I think this is very important, otherwise ... can be used only very rarely.

For instance, method_missing typically want to access the method name like:

```ruby
def method_missing(name, ...)
  if name.to_s.end_with?('?')
    self[name]
  else
    fallback(name, ...)
  end
end
```

See the original feature: https://bugs.ruby-lang.org/issues/16253#note-19.

I think most people expect def method_missing(name, ...) to work.

Related issues:
Related to Ruby master - Feature #16253: Shorthand "forward everything" syntax
Closed

Also I believe ... could be a good way to do delegation in all versions for lexical cases (the majority):

```ruby
# Could be some constant in a gem
ARGS = RUBY_VERSION < "2.7" ? "*args, &block" : "...

class_eval <<RUBY
  def method_missing(name, #{ARGS})
    if name.to_s.end_with?('?')
      self[name]
    else
      fallback(name, #{ARGS})
    end
  end
RUBY
```

And while Redmine doesn't syntax highlight <<RUBY, at least GitHub and RubyMine do.

#4 - 11/28/2019 04:56 AM - matz (Yukihiro Matsumoto)
- Status changed from Open to Rejected

I know trailing ... can be very useful from C experience. But the primary purpose of Ruby ... is method delegation. We are not going to extend the role of ... in the language (at least for now).

Matz.
matz (Yukihiro Matsumoto) wrote:

I know trailing ... can be very useful from C experience. But the primary purpose of Ruby ... is method delegation. We are not going to extend the role of ... in the language (at least for now).

That is surprising. It makes ... unusable in many delegation use cases which need to extract the first(s) arguments. The above method_missing is also delegation, isn't it?

What's your solution for that case? Using ruby2_keywords def method_missing(name, *args) and then having to change it to def method_missing(name, *args, **kwargs) once ruby2_keywords is removed?

Defining method_missing is not something rare in Ruby. It seems a shame ... can't be used there, even though it would a very good place to use ... (delegation in method_missing is almost always lexical).

BTW, R has ... and it supports leading arguments. And of course the construct that ... replaces, that is *args, &block as "all arguments" supports leading arguments too.

matz (Yukihiro Matsumoto) Could you reply to this?

Particularly:

But the primary purpose of Ruby ... is method delegation.

Indeed, and I believe we also want to extract leading arguments in many delegation use cases. Not supporting leading arguments is a obvious limitation and I would think unexpected for many rubyists. As an example, it makes ... unusable for method_missing, which is a place where delegation often happens.

I think the decision was too quick, maybe because I set target version 2.7. It won't be in 2.7 since that's released, but let's consider it for future releases.

In the DevelopersMeeting20191017Japan log there was "Future work: lead argument handling is postponed", so clearly there was the intention of adding it later.

We have found out that #method_missing (and #send) needed leading arguments otherwise we cannot use argument forwarding for them. I changed my mind. Accepted.

Matz.