Actor Pattern in Ruby

FUNGOSTUDIOS

Agenda

- Introduzione a Ruby
- Celluloid
- DCell

Perchè Ruby?

- E' un linguaggio molto espressivo
- Flessibile
- Adatto a definire Domain Specific Language (DSL)
- Vasta disponibiltà di librerie (*gem*)

Introduzione a Ruby

Getting Started

Le versioni di Ruby

- Ruby è stato creato nel 1993
- Le versioni comunemente usate oggi sono quelle >= 1.9.2
- L'ulitma versione è la 2.0

Le versioni di Ruby

- Ruby è stato creato nel 1993
- Le versioni comunemente usate oggi sono quelle >= 1.9.2
- L'ulitma versione è la 2.0

```
matteo@fungo ~$ ruby -v
ruby 2.0.0p247 (2013-06-27 revision 41674) [x86_64-darwin12.3.0]
```

Installazione

- Windows
  - Ruby Installer http://rubyinstaller.org
- Linux e Mac OS X
  - rvm (Ruby Version Manager) http://rvm.io
  - CLI che permette di installare e gestire diverse versioni di Ruby nella stessa macchina

Installare Ruby con rvm

1. Installare rvm

```
matteo@fungo ~$ curl -L https://get.rvm.io | bash -s stable
```

2. Installere ruby 2.0

```
matteo@fungo ~$ rvm install ruby 2.0
```

```
matteo@fungo ~$ rvm use --default 2.0
```

3. Che versioni di Ruby sono state installate?

```
matteo@fungo ~$ rvm list
```

Eseguire codice Ruby interattivamente

1. Usare irb

```
matteo@fungo ~$ irb
2.0.0p247 :001 > puts 'Hello, World!'
Hello, World!
 => nil
2.0.0p247 :002 > exit
matteo@fungo ~$
```

Eseguire codice Ruby interattivamente

1. Usare l'interprete Ruby

```
matteo@fungo ~$ ruby
puts 'Hello, World!'
^D
Hello, World!
matteo@fungo ~$
```

Eseguire programmi Ruby

```
matteo@fungo ~$ cat hello_world.rb
puts 'Hello, World!'
puts "It's #{Time.now}"
matteo@fungo ~$ ruby hello_world.rb
Hello, World!
It's 2013-11-16 11:49:21 +0100
matteo@fungo ~$
```

In Ruby, tutto è un oggetto!

- No, really!

```
//Java code here
int i = 10;
Integer.toString(i);
```

In Ruby, tutto è un oggetto!

- No, really!

```
//Java code here
int i = 10;
Integer.toString(i);
```

```
//Ruby here
i = 10
i.to_s
-42.abs
```

In Ruby, tutto è un oggetto!

- No, really!

```
//Java code here
int i = 10;
Integer.toString(i);
```

```
//Ruby here
i = 10
i.to_s
-42.abs
```

- Anche `nil` è un oggetto!

```
2.0.0p247 :002 > nil.nil?
 => true
2.0.0p247 :003 > nil.class
 => NilClass
```

Class

- Per definire una classe si usa la keyword `class`

```
class Song
  # some code here
end
```

Class

- Per definire una classe si usa la keyword `class`

```
class Song
  # some code here
end
```

- Le variabili di istanza sono variabili che inizano con il carattere @

```
class Song
  @title = ''
end
```

Ruby.new

- Per creare una nuova istanza di un oggetto si utilizza il costruttore **new**
  - Chiama il metodo initialize

Ruby.new

- Per creare una nuova istanza di un oggetto si utilizza il costruttore **new**
  - Chiama il metodo initialize

```
class Song
  def initialize(title)
    @title = title
  end
end

first_song = Song.new('Foo')
second_song = Song.new('Bar')
```

Mandare messaggi agli oggetti

- Per invocare un metodo di un oggetto, in Ruby, si manda un messaggio
  - nome del metodo
  - opzionalmente dei parametri di cui il metodo necessita
- Quando un oggetto riceve un messaggio
  1. Cerca il metodo richiesto
  2. Se lo trova, esegue il codice corrispondente

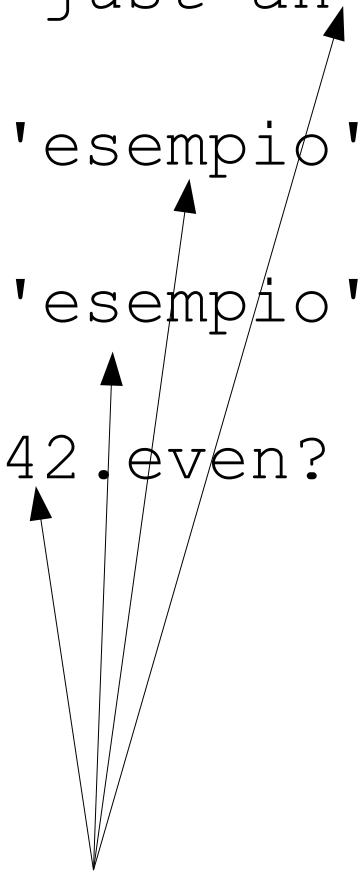
Mandare messaggi agli oggetti

```
2.0.0p247 :011 > 'just an example'.length
 => 15
2.0.0p247 :012 > 'esempio'.index('c')
 => nil
2.0.0p247 :013 > 'esempio'.index('p')
 => 4
2.0.0p247 :014 > 42.even?
 => true
2.0.0p247 :015 >
```

receiver.method(params)

Mandare messaggi agli oggetti

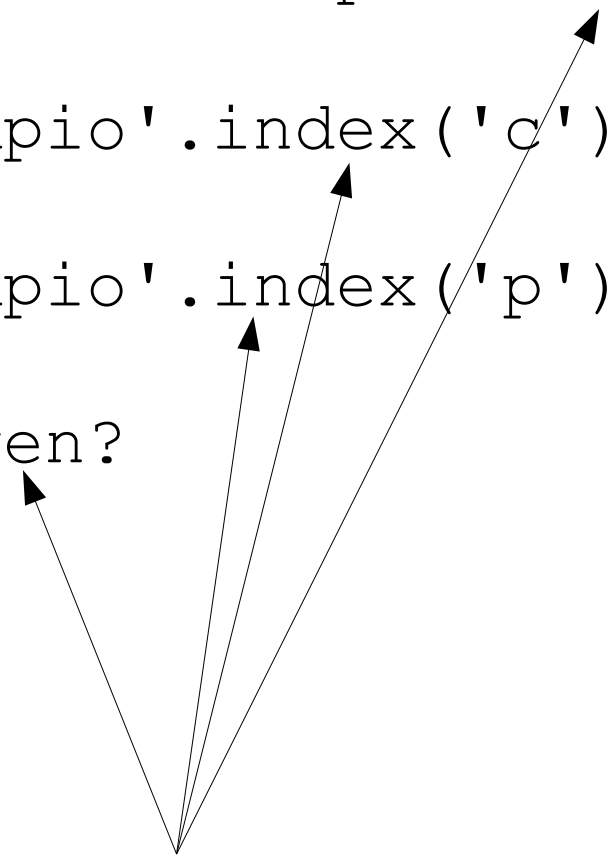
```
2.0.0p247 :011 > 'just an example'.length
 => 15
2.0.0p247 :012 > 'esempio'.index('c')
 => nil
2.0.0p247 :013 > 'esempio'.index('p')
 => 4
2.0.0p247 :014 > 42.even?
 => true
2.0.0p247 :015 >
```



receiver.**method**(**params**)

Mandare messaggi agli oggetti

```
2.0.0p247 :011 > 'just an example'.length
 => 15
2.0.0p247 :012 > 'esempio'.index('c')
 => nil
2.0.0p247 :013 > 'esempio'.index('p')
 => 4
2.0.0p247 :014 > 42.even?
 => true
2.0.0p247 :015 >
```



receiver.**method**(**params**)

Definire un metodo

```
matteo@fungo ~$ ruby
def say_hello(name)
  result = 'Hello, ' + name
  return result
end

puts say_hello('John')
puts say_hello('Mary')
^D
Hello, John
Hello, Mary
```

Definire un metodo

```
matteo@fungo ~$ ruby
def say_hello(name)
  result = 'Hello, ' + name
  return result
end

puts say_hello('John')
puts say_hello('Mary')
^D
Hello, John
Hello, Mary
```

puts e say_hello sono due metodi

- le parentesi per passare i parametri sono opzionali
- usarle sempre, tranne nei casi più semplici (puts)

Ereditarietà

- Esistono due modi di estendere le classi
  - Ereditarietà

    ```
    class Foo < Bar
    ```

  - Mix-in

    includo un modulo all'interno della classe

Moduli

- I moduli in Ruby sono un modo per
  - Ragruppare metodi, classi e costanti in un namespace
  - Estendere le funzionalità di un classe attraverso l'uso del mix-in
- Un modulo NON si istanzia! NON è una classe, è solo un contenitore

Namespace

- Progetti software di dimensioni discrete hanno la necessità di evitare collisioni sui nomi
  - I moduli permetto di definire un namespace, dando la possibiltà di disambiguare nomi che altrimenti sarebbero identici
- Un modulo si definisce con la keyword `module`

```
module MyModule
  # code here
end
```

Mix-in

- I moduli possono essere usati anche per estendere una classe
  - Non è molto diverso dall'ereditarietà
  - Ma non c'e' nessun legame di “parentela” tra modulo e classe
- Per estendere una classe devo includere un modulo

```
class MyClass
  include MyModule
  # ...
end
```

Mix-in esempio

- Definisco un modulo Debug

```
module Debug
  def me
    "#{self.class.name}: “ +
      ID ##{self.object_id} #{self.name}"
  end
end
```

Mix-in esempio

- Definisco un modulo Debug

```
module Debug
  def me
    "#{self.class.name}: " +
      ID ##{self.object_id} #{self.name}"
  end
end
```

```
class AClass
  include Debug
  attr_reader :name

  def initialize(name)
    @name = name
  end
end
```

Mix-in esempio

- De

```
mod
  d

  e
end
```

```
class AnotherClass
  include Debug
  attr_accessor :name
end

 > a = Aclass.new('a-team')
 > b = AnotherClass.new
 > b.name = 'Magnum PI'
 > a.me
 => "AClass: ID #70136434336820 a-team"
 > b.me
 => "AnotherClass: ID #70136434371260 Magnum PI"
```

```
class Aclass
  include Debug
  attr_reader :name

  def initialize(name)
    @name = name
  end
end
```

Welcome to the Ruby World

Actor Pattern in Ruby

https://github.com/celluloid/celluloid

Le 4 regole d'ora sulla concorrenza

1. Don't do it
2. If you must do it, don't share data across threads.
3. If you must share data across threads, don't share mutable data.
4. If you must share mutable data across threads, synchronize access to that data.

Celluloid

- Programmazione concorrente: OOP + primitive di sincronizzazione
- Celluloid combina i due aspetti
  - Si occupa di gestire la parte “difficile”
  - Non dovremo gestire esplicitamente la sincronizzazione tra diversi thread
  - Viene tutto astratto con il modello degli Actor

Actor

- Pensiamo il nostro sistema in termini di componenti concorrenti
  - Ogni componente è un thread
  - Ogni componente può comunicare con il mondo esterno ricevendo e mandando messaggi
  - Lo stato del componente non è mai condiviso con il resto del sistema!
- Meccanismi di Fault Tollerance

Chuck Norris

```
require 'celluloid'

class Chuck
  include Celluloid
  def initialize(name)
    @name = name
  end

  def set_status(state)
    @status = state
  end

  def report
    "#{@name} is #{@status}"
  end
end
```

Chuck Norris

```
require 'celluloid'

class Chuck
  include C
  def initi
    @name =
  end

  def set_s
    @status
  end

  def report
    "#{@name} is #{@status}"
  end
end
```

```
> chuck = Chuck.new('Chuck Norris')
=> #<Celluloid::ActorProxy(Chuck:...>
> chuck.set_status('immortal')
=> "immortal"
> chuck.report
=> "Chuck Norris is immortal"
> chuck.async.set_status('async immortal')
=> nil
> chuck.report
=> "Chuck Norris is async immortal"
```

Linking

- Quando un actor lancia un eccezione (non gestita) muore

Linking

- Quando un actor lancia un eccezione (non gestita) muore

```
class JamesDean
  include Celluloid
  class CarInMyLaneError < StandardError; end

  def drive_little_bastard
    raise CarInMyLaneError,
      "that guy's gotta stop. he'll see us"
  end
end
```

Linking

- Quando un actor lancia un eccezione (non gestita) muore

```
class JamesDean
  include Celluloid
  class CarInMyLaneError < StandardError; end

  def drive_little_bastard
    raise CarInMyLaneError,
      "that guy's gotta stop. he'll see us"
  end
end
```

```
>> james = JamesDean.new
 => #<Celluloid::Actor(JamesDean:0x1068)>
>> james.async.drive_little_bastard
 => nil
>> james
 => #<Celluloid::Actor(JamesDean:0x1068) dead>
```

Linking

- Un actor può legarsi ad un altro per intercettare eventuali crash

Linking

- Un actor può legarsi ad un altro per intercettare eventuali crash

```
class ElizabethTaylor
  include Celluloid
  trap_exit :actor_died

  def actor_died(actor, reason)
    p "Oh no! #{actor.inspect} has died
      because of a #{reason.class}"
  end
end
```

Linking

- Un actor può legarsi ad un altro per intercettare eventuali crash

```
class ElizabethTaylor
  include Celluloid
  trap_exit :actor_died

  def actor_died(actor, reason)
    p "Oh no! #{actor.inspect} has died
      because of a #{reason.class}"
  end
end
```

```
>> james = JamesDean.new
>> elizabeth = ElizabethTaylor.new
>> elizabeth.link james
>> james.async.drive_little_bastard
 => nil
"Oh no! #<Celluloid::Actor(JamesDean:0x11b8) dead> has died
because of a JamesDean::CarInMyLaneError"
```

Ancora Linking

```
class PorscheSpider
  include Celluloid
  class CarInMyLaneError < StandardError; end

  def drive_on_route_466
    raise CarInMyLaneError, "head on collision :("
  end
end

class JamesDean
  include Celluloid

  def initialize
    @little_bastard = PorscheSpider.new_link
  end

  def drive_little_bastard
    @little_bastard.drive_on_route_466
  end
end
```

Ancora Linking

```
class PorscheSpider
  include Celluloid

>> james = JamesDean.new
 => #<Celluloid::Actor(JamesDean:0x1108)
      @little_bastard=#<Celluloid::Actor(PorscheSpider:0x10ec)>>
>> elizabeth = ElizabethTaylor.new
 => #<Celluloid::Actor(ElizabethTaylor:0x1144)>
>> elizabeth.link james
 => #<Celluloid::Actor(JamesDean:0x1108) ....)>>
>> james.async.drive_little_bastard
 => nil
"Oh no! #<Celluloid::Actor(JamesDean:0x1108) dead> has died
because of a PorscheSpider::CarInMyLaneError"

  end

  def drive_little_bastard
    @little_bastard.drive_on_route_466
  end
end
```

Supervisor

- Il supervisor è responsabile di una parte del sistema
  - Se questa fallisce, la riavvia
  - Posso creare gerarchie di attori, e gestirle mediante un supervisor

Supervisor - Esempio

```
class PorscheSpider
  include Celluloid
  class CarInMyLaneError < StandardError; end

  def drive_on_route_466
    raise CarInMyLaneError, "head on collision :("
  end
end

class ChuckNorris
  include Celluloid

  def initialize
    @little_bastard = PorscheSpider.new_link
  end

  def drive_little_bastard
    @little_bastard.drive_on_route_466
  end
end
```

Supervisor - Esempio

```
class PorscheS
  include Cel
  class CarIn

  def drive_o
    raise Car
  end
end

class ChuckNorris
  include Celluloid

  def initialize
    @little_bastard = PorscheSpider.new_link
  end

  def drive_little_bastard
    @little_bastard.drive_on_route_466
  end
end
```

```
class ChuckFan
  include Celluloid
  trap_exit :oh_chuck!

  def oh_chuck!(actor, reason)
    p "OMG! Is #{@actor.inspect} still alive?"
  end
end
```

Supervisor - Esempio

```
class PorscheSpider
  include Celluloid
```

```
>> ChuckNorris.supervise_as :chuck
>> chuck = Celluloid::Actor[:chuck]
>> fan = ChuckFan.new
>> fan.link chuck
>> chuck.async.drive_little_bastard
 => nil
>> E, [2013-11-27T13:08:20.396379 #2656] ERROR -- : Porsche..
PorscheSpider::CarInMyLaneError: head on collision :(
E, [2013-11-27T13:08:20.396633 #2656] ERROR -- : ChuckNorris crashed!
PorscheSpider::CarInMyLaneError: head on collision :(
"OMG! Is nil still alive?"
>> chuck
 => #<Celluloid::ActorProxy(ChuckNorris) dead>
>> Celluloid::Actor[:chuck]
 => #<Celluloid::ActorProxy(ChuckNorris:0x3fe2f202a584
      @little_bastard=#<Celluloid::ActorProxy(PorscheS...>>
```

DCell

https://github.com/celluloid/dcell

DCell

- DCell is a simple and easy way to build distributed applications in Ruby
- DCell is a distributed extension to Celluloid

Dipendenze

- DCell si basa su 0mq
  - Dobbiamo installare le librerie di 0mq
    - Es: `brew install zeromq`
- DCell ha bisogno di mantenere uno stato globale del sistema nel ***registro***
  - Redis (default)
  - Zookeeper
  - Cassandra

Getting Started

- Dcell si compone di una rete di nodi
  - Ogni host può contenere più di un nodo
  - Ogni nodo è identificato da un ID (default hostname)
  - Ogni nodo deve essere raggiungibile via 0mq

```
DCell.start :id => "node42", :addr =>
"tcp://127.0.0.1:2042"
```

- I nodi di uno stesso cluster condividono lo stesso registro e la stessa directory

Getting Started

- Dcell si compone di una rete di nodi
  -
  -
  -

```
DCell.start(:id => id, :addr => addr)

Clock.supervise_as :clock
Orologio.supervise_as :orologio
node = DCell::Node[id]
DCell::Global[:clock] = node[:clock]
DCell::Global[:orologio] = node[:orologio]
```

- I nodi di uno stesso cluster condividono lo stesso registro e la stessa directory

dcell-web-time

- Progetto esempio by FunGo Studios
  - Credits: Giovanni Cappellotto e Matteo Centenaro

https://github.com/FunGoStudios/dcell-web-time

FUNGOSTUDIOS

Thank You!

FUNGOSTUDIOS

matteo@playgowar.com

nicola@fungostudios.com