Ruby - Feature #10440

Optimize keyword and splat argument

10/28/2014 06:19 AM - ko1 (Koichi Sasada)

Status:	Closed
Priority:	Normal
Assignee:	ko1 (Koichi Sasada)
Target version:	2.2.0

Description

Abstract

Change data structure of call_info and rewrite all of method argument fitting code to optimize keyword arguments and a splat argument. My measurement shows that this optimization is x10 faster than current code of method dispatch with keyword argument.

Background

This feature focuses two issues about keyword arguments and a splat argument.

(1) Keyword arguments

Caller site of keyword arguments are introduced from Ruby 1.9.3, it is lik calling method with foo(k1: v1, k2: v2). This method invocation means that passing one Hash object as an argument of method foo, like foo({k1: v1, k2: v2}).

Callee site of keyword arguments are introduced from Ruby 2.0.0. We can write method definition like "def foo(k1: v1, k2: v2)". This is compiled to:

```
def foo(_kw) # _kw is implicit keyword
  # implicit plologue code
  k1 = _kw.key?(:k1) ? _kw[:k1] : v1
  k2 = _kw.key?(:k2) ? _kw[:k2] : v2
  # method body
  ...
end
```

foo(k1: v1, ...) makes one Hash object and defined method receives one Hash object. It is consistent between caller site and callee site.

However, there are several overhead.

(1-1) Making Hash object for each method invocation.
(1-2) Hash access code in implicit plologue code is overhead.

I had measured this overhead and result is http://www.atdot.net/~ko1/diary/201410.html#d11.

```
def foo0
end
def foo3 a, b, c
end
def foo6 a, b, c, d, e, f
end
def foo_kw6 k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil
end

ruby 2.2.0dev (2014-10-10 trunk 47867) [i386-mswin32_110]
       user     system      total        real
call foo0
   0.140000   0.000000   0.140000 (  0.134481)
call foo3
   0.141000   0.000000   0.141000 (  0.140427)
call foo6
   0.171000   0.000000   0.171000 (  0.180837)
```

```
call foo_kw6 without keywords
   0.593000   0.000000   0.593000 (  0.595162)
call foo_kw6 with 1 keyword
   1.778000   0.016000   1.794000 (  1.787873)
call foo_kw6 with 2 keyword, and so on.
   2.028000   0.000000   2.028000 (  2.034146)
   2.247000   0.000000   2.247000 (  2.255171)
   2.464000   0.000000   2.464000 (  2.470283)
   2.621000   0.000000   2.621000 (  2.639155)
   2.855000   0.000000   2.855000 (  2.863643)
```

You can see that "call foo6" is 5 times faster than "call foo_kw6 with 6 keyworsd".

The fact is that "calling keyword argument is slower than normal method dispatch.

Such small code is compile to the following VM codes.

```
def foo k1: 1, k2: 2
end

== disasm: <RubyVM::InstructionSequence:foo@../../trunk/test.rb>========
local table (size: 4, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, keyword: 2
                        @2] s0)
[ 4] k1         [ 3] k2         [ 2] ?
0000 getlocal_OP__WC__0 2                                             (   1)
0002 dup
0003 putobject          :k1
0005 opt_send_simple    <callinfo!mid:key?, argc:1, ARGS_SKIP>
0007 branchunless       18
0009 dup
0010 putobject          :k1
0012 opt_send_simple    <callinfo!mid:delete, argc:1, ARGS_SKIP>
0014 setlocal_OP__WC__0 4
0016 jump               21
0018 putobject_OP_INT2FIX_O_1_C_
0019 setlocal_OP__WC__0 4
0021 dup
0022 putobject          :k2
0024 opt_send_simple    <callinfo!mid:key?, argc:1, ARGS_SKIP>
0026 branchunless       37
0028 dup
0029 putobject          :k2
0031 opt_send_simple    <callinfo!mid:delete, argc:1, ARGS_SKIP>
0033 setlocal_OP__WC__0 3
0035 jump               41
0037 putobject          2
0039 setlocal_OP__WC__0 3
0041 pop
0042 trace              8
0044 putnil
0045 trace              16                                            (   2)
0047 leave                                                            (   1)
```

(2) A Splat argument and a rest parameter

Splat argument is N-length array object and it is handled as N-th normal arguments.

```
ary = [1, 2, 3]
foo(*ary)
foo(1, 2, 3) # These two method invocation is completely same.
```

Also a method can be accept any number of arguments by a rest parameter.

```
def foo(*rest)
  p rest
end
```

```
foo(1, 2) #=> [1, 2]
foo(1, 2, 3) #=> [1, 2, 3]
```

Combination of this splat argument and rest parameter, we should use very long array.

```
def foo(*rest)
  rest.size
end

foo((1..1_000_000).to_a) #=> should be 1000000
```

However, current implementation try to put all elements of a splat argument onto the VM stack, and it causes Stack overflow error.

```
test.rb:5:in `<main>': stack level too deep (SystemStackError)
```

And also delegation methods, which passes a splat argument and receives a rest argument, can be run faster without splatting all elements onto the VM stack.

Proposal: change call_info and rewrite argument fitting code

Basic idea is to passing caller arguments without any modification with a structure (meta) data.

(in other name, "Let it go" patch # sorry, this patch doesn't increase frozen objects.)

The patch is here: https://github.com/ko1/ruby/compare/kwopt

For keyword arguments

```
# example code
def foo(k1: default_v1, k2: default_v2)
  # method body
end

foo(k1: v1, k2: v2) # line 6
```

On line 6, only push values (v1 and v2) and pass keys (k1 and k2) info via a call_info strucutre.

In `send' instruction (line 2), fill local variables (k1, k2) with passed keyword values (v1, v2) with keys info in call_info.

Especially, default values (default_v1, default_v2) are immediate values such as nil, Fixnum and so on, we record such immediate values in compile time and set default values in send instruction. This technique reduce checking overhead in prologue code.

This case, disassembled code is here:

```
# ------------------------------------------------------------------------
# target program:
# ------------------------------------------------------------------------
# example code
def foo(k1: 1, k2: 2)
  # method body
end

foo(k1: 100, k2: 200) # line 6
# ------------------------------------------------------------------------
# disasm result:
# ------------------------------------------------------------------------
== disasm: <RubyVM::InstructionSequence:<main>@../../gitruby/test.rb>===
0000 trace            1                                               (   2)
0002 putspecialobject 1
0004 putspecialobject 2
0006 putobject        :foo
0008 putiseq          foo
0010 opt_send_simple  <callinfo!mid:core#define_method, argc:3, ARGS_SKIP>
0012 pop
0013 trace            1                                               (   6)
```

```
0015 putself
0016 putobject        100
0018 putobject        200
0020 opt_send_simple  <callinfo!mid:foo, argc:2, kw:2, FCALL|ARGS_SKIP>
0022 leave
== disasm: <RubyVM::InstructionSequence:foo@../../gitruby/test.rb>======
local table (size: 4, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, keyword: 2@2] s0)
[ 4] k1         [ 3] k2         [ 2] ?
0000 trace            8                                               (   2)
0002 putnil
0003 trace            16                                              (   4)
0005 leave
# ----------------------------------------------------------------------
```

Splat argument and rest argument

Instead of pushing all elements of a splat argument, we pass argument with flag (meta-data).

Evaluation

Benchmark with same program on different platform.

```
current trunk: ruby 2.2.0dev (2014-10-27 trunk 48154) [x86_64-linux]

       user     system      total        real
   0.070000   0.000000   0.070000 (  0.063836)
   0.070000   0.000000   0.070000 (  0.067525)
   0.070000   0.000000   0.070000 (  0.074835)
   0.270000   0.000000   0.270000 (  0.271872)
   1.170000   0.000000   1.170000 (  1.166828)
   1.320000   0.000000   1.320000 (  1.322710)
   1.480000   0.000000   1.480000 (  1.484837)
   1.680000   0.000000   1.680000 (  1.675304)
   1.780000   0.000000   1.780000 (  1.785633)
   1.970000   0.000000   1.970000 (  1.966972)

modified: ruby 2.2.0dev (2014-10-27 trunk 48158) [x86_64-linux]

       user     system      total        real
   0.080000   0.000000   0.080000 (  0.074382)
   0.090000   0.000000   0.090000 (  0.095778)
   0.080000   0.000000   0.080000 (  0.078085)
   0.110000   0.000000   0.110000 (  0.114086)
   0.110000   0.000000   0.110000 (  0.111416)
   0.120000   0.000000   0.120000 (  0.118595)
   0.130000   0.000000   0.130000 (  0.129644)
   0.140000   0.000000   0.140000 (  0.136531)
   0.160000   0.000000   0.160000 (  0.157686)
   0.150000   0.000000   0.150000 (  0.154985)
```

The performance of keyword arguments are dramatically improved.

And now, we can pass a big splat argument with a rest argument.

```
def foo(*rest)
  rest.size
end

p foo(*(1..1_000_000).to_a) #=> 1_000_000
```

Current evaluation of benchmark set is here: http://www.atdot.net/sp/view/gxr4en/readonly

The number is ratio compare with current trunk. Higher is fast (lower is slow than current implementation). This result shows that this patch introduced some overhead, especially yield() syntax. This is because I unify method invocation code and block invocation code, and eliminate fast pass for simple block invocation. I will add this fast pass and the results will be recovered.

Related issues:	
Related to Ruby - Bug #11663: Segfault when using multiple keywords if the fi...	**Closed**
Related to Ruby - Bug #12002: **param notation seems to be creating a new has...	**Rejected**

Associated revisions

Revision fbebd502f9f374d1eef31c63c10c7d8adcd63280 - 11/02/2014 06:02 PM - ko1 (Koichi Sasada)

- rewrite method/block parameter fitting logic to optimize
  keyword arguments/parameters and a splat argument.
  [Feature #10440] (Details are described in this ticket)
  Most of complex part is moved to vm_args.c.
  Now, ISeq#to_a does not catch up new instruction format.
- vm_core.h: change iseq data structures.
  - introduce rb_call_info_kw_arg_t to represent keyword arguments.
  - add rb_call_info_t::kw_arg.
  - rename rb_iseq_t::arg_post_len to rb_iseq_t::arg_post_num.
  - rename rb_iseq_t::arg_keywords to arg_keyword_num.
  - rename rb_iseq_t::arg_keyword to rb_iseq_t::arg_keyword_bits.
    to represent keyword bitmap parameter index.
    This bitmap parameter shows that which keyword parameters are given
    or not given (0 for given).
    It is refered by `checkkeyword' instruction described bellow.
  - rename rb_iseq_t::arg_keyword_check to rb_iseq_t::arg_keyword_rest
    to represent keyword rest parameter index.
  - add rb_iseq_t::arg_keyword_default_values to represent default
    keyword values.
  - rename VM_CALL_ARGS_SKIP_SETUP to VM_CALL_ARGS_SIMPLE
    to represent
    (ci->flag & (SPLAT|BLOCKARG)) &&
    ci->blockiseq == NULL &&
    ci->kw_arg == NULL.
- vm_insnhelper.c, vm_args.c: rewrite with refactoring.
  - rewrite splat argument code.
  - rewrite keyword arguments/parameters code.
  - merge method and block parameter fitting code into one code base.
- vm.c, vm_eval.c: catch up these changes.
- compile.c (new_callinfo): callinfo requires kw_arg parameter.
- compile.c (compile_array_): check the last argument Hash object or
  not. If Hash object and all keys are Symbol literals, they are
  compiled to keyword arguments.
- insns.def (checkkeyword): add new instruction.
  This instruction check the availability of corresponding keyword.
  For example, a method "def foo k1: 'v1'; end" is cimpiled to the
  following instructions.
  0000 checkkeyword   2, 0 # check k1 is given.
  0003 branchif   9   # if given, jump to address #9
  0005 putstring   "v1"
  0007 setlocal_OP__WC__0 3   # k1 = 'v1'
  0009 trace   8
  0011 putnil
  0012 trace   16
  0014 leave
- insns.def (opt_send_simple): removed and add new instruction
  "opt_send_without_block".
- parse.y (new_args_tail_gen): reorder variables.
  Before this patch, a method "def foo(k1: 1, kr1:, k2: 2, **krest, &b)"
  has parameter variables "k1, kr1, k2, &b, internal_id, krest",
  but this patch reorders to "kr1, k1, k2, internal_id, krest, &b".
  (locate a block variable at last)
- parse.y (vtable_pop): added.
  This function remove latest `n' variables from vtable.
- iseq.c: catch up iseq data changes.
- proc.c: ditto.
- class.c (keyword_error): export as rb_keyword_error().
- common.mk: depend vm_args.c for vm.o.
- hash.c (rb_hash_has_key): export.
- internal.h: ditto.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@48239 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision fbebd502 - 11/02/2014 06:02 PM - ko1 (Koichi Sasada)

- rewrite method/block parameter fitting logic to optimize
  keyword arguments/parameters and a splat argument.
  [Feature #10440] (Details are described in this ticket)
  Most of complex part is moved to vm_args.c.
  Now, ISeq#to_a does not catch up new instruction format.
- vm_core.h: change iseq data structures.
  - introduce rb_call_info_kw_arg_t to represent keyword arguments.
  - add rb_call_info_t::kw_arg.
  - rename rb_iseq_t::arg_post_len to rb_iseq_t::arg_post_num.
  - rename rb_iseq_t::arg_keywords to arg_keyword_num.
  - rename rb_iseq_t::arg_keyword to rb_iseq_t::arg_keyword_bits.
    to represent keyword bitmap parameter index.
    This bitmap parameter shows that which keyword parameters are given
    or not given (0 for given).
    It is refered by `checkkeyword' instruction described bellow.
  - rename rb_iseq_t::arg_keyword_check to rb_iseq_t::arg_keyword_rest
    to represent keyword rest parameter index.
  - add rb_iseq_t::arg_keyword_default_values to represent default
    keyword values.
  - rename VM_CALL_ARGS_SKIP_SETUP to VM_CALL_ARGS_SIMPLE
    to represent
    (ci->flag & (SPLAT|BLOCKARG)) &&
    ci->blockiseq == NULL &&
    ci->kw_arg == NULL.
- vm_insnhelper.c, vm_args.c: rewrite with refactoring.
  - rewrite splat argument code.
  - rewrite keyword arguments/parameters code.
  - merge method and block parameter fitting code into one code base.
- vm.c, vm_eval.c: catch up these changes.
- compile.c (new_callinfo): callinfo requires kw_arg parameter.
- compile.c (compile_array_): check the last argument Hash object or
  not. If Hash object and all keys are Symbol literals, they are
  compiled to keyword arguments.
- insns.def (checkkeyword): add new instruction.
  This instruction check the availability of corresponding keyword.
  For example, a method "def foo k1: 'v1'; end" is cimpiled to the
  following instructions.
  0000 checkkeyword   2, 0 # check k1 is given.
  0003 branchif   9   # if given, jump to address #9
  0005 putstring   "v1"
  0007 setlocal_OP__WC__0 3   # k1 = 'v1'
  0009 trace   8
  0011 putnil
  0012 trace   16
  0014 leave
- insns.def (opt_send_simple): removed and add new instruction
  "opt_send_without_block".
- parse.y (new_args_tail_gen): reorder variables.
  Before this patch, a method "def foo(k1: 1, kr1:, k2: 2, **krest, &b)"
  has parameter variables "k1, kr1, k2, &b, internal_id, krest",
  but this patch reorders to "kr1, k1, k2, internal_id, krest, &b".
  (locate a block variable at last)
- parse.y (vtable_pop): added.
  This function remove latest `n' variables from vtable.
- iseq.c: catch up iseq data changes.
- proc.c: ditto.
- class.c (keyword_error): export as rb_keyword_error().
- common.mk: depend vm_args.c for vm.o.
- hash.c (rb_hash_has_key): export.
- internal.h: ditto.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@48239 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

History

#1 - 10/28/2014 09:39 AM - ko1 (Koichi Sasada)

I support fastpath for yield.

I updated benchmark (show in below) includes yield().

```
# trunk
                                                              user     system      total        real
foo0                                                      0.300000   0.000000   0.300000 (  0.305596)
foo3 1, 2, 3                                              0.320000   0.000000   0.320000 (  0.319955)
```

```
foo6 1, 2, 3, 4, 5, 6                                                0.350000   0.000000   0.350000 (  0.347484)
foo_kw6                                                              1.250000   0.000000   1.250000 (  1.252436)
foo_kw6 k1: 1                                                        4.820000   0.000000   4.820000 (  4.818321)
foo_kw6 k1: 1, k2: 2                                                 5.390000   0.000000   5.390000 (  5.395304)
foo_kw6 k1: 1, k2: 2, k3: 3                                          6.080000   0.000000   6.080000 (  6.076292)
foo_kw6 k1: 1, k2: 2, k3: 3, k4: 4                                   6.700000   0.000000   6.700000 (  6.701519)
foo_kw6 k1: 1, k2: 2, k3: 3, k4: 4, k5: 5                            7.280000   0.000000   7.280000 (  7.279225)
foo_kw6 k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6                     7.960000   0.000000   7.960000 (  7.971349)
iter0{}                                                              0.430000   0.000000   0.430000 (  0.432206)
iter1{}                                                              0.440000   0.000000   0.440000 (  0.438057)
iter1{|a|}                                                           0.430000   0.000000   0.430000 (  0.437687)
iter3{}                                                              0.450000   0.000000   0.450000 (  0.450404)
iter3{|a|}                                                           0.450000   0.000000   0.450000 (  0.443590)
iter3{|a, b, c|}                                                     0.440000   0.000000   0.440000 (  0.445780)
iter6{}                                                              0.470000   0.000000   0.470000 (  0.464586)
iter6{|a|}                                                           0.460000   0.000000   0.460000 (  0.464179)
iter6{|a, b, c, d, e, f, g|}                                         0.470000   0.000000   0.470000 (  0.464736)
iter0{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}        1.300000   0.000000   1.300000 (  1.305616)
iter_kw1{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     5.320000   0.000000   5.320000 (  5.317812)
iter_kw2{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     5.980000   0.000000   5.980000 (  5.986901)
iter_kw3{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     6.680000   0.000000   6.680000 (  6.678429)
iter_kw4{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     7.340000   0.000000   7.340000 (  7.346109)
iter_kw5{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     7.890000   0.000000   7.890000 (  7.885493)
iter_kw6{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     8.440000   0.000000   8.440000 (  8.444766)
```

```
# modified
```

```
foo0                                                                 0.300000   0.000000   0.300000 (  0.304503)
foo3 1, 2, 3                                                         0.320000   0.000000   0.320000 (  0.319673)
foo6 1, 2, 3, 4, 5, 6                                                0.340000   0.000000   0.340000 (  0.334695)
foo_kw6                                                              0.480000   0.000000   0.480000 (  0.479434)
foo_kw6 k1: 1                                                        0.560000   0.000000   0.560000 (  0.558796)
foo_kw6 k1: 1, k2: 2                                                 0.570000   0.000000   0.570000 (  0.578939)
foo_kw6 k1: 1, k2: 2, k3: 3                                          0.610000   0.000000   0.610000 (  0.607001)
foo_kw6 k1: 1, k2: 2, k3: 3, k4: 4                                   0.610000   0.000000   0.610000 (  0.604850)
foo_kw6 k1: 1, k2: 2, k3: 3, k4: 4, k5: 5                            0.610000   0.000000   0.610000 (  0.616563)
foo_kw6 k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6                     0.630000   0.000000   0.630000 (  0.622955)
iter0{}                                                              0.410000   0.000000   0.410000 (  0.414572)
iter1{}                                                              0.430000   0.000000   0.430000 (  0.424715)
iter1{|a|}                                                           0.430000   0.000000   0.430000 (  0.430664)
iter3{}                                                              0.430000   0.000000   0.430000 (  0.435802)
iter3{|a|}                                                           0.440000   0.000000   0.440000 (  0.434144)
iter3{|a, b, c|}                                                     0.430000   0.000000   0.430000 (  0.436183)
iter6{}                                                              0.460000   0.000000   0.460000 (  0.453823)
iter6{|a|}                                                           0.450000   0.000000   0.450000 (  0.457398)
iter6{|a, b, c, d, e, f, g|}                                         0.450000   0.000000   0.450000 (  0.451839)
iter0{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}        0.590000   0.000000   0.590000 (  0.583469)
iter_kw1{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     0.670000   0.000000   0.670000 (  0.668229)
iter_kw2{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     0.680000   0.000000   0.680000 (  0.682113)
iter_kw3{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     0.700000   0.000000   0.700000 (  0.698496)
iter_kw4{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     0.710000   0.000000   0.710000 (  0.712838)
iter_kw5{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     0.720000   0.000000   0.720000 (  0.720042)
iter_kw6{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}     0.720000   0.000000   0.720000 (  0.719584)
```

Seems good.

Updated benchmark code:

```
require 'benchmark'

def foo0
end
def foo3 a, b, c
end
def foo6 a, b, c, d, e, f
end
def foo_kw6 k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil
end

def iter0
  yield
end

def iter1
  yield 1
```

```
end

def iter3
  yield 1, 2, 3
end

def iter6
  yield 1, 2, 3, 4, 5, 6
end

def iter_kw1
  yield k1: 1
end

def iter_kw2
  yield k1:1, k2: 2
end

def iter_kw3
  yield k1:1, k2: 2, k3: 3
end

def iter_kw4
  yield k1:1, k2: 2, k3: 3, k4: 4
end

def iter_kw5
  yield k1:1, k2: 2, k3: 3, k4: 4, k5: 5
end

def iter_kw6
  yield k1:1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6
end

test_methods = %Q{
  foo0
  foo3 1, 2, 3
  foo6 1, 2, 3, 4, 5, 6
  foo_kw6
  foo_kw6 k1: 1
  foo_kw6 k1: 1, k2: 2
  foo_kw6 k1: 1, k2: 2, k3: 3
  foo_kw6 k1: 1, k2: 2, k3: 3, k4: 4
  foo_kw6 k1: 1, k2: 2, k3: 3, k4: 4, k5: 5
  foo_kw6 k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6
  iter0{}
  iter1{}
  iter1{|a|}
  iter3{}
  iter3{|a|}
  iter3{|a, b, c|}
  iter6{}
  iter6{|a|}
  iter6{|a, b, c, d, e, f, g|}
  iter0{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}
  iter_kw1{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}
  iter_kw2{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}
  iter_kw3{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}
  iter_kw4{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}
  iter_kw5{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}
  iter_kw6{|k1: nil, k2: nil, k3: nil, k4: nil, k5: nil, k6: nil|}
}

N = 5_000_000

max_line = test_methods.each_line.max_by{|line| line.strip.size}
max_size = max_line.strip.size

Benchmark.bm(max_size){|x|

  str = test_methods.each_line.map{|line| line.strip!
    next if line.empty?
    %Q{
      x.report(#{line.dump}){
        N.times{#{line}}
```

```
      }
    }
  }.join("\n")
  eval str
}
```

#2 - 10/28/2014 11:31 PM - normalperson (Eric Wong)

Cool.
My only concern is making call_info and iseq structs bigger.

I think most of the iseq->arg_keyword_* fields can be moved to a
separate allocation (like catch table) because they are not common
and space may be saved that way.
We may do that after merging this optimization.

call_info is harder to shrink (but more common than iseq, so
size changes have more effect...)

#3 - 10/29/2014 01:04 AM - matz (Yukihiro Matsumoto)

Koichi, you haven't described incompatibility. Does that mean no behavior change?
If so, go ahead and make it fast. Then tune it for memory consumption, as Eric pointed out.

Matz.

#4 - 11/02/2014 05:26 PM - ko1 (Koichi Sasada)

Eric Wong wrote:

> My only concern is making call_info and iseq structs bigger.
>
> I think most of the iseq->arg_keyword_* fields can be moved to a
> separate allocation (like catch table) because they are not common
> and space may be saved that way.
> We may do that after merging this optimization.

I understand that arg_keyword* are very large. I will change them by using flags.

(I will introduce iseq_arg_flags to indicate "has keywords", "has rest arguments" and so on)

> call_info is harder to shrink (but more common than iseq, so
> size changes have more effect...)

I believe we can use similar technique.

#5 - 11/02/2014 05:27 PM - ko1 (Koichi Sasada)

Yukihiro Matsumoto wrote:

> Koichi, you haven't described incompatibility. Does that mean no behavior change?
> If so, go ahead and make it fast. Then tune it for memory consumption, as Eric pointed out.

Now, I can observe no compatibility issues.

#6 - 11/02/2014 06:03 PM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

- % Done changed from 0 to 100

Applied in changeset r48239.

- rewrite method/block parameter fitting logic to optimize
  keyword arguments/parameters and a splat argument.
  [Feature #10440] (Details are described in this ticket)
  Most of complex part is moved to vm_args.c.
  Now, ISeq#to_a does not catch up new instruction format.
- vm_core.h: change iseq data structures.
  - introduce rb_call_info_kw_arg_t to represent keyword arguments.
  - add rb_call_info_t::kw_arg.

    - rename rb_iseq_t::arg_post_len to rb_iseq_t::arg_post_num.
    - rename rb_iseq_t::arg_keywords to arg_keyword_num.
    - rename rb_iseq_t::arg_keyword to rb_iseq_t::arg_keyword_bits.
      to represent keyword bitmap parameter index.
      This bitmap parameter shows that which keyword parameters are given
      or not given (0 for given).
      It is refered by `checkkeyword' instruction described bellow.
    - rename rb_iseq_t::arg_keyword_check to rb_iseq_t::arg_keyword_rest
      to represent keyword rest parameter index.
    - add rb_iseq_t::arg_keyword_default_values to represent default
      keyword values.
    - rename VM_CALL_ARGS_SKIP_SETUP to VM_CALL_ARGS_SIMPLE
      to represent
      (ci->flag & (SPLAT|BLOCKARG)) &&
      ci->blockiseq == NULL &&
      ci->kw_arg == NULL.
- vm_insnhelper.c, vm_args.c: rewrite with refactoring.
    - rewrite splat argument code.
    - rewrite keyword arguments/parameters code.
    - merge method and block parameter fitting code into one code base.
- vm.c, vm_eval.c: catch up these changes.
- compile.c (new_callinfo): callinfo requires kw_arg parameter.
- compile.c (compile_array_): check the last argument Hash object or
  not. If Hash object and all keys are Symbol literals, they are
  compiled to keyword arguments.
- insns.def (checkkeyword): add new instruction.
  This instruction check the availability of corresponding keyword.
  For example, a method "def foo k1: 'v1'; end" is cimpiled to the
  following instructions.
  0000 checkkeyword   2, 0 # check k1 is given.
  0003 branchif   9   # if given, jump to address #9
  0005 putstring   "v1"
  0007 setlocal_OP__WC__0 3   # k1 = 'v1'
  0009 trace   8
  0011 putnil
  0012 trace   16
  0014 leave
- insns.def (opt_send_simple): removed and add new instruction
  "opt_send_without_block".
- parse.y (new_args_tail_gen): reorder variables.
  Before this patch, a method "def foo(k1: 1, kr1:, k2: 2, **krest, &b)"
  has parameter variables "k1, kr1, k2, &b, internal_id, krest",
  but this patch reorders to "kr1, k1, k2, internal_id, krest, &b".
  (locate a block variable at last)
- parse.y (vtable_pop): added.
  This function remove latest `n' variables from vtable.
- iseq.c: catch up iseq data changes.
- proc.c: ditto.
- class.c (keyword_error): export as rb_keyword_error().
- common.mk: depend vm_args.c for vm.o.
- hash.c (rb_hash_has_key): export.
- internal.h: ditto.

#7 - 11/03/2014 04:09 AM - ko1 (Koichi Sasada)

micro benchmark result:
https://docs.google.com/spreadsheets/d/1Mm3Y5RE0fgmNxVMA0Ba0DWH44qbvB_n_5vg9Gh4Rgzg/edit?usp=sharing

In some cases, it is x10 faster.

#8 - 11/06/2015 01:48 AM - nobu (Nobuyoshi Nakada)

- Related to Bug #11663: Segfault when using multiple keywords if the first keyword is invalid added

#9 - 01/18/2016 09:09 PM - wanabe (_ wanabe)

*- Related to Bug #12002: **param notation seems to be creating a new hash in ruby 2.2.0 added*