DI is for us?

.。oO(さっちゃんですよヾ(〃l _ l)ﾉ ☆)
#多言語使用者

DI : 依存性の注入

DI (Dependency injection) ⊂ Dependency inversion principle

SOLI**D**

Dependency inversion principle

- High level modules should not depend upon low level modules. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend on abstractions.

https://web.archive.org/web/20150905081103/http://www.objectmentor.com/resources/articles/dip.pdf

SOLID

依存性逆転の原則

- 上位レベルのモジュールは下位レベルのモジュールに依存すべきではない。両方とも抽象に依存すべきである。
- 抽象は詳細に依存してはならない。詳細が抽象に依存すべきである。

https://web.archive.org/web/20150905081103/http://www.objectmentor.com/resources/articles/dip.pdf

SOLID

Dependency inversion principle

```
def do_detail_1
  if ENV["APP_ENV"] == "test"
    do_as_test
  else
    do_as_prod
  end
end

def do_detail_2
  if ENV["APP_ENV"] == "test"
    do_as_test
  else
    do_as_prod
  end
end
```

SOLID

Dependency inversion principle

```ruby
def runner
  if ENV["APP_ENV"] == "test"
    do_as_test
  else
    do_as_prod
  end
end

def do_detail_1; runner; end

def do_detail_2; runner; end
```

SOLID

Dependency inversion principle

```
class TestRunner
  def run; end
end

class ProdRunner
  def run; end
end

def runner
  h = Hash.new(-> { ProdRunner.new })
  h["test"] = -> { TestRunner.new }
  h[ENV["APP_ENV"]].()
end

def do_detail_1; runner.run; end

def do_detail_2; runner.run; end
```

SOLI**D**

Dependency inversion principle

That is normal *refactoring* in Ruby.

DI (Dependency injection) @ OOP

like PHP.

There is *DI container*.

```
$c = new Container();
if ($_ENV['APP_ENV'] == 'test') {
  $c['runner'] = $c->factory(function ($c) {
    return new TestRunner();
  });
} else {
  $c['runner'] = $c->factory(function ($c) {
    return new ProdRunner();
  });
}

$c['runner']->run();
```

https://github.com/Ranyuen/Di

What is *DI container*?

Functions of DI container

- Manage to generate objects.
- Inject dependent objects.
- Resolve a dependency graph.

Functions of DI container

- Manage to generate objects.

Singleton.

```
$c['momonga'] = new Momonga();
assert($c['momonga'] === $c['momonga']);
```

```
$c['momonga'] = function ($c) { return new Momonga(); }
assert($c['momonga'] === $c['momonga']);
```

```
$c['momonga'] = new Momonga();
$c->facade('M', 'momonga');
assert($c['momonga'] === M);
```

Functions of DI container

- Manage to generate objects.

Factory.

```
$c['momonga'] = $c->factory(function ($c) {
  return new Momonga();
});
assert($c['momonga'] !== $c['momonga']);
```

Functions of DI container

- Inject dependent objects.

```php
class Momonga {
  /** @inject */
  public $name;
}

$c['name'] = '百々ん蛾';
$momonga = $c->newInstance('Momonga', []);
assert('百々ん蛾' === $momonga->name);
```

Functions of DI container

- Resolve a dependency graph.

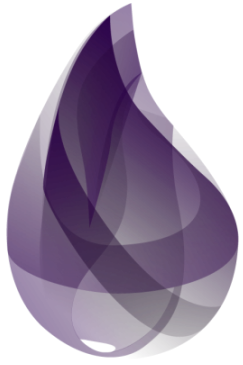
```
class Song {
  /** @inject */
  public $code;
}
class Momonga {
  /** @inject */
  public $song;
  public function sing() {
    var_dump($this->song->code);
  }
}

$c['code'] = 'CM7';
$c['song'] = $c->factory(function ($c) {
  return $c->newInstance('Song', []);
});
$c->newInstance('Momonga', [])->sing();
```

`Momonga` needs `'song'` needs `'code'`.

DI (Dependency injection) @ FP

like Elixir.



Get through args.

```
defmodule Main do
  def main(runner), do: runner.()
end

runner =
  if "test" == System.get_env("APP_ENV"),
    do: TestRunner,
    else: ProdRunner
Main.main(runner)
```

`Process.register/2`.

```
defmodule Runner do
  def start do
    spawn fn ->
      Process.register(self(), __MODULE__)
      loop()
    end
    Process.sleep(1)
  end

  defp loop do
    receive do
      {:run, from} -> send(from, {:ok, :momonga})
    end
    loop()
  end
end

Runner.start

# …
```

`Process.register/2`.

```
# …

defmodule Main do
  def main do
    send(Runner, {:run, self()})
    v = receive do: ({:ok, v} -> v)
    IO.inspect(v)
  end
end

Main.main
```

1. Function args is not composable.
2. `Process.register/2` makes singleton only, not composable & the performance isn't good.

I want to composable effect handlers, as data.

Algebraic effects.

```
defmodule Runner do
  @behaviour Context

  @impl Context
  def init(args), do: args

  @impl Context
  def handle(:run, context, state),
    do: {:reply, :momonga, state}
  def handle(_, _, _), do: :ignore
end

defmodule Main do
  def main(context) do
    {v, context} = Context.perform(context, :run)
    IO.inspect(v)
  end
end

ctx = %Context{} |> Context.add(Runner, nil)
Main.main(ctx)
```

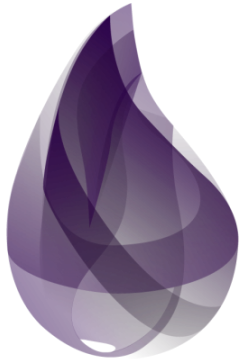
Algebraic effects.

```
defmodule Song do
  def init(code), do: %{code: code}
  def handle({__MODULE__, :code}, _, state),
    do: {:reply, state.code, state}
  def handle(_, _, _), do: :ignore
end

defmodule Momonga do
  def init(_), do: nil
  def handle(:sing, context, state) do
    {code, context} =
      Context.perform(context, {Song, :code})
    IO.inspect(code)
    {:replay, nil, state, context}
  end
  def handle(_, _, _), do: :ignore
end

ctx =
  %Context{}
  |> Context.add(Momonga, nil)
  |> Context.add(Song, "CM7")
Context.perform(context, :sing)
```

Both in PHP & Elixir, DI should



- be composable. (Injection)
- manage a state of data. (Factory)
- reolve a dependency graph. (Container)

```
def runner
  h = Hash.new(-> { ProdRunner.new })
  h["test"] = -> { TestRunner.new }
  h[ENV["APP_ENV"]].()
end

def do_detail_1; runner.run; end

def do_detail_2; runner.run; end
```

- ~~be composable.~~
- manage a state of data.
- ~~reolve a dependency graph.~~

Create a DI container?

```
c = Container.new do |c|
  c[:a] = 42
  c[:b] = B.new(c[:a])
  c.factory(:b_new) {|c| B.new(c[:a]) }
end
```

http://c4se.hatenablog.com/entry/2015/05/03/004218

- ~~be composable.~~
- manage a state of data.
- reolve a dependency graph.

`instance_eval`.

```
class Momonga
  def sing; p @code; end
end

C = Struct.new(:_) do
  def code; 'CM7'; end

  def momonga
    __c = self
    Momonga.new.tap do |m|
      m.instance_eval { @code = __c.code }
    end
  end
end
C.new.momonga.sing
```

- be composable.
- manage a state of data.
- reolve a dependency graph.

`define_singleton_method`.

```
class Momonga
  def sing; p code; end
end

C = Struct.new(:_) do
  def code; 'CM7'; end

  def momonga
    __c = self
    Momonga.new.tap do |m|
      m.define_singleton_method(:code) { __c.code }
    end
  end
end
C.new.momonga.sing
```

- be composable.
- manage a state of data.
- reolve a dependency graph.

`method_missing`.

```
module C
  def method_missing(name, *args); Ctx[name][]; end
end

class Momonga
  include C
  def sing; p code; end
end

C.const_set('Ctx', {
  code: -> { 'CM7' },
  momonga: -> { Momonga.new },
})
c = Struct.new(:_) { include C }.new
c.momonga.sing
```

- be composable.
- manage a state of data.
- reolve a dependency graph.

`TracePoint`.

```
class Momonga
  def sing; p code; end
end

C = Struct.new(:_) do
  def code; 'CM7'; end
  def momonga; Momonga.new; end
end
__c = C.new
TracePoint.trace(:call, :c_call) do |tp|
  next unless tp.method_id == :initialize
  case tp.self
  when Momonga
    tp.self.define_singleton_method(:code) { __c.code }
  end
end
__c.momonga.sing
```

- be composable.
- manage a state of data.
- reolve a dependency graph.

Ruby is dynamic☆

It's fun to re-think OOP, FP in Ruby dynamics.