

Ruby master - Feature #11588

Implement structured warnings

10/13/2015 03:15 PM - djberg96 (Daniel Berger)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Ruby's current warning system is lacking. Warnings are controlled by the `-W` flag on the command line, and are generated via the `Kernel#warn` method within code. There are a host of problems with this approach to warnings.

First, warnings aren't currently testable. With `Test::Unit`, for example, I can ensure that specific errors are raised in certain conditions via the `assert_raise` method. There is no analogue for warnings. It would be nice if there were so I could test them.

Second, there is no backtrace information provided with warnings. If I discover a warning I have to wade through the source and figure out where it was generated, because a `Kernel#warn` call does not provide a line number or method name that I can refer back to, unless it happened to be generated by `rb_warn()`. For large code bases that can be problematic.

Third, and most significantly, with warning flags it's all or nothing. I cannot enable or disable specific kinds of warnings. Perl, for example, implements warning control through pragmas. So, for example, I can specify "no warnings uninitialized" in a Perl program and warnings about uninitialized variables go away. With Ruby it's off, on, or even-more-on (`-W0`, `-W1` or `-W2`).

What I would like to see are structured warnings. By "structured warnings" I mean a system analogous to the `Error` class, except that a warning would only emit text to `STDERR`, not cause the interpreter to exit. In our hypothetical `Warning` class you still have backtrace information available. And, like `Exceptions`, there would be a standard hierarchy, with `Warning` at the top, `StandardWarning`, `UninitializedWarning`, `RedefinedMethodWarning`, `DeprecatedMethodWarning`, etc. Whatever we can think of.

Such a system would allow you to raise specific warnings within your code:

```
class Foo
  def old_method
    warn DeprecatedMethodWarning, 'This method is deprecated. Use new_method instead'
    # Do stuff
  end
end
```

The ability to explicitly raise specific types of warnings then makes them testable:

```
require 'test/unit'
class TC_Foo_Tests < Test::Unit::TestCase
  def setup
    @foo = Foo.new
  end

  # Assume we've added an assert_warn method to Test::Unit
  def test_old_method
    assert_warn(DeprecatedMethodWarning) { @foo.old_method }
  end
end
```

And, for sake of backwards compatibility and convenience, a call to `Kernel#warn` without an explicit warning type would simply raise a `StandardWarning` in the same way that "raise" without an explicit error type raises a `StandardError`. You may be wondering about `rescue/retry` semantics. My opinion on the matter is that warnings should not be rescuable. They are meant to be informational. They are not meant to control program flow. This also lets us avoid having to worry about `retry` semantics. Not that anyone would `retry` based on a warning in practice.

Unlike `Exceptions` you could permanently or temporarily disable warnings to suit your particular preferences in the system I have in mind. For example, in the `win32-file` library I'm well aware that I've gone and redefined some core `File` methods. When I run any code that uses `win32-file` with the `-w` flag, I get "method redefined" warnings. I don't want to see those because I neither need nor want to be reminded about them. So, using our hypothetical `RedefinedMethodWarning` class, I could disable them like so:

```
RedefinedMethodWarning.disable # No more warnings about method redefinitions!
```

Or, with block syntax, we could disable a particular warning temporarily:

```
# Don't bug me about deprecated method warnings within this block, I know what I'm doing.
DeprecatedMethodWarning.disable{
  [1,2,3,4,5].indexes(1,3) # Array#indexes is a deprecated method
}

# But here I would get a warning since it's outside the block:
[1,2,3,4,5].indexes(1,3)
```

Unlike the current warning system, this would allow users to still receive other types of warnings, instead of the on/off switch we have now. And, in case you were wondering why the structured_warnings library isn't quite sufficient, the answer is that it still can't hook into the existing warnings being raised in core Ruby via `rb_warn()`, like uninitialized variables or redefined methods.

Related issues:

Related to Ruby master - Feature #12026: Support warning processor	Closed
Related to Ruby master - Feature #17122: Add category to Warning#warn	Closed

History

#1 - 11/09/2015 08:02 AM - matz (Yukihiko Matsumoto)

For me, warning is an interface (or indication) to humans. Unlike exceptions, they rarely need to be handled by machines. Considering that fact, this proposed structured warning is too big and complex. Maybe we need something to handle warnings like `assert_deprecated` in Rails.

Matz.

#2 - 12/06/2015 04:21 AM - djberg96 (Daniel Berger)

Matz, warnings may not affect control flow, but I've seen enough real world application logs filled with warnings we couldn't control to dispute your assertion that they don't need to be handled by machines. And right now they can't really be handled by machines or humans very well.

Something like `ActiveSupport::Deprecation` would be good, though I don't know how much different that is than what I'm proposing.

#3 - 12/06/2015 08:48 PM - avdi (Avdi Grimm)

#3 is a big deal IMHO. Having finer-grained control over which warnings are shown is long overdue.

On Tue, Oct 13, 2015 at 11:25 AM djberg96@gmail.com wrote:

Issue [#11588](#) has been reported by Daniel Berger.

Feature [#11588](#): Implement structured warnings
<https://bugs.ruby-lang.org/issues/11588>

- Author: Daniel Berger
- Status: Open
- Priority: Normal

* Assignee:

Ruby's current warning system is lacking. Warnings are controlled by the `-W` flag on the command line, and are generated via the `Kernel#warn` method within code. There are a host of problems with this approach to warnings.

First, warnings aren't currently testable. With `Test::Unit`, for example, I can ensure that specific errors are raised in certain conditions via the `assert_raise` method. There is no analogue for warnings. It would be nice if there were so I could test them.

Second, there is no backtrace information provided with warnings. If I discover a warning I have to wade through the source and figure out where it was generated, because a `Kernel#warn` call does not provide a line number or method name that I can refer back to, unless it happened to be generated by `rb_warn()`. For large code bases that can be problematic.

Third, and most significantly, with warning flags it's all or nothing. I cannot enable or disable specific kinds of warnings. Perl, for example, implements warning control through pragmas. So, for example, I can specify "no warnings uninitialized" in a Perl program and warnings about

uninitialized variables go away. With Ruby it's off, on, or even-more-on (-W0, -W1 or -W2).

What I would like to see are structured warnings. By "structured warnings" I mean a system analogous to the Error class, except that a warning would only emit text to STDERR, not cause the interpreter to exit. In our hypothetical Warning class you still have backtrace information available. And, like Exceptions, there would be a standard hierarchy, with Warning at the top, StandardWarning, UninitializedWarning, RedefinedMethodWarning, DeprecatedMethodWarning, etc. Whatever we can think of.

Such a system would allow you to raise specific warnings within your code:

```
class Foo
  def old_method
    warn DeprecatedMethodWarning, 'This method is deprecated. Use
new_method instead'
    # Do stuff
  end
end
```

The ability to explicitly raise specific types of warnings then makes them testable:

```
require 'test/unit'
class TC_Foo_Tests < Test::Unit::TestCase
  def setup
    @foo = Foo.new
  end

  # Assume we've added an assert_warn method to Test::Unit
  def test_old_method
    assert_warn(DeprecatedMethodWarning){ @foo.old_method }
  end
end
```

And, for sake of backwards compatibility and convenience, a call to Kernel#warn without an explicit warning type would simply raise a StandardWarning in the same way that "raise" without an explicit error type raises a StandardError. You may be wondering about rescue/retry semantics. My opinion on the matter is that warnings should not be rescuable. They are meant to be informational. They are not meant to control program flow. This also lets us avoid having to worry about retry semantics. Not that anyone would retry based on a warning in practice.

Unlike Exceptions you could permanently or temporarily disable warnings to suit your particular preferences in the system I have in mind. For example, in the win32-file library I'm well aware that I've gone and redefined some core File methods. When I run any code that uses win32-file with the -w flag, I get "method redefined" warnings. I don't want to see those because I neither need nor want to be reminded about them. So, using our hypothetical RedefinedMethodWarning class, I could disable them like so:

```
RedefinedMethodWarning.disable # No more warnings about method
redefinitions!
```

Or, with block syntax, we could disable a particular warning temporarily:

```
# Don't bug me about deprecated method warnings within this block, I
know what I'm doing.
DeprecatedMethodWarning.disable{
  [1,2,3,4,5].indexes(1,3) # Array#indexes is a deprecated method
}

# But here I would get a warning since it's outside the block:
[1,2,3,4,5].indexes(1,3)
```

Unlike the current warning system, this would allow users to still receive other types of warnings, instead of the on/off switch we have now. And, in case you were wondering why the structured_warnings library isn't quite sufficient, the answer is that it still can't hook into the existing warnings being raised in core Ruby via rb_warn(), like uninitialized variables or redefined methods.

--

#4 - 03/17/2016 06:51 AM - shyouhei (Shyouhei Urabe)

- Related to Feature #12026: Support warning processor added

#5 - 08/14/2020 02:39 AM - shyouhei (Shyouhei Urabe)

- Related to Feature #17122: Add category to Warning#warn added