Classbox入門

Shugo Maeda

2010-11-06

自己紹介

- 前田修吾
- Rubyコミッタ
- ネットワーク応用通信研究所取締役
- Rubyアソシエーション副理事長

本日のテーマ

- Classbox入門
  - のはずでしたが…

本日のテーマ

- Classboxの代りに新機構Refinementを提案

Classboxとは?

- クラスの拡張手段

Rubyのクラスの拡張方法

- サブクラス化
- mix-in
- 特異メソッド
- オープンクラス

サブクラス化

```
class Person
  attr_accessor :name
end

class Employee < Person
  attr_accessor :monthly_salary
end
```

サブクラス化の特徴

- 普通の単一継承
- サブクラスのインスタンスにのみ影響
- 単なる実装の継承手段
  - LSP違反が容易

LSP

- Liskov Substitution Principle
  - リスコフの置換原則
- サブタイプのインスタンスは、スーパータイプのインスタンスと同じように振る舞わなければならない
  - スーパータイプのインスタンスは、サブタイプのインスタンスで置換可能

LSPの例

```
def print_name(person)
  puts person.name
end

shugo = Person.new
shugo.name = "Shugo Maeda"
print_name(shugo) #=> Shugo Maeda
matz = Employee.new
matz.name = "Yukihiro Matsumoto"
print_name(matz) #=> Yukihiro Matsumoto
```

典型的なLSP違反

```
class Rectangle
  attr_accessor :width, :height
end

class Square < Rectangle
  def set_size(x) @height = @width = x end
  alias width= set_size
  alias height= set_size
end

def set_size(rect)
  rect.width = 80; rect.height = 60
end
square = Square.new
set_size(square)
p square.width #=> not 80, but 60!
```

RubyのLSP違反

```ruby
class Employee < Person
  undef name
end

def print_name(person)
  puts person.name
end

matz = Employee.new
matz.name = "Yukihiro Matsumoto"
print_name(matz) #=> undefined method `name'...
```

サブクラス化 != サブタイプ化

- サブクラス化は実装のみの継承
- Duck typing

mix-in

```
class; Stream; ... end
module Readable; ... end
module Writable; ... end

class ReadStream < Stream
  include Readable
end
class WriteStream < Stream
  include Writable
end
class ReadWriteStream
  include Writable, Readable
end
```

mix-inの特徴

- 限定された多重継承
  - モジュールのみ複数継承可能
  - モジュールはインスタンスを持てない
- モジュールは定数などの名前空間としても利用される

特異メソッド

```
matz = Person.new
def matz.design_ruby
  ...
end
matz.design_ruby
shugo = Person.new
shugo.design_ruby #=> NoMethodError
```

特異メソッドの特徴

- クラスの利用者側で、クラスの機能を拡張できる
- 特定のインスタンスの挙動のみを変える
- 一部のオブジェクトには定義できない
  - Integerインスタンスなど

オープンクラス

```
# Personを再オープンし、コードを追加
class Person
  attr_accessor :age
end
shugo = Person.new
shugo.name = "Shugo Maeda"
shugo.age = 34
```

オープンクラスの特徴

- クラスの利用者側で、クラスの機能を拡張できる
- クラスをグローバルに拡張する

応用例

- Ruby on Rails
  - ActiveSupport
  - プラグイン
- jcode
- mathn

LSPとオープンクラス

- s/サブタイプ/オープン後のクラス/g
- s/スーパータイプ/オープン前のクラス/g
- オープン後のクラスのインスタンスは、オープン前のクラスのインスタンスと同じように振る舞うべき

LSP違反

```
p 1 / 2 #=> 0
require "mathn"
p 1 / 2 #=> (1/2)
```

まとめ

サブクラス化	利用側で拡張できない
mix-in	利用側で拡張できない
特異メソッド	オブジェクト単位
オープンクラス	影響がグローバル

拡張性とモジュール性

- サブクラス化・mix-in・特異メソッドは拡張性が低い
- オープンクラスはモジュール性が低い

必要なもの

- 利用側で拡張でき…
- クラス単位の…
- 影響が局所的な…
- クラス拡張

解決策

- selector namespace
- Classbox

selector namespace

- SmallScriptやECMAScript 4の機能
- メソッド名（selector）の名前空間
- 名前空間を他の名前空間にimportできる
- importの影響範囲は静的に決まる

Classbox

- SqueakやJavaの拡張機能
- classboxという単位でクラスを定義・拡張
- classboxは他のclassboxにimportできる
- importの影響範囲は動的に決まる
  - local rebinding

なぜRubyKaigiで勘違いしたか?

- どこかでClassboxはレキシカルだと聞いたような…
- 「重要な違いはrefineによって加えた修正の有効範囲はそのClassboxに**レキシカル**に限定されるということ。」
  - Matzにっき 2006-01-04
  - http://www.rubyist.net/~matz/20060104.html

Classbox/Jの例

```
package Foo;
public class Foo { ... }

package Bar;
import Foo;
refine Foo { public void bar() { ... } }

package Baz;
import Bar;
public class Baz {
  public static void main(String[] args) {
    new Foo().bar();
  }
}
```

local rebindingは必要?

- モジュール性の低下
  - 呼出し先は元の有効挙動を期待している可能性
  - mathnのような用途では危険
- 特異メソッドやオープンクラスで代替可能
  - ただし、影響範囲は違う

Refinement

- 今回新しく実装した機能
- モジュール単位でクラス拡張を定義
- 明示的に有効範囲を指定
- local rebindingはない
- 構文はClassbox/Jに近い

構文

```
module MathN
  refine Fixnum do
    def /(other) quo(other) end
  end
end

class Foo
  use MathN

  def bar
    p 1 / 2 #=> (1/2)
  end
end
p 1 / 2 #=> 0
```

Module#refine

- refine(klass, &block)
- klassへのメソッド定義をblock内で行う
- selfのモジュール内と、そのモジュールをuseした範囲でのみ有効
- クラスに対しても呼出し可能

クラスローカル拡張

```
class Foo
  refine Fixnum do
    def /(other) quo(other) end
  end

  def bar
    p 1 / 2 #=> (1/2)
  end
end
p 1 / 2 #=> 0
```


Kernel#use

- use(mod)
- modでrefineされた機能を取り込む
- useが呼び出されたファイル、モジュール、クラス、メソッド内でのみ有効

useの例

```
use A # このファイル内で有効
module Foo
  use B # Foo内(Barも含む)で有効

  class Bar
    use C # Foo::Bar内で有効
  end
end
```


Module#use

- use(mod)
- Kernel#useと同じ機能
- + 再オープン・継承のサポート

Module#useの例

```
module A; refine(X) { ... } end
module B; refine(X) { ... } end
class Foo; use A end
class Foo
  # 再オープン時もAが有効
end
module Bar
  use B
  class Baz < Foo
    # サブクラスでもAが有効
    # BよりもAが優先される
  end
end
```


useとinclude

```
module A; refine(X) { ... } end
module Foo; use A end
class Bar
  include Foo
  # includeしてもAは有効にならない
end
```


優先順位

- 後でuseされたものが優先される
- サブクラスでuseされたものが優先される
- 外側のクラス・モジュールでuseされたものより、現在のクラスやそのスーパークラスでuseされたものが優先される
- refineされたクラスのサブクラスを定義した場合、サブクラスのメソッドの方がrefineで定義されたメソッドより優先される

優先順位の例

```
class Foo; end
module Bar; refine Foo do end end
module Baz; refine Foo do end end
class Quux < Foo; end
class Quuux
  use Bar
end
module Quuuux
  use Baz
  class  Quuuuux < Quuux
    def foo
      # Quux -> Bar -> Baz -> Foo
      Quux.new.do_something
    end
  end
end
```

refine前の機能の利用

- refine内でのsuperはrefine前のメソッドを呼ぶ
- 他のrefinementが前にuseされている場合は、refine前の定義よりもそのrefinementの定義が優先される
- refineされるクラスの定数やクラス変数もアクセス可能

superの例

```
module FloorExtension
  refine Float do
    def floor(d=nil)
      if d
        x = 10 ** d
        return (self * x).floor.to_f / x
      else
        return super()
      end
    end
  end
end
use FloorExtension
p 1.234567890.floor #=> 1
p 1.234567890.floor(4) #=> 1.2345
```


*_eval

- {instance,module,class}_evalの中では、useされたモジュールのrefinementが有効になる

*_evalの例

```
class Foo
  use MathN
end
Foo.class_eval do
  p 1 / 2 #=> (1/2)
end
Foo.new.instance_eval do
  p 1 / 2 #=> (1/2)
end
```


互換性

- 文法の変更はなし
- Refinementを使わないコードは今まで通り動作
- 既存のコードでrefineやuseという名前のメソッドを定義している場合は問題が起こる可能性がある

応用例

- 標準クラスの拡張
- 内部DSL
- メソッドのネスト

標準クラスの拡張

- 拡張が有効な範囲を限定
- フレームワークが提供するスーパークラスで useする

例

```
module ActiveRecord
  class Base
    use ActiveSupport
    ...
  end
end

class Account < ActiveRecord::Base
  def has_name?
    return !name.blank?
  end
end
```


内部DSL

- DSL用のメソッドの有効範囲を限定
- instance_evalやmodule_evalを利用

例

```
module Spec
  module Expectations
    refine Object do
      def should ... end
      ...
    end
  end
end

def it(msg, &block)
  Spec::Expectations.module_eval(&block)
end

it "returns 0 for all gutter game" do
  bowling = Bowling.new
  20.times { bowling.hit(0) }
  bowling.score.should == 0
end
```


メソッドのネスト

```
def fact(n)
  # fact_iterはfactの中だけで有効なrefinementに定義される
  def fact_iter(product, counter, max_count)
    if counter > max_count
      product
    else
      fact_iter(counter * product, counter + 1, max_count)
    end
  end

  fact_iter(1, 1, n)
end
```


ベンチマーク

- make benchmarkを1回だけ実行
- 環境
  - CPU: Intel Core 2 Duo U7600 1.2GHz
  - メモリ: 2MB
  - OS: Linux 2.6.34 (Ubuntu 10.04)

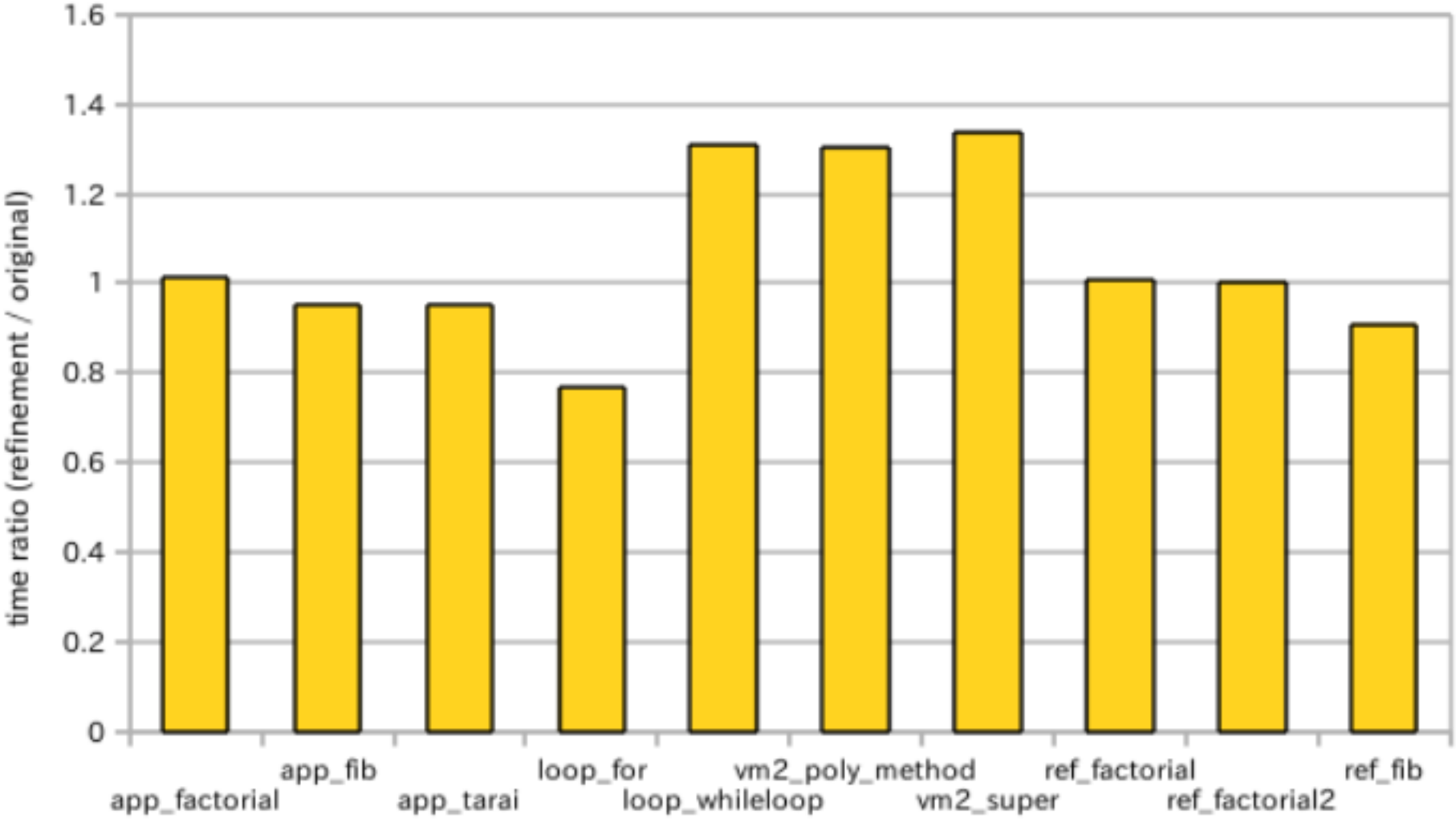
追加ベンチマーク

- ref_factorial
- ref_fib
  - refineを使用
- ref_factorial2
  - ネストしたメソッドを使用

ベンチマーク結果

- 実行時間が若干増加
  - 平均3%程度

抜粋


time ratio (refinement / original)
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
app_factorial
app_fib
app_tarai
loop_for
loop_whileloop
vm2_poly_method
vm2_super
ref_factorial
ref_factorial2
ref_fib

今後の課題

- includeとuseの分離は必要?
  - includeにuse相当の機能を追加してもよい?
- use or using?
- モジュールのrefine
- 実装の改善

まとめ

- 拡張性とモジュール性を両立させる新機構Refinementを提案しました

ご静聴ありがとうございました

59