Production Breakpoints

Dale Hamel

07/24/19 05:48:45 AM UTC

Contents

Ruby Production Breakpoints

This document is also available in epub and pdf format if you prefer.

Unmerged Patches

- Make iseq eval consistent with Kernel eval [1] is an upstream pull request that is assumed here. If it fails, it will fall back to eval'ing from source which is less performant. More details of this patch are below. The patch is based on draft work from Nobu [2] a few years ago on Ruby Feature 12093.

How this works

Ruby "production breakpoints" rewrite the source code of a method with the targeted lines to include a wrapper around those lines.

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.

When a breakpoint line is executed, we can use the Linux Kernel to interrupt our application and retrieve some data we've prepared for it.

Unless a breakpoint is both enabled and attached to by a debugger, it shouldn't change execution

Compared to ruby tracepoint API

Ruby now supports tracing only particular methods [3] instead of all methods, and looks to be aiming to add a similar sort of debugging support natively.

Presently the documentation for this feature is very sparse and it is very new [4] so more exploration into the benefits of both approaches is merited.

The approach taken here uses a metaprogramming external to the ruby VM, whereas the ruby tracing API uses code that is internal to Ruby.

One clear benefit of this approach over the ruby native approach is that attaching a debugger enables the tracing logic with this approach, whereas the built-in Ruby approach cannot detect this under its current implementation, and is an 'all or nothing' approach still - though that may suffice in some contexts.

Injecting breakpoints

To do this, we will need to actually rewrite Ruby functions to inject our tracing code around the selected line or lines. Ruby 2.6 + has built-in AST parsing, so we can use this to determine what needs to be redefined, in order to add our tracer.

The AST parsing will show us the scope of the lines that the user would like to trace, and will load the method scope of the lines, in order to inject the tracing support. This modified ruby code string can then be evaluated in the scope of an anonymous module, which is prepended to the parent of the method that has been redefined.

This will put it at the tip of the chain, and override the original copy of this method. Upon unprepending the module, the original definition should be what is evaluated by Ruby's runtime Polymorphic method message mapping.

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:

```
{
  "breakpoints": [
    {
      "type": "inspect",
      "source_file": "test/ruby_sources/config_target.rb",
      "start_line": 7,
      "end_line": 9,
      "trace_id": "config_file_test"
    }
  ]
}
```

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:

```
def handle(caller_binding, &block)
  return super(caller_binding, &block) unless
  ↪ @tracepoint.enabled?
  start_time = StaticTracing.nsec
  val = super(caller_binding, &block)
  duration = StaticTracing.nsec - start_time
  @tracepoint.fire(duration)
  return val
end
```

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.

```
def handle(caller_binding, &block)
  return super(caller_binding, &block) unless
  ↪ @tracepoint.enabled?
```

```
        val = super(caller_binding, &block)
        locals = caller_binding.local_variables
        locals.delete(:local_bind) # we define this, so we'll get
↪   rid of it
        vals = locals.map { |v| [v,
↪   caller_binding.local_variable_get(v) ]}.to_h
        @tracepoint.fire(vals.inspect)
        return val
      end
```

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.

```
      def handle(caller_binding, &block)
        return super(caller_binding, &block) unless
        ↪  @tracepoint.enabled?
        val = super(caller_binding, &block)
        @tracepoint.fire(val.inspect)
        return val
      end
```

Internals

Loading breakpoints

This gem leverages the ruby-static-tracing [5] gem [6] 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 [7]

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

The 'unmixer' gem hooks into the ruby internal header API, and provides a back-door into the RubyVM source code to 'unappend' 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.

```
def install
  @injector_module =
  ↪ build_redefined_definition_module(@node)
```

```
      @ns.prepend(@injector_module)
    end

    # FIXME saftey if already uninstalled
    def uninstall
      @ns.instance_eval{ unprepend(@injector_module) }
      @injector_module = nil
    end
```

Dynamically redefined Method

note this description is of early work, a branch is under development that will default to using precompiled source and only fall-back to eval.

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:

```
    def build_redefined_definition_module(node)

      # This is the metaprogramming to inject our breakpoint
      ↪ handler around the original source code
      handler = "local_bind=binding;
↪ ProductionBreakpoints.installed_breakpoints[:#{@trace_id}].handle(local_bind)"

      # This is needed to keep the execution of the remaining
      ↪ lines of the method within the same binding
      finisher =
↪ "ProductionBreakpoints.installed_breakpoints[:#{@trace_id}].finish(local_bind)"

      # This injects our handler and finisher blocks into the
      ↪ original source code, treating the code
      # in between as string literals to be evaluated
      injected =
↪ @parser.inject_metaprogramming_handlers(handler, finisher,
                                          node.first_lineno,
↪ node.last_lineno, @start_line, @end_line)
      #ProductionBreakpoints.logger.debug(injected)
```

```
        Module.new { module_eval{ eval(injected); eval('def
        ↪  production_breakpoint_enabled?; true; end;') } }
      end
```

And the outcome looks lomething like this:

```
# def some_method
# local_bind=binding;
↪  ProductionBreakpoints.installed_breakpoints[:test_breakpoint_install].handle(local_bind)
↪  do
# <<-EOS
#       a = 1
#       sleep 0.5
#       b = a + 1
# EOS
# end
#
↪  ProductionBreakpoints.installed_breakpoints[:test_breakpoint_install].finish(local_bind)
↪  do
# <<-EOS
# EOS
# end
#     end
```

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

```
      # Allows for specific handling of the selected lines
      def handle(caller_binding)
        eval(yield, caller_binding)
      end

      # Execute remaining lines of method in the same binding
      def finish(caller_binding)
        eval(yield, caller_binding)
      end
```

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.

Pre-compiled sources

The source segments above can be compiled using `RubyVM::InstructionSequence`, and using [1], the precompiled code can be evaluated in the context of the binding, rather than having to parse and compile it each time that it needs to be executed. This cuts out a lot of unecessary and redundant processing that `Kernel.eval` would do, which would slow down code that is being traced when it doesn't need to.

If the precompiled sources cannot be executed, they will fallback to evaluating from source

Ruby iseq eval patch explanation

This work depends on an unmerged patch [1] to Ruby in order to be feasible to run in Production environments.

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 pre-compile 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.

This updates the signature and docs to:

```
/*
 *  call-seq:
 *     iseq.eval([binding]) -> obj
 *
 *  Evaluates the instruction sequence and returns the result.
 *
 *      RubyVM::InstructionSequence.compile("1 + 2").eval #=> 3
 *
 *  If <em>binding</em> is given, which must be a Binding object,
↪  the
 *  evaluation is performed in its context.
 *
 *      obj = Struct.new(:a, :b).new(1, 2)
 *      bind = obj.instance_eval {binding}
 *      RubyVM::InstructionSequence.compile("a + b").eval(bind)
↪  #=> 3
 */
```

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.

```
static VALUE
iseqw_eval(int argc, const VALUE *argv, VALUE self)
{
    VALUE scope;

    if (argc == 0) {
        rb_secure(1);
        return rb_iseq_eval(iseqw_check(self));
    }
    else {
        rb_scan_args(argc, argv, "01", &scope);
```

```
        rb_secure(1);
        return rb_iseq_eval_in_scope(iseqw_check(self), scope);
    }
}
```

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_in_scope(const rb_iseq_t *iseq, VALUE scope)
{
    rb_execution_context_t *ec = GET_EC();
    rb_binding_t *bind = Check_TypedStruct(scope,
↪   &ruby_binding_data_type);

    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));
    }

    return vm_exec(ec, TRUE);
}
```

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

[2] “Nobu’s original patch for iseq.eval_with.” [Online]. Available: https://github.com/ruby/ruby/pull/2298/commits/5d0c8c83d8fdea16c403abbc80c160dddb92ef8

[3] “Ruby 2.6 adds method tracing.” [Online]. Available: https://bugs.ruby-lang.org/issues/15289

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

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

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

[7] “USDT Report Doc.” [Online]. Available: https://bpf.sh/usdt-report-doc/index.html