Ruby master - Feature #2065
An ancestors iterator
09/09/2009 12:38 AM - bahuvrihi (Simon Chiang)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td>mame (Yusuke Endoh)</td>
</tr>
<tr>
<td>Target version:</td>
<td>2.0.0</td>
</tr>
</tbody>
</table>

Description
BEGIN
I have implemented DSLs that add features to a class/module that should be inherited like methods. In those cases I end up iterating ancestors to find the first time a feature has been added (the same way I imagine methods are determined).

The issue is that SomeClass.ancestors regenerates the ancestors array each time it is called. Therefore this is relatively slow:

SomeClass.ancestors.each do |ancestor|
  # ...
end

It would be nice if there were a method that iterates ancestors without generating the ancestors array:

SomeClass.each_ancestor do |ancestor|
  # ...
end

This could improve the performance of DSLs that want to support method-like inheritance.
END

History
#1 - 10/21/2009 08:46 AM - rogerdpack (Roger Pack)
BEGIN
You could write your own method:

class Object
  def ancestors_cached
    @ancestors ||= ancestors
  end
end

Would that be sufficient? (It's harder to implement that in core since it would need to be invalidated whenever a class hierarchy changes).
-r

#2 - 10/22/2009 09:31 AM - bahuvrihi (Simon Chiang)
BEGIN
Actually I think that gets to the heart of the matter. It's not sufficient to cache the ancestors because I want to iterate the most current class hierarchy. Doing so allows added features to really be method-like in their inheritance.

I figure somewhere in core the class hierarchy must already be iterated to generate the ancestors array. I'm envisioning a hook that lets you access that iteration and most likely break out of it at an early point in the ancestry.

As a side note, in my use case I offer caching like you suggest as an option to improve performance. However it's exactly as you point out... hard to invalidate at the right time.

#3 - 03/19/2010 01:28 AM - mame (Yusuke Endoh)
BEGIN

Hi,

It would be nice if there were a method that iterates ancestors without generating the ancestors array:
SomeClass.each_ancestor do |ancestor|
  # ...
end

This could improve the performance of DSLs that want to support method-like inheritance.

Show a benchmark. I bet it is not bottleneck in your application.

It is endless to provide each_* method corresponding to any methods that returns an array.

I'll close this ticket unless there is no objection.

As a side note, in my use case I offer caching like you suggest as an option to improve performance. However it's exactly as you point out... hard to invalidate at the right time.

Cannot Module#included be used?

class Class
  AncestorsCache = {}
  def ancestors_cached
    AncestorsCache[self] ||= ancestors
  end
end

class Module
  def included(x)
    Class::AncestorsCache.clear
  end
end

class C; end
p C.ancestors_cached #=> [C, Object, Kernel, BasicObject]

module M; end
class C; include M; end
p C.ancestors_cached #=> [C, M, Object, Kernel, BasicObject]

--
Yusuke Endoh mame@tsg.ne.jp
=end

#4 - 03/19/2010 05:37 AM - bahuvrihi (Simon Chiang)

- File cache_benchmark.rb added

=begin
Ok, I attached a benchmark that is designed to measure the time required for generating the ancestors array. It does so by comparing 'klass.ancestors.each' vs caching 'klass.ancestor' and iterating 'cache.each' (ie in the second case the ancestors array is only generated once).

Running the attached script I get results like the following:

% ruby cache_benchmark.rb
Benchmark without cache
user  system total   real
A. value(:one)  0.300000  0.000000  0.300000 ( 0.307086)
B. value(:two)  0.310000  0.000000  0.310000 ( 0.305512)
C. value(:three)  0.310000  0.000000  0.310000 ( 0.307264)

Benchmark with cache
user  system total   real
A. value(:one)  0.240000  0.000000  0.240000 ( 0.242152)
B. value(:two)  0.240000  0.000000  0.240000 ( 0.244592)
C. value(:three)  0.250000  0.000000  0.250000 ( 0.243842)
Array.new  0.050000  0.000000  0.050000 ( 0.058072)

This is for 100k iterations, so no it's not a bottleneck but likewise that's not why I make this request. I make the request because it would be significantly faster to not generate the ancestors array each time (~20% in this case, which illustrates what is probably the maximum increase for getting an each_ancestor iterator). Notice that the increase in performance corresponds neatly with the time it take to generate 100k arrays.

I imagine somewhere the in-memory ancestry is iterated to make the ancestors array, right? If so I think exposing that iterator would be helpful to DSLs that implement method-like inheritance.

08/07/2021
Also I agree that adding each_* methods for every method that return an array would be ridiculous but I'm only asking for a specific one with a specific purpose ;)

The module include method may work to the same end -- I need to explore further -- but I still make this request because it would be less technical to use an each_ancestor iterator, and ultimately less error-prone given all the things that ruby can do with including, extending, undefining and redefining constants. Thank you for the suggestion, and for looking at this.

=end

#5 - 03/19/2010 12:41 PM - mame (Yusuke Endoh)
=begin
Hi Simon,

2010/3/19 Simon Chiang redmine@ruby-lang.org:

This is for 100k iterations, so no it's not a bottleneck but likewise that's not why I make this request. I make the request because it would be significantly faster to not generate the ancestors array each time (~20% in this case, which illustrates what is probably the maximum increase for getting an each_ancestor iterator).

I cannot get your point. If you admit it is not bottleneck, I wonder why you hope it faster.
You should know adding a method is not free in terms of maintenance.

Ok, I attached a benchmark that is designed to measure the time required for generating the ancestors array.

Thanks, I could understand what you want to do.

If you want speed at any rate, how about manual search by using Class#superclass?
It is probably faster than each_ancestor because it does not yield a block.

module Dsl2
def self.extended(base)
  base.registry ||= {} #end
end
def inherited(base)
  base.registry ||= {} #end
end
attr_accessor :registry
def set(key, value)
  registry[key] = value #end
end
def value(key)
  klass = self
  while klass
    if klass.registry.has_key?(key)
      return klass.registry[key]
    end
    klass = klass.superclass
  end
end
end

Benchmark without cache

<table>
<thead>
<tr>
<th>user</th>
<th>system</th>
<th>total</th>
<th>real</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.340000</td>
<td>0.000000</td>
<td>0.340000 ( 0.34906)</td>
</tr>
<tr>
<td>B</td>
<td>0.340000</td>
<td>0.000000</td>
<td>0.340000 ( 0.34900)</td>
</tr>
<tr>
<td>C</td>
<td>0.360000</td>
<td>0.000000</td>
<td>0.360000 ( 0.357215)</td>
</tr>
</tbody>
</table>

Benchmark with cache

<table>
<thead>
<tr>
<th>user</th>
<th>system</th>
<th>total</th>
<th>real</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.240000</td>
<td>0.000000</td>
<td>0.240000 ( 0.235161)</td>
</tr>
<tr>
<td>B</td>
<td>0.230000</td>
<td>0.000000</td>
<td>0.230000 ( 0.236722)</td>
</tr>
<tr>
<td>C</td>
<td>0.240000</td>
<td>0.000000</td>
<td>0.240000 ( 0.233931)</td>
</tr>
<tr>
<td>Array.new</td>
<td>0.080000</td>
<td>0.000000</td>
<td>0.080000 ( 0.084003)</td>
</tr>
</tbody>
</table>
Benchmark with manual search

<table>
<thead>
<tr>
<th></th>
<th>user</th>
<th>system</th>
<th>total</th>
<th>real</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2.value(:one)</td>
<td>0.130000</td>
<td>0.000000</td>
<td>0.130000</td>
<td>0.122530</td>
</tr>
<tr>
<td>B2.value(:two)</td>
<td>0.120000</td>
<td>0.000000</td>
<td>0.120000</td>
<td>0.120361</td>
</tr>
<tr>
<td>C2.value(:three)</td>
<td>0.110000</td>
<td>0.000000</td>
<td>0.110000</td>
<td>0.119009</td>
</tr>
</tbody>
</table>

--

Yusuke ENDOH mame@tsg.ne.jp

Attachment: benchmark.rb
=begin

#6 - 03/20/2010 12:17 AM - bahuvrihi (Simon Chiang)
=begin

So to clarify... How much any piece of code of a bottleneck depends on how frequently the code gets run, and that is application-specific. In my own application, it IS enough of a bottleneck for me to make this request because I run the code very frequently. In other applications, it may not be a bottleneck.

For DSLs that want to have method-like inheritance, the lack of an ancestors iterator is a potential and unnecessary bottleneck. An each_ancestor iterator would, as per the benchmarks, significantly speed up a useful programming pattern that I think would be used more frequently if it were inherently faster. I hope that clarifies the purpose of this request.

Using superclass would work in the example I gave you but not in general because as I understand it superclass does not visit modules. Using ancestors allows for method-like 'inheritance' because you can visit both modules and superclasses. Using superclass limits you to the class hierarchy.

=end

#7 - 03/20/2010 12:45 AM - mame (Yusuke Endoh)
=begin

Hi,

2010/3/20 Simon Chiang redmine@ruby-lang.org:

So to clarify... How much any piece of code of a bottleneck depends on how frequently the code gets run, and that is application-specific. In my own application, it IS enough of a bottleneck for me to make this request because I run the code very frequently. In other applications, it may not be a bottleneck.

For DSLs that want to have method-like inheritance, the lack of an ancestors iterator is a potential and unnecessary bottleneck. An each_ancestor iterator would, as per the benchmarks, significantly speed up a useful programming pattern that I think would be used more frequently if it were inherently faster. I hope that clarifies the purpose of this request.

I doubt if there is few real application whose bottleneck is Module#ancestors.

Using superclass would work in the example I gave you but not in general because as I understand it superclass does not visit modules. Using ancestors allows for method-like 'inheritance' because you can visit both modules and superclasses. Using superclass limits you to the class hierarchy.

Indeed.
I'd like to make up for my stupid suggestion by giving a patch :-) I can commit this if matz approves.

diff --git a/class.c b/class.c
index fed2edf..51bc162 100644
--- a/class.c
+++ b/class.c
@@ -763,6 +763,42 @@ rb_mod_ancestors(VALUE mod)
 return ary;
 }

+/*
 * call-seq:
 *   mod.each_ancestor -> nil
 * *
 * Yields each module included in mod (including mod
 * itself).
 * *
 * module Mod
 * include Math

 08/07/2021
* include Comparable
* end
* *
* * Mod.each_ancestor {|c| p c } #=> Mod, Comparable, Math
* *
* Math.each_ancestor {|c| p c } #=> Math
* */ +VALUE +rb_mod_each_ancestor(VALUE mod) +{
* VALUE p; +
* RETURN_ENUMERATOR(mod, 0, 0); +
* for (p = mod; p; p = RCLASS_SUPER(p)) {
* if (FL_TEST(p, FL_SINGLETON))
* continue;
* if (BUILTIN_TYPE(p) == T_ICLASS) {
* rb_yield(RBASIC(p)->klass);
* }
* else {
* rb_yield(p);
* }
* }
* return Qnil; +} + #define VISI(x) ((x)&NOEX_MASK) #define VISI_CHECK(x, f) (VISI(x) == (f))

diff --git a/object.c b/object.c
index 0421824..6c8766e 100644
--- a/object.c
+++ b/object.c
@@ -2449,6 +2449,8 @@ rb_f_array(VALUE obj, VALUE arg)
 return rb_Array(arg);
 }
+extern VALUE rb_mod_each_ancestor(VALUE mod);
+
*/ +VALUE +rb_mod_each_ancestor(VALUE mod) +{
VALUE p; +
RETURN_ENUMERATOR(mod, 0, 0); +
for (p = mod; p; p = RCLASS_SUPER(p)) {
if (FL_TEST(p, FL_SINGLETON))
continue;
if (BUILTIN_TYPE(p) == T_ICLASS) {
rb_yield(RBASIC(p)->klass);
}
else {
rb_yield(p);
}
return Qnil; +} + #define VISI(x) ((x)&NOEX_MASK) #define VISI_CHECK(x, f) (VISI(x) == (f))
diff --git a/test/ruby/test_module.rb b/test/ruby/test_module.rb
index f905431..4330d69 100644
--- a/test/ruby/test_module.rb
+++ b/test/ruby/test_module.rb
@@ -216,6 +216,15 @@ class TestModule < Test::Unit::TestCase
 remove_rake_mixins(remove_json_mixins(remove_pp_mixins(String.ancestors))))
} end
+CLASS_EVAL = 2 @@class_eval = 'b'
--
Yusuke ENDOH mame@tsg.ne.jp
=end

#8 - 03/20/2010 01:40 AM - now (Nikolai Weibull)
=begin
On Fri, Mar 19, 2010 at 16:45, Yusuke ENDOH mame@tsg.ne.jp wrote:

---
Yusuke ENDOH mame@tsg.ne.jp
=end
I'd like to make up for my stupid suggestion by giving a patch :-)  
I can commit this if matz approves.

If I understood everything so far, we're talking about 0.05–0.07 seconds per 100 000 calls. In what piece of code is that a bottleneck?

#9 - 03/20/2010 01:42 AM - now (Nikolai Weibull)
=begin
On Fri, Mar 19, 2010 at 17:40, Nikolai Weibull now@bitwi.se wrote:

On Fri, Mar 19, 2010 at 16:45, Yusuke ENDOH mame@tsg.ne.jp wrote:

I'd like to make up for my stupid suggestion by giving a patch :-)  
I can commit this if matz approves.

If I understood everything so far, we're talking about 0.05–0.07 seconds per 100 000 calls. In what piece of code is that a bottleneck?

I also wonder how much having that extra method slows down method lookup overall.

=end

#10 - 03/20/2010 01:50 AM - mame (Yusuke Endoh)
=begin
2010/3/20 Nikolai Weibull now@bitwi.se:

On Fri, Mar 19, 2010 at 17:40, Nikolai Weibull now@bitwi.se wrote:

On Fri, Mar 19, 2010 at 16:45, Yusuke ENDOH mame@tsg.ne.jp wrote:

I'd like to make up for my stupid suggestion by giving a patch :-)  
I can commit this if matz approves.

If I understood everything so far, we're talking about 0.05–0.07 seconds per 100 000 calls. In what piece of code is that a bottleneck?

I also want to know.

I also wonder how much having that extra method slows down method lookup overall.

:-)

--
Yusuke ENDOH mame@tsg.ne.jp
=end

#11 - 03/20/2010 01:40 PM - bahuvrihi (Simon Chiang)
=begin
Well, what can I say? When you put it that way I'm sure this can all seem quite trivial in the grand scheme. Truth is most bottlenecks, including this one, are not so bad... just wait a second. Ruby will be fine with or without an each_ancestors iterator; it's my opinion it would be better with one.

But this suggestion it would slowdown method lookup overall -- is that an issue? I don't know. It is interesting because if having extra methods significantly slows down the lookup then I wonder how much faster ruby would be without the many generally unused methods there are in the language. I'm not trying to be provocative with this question, it is to me an interesting point.

Thanks for writing up a patch! I think it's an improvement but of course I'm not always right. :)

08/07/2021
Hello,

I'm not keen to import this feature.

I'm closing the ticket. Please reopen it if you find a piece of code in the wild whose bottleneck is solved by #each_ancestor.

--
Yusuke Endoh mame@tsg.ne.jp

Files

<table>
<thead>
<tr>
<th>Filename</th>
<th>Size</th>
<th>Date</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_benchmark.rb</td>
<td>1.79 KB</td>
<td>03/19/2010</td>
<td>bahuvrihi (Simon Chiang)</td>
</tr>
</tbody>
</table>