Ruby 初級者向けレッスン第 5 回

かずひこ@株式会社 ネットワーク応用通信研究所

2006 年 1 月 28 日

Ruby で学ぼう継承と委譲

- 継承のおさらい
- 継承の問題
- 委譲について
- 委譲の仕組み

継承のおさらい

- スーパークラスにできることを継承する

```
class A
  def hello
    puts 'hello'
  end
end

class B < A # B クラスは A クラスを継承
end

B.new.hello
```

キューを作ろう

- 先に入ったものから先に出る（FIFO）リスト。要するに待ち行列。

initialize
: 空のキューを作る

enq(x)
: キューの最後に x を追加する

deq
: キューの最初の要素を取り除く

peek
: キューの最初の要素を返す（キューは変更しない）

length
: キューの要素の数を返す

empty?
: キューが空なら真、空でなければ偽

- 実行例

```
q = MyQueue.new
q.enq(1)      # 後ろに 1 を追加
q.enq(4)      # 後ろに 4 を追加
q.enq(2)      # 後ろに 2 を追加
q.peek #=> 1 # 先頭を問い合わせる
q.deq         # 先頭を取り除く
q.peek #=> 4 # 先頭を問い合わせる
```

キューと Array の違い

- length と empty?は Array の同名メソッドと同じ
- enq は Array#push と同じ
- deq は Array#shift と同じ
- peek は Array#first と同じ

キューのテストコード

- test_queue.rb

```
require 'myqueue1' # ← ここを'myqueue2'や'myqueue3'に変える
require 'test/unit'

class TestMyQueue < Test::Unit::TestCase
  def setup
    @queue = MyQueue.new
  end

  def test_empty?
    assert(@queue.empty?, 'a new queue is empty.')
```

```
  end

  def test_enq_and_peek
    @queue.enq(3)
    assert_equal(3, @queue.peek, 'peek returns the first value.')
  end

  def test_enq_and_length
    @queue.enq(3)
    assert_equal(1, @queue.length, 'enq increments the length.')
    @queue.enq(5)
    assert_equal(2, @queue.length, 'enq increments the length.')
  end

  def test_enq_and_empty?
    @queue.enq(3)
    assert_equal(false, @queue.empty?, 'a queue with data is not empty.')
  end

  def test_enq_enq_deq_and_length
    @queue.enq(3)
    @queue.enq(5)
    @queue.deq
    assert_equal(1, @queue.length, 'deq decrements the length.')
  end

  def test_enq_enq_deq_and_peek
    @queue.enq(3)
    @queue.enq(5)
    assert_equal(3, @queue.peek, 'peek returns the first value.')
    @queue.deq
    assert_equal(5, @queue.peek, 'peek returns the first value.')
  end
end
```

継承で作ろう（例 1）

- Array を継承して足りないメソッドを定義する（myqueue1.rb）。

```
class MyQueue < Array
  def enq(x)
    self.push(x)
  end
```



```
  def deq
    self.shift
  end

  def peek
    self.first
  end
end
```

継承で作ろう（例 2）

- わざわざ定義しなくても別名で済む（myqueue2.rb）。

```
class MyQueue < Array
  alias :enq :push
  alias :deq :shift
  alias :peek :first
end
```

継承の問題

- 必要のないこともできてしまう。

```
q = MyQueue.new
q.enq(1)
q.enq(2)
q.enq(3)
p q.peek #=> 1
p q[1] #=> 2    # キューの仕様外
p q.last #=> 3  # キューの仕様外
```

- スーパークラスとの結び付きが密接になる。
- スーパークラスの内部構造の変化に追従する必要がある。
- スーパークラスとメソッド名やインスタンス変数名が重複するとややこしいことになる。
- 「継承は最後の武器だ……それじゃ忍者部隊月光か」by まつもとさん

委譲とは？

- あるクラスの全てを「持ってくる（継承する）」のではなく、必要なものだけを「相手にお願い（委譲）」する。
- 委譲先はスーパークラスでなくてよいので、関係の弱いクラス同士で使える。

キューの処理の委譲

- length と empty?は Array の同名メソッドに委譲
- enq は Array#push に委譲
- deq は Array#shift に委譲
- peek は Array#first に委譲

委譲のためのライブラリ forwardable.rb

- 以下の二つのモジュールを提供する（今回とりあげるのは前者）。

Forwardable
: クラスに対してメソッドの委譲機能を定義するモジュール

SingleForwardable
: オブジェクトに対してメソッドの委譲機能を定義するモジュール

委譲で作ろう

- myqueue3.rb

```
require 'forwardable'

class MyQueue
  extend Forwardable

  def initialize
    @q = [] # 委譲するオブジェクトの準備
  end

  # 同名メソッドへの委譲
  def_delegators(:@q, :length, :empty?)

  # 別名メソッドへの委譲
  def_delegator(:@q, :push, :enq)
  def_delegator(:@q, :shift, :deq)
  def_delegator(:@q, :first, :peek)
end
```

- forwardable.rb のロード

```
require 'forwardable'
```


- MyQueue「クラス」に Forwardable モジュールのメソッドを追加

```
extend Forwardable
```

- 処理の委譲先に@q という Array を準備する

```
def initialize
  @q = [] # 委譲するオブジェクトの準備
end
```

- length メソッドと empty?メソッドは@q に委譲

```
def_delegators(:@q, :length, :empty?)
```

- enq メソッドは@q の push メソッドに委譲

```
def_delegator(:@q, :push, :enq)
```

- deq メソッドは@q の shift メソッドに委譲

```
def_delegator(:@q, :shift, :deq)
```

- peek メソッドは@q の first メソッドに委譲

```
def_delegator(:@q, :first, :peek)
```

Forwardable モジュールのメソッド

def_instance_delegators(accessor, *methods)
: methods で渡されたメソッドのリストを accessor に委譲する（別名 def_delegators）

def_instance_delegator(accessor, method, ali = method)
: メソッド ali が呼ばれた時に accessor に対し method を呼び出す（別名 def_delegator）

演習問題：スタックを作ろう

- 後に入ったものから先に出る（LIFO）リスト。皿を積んで上から順に取る感じ。

initialize
: 空のスタックを作る

push(x)
: スタックの最後に x を追加する

pop
: スタックの最後の要素を取り除く

peek
: スタックの最後の要素を返す（スタックは変更しない）

length
: キューの要素の数を返す

empty?
: スタックが空なら真、空でなければ偽

- 実行例

```
q = MyStack.new
q.push(1)    # 後ろに 1 を追加
q.push(4)    # 後ろに 4 を追加
q.push(2)    # 後ろに 2 を追加
q.peek #=> 2 # 末尾を問い合わせる
q.pop        # 末尾を取り除く
q.peek #=> 4 # 末尾を問い合わせる
```

- スタックを Forwardable モジュールによる委譲を使って実装しよう
- できればテスト・ファーストで実装しよう

委譲の仕組みを見よう

- forwardable.rb を読もう
  - /usr/lib/ruby/1.8/forwardable.rb (UNIX)
  - C:\ruby\lib\ruby\1.8\forwardable.rb (WindowsXP)
- Forwardable モジュールはわずか 28 行

```
# = forwardable - Support for the Delegation Pattern
#
#    $Release Version: 1.1$
#    $Revision: 1.2.2.1 $
#    $Date: 2005/09/26 13:59:45 $
#    by Keiju ISHITSUKA(keiju@ishitsuka.com)
#
#    Documentation by James Edward Gray II and Gavin Sinclair
#
# == Introduction
#
# This library allows you delegate method calls to an object, on a method by
# method basis.  You can use Forwardable to setup this delegation at the class
```



```
# level, or SingleForwardable to handle it at the object level.
#
# == Notes
#
# Be advised, RDoc will not detect delegated methods.
#
# <b>forwardable.rb provides single-method delegation via the
# def_delegator() and def_delegators() methods.  For full-class
# delegation via DelegateClass(), see delegate.rb.</b>
#
# == Examples
#
# === Forwardable
#
# Forwardable makes building a new class based on existing work, with a proper
# interface, almost trivial.  We want to rely on what has come before obviously,
# but with delegation we can take just the methods we need and even rename them
# as appropriate.  In many cases this is preferable to inheritance, which gives
# us the entire old interface, even if much of it isn't needed.
#
#   class Queue
#     extend Forwardable
#
#     def initialize
#       @q = [ ]    # prepare delegate object
#     end
#
#     # setup prefered interface, enq() and deq()...
#     def_delegator :@q, :push, :enq
#     def_delegator :@q, :shift, :deq
#
#     # support some general Array methods that fit Queues well
#     def_delegators :@q, :clear, :first, :push, :shift, :size
#   end
#
#   q = Queue.new
#   q.enq 1, 2, 3, 4, 5
#   q.push 6
#
#   q.shift    # => 1
#   while q.size > 0
#     puts q.deq
#   end
```

```
#
#   q.enq "Ruby", "Perl", "Python"
#   puts q.first
#   q.clear
#   puts q.first
#
# <i>Prints:</i>
#
#   2
#   3
#   4
#   5
#   6
#   Ruby
#   nil
#
# === SingleForwardable
#
#    printer = String.new
#    printer.extend SingleForwardable        # prepare object for delegation
#    printer.def_delegator "STDOUT", "puts"  # add delegation for STDOUT.puts()
#    printer.puts "Howdy!"
#
# <i>Prints:</i>
#
#    Howdy!

#
# The Forwardable module provides delegation of specified
# methods to a designated object, using the methods #def_delegator
# and #def_delegators.
#
# For example, say you have a class RecordCollection which
# contains an array <tt>@records</tt>.  You could provide the lookup method
# #record_number(), which simply calls #[] on the <tt>@records</tt>
# array, like this:
#
#   class RecordCollection
#     extends Forwardable
#     def_delegator :@records, :[], :record_number
#   end
#
# Further, if you wish to provide the methods #size, #<<, and #map,
```



```
# all of which delegate to @records, this is how you can do it:
#
#   class RecordCollection
#     # extends Forwardable, but we did that above
#     def_delegators :@records, :size, :<<, :map
#   end
#
# Also see the example at forwardable.rb.
#
module Forwardable

  @debug = nil
  class<<self
    # force Forwardable to show up in stack backtraces of delegated methods
    attr_accessor :debug
  end

  #
  # Shortcut for defining multiple delegator methods, but with no
  # provision for using a different name.  The following two code
  # samples have the same effect:
  #
  #   def_delegators :@records, :size, :<<, :map
  #
  #   def_delegator :@records, :size
  #   def_delegator :@records, :<<
  #   def_delegator :@records, :map
  #
  # See the examples at Forwardable and forwardable.rb.
  #
  def def_instance_delegators(accessor, *methods)
    for method in methods
      def_instance_delegator(accessor, method)
    end
  end

  #
  # Defines a method _method_ which delegates to _obj_ (i.e. it calls
  # the method of the same name in _obj_).  If _new_name_ is
  # provided, it is used as the name for the delegate method.
  #
  # See the examples at Forwardable and forwardable.rb.
  #
```

```
  def def_instance_delegator(accessor, method, ali = method)
    accessor = accessor.id2name if accessor.kind_of?(Integer)
    method = method.id2name if method.kind_of?(Integer)
    ali = ali.id2name if ali.kind_of?(Integer)

    module_eval(<<-EOS, "(__FORWARDABLE__)", 1)
      def #{ali}(*args, &block)
begin
  #{accessor}.__send__(:#{method}, *args, &block)
rescue Exception
  $@.delete_if{|s| /^\\(__FORWARDABLE__\\):/ =~ s} unless Forwardable::debug
  Kernel::raise
end
      end
    EOS
  end

  alias def_delegators def_instance_delegators
  alias def_delegator def_instance_delegator
end

#
# The SingleForwardable module provides delegation of specified
# methods to a designated object, using the methods #def_delegator
# and #def_delegators.  This module is similar to Forwardable, but it works on
# objects themselves, instead of their defining classes.
#
# Also see the example at forwardable.rb.
#
module SingleForwardable
  #
  # Shortcut for defining multiple delegator methods, but with no
  # provision for using a different name.  The following two code
  # samples have the same effect:
  #
  #   single_forwardable.def_delegators :@records, :size, :<<, :map
  #
  #   single_forwardable.def_delegator :@records, :size
  #   single_forwardable.def_delegator :@records, :<<
  #   single_forwardable.def_delegator :@records, :map
  #
  # See the example at forwardable.rb.
  #
```



```
  def def_singleton_delegators(accessor, *methods)
    for method in methods
      def_singleton_delegator(accessor, method)
    end
  end

  #
  # Defines a method _method_ which delegates to _obj_ (i.e. it calls
  # the method of the same name in _obj_).  If _new_name_ is
  # provided, it is used as the name for the delegate method.
  #
  # See the example at forwardable.rb.
  #
  def def_singleton_delegator(accessor, method, ali = method)
    accessor = accessor.id2name if accessor.kind_of?(Integer)
    method = method.id2name if method.kind_of?(Integer)
    ali = ali.id2name if ali.kind_of?(Integer)

    instance_eval(<<-EOS, "(__FORWARDABLE__)", 1)
       def #{ali}(*args, &block)
begin
  #{accessor}.__send__(:#{method}, *args,&block)
rescue Exception
  $@.delete_if{|s| /^\\(__FORWARDABLE__\\):/ =~ s} unless Forwardable::debug
  Kernel::raise
end
       end
    EOS
  end

  alias def_delegators def_singleton_delegators
  alias def_delegator def_singleton_delegator
end
```

使用例を見よう

- http://raa.ruby-lang.org/gonzui/ **で、クラス名（**Forwardable**）やライブラリ名（**forwardable**）で検索してみよう。**

使用例（ウェブアプリケーションフレームワーク arrow）

- arrow/lib/arrow/applet.rb

```
### Add some Hash-ish methods for convenient access to FormValidator#valid.
class FormValidator
  unless method_defined?( :[] )
    extend Forwardable
    def_delegators :@form, *(Hash::instance_methods(false) - [:[], :[]=])

    def []( key )
      @form[ key.to_s ]
    end

    def []=( key, val )
      @form[ key.to_s ] = val
    end
  end
end
```

- '[]' と'[]=' 以外の全ての Hash のメソッドを委譲して、'[]' と'[]=' はキーを String にしてから参照するようにしている

その他の委譲のライブラリ

delegate.rb
: 明示的に指定したメソッドだけを委譲する forwardable と異なり、delegate はほとんどのメソッドを委譲する

まとめ

- 「継承は最後の武器」
- ライブラリのソースを読もう
- 使用例のソースを読もう

参考文献

『C Magazine 2002 年 8 月号 - なぁRuby を読もうじゃないか 第 7 回 delegate.rb と weakref.rb』

『Rubyist Magazine 0012 号 - 標準添付ライブラリ紹介 第 6 回 委譲』
: http://jp.rubyist.net/magazine/?0012-BundledLibraries

『Ruby リファレンスマニュアル - Forwardable』
: http://www.ruby-lang.org/ja/man/?cmd=view;name=Forwardable

今後の情報源

公式 Web サイト
: http://www.ruby-lang.org/

リファレンスマニュアル
: http://www.ruby-lang.org/ja/man/

日本 Ruby の会
: http://jp.rubyist.net/

Rubyist Magazine
: http://jp.rubyist.net/magazine/

ふぇみにん日記
: http://kazuhiko.tdiary.net/