  #=>NilClass
3  false.class
4  #=>FalseClass
5  true.class
6  #=>TrueClass
```

Numeri

Nei numeri si può usare il carattere _ (underscore) come separatore per aumentarne la leggibilità: sia `1234` che `1_234` corrispondono a `1234`.
I numeri in virgola possono essere espressi sia con il punto come separatore decimale che in notazione esponenziale: `0.000134` o `134e-6`.

```
1  1234.class
2  #=>Integer
3  1.234.class
4  #=>Float
```

Per esprimere numeri in altre notazioni si usano dei prefissi: `0d` la notazione decimale, `0x` esadecimale, `0` o `0o` ottale, `0b` binaria.

Stringhe

La notazione più comune per dichiarare una stringa è tra doppi apici: `"questa è una stringa"`.
Si può usare anche quella tra apici singoli `'questa è una stringa'`, ma in questo caso non è permessa né l'intepolazione, né i vari caratteri come `\n`, `\t` e così via.
Le stringhe possono essere create anche con `%(questa è una stringa)` che con `%q(questa è una stringa)`, che equivalgono rispettivamente alle forme con apice doppio o singolo.
Stringhe adiacenti sono automaticamente concatenate. È possibile costruire una stringa di un solo carattere con `?`.

```
num = 4
#=>4
"oggi siamo in #{num}"
#=>"oggi siamo in 4"
'oggi siamo in #{num}'
#=>"oggi siamo in \#{num}"
%(oggi siamo in #{num})
#=>"oggi siamo in 4"
%q(oggi siamo in #{num})
#=>"oggi siamo in \#{num}"
"#{num}" "a" '1'  "b" "c"
#=>"4a1bc"
?X
#=>"X"
```

Stringhe lunghe (HEREDOC)

Come in altri linguaggi sono supportate le stringhe lunghe o here document con alcune modalità. Il *marcatore* di inizio e fine può essere qualunque e per convenzione si usa tutto maiuscolo.

```
s1 = <<HD
  - Il terminale deve essere allineato. -
  sono una stringa
  lunga
HD

s2 = <<-HD
  - Il terminale può essere indentato. -
  sono una stringa
  lunga
  HD

s3 = <<~HD
  - Il terminale può essere indentato e la stringa viene allineata. -
  sono una stringa
  lunga
  HD

puts s1
puts s2
puts s3
```

sorgente

Altri usi interessanti delle heredoc: invocare un metodo sulla stringa e lanciare un programma esterno catturandone l'output.
La forma «`‘HD‘` equivale ad invocare `Kernel#‘` che è un metodo che lancia un programma come sottoprocesso.

```
s1 = <<HD.split
  sono una stringa
  lunga
HD

s2 = <<`HD`
  date
HD
```

```ruby

p s1
puts s2
```

sorgente

Simboli

I simboli sono dei nomi costanti all'interno dell'interprete, una volta allocati vengono recuperati. Prima della versione 2.2 tutti gli oggetti della classe Symbol permanevano per tutta la durata dell'esecuzione (portando ad alcuni problemi), oggi quelli creati durante l'esecuzione (tramite il metodo `to_sym`) sono eleggibili per il *garbage collector*.
Hanno un uso estensivo nella programmazione **Ruby** specialmente come chiavi per i dizionari (Hash) dove sono (o meglio erano) essenzialmente più veloci delle stringhe non venendo riallocati ogni volta.

```ruby
module A
  $str_a = "pippo"
  $sym_a = :pippo
end

puts $str_a.object_id

module B
  $str_a = "pippo2"
  $str_b = "pippo"
  $sym_b = :pippo
end

puts $str_a.object_id
puts $str_b.object_id
puts $sym_a.object_id
puts $sym_b.object_id
```

sorgente

Array

L'array è simile come in molti altri linguaggi ma può essere *eterogeneo* e si utilizzano le `[]`.

```ruby
e = [1+2]
#=>[3]
c = [1 + 2, 3 + 1]
#=>[3, 4]
d = [1, [2, 3], 2]
#=>[1, [2, 3], 2]
c = [1, "a"]
#=>[1, "a"]
b = ["a", "b", "c"]
#=>["a", "b", "c"]
a = [1, 2, 3]
#=>[1, 2, 3]
```

Espressioniregolari (Regexp)

Le espressioni regolari sono simili a quelle del *Perl* e vengono inizializzate in vari modi:

- `/espressione regolare/[flag]`
- `%r{espressione regolare}[flag]`

Se si usa la forma `%r` si possono usare vari delimitatori:

```ruby
%r[.*]
#=>/.*/
%r!.*!
#=>/.*/
```

```ruby
%r .*
#=>/.*/
%r^.*^
#=>/.*/
%r|.*|
#=>/.*/
```

%

Come si è visto con il segno `%` si possono creare le stringhe o le espressioni regolari ma non solo:

```ruby
%i(a b c d)
#=>[:a, :b, :c, :d]
%q(questa è una stringa)
#=>"questa è una stringa"
%r(.*)
#=>/.*/
%s(simbolo)
#=>:simbolo
%w(a b c d)
#=>["a", "b", "c", "d"]
%x(echo "comando echo invocato")
#=>"comando echo invocato\n"
```

L'operatore base per il *matching* di una espressione regolare è =~ che più o meno equivale al metodo `match` (il metodo restituisce una istanza di MatchData invece dell'indice iniziale o `nil`).

```ruby
md = /(.)(.)(\d+)(\d)/.match("Pippo123.")
#=> MatchData "po123" 1:"p" 2:"o" 3:"12" 4:"3"
first_occ_index = /(.)(.)(\d+)(\d)/ =~ "Pippo123."
#=>3
first_occ_index
#=>3
```

Hash (Dizionari)

L'`Hash` è una delle strutture probabilmente più usate in ambito **Ruby**.
Non ne è garantito l'ordine dichiarativo, il linguaggio cercherà in base alla struttura la migliore ottimizzazione in memoria.
L'hash è creato usando delle coppie *chiave* => *valore* tra `{}`.

```ruby
h = { "k" => 1, "k2" => "valore" }
#=>{"k"=>1, "k2"=>"valore"}
h = { "k":1, "k2": "valore" }
#=>{:k=>1, :k2=>"valore"}
h = { k: 1, k2: "valore" }
#=>{:k=>1, :k2=>"valore"}
h = { :k => 1, :k2 => "valore" }
#=>{:k=>1, :k2=>"valore"}
h = { :k => 1, :k2 => "valore", a: [1, 2, 3], h: {1 => "1", 2 => "2" }}
#=>{:k=>1, :k2=>"valore", :a=>[1, 2, 3], :h=>{1=>"1", 2=>"2"}}
```

Come chiave può essere usato praticamente qualunque valore, visto che la chiave stessa è un oggetto.

```ruby
h = {[1,2,3] => "chiave come array"}
#=>{[1, 2, 3]=>"chiave come array"}
h = {{a: 1, b: 2}  => "chiave come hash"}
#=>{{:a=>1, :b=>2}=>"chiave come hash"}
h = {(1..2) => "chiave come range"}
#=>{1..2=>"chiave come range"}
h = { ->(a, b) { a  + b } => "chiave come proc/lambda" }
#=>{#<Proc:0x00007ff2d660bf00@(pry):22 (lambda)>=>"chiave come proc/lambda"}
```

Fino alla versione 2.2 di **Ruby** era consigliabile per motivi prestazionali usare i simboli come chiavi che come detto sopra sono allocati una sola volta. Con la versione 2.2 le stringhe come chiavi sono state ottimizzate e la differenza di prestazione è minima (anche se sempre a favore dei simboli).

Range (intervalli)

I Range sono intervalli di valori, l'estremo finale può essere incluso od escluso. Anche qui si possono creare intervalli di qualsiasi valore.

```
(1..5)
#=>1..5
(1...5)
#=>1...5
(1...5).to_a
#=>[1, 2, 3, 4]
(1..5).to_a
#=>[1, 2, 3, 4, 5]
('a'..'z').to_a
#=>["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w",
```

Un intervallo può contenere qualsiasi oggetto che risponda all'operatore <=> e se usato come una enumerazione a `succ`. <=> ha solo tre possibili valori di ritorno: `-1`, `0`, `1`.

- a <=> b, se a < b allora -1
- a <=> b, se a = b allora 0
- a <=> b, se a > b allora 1
- se a e b non sono comparabili allora `nil`

Una classe di esempio da usare all'interno di un range.

```
class StrOfLetter
  # si include (mixin) il modulo Comparable per avere gli operatori
  # >, < , >=, <=, ==, and between? gratuitamente
  include Comparable
  attr :lenght

  def initialize(lenght)
    @letter = 'A'
    @lenght = lenght
  end

  def <=>(other)
    @lenght <=> other.lenght
  end

  def succ
    StrOfLetter.new(@lenght + 1)
  end

end

range = (StrOfLetter.new(1)..StrOfLetter.new(5))
p range.to_a
```

sorgente

Arricchiamo la classe, aggiungendo `inspect` che ritorna una rappresentazione *umana* dell'oggetto e `to_s`.

```
class StrOfLetter
  include Comparable
  attr :lenght

  def initialize(lenght)
    @letter = 'A'
    @lenght = lenght
  end

  def <=>(other)
    @lenght <=> other.lenght
  end

  def succ
    StrOfLetter.new(@lenght + 1)
  end
```

```ruby

  def to_s
    value = inspect
    "word: #{value} size: #{value.size}"
  end

  def inspect
    @letter * @lenght
  end
end

range = (StrOfLetter.new(1)..StrOfLetter.new(5))
p range.to_a
p range.to_a[1].to_s
```

sorgente

Proc e lambda

Le *proc* sono *funzioni anonime*. In **Ruby** le *lambda* e le *proc* hanno delle sottili differenze:

- `lambda {}` o `->{}`, agiscono come funzioni in sé, chiamando `return` all'interno del blocco escono semplicemente.
- `Proc.new {}` o `proc {}`, agisce come parte del metodo chiamante, uscire con `return` fa uscire anche il chiamante.

Vediamo le differenze.

```ruby
@lmbd1 = lambda {return "sono una lambda 1"}
@lmbd2 = ->{return "sono una lambda 2"}
@proc = Proc.new {return "sono una proc"}

def call_lmbd
  puts "calling lambda..."
  puts @lmbd1.call
  puts @lmbd2.call
  puts "continuo..."
end

def call_proc
  puts "calling proc..."
  puts @proc.call
  puts "continuo..."
end

call_lmbd
call_proc
```

sorgente

Essendo funzioni anonime possono accettare parametri ed anche qui ci sono differenze:

- `lambda {}` o `->{}`, richiedono che i parametri siano rispettati e permettono quelli di default.
- `Proc.new` o `proc {}`, non richiedono di rispettare i parametri.

```ruby
l = ->(a, b = 1) { a + b }
#=> Proc:0x00007ff2d63dccc0@(pry):65 (lambda)
l.call(3)
#=>4
l.call
ArgumentError: wrong number of arguments (given 0, expected 1..2)
from (pry):65:in `block in __pry__'
pr = Proc.new {|a, b| a + b }
#=> Proc:0x00007ff2d5462b28@(pry):68
pr.call
NoMethodError: undefined method `+' for nil:NilClass
from (pry):68:in `block in __pry__'
pr = Proc.new {|a, b| 1 + 1 }
#=> Proc:0x00007ff2d54dd670@(pry):70
pr.call
#=>2
```

Costanti e `freeze`

Per essere considerato una *costante* il nome di un oggetto deve iniziare per lettera maiuscola. Sono così costanti i nomi delle *classi* e dei *moduli*. Per convenienza le costanti che contengono solo valori sono dichiarate tutto in maiuscolo.
Le costanti in **Ruby** in realtà sono *mutabili* e questo può creare confusione, il linguaggio si limita ad emettere un avvertimento in caso di sostituzione ma ignora in altri casi.

```
COSTANTE = 1
#=> 1
COSTANTE = 2
warning: already initialized constant COSTANTE
warning: previous definition of COSTANTE was here
#=> 2

NAME = "hello"
#=> "hello"
NAME << "world"
#=> "helloworld"
```

Per ovviare a questo possiamo usare il metodo `freeze`.

```
FREEZED = "hello".freeze
#=> "hello"
FREEZED << "world"
#FrozenError: can't modify frozen String
```

Attenzione che però `freeze` *congela* ed ha senso solo su oggetti considerati mutabili come stringhe o array, solo l'istanza su cui è invocato è *congelata*.

```
a = [1, 2, 3]
#=> [1, 2, 3]
a.freeze
#=> [1, 2, 3]
a << 4
FrozenError: can't modify frozen Array

a = [1, 2, 3, 4] #riassegnando si crea un nuovo oggetto non congelato
#=> [1, 2, 3, 4]
a << 4
#=> [1, 2, 3, 4, 4]
```

È stato annunciato che Ruby 3 avrà le stringhe immutabili di default.

Strutture di controllo e cicli

In **Ruby** più o meno tutto è un'*espressione* e quindi in qualche modo restituirà un valore. L'espressione condizionale `if-else-end` restituirà il valore assegnato per ultimo. Come scorciatoia è presente anche l'operatore ternario `?:`.
L'espressione `unless-else-end` equivale a `if not true-else-end`.

```
a = 10
#=> 10
ret = if a > 9
        "a è maggiore di nove"
      else
        "a non è maggiore di nove"
      end
#=> "a è maggiore di nove"
ret
#=> "a è maggiore di nove"

ret = a > 9 ? "vero" : "falso"
#=> "vero"
```

Ci sono le classiche strutture per i cicli come `for value in range do-end`, `while-end` o `begin-end-while`, `until-do-end` o `begin-end-until` ma sono veramente poco utilizzare in **Ruby** preferendo altri metodi.

```
for n in 1..5 do
   puts "questo loop ciclerà #{n} volte"
 end
#=> questo loop ciclerà 1 volte
#=> ...
#=> 1..5
```

```
x = 5
#=> 5
while x > 0
  x -= 1
  puts "questo loop ciclerà #{x} volte"
end
#=> questo loop ciclerà 4 volte
#=> ...
#=> nil
```

Chi programma in **Ruby** preferisce invece l'iterazione attraverso i vari metodi delle *collezioni*.

```
(1..5).each do |n|
  puts "questo loop ciclerà #{n} volte"
end
#=> questo loop ciclerà 1 volte
#=> ...
#=> 1..5

5.times.each { |n| puts "questo loop ciclerà #{n} volte" }
#=> ...
```

Il *blocco* `do-end` si può scrivere anche con `{}` ed è in effetti una *proc*, quindi il metodo `each` accetta come parametro una proc che chiama iterativamente per ogni elemento della collezione.

Possiamo scrivere una personale versione di `each`

```
def my_each(coll, &block)
  for e in coll do
    block.call(e)
  end
end

my_each(1..5) { |n|
  puts "questo loop ciclerà #{n} volte"
}
```

sorgente

Il parametro `&block` è la proc che passeremo alla funzione e che sarà invocata nel ciclo sulla collezione.

Metodi

I metodi sono definiti attraverso la parola chiave `def`, hanno un nome ed eventualmente dei parametri. Restituiscono l'ultimo valore nel blocco esecutivo rendendo di fatto relegata ad usi speciali la parole `return`. Si tende a non forzare mai l'uscita dal metodo nel mezzo di una computazione.

```
def met(par1, par2, par3)
  puts par1
  puts par2
  puts par3
end
met(1,2,3)
#=> 1
#=> 2
#=> 3
#=> nil
```

Sono possibili parametri di default che non necessariamente dovranno essere in cima o in fondo alla lista quanto piuttosto raggruppati insieme.

```ruby
def met(par1 = 1, par2 = 2, par3)
  puts par1
  puts par2
  puts par3
end
met(3)
#=> 1
#=> 2
#=> 3
#=> nil
```

Dichiarare il metodo come `def met(par1 = 1, par2, par3 = 3)` genererà un errore sintattico.
Il linguaggio supporta oltre agli *argomenti posizionali* anche quelli nominali che seguono le stesse regole di quelli di default.

```ruby
def met(primo:, secondo: 2)
   primo + secondo
end

met(primo: 1)
#=> 3
```

Sono possibili numeri variabili di argomenti ed argomenti hash.

```ruby
def met(*args)
   p args
end

met(1,2,3)
[1, 2, 3]
#=> [1, 2, 3]

def met2(**hash)
   p hash
end

met2(a: 1, b: 2)
{:a=>1, :b=>2}
#=> {:a=>1, :b=>2}
```

la decomposizione di un oggetto che risponde al metodo `to_ary` come un Array.

```ruby
 def met((a, b, *c))
   p a: a, b: b, c: c
 end

met([1, 2, 3, 4, 5])
#=> {:a=>1, :b=>2, :c=>[3, 4, 5]}
```

I metodi hanno sempre un parametro implicito che può essere esplicitato nella dichiarazione: il *blocco*. Come visto può giocare un ruolo importante nei metodi che lavorano sulle collezioni ma non solo. Il parametro *blocco* deve nel caso sia esplicitato essere l'ultimo della lista.

```ruby
def met
  yield
end

met
#=> LocalJumpError: no block given (yield)

met { p "sono il blocco" }
#=> "sono il blocco"
```

Per valutare o meno la presenza di un blocco implicito è disponibile `block_given?`.

```ruby
def met
   if block_given?
     yield
   else
     "nessun blocco disponibile"
   end
 end

met
#=> "nessun blocco disponibile"

met { p "ci sono" }
#=> "ci sono"
```

I blocchi sono delle *proc* (funzioni anonime) e quindi possono avere parametri come già visto nella funzione `my_each` che possiamo scrivere anche:

```ruby
def my_each(coll)
  for e in coll do
    yield e
  end if block_given?
end

my_each(1..5) { |n|
  puts "questo loop ciclerà #{n} volte"
}
```

sorgente

In **Ruby** esistono metodi che teminano con `?`, è una convenzione per indicare i metodi il cui intento è puramente di controllo *booleano*.

```ruby
@variabile_non_inizializzata.nil?
#=> true

[1, 2, 3].include?(2)
#=> true

File.exist?("nome.txt")
#=> false
```

Oltre a questi ci sono quelli che terminano per `!`, come sopra è una convenzione per indicare i *metodi distruttivi* cioè quelli che modificano direttamente l'oggetto su cui sono invocati. Spesso sono presenti in due versioni *distruttiva* e non.

```ruby
a = [1, 1, 2, 2, 3]
#=> [1, 1, 2, 2, 3]
b = a.uniq
#=> [1, 2, 3]
a
#=> [1, 1, 2, 2, 3]
 a.uniq!
#=> [1, 2, 3]
a
#=> [1, 2, 3]
```

Eccezioni

La gestione delle eccezioni è molto semplice.

```ruby
begin
  1 / 0
rescue
  puts "Eccezione generica"
end
```

```ruby
begin
  1 / 0
rescue Exception => e
  puts e
end
```

```ruby
begin
  1 / 0
rescue ZeroDivisionError => e
  puts "Exception Class:     #{ e.class.name }"
  puts "Exception Message:   #{ e.message }"
  puts "Exception Backtrace: #{ e.backtrace }"
end
```

Una eccezione viene lanciata con `raise`.

```ruby
begin
  raise ArgumentError.new("errore sugli argomenti")
rescue ArgumentError => e
  puts e.message
end
```

Le eccezioni sono delle semplici classi **Ruby**.

```ruby
class PermissionDeniedError < StandardError

  attr_reader :action

  def initialize(message, action)
    super(message)
    @action = action
  end
end

begin
  raise PermissionDeniedError.new("Permesso negato", :delete)
rescue PermissionDeniedError => e
  puts "Exception Class:     #{ e.class.name }"
  puts "Exception Message:   #{ e.message }"
  puts "Exception Action:   #{ e.action }"
  puts "Exception Backtrace: #{ e.backtrace }"
end
```

sorgente

Organizzazione per file.

Un progetto non è mai composto di un solo file a meno che non sia banale. **Ruby** come praticamente tutti i linguaggi moderni, ha alcuni meccanismi per poter controllare questo aspetto.

```ruby
load 'filename'
require 'filename'
```

Con `load` o `require` si caricano file esterni nello spazio di visibilità del richiedente. I file di per sé non sono *moduli*. La differenza tra i due è che `load` è una direttiva che caricherà ogni volta il file mentre `require` solo alla prima. Un uso tipico è nei *web framework* dove in ambiente di *sviluppo* è usata la `load` e in produzione la `require`.
I file se non hanno un percorso assoluto o relativo al chiamante verranno cercati nell'elenco dei percorsi di ricerca contenuti in `$:` o `$LOAD_PATH`.
Queste *variabili* sono manipolabili per aggiungere a *runtime* percorsi di ricerca.

```ruby
$LOAD_PATH.unshift('/aaaa') unless $LOAD_PATH.include?('/aaaa')
```

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

Programmazione orientata agli oggetti

Ruby è un linguaggio orientato agli oggetti *puro* dove tutto è un oggetto e tutto è valutato. Supporta l'ereditarieta a singolo genitore e la possibilità di incorporare uno o più moduli. I moduli sono principalmente delle collezioni di metodi più o meno polimorfi il cui scopo è quello di fornire un ambiente di visibilità e supportare il *mixin*. Alcuni moduli sono incorporati a livello di interprete come per esempio il modulo `Kernel` importato all'interno della classe `Object` che è la *radice di ogni oggetto* in **Ruby** (non è proprio vero, `BasicObject` è il vero antenato ma ha un uso interno o speciale come per esempio creare una nuova gerarchia indipendente). Ogni istanza o classe ha perciò accesso ai metodi del `Kernel`. Un modulo è creato tramite la parola chiave `module`, mentre la classe con `class`.

Classi e Moduli

```ruby
module MyModule
  def met(v)
    p v
  end
end

class MyClass
  include MyModule
end

k = MyClass.new

k.met("metodo dal modulo")
```

sorgente

I moduli possono essere annidati.

```ruby
module MyModule
  def met(v)
    p v
  end
  module MyModuleNested
    def met2(v)
      p "nested module: #{v}"
    end
  end
end

class MyClass
  include MyModule
  include MyModule::MyModuleNested
end

k = MyClass.new

k.met("metodo dal modulo")
k.met2("metodo dal modulo")
```

sorgente

L'operatore di visibilità : :

```ruby
CONSTANT = "sono una costante sulla radice"

module MyModule
  CONSTANT = "sono una costante"

  def c
    p CONSTANT
  end
```

```ruby

  module MyModuleNested
    def c_from_parent
      p CONSTANT
    end

    def c_from_root
      p ::CONSTANT
    end
  end
end

class MyClass
  include MyModule
  include MyModule::MyModuleNested
end

k = MyClass.new

k.c_from_parent
k.c_from_root
```

sorgente

Anche le classi possono essere annidate.

```ruby
class MyClass
  class Pippo
    def met
      p "sono un metodo di Pippo"
    end
  end

  def pippo_new
    Pippo.new
  end
end

k = MyClass.new

k.pippo_new.met
```

sorgente

Ereditarietà

La classe `Child` è dichiarata figlia di `Parent` e ridefinisce il metodo `age` che restituisce l'età. Per poter accedere al metodo del genitore si è usata la direttiva `alias` prima della ridefinizione. È anche una dimostrazione di come la stessa dichiarazione di classe sia una istanza della classe `Class` e non meramente una *rappresentazione letterale* di un tipo. Nella ridefinizione del metodo della classe figlia si può accedere a quello originale del genitore tramite `super`.

```ruby
class Parent
  def age
    "l'età di Parent è 56"
  end
end

class Child < Parent

  alias :parent_age :age

  def age
    "l'età di Child è 13"
  end
end
```

```ruby

c = Child.new

p c.age
p c.parent_age
```

sorgente

Visibilità

Ruby ha tre tipi di visibilità nelle classi e nei moduli: `private`, `protected`, `public`. Il predefinito è `public`, da notare che `private` è simile al `protected` di Java o C++. I metodi `protected` e `private` sono simili e sono invocabili all'interno della stessa gerarchia, gli ultimi non possono avere un *ricevente* nemmeno `self`.

```ruby
class Parent
  def met_public
    "sono pubblico"
  end

  def met_private
    "sono privato"
  end

  def met_protected
    "sono protetto"
  end

  protected :met_protected
  private :met_private
end

class Child < Parent
  def call_protected
    self.met_protected
  end

  def call_private
    self.met_private
  end
end

c = Child.new
puts c.met_public
puts c.call_protected
puts c.call_private
```

sorgente

Metodi di classe e metaclassi

I metodi di classe assomigliano nella pratica ai metodi *statici* di altri linguaggi, ma non lo sono. Non va dimenticato che la classe è una istanza della classe `Class`, quindi tali metodi sono in relazione diretta alla classe ed a essa sono applicati. Sono dichiarati all'interno di un blocco `class « self-end` (metaclasse) oppure come `self.nome_del_metod`
Qui una definizione di un metodo di classe per la generazione dinamica di *getter* e *setter*.

```ruby
class Parent
  class << self
    def attr_acc(s)
      define_method("#{s}=") do |v|
        instance_variable_set("@#{s}", v)
      end
      define_method("#{s}") do
        instance_variable_get("@#{s}")
      end
    end
```

```
  end
end

class Child < Parent
  attr_acc :value

  def sum(a)
    a + @value
  end
end

c1 = Child.new
c1.value = 10
p c1.sum(10)
```

sorgente

Il metodo di classe può essere invocato oltre che all'interno della definizione anche tramite *notazione a punto* sul nome della classe.

```
class Parent
  class << self
    def attr_acc(s)
      define_method("#{s}=") do |v|
        instance_variable_set("@#{s}", v)
      end
      define_method("#{s}") do
        instance_variable_get("@#{s}")
      end
    end
  end
end

class Child2 < Parent
  def sum(a)
    a + @value
  end
end
Child2.attr_acc(:value)

c2 = Child2.new
c2.value = 25
p c2.sum(10)
```

sorgente

Classi aperte

In **Ruby** le classi sono aperte.

```
class String
   def mio_metodo
     puts self
   end
 end

"sono un stringa".mio_metodo
#=> sono un stringa
```

Da adesso in poi ogni *stringa* avrà a disposizione il metodo `mio_metodo`.
È una caratteristica potente che ovviamente andrebbe dosata con cura ed evitata il più possibile.

Programmazione funzionale

Ruby è un linguaggio multiparadigma e ha, come accennato, alcune caratteristiche dei *linguaggi funzionali*. È *impuro* considerando che i valori sono *mutabili* in maniera predefinita ed addirittura le *costanti* (alla versione 2.5.x)

sono modificabili pur emettendo degli avvertimenti. Si è accennato al fatto che chi programma in **Ruby** non faccia molto uso di cicli *imperativi* ma preferisca tutta una serie di metodi delle *collezioni* che in qualche modo accettino funzioni da applicare.

```
5.times.each { |n| puts "ciclo numero: #{n}" }
#=> ciclo numero: 0
#=> ciclo numero: 1
#=> ciclo numero: 2
#=> ciclo numero: 3
#=> ciclo numero: 4
```

`5.times` è un metodo dei numeri che di fatto genera un *range* da `0..4`: *partendo da zero per cinque volte*.
La tendenza quindi è ad in qualche modo trasformare un valore od usare una *collezione* per questo scopo.

each

Il metodo è un ciclo imperativo, quindi il suo valore restituito è ignorabile pur avendolo: restituisce semplicemente l'oggetto su cui è invocato. Altri linguaggi lo hanno magari come `forEach` per esempio.
Anche gli `Hash` sono degli *enumerabili* e quindi è applicabile anche ad essi. Il blocco passato prenderà invece che un argomento, due: *chiave* e *valore*.

```
h = {pane: 2, carne: 10, frutta: 5}
h.each {|k, v| puts "il prezzo di #{k} è #{v}" }
#=> il prezzo di pane è 2
#=> il prezzo di carne è 10
#=> il prezzo di frutta è 5
```

Applicando un blocco con un solo argomento, il valore passato sarà una *coppia*:

```
h.each {|k| puts "chiave, valore: #{k}" }
#=> chiave, valore: [:pane, 2]
#=> ...
```

Nel caso ci servisse l'indice corrente `each_with_index {|el, index| }`, se volessimo operare qualche trasformazione:

```
%w(pane latte burro).each_with_object("") { |e, s| s << e + ','}
#=> "pane,latte,burro,"
```

Quello che fa `join` praticamente ma peggio :

```
%w(pane latte burro).join(',')
#=> "pane,latte,burro"
```

map

Due delle operazioni fondamentali della *programmazione funzionale* sono: operare su una *collezione* restituendo un'altra *collezione* (*map*) e trasformare una collezione in qualcosa d'altro (*reduce*).
La `map` è simile ad `each` a parte il fatto che restituisce una nuova *collezione*.

```
(1..5).map { |n| n + 1 }
#=> [2, 3, 4, 5, 6]

%w(roma genova perugia livorno).map {|c| c.upcase }
#=> ["ROMA", "GENOVA", "PERUGIA", "LIVORNO"]

%w(roma genova perugia livorno).map {|c| c.capitalize }
#=> ["Roma", "Genova", "Perugia", "Livorno"]
```

`map` funziona anche per gli `Hash` anche se restituisce un *array*:

```
{pane: 2, carne: 10, frutta: 5}.map {|k, v| "#{k}: #{v}"}
#=> ["pane: 2", "carne: 10", "frutta: 5"]
```

`reduce` (in altri linguaggi `fold`)

L'operazione di `reduce` è leggermente più articolata, ma non per questo complessa. La funzione oltre al *blocco* da applicare accetterà un valore iniziale che ne determinerà anche il tipo del valore in uscita. Il blocco dovrà avere due argomenti: un accumulatore e l'oggetto corrente dell'iterazione.
Sommiamo tutti i valori di una lista:

```
(1..5).reduce(0) {|memo, v| memo += v}
=> 15
```

Troviamo quanti alimenti per tipo:

```
TIPI_ALIMENTO = {
  ”pere”: ”frutta”,
  ”mele”: ”frutta”,
  ”succo”: ”bibita”,
  ”manzo”: ”carne”,
  ”sarago”: ”pesce”,
  ”mozzarella”: ”latticino”
}

def trova_per_tipo(t)
  TIPI_ALIMENTO.reduce([]) {|memo, c| memo << c[0].to_s if c[1] == t; memo}
end

p trova_per_tipo(”frutta”)
p trova_per_tipo(”pesce”)
```

sorgente
`reduce` ed `inject` sono degli `alias` e non ci sono penalità nell'usare l'uno o l'altro come i tutti gli `alias`.

`reduce` ha altre forme:

```
(1..5).reduce(:+)
#=> 15

(1..5).reduce(10, :+)
#=> 25
```

Se non è specificato il valore iniziale come argomento verrà preso il primo elemento della *collezione*. Il *simbolo* (in questo caso `:+`) è quello riferito al un nome di metodo in grado di essere invocato per la classe dell'elemento.

Vediamo come potremmo cambiare il nostro *filtro* per il cibo:

```
TIPI_ALIMENTO = {
  ”pere”: ”frutta”,
  ”mele”: ”frutta”,
  ”succo”: ”bibita”,
  ”manzo”: ”carne”,
  ”sarago”: ”pesce”,
  ”mozzarella”: ”latticino”
}

class AccArray < Array
  def initialize(t)
    @t = t
  end

  def get(tuple)
    self << tuple[0].to_s if tuple[1] == @t
    self
  end
end

def trova_per_tipo(t)
  TIPI_ALIMENTO.reduce(AccArray.new(t), :get)
```

```
end

p trova_per_tipo(”frutta”)
p trova_per_tipo(”pesce”)
```

sorgente

Usando una `lambda` si sarebbe potuto scrivere:

```
# ...
def trova_per_tipo(t)
  get = ->(memo, c) {memo << c[0].to_s if c[1] == t; memo}
  TIPI_ALIMENTO.reduce([], &get)
end
# ...
```

`flat_map`

Alle volte si ha l'esigenza di concatenare i risultati dell'applicazione di un blocco ad una *collezione*. `flat_map` fa esattamente questo e potrebbe per esempio essere usata per *spianare* un array di array.

```
[1, [2, 3], [4,5,6], 7, 8, 9].flat_map { |e| e }
#=> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Anche se `flatten` lavora su più livelli (su un solo livello forse `flat_map` è più veloce):

```
[1, [2, 3], [4,5,6], [[7,8]]].flatten
#=> [1, 2, 3, 4, 5, 6, 7, 8]
```

Inserire qualcosa tra gli elementi:

```
%w(Paola Marco).flat_map {|e| [e, ”ciao!”]}
#=> [”Paola”, ”ciao!”, ”Marco”, ”ciao!”]
%w(Paola Marco).flat_map {|e| ”#{e} ciao!”}
#=> [”Paola ciao!”, ”Marco ciao!”]
```

`zip` ovvero come mischiare il tutto

`zip` si fa prima a vedere che a spiegare:

```
%w(Paola Giorgio Andrea)
#=> [”Paola”, ”Giorgio”, ”Andrea”]
%w(buongiorno buonasera buonanotte)
#=> [”buongiorno”, ”buonasera”, ”buonanotte”]
nomi.zip(saluti)
#=> [[”Paola”, ”buongiorno”], [”Giorgio”, ”buonasera”], [”Andrea”, ”buonanotte”]]
titoli = %w(Dott. Sig. Caro)
#=> [”Dott.”, ”Sig.”, ”Caro”]
titoli.zip(nomi, saluti)
#=> [[”Dott.”, ”Paola”, ”buongiorno”], [”Sig.”, ”Giorgio”, ”buonasera”], [”Caro”, ”Andrea”, ”buonanotte”]]
```

```
nomi = %w(Paola Giorgio Andrea)
saluti = %w(buongiorno buonasera buonanotte)
titoli = %w(Dott. Sig. Caro)

titoli.zip(nomi, saluti).map { |e| [e.take(2), ”#{e.last}!”].flatten.join(' ') }
#=> [”Dott. Paola buongiorno!”, ”Sig. Giorgio buonasera!”, ”Caro Andrea buonanotte!”]
```

Prepariamo qualche lettera...

```
nomi = %w(Paola Giorgio Andrea)
saluti = %w(buongiorno buonasera buonanotte)
titoli = %w(Dott. Sig. Caro)
lettera = <<~HD
Le scrivo mio malgrado per comunicarle che...

```

```
Distinti saluti

Dr. Mario Rossi
HD

lettere = titoli.zip(nomi, saluti).map { |e|
  [[e.take(2), "#{e.last}."].flatten.join(' '), lettera]
}

lettere.each_with_index { |l, i|
  name = "lettera-#{i}.txt"
  puts "salvo il file #{name}"
  File.open(name, 'w') { |f| f << l.join("\n\n") }
}
```

sorgente

Tail recursion

La *ricorsione a coda* è una tecnica tipica nei linguaggio funzionali ed in breve consiste nel recuperare la chiamata di funzione non allocando lo *stack* delle chiamate. In **Ruby** la questione è *particolare* e non la supporta in maniera predefinita ma può essere attivata a livello di compilazione o a runtime; inoltre non è sempre garantita.

```
// vm_opts.h
//...
/* Compile options.
 * You can change these options at runtime by VM::CompileOption.
 * Following definitions are default values.
 */

#define OPT_TAILCALL_OPTIMIZATION       0
#define OPT_PEEPHOLE_OPTIMIZATION       1
#define OPT_SPECIALISED_INSTRUCTION     1
#define OPT_INLINE_CONST_CACHE          1
#define OPT_FROZEN_STRING_LITERAL       0
#define OPT_DEBUG_FROZEN_STRING_LITERAL 0

// ...
```

Facciamo qualche benchmark su due implementazioni di una funzione `fib(n)`, la classica funzione di *Fibonacci* dove i primi due numeri sono 1 mentre quelli che seguono la somma dei due precedenti.

```
require 'benchmark'
include Benchmark

high_limit = 39

def fib(x)
  x < 2 ? 1 : fib(x - 1) + fib(x - 2)
end

def fib_range(r)
  r.map { |n| fib(n) }
end

Benchmark.benchmark(CAPTION, 7, FORMAT, ">total:", ">avg:") { |x|
  rep = x.report(:fib_range){ fib_range(1..high_limit) }
  [rep, rep]
}
```

sorgente

```
fib_range 20.439008   0.000000  20.439008 ( 20.438642)
```

Cambiamo la nostra implementazione usando una *ricorsione in coda*

```ruby
require 'benchmark'
include Benchmark

high_limit = 39

def fib(x)
  def fun(x, nxt, cur)
    x == 0 ? cur : fun(x - 1, cur + nxt, nxt)
  end
  fun(x, 1, 0)
end

def fib_range(r)
  r.map { |n| fib(n) }
end

Benchmark.benchmark(CAPTION, 7, FORMAT, ">total:", ">avg:") { |x|
  rep = x.report(:fib_range){ fib_range(1..high_limit) }
  [rep, rep]
}
```

sorgente

```
fib_range     0.000069   0.000000   0.000069 (  0.000065)
```

Attiviamo l'ottimizzazione ma il codice deve essere *ricompilato* a runtime in questo caso.

```ruby
require 'benchmark'
include Benchmark

high_limit = 39 #10100

def fib_range_recomp(r)
  RubyVM::InstructionSequence.compile_option = {
    tailcall_optimization: true,
    trace_instruction: false
  }
  RubyVM::InstructionSequence.new(<<-EOS).eval
    def fib(x)
      def fun(x, nxt, cur)
        x == 0 ? cur : fun(x - 1, cur + nxt, nxt)
      end
      fun(x, 1, 0)
    end
    def fib_range(r)
      r.map { |n| fib(n) }
    end
    fib_range(#{r})
    EOS
end

Benchmark.benchmark(CAPTION, 7, FORMAT, "> total:", "> avg:") { |x|
  rep = x.report(:fib_range_recomp){ fib_range_recomp(0..high_limit) }
  [rep, rep]
}
```

sorgente

```
fib_range_recomp  0.000155   0.000009   0.000164 (  0.000159)
```

Questa versione è ovviamente più lenta della sua controparte non *ricompilata* ma lo scopo non era la velocità quanto il superamento del limite della ricorsività. Se tentate con iterazioni oltre i 10000 e qualcosa le prime due falliranno con uno `SystemStackError`.

Memoization

Questa è una tecnica spesso usata nella *programmazione funzionale* ma non solo (non è specifica) e permette di immagazzinare i risultati di una computazione per non ricalcolarli ogni volta. Quindi se ci aspettiamo dei risultati coerenti e ripetitivi potrebbe essere utile.

```ruby
module Memoization
  def memoize(mname)
    @@lt ||= Hash.new { |h, k| h[k] = {} }
    fun = Module.instance_method(mname)
    define_method(mname) { |*args|
      @@lt[mname][args] = fun.bind(self).call(*args) unless @@lt[mname].include?(args)
      @@lt[mname][args]
    }
  end
end
```

sorgente

Il modulo `Memoization` ha un solo metodo `memoize` che si dovrà occupare di creare un `Hash` come variabile di classe.
L'hash avrà una struttura di questo tipo: `{:fib=>{[0]=>0, [1]=>1, ...}` una volta costruito. Il metodo `Module.instance_` è in grado di recuperare il metodo come `#<UnboundMethod: Module(Object)#nome_della_funzione>` ed è per questo che l'invocazione dovrà *collegarlo* alla classe corrente (*self*).

Ora che abbiamo un modulo *Memoization* usiamolo con le funzioni per la sequenza di Fibonacci di cui sopra.

```ruby
require 'benchmark'
include Benchmark

require './memoization'
include Memoization

high_limit = 20000

memoize(
  def fib(x)
    x < 2 ? x : fib(x - 1) + fib(x - 2)
  end
)

def fib_range(r)
  r.map { |n| fib(n) }
end

Benchmark.benchmark(CAPTION, 7, FORMAT, ">total:", ">avg:") do |x|
  rep = x.report(:fib_range){ fib_range(0..high_limit) }
  [rep, rep]
end
```

sorgente

La versione precedente oltretutto non avrebbe retto 20000 iterazioni.

Metaprogrammazione

La natura dinamica ed altamente riflessiva rendono **Ruby** molto adatto alla *metaprogrammazione* ed alcuni esempi sono stati fatti in precedenza. Per questo supporto ci sono numerosi metodi che permettono la manipolazione a *runtime* degli oggetti istanziati.

`define_method`

Il metodo permette di *definire* all'interno della classe nuovi metodi. Va ricordato che le classi sono *aperte* e che in ogni momento si può aggiungere qualcosa. Con `define_method` lo possiamo fare in due modi visto che è un *metodo di classe* (appartiene al modulo `Module` come la maggior parte di questi metodi).

```ruby
PERSON = {
  name: "Mario",
  lastname: "Rossi"
}
```

```ruby

class HashAccessor

  def initialize(data)
    @data = data
  end

  define_method(:name) { @data[:name] }
  define_method(:"name=") { |v| @data[:name] = v }
  define_method(:lastname) { @data[:lastname] }
  define_method(:"lastname=") { |v| @data[:lastname] = v }

end

person = HashAccessor.new(PERSON)

puts person.name
person.name = "Maria"
puts person.name
puts person.lastname
```

sorgente

Questa forma è poco pratica, se volessimo aggiungere una coppia *key-value* a `DATA`?

Così invece non abbiamo problemi. Ogni *instanza* ha sempre la possibilità di rintracciare la classe a cui appartiene con `self.class` ed il metodo `send` manderà un messaggio a quell'oggetto (la classe è essa stessa una istanza della classe `Class`).

```ruby
PERSON = {
  name: "Mario",
  lastname: "Rossi"
}

class HashAccessor

  def initialize(data)
    @data = data
    prepareClass
  end

  def prepareClass
    @data.each { |key, value|
      self.class.send(:define_method, key.to_s) { @data[key] }
      self.class.send(:define_method, "#{key.to_s}=") { |v| @data[key] = v }
    }
  end

end

person = HashAccessor.new(PERSON)

puts person.name
person.name = "Maria"
puts person.name
```

sorgente

Ruby ha mutuato il gergo da *Smalltalk* dove il metodo è un *messaggio* verso la classe in grado di comprenderlo. Quindi l'invocazione di un metodo è inevitabilmente l'invio di un messaggio.

```ruby
d = HashAccessor.new({key: "value"})
#=> #<HashAccessor:0x00007f5a4827e030 @data={:key=>"value"}>
d.key
#=> "value"
d.send(:key)
#=> "value"
d.send(:"key=", "pippo")
#=> "pippo"
d.key
#=> "pippo"
```

Possiamo giocare con le lettere dell'alfabeto:

```ruby
class Alfabeto
  def initialize
    @text = []
  end

  def text
    @text.join
  end

  #https://unicode-table.com/en/search/?q=heart
  # □ = U+1F495
  (('a'..'z').to_a << '_' << '□').each { |l|
    define_method(l.to_sym) {
      @text << (l == '_' ? " " : l)
      self
    }
  }
end

alf = Alfabeto.new
p alf.m.a.r.i.o._.r.o.s.s.i._.□.text
```

sorgente

Con `instance_methods`, `methods`, `private_instance_methods`, `private_methods`, `protected_instance_methods`, `protected_methods`, `public_instance_methods`, `public_methods`, `singleton_methods` si può avere una lista delle varie tipologie di metodi di classi ed istanze. Più o meno analoghe funzioni esistono per le variabili.
Come si può aggiungere si può togliere: `method_removed`, `remove_class_variable`, `remove_const`, `remove_instance_va (Object)`, `remove_method (Module)`, `singleton_method_removed`.

Per consultare la documentazione cercare soprattutto in `Object`, `Module`, `BasicObject`, `Class`.

Batterie incluse e documentazione

Ruby è fornito con le *batterie incluse*. Ha una estesa libreria di base consultabile online a questo url: `http://ruby-doc.org/core`; oltre ciò una libreria standard di funzionalità anche questa consultabile online: `http://ruby-doc.org/stdlib/`.

Con la distribuzione sono presenti due comandi per gestire la documentazione: `ri` per consultare la documentazione allegata ed `rdoc` per generare nuova documentazione estraendo dai sorgenti i commenti.

```
□ ri Array
= Array < Object

------------------------------------------------------------------------
= Includes:
Enumerable (from ruby core)

(from ruby core)
------------------------------------------------------------------------
Arrays are ordered, integer-indexed collections of any object.
...
```

`rdoc` è in grado di generare documentazione in vari formati da HTML al formato usato da `ri` con il parametro `-r` o `--ri=`.

RDoc può usare vari sistemi di formattazione dal proprio (RDoc::Markup) come predefinito a *Markdown*, *TomDoc* e *RD*.

```ruby
##
# La classe *Pippo* rappresenta un elemento bla, bla

class Pippo
  ##
```

```ruby
  # inizializza un oggetto di classe Pippo dato l'argomento +value+
  #
  # Farà tante cose interessanti
  # * prima cosa
  # * seconda cosa
  # * terza cosa

  def inizialize(value)
    # ...
  end

end
```

La documentazione: https://ruby.github.io/rdoc/

RubyGems - Bundler

Ruby possiede dei sofisticati gestori di librerie sia globali che locali al progetto.

RubyGems

Ruby possiede un gestore di librerie (chiamate *gemme*) il cui comando è `gem`. Una *gemma* si installa nel sistema con il comando: `gem install <nome della gemma>`.
Nella fase di installazione viene anche generata la documentazione tramite `rdoc`, generazione che si può inibire coi parametri `--no-ri` e `--no-rdoc`.
RubyGems installa le *gemme* in un percorso predefinito gestibile con delle variabili di ambiente e fornisce all'interprete le giuste direttive per poterle trovare e richiedere (`require`) durante l'esecuzione.
La lista delle gemme installate può essere consultata con `gem list` mentre quelle disponibili online da poter installare con `gem list --remote`.
La disinstallazione avverrà col comando `gem uninstall <gemma>` che in caso di più versioni installate presenterà una scelta.

Sito web: https://rubygems.org/

Bundler

Bundler è un sofisticato gestore di gemme per i progetti. È esso stesso una gemma e va installato (ancora) con `gem install bundler`.
All'interno della cartella del progetto si creerà un file di testo chiamato *Gemfile* con `bundle init` e si aggiungeranno le direttive sulle librerie e versioni da installare.

```
source 'https://rubygems.org'
ruby '2.5.0'
gem 'thin'
gem 'oj'
gem 'rake'
gem 'activesupport', '>= 3.1'
gem 'bcrypt'
gem 'sass'
gem 'slim'
gem 'sqlite3'
gem 'sequel'
gem 'carrierwave'
gem 'carrierwave-sequel', :require => 'carrierwave/sequel'
gem 'mini_magick', :require => 'mini_magick'
gem 'kramdown'
gem 'paypal-sdk-rest'
gem 'padrino', '0.14.2'
```

Il comando `bundle install` provvederà allo scaricamento dalla rete ed alla loro installazione, generando nel contempo il file *Gemfile.lock* dove saranno salvati tutti i nomi e versioni comprese le dipendenze delle *gemme* richieste. L'applicazione dovrà poi essere eseguita tramite `bundle exec [--keep-file-descriptors] comando` che provvederà ad impostare l'ambiente giusto per l'esecuzione.

Sito web: https://bundler.io/

Web Framework

In **Ruby** sono stati scritti alcuni dei *web framework* più usati nello sviluppo di *applicazioni web*, un nome su tutti *Ruby On Rails* (https://rubyonrails.org/). *Rails* in abbreviazione, è un *full stack framework* ovvero un insieme di librerie e di metodologie che comprende l'intero aspetto dello *sviluppo web*.
Nel tempo, per alcuni, *Rails* è diventato un ambiente troppo stringente e rigido e quindi altri *framework* più agili sono apparsi.
In risposta a *Rails* apparve *Merb* che non era altro che un *Rails* smontato e reso più modulare. *Merb* rientrò in qualche modo in quello che poi fu *Rails 3*.

Con *Camping* di *_why* (https://en.wikipedia.org/wiki/Why_the_lucky_stiff una delle figure più interessanti nella comunità **Ruby**) iniziò la via dei *microframework*, più flessibili ed adattabili alle varie esigenze. *Sinatra* (http://sinatrarb.com che è basato su *Rack* https://rack.github.io/) è diventato di fatto il *microframework* di elezione, tanto da vantare imitazioni in molti altri linguaggi di programmazione.

Sinatra

Una applicazione *Sinatra* è molto semplice dopo aver installato la *gemma* con `gem install sinatra`.

```ruby
require 'sinatra'

get '/' do
  'Hello world!'
end
```

sorgente

`get '/' do-end` è una rotta HTTP di tipo `GET`. In **Ruby** se non c'è ambiguità le parentesi possono essere omesse e quindi equivale a:

```ruby
require 'sinatra'

get('/') {
  'Hello world!'
}
```

sorgente

Una rotta è un metodo (`get`, `post`, `put`, `patch`, `delete`, `options`, `link`, `unlink`) che ha come argomento minimo il percorso che si vuole esporre ed un blocco che elaborerà la risposta.

Il percorso della rotta può essere descritto in vari modi per esempio inserendo una variabile.

```ruby
require 'sinatra'

get '/hello/:who' do
  "Hello #{params[:who]}"
end
```

sorgente

Una chiamata a http://localhost:4567/hello/mario scriverà *Hello mario*. L'`Hash params` è disponibile all'interno del blocco per *catturare* le parti o le *query* indicate nell'*URL*. Un altro modo di catturare i parametri fornendo argomenti al blocco.

```ruby
require 'sinatra'

get '/hello/:who' do |who|
  "Hello #{who}"
end
```

sorgente

Lo *splat* è disponibile per fornire parti anonime dell'URL.

```ruby
require 'sinatra'

get '/*/*/ciao' do
  "ciao #{params[:splat].join(" ")}, benvenuto!"
end
```

sorgente

Si possono avere diversi tipi di rotte: *espressioni regolari*, con *parametri opzionali*, *query parameters* …
I metodi HTTP in *Sinatra* possono accettare ulteriori argomenti per stabilire quale metodo invocare.

```ruby
get '/solo_firefox', :provides => ['html'] do
  "<html><head></head><body>hello</body></html>
end
```

Un test banale sullo *UserAgent*.

```ruby
require 'sinatra'

get "/", :agent => /.*Vivaldi.*/ do
  p request.user_agent
  "è Vivaldi"
end

get "/" do
  "non è Vivaldi"
end
```

sorgente

Si possono creare delle condizioni personalizzate.

```ruby
require 'sinatra'

def current_user
  {name: "Mario", lastname: "Rossi", role: :any}
end

set(:auth) do |*roles|
  condition do
    if roles.include?(current_user[:role])
      true
    else
      redirect "login", 303
    end
  end
end

get "/admin", :auth => [:admin] do
  "Amministrazione"
end

get "/area", :auth => [:any] do
  "Area pubblica"
end

get "/login" do
  "LOGIN"
end
```

sorgente

Le impostazioni di Sinatra

Il framework dispone di alcune impostazioni predefinite e personalizzabili e se ne possono creare di nuove da usare nell'applicazione.

```
require 'sinatra'

set :riferimento, valore
set :riferimento_ritardato, ->() { "qualcosa da fare }

enable :logging #abilita un'impostazione booleana
disable: dump_errors #disabilita un'impostazione booleana

set :root, File.dirname(__FILE__) #la radice dell'applicazione
enable :static #Se il sistema userà o non userà la public per i file statici
set :public_folder, ->(){ File.join(root, "static") } #Imposta la directory dove risiederanno i file statici.
set :views, ->(){ File.join(root, "views") } #Imposta la directory dove risiederanno i template dei vari generatori di

set :server, %w[puma, thin, webrick] #la lista dei server Rack compliant da utilizzare nell'ordine.
set :port, 3000 #la porta di ascolto del server
set :bind, '0.0.0.0' #L'interfaccia dove il server ascolterà

get "/" do
  "Hello"
end
```

I template in Sinatra

Il framework ha il supporto per numerosi sistemi di *templating* e relativa applicazione di *layout*.

Tra i principali:

Haml (http://haml.info/), *Erb* (incluso in Ruby), *Liquid* (https://shopify.github.io/liquid/), *Markdown* (https://kramdown.gettalong.org/), *SCSS* (http://sass-lang.com/), *Less* (http://lesscss.org/). Vedere la documentazione per la lista completa dei tipi e delle dipendenze (http://sinatrarb.com/intro.html).

Ognuno dei sistemi ha associato un metodo che deve essere invocato per la trasformazione dal modello al contenuto servito.

Per *Slim* per esempio, `slim :index` caricherà dalla directory *views* (se non impostata diversamente) un file chiamato `index.slim`.

Un piccolo progetto

Si presume che *Bundler* sia installato, altrimenti: `gem install bundler`.
Creeremo una struttura di cartelle in questo modo:

```
├── helpers
├── public
│   └── images
└── views
    ├── layouts
    └── scss
```

Per prima cosa sulla radice scriveremo il *Gemfile* per l'impostazione delle gemme di progetto

```
source "https://rubygems.org"

gem 'thin'
gem 'sass'
gem 'slim'
gem 'sinatra'
```

sorgente

Con `bundle install` si installeranno nel sistema e verrà generato il file *Gemfile.lock*. Si userà come web server *Thin*, come sistema di template *Slim* e come linguaggio per i *CSS SASS*.

Per provare se il sistema troverà il file *HTML* statico ne faremo uno nella cartella *public* col nome di test.html.

```
<html>
  <head>
    <title>Titolo</title>
  </head>
  <body>
    Pagina statica
    <p>
      <a href="/">HOME</a>
    </p>
  </body>
</html>
```

sorgente

Inseriremo anche una immagine dal nome *image.png* nella cartella *public/images* e la nostra parte *statica* è pronta.

sorgente

Ci serviranno alcuni metodi di aiuto (*helpers*) da usare all'interno del *template* quindi scriveremo un file all'interno della cartella *helpers* che chiameremo *main_helpers.rb*.

```
module MainHelpers

  def link_to(text, href, **options)
```

```
    %(<a href=#{href} title=#{options[:title].to_s} class=#{options[:class]}>#{text}</a>)
  end

  def image(name, **options)
    %(<img src="/images/#{name}" alt="#{options[:alt].to_s} />)
  end
end
```

sorgente

Sono due classici *metodi helper* ritrovabili anche in *Rails* per esempio.
Passiamo alla applicazione vera e propria.

```
require 'sinatra'
require './helpers/main_helpers'
helpers MainHelpers

set :server, %w[thin]
set :port, 3000
set :bind, '0.0.0.0'

get "/" do
  slim :index, :layout => :"layouts/layout", :locals => {year: 2018, author: "Braccobaldo"}
end

get "/stylesheets/style.css", :provides => :css do
  scss :"scss/style"
end
```

sorgente

L'applicazione è molto semplice, si è richiesto *Sinatra*, il file degli *helpers* e si è incluso il modulo con una direttiva *Sinatra* fornita dal framework. Si è impostato il *web server*, la sua porta e l'indirizzo di ascolto.
Alla rotta *root* si è assegnato un layout (che vedremo dopo) ed un `Hash` che contiene delle variabili locali al template che poi verrà renderizzato. *Slim* supporta questo meccanismo, altri sistemi come *Markdown* no.

Prepariamo il primo *template* in *Slim* dentro la cartella *views* chiamato *index.slim*.

```
h1 Pagina Web
p Sono il testo della pagina web

p==link_to("pagina statica", "/test.html")

p.image== image("image.png")
```

sorgente

Slim è una sorta di HTML ridotto senza parentesi angolari e si rimanda alla documentazione per approfondire (http://slim-lang.com/).
Come si vede si sono usati i metodi helper per generare i *tag* dell'ancora e dell'immagine.

Ci manca il layout.

```
doctype html
html
  head
    title Web Page
    link rel="stylesheet" href="/stylesheets/style.css" type="text/css" media="all"
  body
    div#header
    div#content
      == yield
    div#footer Copyright © #{year} #{author}
```

sorgente

Questo scheletro verrà riempito con il template nel punto in cui si trova `yield` renderizzando il tutto e servendolo al browser.

Per ultimo abbiamo il file dello stile CSS.

```
body {
  font-size: 16px;
  font-family: "Arial";
  background-color: #cccccc;

  a {
    color: #ffffff;
    font-weight: bold;
    background-color: #0076ff;
    padding: 0.6em;
    border: 1px solid #0226ff;

    &:hover {
      background-color: #0226ff;
    }
  }
}
```

sorgente
Possiamo poi fare un file di shell, renderlo eseguibile e lanciarlo.

```
#!/bin/bash

#giusto per essere sicuri che ci siano le gemme
bundle install

bundle exec ruby app.rb
```

o semplicemente lanciare la linea di comando indicata.
La struttura finale del progetto.

```
├── helpers
│   └── main_helpers.rb
├── public
│   ├── images
│   │   └── image.png
│   └── test.html
├── views
│   ├── layouts
│   │   └── layout.slim
│   ├── scss
│   │   └── style.scss
│   └── index.slim
├── app.rb
├── Gemfile
├── Gemfile.lock
└── start.sh
```

Conclusioni

Penso sempre molto a come concludere un discorso su **Ruby** ma poi alla fine non mi viene in mente altro che:

- Ruby è divertente.

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com