Production Breakpoints

Dale Hamel

12/03/19 03:20:31 AM UTC

Table of Contents

This is approximately a 15-minute read.

This document is also available in [epub](./output/doc.epub) and [pdf](./output/doc.pdf) format if you prefer.

# Status

* Still very much under heavy development and experimentation
* Shifted direction to be tied to Ruby TracePoints
* Original implementation tied to eval to re-interpret method code is still more functional, but less practical for production use.

# Injecting breakpoints

There are multiple ways to inject breakpoints into running code, each with different tradeoffs.

## Unmixer Approach

I set out to prove if I could build a line-targeting debugger with USDT tracepoints, and did so via modifying the class hierarchy, generating an override method on-the-fly.

I was able to do this with “eval”, and be passing the current binding to a block that calls eval, I was able to wrap the original code in my own block handler, but still be able to execute it in the current context!

This was a proof of concept at least. However, I quickly learned from my colleague Alan Wu, that this type of monkeypatching had dangerous implications to the ruby method cache which I had not considered.

Ruby “production breakpoints” would originally use its AST parsing to rewrite the source code of a method with the targeted lines to include a wrapper around those lines. This is still the most powerful way to do this from outside of the RubyVM I have been able to determine.

The method is redefined by prepending a module with the new definition to the parent of the original method, overriding it. To undo this, the module can be ‘unprepended’ restoring the original behavior.

## Using the Ruby TracePoint API

Ruby now supports tracing only particular methods [[1](#ref-ruby-tracing-feature-15289)] instead of all methods, and looks to be aiming to add a similar sort of debugging support natively.

The docs are not very thorough in the official rubydoc for this [[2](#ref-ruby-2-6-tracing-docs)], but it is further documented in the [[3](#ref-ruby-prelude-targetted-tracing)] prelude.rb file:

# call-seq:  
 # trace.enable(target: nil, target\_line: nil, target\_thread: nil) -> true or false  
 # trace.enable(target: nil, target\_line: nil, target\_thread: nil) { block } -> obj  
 #  
 # Activates the trace.  
 #  
 # Returns +true+ if trace was enabled.  
 # Returns +false+ if trace was disabled.  
 #  
 # trace.enabled? #=> false  
 # trace.enable #=> false (previous state)  
 # # trace is enabled  
 # trace.enabled? #=> true  
 # trace.enable #=> true (previous state)  
 # # trace is still enabled  
 #  
 # If a block is given, the trace will only be enabled within the scope of the  
 # block.  
 #  
 # trace.enabled?  
 # #=> false  
 #  
 # trace.enable do  
 # trace.enabled?  
 # # only enabled for this block  
 # end  
 #  
 # trace.enabled?  
 # #=> false  
 #  
 # +target+, +target\_line+ and +target\_thread+ parameters are used to  
 # limit tracing only to specified code objects. +target+ should be a  
 # code object for which RubyVM::InstructionSequence.of will return  
 # an instruction sequence.  
 #  
 # t = TracePoint.new(:line) { |tp| p tp }  
 #  
 # def m1  
 # p 1  
 # end  
 #  
 # def m2  
 # p 2  
 # end  
 #  
 # t.enable(target: method(:m1))  
 #  
 # m1  
 # # prints #<TracePoint:line@test.rb:5 in `m1'>  
 # m2  
 # # prints nothing  
 #  
 # Note: You cannot access event hooks within the +enable+ block.  
 #  
 # trace.enable { p tp.lineno }  
 # #=> RuntimeError: access from outside  
 #

Alan Wu [[4](#ref-xrxr)] tipped me off to this, and showed by an initial prototype gist [[5](#ref-xrxr-gist)], which succinctly shows a basic usage of this:

class Foo  
 def hello(arg = nil)  
 puts "Hello #{arg}"  
 end  
end  
  
one = Foo.new  
two = Foo.new  
  
one.hello  
two.hello  
  
trace = TracePoint.new(:call) do |tp|  
 puts "intercepted! arg=#{tp.binding.local\_variable\_get(:arg)}"  
end  
trace.enable(target: Foo.instance\_method(:hello)) do  
 one.hello(:first)  
 two.hello(:second)  
end

From this, we can build handlers that run using the RubyVM TracePoint object.

This object has full access to the execution context of the caller it would seem.

# Specifying breakpoints

A global config value:

ProductionBreakpoints.config\_file

Can be set to specify the path to a JSON config, indicating the breakpoints that are to be installed:

FIXME {.json include=src/ruby-production-breakpoints/test/config/test\_load.json}

These values indicate:

* type: the built-in breakpoint handler to run when the specified breakpoint is hit in production.
* source\_file: the source repository-root relative path to the source file to install a breakpoint within. (note, the path of this source folder relative to the host / mount namespace is to be handled elsewhere by the caller that initiates tracing via this gem)
* start\_line: The first line which should be evaluated from the context of the breakpoint.
* end\_line: The last line which should be evaluated in the context of the breakpoint
* trace\_id: A key to group the output of executing the breakpoint, and filter results associated with a particular breakpoint invocation

Many breakpoints can be specified. Breakpoints that apply to the same file are added and removed simultaneously. Breakpoints that are applied but not specified in the config file will be removed if the config file is reloaded.

# Built-in breakpoints

## Latency

Output: Integer, nanoseconds elapsed

The latency breakpoint provides the elapsed time from a monotonic source, for the duration of the breakpoint handler.

This shows the time required to execute the code between the start and end lines, within a given method.

Handler definition:

TRACEPOINT\_TYPES = [Integer].freeze  
  
 class << self  
 def start\_times  
 @start\_times ||= {}  
 end  
  
 def start\_times\_set(tid, ns)

## Locals

Output: String, key,value via ruby inspect

The ‘locals’ breakpoint shows the value of all locals.

NOTE: due to limitations in eBPF, there is a maximum serializable string size. Very complex objects cannot be efficiently serialized and inspected.

# Show local variables and their values  
 class Locals < Base # FIXME: refactor a bunch of these idioms into Base  
 TRACEPOINT\_TYPES = [String].freeze  
  
 def handle(vm\_tracepoint)  
 return unless @tracepoint.enabled?  
  
 locals = vm\_tracepoint.binding.local\_variables  
 vals = locals.map do |v|

## Inspect

Output: String, value via ruby inspect

The inspect command shows the inspected value of whatever the last expression evaluated to within the breakpoint handler block.

class Inspect < Base  
 TRACEPOINT\_TYPES = [Integer, String].freeze  
  
 # FIXME!!! Could be \*\*VERY\*\* Unsafe  
 # If inspect is allowed to be called on a method, it wil be extremely unsafe.  
 # it will actually call the method a second time, which modifies the code

## Ustack

Output: String, simplified stack caller output

# Internals

## Installing breakpoints

This gem leverages the ruby-static-tracing [[6](#ref-ruby-static-tracing)] gem [[7](#ref-ruby-static-tracing-gem)] which provides the **secret sauce** that allows for plucking this data out of a ruby process, using the kernel’s handling of the Intel x86 “Breakpoint” int3 instruction, you can learn more about that in the USDT Report [[8](#ref-usdt-report-doc)]

For each source file, a ‘shadow’ ELF stub is associated with it, and can be easily found by inspecting the processes open file handles.

After all breakpoints have been specified for a file, the ELF stub can be generated and loaded. To update or remove breakpoints, this ELF stub needs to be re-loaded, which requires the breakpoints to be disabled first. To avoid this, the scope could be changed to be something other than file, but file is believed to be nice and easily discoverable for now.

The tracing code will noop, until a tracer is actually attached to it, and should have minimal performance implications.

## Ruby Override technique via Unmixer

The Unmixer gem hooks into the ruby internal header API, and provides a back-door into the RubyVM source code to unprepend classes or modules from the global hierarchy.

An anonymous module is created, with the modified source code containing our breakpoint handler.

To enable the breakpoint code, this module is prepended to the original method’s parent. To undo this, the module is simply ‘unprepended’, a feature unmixer uses to tap into Ruby’s ancestry hierarchy via a native extension.

@tracepoint = StaticTracing::Tracepoint.new(@provider\_name,  
 @name,  
 \*self.class.const\_get('TRACEPOINT\_TYPES'))  
 @trace\_lines = (@start\_line..@end\_line).to\_a  
 @vm\_tracepoints = {}  
 end  
  
 def install  
 @trace\_lines.each do |line|  
 puts "Adding tracepoint for #{line}"

## Dynamically redefined methods

We define a ‘handler’ and a ‘finisher’ block for each breakpoint we attach.

Presently, we don’t support attaching multiple breakpoints within the same function, but we could do so if we applied this as a chain of handers followed by a finalizer, but that will be a feature for later. Some locking should exist to ensure the same method is not overriden multiple times until this is done.

These hooks into our breakpoint API are injected into the method source from the locations we used the ruby AST libraries to parse:

end  
end

And the outcome looks something like this:

This is called when the breakpoint is handled, in order to evaluate the whole method within the original, intended context:

def uninstall  
 @vm\_tracepoints.each\_value(&:disable)  
 end  
  
 def load  
 @tracepoint.provider.enable  
 end  
  
 def unload

This caller binding is taken at the point our handler starts, so it’s propagated from the untraced code within the method. We use it to evaluate the original source within the handler and finalizer, to ensure that the whole method is evaluated within the original context / binding that it was intended to. This should make the fact that there is a breakpoint installed transparent to the application.

The breakpoint handler code (see above) is only executed when the handler is attached, as they all contain an early return if the shadow “ELF” source doesn’t have a breakpoint installed, via the enabled? check.

# Experimentation with caching iseq to reduce overhead of eval

* [Make iseq eval consistent with Kernel eval](https://github.com/ruby/ruby/pull/2298) [[9](#ref-ruby-iseq-eval-patch)] is an attempt to revive an older patch to allow for evaluate an instruction sequence against an arbitrary binding. More details of this patch are [below](#ruby-iseq-eval-patch-explanation). The patch is based on draft work from Nobu [[10](#ref-nobu-eval-with-patch)] a few years ago on [Ruby Feature 12093](https://bugs.ruby-lang.org/issues/12093).

Ruby allows passing a binding to Kernel.eval, to arbitratily execute strings as ruby code in the context of a particular binding. Ruby also has the ability to precompile strings of Ruby code, but it does not have the ability to execute this within the context of a particular binding, it will be evaluated against the binding that it was compiled for.

This patch changes the call signature of eval, adding an optional single argument, where none are currently accepted. This doesn’t change the contract with existing callers, so all tests pass. However, there is a major flaw in the design.

## Why the patch doesn’t work

Ultimately, this patch has been rejected for good reason, but I figured I’d explain what the obstacle is in case someone can figure out a clever solution!

Take for example this test case:

def bind  
 a = 1  
 b = 2  
 binding  
end  
  
iseq = RubyVM::InstructionSequence.compile("a + b")  
val = iseq.eval(bind)

We will get an error!

NameError: undefined local variable or method `a'

This is because compiled source may contain undefined references which may be assumed to be method calls or local variables. In the ruby instruction sequence, it is assumed to be a method call:

== disasm: #<ISeq:<compiled>@<compiled>:1 (1,0)-(1,5)> (catch: FALSE)  
0000 putself ( 1)[Li]  
0001 opt\_send\_without\_block <callinfo!mid:a, argc:0, FCALL|VCALL|ARGS\_SIMPLE>, <callcache>  
0004 putself  
0005 opt\_send\_without\_block <callinfo!mid:b, argc:0, FCALL|VCALL|ARGS\_SIMPLE>, <callcache>  
0008 opt\_plus <callinfo!mid:+, argc:1, ARGS\_SIMPLE>, <callcache>  
0011 leave

In the other test case, where the variables are wrapped in a struct:

obj = Struct.new(:a, :b).new(1, 2)  
bind = obj.instance\_eval {binding}  
iseq = RubyVM::InstructionSequence.compile("a + b")  
val = iseq.eval(bind)

They are accessible because the struct is able receive the call, where the local variables in the binding object above wouldn’t.

If a way can be devised for local variables to be added to the binding as above but in a more elegant / transparent way, this approach could regain its efficacy.

## Patch Description

The rejected patch updated the signature and docs to:

/\*  
 \* call-seq:  
 \* iseq.eval -> obj  
 \*  
 \* Evaluates the instruction sequence and returns the result.  
 \*  
 \* RubyVM::InstructionSequence.compile("1 + 2").eval #=> 3  
 \*/  
static VALUE  
iseqw\_eval(VALUE self)  
{  
 rb\_secure(1);  
 return rb\_iseq\_eval(iseqw\_check(self));  
}

Note that this is based on the signature of the Kernel.eval method:

/\*  
 \* call-seq:  
 \* eval(string [, binding [, filename [,lineno]]]) -> obj  
 \*  
 \* Evaluates the Ruby expression(s) in <em>string</em>. If  
 \* <em>binding</em> is given, which must be a Binding object, the  
 \* evaluation is performed in its context. If the optional  
 \* <em>filename</em> and <em>lineno</em> parameters are present, they  
 \* will be used when reporting syntax errors.  
 \*  
 \* def get\_binding(str)  
 \* return binding  
 \* end  
 \* str = "hello"  
 \* eval "str + ' Fred'" #=> "hello Fred"  
 \* eval "str + ' Fred'", get\_binding("bye") #=> "bye Fred"  
 \*/

Where the:

* First argument string to Kernel.eval is not necessary in the iseq version, as it is implied as part of self that was used to compile the iseq.
* Second argument, binding, is optional. This becomes the first argument of iseq.eval, as reasoned above
* Third optional argument, filename, is specified when an iseq is created so is not needed
* Fourth optional argument, lineno, is also specified when an iseq is created so is not needed

To implement this new call signature, the definition of iseqw\_eval is updated to check the number of arguments.

/\*  
 \* Returns a human-readable string representation of this instruction  
 \* sequence, including the #label and #path.  
 \*/  
static VALUE  
iseqw\_inspect(VALUE self)  
{  
 const rb\_iseq\_t \*iseq = iseqw\_check(self);  
 const struct rb\_iseq\_constant\_body \*const body = iseq->body;  
 VALUE klass = rb\_class\_name(rb\_obj\_class(self));  
  
 if (!body->location.label) {  
 return rb\_sprintf("#<%"PRIsVALUE": uninitialized>", klass);  
 }  
 else {

If no arguments are specified, it does what it always did. If an argument is specified, it scans for the argument and uses it as the binding, for a new VM method rb\_iseq\_eval\_in\_scope :

VALUE  
rb\_iseq\_eval\_main(const rb\_iseq\_t \*iseq)  
{  
 rb\_execution\_context\_t \*ec = GET\_EC();  
 VALUE val;  
  
 vm\_set\_main\_stack(ec, iseq);  
 val = vm\_exec(ec, TRUE);  
 return val;  
}  
  
int  
rb\_vm\_control\_frame\_id\_and\_class(const rb\_control\_frame\_t \*cfp, ID \*idp, ID \*called\_idp, VALUE \*klassp)  
{  
 const rb\_callable\_method\_entry\_t \*me = rb\_vm\_frame\_method\_entry(cfp);

This definition is based on the approach used under the hood for Kernel.eval, when it calls eval\_string\_with\_scope:

static VALUE  
eval\_string\_with\_scope(VALUE scope, VALUE src, VALUE file, int line)  
{  
 rb\_execution\_context\_t \*ec = GET\_EC();  
 rb\_binding\_t \*bind = Check\_TypedStruct(scope, &ruby\_binding\_data\_type);  
 const rb\_iseq\_t \*iseq = eval\_make\_iseq(src, file, line, bind, &bind->block);  
 if (!iseq) {  
 rb\_exc\_raise(ec->errinfo);  
 }  
  
 vm\_set\_eval\_stack(ec, iseq, NULL, &bind->block);  
  
 /\* save new env \*/  
 if (iseq->body->local\_table\_size > 0) {  
 vm\_bind\_update\_env(scope, bind, vm\_make\_env\_object(ec, ec->cfp));  
 }  
  
 /\* kick \*/  
 return vm\_exec(ec, TRUE);  
}

[1] “Ruby 2.6 adds targetted tracing.” [Online]. Available: <https://bugs.ruby-lang.org/issues/15289>

[2] “Ruby 2.6 Method Tracing Docs.” [Online]. Available: <https://ruby-doc.org/core-2.6/TracePoint.html#method-i-enable>

[3] S. Hiroshi, N. Nakada, and K. Sasada, “Ruby 2.6 targetted tracing prelude docs.” [Online]. Available: <https://github.com/ruby/ruby/blame/master/prelude.rb#L140-L173>

[4] A. Wu, “Alan wu’s Github.” [Online]. Available: <https://github.com/XrXr>

[5] “Demo of targetted tracing by Alan Wu.” [Online]. Available: <https://gist.github.com/XrXr/f6cc2f1a4b6d3341e3d3fc749bcd5255>

[6] “Ruby Static Tracing Github.” [Online]. Available: <http://github.com/dalehamel/ruby-static-tracing>

[7] “Ruby Static Tracing Gem.” [Online]. Available: <https://rubygems.org/gems/ruby-static-tracing/>

[8] D. Hamel, “USDT Report Doc.” [Online]. Available: <https://bpf.sh/usdt-report-doc/index.html>

[9] “Make iseq eval consistent with Kernel eval.” [Online]. Available: <https://github.com/ruby/ruby/pull/2298>

[10] “Nobu’s original patch for iseq.eval\_with.” [Online]. Available: <https://github.com/ruby/ruby/pull/2298/commits/5d0c8c83d8fdea16c403abbc80c160ddddb92ef8>