Extracted from:

Ruby Performance Optimization

Why Ruby Is Slow, and How to Fix It

This PDF file contains pages extracted from *Ruby Performance Optimization*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2015 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Ruby Performance Optimization

Why Ruby Is Slow, and How to Fix It

Alexander Dymo

edited by Michael Swaine

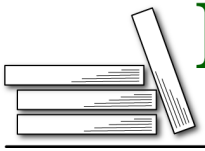
Ruby Performance Optimization

Why Ruby Is Slow, and How to Fix It

Alexander Dymo

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Copyright © 2015 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-069-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2015

Optimize Your Iterators

To a Ruby newcomer, Ruby iterators typically look like a convenient syntax for loops. In fact, iterators are such a good abstraction that even seasoned developers often forget that they really are nothing more than methods of `Array` and `Hash` classes with a block argument.

However, keeping this in mind is important for performance. We talked in *Modify Arrays and Hashes in Place, on page ?* about the importance of in-place operations on hashes and arrays. But that's not the end of the story.

Because a Ruby iterator is a function of an object (`Array`, `Range`, `Hash`, etc.), it has two characteristics that affect performance:

1. Ruby GC will not garbage collect the object you're iterating before the iterator is finished. This means that when you have a large list in memory, that whole list will stay in memory even if you no longer need the parts you've already traversed.
2. Iterators, being functions, can and will create temporary objects behind the scenes. This adds work for the garbage collector and hurts performance.

Compounding these performance hits, iterators (just like loops) are sensitive to the algorithmic complexity of the code. An operation that by itself is just a tad slow becomes a huge time sink when repeated hundreds of thousands of times.

So let's see when exactly iterators become slow and what can we do about that.

Free Objects from Collections During Iteration

Let's assume we have a list of objects, say one thousand elements of class `Thing`. We iterate over the list, do something useful, and discard the list. I've seen and written a lot of such code in production applications. For example, you read data from a file, calculate some stats, and return only the stats.

```
class Thing; end
list = Array.new(1000) { Thing.new }

list.each do |item|
  # do something with the item
end
list = nil
```

Obviously we can't deallocate list before each finishes. So it will stay in memory even if we no longer need access to previously traversed items. Let's prove that by counting the number of Thing instances before each iteration.

chp2/each_bang.rb

```
class Thing; end
list = Array.new(1000) { Thing.new }
puts ObjectSpace.each_object(Thing).count # 1000 objects

list.each do |item|
  GC.start
  puts ObjectSpace.each_object(Thing).count # same count as before
  # do something with the item
end

list = nil
GC.start
puts ObjectSpace.each_object(Thing).count # everything has been deallocated
```

```
$ ruby -I . each_bang.rb
1000
1000
«...»
1000
1000
0
```

As expected, only when we clear the list reference does the whole list get garbage collected. We can do better by using a while loop and removing elements from the list as we process them, like this:

chp2/each_bang.rb

```
class Thing; end
list = Array.new(1000) { Thing.new }   # allocate 1000 objects again
puts ObjectSpace.each_object(Thing).count

while list.count > 0
  GC.start    # this will garbage collect item from previous iteration
  puts ObjectSpace.each_object(Thing).count # watch the counter decreasing
  item = list.shift
end

GC.start    # this will garbage collect item from previous iteration
puts ObjectSpace.each_object(Thing).count # watch the counter decreasing
```

```
$ ruby -I . each_bang.rb
1000
999
«...»
2
1
```

See how the object counter decreases as we loop through the list? I'm again running GC before each iteration to show you that all previous elements are garbage and will be collected. In the real world you wouldn't want to force GC. Just let it do its job and your loop will neither take too much time nor run out of memory.

Don't worry about negative effects of list modification inside the loop. GC time savings will outweigh them if you process lots of objects. That happens both when your list is large and when you load linked data from these objects—for example, Rails associations.

Use the Each! Pattern

If we wrap our loop that removes items from an array during iteration into a Ruby iterator, we'll get what its creator, Alexander Goldstein, called "Each!". This is how the simplest each! iterator looks:

chp2/each_bang_pattern.rb

```
class Array
  def each!
    while count > 0
      yield(shift)
    end
  end
end

Array.new(10000).each! { |element| puts element.class }
```

This implementation is not 100% idiomatic Ruby because it doesn't return an Enumerator if there's no block passed. But it illustrates the concept well enough. Also note how it avoids creating Proc objects from anonymous blocks (there's no &block argument).

Avoid Iterators That Create Additional Objects

It turns out that some Ruby iterators (not all of them as we will see) internally create additional Ruby objects. Compare these two examples:

chp2/iterator_each1.rb

```
GC.disable
before = ObjectSpace.count_objects

Array.new(10000).each do |i|
  [0,1].each do |j|
  end
end

after = ObjectSpace.count_objects
```

```
puts "# of arrays: %d" % (after[:T_ARRAY] - before[:T_ARRAY])
puts "# of nodes: %d" % (after[:T_NODE] - before[:T_NODE])
```

```
$ ruby -I . iterator_each1.rb
# of arrays: 10001
# of nodes: 0
```

chp2/iterator_each2.rb

```
GC.disable
before = ObjectSpace.count_objects

Array.new(10000).each do |i|
  [0,1].each_with_index do |j, index|
  end
end

after = ObjectSpace.count_objects
puts "# of arrays: %d" % (after[:T_ARRAY] - before[:T_ARRAY])
puts "# of nodes: %d" % (after[:T_NODE] - before[:T_NODE])
```

```
$ ruby -I . iterator_each2.rb
# of arrays: 10001
# of nodes: 20000
```

As you'd expect, the code creates 10,000 temporary [0,1] arrays. But something fishy is going on with the number of T_NODE objects. Why would each_with_index create 20,000 extra objects?

The answer is in the Ruby source code. Here's the implementation of each:

```
VALUE
rb_ary_each(VALUE array)
{
    long i;
    volatile VALUE ary = array;
    RETURN_SIZED_ENUMERATOR(ary, 0, 0, ary_enum_length);
    for (i=0; i<RARRAY_LEN(ary); i++) {
        rb_yield(RARRAY_AREF(ary, i));
    }
    return ary;
}
```

Compare it to the implementation of and each_with_index.

```
enum_each_with_index(int argc, VALUE *argv, VALUE obj)
{
    NODE *memo;
    RETURN_SIZED_ENUMERATOR(obj, argc, argv, enum_size);
    memo = NEW_MEMO(0, 0, 0);
    rb_block_call(obj, id_each, argc, argv, each_with_index_i, (VALUE)memo);
    return obj;
}
```

```
static VALUE
each_with_index_i(RB_BLOCK_CALL_FUNC_ARGLIST(i, memo))
{
    long n = RNODE(memo)->u3.cnt++;
    return rb_yield_values(2, rb_enum_values_pack(argc, argv), INT2NUM(n));
}
```

Even if your C-fu is not that strong, you'll still see that each_with_index creates an additional NODE *memo variable. Because our each_with_index loop is nested in another loop, we get to create 10,000 additional nodes. Worse, the internal function each_with_index_i allocates one more node. Thus we end up with the 20,000 extra T_NODE objects that you see in our example output.

How does that affect performance? Imagine your nested loop is executed not 10,000 times, but 1 million times. You'll get 2 million objects created. And while they can be freed during the iteration, GC still gets way too much work to do. How's that for an iterator that you would otherwise easily mistake for a syntactic construct?

It would be nice to know which iterators are bad for performance and which are not, wouldn't it? I thought so, and so I calculated the number of additional T_NODE objects created per iterator. The table on page 12 summarizes the results for commonly used iterators.

Iterators that create 0 additional objects are safe to use in nested loops. But be careful with those that allocate two or even three additional objects: all?, each_with_index, inject, and others.

Looking at the table, we can also spot that iterators of the Array class, and in some cases the Hash class, behave differently. It turns out that Range and Hash use default iterator implementations from the Enumerable module, while Array reimplements most of them. That not only results in better algorithmical performance (that was the reason behind the reimplementation), but also in better memory consumption. This means that most of Array's iterators are safe to use, with the notable exceptions of each_with_index and inject.

Watch for Iterator-Unsafe Ruby Standard Library Functions

Iterators are where the algorithmic complexity of the functions you use matters, even in Ruby. One millisecond lost in a loop with one thousand iterations translates to a one-second slowdown. Let me show which commonly used Ruby functions are slow and how to replace them with faster analogs.

• Click HERE to purchase this book now. discuss

Iterator	Enum†	Array	Range	Iterator	Enum†	Array	Range
all?	3	3	3	fill	0	—	—
any?	2	2	2	find	2	2	2
collect	0	1	1	find_all	1	1	1
cycle	0	1	1	grep	2	2	2
delete_if	0	—	0	inject	2	2	2
detect	2	2	2	map	0	1	1
each	0	0	0	none?	2	2	2
each_index	0	—	—	one?	2	2	2
each_key	—	—	0	reduce	2	2	2
each_pair	—	—	0	reject	0	1	0
each_value	—	—	0	reverse	0	—	—
each_with_index	2	2	2	reverse_each	0	1	1
each_with_object	1	1	1	select	0	1	0

Table 1—Number of additional T_NODE objects created by an iterator

† Enum is Enumerable

Date#parse

Date parsing in Ruby has been traditionally slow, but this function is especially harmful for performance. Let's see how much time it uses in a loop with 100,000 iterations:

chp2/date_parsing1.rb

```
require 'date'
require 'benchmark'

date = "2014-05-23"
time = Benchmark.realtime do
  100000.times do
    Date.parse(date)
  end
end
puts "%.3f" % time
```

```
$ ruby date_parsing1.rb
0.833
```

Each Date#parse call takes a minuscule 0.02 ms. But in a moderately large loop, that translates into almost one second of execution time.

A better solution is to let the date parser know which date format to use, like this:

chp2/date_parsing2.rb
```
require 'date'
require 'benchmark'

date = "2014-05-23"
time = Benchmark.realtime do
  100000.times do
    Date.strptime(date, '%Y-%m-%d')
  end
end
puts "%.3f" % time
```

```
$ ruby date_parsing2.rb
0.182
```

That is already 4.6 times faster. But avoiding date string parsing altogether is even faster:

chp2/date_parsing3.rb
```
require 'date'
require 'benchmark'

date = "2014-05-23"
time = Benchmark.realtime do
  100000.times do
    Date.civil(date[0,4].to_i, date[5,2].to_i, date[8,2].to_i)
  end
end
puts "%.3f" % time
```

```
$ ruby date_parsing3.rb
0.108
```

While slightly uglier, that code is almost eight times faster than the original, and almost two times faster than the Date#strptime version.

Object#class, Object#is_a?, Object#kind_of?

These have considerable performance overhead when used in loops or frequently used functions like constructors or == comparison operators.

chp2/class_check1.rb
```
require 'benchmark'

obj = "sample string"
time = Benchmark.realtime do
  100000.times do
    obj.class == String
  end
end
puts time
```

```
$ ruby class_check1.rb
0.022767841
```

chp2/class_check2.rb

```
require 'benchmark'

obj = "sample string"
time = Benchmark.realtime do
  100000.times do
    obj.is_a?(String)
  end
end
puts time
```

```
$ ruby class_check2.rb
0.019568893
```

In a moderately large loop, again 100,000 iterations, such checks take 19–22 ms. That doesn't sound bad, except that, for example, a Rails application can call comparison operators more than 1 million times per request and spend longer than 200 ms doing type checks.

It's a good idea to move type checking away from iterators or frequently called functions and operators. If you can't, unfortunately there's not much you can do about that.

BigDecimal::==(String)

Code that gets data from databases uses big decimals a lot. That is especially true for Rails applications. Such code often creates a BigDecimal from a string that it reads from a database, and then compares it directly with strings.

The catch is that the natural way to do this comparison is unbelievably slow in Ruby version 1.9.3 and lower:

chp2/bigdecimal1.rb

```
require 'bigdecimal'
require 'benchmark'

x = BigDecimal("10.2")
time = Benchmark.realtime do
  100000.times do
    x == "10.2"
  end
end
puts time
```

```
$ rbenv shell 1.9.3-p551
$ ruby bigdecimal1.rb
0.773866128
```

```
$ rbenv shell 2.0.0-p598
$ ruby bigdecimal1.rb
0.025224029
$ rbenv shell 2.1.5
$ ruby bigdecimal1.rb
0.027570681
$ rbenv shell 2.2.0
$ ruby bigdecimal1.rb
0.02474011096637696
```

Older Rubys have unacceptably slow implementations of the BigDecimal::== function. This performance problem goes away with a Ruby 2.0 upgrade. But if you can't upgrade, use this smart trick. Convert a BigDecimal to a String before comparison:

chp2/bigdecimal2.rb

```
require 'bigdecimal'
require 'benchmark'

x = BigDecimal("10.2")
time = Benchmark.realtime do
  100000.times do
    x.to_s == "10.2"
  end
end
puts time
```

```
$ rbenv shell 1.9.3-p545
$ ruby bigdecimal2.rb
0.195041792
```

This hack is three to four times faster—not forty times faster, as in the Ruby 2.x implementation, but still an improvement.