Ruby 初級者向けレッスン第 4 回

かずひこ@株式会社 ネットワーク応用通信研究所

2005 年 11 月 26 日

Ruby で学ぼう例外処理

- 例外処理とは？
- いろいろな例外
- 例外処理の実装
- 例外を発生させる
- 独自の例外クラス

例外処理とは？

- 実行時のエラーを処理するための方法
- エラーが起きた時にどうするかを記述できる
- エラーの処理を、検出したい処理ごとにその近くに記述する必要がなく、正常時の処理と例外の処理を分離できる
- だからプログラムが書きやすくなる
- だからプログラムが読みやすくなる

いろいろな例外（1）

- 整数を 0 で割算

```
# sample1.rb
p 1/0

$ ruby sample1.rb
sample1.rb:1:in `/': divided by 0 (ZeroDivisionError)
        from sample1.rb:1
```

- sample1.rb の 1 行めで例外オブジェクトが発生

- 例外のクラスは ZeroDivisionError
- 例外のメッセージは「divided by 0」
- 例外までの足跡（バックトレース）は sample1.rb の 1 行め

いろいろな例外（2）

- 存在しないファイルを読む

```
# sample2.rb
p File.read('/not/found/file')

$ ruby sample2.rb
sample2.rb:1:in `read': No such file or directory - /not/found/file (Errno::ENOENT)
        from sample2.rb:1
```

いろいろな例外（3）

- メソッド名の typo

```
# sample3.rb
p Array.now

$ ruby sample3.rb
sample3.rb:1: undefined method `now' for Array:Class (NoMethodError)
```

いろいろな例外（4）

- 文法エラー

```
# sample4.rb
p 1 /

$ ruby sample4.rb
sample4.rb:1: syntax error
```

いろいろな例外（5）

- メソッド呼び出しの無限ループ

```
# sample5.rb
def foo
  foo
end

foo

$ ruby sample5.rb
sample5.rb:2:in ‘foo’: stack level too deep (SystemStackError)
    from sample5.rb:2:in ‘foo’
    from sample5.rb:2:in ‘foo’
    ...
```

定義済み例外クラス

- **あらかじめ以下の例外クラスが定義されている**

```
Exception
  ├ NoMemoryError
  ├ ScriptError
  │   ├ LoadError
  │   ├ NotImplementedError
  │   └ SyntaxError
  ├ SignalException
  │   └ Interrupt
  ├ StandardError
  │   ├ ArgumentError
  │   ├ IndexError
  │   ├ IOError
  │   │   └ EOFError
  │   ├ LocalJumpError
  │   ├ NameError
  │   │   └ NoMethodError
  │   ├ RangeError
  │   │   └ FloatDomainError
  │   ├ RegexpError
  │   ├ RuntimeError
  │   ├ SecurityError
  │   ├ SystemCallError
  │   ├ SystemStackError
  │   ├ ThreadError
  │   ├ TypeError
  │   └ ZeroDivisionError
```



```
  └ SystemExit
```

例外処理の実装

- **文法**

```
begin
  式..
[rescue [例外クラス,..] [=> 例外変数名]
  式..]..
[else
  式..]
[ensure
  式..]
end
```

rescue 節

- **単純な** `rescue` **の例**

```
# sample1a.rb（整数を 0 で割算）
begin
  p 1/0
rescue
  puts "Class : #{$!.class}"
  puts "Message : #{$!.message}"
  puts "Backtrace :"
  puts $!.backtrace.join("\n")
end

$ ruby sample1a.rb
Class : ZeroDivisionError
Message : divided by 0
Backtrace :
sample1a.rb:3:in ‘/’
sample1a.rb:3
```

rescue する例外の指定

- `rescue` **節に例外クラスを指定すると（複数指定可）、指定した例外クラスとそのサブクラスの例外だけを補足する**
- **無指定時は** `StandardError` **を指定したのと同じ**

- ZeroDivisionError の場合

```
rescue #=> 補足する
rescue StandardError #=> 補足する
rescue ZeroDivisionError #=> 補足する
rescue TypeError #=> 補足しない
```

- LoadError や SyntaxError は、StandardError のサブクラスではないので、ただの rescue では補足できない
- 何でも補足してしまうと、かえって問題に気づきにくい

例外オブジェクトを変数に代入

- 'rescue [例外クラス,..] => 変数名' とすると、指定された変数に$!と同様に発生した例外が代入される

```
# sample1b.rb（整数を 0 で割算）
begin
  p 1/0
rescue => err
  puts "Class : #{err.class}"
  puts "Message : #{err.message}"
  puts "Backtrace :"
  puts err.backtrace.join("\n")
end
```

例外は呼び出し元にさかのぼる

- メソッドの呼び出し元にさかのぼって例外が返る

```
# sample1c.rb（整数を 0 で割算）
def method1
  return 1/0
end

def method2
  method1
end

begin
  method2
rescue
```



```
  puts "Class : #{$!.class}"
  puts "Message : #{$!.message}"
  puts "Backtrace :"
  puts $!.backtrace.join("\n")
end

$ ruby sample1c.rb
Class : ZeroDivisionError
Message : divided by 0
Backtrace :
sample1c.rb:3:in `/'
sample1c.rb:3:in `method1'
sample1c.rb:7:in `method2'
sample1c.rb:11
```

else 節

- else 節は、例外が発生しなかった時に実行される

```
# sample1d.rb（整数を 0 以外で割算）
begin
  p 1 / 1
rescue ZeroDivisionError
  puts 'ZeroDivisionError raised'
else
  puts 'nothing raised'
end
```

ensure 節

- ensure 節は、例外が発生してもしなくても実行される

```
# sample1e.rb（整数を 0 で割算するかも）
begin
  p 1 / rand(2)
rescue ZeroDivisionError
  puts 'ZeroDivisionError raised'
else
  puts 'nothing raised'
ensure
  puts 'always run ensure section'
end
```

rescue の中の retry

- rescue 節の中で retry を使うと、begin の最初からもう一度実行する

```
# sample1f.rb（整数を 0 で割算）
begin
  p 1 / rand(2)
rescue ZeroDivisionError
  puts "try again"
  retry
end
```

- 何度やってもダメな時に使うと無限ループに

rescue 修飾子

- ’式 1 rescue 式 2’ のようにも書ける（rescue 修飾子）
- 捕捉する例外クラスは指定できない（つまり StandardError サブクラスだけ）

```
a = (1 / 0 rescue 2)
p a #=> 2
```

例外を発生させる

- 問題が起きた時に自動的に例外は発生するが、組み込み関数の raise を用いて明示的に例外を発生させることもできる
- 文法は以下のとおり
  - raise
  - raise(例外クラス)
  - raise(メッセージ)
  - raise(例外クラス, メッセージ [, バックトレース])
  - (第一引数の「例外クラス」は「例外オブジェクト」も可)
- 以下の raise の行を ’raise(TypeError)’ や ’raise("message")’ や ’raise(TypeError, "message")’ のように変えて実行してみよう

```
# sample6.rb（例外を発生させる）
begin
  raise
rescue
  puts "Class : #{$!.class}"
  puts "Message : #{$!.message}"
  puts "Backtrace :"
  puts $!.backtrace.join("\n")
end
```


- 引数がない raise は、同じスレッドの同じブロック内で最後に rescue された例外オブジェクト（$!）を再発生させる
- そのような例外が存在しなければ RuntimeError 例外（メッセージは””）を発生させる

```
# sample6a.rb（整数を 0 で割算）
begin
  p 1/0
rescue
  puts "Class : #{$!.class}"
  puts "Message : #{$!.message}"
  puts "Backtrace :"
  puts $!.backtrace.join("\n")
  raise
end
```

余談：組み込み関数

- Kernel モジュールで定義されているメソッドは、どこからでもレシーバなしに呼び出せるので「組み込み関数」とか「関数風メソッド」とか呼ばれる
- raise の他に p, print, puts, require, system など

独自の例外クラス

- 定義済みの例外クラスの他に、独自の例外クラスを作ることができる
- Exception クラスや StandardError クラスのサブクラスとして定義する

```
# sample7.rb（独自の例外クラス）
class MyException < StandardError
end

begin
  raise(MyException, ’a message of my exception’)
rescue
  puts "Class : #{$!.class}"
  puts "Message : #{$!.message}"
  puts "Backtrace :"
  puts $!.backtrace.join("\n")
end
```

演習問題

- 以下のメソッド num_check を定義しましょう
  - 引数を一つだけとる
  - 引数が整数でなければ TypeError（メッセージは 'not integer'）を発生
  - 引数が負なら NegativeError（メッセージは 'negative value'）を発生
  - そうでなければ引数の値を返す
- num が整数かは num.kind_of?(Integer) で判定できる
- 書ける人はテスト・ファーストで書こう

まとめ

- プログラムにエラー処理はつきもの
- 例外を使うと上品なプログラムになる
- rescue の対象は適宜限定しよう

参考文献

『たのしい Ruby』
ISBN:4797314087

『プログラミング Ruby』
ISBN:4894714531

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/