# Rubyのリフレクション vol.3

大林一平

KMC/Dept. Math., Kyoto U.

2011/09/10

# Agenda

- 自己紹介
- 目標
- メタプログラミングについて
- Ruby のメタプログラミング
- クラスと特異クラス
- Module#define_method
- define_attr

# 自己紹介

- 大林一平
- 京大マイコンクラブ (KMC)
- 京都大学数学教室
- Ruby/SDL, RRSE
- るりま

# 目標

Ruby メタプログラミングに関する理解を深めること.
私が Ruby 会議 2011 での LT の内容が理解できる程度をめざす.
今回の話では Ruby のバージョンは 1.9.2 を想定している

# メタプログラミング

メタプログラミングとは，プログラムをプログラムすること．

# メタプログラミングの力

メタプログラミングで何ができるか？

- 抽象化
  - 通常の抽象化機構では実現できない種類の抽象化ができる
- 文法の拡張
  - 実現したいことをより「自然な書き方」で実現できる

# プログラムとは何か？

プログラム対象となるプログラムとは何か？

# プログラムとは何か？

プログラム対象となるプログラムとは何か？

- ソースコード
- AST
- コンパイル済み実行バイナリ or バイトコード
- 実行中のプログラム (実行時の状態)

- ソースコード
  - ソースコードジェネレータ
  - C のマクロ
- 構文木/AST
  - Lisp/Scheme のマクロ
  - camlp4
  - C++のテンプレートもかなり特殊&限定的ながらここに含まれる
- コンパイル済み実行バイナリ or バイトコード
  - Java 関連ではわりとある
- 実行時の状態
  - 動的な言語ではこれが使えることが多い

これらのうち複数を組合せる場合もある

また，実行前/実行時のいずれに実現するかでも分類できる

- ソースコード生成
  - 実行前に生成→普通のコードジェネレータ
  - 実行時に生成→ eval
- 構文木
  - Lisp/Scheme のマクロは実行前に構文木を操作する
  - LINQ は実行時に構文木を見る
- 実行時環境は実行時にしか操れない

実行前メタプログラミングは強力であるが，実行前の計算という実行時計算とは異なる計算モデルを持ち込むため，複雑になるという欠点がある．

# Rubyのメタプログラミング

- 基本的に実行時の操作
  - 実行前メタプログラミングのための特別の機能はない
  - 普通の Ruby プログラムの一部として実現される
  - operational に理解すれば良いので比較的わかりやすい
- 構文木の取得，操作は基本的にできない
- バイトコードも改変不可

# いつメタプログラミングするべきか

- Ruby は十分高度で柔軟な抽象化機構をそなえている
- メタプログラミングはプログラムの理解を難しくする
- そのため，通常のスクリプティングでは必要ない
- 高度な/特別な抽象化を必要とするのは，プログラムが大規模であったり，複数の人間がかかわるとき
- ライブラリとして高度な抽象化が必要な時などに有用

# Rubyのメタプログラミングを学ぶために

- Rubyの基本的なクラス (BasicObject, Object, Module, Class) が重要
- Rubyの内部構造について意識する

# クラス

クラスは以下の機能を持つオブジェクトである

- メソッド，定数などを保持する
- メソッドを検索する
- そのクラスに属するオブジェクトを生成する

ここからオブジェクトを生成する機能を除いた機能を持つのがモジュールである．

# メソッド呼びだし

Rubyではメソッド呼びだしは以下のように行なわれる

- オブジェクトの所属するクラスにメソッド探索を依頼する
  - クラスは自分自身が保持しているメソッド群からその名前のものを探す
  - もし見付からなかった場合は，スーパークラスに探索を依頼
- もしメソッドが見付かったら，それを呼びだす
- 見付からなかったら，method_missingを呼びだす.
  - 通常は，これが NoMethodError を生成する

# 特異メソッドと特異クラス

- ある一つのオブジェクトにしか定義されていないメソッドを特異メソッドと呼ぶ
  - クラスメソッドはクラスオブジェクトの特異メソッドである

# 特異メソッドと特異クラス

- ある一つのオブジェクトにしか定義されていないメソッドを特異メソッドと呼ぶ
  - クラスメソッドはクラスオブジェクトの特異メソッドである
- メソッドはクラスが保持する

# 特異メソッドと特異クラス

- ある一つのオブジェクトにしか定義されていないメソッドを特異メソッドと呼ぶ
  - クラスメソッドはクラスオブジェクトの特異メソッドである
- メソッドはクラスが保持する
- よって他のクラスと共有されないクラスが必要である

# 特異メソッドと特異クラス

- ある一つのオブジェクトにしか定義されていないメソッドを特異メソッドと呼ぶ
  - クラスメソッドはクラスオブジェクトの特異メソッドである
- メソッドはクラスが保持する
- よって他のクラスと共有されないクラスが必要である
- これを特異クラスと呼ぶ
- 実際にはリソースを節約するため，必要になるまでオブジェクトの特異クラスは生成されない

# 特異クラス

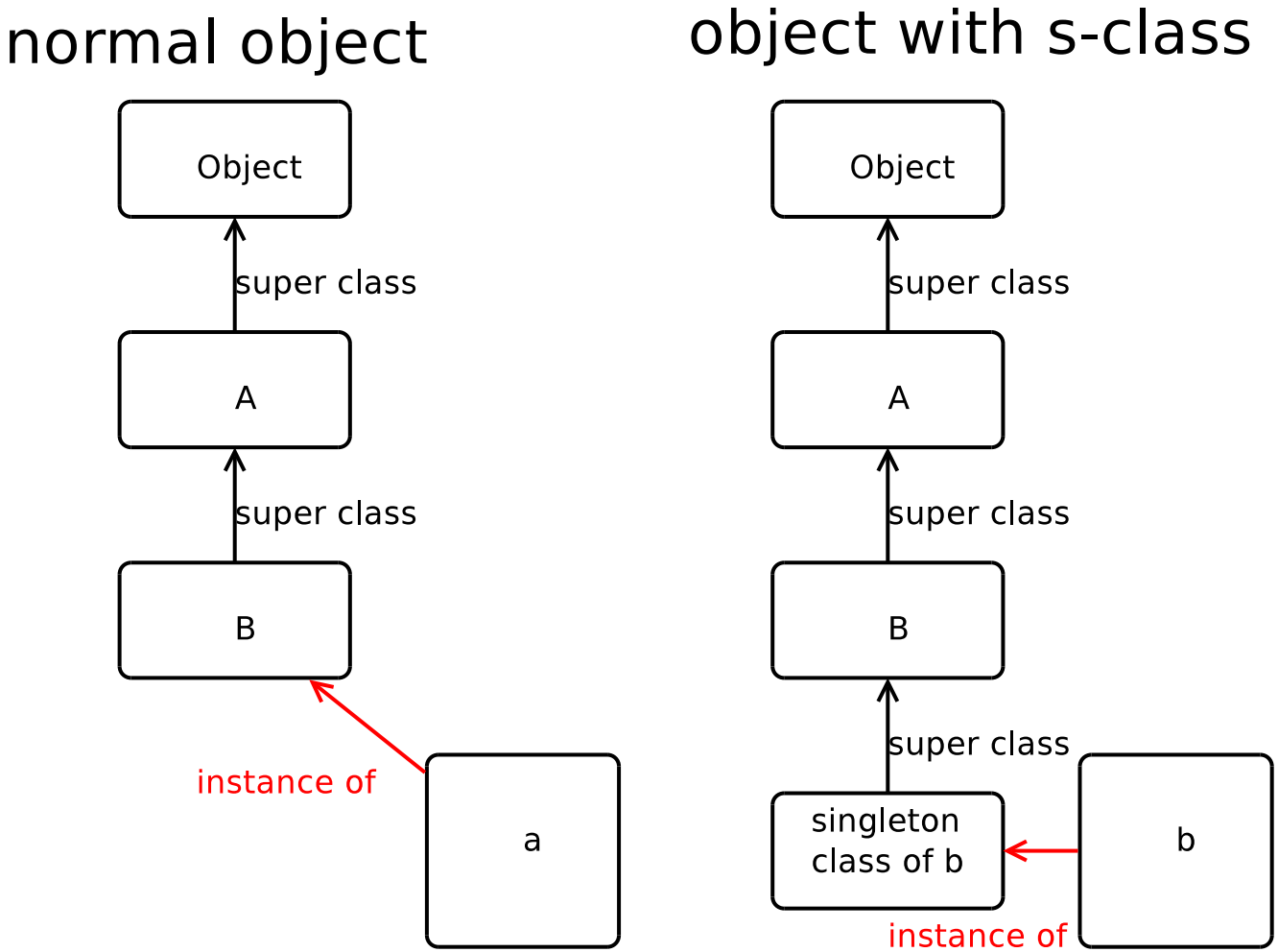
特異クラスを使うためには以下の文法を使う

```
class << obj
   :
end
```

Object#singleton_class というメソッドでオブジェクトの特異クラスを取り出すことができる.

# 特異クラス

特異クラスは，以下のようにオブジェクトとそれが属するクラスの間に挟み込まれる．



特異クラスは，一般的には「クラスメソッド，モジュール関数，などの定義先」という使い方がなされる．
また，様々なメタプログラミングのトリックのためにも用いられる．

# 演習問題

- irb で Object#singleton_class を呼びだしてみる.
- 取り出した特異クラスを操作してみる
  - instance_methods を呼びだしてみる
  - メソッドを定義する

# メソッド定義

メソッド定義は，通常，

```
class A
  def f
    :
  end
  :
end
```

というように，class ... end の間に，def ... end と書くことで行う．これによって，クラス A のメソッドテーブルに f を登録する．

# Module#define_method

Module#define_method というメソッドでもメソッドを定義できる．使いかたは以下の通り

```
class A
  define_method(:plus_1){|x| x+1}
end
```

これで 1 を足した値を返すメソッドが定義される．

これにより，メソッドを定義するメソッドを定義できる

```
module A
  def define_hello
    define_method(:hello){ puts "Hello, world!"
  end
end

class B
  extend A
  define_hello
end

B.new.hello
```

Module#define_method のおもしろい所は，def ... end とは異なり，ローカル変数スコープを区切らないことである．

```
class A
  y = 3
  define_method(:plus_3){|x| x+y}
end
```

メソッドの実体を外から Proc で持ってくることもできる所も興味深い．

# 演習問題

- あるクラスに 0 を返す zero というメソッドを define_methods で定義する
- define_zero というメソッドを定義し，これを呼びだすと「そのクラスに 0 を返す zero というメソッドが定義される」ようにする
- define_n というメソッドを定義し，これを呼びだすと「そのクラスに引数で渡したメソッドを返す n というメソッドが定義される」ようにする

# Module#define_method応用編

ここまで話してきたことの応用として,

- クラス変数のようなものをクラス変数を使わずに定義する
- インスタンス変数のようなものをインスタンス変数を使わずに定義する

をやってみましょう.

define_method を使うと，クラス変数もどきを定義できるというのは，define_method を嗜む Rubyist にはわりと良く知られた話です.

```
module A
  def define_class_attr(name)
    v = nil
    define_method(name){ v }
    define_method("#{name}="){|x| v = x}
  end
end
```

```
class X
  extend A
  define_class_attr :y
end

a = X.new
b = X.new
a.y = 2
a.y # => 2
b.y # => 2
b.y = 3
a.y # => 3
```

# では，インスタンス変数もどきは作れるか？

```
class Y
  extend B
  define_attr :z
end

a = Y.new
b = Y.new
a.z = 2
p a.z # => 2
p b.z # => nil
b.z = 3
p a.z # => 2
p b.z # => 3
```

- 当然インスタンス変数は使わないとする
  - 値は適当な環境に保持させる
- define_attr の実行は一度しか行なわれない
  - すなわち各オブジェクトごとに環境を作ることはできない

- つまりオブジェクト毎に確実に呼ばれるメソッドを考える

- つまりオブジェクト毎に確実に呼ばれるメソッドを考える
- new と initialize
  - initialize は何かと面倒
  - new を hook する

- つまりオブジェクト毎に確実に呼ばれるメソッドを考える
- new と initialize
  - initialize は何かと面倒
  - new を hook する
- もう一つ存在する

- つまりオブジェクト毎に確実に呼ばれるメソッドを考える
- new と initialize
  - initialize は何かと面倒
  - new を hook する
- もう一つ存在する
  - 定義しようとしているメソッドそのもの

# というわけでこうなる

```
module B
  def define_attr(name)
    define_method(name) {
      nil
    }
    define_method("#{name}="){|v|
      singleton_class.module_eval {
        define_method(name){ v }
        define_method("#{name}="){|w| v = w}
      }
      v
    }
  end
end
```

- 真面目にするんだったらスレッドについて考えましょう
  - 要排他

- 真面目にするんだったらスレッドについて考えましょう
  - 要排他
- そこまで真面目にするようなことか，という話も
  - marshaling で保存されない

- 真面目にするんだったらスレッドについて考えましょう
  - 要排他
- そこまで真面目にするようなことか，という話も
  - marshaling で保存されない
- define_method で define_method
  - 共有される環境を好きに作れる
  - 他の使い道は何だろう？

# 質問

ご静聴ありがとうございました

質問をどうぞ

# 代入するたびに再定義.

```
module D
  def define_attr(name)
    define_method(name) {
      nil
    }
    define_method("#{name}="){|v|
      singleton_class.module_eval {
        define_method(name){ v }
      }
      v
    }
  end
end
```