Ruby master - Feature #13901

Add branch coverage

09/15/2017 06:23 AM - mame (Yusuke Endoh)

Status:	Closed
Priority:	Normal
Assignee:	
Target version:	

Description

I plan to add "branch coverage" (and "method coverage") as new target types of coverage.so, the coverage measurement library. I'd like to introduce this feature for Ruby 2.5.0. Let me to hear your opinions.

Basic Usage of the Coverage API

The sequence is the same as the current: (1) require "coverage.so", (2) start coverage measurement by Coverage.start, (3) load a program being measured (typically, a test runner program), and (4) get the result by Coverage.result.

When you pass to Coverage.start with keyword argument "branches: true", branch coverage measurement is enabled.

test.rb

```
require "coverage"
Coverage.start(lines: true, branches: true)
load "target.rb"
p Coverage.result
```

target.rb

```
1: if 1 == 0
2:   p :match
3: else
4:   p :not_match
5: end
```

By measuring coverage of target.rb, the result will be output (manually formatted):

```
$ ruby test.rb
:not_match
{".../target.rb" => {
  :lines => [1, 0, nil, 1, nil],
  :branches => {
    [:if, 0, 1] => { [:then, 1, 2] => 0, [:else, 2, 4] => 1 }
  }
}
```

[:if, 0, 1] reads "if branch at Line 1", and [:then, 1, 2] reads "then clause at Line 2". So, [:if,0,1] => { [:then,1,2]=>0, [:else,2,4]=>0 } reads "the branch from Line 1 to Line 2 has never executed, and the branch from Line 1 to Line 4 has executed once."

The second number (0 of [:if, 0, 1]) is a unique ID to avoid conflict, just in case where multiple branches are written in one line. This format of a key is discussed in "Key format" section.

Why needed

Traditional coverage (line coverage) misses a branch in one line. Branch coverage is useful to find such untested code. See the following example.

target.rb

```
p(:foo) unless 1 == 0
p(1 == 0 ? :foo : :bar)
```

The result is:

```
{".../target.rb" => {
  :lines => [1, 1],
  :branches => {
    [:unless, 0, 1] => { [:else, 1, 1] => 0, [:then, 2, 1] => 1 },
    [:if, 3, 2] => { [:then, 4, 2] => 0, [:else, 5, 2] => 1 }
  }
}}
```

Line coverage tells coverage 100%, but branch coverage shows that the unless statement of the first line has never taken true and that the ternary operator has never taken true.

Current status

I've already committed the feature in trunk as an experimental feature. To enable the feature, you need to set the environment variable COVERAGE_EXPERIMENTAL_MODE = true. I plan to activate this feature by default by Ruby 2.5 release, if there is no big problem.

Key format

The current implementation uses [<label>, <unique ID>, <lineno>], like [:if, 0, 1], to represent the site of branch. <unique ID> is required for the case where multiple branches are in one line.

I think this format is arguable. I thought of some other candidates:

- [<label>, <lineno>, <column-no>]: A big problem, how should we handle TAB character?
- [<label>, <offset from file head>]: Looks good for machine readability.
- Are <label> and <lineno> needed? They are useful for human, but normally, this result will be processed by a visualization script (such as SimpleCov).

What do you think? I'm now thinking that [<label>, <lineno>, <offset from file head>] is reasonable, but it is hard for me to implement. I'll try later but parse.y is so difficult... (A patch is welcome!)

Compatibility

This API is 100% compatible. If no keyword argument is given, the result will be old format, i.e., a hash from filename to an array that represents line coverage.

```
# compatiblie mode
Coverage.start
load "target.rb"
p Coverage.result
#=> {".../target.rb" => [1, 1, 1, ...] }

# If "lines: true" is given, the result format differs slightly
Coverage.start(lines: true)
load "target.rb"
p Coverage.result
#=> {".../target.rb" => { :lines => [1, 1, 1, ...] } }
```

Method coverage

Method coverage is also supported. You can measure it by using Coverage.start(methods: true).

target.rb

```
 1: def foo
 2: end
 3: def bar
 4: end
 5: def baz
 6: end
 7:
 8: foo
 9: foo
10: bar
```

result (manually formatted)

```
{".../target.rb"=> {
  :methods => {
    [:foo, 0, 1] => 2,
    [:bar, 1, 3] => 1,
    [:baz, 2, 5] => 0,
  }
}}
```

Notes

- if statements whose condtion is literal, such as if true and if false, are not considered as a branch.
- while, until, and case are also supported. See Examples 2 and 3.
- This proposal is based on #9508. The proposal has some spec-level issues, but the work was really inspiring me.

Future work

- Someone may want to know how many times an one-line block is executed, such as n.times { }.
- Someone may want to know how many times each method call is executed, such as obj.foo.bar.baz (For example, if method foo always raises an exception, calls to bar and baz are not executed.)

Some examples

Example 1

target.rb

```
1: if 1 == 0
2:   p :match1
3: elsif 1 == 0
4:   p :match2
5: else
6:   p :not_match
7: end
```

result (manually formatted)

```
{"target.rb" => {
  :lines => [1, 0, 1, 0, nil, 1, nil],
  :branches => {
    [:if, 1] => { [:then, 2] => 0, [:else, 3] => 1 },
    [:if, 3] => { [:then, 4] => 0, [:else, 5] => 1 },
  }
}
```

Example 2

target.rb

```
 1:case :BOO
 2:when :foo then p :foo
 3:when :bar then p :bar
 4:when :baz then p :baz
 5:else p :other
 6:end
 7:
 8:x = 3
 9:case
10:when x == 0 then p :foo
```

```
11:when x == 1 then p :bar
12:when x == 2 then p :baz
13:else p :other
14:end
```

result (manually formatted)

```
{"target.rb" => {
  :lines => [1, 0, 0, 0, 1, nil, nil, 1, 1, 1, 1, 1, 1, nil],
  :branches => {
    [:case, 1] => {
      [:when, 2] => 0,
      [:when, 3] => 0,
      [:when, 4] => 0,
      [:else, 5] => 1
    },
    [:case, 8] => {
      [:when, 9] => 0,
      [:when, 10] => 0,
      [:when, 11] => 0,
      [:else, 12] => 1
    }
  }
}
```

Example 3

target.rb

```
 1:n = 0
 2:while n < 100
 3:  n += 1
 4:end
```

result (manually formatted)

```
{"target.rb" => {
  :lines => [1, 101, 100, nil],
  :branches => {
    [:while, 2] => {
      [:body, 3] => 100,
      [:end, 5] => 1
    }
  }
}
```

Related issues:

Related to Ruby master - Feature #9508: Add method coverage and branch covera...	**Closed**

History

#1 - 09/15/2017 06:24 AM - mame (Yusuke Endoh)

- Backport deleted (2.2: UNKNOWN, 2.3: UNKNOWN, 2.4: UNKNOWN)

- Tracker changed from Bug to Feature

#2 - 09/15/2017 06:24 AM - mame (Yusuke Endoh)

- Subject changed from Add branch coverage coverage to Add branch coverage

#3 - 11/01/2017 08:17 PM - PragTob (Tobias Pfeiffer)

Hi there,

thanks for this! Any indication if this will land in 2.5 and if it's still the same as specified here?

Background: I'm currently maintaining simplecov (https://github.com/colszowka/simplecov), which will be the primary user of this new feature I imagine. Getting a head start on implementing this might be nice :)

Thanks a lot for all your work!
Tobi

#4 - 11/02/2017 08:53 PM - marcandre (Marc-Andre Lafortune)

My friend Maxime Lapointe and I have been hacking on a pure Ruby gem called “DeepCover” to do branch/method/everything coverage, so we thought we should post some info here for feedback.

We are working actively on it, but the base is written and functional, so it’s probably time to start introducing it to the community at large:
https://github.com/deep-cover/deep-cover

As an example, here's the coverage we can produce for
ActiveSupport: https://deep-cover.github.io/rails-cover/activesupport/
ActiveRecord: https://deep-cover.github.io/rails-cover/activerecord/

Note: we currently use istanbul for output (far from perfect), but are working on a much nicer direct HTML output.

An example of expression that the builtin coverage doesn’t (yet) detect is line 190 of
https://deep-cover.github.io/rails-cover/activerecord/lib/active_record/schema_dumper.rb.html

DeepCover is based on the awesome parser gem. We instrument any Ruby code such that we can know, for any node, how many time it’s been executed. More precisely, for any node, we know how many times control flow has "entered" the node, how many times it has "exited" the node normally, and by deduction how many times control flow has been interrupted (raise, throw, return, next, etc...).

Since it's pure Ruby, it is very easy to then customize the analysis results. For example it is possible to require coverage of implicit else in a case statement or not, for default arguments to be covered too (or not, or only if they're not simple literals), or allow an uncovered raise to be ignored. It's also compatible with Ruby 2.0+ and JRuby.

We aim to provide a powerful API, as well as one that can replace MRI’s while maintaining compatibility.

If experienced Rubyists want to delve into it and contribute, this would be awesome. Please contact us so we can coordinate the efforts.

#5 - 11/02/2017 11:20 PM - mame (Yusuke Endoh)

Marc-Andre, thank you for letting me know! I didn't know it.

DeepCover is so great! I'd like to cooperate with you!
What do you think about the format of Coverage.result? If you have any request, let me know. I'd like to consider it. It would be difficult for the flexibility of pure Ruby, but we'd like to provide a feature for filtering and controlling measurement target. I'm also afraid about performance degradation by pure Ruby implementation. How long does it take to run the test suite of ActiveSupport with and without coverage measurement? Anyway, it would be a good diversity that there are multiple approaches. Thank you for your effort!

Our goal is also the visualization like Istanbul, finer coverage measurement than a line. For the purpose, Yuichiro Kaneko and I are working on improvement of NODE representation so that each NODE object keeps track of not only lineno of the original code, but also more precise and useful position information:

- The first lineno and first column number that the NODE begins
- The last lineno and last column number that the NODE ends

Yuichiro Kaneko has already implemented the column number of the beggining of the NODE in trunk. Here is the current format of Coverage.result for Example 1:

```
{"/home/mame/work/ruby/target.rb"=>
  {:lines=>[1, 0, 1, 0, nil, 1, nil],
   :branches=>
    {[:if, 0, 3, 0]=>{[:then, 1, 4, 2]=>0, [:else, 2, 6, 2]=>1},
     [:if, 3, 1, 0]=>{[:then, 4, 2, 2]=>0, [:else, 5, 3, 0]=>1}},
   :methods=>{}}}
```

[:if, 0, 3, 0] reads "the if statement that starts from line 3 and column 0", and [:then, 1, 4, 2] reads "the if statement that starts from line 4 and column 2". (Please ignore the second element. When the column number is not implemented yet, it was needed as a unique number to avoid a conflict.)

It is already possible to identify the last position, but we need more work. This is just an implementation issue, but a NODE object has just one word for keeping the information because of the GC limitation. I'm trying to detach NODEs from GC management, so that we can freely change the data structure of NODEs. (This is my personal idea, but I want this work to lead to the embedded parser API like "perser" gem.)

#6 - 11/03/2017 06:31 AM - marcandre (Marc-Andre Lafortune)

mame (Yusuke Endoh) wrote:

> What do you think about the format of Coverage.result?

Having a character position is a big improvement on just a line #, that's for sure. It makes it possible to pinpoint exactly what we're talking about, and with the help of parser or similar, makes it possible to get all the rest of the information that might be wanted.

I imagine you are planning on covering ||, &&, &. and ? :, right?

> I'm also afraid about performance degradation by pure Ruby implementation. How long does it take to run the test suite of ActiveSupport with and without coverage measurement?

I'm getting a 15% increase in time to run the tests for activesupport, for example (not counting instrumenting or analysis, just running rake).

We have not looked much into this, but I'm not worried about this. It's not something that's needed quickly usually (since you want to run the full test suite), and a pure Ruby implementation should only scale linearly in the worst case.
Our implementation uses only trackers of the form $_global[1][2] += 1, and often storage in a local variable (like temp = foo.bar(1,2,3); $_global[1][2] +=2; temp) or some condition like ... if temp or && $_global_var[...] +=1. This is all optimized by Ruby and doesn't require method dispatch. We also use the strict minimum number of trackers required (except for multiple assignments where we use one more tracker than theoretically possible).

> It is already possible to identify the last position, but we need more work. This is just an implementation issue, but a NODE object has just one word for keeping the information because of the GC limitation. I'm trying to detach NODEs from GC management, so that we can freely change the data structure of NODEs. (This is my personal idea, but I want this work to lead to the embedded parser API like "perser" gem.)

This sounds great. The precise beginning (column and line) is the only thing strictly necessary though; the rest could be obtained by parsing the code with parser and locating the node in question. In any case, tools may have to do that anyways for really good output, because beginning and end is not necessarily enough. For example parser has very detailed information about a node; it can tell you the beginning and end of the whole node's expression, but also of the keyword, for example, etc. Also it gives information about children nodes. We use this to output very precisely, for example the punctuation like ,(); are considered non executable, etc.

For example, deep-cpver will color 0&.nonzero?&.foo(1, 2, 3).nil? all in green, but "foo(1, 2, 3)" in red (well, the (), are gray, actually)

A minor issue is that branch coverage will not solve cases of exceptions being raised. Check the last line (char coverage) of:

$ deep-cover -e "puts(:a, notdefined, :c) rescue nil"

The first argument (:a) is in green, the last one (:c) in red, because notdefined raised a NoMethodError. Is there a plan to implement such detailed coverage?

#7 - 11/03/2017 08:57 AM - mame (Yusuke Endoh)

marcandre (Marc-Andre Lafortune) wrote:

> I imagine you are planning on covering ||, &&, &. and ? :, right?

? : has been already supported. Yuichiro has experimentally implemented &. coverage. I think we should support || and && too.
(Formerly, I had no intention of supporting them, but Yuichiro implemented column numbers, so now we will be able to do that.)

> > I'm also afraid about performance degradation by pure Ruby implementation. How long does it take to run the test suite of ActiveSupport with and without coverage measurement?
>
> I'm getting a 15% increase in time to run the tests for activesupport, for example (not counting instrumenting or analysis, just running rake).

Awesome. Seems a good approach in the case where instrumenting code itself is acceptable.

> > It is already possible to identify the last position, but we need more work. This is just an implementation issue, but a NODE object has just one word for keeping the information because of the GC limitation. I'm trying to detach NODEs from GC management, so that we can freely change the data structure of NODEs. (This is my personal idea, but I want this work to lead to the embedded parser API like "perser" gem.)
>
> This sounds great. The precise beginning (column and line) is the only thing strictly necessary though; the rest could be obtained by parsing the code with parser and locating the node in question.

obj.foo.foo has two NODE_CALLs. It is difficult to distinguish them by only the beginning, unless we have other information about the node in question. Honestly, I'm unsure if we can distinguish all relevant NODEs by only a pair of the beginning and the end.

> For example, deep-cpver will color 0&.nonzero?&.foo(1, 2, 3).nil? all in green, but "foo(1, 2, 3)" in red (well, the (), are gray, actually)

This is what I'm now aiming (Ko1 called it "callsite coverage"). Great.

> A minor issue is that branch coverage will not solve cases of exceptions being raised. Check the last line (char coverage) of:
>
> $ deep-cover -e "puts(:a, notdefined, :c) rescue nil"

> The first argument (:a) is in green, the last one (:c) in red, because notdefined raised a NoMethodError. Is there a plan to implement such detailed coverage?

I had no thought of counting trivial evaluation such as a literal, but if your pure-Ruby approach does not cause performance issue, now I'd like to consider supporting it. It would be difficult (for me) to implement all in 2.5, though.

#8 - 11/03/2017 04:34 PM - marcandre (Marc-Andre Lafortune)

mame (Yusuke Endoh) wrote:

> obj.foo.foo has two NODE_CALLs. It is difficult to distinguish them by only the beginning, unless we have other information about the node in question. Honestly, I'm unsure if we can distinguish all relevant NODEs by only a pair of the beginning and the end.

It depends what you consider the beginning position, but if you return the beginning of the method sending (4 vs 8 in your example, or 5 vs 9), then it's easy to distinguish between the two NODE_CALLs. These correspond to dot, or selector range in parser (try ruby-parser -L -e "obj.foo.foo" :-)

If you only provide only the beginning of the expression (1 vs 1), then yes, the end position is needed in that case.

> > For example, deep-cpver will color 0&.nonzero?&.foo(1, 2, 3).nil? all in green, but "foo(1, 2, 3)" in red (well, the (), are gray, actually)
>
> This is what I'm now aiming (Ko1 called it "callsite coverage"). Great.

Cool :-)

> > A minor issue is that branch coverage will not solve cases of exceptions being raised. Check the last line (char coverage) of:
> >
> > $ deep-cover -e "puts(:a, notdefined, :c) rescue nil"
> >
> > The first argument (:a) is in green, the last one (:c) in red, because notdefined raised a NoMethodError. Is there a plan to implement such detailed coverage?
>
> I had no thought of counting trivial evaluation such as a literal, but if your pure-Ruby approach does not cause performance issue, now I'd like to consider supporting it. It would be difficult (for me) to implement all in 2.5, though.

Just to be clear: literals (or anything that can not change control flow) do not cause *any performance loss at all* for us.

We always deduce their execution from normal control flow. E.g. in var = [1, 2, 3], to know if 3 was executed, we check if it's previous sibbling (2) was executed. To know that, we check if 1 was executed. 1 has no previous sibbling, so we check the parent []. It itself checks the parent a =, which finally asks bring us to our parent root. Only this parent introduces a small performance hit with a counter $global[][] += 1. So we get the execution count of the literals, the array creation and the variable assignment for free.

To summarize our strategy: a Node is not responsible to know how many times it was entered by control flow, that is the responsibility of the parent node. A Node's responsibility is to know how many times control flow exited it normally. For many nodes like literals, exit = entry so there's nothing special to be done. A "send" node must add a $tracker+=1 after the call to know that. I need to credit Maxime Lapointe for pointing us in that direction early on.

#9 - 11/03/2017 11:38 PM - mame (Yusuke Endoh)

marcandre (Marc-Andre Lafortune) wrote:

> mame (Yusuke Endoh) wrote:
>
> > obj.foo.foo has two NODE_CALLs. It is difficult to distinguish them by only the beginning, unless we have other information about the node in question. Honestly, I'm unsure if we can distinguish all relevant NODEs by only a pair of the beginning and the end.
>
> It depends what you consider the beginning position, but if you return the beginning of the method sending (4 vs 8 in your example, or 5 vs 9), then it's easy to distinguish between the two NODE_CALLs. These correspond to dot, or selector range in parser (try ruby-parser -L -e "obj.foo.foo" :-)

The parser gem produces a good work.

> > I had no thought of counting trivial evaluation such as a literal, but if your pure-Ruby approach does not cause performance issue, now I'd like to consider supporting it. It would be difficult (for me) to implement all in 2.5, though.
>
> Just to be clear: literals (or anything that can not change control flow) do not cause *any performance loss at all* for us.

> We always deduce their execution from normal control flow. E.g. in var = [1, 2, 3], to know if 3 was executed, we check if it's previous sibbling (2) was executed. To know that, we check if 1 was executed. 1 has no previous sibbling, so we check the parent []. It itself checks the parent a =, which finally asks bring us to our parent root. Only this parent introduces a small performance hit with a counter $global[][] += 1. So we get the execution count of the literals, the array creation and the variable assignment for free.
>
> To summarize our strategy: a Node is not responsible to know how many times it was entered by control flow, that is the responsibility of the parent node. A Node's responsibility is to know how many times control flow exited it normally. For many nodes like literals, exit = entry so there's nothing special to be done. A "send" node must add a $tracker+=1 after the call to know that. I need to credit Maxime Lapointe for pointing us in that direction early on.

I see... It is a very interesting approach, but it seems to completely ignore asynchronous interrupt, such as signal and Thread#raise. I wonder if it is not a good design decision for the embedded coverage measurement library. I'll think about it. Thank you for teaching me, anyway.

#10 - 11/04/2017 12:13 AM - marcandre (Marc-Andre Lafortune)

mame (Yusuke Endoh) wrote:

> I see... It is a very interesting approach, but it seems to completely ignore asynchronous interrupt, such as signal and Thread#raise. I wonder if it is not a good design decision for the embedded coverage measurement library. I'll think about it. Thank you for teaching me, anyway.

Indeed, it ignores that completely. I can not even remotely imagine a case where it could possibly matter.

#11 - 11/04/2017 01:09 AM - MaxLap (Maxime Lapointe)

marcandre (Marc-Andre Lafortune) wrote:

> mame (Yusuke Endoh) wrote:
>
> > I see... It is a very interesting approach, but it seems to completely ignore asynchronous interrupt, such as signal and Thread#raise. I wonder if it is not a good design decision for the embedded coverage measurement library. I'll think about it. Thank you for teaching me, anyway.
>
> Indeed, it ignores that completely. I can not even remotely imagine a case where it could possibly matter.

To add a little bit to the discussion: The only information that you lose when an asynchronous interrupt like Thread.raise happens are the details on which ones of multiple sequential things that can never raise/change control flow were done or not (if the interrupt happened during the execution of the sequence). By that I mean things such as:

```
my_var = 1
['here is', 'a big' , 'array', ['of', 6, 'literals']]
other_var = my_var
a_hash = {1: 3}
```

It doesn't matter, in the sense of Coverage, weither the assignment of one variable to the other happened or not, or that the array creation was completed or not. These are things without side effects for the program that literally cannot fail (aside from asynchronous interrupt and OutOfMemory).

As soon as something "interesting" can happen, we have trackers to know if the thing was interrupted or not. By interesting, I mean literally anything that can raise an exception or change control flow: send, define a method, if, case, loops, constants, rescue, ensure, etc...

#12 - 11/18/2017 05:26 AM - marcandre (Marc-Andre Lafortune)

We're thinking about an output format (for simplecov in particular) and I wanted to suggest for branch, method and "callsite" coverage to separate the "map" data from the "runs" data.

What I call the "map" data is all the information that specifies what the branches/methods/callsites are (type, line, column, ...), while the "runs" data is the number of time it is run (or eventually nil if it is not executable).

```
{ "path" => {
      :callsite_map    => [range, other_range, …],
      :callsite_runs   => [runs,  other_runs, …],
      :branch_map  => [branch,    other_branch, …],
      :branch_runs => [[runs, …], [other_runs, …], …],
      ...
  },
  "other_path" …
}
```

The reason to separate the run data from the map is to allow for easier merging, and more efficient output in different files more efficiently too. The map can be written once while the runs can be written as many times as needed and merged at the end.

Imagine a big app that runs a test suite in 42 parallel processes, and that callsite coverage is generated. One could generate the callsite map once, write it down, and each of the 42 process could write the runs at the end. The map for each callsite would probably be at least 4 integers (start_line, start_column, end_line, end_column), with maybe a type (:send, :def, etc.), so would be quite a bit larger than each runs file. The 43 files would be easier to merge later on too.

This splitting is what Istanbul (the coverage tool of Google's Babel) is doing. It is the direction that DeepCover is likely to take too.

Note that this is "compatible" with the existing line coverage in the sense that for line coverage, the "map" is trivial and thus not needed; only the runs are output. It's only for the other coverage types that the map has to be specified.

#13 - 02/28/2018 01:06 AM - mame (Yusuke Endoh)

- Status changed from Open to Closed

Thank you marcandre for proposing the format, and really sorry for my super-late reply.

I considered it seriously (before release of 2.5.0). I understand its advantage (easy and efficient to merge). But I thought it was difficult to implement it for method coverage. The order and the count of methods in coverage data may vary because some methods may or may not be defined according to environment due to Ruby's dynamic property (i.e., conditional define_method). In this case, merging method coverage data blindly depending upon only the index of "runs" data, may lead to broken coverage data.

To make it easy to merge coverage data, I released coverage-helpers gem which includes:

- Coverage::Helpers.merge(*covs): Sum up all coverage results.
- Coverage::Helpers.diff(cov1, cov2): Extract the coverage results that is covered by cov1 but not covered by cov2.

I hope this gem helpful for users.

I'm closing this ticket since Ruby 2.5.0 has been already released, but let me know if you have any idea to make coverage.so helpful for any use case (including DeepCover).

#14 - 01/21/2019 08:00 AM - mame (Yusuke Endoh)

- Related to Feature #9508: Add method coverage and branch coverage metrics added