ThreadGroupクラスの強化とその利用法

Improvement of ThreadGroup class and its Usage

永井　秀利

Hidetoshi NAGAI

（九州工業大学 / Rubyist九州）

Kyutech
Kyushu Institute of Technology

RubyKaigi2011
July 16 -18, 2011 at Tokyo, Japan

自己紹介

- 名前：永井 秀利（Hidetoshi NAGAI）
- 属性：Ruby/Tkの人で，迷惑かけまくりのポンコツコミッター
- 所属：九州工業大学大学院情報工学研究院
  知能情報工学研究系知能情報メディア部門 助教
- 現在の主たる研究テーマ：
  「表面筋電による黙声認識に関する研究」
  （Inaudible Speech Recognition based on Surface-EMG）
  - いわゆる「くちパク」で喋った内容を認識する研究
  - カメラでの読唇に比べ，電極装着が必要なのが欠点だが，口が見えずとも強弱変化も含めて認識できるのが利点

この発表の主題

ThreadGroupクラス

RubyのThreadGroupクラス

- スレッドを束ねるもの
  - 生成したスレッドは親と同じThreadGroupに所属
  - 死んだスレッドはThreadGroupの管理下から外れる
- スレッドは必ずいずれかのThreadGroupに属する
- スレッドが所属するThreadGroupは動的に変更可能
- ThreadGroupオブジェクト間に関係や階層はない

JavaのThreadGroup

- スレッド生成時に引数で指定
  - 指定がなければ親スレッドと同じThreadGroupに所属
  - 所属するThreadGroupを後から変更することは不可能
- ThreadGroup間に階層関係あり
- RubyのThreadGroupよりも少しだけ多機能

PythonのThreadGroup

未実装

スレッド生成時の予約引数であるのを見ると，Javaを意識？

【 ここで質問 】

ThreadGroupクラスって

活用したことがありますか？

ほぼ全員がNO

…　と予想したけど，きっと外れてないよね

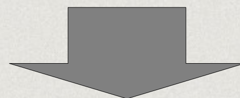
せっかく存在するのに使わない／使えないのは

もったいない

ならば，もっと使う気になるように強化できないか？

現在のThreadGroup / スレッド管理の課題

- ThreadGroup自体の問題
  - スレッド群を取り扱うための機能の不足
  - 操作権限に関する概念の欠如
- スレッド管理の問題
  - pollingしたり，個々に監視スレッドを用意するなどの余計なコストをかけない限り，スレッド群を終了順に処理できない
  - 上記と同様のコストをかけない限り，スレッド終了に対して即応できない



ThreadGroupの強化でこうした問題に対処したい

ThreadGroup強化のポイント

- スレッド（群）の操作性の向上
- ThreadGroup間の操作権限の設定
- 例外終了を含むスレッドの整列化（thread queue）
- 閉鎖された実行空間の生成（local space）

最初のものほど単純な強化で，後になるほど特殊な強化

なぜ拡張ライブラリではなく直接強化なのか？

Rubyのcoreに立ち入って変更を加えねば
実装不可能or無駄なコストがかかる機能が多いため

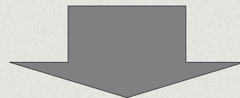
ThreadGroup強化のポイント

- スレッド（群）の操作性の向上
- ThreadGroup間の操作権限の設定
- 例外終了を含むスレッドの整列化（thread queue）
- 閉鎖された実行空間の生成（local space）

最初のものほど単純な強化で，後になるほど特殊な強化

操作性に関する課題

- 指定したThreadGroupに新しくThreadを生成する程度のことすら不便なのが現状
- スレッドを束ねるものでありながら，スレッド群として操作するメソッドがない
  - ThreadGroup#listで一旦配列化しての処理が必要
  - 対象はある瞬間のスレッド群で十分か？
処理中にスレッドが追加されうることも考慮すべきか？



こうした単純な機能不足を補うためにメソッドを追加

操作性に関する課題例：スレッドの生成と所属

指定したThreadGroupに新しくスレッドを生成する場合，

方法1：現スレッドを対象ThreadGroupに移し，スレッドを生成した後に復帰

```
err = nil
cur_thgrp = Thread.current.group
begin
  thgrp.add Thread.current
  Thread.new(args){ ... }
rescue Exception=>err
  # ここではThreadGroupが移されたまま
ensure
  cur_thgrp.add Thread.current
end
raise err if err
```

方法2：生成するスレッドを一旦sleepさせ，ThreadGroup移動後にrun

```
body = proc{ ... }
th = Thread.new(args,body){|a,blk|
  sleep
  blk.call(a)
}
thgrp.add th
Thread.pass while th.status!=“sleep”
th.run
```

ThreadGroup#new_thread

- 今回の強化案の中で最も単純な強化例のひとつ
- スレッド生成とThreadGroup設定とが単一処理にできないのが問題なので，C言語レベルで単一処理を実装
- 引数はThread.newと同じ

改善案：ThreadGroup#new_theadを導入

```
thgrp.new_thread(args){ ... }
```

Threadクラスのサブクラスへの対応はThreadGroup#new_thead_ofを導入

```
class MyThread < Thread; end
thgrp.new_thread_of(MyThread, args){ ... }
```

操作性向上のための他の強化例

- 単純な機能追加を個々に解説しても仕方がないので省略
- 操作性に直接関係はしないが，スレッドプログラムに影響するかもしれない例
  - ThreadGroup#[ ], ThreadGroup#[ ]=
    - Thread固有データと同様，ThreadGroup固有データを導入
    - スレッド群に共通の環境変数のようなもの
  - ThreadGroup#max_threads
    ThreadGroup#max_threads=
    - thread bomb対策として，所属できるスレッド数を制限
    - デフォルトでは制限なし

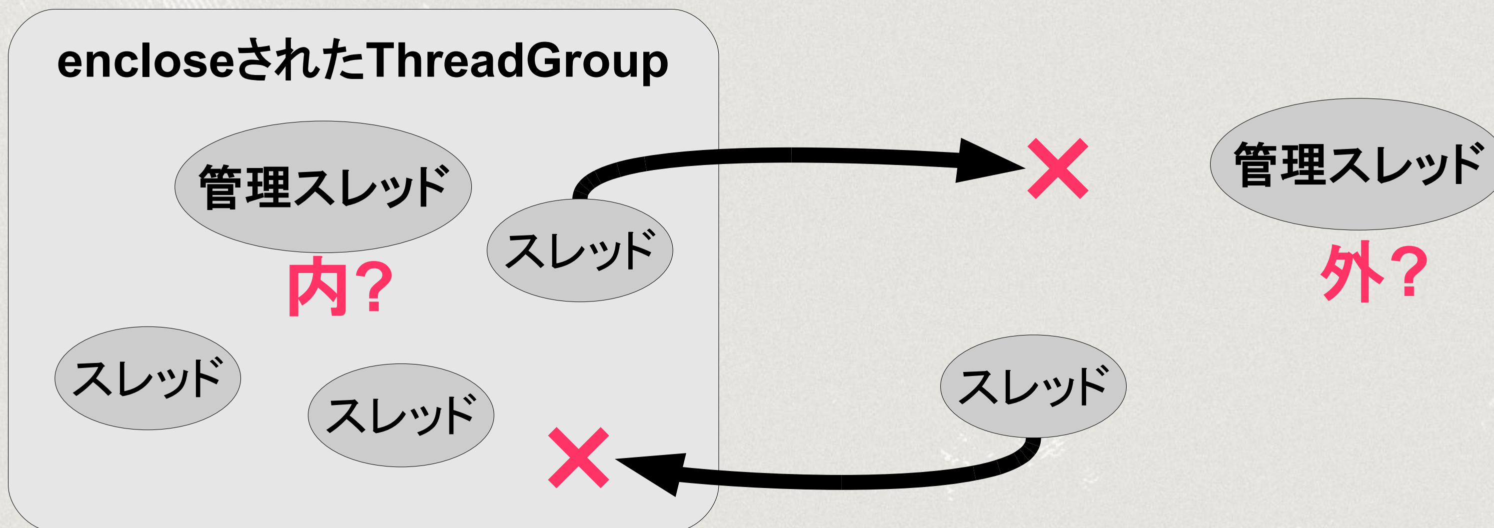
ThreadGroup強化のポイント

- スレッド（群）の操作性の向上
- ThreadGroup間の操作権限の設定
- 例外終了を含むスレッドの整列化（thread queue）
- 閉鎖された実行空間の生成（local space）

最初のものほど単純な強化で，後になるほど特殊な強化

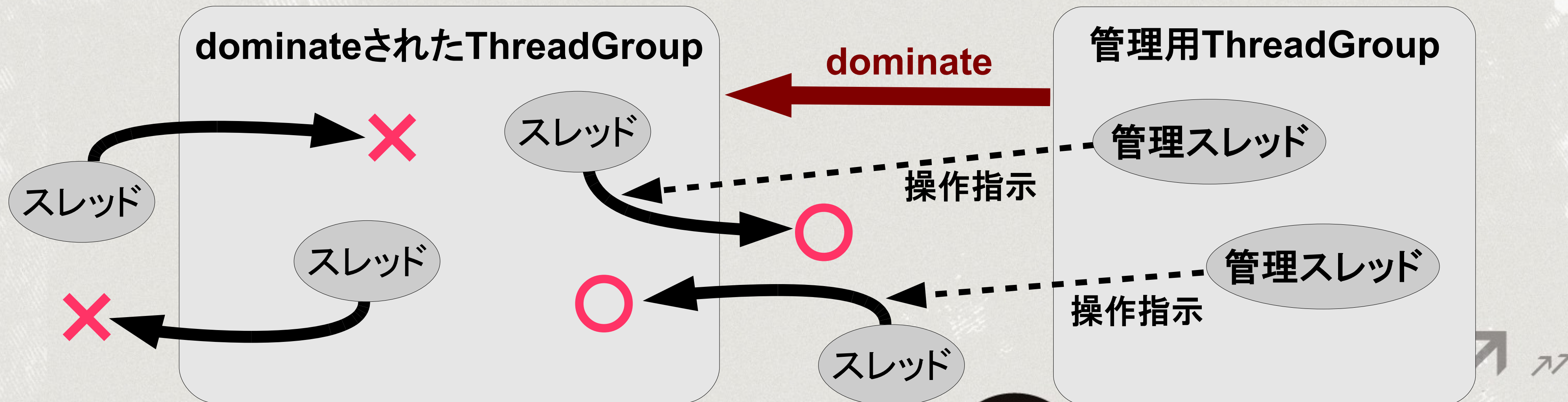
encloseされたThreadGroupとその管理

- 特定目的のスレッド群を一つのThreadGroupで管理
  - encloseにより，管理からの離脱を防止
    - 勝手に他所に逃げられないように
    - 勝手に他所から混ぜ込まれないように
  - 管理用のスレッドはどこに置くべきか？　内部？ 外部？



dominate

- ThreadGroupの操作権限を定めるもの
  - dominateされたThreadGroupの操作は，それをdominateしているThreadGroupに属するスレッドからのみに許可
  - ThreadGroup指定なので，複数スレッドの権限管理が容易
  - normal → dominated → enclosed → frozen は不可逆



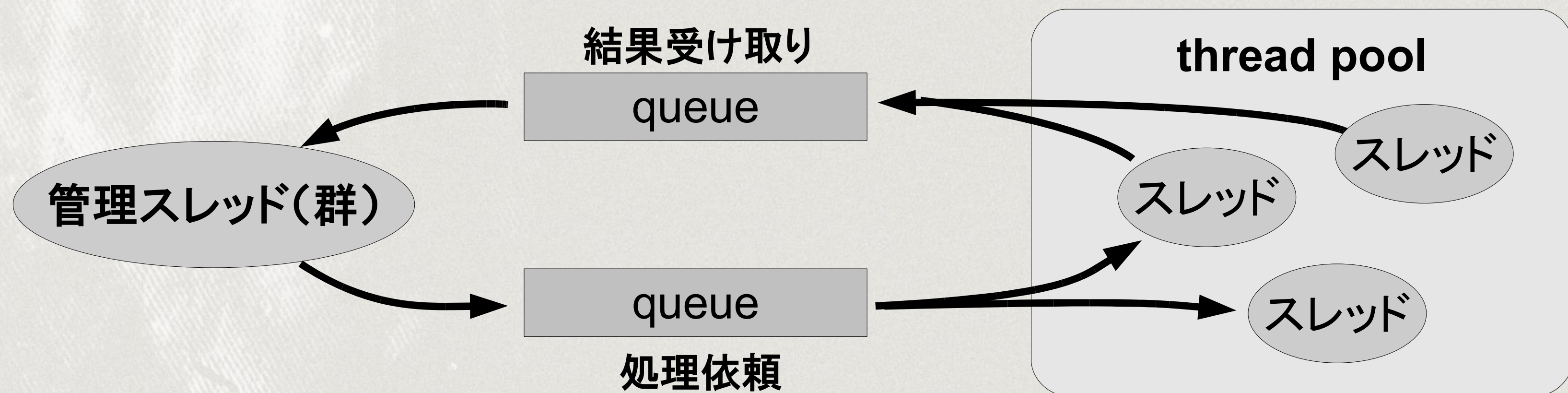
ThreadGroup強化のポイント

- スレッド（群）の操作性の向上
- ThreadGroup間の操作権限の設定
- 例外終了を含むスレッドの整列化（thread queue）
- 閉鎖された実行空間の生成（local space）

最初のものほど単純な強化で，後になるほど特殊な強化

複数スレッドへの処理分散

手隙のスレッドに処理を割り振る際にありがちなスタイル



- スケジューリングの主体はどちらにあるべきか？
- 処理スレッドが例外終了した場合は？ 即応性の確保は？
  - pollingで監視？ 個別に監視専用スレッドで包む？
  - 個々の処理スレッドに例外処理を書いた上で，監視専用の系統を用意する？ → 主導権が処理スレッド側に

thread queue

- 管理スレッド側に主導権をもたせやすくするための仕組み
- ThreadGruopごとに1個のthread queueを利用可能
  - 例外終了の場合を含め，終了したスレッドを終了順にqueueに送り込むように設定できる
    - スレッドの状態変化に対する即応性の確保
  - ThreadGroup.queueingによって，現在のスレッドがsleepすると同時にthread queueに入ることも可能
    - queue入りとsleepとが単一処理でないことのリスクの回避
    - 現在のスレッド側がqueueの場所を知る必要はない
    - dominateと組み合わせれば，どのthread queueに入るかの決定権は管理スレッド側が握る

thread queue利用の例（1）

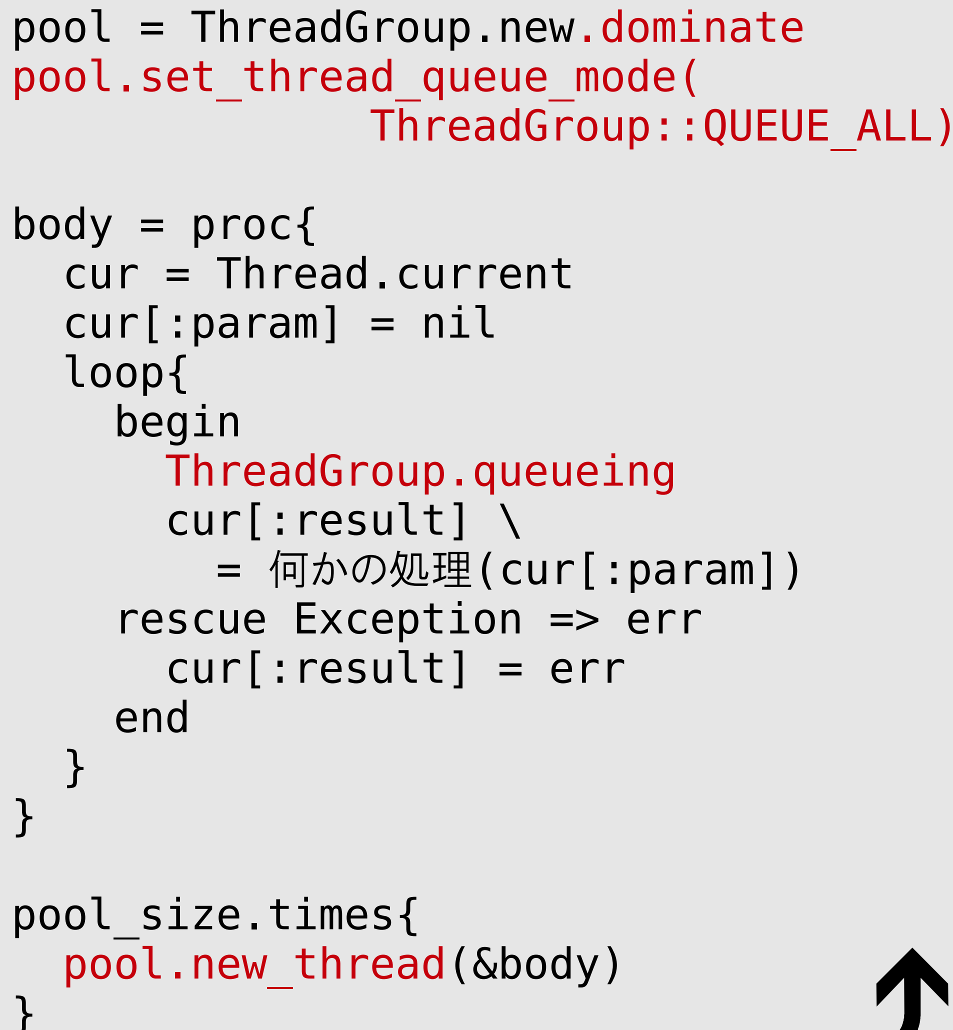
- 複数の問い合わせを並行して行い，最初に得られた結果を利用

```
thgrp = ThreadGroup.new
thgrp.set_thread_queue_mode(ThreadGroup::QUEUE_ALL)

para_cnt.times{
  thgrp.new_thread{  ...問い合わせ処理... }
}

para_cnt.times{
  if (th = thgrp.thread_queue_pop) == false
    thgrp.kill # 不要となったスレッドを強制終了
    return th.value
  end
}

raise RuntimeError, “ ... ”
```

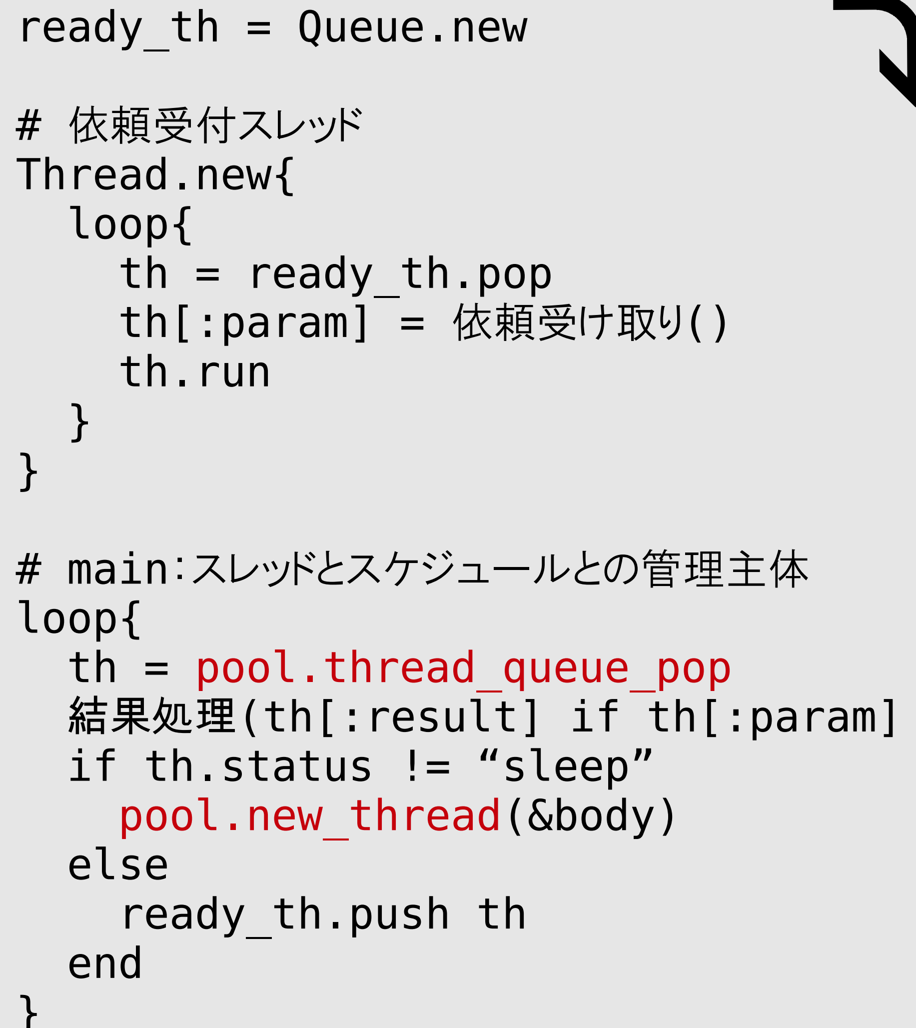
※ 赤字が強化案のメソッド

thread queue利用の例（2）

- thread poolの実装の一例



```
pool = ThreadGroup.new.dominate
pool.set_thread_queue_mode(
              ThreadGroup::QUEUE_ALL)

body = proc{
  cur = Thread.current
  cur[:param] = nil
  loop{
    begin
      ThreadGroup.queueing
      cur[:result] \
        = 何かの処理(cur[:param])
    rescue Exception => err
      cur[:result] = err
    end
  }
}

pool_size.times{
  pool.new_thread(&body)
}
```



```
ready_th = Queue.new

# 依頼受付スレッド
Thread.new{
  loop{
    th = ready_th.pop
    th[:param] = 依頼受け取り()
    th.run
  }
}

# main:スレッドとスケジュールとの管理主体
loop{
  th = pool.thread_queue_pop
  結果処理(th[:result] if th[:param]
  if th.status != “sleep”
    pool.new_thread(&body)
  else
    ready_th.push th
  end
}
```

ThreadGroup強化のポイント

- スレッド（群）の操作性の向上
- ThreadGroup間の操作権限の設定
- 例外終了を含むスレッドの整列化（thread queue）
- 閉鎖された実行空間の生成（local space）

最初のものほど単純な強化で，後になるほど特殊な強化

local space

- 今回の強化案の中で最も特殊で最も批判されている要素
- ThreadGroup内に閉鎖された実行空間（環境）を構築
  - 局所的に有効なメソッドや定数の実現
  - その空間だけで有効なクラス（名）/モジュール（名）
- 所属ThreadGroupの変更により，動的な環境変更が可能
  - 能動的にも受動的にも変更可能
  - ThreadGroupのencloseにより，変更を禁止することも可能

local spaceの用途

- スクリプトの実行テスト環境として
  - local spaceの中で組み込みクラスを汚染したとしても，その外には影響を及ぼさない
- sandbox作成の支援
  - safe levelの保護（実行防止）とは方向が違う話（汚染防止）
- 定義の衝突時に，個別の実行環境で共存させるために
  - Module#mixという話もあるが，混ぜて新しい環境を作る形では解決できない場合もあるのでは？
  - 例えばmathnライブラリが有効な環境と無効な環境とか

local spaceによる環境共存の例

- 今回の強化案でのmathnライブラリの有効/無効を模した動作例（スクリプトの実行テスト環境の動作例として見てもよい）

```
# 模擬mathn環境(local space)作成と同環境内での組み込みクラスへの変更
mathn_env = ThreadGroup.new.make_local_space
def mathn_env.eval(&b)
  new_thread(&b).value
end

mathn_env.eval{
  class Fixnum
    alias div quo # 試験実装では演算子メソッドに未対応なため，演算子”/”の代りにdivを変更
  end
}

# 通常環境での実行と模擬mathn環境での実行との違い
p 1.div(3)                         #=> 0
p mathn_env.eval{ 1.div(3) }  #=> (1/3)
```

～ なぜThreadGroupに実装するのか？～

Moduleの拡張ではだめなのか？

Kernel.loadでのように，Module内への定義封じ込めは？

- 問題となるのは「関数的メソッド」
  - Moduleをトップレベルとしたのではモジュールメソッドとして定義されてしまい，関数的メソッドのスコープが得られない
- 特別な設定をしたModuleをトップレベルとしたときは，関数的メソッド的な扱いをするというのはどうか？
  - module_evalや手続きオブジェクト実行等の際，どのlocal spaceでの実行となるのかで混乱する
  - 仮にトップレベルを固定するとしても，いつの時点での固定かが不明だし，「一時的に変更」が不可能なのも嬉しくない

～ なぜThreadGroupに実装するのか？～

Moduleの拡張ではだめなのか？

Kernel.loadでのように，Module内への定義封じ込めは？

- 問題となるのは「関数的メソッド」
  - Moduleをトップレベルとしたのではモジュールメソッドとして定義されてしまい，関数的メソッドのスコープが得られない
- 特別な設定をしたModuleをトップレベルとしたときは，関数的メソッド的な扱いをするというのはどうか？
  - module_evalや手続きオブジェクト実行等の際，どのlocal spaceでの実行となるのかで混乱する
  - 仮にトップレベルを固定するとしても，いつの時点での固定かが不明だし，「一時的に変更」が不可能なのも嬉しくない

～ なぜThreadGroupに実装するのか？～

特別なクラスを作るのではだめなのか？

Moduleを拡張しようとするからmodule_evalなどで問題になるので，local space管理専用のクラスを作れば？

- local space管理オブジェクトでのinstance_evalで封じ込めるという方法はあり得る
- 注意すべきはThread生成
  - 封じ込めは引き継がれるべきなので，local space管理オブジェクトの情報はThread単位で持つ必要がある
  - Threadクラスとlocal space管理クラスとは，切り離せない関係となるのでは？

～ なぜThreadGroupに実装するのか？～

特別なクラスを作るのではだめなのか？

Moduleを拡張しようとするからmodule_evalなどで問題になるので，local space管理専用のクラスを作れば？

- local space管理オブジェクトでのinstance_evalで封じ込めるという方法はあり得る
- 注意すべきはThread生成
  - 封じ込めは引き継がれるべきなので，local space管理オブジェクトの情報はThread単位で持つ必要がある
  - Threadクラスとlocal space管理クラスとは，切り離せない関係となるのでは？

なぜThreadGroupに実装するのか？

- スレッドと切り離して考えられないなら，Threadオブジェクトの情報の一部としてしまうべきでは？
  - 個々のスレッドごとにlocal space設定を管理するのは煩雑
  - 複数スレッドでの協調を考えると，どのスレッド群が共通環境なのかは重要
- スレッド群で考えた方が望ましいなら，ThreadGroup単位で情報を持たせれば十分では？
  - ThreadGroupに直接情報を持たせれば十分で，わざわざ特別なクラスを作る必要はなさそう
  - local space作成を明示的に宣言しない限り，ThreadGroup生成時のlocal spaceを共有することとすれば良い

なぜThreadGroupに実装するのか？

- スレッドと切り離して考えられないなら，Threadオブジェクトの情報の一部としてしまうべきでは？
  - 個々のスレッドごとにlocal space設定を管理するのは煩雑
  - 複数スレッドでの協調を考えると，どのスレッド群が共通環境なのかは重要
- スレッド群で考えた方が望ましいなら，ThreadGroup単位で情報を持たせれば十分では？
  - ThreadGroupに直接情報を持たせれば十分で，わざわざ特別なクラスを作る必要はなさそう
  - local space作成を明示的に宣言しない限り，ThreadGroup生成時のlocal spaceを共有することとすれば良い

local spaceの解放

- 不要になったlocal spaceはどうなる？
  - ThreadGroupがGCに回収されて消滅するときに，local spaceの登録内容（共有されていなければ）も消滅する
  - 消滅したlocal spaceの中だけで定義されていたクラス/モジュールは名前を失い，無名クラス/モジュールとなる
    - オブジェクトも参照も存在しなければGCで回収される
- local spaceで読み込むようにすれば，条件付きでライブラリのアンロードがGCのタイミングで可能になるかも？
  - きちんと検討はしてないが，pure Rubyならもしかしたら…
  - 特別なコーディングを条件とすれば，DLLもあるいは…

おわりに

- ごめんなさい．まだ強化案の詳細についての改訂を続けていることもあり，実装は完了していません
  - [ruby-dev:43901]に旧版の強化案と極一部の機能だけの試験実装のサンプルはあります（すぐSEGVするけど…）
- 批判意見多数なので，これらの案は消えていくだけかも…
  - coreに変更を加える必要があるので，十分な合意が形成されなければ導入は無理でしょう
  - 実装希望複数ならまだしも，現状はほぼ孤立無援なので…
- 一部なりとも賛同いただける部分があれば，声をあげるなどの支援をいただけますと嬉しいです

ご清聴ありがとうございました

Kyutech
Kyushu Institute of Technology

RubyKaigi2011
July 16 -18, 2011 at Tokyo, Japan