Ruby master - Feature #13434

better method definition in C API

04/14/2017 01:07 AM - normalperson (Eric Wong)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Current ways to define and parse arguments in the Ruby C API are clumsy,
slow, and impede potential optimizations.

The current C API for defining (rb_define_{singleton_}, method),
and parsing (rb_scan_args, rb_get_kwargs) is orthogonal but inefficient.

rb_get_kwargs creates garbage which pure Ruby kwarg methods do not.
[Feature #11339] was an ugly workaround to use Ruby wrapper methods
for IO#*nonblock methods to avoid garbage from rb_get_kwargs.

Furthermore, it should be possible to annotate args for C functions as
"read-only, use-once" or similar. In other words, it should be possible to
implement my idea from [ruby-core:80626] where method lookup can be done
out-of-order in some cases, and allow optimizations such as replacing
"putstring" insns with garbage-free "putobject" insns for constants strings
without introducing backwards incompatibility for Rubyists.

We can also get rid of the limited basic op redefinition checks and
implement more generic versions of opt_aref_with / opt_aset_with
for more functions that can take frozen string args.

The "read-only, use-once" annotation can even make it safe for
a dynamic strings to be immediately recycled to reduce garbage.

So we could annotate "puts" and IO#write in a way that causes the VM to
immediately recycle its argument if it's a dynamically-generated string:

```
puts "#{dynamic} #{string(:here)}"
```

I am not good at API design; so I'm not sure what it should look like.

Perhaps sendmsg_nonblock may be implemented like:

```
struct rb_method_info {
    /* to be filled in by rb_def_method ... */
};

static VALUE
sendmsg_nonblock(struct rb_method_info *info, int argc, VALUE *argv, VALUE self)
{
    VALUE mesg, flags, dest_sockaddr, control, exception;

    rb_get_args(info, argc, argv,
        &mesg, &flags, &dest_sockaddr, &control, &exception);

    ...
}

/*
 * ALLCAPS variable names mean read-only (like "constants" in Ruby)
 * "1" prefix means use only once, eligible for immediately recycle
 * if dynamic string
 */
```

```
rb_def_method(rb_cBasickSocket, sendmsg_nonblock,
              "sendmsg_nonblock(1MESG "
                "1FLAGS = 0), "
                "1DEST_SOCKADDR = nil), "
                "*1CONTROL, exception: true)", -1);

/* rb_hash_aset can be done as:
 * where 0KEY (not "1" prefix) means it is constant and persistent,
 * and "val" (all lower case, no prefix) means it is a normal
 * variable which can persistent after the function returns
 */
rb_def_method(rb_Hash, rb_hash_aset, "[0KEY]=val", 2);
```

Thoughts?

The existing C API must continue to work, so 3rd-party extensions can
migrate to the new API slowly.

History

#1 - 04/14/2017 05:29 AM - naruse (Yui NARUSE)

- Description updated

#2 - 04/18/2017 09:14 AM - naruse (Yui NARUSE)

I agree with the concept.

From r55102, rb_scan_args is statically resolved by C compilers on some environment, rb_get_kwargs is still inefficient.
To allow C compilers statically resolve them, Ruby method in C should be defined in more machine friendly way.

I thought the new API should use C struct (write rb_method_info by hand?).
Anyway we should list up the requirement of the new API, for example

- readonly/unused flag for arguments.
- whether the caller requires return single value, multiple value (like Perl's wantarray), or not.

#3 - 04/18/2017 08:13 PM - normalperson (Eric Wong)

naruse@airemix.jp wrote:

> Issue #13434 has been updated by naruse (Yui NARUSE).
>
> I agree with the concept.
>
> From r55102, rb_scan_args is statically resolved by C compilers on some environment, rb_get_kwargs is still inefficient.
> To allow C compilers statically resolve them, Ruby method in C should be defined in more machine friendly way.

Cool, I forgot about that rb_scan_args optimization. Maybe we
can use similar optimization for defining methods, too,
to speed up VM startup.

> I thought the new API should use C struct (write rb_method_info by hand?).

I think the new API should resemble pure Ruby method definition
for ease-of-learning. But, we will need a new way to mark
unused/readonly...

> Anyway we should list up the requirement of the new API, for example
>
> - readonly/unused flag for arguments.
> - whether the caller requires return single value, multiple value (like Perl's wantarray), or not.

Agreed on all of these.

For wantarray, I think we can support returning klass==0
(hidden) array from C methods. We then teach the VM to treat
klass==0 Array return values as a special case: the VM will set
klass=rb_cArray lazily if capturing the array is required.

In other words, this example:

```
static VALUE cfunc(VALUE self)
{
    VALUE ret = rb_ary_tmp_new(3);
    rb_ary_push(ret, INT2FIX(1));
    rb_ary_push(ret, INT2FIX(2));
    rb_ary_push(ret, INT2FIX(3));

    return ret; /* klass == 0 */
}
```

Example 1, return value is discarded immediately:

```
a, b, c = cfunc
```

In the above case, the VM calls:

```
rb_ary_clear(ret);
rb_gc_force_recycle(ret)
```

after assigning a, b, c since the temporary array is no longer used.

Example 2, return value is preserved:

```
a, b, c = ary = cfunc
```

In the above case, the VM calls:

```
rb_obj_reveal(ret, rb_cArray)
```

since it is assigned to ary.

#4 - 04/18/2017 08:32 PM - normalperson (Eric Wong)

Also, this is a bit far off; but a potential future optimization is even being able to use readonly markers in C methods to infer readonly args use in pure Ruby methods.

For example, Rack::Request is:

```
def get_header(name)
  @env[name]
end
```

where @env is a Hash. Since rb_hash_aref can be marked with the key being read-only, the end goal is to make even pure Ruby method calls like:

```
get_header("HTTP_FOO")
```

avoid allocation, just as current calls to @env["HTTP_FOO"] get optimized with opt_aref_with. And we should do this without introducing any incompatibility.

Again, I am 100% against making frozen_string_literal the default because it introduces backwards incompatibility.

#5 - 06/16/2017 09:01 AM - nobu (Nobuyoshi Nakada)

I don't like "mini-language" which needs a parser.

#6 - 06/20/2017 05:51 AM - normalperson (Eric Wong)

nobu@ruby-lang.org wrote:

> I don't like "mini-language" which needs a parser.

OK, what about an API similar to pthread_attr_set*?

Hash#[]=

```
rb_method_attr_set_required(&attr, 0);
```

```
rb_method_attr_set_const(&attr, 0);
rb_method_attr_set_persist(&attr, 0);
rb_method_attr_set_required(&attr, 1);
rb_def_method(rb_Hash, rb_hash_aset, &attr);
```

Maybe the above is too verbose:

```
rb_method_attr_setfl(&attr, 0, RB_CONST|RB_PERSIST|RB_REQUIRED);
rb_method_attr_setfl(&attr, 1, RB_REQUIRED);
rb_def_method(rb_Hash, "[]=", rb_hash_aset, &attr);
```

For "exception: (true|false)"

```
rb_method_attr_setkw_const(&attr, "exception", Qtrue);
```

#7 - 06/30/2017 12:08 AM - normalperson (Eric Wong)

Eric Wong normalperson@yhbt.net wrote:

> rb_method_attr_setfl(&attr, 0, RB_CONST|RB_PERSIST|RB_REQUIRED);
> rb_method_attr_setfl(&attr, 1, RB_REQUIRED);
> rb_def_method(rb_Hash, "[]=", rb_hash_aset, &attr);
>
> For "exception: (true|false)"
>
> rb_method_attr_setkw_const(&attr, "exception", Qtrue);

I'm investigating implementing something along these lines;
not my area of expertise but I think I can learn something.
Will report back in a few days (hope I do not sidetracked into
something else :x).

#8 - 06/30/2017 06:05 AM - ko1 (Koichi Sasada)

As I wrote before, I against this idea. My idea is to write definitions in Ruby with special form.

Comparison:

- Write info with C
  - Easy to learn (people should know only C syntax)
  - Easy to implement
- Write info with Ruby w/ special firm
  - Easy to write
  - Easy to read
  - Easy to learn (people should know Ruby syntax and some restrictions) (I agree it is possible that "some restrictions" will confuse people.)
  - We can compile it and know all of information before running Ruby (*1)

I'll explain more about (*1).

Now we can't know all of methods defined in C level before running Ruby interpreter. So MRI needs to process all of method definitions.

```
  rb_define_method(...); // allocate table entry and insert it
  rb_define_method(...); // allocate table entry and insert it
  rb_define_method(...); // allocate table entry and insert it...
  ...
```

If we know the all sets of methods, we pre-allocate table.

```
  table_allocate(3)
  rb_define_method(...); // insert it
  rb_define_method(...); // insert it
  rb_define_method(...); // insert it...
```

Moreover, we can prepare tables.

```
struct method_entries table = [] = {
  "foo", foo_func, ...,
  "bar", bar_func, ...,
  "baz", baz_func, ...,
}
```

```
//
  rb_define_methods_with_table(cString, table); // convert C array to MRI method table
```

Furthermore, we can defer converting until first method call.

```
struct method_entries table = [] = {
  "foo", foo_func, ...,
  "bar", bar_func, ...,
  "baz", baz_func, ...,
}

Init_String{
  ...
  rb_define_methods_with_table(rb_cString, table); // register table to rb_cString
  ...
}

//

  p "xyzzy".upcase # the first time we convert registerred table to MRI method table
```

I'm suspect that most of classes are not used (think about many of Exception class) so that this kind of optimization will improve speed (reduce boot time) and memory efficiency.

#9 - 06/30/2017 08:09 AM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> As I wrote before, I against this idea. My idea is to write
> definitions in Ruby with special form.

Sorry, I wasn't sure what you wanted the last time this came up.
I guess it was around https://bugs.ruby-lang.org/issues/11339
Particularly:
[ruby-core:69990] https://public-inbox.org/ruby-core/55A72930.40305@atdot.net/

I remember not liking Ricsin syntax, but maybe a Ruby API can
be better than Ricsin...

> Comparison:
>
> - Write info with C
>   - Easy to learn (people should know only C syntax)
>   - Easy to implement
> - Write info with Ruby w/ special firm
>   - Easy to write
>   - Easy to read
>   - Easy to learn (people should know Ruby syntax and some restrictions) (I agree it is possible that "some restrictions" will confuse people.)
>   - We can compile it and know all of information before running Ruby (*1)

OK, I agree.

excellent explanation.

> Moreover, we can prepare tables.

OK, I like this part.

> Furthermore, we can defer converting until first method call.

OK that sounds excellent! :)

> `I'm suspect that most of classes are not used (think about many of Exception class) so that this kind of optimization will improve speed (reduce boot time) and memory efficiency.`

Yes, that would be great. However, we will take into account
fork and CoW savings.

Should I try to implement your table idea? Or did you already
start? You are more familiar with this, but maybe I can try...

In addition to improved kwarg handling for C methods, my other
goal is to be able to mark read-only/use-once/const/etc. args to
avoid unnecessary allocations at runtime. This will be more
flexible than current optimizations (opt_aref_with, opt_aset_with, etc).

#10 - 07/13/2017 04:29 AM - ko1 (Koichi Sasada)

I wrote the following sentences in hastily so sorry if it has English grammar problems.

normalperson (Eric Wong) wrote:

> Sorry, I wasn't sure what you wanted the last time this came up.
> I guess it was around https://bugs.ruby-lang.org/issues/11339
> Particularly:
> [ruby-core:69990] https://public-inbox.org/ruby-core/55A72930.40305@atdot.net/
>
> I remember not liking Ricsin syntax, but maybe a Ruby API can
> be better than Ricsin...

Yes. and I understand.

> Yes, that would be great. However, we will take into account
> fork and CoW savings.

Maybe this table will be read-only table so that no CoW issue.

> Should I try to implement your table idea? Or did you already
> start? You are more familiar with this, but maybe I can try...

Sure. But we need to consider the strategy about this issue.
Note that lazy table loading is common with the following approach S1 and S2.

(Strategy-1) Define a table in C and use it.

This is very straight forward approach. Define table;

```
struct method_define_table_entry {
  const char *method_name;
  ID method_id;            /* we can collaborate with id.c for built-in methods. */
                           /* if we can't (extension libraries, method_name is used */
  func_type func();
  int arity;
  method_type type; /* maybe union of method type bits. visibility, and more */
}
```

There are no jump from current implementation. But not so much fruits.

(Strategy-2: S2) Use ISeq binaries also for C methods.

To use keyword (and rest) arguments optimization for ISeq in C methods, we need to make ISeq wrapper. To achieve this goal, we can wrap C methods with ISeq. In otherwords, C methods are implemented as normal ISeq type methods and invoke them with new insn (or opt_call_c_function). Compiled ISeq can be dumped with binary translation and MRI can load it.

We can aggregates all of binary and method table only knows the index of iseq.

```
struct method_define_table_entry {
  long iseq_entry_index; /* because iseq knows it name. */
                         /* however, if we want to encourage pre-defined ID,
                            then we can add ID on it */
};
```

Of course, we don't need to define struct, but only array is enough.

We have further advantage with this approach.

- we can note method parameters like normal Ruby methods.
- we can use exception handling in Ruby. rb_protect() and so on is difficult to use (and slow).
- (spec change) we can put such definition locations (like lib/built-in/string.rb) in backtrace. <- we need to discuss it is preferable or not.
- we can cleanup most of C-func related codes because all methods will be unified to iseq. For example, we can use same trace point probes.

BTW I had introduced VM_FRAME_FLAG_CFRAME last year because to achieve this approach.

Issues on this approach:

- ISeq call is slower than C-call because several problems. I believe we can overcome this issue.
- Current ISeq binary dumper is not space efficient (dumped iseq is huge because I don't use any compression techniques). Of course we can improve it (but we need to care about loading time).

  In addition to improved kwarg handling for C methods, my other
  goal is to be able to mark read-only/use-once/const/etc. args to
  avoid unnecessary allocations at runtime. This will be more
  flexible than current optimizations (opt_aref_with, opt_aset_with, etc).

Sure. That is also my goal in long time (build a knowledge database of C-implemented behavior). With chatting with nobu, we consider several notation.

```
class String

  # @pure func <- comment notation
  def to_s
    C.attr :pure # <- method notation it doesn't affect run-time behavior.
    self
  end


  # rep: ... <- comment notation
  def gsub(pat, rep = nil)
    C.attr(rep: %i(const dont_escape)) # <- method notation
    C.call :str_gsub(pat, rep)
    # or (*1)
    if rep
      C.call :str_gsub_with_repl(pat, repl)
    else
      C.call :str_gsub_with_block(pat)
    end
  end
end
```

*1: For performance, we may need to introduce special form to branch by arguments. But it should be only performance critical case, such as Array#[].

#11 - 07/13/2017 07:41 AM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> I wrote the following sentences in hastily so sorry if it has
> English grammar problems.

no problem.

> normalperson (Eric Wong) wrote:
>
> > Yes, that would be great. However, we will take into account
> > fork and CoW savings.
>
> Maybe this table will be read-only table so that no CoW issue.

OK.

> > Should I try to implement your table idea? Or did you already
> > start? You are more familiar with this, but maybe I can try...
>
> Sure. But we need to consider the strategy about this issue.
> Note that lazy table loading is common with the following approach S1 and S2.

Sidenote:

Unfortunately, I don't think I can start a major new feature
at this time. My situation is different than 2 weeks ago.

I will of course continue to support/improve existing
proposals like Feature #13618, but I
will probably focus smaller changes instead of big ones.

> (Strategy-1) Define a table in C and use it.
>
> There are no jump from current implementation. But not so much fruits.

Right.

> (Strategy-2: S2) Use ISeq binaries also for C methods.

OK, I like this :)

all good, no comments.

- Current ISeq binary dumper is not space efficient (dumped iseq is huge because I don't use any compression techniques). Of course we can improve it (but we need to care about loading time).

For loading time, smaller binary dump means less I/O and maybe
less allocations. This is more noticeable for people on
rotational disks.

I also think iseq can be compressed by sharing repeated
method/constant lookups:

```
File.unlink(f) if File.exist?(f)
```

or:

```
str.gsub!(pat1, rep1)
str.gsub!(pat2, rep2)
str.gsub!(pat3, rep3)
```

Ideally, that is only one method lookup and cache for #gsub,
not 3 lookups + 3 entries.

Of course, we will need to detect/annotate those methods and
operands have no side effects which can expire caches.

> In addition to improved kwarg handling for C methods, my other
> goal is to be able to mark read-only/use-once/const/etc. args to
> avoid unnecessary allocations at runtime. This will be more
> flexible than current optimizations (opt_aref_with, opt_aset_with, etc).

> Sure. That is also my goal in long time (build a knowledge database of C-implemented behavior). With chatting with nobu, we consider several notation.
>
> ```
> class String
>
>   # @pure func <- comment notation
>   def to_s
>     C.attr :pure # <- method notation it doesn't affect run-time behavior.
>     self
>   end
> ```

I prefer method notation since it is less likely to conflict
(RubyVM::C, not 'C' :) Comments might conflict with documentation.

```
  # rep: ... <- comment notation
  def gsub(pat, rep = nil)
    C.attr(rep: %i(const dont_escape)) # <- method notation
    C.call :str_gsub(pat, rep)
```

```
    # or (*1)
    if rep
      C.call :str_gsub_with_repl(pat, repl)
    else
      C.call :str_gsub_with_block(pat)
    end
  end
end
```

Anyways, I like this S2 since it still looks like Ruby :)

> *1: For performance, we may need to introduce special form to branch by arguments. But it should be only performance critical case, such as Array#[].

OK.