Ruby の ThreadGroup クラスの機能拡張の試みについて

永 井 秀 利†

ThreadGroup クラスは Ruby の標準組み込みクラスの一つであり，実行中の Ruby インタープリタは常に一つ以上の ThreadGroup オブジェクトを持つ．そのような存在であるにもかかわらず，ThreadGroup クラスは十分に活用されているとは言い難い．その原因の一つは，ThreadGroup クラスの機能不足にあると言えよう．そこで本稿では，ThreadGroup という存在に想定しうる役割として「スレッド群のコンテナ」,「スレッド群の管理母体」,「スレッド群の共通実行環境」という 3 つを考え，それらの観点に基づいて Ruby の ThreadGroup クラスの現状と強化案を述べる．

A Plan to Improve ThreadGroup Class in Ruby

Hidetoshi Nagai†

In Ruby language, ThreadGroup class is one of standard classes. Although at least one ThreadGroup object exists on a running Ruby interpreter, it is difficult to say that ThreadGroup class is fully utilized in Ruby scripts. One of the reason of why is poor ability of ThreadGroup class. In this paper, I discuss about ThreadGroup class in three types of usages, that is, “a container of threads”, “an administrator of threads”, and “a common environment of threads”. From those points of view, I describe current status of Ruby’s ThreadGroup class and a plan to improve the class.

1. はじめに

オブジェクト指向スクリプト言語 Ruby において，ThreadGroup クラスは標準組み込みのクラスの一つである．Ruby のスレッドは必ずいずれかの ThreadGroup に所属するため，Ruby スクリプトが実行される (一つ以上のスレッドが稼動している) 際には少なくとも 1 個の ThreadGroup オブジェクトが存在している．

そのように必ず存在するオブジェクトであるにも関わらず，現状の Ruby の ThreadGroup オブジェクトは十分に活用されているとは言い難い．その原因の最たるものは，現在の ThreadGroup オブジェクトがスレッドの集合を扱うための最低限の機能しか有していないことであろう．

そこで本稿では，ThreadGroup という存在の活用可能性の模索を行いつつ，Ruby の ThreadGroup クラスをより有効に活用しやすくするための一案としての機能強化の試みについて述べる．

† 九州工業大学
Kyushu Institute of Technology
nagai@ai.kyutech.ac.jp

2. Ruby の ThreadGroup クラス

ThreadGroup の一般的な役割は，一言で言えば「スレッドを束ねるもの」である．Ruby の ThreadGroup は，あるスレッドが生成した子スレッドをひとまとめにしておいて，一度に強制終了させるようなユースケースを念頭において設計[2] されている．スレッドの集合を配列に格納して束ねる場合と ThreadGroup で管理する場合との違いは，新たに生成された子スレッドは親スレッドの ThreadGroup を引き継ぐ (自動的に親と同じ ThreadGroup に所属する) 点と，終了したスレッドは TheadGroup による操作対象からは外れる (所属する ThreadGroup は維持される) という点である．これにより，単に配列に格納している場合と異なり，スレッドの増減の捕捉と配列格納状態の管理とにかけるコストを減ずることができる．

Ruby のスレッドは必ずいずれか一つの ThreadGroup に属する仕組みとなっている．Ruby 起動時の唯一のスレッドである main スレッドは，デフォルトで ThreadGroup::Default という定数で管理された起動時唯一の ThreadGroup オブジェクトに所属する．

Ruby の ThreadGroup の特徴の一つとして，スレッ

ドが所属する ThreadGroup を動的に変更可能であることが挙げられる．これは main スレッドにおいても例外ではなく，main スレッドが唯一のスレッドである状況であっても，新たな ThreadGroup オブジェクトを生成して自らその ThreadGroup の所属に移行する (当然，ThreadGroup::Default 所属のスレッドは存在しなくなる) ことも可能である．しかしながら，スレッド生成時に所属する ThreadGroup を指定することはできず，少なくとも一時的には親と同じ ThreadGroup に所属せざるを得ない．

Ruby の ThreadGroup オブジェクト間に関係や階層はなく，スレッド移行元や移行先の ThreadGrouop に関して権限等の概念は存在しない．また，Ruby のスレッド間にも移行権限の上下関係はないため，子スレッドが親スレッドの ThreadGroup を変更することも許される．スレッド移行の制約となるのは，移行元ないし移行先の ThreadGroup オブジェクトの状態のみである．

他言語の ThreadGroup の例として，しばしば Ruby と対比される Java と Python とでの例を見る．

Java にも ThreadGroup クラス[4] は存在し，Ruby の ThreadGroup よりも少しだけ多機能なものとなっている．Java の ThreadGroup オブジェクト間には階層関係が存在する．スレッドが所属する ThreadGroup はスレッド生成時に引数で指定され，後で変更することはできない．特に指定がなければ親スレッドと同じ ThreadGroup に所属する．

Python の ThreadGroup[5] は，スレッド生成時の予約引数としては存在しているものの，現状では未実装である．指定のスタイルは Java と類似しており，Java を意識した実装にすることを想定しているものと推察される．

3. ThreadGroup に想定可能な役割と現状

ThreadGroup という存在に想定しうる役割について考える．

ThreadGroup というものに対する極めて一般的な認識は，「スレッド群のコンテナ」であろう．コンテナに収容する対象がスレッドであることから，管理下のスレッドを確実に拘束し，状態変化の監視や捕捉を行うための「スレッド群の管理母体」としての役割も期待される．逆にスレッド側からの視点で見ると，ThreadGroup は複数のスレッドによって共有された存在とも見なせる．これは，子スレッドにも引き継がれる「スレッド群の共通実行環境」として捉えることが可能である．

```
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
```

図 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する例

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

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

Ruby の ThreadGroup の現状をそれぞれの役割の観点で見る．

3.1 コンテナとして見た場合の現状

子スレッド生成時の所属や終了スレッドの除外のような自動的操作はあるが，所属するスレッド群を明示的に操作するためのインスタンスメソッドはほとんど存在しない．存在するのは，指定したスレッドを追加する `ThreadGroup#add` と，所属するスレッドの内で生きているスレッドの集合を返す `ThreadGroup#list` くらいである．スレッドを束ねるものでありながら，それを集合として操作するメソッドを持たない．

`ThreadGroup#add` を用いたスレッド追加にも問題がある．現状の Ruby ではスレッド生成時に ThreadGroup を指定する手段がないため，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．それゆえ，現状で指定した ThreadGroup にスレッドを生成するには，例えば図 1 や図 2 のような手段を取る必要があり，利便性を欠く．

3.2 管理母体として見た場合の現状

Ruby のスレッドは所属する ThreadGroup を動的に変更できるため，同時にそれを制限する機構も有する．この制限は ThreadGroup オブジェクトの状態に

よって規定される．

ThreadGroup オブジェクトは，通常 (normal) 状態の他に freeze された状態と enclose された状態とを持つ．ThreadGroup オブジェクトが freeze された場合，その ThreadGroup への新たなスレッドの追加やその ThreadGroup から他の ThreadGroup へのスレッド移行はできなくなる．これはその ThreadGroup 内のスレッドであっても同じであり，新たな子スレッドを生成しようとした場合は例外を発生する．

ThreadGroup オブジェクトの freeze は，もともとの設計思想である「スレッド群を一度に強制終了」に基づくものであると言える．すなわち，強制終了対象スレッドによる新たなスレッドの生成を防止し，対象スレッド群を確実にすべて終了させることが目的であろう．freeze 状態の ThreadGroup オブジェクトの再利用は事実上不可能となるが，だからと言って通常状態の ThreadGroup オブジェクトのまま利用したのではスレッド移行を防止することはできず，スレッド群の拘束の確実性に欠ける．そうした際に有用なのが ThreadGroup の enclose である．enclose された ThreadGroup では freeze 時と同様にスレッドの出入りが禁止されるが，その ThreadGroup 内のスレッドが子スレッドを生成することは可能である．

スレッドを管理する権限を持つ存在として ThreadGroup を見た場合，管理機構が安全に機能するためにはその ThreadGroup オブジェクトに対する操作権限も重要となる．しかしながら，現状では ThreadGroup の操作権限に関する概念は欠如している．

ThreadGroup の管理対象となるスレッド集合の変化は，スレッドが終了した際にも発生する．それに即応して何らかの処理を行いたいという要求は十分にありうる．スレッド終了の監視対象が一つのスレッドのみであれば簡単だが，対象が複数のスレッドである場合に各スレッドの終了に即応する手段は，直接の機能としては用意されていない．監視側の処理と組み合わせて対象となる個々のスレッドの処理内容で対策を施しておかない限り，polling による監視を行ったり，対象スレッドごとに個別に監視スレッドを用意したりが必要となる．

3.3 共通環境として見た場合の現状

Ruby のスレッドは，そのスレッドに固有のデータを保持するための仕組みを持つが，これは「そのスレッド上」という環境における環境変数として捉えることができる．ThreadGroup オブジェクトを，それに属するスレッド群に跨る共通環境と捉えるなら，スレッド固有データと同様に，その環境の特徴あるいは属性を規定するような環境変数としての ThreadGroup 固有データが欲しいと考えても不自然ではあるまい．しかしながら現状では，そうした機能は組み込まれていない．

ThreadGroup オブジェクトの enclose や freeze は，その ThreadGroup オブジェクトによって規定される環境への封じ込めとして機能させることができるが，それ以外には ThreadGroup オブジェクトを環境として取り扱うことを支援する機能はない．

4. コンテナとしての役割における強化

ThreadGroup をコンテナとして捉えた場合の強化のポイントは，所属するスレッド (群) の操作性の向上である．

4.1 スレッドの生成と所属

問題は，スレッド生成と ThreadGroup 設定とをプリミティブに実行する手段がないことである．この点は Ruby スクリプトのレベルでは解決不可能であるため，Ruby インタープリタのソースレベルでの実装が必要となる．

Ruby のスレッドのコンストラクタ (`Thread.new` メソッド) が取る引数は，スレッドの開始時の値を確実にそのスレッド固有のローカル変数に渡すために用いられる．互換性を考えると，ThreadGroup 指定を Java や Python と同様の形で単純に追加する方法は選択しづらい．もちろん，ThreadGroup 指定を強いる別種のコンストラクタメソッドを Thread クラスに追加する選択肢はある．しかし，現状の Ruby では ThreadGroup 指定を `ThreadGroup#add` メソッドで行っている (ThreadGroup が思考の中心となっている) ことを鑑みると，ThreadGroup にスレッド生成メソッドを追加するほうが望ましいと考える．

そこで，新しいスレッドを特定の ThreadGroup に所属するように生成するような `ThreadGroup#new_thread(*`*arg*`){|*`*arg*`| …}` を ThreadGroup クラスのインスタンスメソッドとして導入する．引数は `Thread.new` メソッドと同じとする．

このメソッドは，スレッドとして Thread クラスのオブジェクトを生成することを想定しているが，Thread クラスのサブクラスのオブジェクトを生成したい場合もありうる．Ruby の Thread クラスはオープンクラスであることもあり Thread クラスのサブクラスを作ってまで用いねばならないケースは稀と考えるが，対応は必要であろう．

今回の強化案では，`ThreadGroup#new_thead` の延長として `ThreadGroup#new_thead_of` メソッドを導

入することとする．第 1 引数に生成したいスレッドのクラス (Thread クラスのサブクラス) オブジェクトを指定する点を除いては，`ThreadGroup#new_thead` メソッドと同じ引数を取る．

Thread クラスのサブクラスまで考えるのであれば，例えば `Thread.new_to(`*thgroup*`, *`*arg`){|*`*arg*`| … }` というように ThreadGroup 指定が可能なコンストラクタを Thread クラスに追加する方が素直かもしれない．だが，Ruby ではコンテナとしての ThreadGroup オブジェクトの方に意識としての操作の主体が置かれているように見受けられること，および，Thread クラスのサブクラスの必要性は低そうであることから，本案では ThreadGroup クラスへの実装を選択している．

4.2 スレッド集合としての操作メソッドの追加

ThreadGroup に属するスレッド集合の操作を単純な集合要素の操作と隔てているのは，スレッド終了や子スレッド生成により，操作中にもスレッド増減が生じる可能性がある点である．ThreadGroup を freeze すればスレッドの増加は防止できるが，その ThreadGroup の再利用は難しくなる．よって，機能強化のためのメソッドは，メソッド呼出し時点のスレッド集合を対象とするものとメソッド呼出し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とに分けることができる．

メソッド呼出し時点のスレッド集合を対象とする追加メソッドとしては，各スレッドを対象とした繰り返しである `ThreadGroup#each{ … }` や各スレッドの例外を発生される `ThreadGroup#raise` などが挙げられる．それに対し，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べき追加メソッドは，スレッドの終了待ちを行う `ThreadGroup#join` やスレッドを強制終了する `ThreadGroup#kill` などである．

これらに加えて今回の強化案では，ThreadGroup に所属可能なスレッドの数を制限するための `ThreadGroup#max_threads` や `ThreadGorup#max_threads=`を追加する．これらは thread bomb 対策として機能することを期待している．

5. 管理母体としての役割における強化

5.1 ThreadGroup 間の操作権限の設定

特定目的のスレッド群を一つの ThreadGroup オブジェクトで管理する際，その ThreadGroup オブジェクトを enclose することは有用である．enclose によって，管理対象たるスレッドが他の ThreadGroup へと

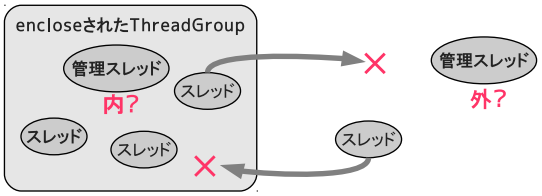

図 3 管理用スレッドを置くべき位置の問題

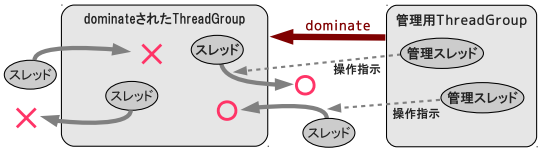

図 4 domminate された ThreadGroup

離脱したり，そのスレッド群の目的とは無関係なスレッドが混ぜ込まれたりすることを防止できる．ただし，そうしたスレッド群に対して管理用のスレッドが必要である場合，それをどこに置くべきか (図 3) は問題となりうる． enclose された ThreadGroup の内側に管理用スレッドを置くことは，管理側のスレッドと被管理側のスレッドとが同列に存在することになるため，あまり嬉しくない．しかし ThreadGroup の外部に置いた場合には，enclose の制約により，必要に応じて新たなスレッドを生成するなどの種類の管理は不可能となる．

そこで本強化案では，ThreadGroup の状態として「dominate」という状態 (図 4) を新設する．これは ThreadGroup の操作権限を定めるためのものである．Ruby の ThreadGroup には階層関係がないが，dominate 状態は ThreadGroup 間に動的に変更可能な支配関係を与える．

dominate された ThreadGroup は，ほとんどの点で enclose された状態と同等である．違いは，その ThreadGroup を dominate している ThreadGroup が支配権を持ち，dominate している ThreadGroup に所属するスレッドであれば dominate されている ThreadGroup にスレッドの生成や移動を実行できることである．

dominate 関係は ThreadGroup 単位での指定であるため，ThreadGroup 管理に必要なスレッドが複数存在したとしても，その権限管理を容易に行える．dominate の導入により，ThreadGroup が取る状態は normal，dominated，enclosed，frozen の 4 種となり，この順に制約が厳しくなる．また，制約がゆるい状態から厳しい状態への変更は可能だが，その逆は不可となっている．

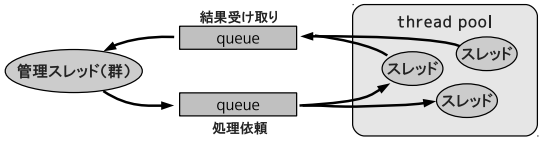

図 5 Ruby での thread pool 実装にありがちなスタイルの例

5.2 例外終了を含むスレッドの整列化

複数スレッドに処理を分散する場合を考える．Ruby において thread pool のスレッドに処理を割り振る際にありがちなスタイルの例を図 5 に示す．

処理分散におけるスケジューリング等の管理の主体は管理スレッド側にあるべきだが，この例のような場合，管理の仕組みを処理スレッドから完全には引きはがすことはできない．管理スレッドは処理スレッドを統制しているとは言えず，処理スレッド群に盲目的に依頼を出しているに過ぎない．処理スレッドが依頼を吸い上げて正常に処理してもらえて初めて全体がうまく働くため，スケジューリングの主体は処理スレッド群が握っているに等しい．

例えば，処理スレッドが例外終了した場合にはそれを捕捉して速やかに何らかの対処を行うことが必要となるはずだが，その即応性の確保は簡単ではない．polling で監視するのは busy loop となるため CPU コストの無駄が生じる．だが polling を避けるには，個々の処理スレッドをひとつひとつ監視専用スレッドで包んでやる必要があり，スレッド数の倍増を招く．そうでなければ，主導権を処理スレッド側に渡してしまい，各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ようにせざるを得ない．

そこで今回の強化案では，thread queue というものを導入する．これは，終了したスレッドの捕捉を主たる目的としたもので，管理スレッド側に主導権を持たせやすくするための仕組みである．

thread queue は，ThreadGroup オブジェクトごとに 1 個まで利用可能とする．各 ThreadGroup オブジェクトが thread queue を利用するかどうかは任意である．

その設置目的に基づき，thread queue は終了したスレッドが終了順に送り込まれる対象として位置付ける．ただし，活用の柔軟性を増すために，対象となるスレッドを「正常終了のみ」や「例外終了のみ」などと設定することも可能とする．Queue クラスのオブジェクトからの要素取り出しの場合と同様に，thread queue からのスレッド取り出し要求の際に thread queue が空であれば，thread queue にスレッドが送り込まれるまで待ち状態となる．これにより，終了したスレッドを終了順に即応性を持って処理することが，処理負荷をあまり増大させることなく可能となる．

```
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, " ... "
```

図 6 並列で問い合わせを行い，最初に得られた結果を利用する例

さらに，`ThreadGroup.queueing` メソッドにより，現在のスレッドが sleep すると同時に thread queue に入ることも可能とする．現在の Ruby では，スレッドが自身の sleep と同時に，その事実を他所に伝達するための手段が用意されていない．そのため，管理するスレッド側が安全に処理を行うためには，sleep したとの情報を何らかの形で受け取ったとしても，本当に sleep しているかどうかを確認することが必要であった．`ThreadGroup.queueing` メソッドによって thread queue への投入と sleep とがプリミティブとなることで，thread queue から取り出したスレッドの状態確認の手間を不要にすることができる．

処理スレッドが `ThreadGroup.queueing` メソッドを呼んだ時に投入される thread queue は，そのスレッドの現在の ThreadGroup の thread queue と決まっているので，処理スレッド側が投入先を決める権限はなく，投入先を知る必要もない．よって，dominate と組み合わせることで，どの thread queue に入るかの決定権を管理スレッド側が握ることができる．

今回の強化案によるメソッドを用いた thread queue 利用法の例を図 6 と図 7 とに示す．

6. 共通環境としての役割における強化

6.1 ThreadGroup 固有データのサポート

スレッドの固有データと同様に ThreadGroup にも固有データを導入し，操作用のメソッドとして `ThreadGroup#[]` と `ThreadGroup#[]=`とを追加する．ThreadGroup の固有データは，スレッド群に共通の環境変数のようなものとして扱うことができる．

```
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
}
```

図 7 スケジューリング権限を管理スレッド側に持たせた thread pool 実装の例

6.2 閉鎖された実行空間の生成

ThreadGroup に，固有の実行空間となる local space を生成する機能を追加する．特定の ThreadGroup オブジェクトに local space を設定するかどうかは任意であり，デフォルトでは設定しない．

local space は一種のダイナミックスコープである．ThreadGroup 内に閉鎖された実行空間 (環境) を構築し，その空間だけで有効なメソッドや定数，クラス (名) やモジュール (名) といったものを実現する．あるスレッドの実行時のメソッド呼び出し等において，そのスレッドが属する ThreadGroup が local space を持ち，かつ，そこにローカルな定義が存在しているならばそちらを優先し，存在しなければグローバルな定義を参照するという仕組みとする．したがって local space が設定された後であっても，ローカルな定義がなされていない範囲では，グローバルに行われたクラス定義や変更などが問題なく反映される．

Ruby において事実上グローバルな定義となる関数的メソッドを local space で定義することも可能である．その場合，その関数的メソッド定義はその local space 内でのみ有効でり，たとえ local space 外に同名の関数的メソッド存在したとしても外部には影響を及ぼさない．

一般的な実行空間指定と異なるのは，動作中のスレッドが所属する ThreadGroup を変更することで依存する local space が変更されるため，能動的にも受動 (他のスレッドによる操作) 的にも動的に実行環境を変更することができる点である．ThreadGroup オブジェクトの enclose ないし dominate によって，動的変更の是非や権限を制御することも可能である．

local space の用途としては，例えば次のようなものが考えられる．

- **スクリプトの実行テスト環境として**
  local space の中であれば，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ことを利用する．
- **sandbox 作成の支援**
  もちろん local space のみで sandbox が作れるわけではないが，Ruby で従来用いられてきた safe level による保護 (実行防止) とは方向が違う保護 (汚染防止) であるため，sandbox の作成を支援する機能の一つとして活用できる．
- **衝突する複数の定義を個別の実行環境で共存させるために**
  例えば Ruby の mathn ライブラリはグローバルに影響を及ぼす．local space を使えば，同ライブラリが有効な環境と無効な環境とを共存させることができる．

local space による環境共存の例として，mathn ライブラリの有効/無効の共存を模した動作例を図 8 に示す．これは，スクリプトの実行テスト環境の動作例としても見ることもできる．

実装において，local space の情報は ThreadGroup オブジェクトの構造体に持たせる．例えばメソッド定義は，クラスまたはモジュールをキーとしたハッシュにより，そのクラスまたはモジュールの追加・変更分のみを納めたテーブル (メソッドの場合はメソッド名

```
# 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)
```

図 8 ThreadGroup の local space による環境共存の例

をキーとしたハッシュ) を管理する．ThreadGroup が持つそうしたテーブルを，メソッド検索等の際に優先して検索することで，local space の登録内容に優先権を与える．

実行中の ThreadGroup オブジェクトは，現在のスレッドの情報から直接に獲得可能であるため，local space の情報に到達するためのコストは小さい．local space を持つ場合，メソッド検索のためにクラスの継承パスを辿る途中でそのクラスまたはモジュールオブジェクトをキーとして local space の追加・変更分のテーブルの有無を検索し，もし存在するならば，メソッド名をキーとしたハッシュ上でローカル定義の有無を検索する．各 ThreadGroup オブジェクトの local space 間には依存関係はないため，ThreadGroup を辿るような検索処理は不要である．

local space を持つ ThreadGroup のスレッドが新たな ThreadGroup を生成する場合，生成を行うスレッドは local space の存在を意識すべきではない．その際，一方での ThreadGroup に属するスレッドで実行した変更が他方でも適切に反映されるように，現在の local space が両 ThreadGroup で共有されるようにする必要がある．ただし，新たな ThreadGroup で local space の利用が宣言された際には，それまでの local space を初期状態とするようにコピーして独立させる．この過程を図 9 に示す．

ThreadGroup が参照を失うとき，local space も参照不能となる．よって，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やモジュールは，名前を

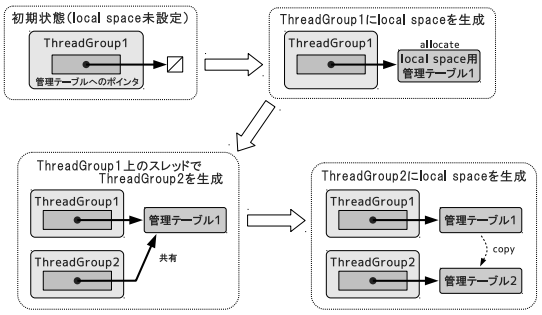

図 9 ThreadGroup の local space の生成と引き継ぎ

失って無名クラスや無名モジュールとなる．もしそのクラスやモジュールに対する参照もオブジェクトも存在しなければ，それらも GC で回収されることになろう．これは，ライブラリを local space に読み込むようにすることにより，不要になった際に消去してメモリを解放できるようになる可能性があることを示唆する．

local space のような機能を組み込む場所が ThreadGroup であることについて，他の方法では実現不可能とまでは言えない．しかしながら，ThreadGroup とスレッドとの関係からあるスレッドがある瞬間にどの定義空間で動作しているかを容易に確認でき，スレッド動作中であっても定義空間の独立や変更が可能であるということは一つの特徴であると考える．

7. おわりに

Ruby の ThreadGroup の機能拡張の一案を示した．誌面の都合で掲載できなかったメソッド等は Ruby の開発者向けメーリングリストに流したメール[3] を参照されたい．現在は実装未完成であるが，部分的な実装によって実装の可能性は確認できている．もし local space を導入した場合には速度低下が生じるはずだが，その具体的な量については未評価である．ただし，増加する処理内容を鑑み，問題となるほどの速度低下はないと予想している．

本強化案に対する Ruby コミュニティでの評判は，残念ながらあまり良くない．ThreadGroup クラスの強化の必要性については賛意があるものの，その内容については一部を除いて反対 (特に local space に対して) が強い．具体的用例を求める声が多いため，とにかくも用例提示と実装完了とが必要と言える．

質疑応答

Q (東大・田中) スレッドをまとめて扱うのはまともに動くときは便利だが，まとめて処理 (kill とか

join とか) するときに途中で失敗するときはどうするのか？

A まだ明確には決めていない．強行して最後まで処理してしまうのか，その時点で終了するのか．はたまた，例外となったスレッドのリストを返すのか．多分，結局は例外を返して呼び出した側に対応を決めてもらうしかなさそう．

Q (東大・田中) そうなってしまうと，まとめて呼ぶ価値が小さくなってしまうのでは？

A その可能性はある．
[事後補足] 異常時にはその時点で中断して例外発生とすべきで，そのようにしても問題なさそう．現状の Thread クラスの kill メソッドは例外を生じないし，join で生じる例外はデッドロックとタイムアウトなので処理を打ち切って対応するしかできそうにない．また，対処後にメソッドを再度呼び出しても，そのことが問題になることはない．

Q (九工大・小出)Ruby ならではの機能を生かして実装している部分はないか？

A スレッドが所属する ThreadGroup を動的に変更可能な点以外では，他の言語での method table とかインスタンス変数管理などの実装を把握していないので比較しにくい．ただ，クラスのオブジェクトが持つテーブルとの差分だけを保持してメソッド等の検索の際に差分の有無をチェックするという形で割と簡単に実装できているとは思う．

Q (九工大・小出)Ruby コミュニティでの評判は良くないとのことだが．今の ThreadGroup の機能で十分だと考えているのか？

A ThreadGroup 強化の必要性自体には賛意がある．local space のようなものを ThreadGrouop に実装することに対してはやり過ぎとの批判が強い．必要なものとそうでないものとを用例に基づいて評価し，どこまでやるのかを考えようという意見もある．

Q (日本ソフトウェア・八木原) 動的に ThreadGroup を切り替えて，参照している local space を切り替えるというのは面白そうだし，他の手法ではできないことができそうな気はするが，アイディアレベルでもいいので具体的な用例はないか？

A 良い例は思いついていない．例えば，特定のメソッドを呼んで処理した後 sleep するだけの小さくて軽いスレッドを複数生成して pool しておき，ThreadGroup 変更で対応する処理を切り替えて wakeup することで，各役割に投入するスレッドの量を負荷状況に応じて変更するというようなことは考えられる．

Q (日本ソフトウェア・八木原) いきなり切り替えるというのは，同期が大変にならないか？何か対策は考えているか？

A 今のところは特に考えていない．local space はメソッド検索の段階に影響するものであるため，いきなり切り替えても，その時点で実行中のメソッドそのものには影響を及ぼさない．
[事後補足] 連続して同じメソッドを呼んでいるはずが，違ったメソッド実体を呼んでしまうようなことは生じうる．ただ，スレッド自身が能動的に ThreadGroup を変更するのではなく，外部の管理スレッドによる操作でそのような切り替えを必要とするのは，本当に緊急時の特殊な状況のみであろう．管理スレッドによる通常の操作としては，sleep しているスレッドに対して切り替えを行うことを想定しており，その場合には同期について心配する必要はないと考える．

Q (電通大・岩崎)ThreadGroup を使わない場合と使った場合とで，これこれこういう点で嬉しいよという形で説明してもらえると嬉しい．

A 例えば thread queue を使った管理について，管理スレッドと処理スレッドとの処理内容がきちんと連携して作られいればよいわけで，thread queue を使わなければ実装できないというわけではない．ただ，枠組みとして用意してあれば，管理スレッドや個々の処理スレッドの記述を省けて定型的に簡潔に記述できるなどの利点があると考える．

Q (電通大・寺田) unix のプロセスグループ，KLI の資源管理などを思い出した．使える heap の制限をかけるなどの役割を持たせるのも良いかも．

A 勉強させていただきます．ありがとうございました．

参 考 文 献

1) Ruby 1.9.2 リファレンスマニュアル (http://doc.ruby-lang.org/ja/1.9.2/doc/index.html)
2) Ruby 開発者向け公式メーリングリスト, メール ID [ruby-dev:43994] (http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-dev/43994)
3) Ruby 開発者向け公式メーリングリスト, メール ID [ruby-dev:43901] (http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-dev/43901)
4) Java マニュアル, http://java.sun.com/j2se/1.5.0/ja/docs /ja/api/overview-summary.html 他
5) Python マニュアル, http://docs.python.org/reference/ index.html 他