em-synchrony について

2011年8月6日

cuzic

自己紹介

- cuzic といいます
  - きゅーじっく　と読みます
- Ruby 暦は かれこれもう１０年くらい
- 近況
  - Nook Simple Touch というデバイスを買いました
  - Android 2.1 が動く、ディスプレイが e-ink の端末
  - 定期巡回している Web 記事を快適に読めてしあわせ
- 今後の勉強会予定
  - 8月20日（土） Coders At Work 読書会 #2
    - 今回と同じ会場です
    - ２週間後！
    - みんな来てください。

今日の議題

従来の並列処理

- 複数処理の同時並行実行について
- マルチスレッドによる同時並行処理
- IO多重化 による同時並行処理

event-machine

- eventmachine とは
- コールバック登録による非同期処理

em-synchrony

- Fiber とは
- 非同期処理の逐次処理的な記述

同時並行で処理を進めて高速化したい！

- 過剰な待ち時間のために、動作が遅かったりします
  - 例） WEBダウンロード
    - ネットワークIO の待ち時間の間は CPU は働きません
    - CPU としては軽い処理なのに、もったいない！

  ⇒ 多数の URL からのダウンロードであれば、同時並行でダウンロードさせれば、高速化可能。
- 同時に処理させる方法の例
  - マルチスレッド
    - スレッド特有の問題が多数存在
      （排他制御、デッドロック、変数の同期 etc）
  - IO 多重化（select(2) の利用など）
    - 状態の管理が大変。慣れるまでは苦労する。
  - マルチプロセス
    - 今回は説明しません

マルチスレッドによる同時並行処理

- マルチスレッドの長所
  - 書きやすい
    同期処理的に書ける
  - （Ruby の場合は）
    自動的にIO 多重化を
    いいかんじにできる
  - （Ruby の場合）たいてい、
    マルチスレッドで十分！
- マルチスレッドの短所
  - スレッド特有の多々の問題
    - 排他制御
    - デッドロック
    - たまに起きる不可解なバグ
  - スレッド切り替えが遅い
    - コンテキスト切替が必要
  - いつ何が動作している不明

```
require 'open-uri'

urls = %W[http://localhost:3000/1
     http://localhost:3000/2 ]
urls.each do |url|
  puts "#{url} #{open(url).read}"
end
```

```
require 'open-uri'

urls = %W[http://localhost:3000/1
          http://localhost:3000/2]
threads = urls.map do |url|
  Thread.start(url) do |url|
    body = open(url).read
    puts "#{uri} #{body}"
  end
end

threads.each{|t| t.join}
```

IO多重化 による同時並行処理

- IO多重化の長所
  - 動きがすばやい
  - スレッド特有の各種問題に悩まされない
- IO多重化の短所
  - 慣れるまで書きにくい
    - 逐次的に処理を書けない
    - 慣れが必要
    - 状態遷移を管理したり、コールバックの登録が常套手段
  - select(2) を使うプログラムを書くのは結構大変

⇒ そういうときには eventmachine

AsyncHTTPClient の作成

巡回したい URL と、
読み込み後にしたい処理
(コールバック)を登録

HTTP 取得処理を実行

取得終了後、登録していた
コールバックを呼び出し

```
sockets = ソケット作成処理(urls)
loop do
  readables, = IO.select sockets
    readables.each do |s|
      バッファ = s.read_nonblock 65536
      sockets.delete s if s.eof?
    end
  end
  break if sockets.empty?
end
登録していたコールバック呼び出し
```

これからの説明

eventmachine とは → eventmachine 標準の HTTP クライアント → em-http-request による HTTPクライアント

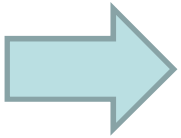
Fiber とは → Fiber を使った非同期処理の逐次的記述 → em-synchrony を使った逐次的記述

EM::Iterator のコールバックによる記述 → EM::Synchrony::Iteratorによる逐次的記述 → その他 落穂拾い

eventmachine とは

- 非常に有名なノンブロッキングIO を実現するライブラリ
  - Heroku 、 Github、 EngineYard などで利用されている
- 広範なプロトコルに対応
  - HTTP、SMTP、MySQL 、PostgreSQL、Memcached、Redis
- 内部的には select(2) による IO 多重化を実現
  - 環境次第では、epoll 、kqueue なども利用可能
    - 非常にスケーラブルな IO 多重化が可能
- 読み込み処理終了などの”イベント”に対応した処理を記述
  - 返り値は使わず、コールバックを登録する
  - コールバックの中は、処理完了後に実行される

```
ret1 = 処理1(引数)
ret2 = 処理2(ret1)
その後の処理
```



```
処理1(引数) do |ret|
  処理2(ret1) do |ret2|
    その後の処理
  end
end
```

サーバーサイドのスクリプト

- sinatra/async
  - WEBフレームワーク Sinatra の非同期 版
  - aget が get の非同期 版
- 仕様
  - /1 と /2 に応答を返す
  - ランダムな秒数待つ
  - “1” や “2” を返す

```
require 'rubygems'
require 'sinatra'
require 'sinatra/base'
require 'sinatra/async'
require 'eventmachine'

class Delayed < Sinatra::Base
  register Sinatra::Async

  aget "/1" do
    waitsec = rand * 2
    EM.add_timer waitsec do
      body {"1"}
    end
  end

  aget "/2" do
    waitsec = rand * 2
    EM.add_timer waitsec do
      body {"2"}
    end
  end
end
```

EventMachine 標準のHTTPClient

- ライブラリが数多くの機能を提供
  - イベントループ
  - HTTPプロトコル
  - コールバック呼出し処理
- client.callback ：
  - 正常取得時の処理を登録
- EM.stop_event_loop
  - イベントループを終了
- EM標準の HTTPClient はちょっと貧弱
  - リダイレクトに非対応
  - プロキシや basic認証にも非対応

```
require 'rubygems'
require 'eventmachine'
require 'uri'

urls = %W[http://localhost:3000/1
          http://localhost:3000/2]
pending = urls.length
EventMachine.run do
  Client = EM::Protocols::HttpClient
  urls.each do |url|
    uri = URI(url)
    client = Client.request(
      :host => uri.host,
      :port => uri.port,
      :request => uri.path,
    )

    client.callback do |response|
      content = response[:content]
      puts "#{url} content"
      pending -= 1
      EM.stop_event_loop if pending == 0
    end
  end
end
```

em-http-request

- EventMachine 用の非同期HTTPライブラリ
- 高機能
  - リダイレクトに追従
  - Basic 認証に対応
  - プロキシ、Sock5に対応
  - 他にもいろいろ
- 使い方はほぼ同じ
  - コールバックに終了後の処理を登録

```
require 'rubygems'
require 'eventmachine'
require 'em-http-request'

urls = %W[http://localhost:3000/1
          http://localhost:3000/2]
pending = urls.length
EventMachine.run do
  Request = EM::HttpRequest
  urls.each do |url|
    client = Request.new(url).get

    client.callback do
      response = client.response
      puts "#{url} #{response}"
      pending -= 1
      EM.stop_event_loop if pending == 0
    end
  end
end
```

Fiber とは

- Fiber とは
  - 軽量スレッド(coroutine)
  - 明示的にスレッドに移ったり、スレッドから戻ったり
- Fiber.yield arg
  - 現在のファイバーの処理を中断
  - 親ファイバーにもどる
  - Fiber#resume の引数が Fiber.yield の返り値
- Fiber#resume arg
  - そのファイバー(self) の中断していた場所から再開
  - Fiber.yield の引数が Fiber#resume の返り値

```
f1 = Fiber.new do |i|
  puts i #=> 1
  Fiber.yield 2
  5
end

f2 = Fiber.new do |i|
  puts 3 #=> 3
  j = Fiber.yield 4
  puts j #=> 6
  7
end

i = f1.resume 1
puts i #=> 2
j = f2.resume i + 1
puts j #=> 4
i = f1.resume
puts i #=> 5
j = f2.resume 6
puts j #=> 7
```

```
（表示）
1
2
3
4
5
6
7
```

em-http-request の Fiber による逐次的記述

- httpget メソッド内で、Fiber を利用
- EM.run の中は上から順に処理を記述
  - コールバックがない！
  - 逐次的に記述できている！
- いくつか問題が・・・
  - 並列処理が行われない
    - １個 取得してから次の処理を実施する
  - 自分で Fiber の処理を記述
    - 汎用的処理なので、ライブラリに切り出したい

```
require 'rubygems'
require 'eventmachine'
require 'em-http-request'
require 'fiber'

urls = %W[http://localhost:3000/1
          http://localhost:3000/2]

def httpget url
  f = Fiber.current
  client = EM::HttpRequest.new(url).get
  client.callback do
    f.resume client
  end
  return Fiber.yield
end

pending = urls.size
EM.run do
  Fiber.new do
    urls.each do |url|
      client = httpget url
      puts "#{url} #{client.response}"

      pending -= 1
      EM.stop_event_loop if pending == 0
    end
  end.resume
end
```

em-synchrony (1)

- em-synchrony を使うと
  - EM.synchrony メソッドを利用可能になる
  - get メソッドが内部的に Fiber を用いるメソッドに置き換わる
  - 同期的な記述でノンブロッキング処理を実現できる
- けど、まだ同時並行処理は行われない
  - １個ずつURLを取得する（涙）

```
require 'rubygems'
require 'eventmachine'
require 'em-http-request'
require 'em-synchrony'
require 'em-synchrony/em-http'

urls = %W[http://localhost:3000/1
          http://localhost:3000/2]

EM.synchrony do
  urls.each do |url|
    request = EM::HttpRequest.new(url)
    res = request.get.response
    puts "#{url} #{res}"
  end
  EM.stop_event_loop
end
```

em-synchrony (2)

- 複数の URL からの取得を同時並行で行うには
  - それぞれの URL からの取得処理を Fiber で囲む
  - それぞれの取得処理が同時に実行される
- 求めるものが得られた！

  できれば、1000個の URL を５個づつ並行に処理したりしたいんだけど・・・

```
require 'rubygems'
require 'eventmachine'
require 'em-http-request'
require 'em-synchrony'
require 'em-synchrony/em-http'

urls = %W[http://localhost:3000/1
          http://localhost:3000/2]

pending = urls.length
EM.synchrony do
  urls.each do |url|
    Fiber.new do
      request = EM::HttpRequest.new(url)
      response = request.get.response
      puts "#{url} #{response}"

      pending -= 1
      EM.stop_event_loop if pending == 0
    end.resume
  end
end
```

EM::Iterator のコールバックによる記述

- EM::Iterator
  - EventMachine 用のイテレータ
  - each、map、inject
  - iter.next で後続の要素を処理
  - iter.return で値を返す
  - new の第２引数で同時実行数を指定可能

```
require 'rubygems'
require 'eventmachine'
require 'em-http-request'

urls = %W[http://localhost:3000/1
          http://localhost:3000/2]

concurrency = 2
pending = urls.length
EventMachine.run do
  EM::Iterator.new(urls, concurrency).each
do |url, iter|
    client = EM::HTTPRequest.new(url).get
    client.callback do
      response = client.response
      puts "#{url} #{response}"
      pending -= 1
      EM.stop_event_loop if pending == 0
      iter.next
    end
  end
end
```

EM::Synchrony::Iterator

- EM::Synchrony::Iterator
  - Fiber を使って、each 以降の処理を、すべての要素の処理終了後に実行するもの
  - Fiber で囲むのを なくせると思っていたら そういうものではなかった。
  - 残りの処理数を管理する 変数 pending が不要

```
require 'rubygems'
require 'eventmachine'
require 'em-http-request'
require 'em-synchrony'
require 'em-synchrony/em-http'

urls = %W[http://localhost:3000/1
          http://localhost:3000/2]

concurrency = 2
EventMachine.synchrony do
  EM::Synchrony::Iterator.new(urls,
concurrency).each do |url, iter|
    Fiber.new do
      client = EM::HttpRequest.new(url)
      response = client.get.response
      puts "#{url} #{response}"
      iter.next
    end.resume
  end
  EM.stop_event_loop
end
```

落穂拾い（1）em-synchrony の他の機能

- 複数の主要プロトコルへの対応
  - redis、mysql、mongodb、memcached
  - 自分で新たなプロトコルに対応するのも容易
- EM::Synchrony::Multi
  - すべての要素が終了したときに処理を実行するときに使う
- EM::Synchrony::ConnectionPool
  - MySQL などのコネクションプーリングで利用可能
- EM::Synchrony::TCPSocket
  - TCPSocket クラスをその気になれば代替できるクラス
  - setsockopt などいくつかのメソッドが非互換
- EM::Synchrony::Thread::ConditionVariable
  - リソースの取得待ちなどが簡単に実現可能

落ち穂拾い(2) いろいろ困ったこと

- Ruby 1.9 が利用可能な環境の構築
  - Cygwin ではコンパイルできない
  - VMWare で一度チャレンジして途中で挫折
    - 理由わすれた
  - VirtualBox で再度トライしてうまくいった
    - VitualBox は VMWare よりずっと使いやすくて便利！
    - Windows プログラムの利用がなければ Cygwin よりいいかも。
- em-http-request のインストール
  - 「gem install **--pre** em-http-request」とすることが必要
  - --pre をつけると、1.0.0.beta4 がインストールされる
    - つけないと、0.3.0 がインストールされる
  - em-http-request 1.0 系列でないと em-synchrony が使えない
  - em-http-request は 0.3 と 1.0 で非互換な部分が多い

まとめ

- スレッドと IO 多重化の２方式で同時並行処理を比較
  - (Ruby の場合は）たいていスレッドで十分
  - スレッド特有のバグ、速度が遅すぎるなどの課題があれば、IO多重化（event machine など）の利用を検討
- Event Machine
  - ノンブロッキングIO を実現するライブラリ
  - 単一イベントループとコールバックによる対応処理の記述
- em-synchrony
  - 逐次的記述でノンブロッキングな処理が実現可能
  - 内部実装に、Fiber を利用
  - 複数の URL からの同時取得処理もかんたん
  - コネクションプール

ご清聴ありがとう
ございました