夏のプログラミング・シンポジウム2011 (2011/09/02)

RubyのThreadGroupクラスの機能拡張の試みについて

九州工業大学　永井 秀利

目的

- RubyのThreadGroup
  - 標準組み込みのクラス
  - 少なくとも1オブジェクトが必ず存在

. . . だが，あまり活用されていない！

  - 有効活用しやすくするための機能強化の試み
  - ThreadGroupというものの活用可能性の模索

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スレッドのThreadGroupを引き継ぐ
  - 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
  - mainスレッドは，デフォルトでThreadGroup::Defaultオブジェクトに所属
- スレッドが所属するThreadGroupは動的に変更可能
  - スレッド生成時にThreadGroupを指定することはできない
- ThreadGroupオブジェクト間に関係や階層はない

他言語のThreadGroup

- 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ThreadGrouop
  - 現状では未実装
  - スレッド生成時の予約引数としては存在
    - Javaと類似した実装にすることを想定か？

ThreadGroupに想定可能な役割

- スレッド群のコンテナ
  - ThreadGroupというものに対する極めて一般的な認識
- スレッド群の管理母体
  - 管理下のスレッドの状態変化の監視・捕捉
- スレッド群の共通実行環境
  - 複数のスレッドによって共有された存在とも見なせる
  - 新たなスレッドにも共有が引き継がれる

．．．RubyのThreadGroupをそれぞれの役割で見る

コンテナとして見た場合の現状

- 不十分な操作用インスタンスメソッド
  - あるのはThreadGroup#addとThreadGroup#list程度
  - スレッドを束ねるものでありながら，集合として操作するメソッドがない
- カレント以外の特定のThreadGroupにスレッドを生成する処理をプリミティブには実行できない

管理母体として見た場合の現状

- スレッドの生成や出入りを制限する機構あり
  - freeze：新たなスレッドの追加を禁止
  - enclose：ThreadGroup間のスレッドの移動は禁止だが，ThreadGropu内部での子スレッド生成は可能
- ThreadGroupの操作権限に関する概念の欠如
- スレッド終了の監視を支援する機構の欠落
  - 個々のスレッドの処理で対策を施しておかない限り，監視にはpollingしたり，個別に監視スレッドを用意したりが必要

共通環境として見た場合の現状

- encloseやfreezeは，そのThreadGroup環境への封じ込めとして機能するが，それ以外に標準的にサポートしている機能は特になし

ThreadGroup強化のポイント

- コンテナとして
  - スレッド（群）の操作性の向上
- 管理母体として
  - 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）
- 共通環境として
  - ThreadGroup固有データのサポート
  - 閉鎖された実行空間の生成（local space）

ThreadGroup強化のポイント

- コンテナとして
  - スレッド（群）の操作性の向上
- 管理母体として
  - 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）
- 共通環境として
  - ThreadGroup固有データのサポート
  - 閉鎖された実行空間の生成（local space）

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にスレッドを生成する場合の現状

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){ ... }
```

スレッド集合としての操作メソッドの追加

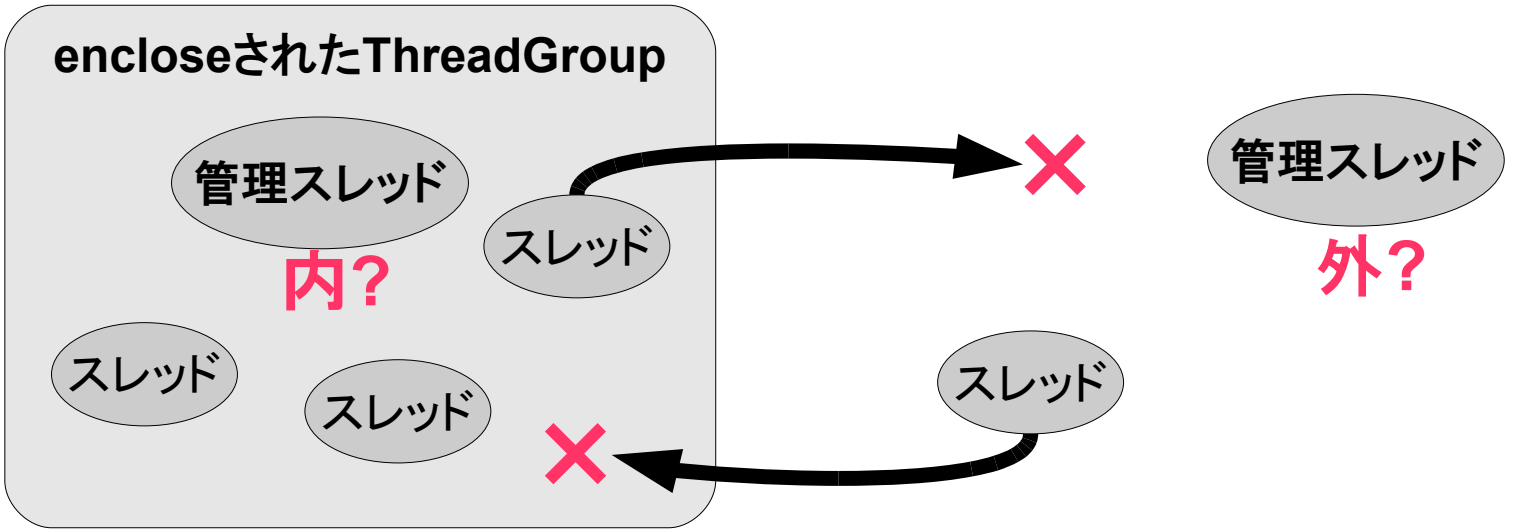
- メソッド呼出し時点のスレッド集合を対象とするもの
  - ThreadGroup#each{ ... }
  - ThreadGroup#raise　　…　など
- メソッドを呼出した時点から処理完了までの間に増加したスレッドも対象とするもの
  - ThreadGroup#join
  - ThreadGroup#kill　　…　など
- 所属可能なスレッド数を制限（thread bomb対策）
  - ThreadGroup#max_threads
  - ThreadGorup#max_threads=

ThreadGroup強化のポイント

- コンテナとして
  - スレッド（群）の操作性の向上
- 管理母体として
  - 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）
- 共通環境として
  - ThreadGroup固有データのサポート
  - 閉鎖された実行空間の生成（local space）

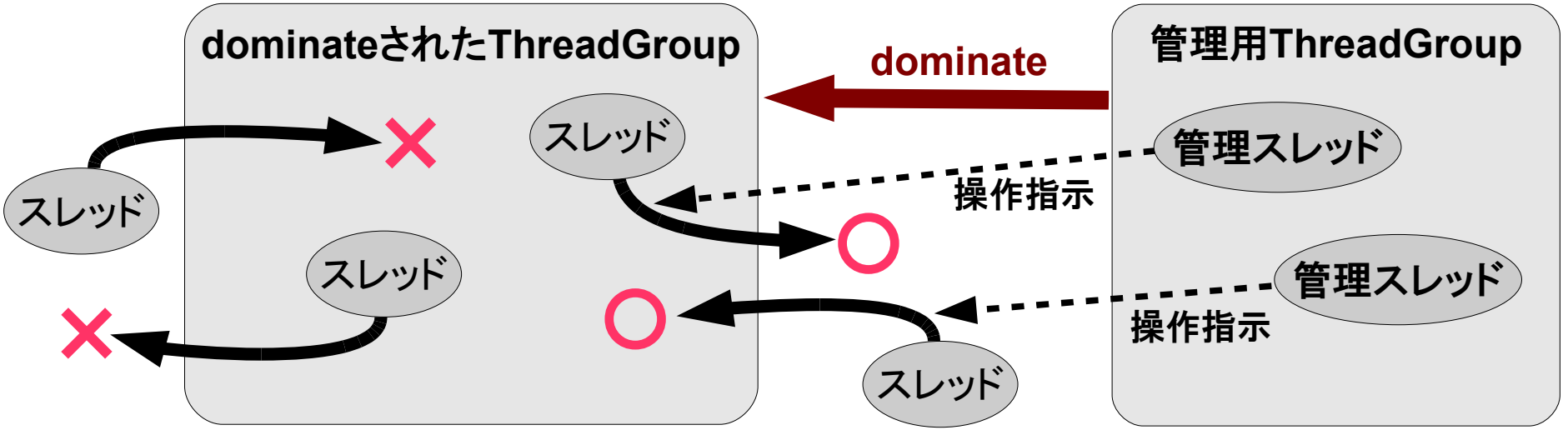
encloseされたThreadGroupとその管理

- 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

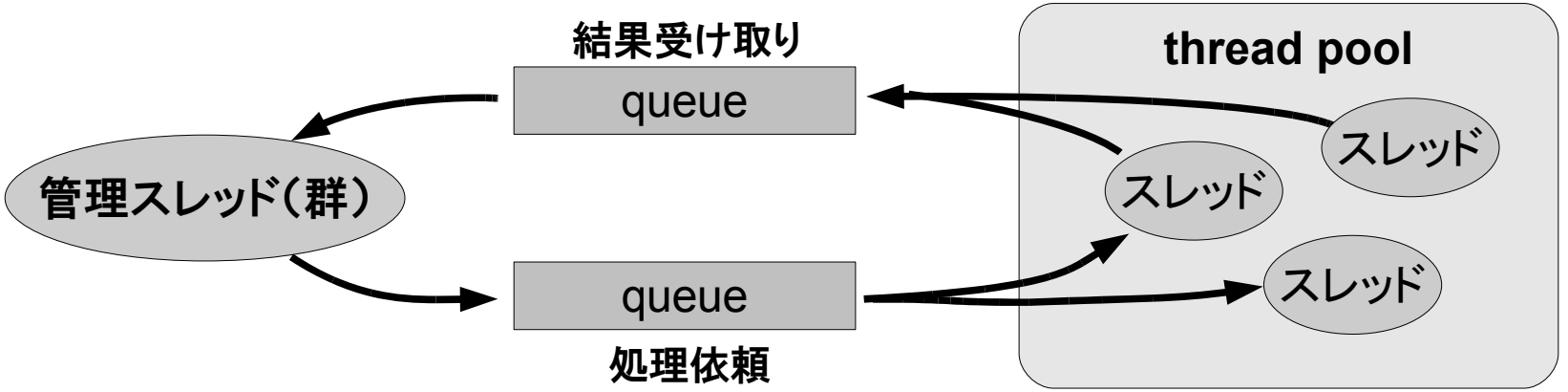


ThreadGroup強化のポイント

- コンテナとして
  - スレッド（群）の操作性の向上
- 管理母体として
  - 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）
- 共通環境として
  - ThreadGroup固有データのサポート
  - 閉鎖された実行空間の生成（local space）

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投入と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

- コンテナとして
  - スレッド（群）の操作性の向上
- 管理母体として
  - 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）
- 共通環境として
  - ThreadGroup固有データのサポート
  - 閉鎖された実行空間の生成（local space）

ThreadGroup固有データ

- Threadの固有データと同様に，ThreadGroupにも固有データを導入
  - ThreadGroup#[ ], ThreadGroup#[ ]=
  - スレッド群に共通の環境変数のようなもの

ThreadGroup強化のポイント

- コンテナとして
  - スレッド（群）の操作性の向上
- 管理母体として
  - 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）
- 共通環境として
  - ThreadGroup固有データのサポート
  - 閉鎖された実行空間の生成（local space）

ThreadGroupのlocal space

- 一種のダイナミックスコープ
  - ThreadGroup依存でローカルな定義が存在すれば，そちらを優先する仕組み
  - 存在しなければグローバルな定義が参照される
- 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（他のスレッドによる操作）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の共存

local spaceによる環境共存の例

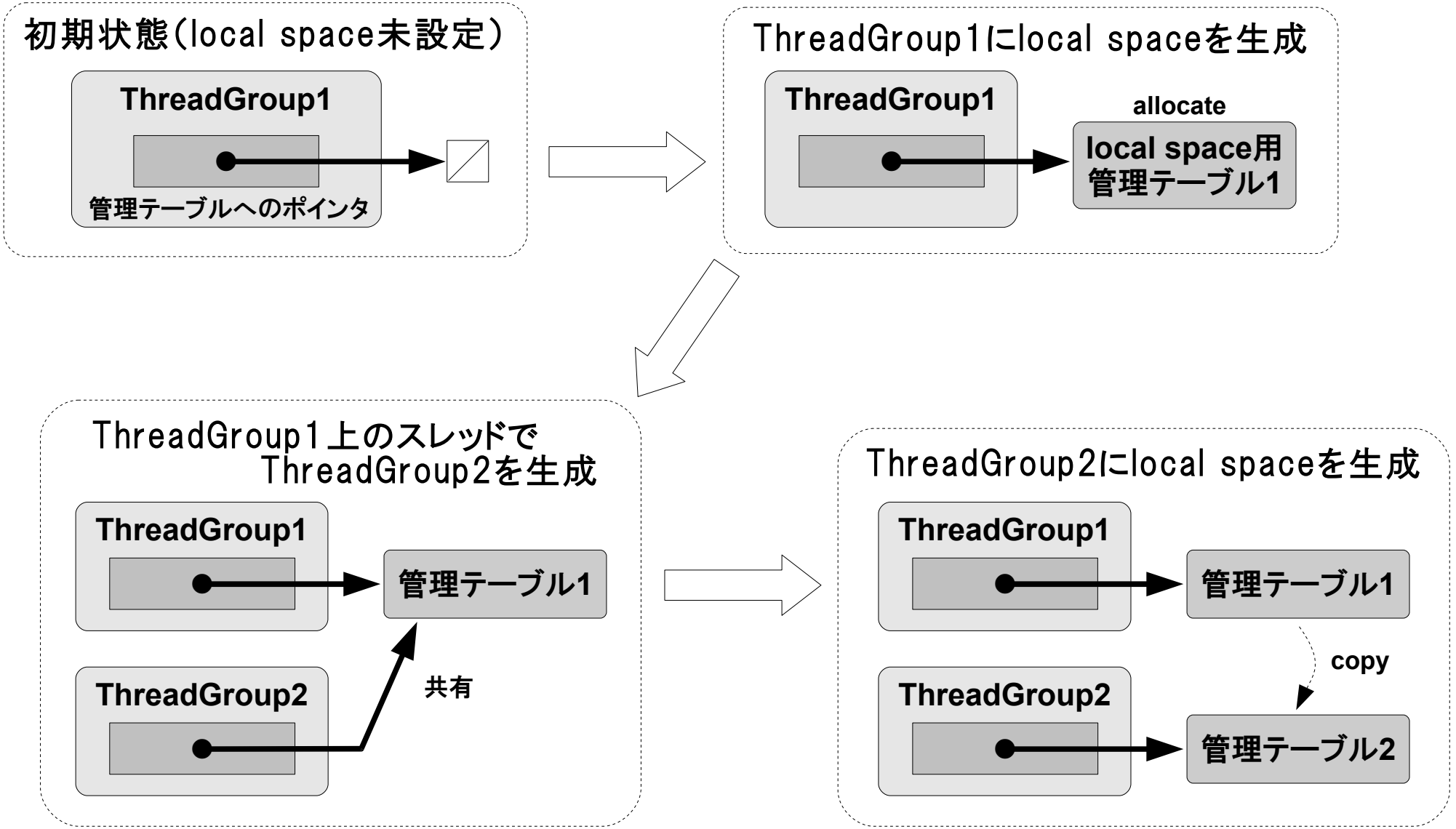
- mathnライブラリの有効/無効の共存を模した動作例
  （スクリプトの実行テスト環境の動作例として見てもよい）

```
# 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)
```

実装方法とコスト

- 実装方法
  - ThreadGroupオブジェクトの構造体に情報を持たせる
  - クラス/モジュールをキーとしたhashにより，メソッド等の追加・変更分のみのテーブルを管理
  - ThreadGroupが持つテーブルを優先して検索
- 処理コストの増加内容
  - ThreadGroupはカレントスレッドから簡単に獲得可能
  - local spaceを持つかどうかの条件分岐
  - 追加・変更分のテーブルの有無の検索
  - メソッド等のローカル定義の有無の検索

local spaceの生成と引き継ぎ



local spaceの解放

- ThreadGroupが参照を失う → local spaceも参照不能
  - ThreadGroupがGCに回収されるときに，local spaceの登録内容（共有されていなければ）も解放する
  - 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
- local spaceに読み込んだライブラリをGCで回収して，使用していたメモリを解放できる可能性あり
  - 十分な検討はしていないが，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…

なぜThreadGroupに組み込むのか？

- モジュールによる定義封じ込めの拡張ではダメか？
  - 関数的メソッドの定義が通常のメソッド定義となり，関数的メソッドとしてのスコープが得られない
  - module_eval等の際のスコープで混乱する
- 他では実現不可能というほどの強い必然性はない
  - 現在はどの定義空間で動いているかが分かりやすそう
  - 実行後に定義空間の独立や変更が可能なのは面白そう
- 定義空間管理のための新たなクラスの導入は？
  - それならそれで構わない
  - 事後変更を可能にするなら対象指定の管理単位はスレッドになりそうであり，それならThreadGroupで十分に思える

おわりに

- RubyのThreadGroupの機能拡張の一案を示した
- 現在は実装未完成
  - 部分的な実装により，実装可能性は確認
  - local spaceを導入した場合の速度低下量は未評価
- Rubyコミュニティでの評判はあまり良くない
  - ThreadGroup強化の必要性については賛意あり
  - 一部を除き反対が強い（特にlocal space）
  - 具体的用例を求める声が多い
- とにかくも用例提示と実装完了とが必要