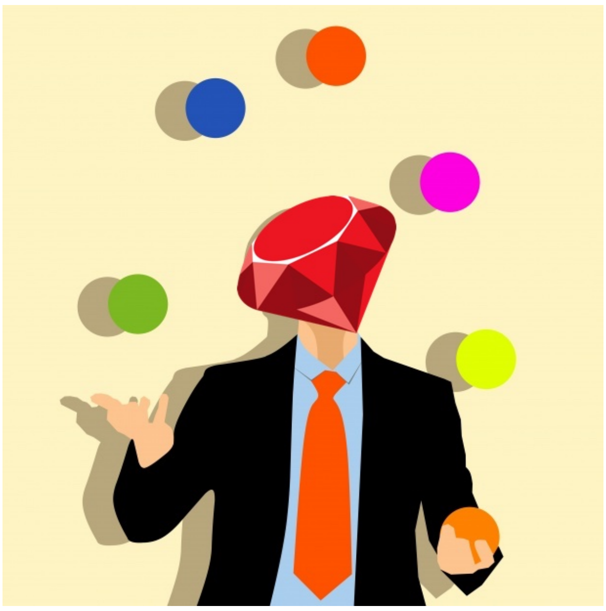
HTTP server in Ruby 3 - Fibers & Ractors

July 25, 2021



This is part #2. Head over to part #1 to learn about HTTP in Ruby.

Motivation

Historically Ruby's been lacking in the concurrency department. Ruby has "native" threads (prior to 1.9 there were only "green"), which means there can be multiple threads controlled by an OS, but only 1 thread can be executed at a time, this is managed via Global Interpreter Lock (GIL). However, native calls and I/O calls can be executed in parallel. During an I/O call, a thread gives up control and waits for a signal that the I/O has finished. This means I/O heavy applications can benefit from multiple threads.

In this article we are going to explore different concurrency modes for our HTTP server:

- Single-threaded
- Multi-threaded
- Fibers
- Ractors

Single-threaded

Let's start simple and see how a single threaded HTTP server could look like (*full code*):

```
def start
    socket = TCPServer.new(HOST, PORT)
    socket.listen(SOCKET_READ_BACKLOG)
    loop do
      conn, _addr_info = socket.accept
      request = RequestParser.call(conn)
      status, headers, body = app.call(request)
      HttpResponder.call(conn, status, headers, body)
    rescue => e
      puts e.message
    ensure
      conn&.close
    end
end
```

Multi-threaded

Like discussed earlier, threads in Ruby help performance when there's a lot of I/O. Most real-life applications are like that. Some of the I/O: read from the incoming TCP connection, call database, call an external API, respond via the TCP connection.

Now let's implement a multi-threaded server and a thread pool (*full code*):

```
def start
  pool = ThreadPool.new(size: WORKERS_COUNT)
  socket = TCPServer.new(HOST, PORT)
  socket.listen(SOCKET_READ_BACKLOG)
  loop do
    conn, _addr_info = socket.accept
    # execute the request in one of the threads
    pool.perform do
      begin
        request = RequestParser.call(conn)
        status, headers, body = app.call(request)
```

```
          HttpResponder.call(conn, status, headers, body)
        rescue => e
          puts e.message
        ensure
          conn&.close
        end
      end
    end
  ensure
    pool&.shutdown
  end
```

Thread pool (simplified implementation):

```
class ThreadPool
  attr_accessor :queue, :running, :size

  def initialize(size:)
    self.size = size

    # threadsafe queue to manage work
    self.queue = Queue.new

    size.times do
      Thread.new(self.queue) do |queue|
        # "catch" in Ruby is a lesser known
        # way to change flow of the program,
        # similar to propagating exceptions
        catch(:exit) do
          loop do
            # `pop` blocks until there's
            # something in the queue
            task = queue.pop
            task.call
          end
        end
      end
    end
  end

  def perform(&block)
    self.queue << block
  end

  def shutdown
    size.times do
      # this is going to make threads
      # break out of the infinite loop
      perform { throw :exit }
    end
  end
end
```

To read more about multi-threaded architecture in a Web Server you can refer to Puma Architecture. Unlike our simple implementation, there is an extra thread that reads from requests and sends connections to the thread pool. We are going to implement this pattern later when dealing with Ractors.

Fibers

Fibers are a lesser-known addition to Ruby 3 (or rather a scheduler interface for Fibers). You can think of Fibers as routines (e.g. goroutines). They are similar to threads, except instead of Operating System managing them, Ruby does. An advantage of that is less context switching, which means we have a smaller performance penalty when switching between Fibers than if OS switches between threads.

Where things get peculiar, is that with Fibers, you are responsible for implementing that scheduler. Thankfully there are several schedulers packaged into gems:

- evt
- libev_scheduler
- Async

So let's see how our server is going to look like with Fibers (*full code*):

```
  def start
    # Fibers are not going to work without a scheduler.
    # A scheduler is on for a current thread.
    Fiber.set_scheduler(Libev::Scheduler.new)

    Fiber.schedule do
      server = TCPServer.new(HOST, PORT)
      server.listen(SOCKET_READ_BACKLOG)
      loop do
        conn, _addr_info = server.accept
        # ideally we need to limit number of fibers
        # via a thread pool, as accepting infinite number
        # of request is a bad idea:
        # we can run out of memory or other resources,
        # there are diminishing returns to too many fibers,
        # without backpressure to however is sending the requests it's ha
        # to properly load balance and queue requests
        Fiber.schedule do
          request = RequestParser.call(conn)
          status, headers, body = app.call(request)
          HttpResponder.call(conn, status, headers, body)
        rescue => e
          puts e.message
        ensure
          conn&.close
        end
      end
    end
  end
```

References:

Don't Wait For Me! Scalable Concurrency for Ruby 3:

https://www.youtube.com/watch?v=Y29SSOS4UOc

Fiber API: https://docs.ruby-lang.org/en/master/Fiber.html

Ractors

Ractors are the shiniest addition to Ruby 3. Ractors are similar to threads, except multiple Ractors **can** execute in parallel, each Ractor has its own GIL.

Unfortunately, the word "toy" is not a metaphor, Ractors have a long way to go before they become usable. In my experiments, I get many segfaults on MacOS. No segfaults on Linux, but bugs still. For example, any time an exception was not rescued inside a Ractor the app has a chance of crashing. What may be even more important, is that 2 Ractors can't interact with the same object unless it is explicitly marked as shared. At the time of writing Ruby standard library has too many global objects, which makes sending an HTTP request not possible.

So let's see how our server is going to look like with Ractors (*full code*):

```
def start
  # the queue is going to be used to
  # fairly dispatch incoming requests,
  # we pass the queue into workers
  # and the first free worker gets
  # the yielded request
  queue = Ractor.new do
    loop do
      conn = Ractor.receive
      Ractor.yield(conn, move: true)
    end
  end
  # workers determine concurrency
  WORKERS_COUNT.times.map do
    # we need to pass the queue and the server so they are available
    # inside Ractor
    Ractor.new(queue, self) do |queue, server|
      loop do
        # this method blocks until the queue yields a connection
        conn = queue.take
        request = RequestParser.call(conn)
        status, headers, body = server.app.call(request)
        HttpResponder.call(conn, status, headers, body)
        # I have found that not rescuing errors does not only kill the
        # but causes random `allocator undefined for Ractor::MovedObjec
```

```
          # which crashes the whole program
        rescue => e
          puts e.message
        ensure
          conn&.close
        end
      end
    end
    # the listener is going to accept new connections
    # and pass them onto the queue,
    # we make it a separate Ractor, because `yield` in queue
    # is a blocking operation, we wouldn't be able to accept new connecti
    # until all previous were processed, and we can't use `send` to send
    # connections to workers because then we would send requests to worke
    # that might be busy
    listener = Ractor.new(queue) do |queue|
      socket = TCPServer.new(HOST, PORT)
      socket.listen(SOCKET_READ_BACKLOG)
      loop do
        conn, _addr_info = socket.accept
        queue.send(conn, move: true)
      end
    end
    Ractor.select(listener)
  end
```

References:
Ractor API: https://docs.ruby-lang.org/en/master/Ractor.html
Ractors specification:
https://github.com/ko1/ruby/blob/dc7f421bbb129a7288fade62afe581279f4d06cd/doc/ractor.md

Performance testing

This article wouldn't be complete if we didn't put our servers to action.

We are going to test each server against several types of tasks:

- CPU computation (code)
- File reading (code)
- HTTP requests (code)

In the table below I am showing a number of requests per second (higher is better). We are loading servers with 4 concurrent requests.

	CPU load	File reading	HTTP requests
Single-threaded	112.95	10932.28	2.84
Multi-threaded	99.91	7207.42	10.92
Fibers	113.45	9922.96	10.89
Ractors	389.97	18391.25	-*

** Ractors didn't run due to the limitation I described earlier, the standard library is not yet equipped to send HTTP requests. Perhaps we could try implementing reading from the server using TCP, but I stopped at that.*

Notes on the results

For the **CPU-heavy task**, we see that Ractors really outperform all other servers, by almost 4x (concurrency we used). Multi-threaded performed worse than single-threaded due to overhead of switching between threads, while fibers performed nearly identical to single-threaded (which is a good thing here, means very little overhead).
One very unpleasant surprise is that Ractors actually performed worse when I executed the following code, no idea how to explain it, lot's of work lies ahead of the Ruby team:

```
10.times do |i|
  1000.downto(1) do |j|
    Math.sqrt(j) * i / 0.2
  end
end
```

For the **file serving** task, I have to note, that I have a very fast NVMe SSD, so it seems that wait time was very low and results resemble the CPU task.

For the **HTTP task**, the slowest one expectedly was single-threaded. Both fibers and multi-threaded server performed nearly 4 times faster (again, with the concurrency of

4, we could go higher than that).

Conclusions

Given what a typical Rails server does, simply goes multi-threaded can yield great results, because usually the slowest parts are I/O bound.
If you have a CPU-heavy computation that you need to do, perhaps Ractors could be your answer, but given the drastic changes they are going to require, I expect it's going to take community many years before they can be applied in practice.
In that sense, Fibers might actually be the most useful addition, you can use them today in limited scope, for example to perform multiple web requests with a lower overhead than threads. I advise caution still, as fibers have their own thread-local variables.

Concurrency Fibers HTTP Ractors Ruby