30分でわかるcallccの使い方

yhara（原　悠）
京大マイコンクラブ

今日のまとめ

- （1） callccって何？
- （2） callccは凄い！
- （3） callccは危ない！

callccって何？

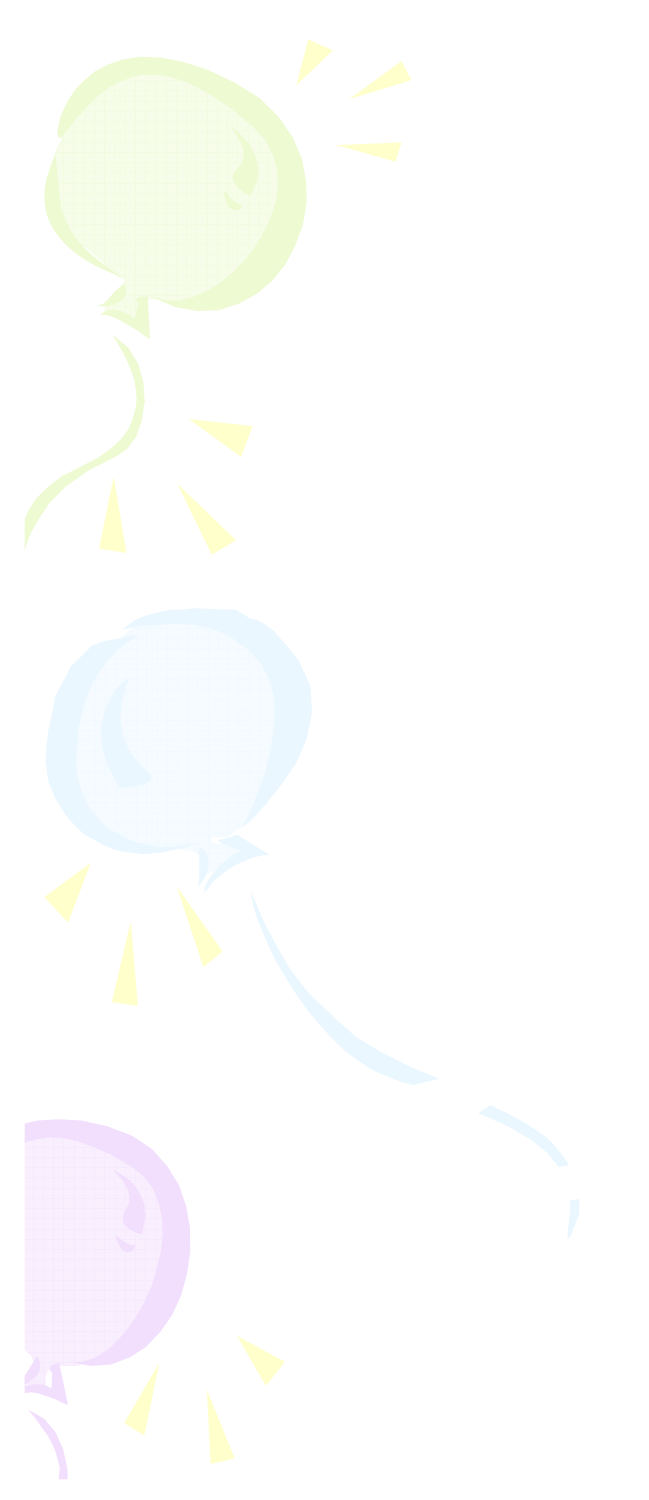
キーワード：継続、Continuation、callcc

プログラムの基本要素

- （1） 順次実行
- （2） 分岐
- （3） 繰り返し

プログラムの基本要素

- （1） 順次実行
- （2） 分岐
- （3） 繰り返し
- （4） ワープ



ワープ？

```ruby
class Foo
  def f
    p 1
    callcc{ |cc| return cc}
    p 2
  end
end

class Bar
  def initialize
    @cc = Foo.new.f
  end

  def g
    p 3
    @cc.call
    p 4
  end
end

Bar.new.g
```

SMILEVIDEO

2007年04月08日 17:26:40 投稿

ドラクエ3 勇者一人旅(チート等なし)part79

勇者「ドラゴン・・・」一部カットあり

登録タグ まつもと 勇者一人旅 DQ3 ドラクエ3 ドラクエ ドラゴンクエスト 日記 ゲーム 勇者一人旅71-80 こんがり直火焼き 【タグ編集】

ドラクエ3　勇者一人旅（チート等なし）part79

勇者「ドラゴン・・・」一部カットあり

登録タグ　まつもと　勇者一人旅　DQ3　ドラクエ3　ドラクエ　ドラゴンク
り回天焼き　【タグ編集】

ドラクエ3 勇者一人旅(チート等なし)part79

勇者「ドラゴン・・・」一部カットあり

登録タグ まつもと 勇者一人旅 DQ3 ドラクエ3 ドラクエ ドラゴンク
り画大焼き 【タグ編集】

Matz...?

class Matumoto

- 王様と話すとセーブできる
- ドラゴンのこうげき！
- 89のダメージ
- まつもとはしんでしまった
- 王様「まつもとよ、しんでしまうとは情けない」
- →セーブしたところからやりなおし

callccはセーブポイントに似ている

- callccでセーブ、cc.callでロード

```
$cc = nil

def hoge
  print 1
  callcc{|cc| $cc = cc}
  print 3
end

hoge
$cc.call
```

callcc＝セーブ
(ccがセーブポイント)

ccはContinuationクラスのインスタンス

cc.call＝ロード

callccはセーブポイントに似ている

- callccでセーブ、cc.callでロード

```
$cc = nil

def hoge
  print 1
  callcc{ |cc| $cc = cc}
  print 2
end

hoge
$cc.call
```

callcc＝セーブ
(ccがセーブポイント)

ccはContinuationクラスのインスタンス

cc.call＝ロード

callccはセーブポイントに似ている

- callcc{ }の返り値:
  - cc.call(arg)で飛んできたときはarg
  - そうでない時(最初の一回)はブロックの返り値

```
$cc = nil

def hoge
  print 1
  print callcc{|cc| $cc = cc; 2}
  print 3
end

hoge
$cc.call(4)
# 1 2 3 4 3 4 3 4 ... と出力される
```

まとめ

- セーブ
  - callcc{ |cc| … }　# ccがセーブポイント
- ロード
  - cc.call
  - callccの「次の処理」から再開される

callccは凄い！

（1）3重ループを一発で脱出

```
for i in (0..10)
  for j in (0..10)
    for k in (0..10)
      if i==j && j==k
        #ループを抜け出したい！
      end
    end
  end
end
```

（1）3重ループを一発で脱出

```
callcc{ |cc|
  for i in (0..10)
    for j in (0..10)
      for k in (0..10)
        if i==j && j==k
          cc.call
        end
      end
    end
  end
}
```

ここ(callccの次の処理)
に飛ぶ

それcatchでできるよ

- catch/throw

```
catch(:escape){
  for i in (0..10)
    for j in (0..10)
      for k in (0..10)
        if i==j && j==k
          throw :escape
        end
      end
    end
  end
}
```

（2）メソッドを「少しずつ」実行する

- ゲームのイベント処理

```
def event
  king.say("おお#{@name}よ、しんでしまうとはふがいない")
  wait_ok
  king.say("そなたにもういちどきかいをあたえよう")
  wait_ok
  king.say("では ゆけ #{@name}よ！")
end
```

  – wait_okをどのように実装すればいい？

```
until input["ok"]   #これは他の処理が
  sleep 1           #止まってしまうのでダメ
end
```

(´･ω･｀)

```
def event(input)
  case @step
  when 0
    king.say “おお#{@name}！ しんでしまうとはふがいない”
    @step += 1
  when 1
    @step += 1 if input[“ok”]
  when 2
    king.say “そなたに もういちど きかいを あたえよう”
    @step += 1
  when 3
    @step += 1 if input[“ok”]
  when 4
    king.say “では ゆけ #{@name}よ！”
  end
end
```

callccを使うと…

- メソッドを「少しずつ」実行できる
  - wait_okが呼ばれたたら、セーブしてreturn
  - 次回の呼び出しはセーブしたところから再開

```
def event
  king.say("おお#{@name}よ、しんでしまうとはふがいない")
  wait_ok
  king.say("そなたにもういちどきかいをあたえよう")
  wait_ok
  king.say("では ゆけ #{@name}よ!")
end
```

（3）全探索を簡単に

- あるマンションに5人の男が住んでいる
  - bakerは5Fではない
  - cooperは1Fではない
  - millerはcooperより上の階にいる
  - smithとfletcherは1つ隣の階にいる…

ありがちな方法

```
for baker in 1, 2, 3, 4, 5
  for cooper in 1, 2, 3, 4, 5
    for fletcher in 1, 2, 3, 4, 5
      for miller in 1, 2, 3, 4, 5
        for smith in 1, 2, 3, 4, 5
          if baker != 5 && cooper != 1 && fletcher != 1 &&
                                     fletcher != 5 && miller > cooper
             return [baker,cooper,fletcher,miller,smith]
        end
      end
    end
  end
end
```

callccを使うと…

```
require "amb“
A = Amb.new
baker   = A.choose(1, 2, 3, 4, 5)    #⇒ 1～5の数字を順に選んでくれる
cooper  = A.choose(1, 2, 3, 4, 5)
fletcher= A.choose(1, 2, 3, 4, 5)
miller  = A.choose(1, 2, 3, 4, 5)
smith   = A.choose(1, 2, 3, 4, 5)

A.assert([baker, cooper, fletcher, miller, smith].uniq.length == 5)
A.assert(baker != 5)
A.assert(cooper != 1)
A.assert(fletcher != 1 && fletcher != 5)
A.assert(miller > cooper)
A.assert((smith - fletcher).abs != 1)
A.assert((fletcher - cooper).abs != 1)

p [baker, cooper, fletcher, miller, smith]
```

(4) eachを「少しずつ」回す

- C++風のイテレータ
  - requrie 'generator' (標準添付)

```
g = Generator.new([1,2,3])
while g.next?
  p g.next
end
```

(5) ppp

- pデバッグ
- 変数名を書くとわかりやすい
- でも面倒
- 自動でできないか？

```
irb> p x
3
```

```
irb> p "x=#{x}"
x=3
```

- 変数の値を調べるには

bindingというメソッドを使う

```
irb> ppp :x
x=3
```

呼び出し元のbindingが欲しい

```
def ppp(symbol)
  set_trace_func{|*args| .. cc.call(eval("binding")}
  b = callcc{|cc| ..}
  if state == :prepare
    return
  else
    puts "#{symbol} = #{eval(symbol, b)}"
  end
end

a = 1
ppp :a
```

Ruby界の3大黒魔術が夢の競演！
- eval系 (eval, *_eval)
- **フック系** (*_missing, set_trace_func)
- callcc

※このコードは抜粋です

呼び出し元のbindingが欲しい

```
def ppp(symbol)
  set_trace_func{|*args| .. cc.call(eval("binding")}
  b = callcc{|cc| ..}
  if state == :prepare
    return
  else
    puts "#{symbol} = #{eval(symbol, b)}"
  end
end

a = 1
ppp :a
```

こんなもん誰が考えたんだ

- Binding.of_caller として Rails (ActiveSupport)で導入されたのが初出？
- 1.8.5以降だと動かない…。

- pppの配布元はこちら
  - http://www.rubyist.net/~rubikitch/computer/ppp/

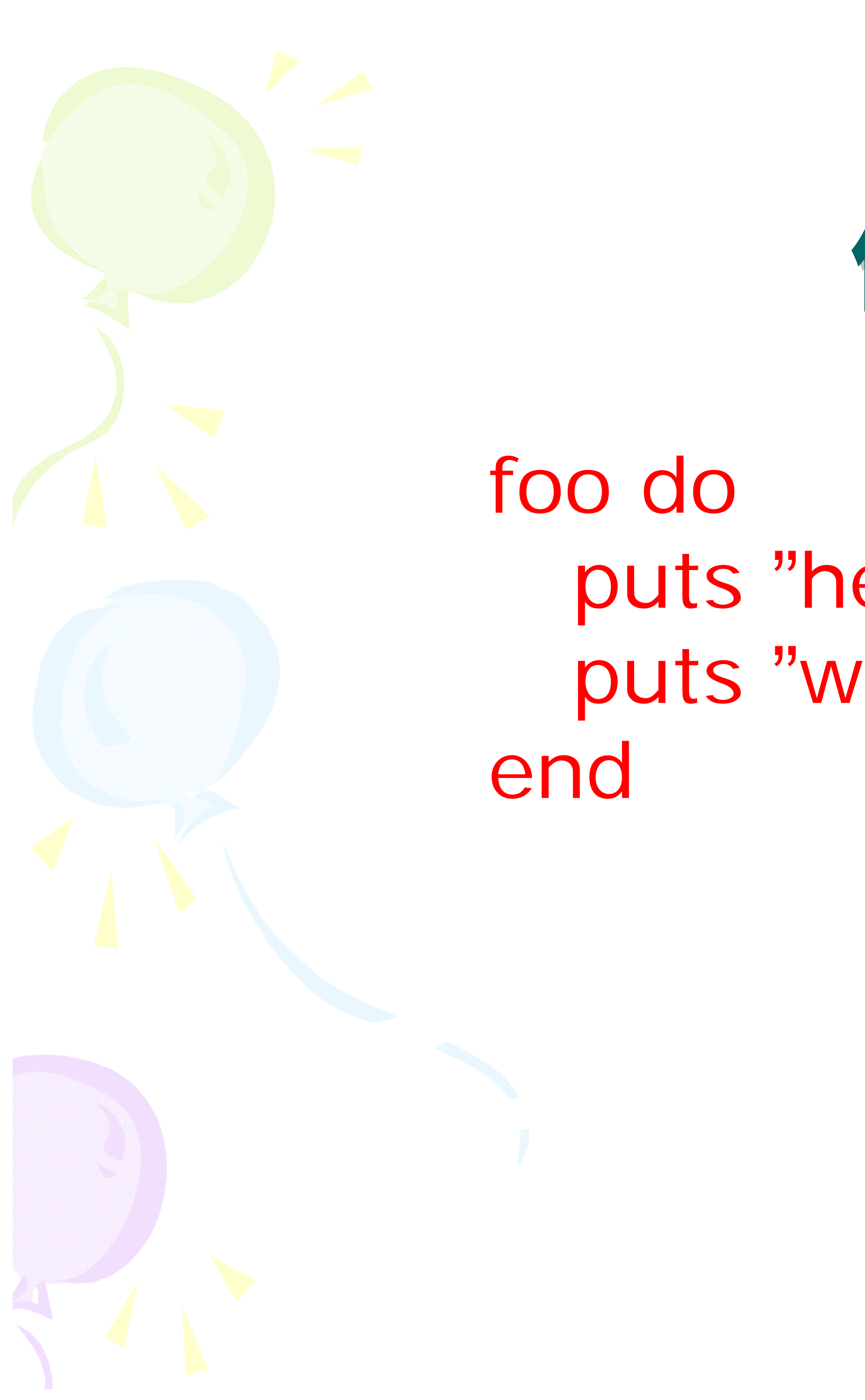
callccは危ない！

なぜ危ない？

- (Rubyの) バグの原因になりやすい
- 油断するとRubyが落ちる

```
[BUG] Segmentation fault:
```

- 「次のようにすると core を吐きます」祭り
  - Rubyist Magazine – Rubyの落とし方 (akr)
  - http://jp.rubyist.net/magazine/?0002-RubyCore



例

```
foo do
  puts ”hello”
  puts ”world”
end
```

```
static VALUE rb_foo(VALUE ary1)
{
    int *p = malloc(...);
    ...
    rb_yield();
    ...
    free(*p);
}
```

```
foo do
    puts ”hello”
    puts ”world”
end
```

```
static VALUE rb_foo(VALUE ary1)
{
    int *p = malloc(...);
    ...
    rb_yield();
    ...
    free(*p);
}
```

```
foo do
    callcc{|$cc| ...}
    puts ”hello”
    puts ”world”
end
$cc.call
```

例

```
static VALUE rb_foo(VALUE ary1)
{
    int *p = malloc(...);
    ...
    rb_yield();
    ...
    free(*p);
}
```

2回freeされてしまう

```ruby
foo do
  callcc{|$cc| ...}
  puts "hello"
  puts "world"
end
$cc.call
```

まとめ

callccを使うと

が簡単に書ける！

callccを使うと

スパゲッティコードが簡単に書ける！

まとめ

- callccはとても強力
  - 処理の流れを自在に操れる
- でも他人に読めないコードになりやすい
  - 見えないところに隠そう
- callccは楽しい
  - 思いもよらないことができる
- でもRubyインタプリタのバグ要因になり易い
  - 将来無くなるかも

おまけ

Fiberについて

callccとFiber

- callccの代表的な使い方：
  - (A) 処理の中断/再開 (generator, wait_ok)
  - (B) 処理のやり直し (amb, ppp)
- callccが危険なのは(B)が可能だから
- じゃあ(A)の機能だけなら残してもいいかも？
- というわけで、1.9にFiberが実装された

Fiber

- Fiber(繊維)⇔Thread(糸)
- Threadとの違い
  - newしただけでは実行されない
  - 勝手にスイッチされない
  - Fiber#yield で移動
- Fiberの特徴
  - 処理を途中で止めることができる
  - 直前のFiberを取れる(Fiber.prev)

```
foo = Fiber.new{
  p 2
  Fiber.prev.yield
}
bar = Fiber.new{
  p 1
  foo.yield
  p 3
}
bar.yield

#1 2 3の順に表示される
```

1.9はどうなる？

- Ruby1.9は継続と“Fiber”をサポート - @IT
  - …というのはガセ(サポートすると決まったわけではない)
- あり得る選択肢:
  - callccもFiberも残る
  - Fiberだけが残る
  - callccだけが残る
  - callccもFiberも残らない
- 今後に注目
  - 1.9.xは今年のクリスマスにリリース予定です

ご清聴ありがとうございました

ご質問は？