Feature 4796(https://www.ruby-forum.com/topic/1811306#1001864) appears to have a bug in it.

While the test listed there does pass the following test does not:

```ruby
require "test/unit"
require "coverage"
require 'tmpdir'
class TestCoverage < Test::Unit::TestCase
def test_restarting_coverage
  Dir.mktmpdir {|tmp|
    Dir.chdir(tmp) {
      File.open("test.rb", "w") do |f|
        f.puts <<-EOS
          def coverage_test_method
            puts :ok
          end
        EOS
      end
      Coverage.start
      require tmp + '/test.rb'
      Coverage.result
      Coverage.start
      coverage_test_method
      result = Coverage.result
      assert_equal 1, result.size
      assert_equal [0, 1, nil], result.first[1] # coverage stats show an empty array here
    }
  }
end
def coverage_test_method
  puts :ok
end
end
```

It appears that while the coverage is finding the correct files it is not giving any coverage stats for those files. Knowing this information would be very helpful in determining test coverage data for individual test files. I'm not very familiar with how the coverage library works, but if you can point me at where to look I can give fixing it a try.

Thanks,

Sean Ferguson

---

**History**

#1 - 03/02/2014 07:08 AM - srawlins (Sam Rawlins)

Good catch. My guess is that #4796 is not implemented in the way you are hoping for, and cannot easily be implemented in this way either.

The original test for #4796 (contributed by xavier-shay in #4796), before r33030 [1], only asserted that the file still existed in the Coverage.result hash, not that the hash value for that file key was correct. r33030 changed the test to in fact assert that the hash value is an empty something (#size == 0).

So #4796 allows Coverage to be restartable, but with the following caveats:
Only new files that are required after Coverage is restarted (and all files that are loaded after Coverage is restarted) will be properly traced.

Files that were required before Coverage is restarted will exist in the Coverage.result hash, but with empty hash values.

This can be changed so that files required before Coverage is restarted will have inaccurate coverage (rather than empty coverage). I think you acknowledge this in your amended test, where you assert that the coverage report for the test file should be [0,1,nil]. That is, the def coverage_test_method line is not covered, because it was parsed earlier, before Coverage was restarted, but the puts :ok line is still covered, because the method is called after Coverage is restarted.

I'm not really in favor of knowingly reporting inaccurate data on restart, but you could get this functionality (and pass your test) with the following patch:

```diff
diff --git a/ext/coverage/coverage.c b/ext/coverage/coverage.c
index 93cb2a5..770d81c 100644
--- a/ext/coverage/coverage.c
+++ b/ext/coverage/coverage.c
@@ -38,8 +38,14 @@
    VALUE path = (VALUE)key;
    VALUE coverage = (VALUE)val;
    VALUE coverages = (VALUE)h;
+   long i;
    coverage = rb_ary_dup(coverage);
-   rb_ary_clear((VALUE)val);
+   for (i=0; i<RARRAY_LEN(val); i++) {
+      if (RARRAY_AREF(val, i) != Qnil)
+         RARRAY_ASET(val, i, INT2FIX(0));
+   }
+   rb_ary_freeze(coverage);
    rb_hash_aset(coverages, path, coverage);
    return ST_CONTINUE;
```


#2 - 06/15/2014 01:51 PM - danmayer (Dan Mayer)

I have been having this problem as well.

It seems good that Coverage.result resets the coverage data, but once Coverage.start is called again it seems like it should start building up new coverage from that point on. In the example code I have for instance I would expect the second call to coverage results, to not show method_a being called, but it should show method_b.

Check this small git repo for an example of the problem I am seeing. https://github.com/danmayer/coverage-bug

Without being able to collect additional coverage data after restarting it makes gathering coverage data via sampling not possible. Perhaps this is the expected output, if so could someone explain expected behavior or why the current way would be preferred.

#3 - 01/05/2018 09:00 PM - naruse (Yui NARUSE)
- Target version deleted (2.2.0)

#4 - 01/14/2018 05:55 PM - danmayer (Dan Mayer)

- ruby -v changed from uby 2.0.0p353 (2013-11-22 revision 43784) [x86_64-darwin12.4.0] to ruby 2.6.0dev (2018-01-13 trunk 61811) [x86_64-darwin15]

I would like to work on closing this out but I believe due to changes in the API expectations since the bug was filed this is likely to not be considered a bug anymore but more likely to be considered to be expanding the feature set. I am thinking the best way to do this is to expand the public API of Coverage a bit.

I have detailed the bug and feature in this small repo to explore the interface: https://github.com/danmayer/coverage-bug

Details copied here to make sure they are in the Ruby system of record.

This shows a bug with Ruby's Coverage, as tracked on ruby bug tracker #9572.

to execute just run: ruby ./example.rb

This demonstrates why Coverage from the ruby std-lib isn't useful for the kind of data I want to get from coverband.

previous output (Ruby <= 2.1.1)

method_a called
"[/Users/danmayer/projects/coverage-bug/covercheck.rb]=>[1, 1, nil, nil, 1, 0, nil, nil, 1, 1, nil, 0, nil, 0,
this doesn't matter and is between coverage
method_b_called
"/Users/danmayer/projects/coverage-bug/covercheck.rb"=>[]

current output:
method_a_called
puts "this line is run between coverage being enabled and shouldn't get tracked"
method_b_called
{"/Users/danmayer/projects/coverage-bug/covercheck.rb"=>[0, 0, nil, nil, 1, 1, nil, nil, 1, 0, nil, 1, 1, 1]}

This could allow some extremely interesting code usage tracking, such as pausing and resuming in before / after hooks in Rails to track usage only in particular routes, controllers, modules, or etc. Using reset you could get just changes during a given run, or user could not reset the data and build up a results set over time, but still only incur the performance overhead on a very specific section of code.

I intend to be able to expand the capabilities of Coverband using this feature as well as reduce the performance costs it currently incurs to run it.

#5 - 01/14/2018 08:31 PM - tenderlovemaking (Aaron Patterson)
- Backport deleted (1.9.3: UNKNOWN, 2.0.0: UNKNOWN, 2.1: UNKNOWN)

I have two questions:

1. How is this different than recording the current coverage with peek_result, then subtracting later? Like this for example:
   https://gist.github.com/tenderlove/eefcce7dcadb0ed48968f57569b9e3f04

2. Is there a reason you need to call Coverage methods from within files whose coverage is being measured? It seem strange for an observation tool to be observing itself.

I really don't like the idea of mutating a global (the coverage information). I can't say I fully understand the problem you're trying to solve, but it seems like being able to record coverage information at any time, then being able to calculate differences would be flexible enough to deal with any coverage situation you need.

#6 - 01/14/2018 09:10 PM - danmayer (Dan Mayer)

Thanks for the reply... So you are right about the calculating the current coverage with peek_result and if we want to avoid complexity, it would be fine to drop the reset method and have the user app handle calculating the difference. The reset portion of this chance is really a convenience method.

The more important piece of the change is being able to pause and resume coverage. Especially being able to continue getting coverage results for files already required. As it can have significant overhead and it is nice to turn it on to collect what you need but only have it running for that.

I will get a better example, tomorrow, that shows a more realistic example than calling Coverage methods directly in files being measured. That doesn't illustrate the idea very well. A very quick way to think about the concept would be a rack middleware that would let you record coverage of any request that had a coverage=true param appended.

    record_coverage do
      calls_lots_of_app_code_some_code
    end

with something like that method handle setting up coverage, handling recording all the information, calculating the diff from peek_result and then formatting it for visualization.

#7 - 01/14/2018 11:53 PM - tenderlovemaking (Aaron Patterson)

The more important piece of the change is being able to pause and resume coverage.

Pause / Resume just seems like ignoring coverage information for some code. This case seems like you can handle by taking snapshots around the execution you want to ignore, and doing the appropriate math.

Especially being able to continue getting coverage results for files already required.

This might be pretty tricky. Enabling code coverage changes the way code is compiled. Since the code is compiled when you require it, you'd have to go back and recompile it. Here is an example:

    require 'coverage'

    code = <<-eoruby
      def method_a
        puts "method_a called"
      end

      def method_b
        puts "method_b called"
      end
    end

    method_a
    method_b
If you run this code, you’ll see that the instructions emitted are different on the second call (specifically calls like 0028 trace 131072). The coverage tool uses these instructions to record coverage information. Files that have been required before the coverage tool has been enabled simply don’t have these instructions.

Pause / Resume just seems like ignoring coverage information for some code.

Yes, the goal would be to ignore it for some code, but with reduced performance costs.

My thought was much of the performance cost was related to tracking the events, if most of the cost is incurred at compile time (as I don't think there is anything I could do about that) the feature might not be very helpful. I was primarily thinking if one could pause the event listeners, and performance would go nearly back to normal.

You can see the POC that I am using to test this idea here: https://github.com/danmayer/ruby/compare/trunk...danmayer:feature/coverage_pause

I get that to get any coverage data Coverage.start would need to be called before requiring files.

Especially being able to continue getting coverage results for files already required

What was getting at here is currently if you call Coverage.result then Coverage.start again, it stops tracking previously tracked files. Only NEWLY required files after the second start are tracked. I think with your current peek_result this issue can already be considered solved. There just isn't a good reason to ever call Coverage.start a second time.

I guess the question and value of the change depends on if the act of turning on and off the above event hooks can result in worthwhile performance benefits. Or if most cost is added once we compile files after calling Coverage.start. Does that make more sense?

I expanded my sample application here: https://github.com/danmayer/coverage-bug/blob/master/example.rb to try to show running

- without coverage at all
- traditional coverage support as it exists today
- being able to pause / resume coverage during execution

The results are below, but I think I need to come up with a more realistic example to get a better idea of the performance impacts. I added some comments to try to make it a bit more clear

# coverage.start is never called
[master]~/projects/coverage-bug ruby example.rb
user system total real
11.261481 0.212573 11.474054 ( 11.548027)
coverage
{}
done

#coverage.start and coverage.result are used
[master]~/projects/coverage-bug COVERAGE=true ruby example.rb
user system total real
11.673899 0.198156 11.872055 ( 11.946569)
coverage
428214, 428214, nil, 2001, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, nil, 1, 0, 0, nil, nil], "/Users/danmayer/projects/coverage-bug/app.rb"=>[1, 1, 2, 2, 2, nil, 2, 2, 1, 0, 0, nil, 1, nil, nil]} done

# coverage.start, coverage.pause, and coverage.resume used along with coverage.peek_result

[master] /-projects/coverage-bug] ENHANCED_COVERAGE=true ruby example.rb

user system total real
11.467796 0.196386 11.664182 ( 11.729636)
12.113387 0.185672 12.299059 ( 12.365711)

done

#9 - 01/15/2018 09:56 PM - dsferreira (Daniel Ferreira)
Aaron, why aren’t we seeing email notifications from your comments? Or is it just me?

#10 - 01/15/2018 10:03 PM - danmayer (Dan Mayer)
I am seeing emails from Aaron's comments.

#11 - 01/15/2018 10:07 PM - dsferreira (Daniel Ferreira)
Well, well, well Aaron. What have you been doing? Gmail has an eye on you!
Your messages are being considered spam. :-)
Is it just me again?

#12 - 03/11/2018 09:03 PM - danmayer (Dan Mayer)
OK, this lead me on a big journey of chasing better benchmarking related to Coverage and Coverband, which is my primary use case for Coverage. I learned a lot about getting better benchmarking data as well as where the actual time costs of working with Coverage is.

I am fine if folks want to close this bug, but I think that calling Stop and Start doesn't match assumptions of what the user would expect.
* As Aaron [Aaron Kelly] Patterson pointed out the code is actually compiled differently, and that isn't undone after calling stop
* All that happens with stop is that we stop listening to the emitted events and updating the internal hash, we also stop adding the extra compiled lines to other required files
* This still makes me think that if you call to stop and then start it should start collecting data for all the available files while coverage was enabled. Not just the most recent set of files.

Further after extensive benchmarking, there is no reason to introduce a Coverage.pause or Coverage.resume feature. While the Ruby branch I built with those features works, benchmarking shows it has nearly 0 impacts on the actual performance of running any significant Ruby code.
* In case anyone ever cares, here is the implementation of pause, resume, reset https://github.com/danmayer/ruby/compare/trunk...danmayer:feature/coverage_pause

For full details on the benchmarking, you can see this project and the readme that benchmarks Coverage in a bunch of different ways including the above pause / resume features. https://github.com/danmayer/coverage_rails_benchmark

In the end, this showed me theCoverage.peek_result is definitely good enough to do what I wanted from the beginning and my assumption of how slow it was to have coverage enabled vs trying to utilize that data was misguided. The deepened understanding I have now of the costs should allow me to improve the performance of Coverband using Coverage opposed to the current Tracepoint API by orders of magnitude, all of which has been possible since Coverage.peek_result was initial introduced, I just didn't understand the full implications of that feature and how it related to performance.

#13 - 10/21/2019 09:53 PM - jeremyevans0 (Jeremy Evans)
- Status changed from Open to Closed