Ruby master - Feature #17004

Provide a way for methods to omit their return value

07/01/2020 09:24 AM - shyouhei (Shyouhei Urabe)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

In ruby, it often is the case for a method's return value to not be used by its caller. Even when a method returns something meaningful, its caller is free to ignore it.

Why not provide a way for a method to know if its return value is needed or not? That adds a room for methods to be optimized, by for instance skipping creation of complex return values.

The following pull request implements RubyVM.return_value_is_used? method, which does that:
https://github.com/ruby/ruby/pull/3271

History

#1 - 07/01/2020 10:40 AM - sawa (Tsuyoshi Sawada)

- Description updated

#2 - 07/01/2020 03:41 PM - headius (Charles Nutter)

In rb_whether_the_return_value_is_used_p I believe whether_the is redundant with p. is also seems unnecessary here, so perhaps rb_return_value_used_p and return_value_used? are good enough?

There is a possible concern if this is used for values that have visible side effects (changing some internal state, setting $~ or $_, possibly raising an exception, ...), so it should be used very carefully. Return value may not be the only effect that's important.

#3 - 07/01/2020 03:42 PM - jeremyevans0 (Jeremy Evans)

I can see definite performance advantages to this in my libraries (specifically Sequel). Knowing that the return value is not used can save expensive database queries to determine what the return value should be.

I think it may be better to make this a private Kernel method, similar to block_given?. RubyVM should only be used for code specific to CRuby, and this should be something that all Ruby implementations should support. I recommend the name return_value_used? for consistency with block_given? (we don't use block_is_given?).

#4 - 07/01/2020 04:02 PM - mame (Yusuke Endoh)

My first impression is that this is a very bad tool to change method behavior depending upon its caller context. However, my second consideration is that such a bad thing is already possible by some ways like Kernel#caller. So I'm neutral. If it is introduced. I'd see an explicit caution such as "DO NOT ABUSE THIS!" in its document.

#5 - 07/01/2020 04:10 PM - Eregon (Benoit Daloze)

What Jeremy said, so in short RubyVM is not a good place for this because it's CRuby-specific (ExperimentalFeatures or Kernel would be OK IMHO).

Do you have measurements on real applications, not just micro-benchmarks?

The masgn case ('1.times {|i| x, y = self, i }') could be done transparently by the VM without exposing any predicate.
Same for core methods like String#slice!.
I'm not sure if it's a good idea to expose this to Ruby (and C ext) users, it seems very low level.
At least, I think we should take advantage of this in language/core before exposing to users.

#6 - 07/01/2020 04:14 PM - Eregon (Benoit Daloze)

As an example, I don't think using such a manual optimization in OptCarrot (default mode) would be appropriate (it would feel like a hack).

#7 - 07/01/2020 04:42 PM - enebo (Thomas Enebo)

Reposted from github issue:
"Having only thought about this for about half an hour I am concerned with this feature. Will this lead to people writing APIs where the users of that API need to worry about whether they are assigning the method or not? More or less assignment could end up changing the semantics of the method. Whether a method is assigned or not it should still do the same thing. This feature will allow API designers to break that."

#8 - 07/01/2020 04:52 PM - enebo (Thomas Enebo)

I will add that although I can see the same potential problem with the C api I am less concerned it will lead to the same problems I outlined in Core MRI code.

#9 - 07/01/2020 05:02 PM - headius (Charles Nutter)

> I'd see an explicit caution such as "DO NOT ABUSE THIS!" in its document.

It will definitely be abused.

I do not think this should be exposed to user code in any way. If you want a method to either return a result or not depending on how it is called, you should define a different method.

We have been discussing this on the JRuby Matrix chat and there are many concerns:

- Users of this API will have to ensure there are no side effects of **any kind** that would be omitted.

That includes exceptions that might be raised, in-memory state changes elsewhere in the system, database changes, IO state changes, potentially even native memory changes if there are C API calls involved. I would argue that the **only** safe places this can be used are to wrap a simple allocation, and even that has side effects (memory effects, out of memory errors, unexpected downstream or C API calls).

- If this is exposed as a user-accessible feature, then every place a call is made at the end of a method will want to use the feature to optionally return nil, as below:

```
def foo
  do_some_work()
  if RubyVM.return_value_used?
    expensive_call_that_can_eliminate_return()
  else
    expensive_call_that_can_eliminate_return()
    nil
  end
end
```

This will lead to a cascading effect as other methods also try to special-case how they make calls in the context of assignment or returns, forcing other methods to also make changes to how they do calls, ad infinatum.

- It will be abused, and used in error, probably more often than it is used correctly. And everyone will try to use it thinking they will use it safely, and they'll probably get it wrong.

One example case was to eliminate some calculate_expensive_report when the report won't be needed. But the report generation itself will have side effects, like caching data in memory, altering some in-memory data model, advancing an IO position or database cursor.

- It is not Ruby.

Ruby is an expression language. This makes it possible for people to opt out of being an expression, changing visible behavior in the process.

...

I would also point out that an inlining JIT that can see through the methods you might call would already be able to do this. Adding this method essentially short-circuits the "safe" way of doing the optimization and hopes that the user will not make a mistake and eliminate some side effect that was intended.

#10 - 07/01/2020 05:04 PM - tenderlovemaking (Aaron Patterson)

The more I think about this, the more it concerns me. If there is some library code like this:

```
def do_something_and_return_report
  something = do_something
  if RubyVM.return_value_used?
    create_report(something)
  else
    nil
  end
end
```

And I am a user of the library. I want to debug my code, so maybe I do this:

puts do_something_and_return_report

If I remove the puts, then the behavior of do_something_and_return_report would be totally different. Even worse, I cannot use do_something_and_return_report in IRB because the behavior of do_something_and_return_report in IRB would be totally different than the behavior

in a script. It would be very confusing to explain "the behavior of do_something_and_return_report is different because IRB used the return value, but your script did not".

This seems like a cool trick, and something that we should use internally to MRI. But I don't think it should be exposed to users. It seems like a situation where the VM and JIT should work harder to optimize code, not library authors or library consumers.

#11 - 07/01/2020 11:42 PM - marcandre (Marc-Andre Lafortune)

Are there examples (for example from known gems) where this would actually be useful?

#12 - 07/02/2020 12:18 AM - duerst (Martin Dürst)

Additional questions:

- Are there any other languages that have such a feature?
- Where there are performance implications, couldn't that be solved by an additional parameter to the methods in question? Or by a better design of interfaces (e.g. different methods for cases where an expensive return value isn't needed)? (jeremyevans0 (Jeremy Evans))

#13 - 07/02/2020 01:01 AM - jeremyevans0 (Jeremy Evans)

duerst (Martin Dürst) wrote in #note-12:

> - Where there are performance implications, couldn't that be solved by an additional parameter to the methods in question? Or by a better design of interfaces (e.g. different methods for cases where an expensive return value isn't needed)? (jeremyevans0 (Jeremy Evans))

Yes, it could definitely be solved by additional method arguments (or keyword arguments). However, that can complicate implementation or may not be possible depending on the method's API (consider a method that already accepts arbitrary arguments and arbitrary keywords).

One specific use case I see for this is Sequel::Dataset#insert (http://sequel.jeremyevans.net/rdoc/classes/Sequel/Dataset.html#method-i-insert). The return value of this method is generally the primary key value of the last inserted row. On some databases, getting that value is expensive, potentially doubling the execution time of the method when using a remote database. If the return value is not needed, the INSERT query could still be performed, but it would not be necessary to issue another query to SELECT the return value.

Due to Sequel::Dataset#insert's flexible API, it would be hard to support this as a method argument. I could add a different method to support it, but then I need to add a separate internal method (more indirection, lower performance), or significant duplication. Additionally, having fewer, more flexible methods can result in an API that is easier to remember and use, compared to an API that has many more methods with less flexible behavior for each (with the tradeoff that the internals become significantly more complex).

One potential advantage of the VM_FRAME_FLAG_DISCARDED flag that may not yet have been anticipated is not a performance advantage, but a usability advantage. Consider the following code:

```
def foo(**kw)
  kw.merge(FORCE_VALUES)
  bar(**kw)
end
```

This code has a bug I've seen new Ruby programmers make. The bug is that Hash#merge returns a new hash, it doesn't modify the existing hash. This is almost certainly a bug, because there is no reason to call Hash#merge without using the return value. The programmer almost certainly wanted the behavior of Hash#merge!. Basically, Hash#merge is a pure function. We could add a way to mark methods as pure functions (e.g. Module#pure_function), and if the method is called with VM_FRAME_FLAG_DISCARDED, Ruby could warn or raise.

While I am in favor of this, I can certainly understand the potential for abuse. However, Ruby has never been a language that avoided features simply because it is possible to abuse them. That being said, I think it would make sense to make this strictly internal initially, and not expose it to C extensions and pure ruby code unless the internal usage demonstrates its usefulness.

#14 - 07/02/2020 05:31 AM - shyouhei (Shyouhei Urabe)

Re: other languages with similar concepts.

- Perl has wantarray. In spite of its name, the intrinsic can be used to distinguish if a return value is needed or not (can tell you if the needed number of return values is zero, one, or many more).
- If we consider warnings on unused return values be a kind of it...
  - C++ since C++17 has [[nodiscard]] function attribute.
  - GCC provides something similar to C as well.
  - In Rust that attribute is called #[must_use].
  - Swift has such warnings default on, and must explicitly annotate a function with @discardableResult if you allow users to ignore them.

#15 - 07/02/2020 04:13 PM - soulcutter (Bradley Schaefer)

jeremyevans0 (Jeremy Evans) wrote in #note-13:

> ```
> def foo(**kw)
>   kw.merge(FORCE_VALUES)
>   bar(**kw)
> ```

```
  end
```

> This code has a bug I've seen new Ruby programmers make. The bug is that Hash#merge returns a new hash, it doesn't modify the existing hash. This is almost certainly a bug, because there is no reason to call Hash#merge without using the return value. The programmer almost certainly wanted the behavior of Hash#merge!. Basically, Hash#merge is a pure function. We could add a way to mark methods as pure functions (e.g. Module#pure_function), and if the method is called with VM_FRAME_FLAG_DISCARDED, Ruby could warn or raise.

What scares me about this is the idea of using interactive debuggers (or even plan-old puts debugging) changing the behavior of the code. You would have to be an expert (primed to think about this behavior) to recognize that simply observing a method means you can't make any assumptions about what happens when you're not observing it. Also, how would you test this behavior?

#16 - 07/02/2020 04:21 PM - jeremyevans0 (Jeremy Evans)

soulcutter (Bradley Schaefer) wrote in #note-15:

> Also, how would you test this behavior?

```
# return value not discarded case
some_method.must_equal :expected_return_value
check_for.must_equal :some_side_effect

# return value discarded case
some_method
check_for.must_equal :some_side_effect
```

#17 - 07/02/2020 04:45 PM - soulcutter (Bradley Schaefer)

jeremyevans0 (Jeremy Evans) wrote in #note-16:

> soulcutter (Bradley Schaefer) wrote in #note-15:
>
> > Also, how would you test this behavior?
>
> ```
> # return value not discarded case
> some_method.must_equal :expected_return_value
> check_for.must_equal :some_side_effect
>
> # return value discarded case
> some_method
> check_for.must_equal :some_side_effect
> ```

That's fair. There's still a warning flag in my head that there's some subtle case where it is trickier to test, but I might be struggling to wrap my head around all the implications. Given that I can't come up with that example at the moment, I retract that problem-statement.

#18 - 07/02/2020 05:26 PM - Eregon (Benoit Daloze)

Agreed as well on the point of "if I observe it with p/puts/IRB I don't want the method call to behave differently.
Debug printing should avoid having side effects, and this makes a significant way to break that.
Sounds also very confusing when benchmarking some method and leaving an unused variable vs removing it and seeing a large difference/maybe the benchmark doesn't do at all what it intended.

#19 - 07/02/2020 08:15 PM - marcandre (Marc-Andre Lafortune)

jeremyevans0 (Jeremy Evans) wrote in #note-13:

> Due to Sequel::Dataset#insert's flexible API, it would be hard to support this as a method argument.

Seems to be that a better API for this is using a block parameter.

```
# need the ID:
insert(...) do |last_inserted_id|
  # ...
end

# don't need the ID:
insert(...)
```

Another possibility is to implement a lazy return value, something like:

```
class LazyID
```

```
  def to_i
    # get the ID
  end
end

def insert(...)
  # ...
  LazyID.new(self)
end
```

I remain unconvinced the proposal is a good idea.

#20 - 07/02/2020 09:11 PM - jeremyevans0 (Jeremy Evans)

marcandre (Marc-Andre Lafortune) wrote in #note-19:

> jeremyevans0 (Jeremy Evans) wrote in #note-13:
>
> > Due to Sequel::Dataset#insert's flexible API, it would be hard to support this as a method argument.
>
> Seems to be that a better API for this is using a block parameter.
>
> ```
> # need the ID:
> insert(...) do |last_inserted_id|
>   # ...
> end
>
> # don't need the ID:
> insert(...)
> ```

Sequel::Dataset#insert already accepts a block, which can be used to iterate over rows returned from the insert statement (when using INSERT RETURNING).

> Another possibility is to implement a lazy return value, something like:
>
> ```
> class LazyID
>   def to_i
>     # get the ID
>   end
> end
>
> def insert(...)
>   # ...
>   LazyID.new(self)
> end
> ```

In general, this approach requires additional allocation in the case where you are using the return value, decreasing performance. In this particular case, it's not possible, because you would probably lose access to the database connection used to insert the record before calling LazyID#to_i, and that database connection is needed to get the value.

#21 - 07/20/2020 05:44 AM - matz (Yukihiro Matsumoto)

To disclose this kind of information to the Ruby level is just too much, I feel. As a compromise, how about experimenting some internal API (via Primitive) for CRuby?

Matz.