debug.gem:
Ruby's new debug functionality

Koichi Sasada
<ko1@cookpad.com>

Hello! Thank you for listening this talk.
I'm Koichi Sasada from Cookpad.
This year, I made a brand-new debugger named "debug gem", so I want to introduce this debug gem in this talk.

About this talk

- Introduce “debug.gem” https://github.com/ruby/debug
  - Newly created debugger for Ruby 2.6 and later
  - Will be bundled with Ruby 3.1 (Dec/2021)
- Demonstrate “debug.gem”
  - Basic usage instructions
  - Advanced features
- The presentation slides with the talk script is available at here: https://www.atdot.net/~ko1/activities/

So that this talk introduce “debug.gem”.

“debug.gem” is already released and you can use this debugger with Ruby 2.6 or later.
This debugger will be shipped with Ruby 3.1 which will be released in December this year.

Ruby 3.1 development branch already merged debug gem so you can try Ruby 3.1 and this new debugger.
New debugger has many novel features that were not previously available in Ruby world, so I hope you enjoy new features.

This talk demonstrate this new debugger, the basic usage and advanced features.
The presentation slides with the talk script is available at this URL.

About Koichi Sasada

- Ruby interpreter developer employed by Cookpad Inc. (2017-) with @mame
  - YARV (Ruby 1.9-)
  - Generational/Incremental GC (Ruby 2.1-)
  - Ractor (Ruby 3.0-)
  - …
- Ruby Association Director (2012-)

Let's introduce my self.
I'm Koichi Sasada, one of Ruby interpreter developers.
Usually I develop Ruby internals, such as virtual machine and garbage collector and so on.
Last year I was working on Ractor implementation for Ruby 3.0.

This year, mainly I spend my time to make this new debugger.

What is a debugger?

- A tool to help debugging
  - To investigate the cause of problems
  - To know the program live state
  - To understand the program
- Basic features
  - CONTROL execution
    - STOP at breakpoints
    - STEP forward to the next line
    - …
  - QUERY program status

4

Before the introduce new debugger, I summarize what is the debugger.

The main purpose of debugger is to help debugging of your applications.
When we get a bug, we need to investigate the cause of problems.

If we got a bug, at first, we check the backtrace to know where the issue comes from.
Also usually, we use “printing” the program state with “p” or “pp” method in Ruby.
Sometimes you may use “binding pry” or “binding irb” methods to use REPL, Read Eval Print Loop console on the specific binding.

Another purpose is to use the debugger to understand the program.
To understand the program, running the program is one of the best ways.
When you want to know the program internals, you also need to use print methods or REPL console.

To help such cases, debugger provides useful features.
Debugger enables to control the execution, for example specifying the

breakpoints we want to stop, step forward only one line, and so on. Also, debugger helps to show the program state such as local variables and so on.

Ruby's existing debuggers

- lib/debug.rb
  - `ruby -r debug script.rb`
  - Standard library, but maybe nobody uses it
- byebug
  - `byebug script.rb`
- debase / ruby-debug-ide
  - Used by IDE (rubymine, vscode, …)

5

In the Ruby world, we already has several debuggers.

lib/debug.rb is standard library which is available from Ruby 1.0.
However maybe nobody uses it and not maintained well.

I think byebug is most popular debugger for Ruby in recent days.
debase or ruby-debug-ide is also popular because they are backend of rubymin and vscode IDEs.

Why create yet another debugger?

- Performance
  - Existing debuggers slow with breakpoints
  - Recent TracePoint API support line-specific
- Native support for remote execution and IDE
- Native support for Ractors
- (and I like to make this kind of tools)

6

So, we already has Ruby's debuggers.
But I decided to make new debugger because of several reasons.

One reason is the performance of existing debuggers.
When we use the breakpoints in a program, existing debuggers can become slower to check breakpoints.
Recent Ruby introduces new TracePoint features to solve the slow-down, so we can improve the performance of debugging.
In other words, we don't need to hesitate to use the debuggers on development.

Also, I want to provide IDE integration.
Sometimes REPL debugger console is useful, but sometimes it is hard to use it.

For example, we need to use "break" command with file and line, or to write "byebug" method in a application to specify the breakpoints. But I want to specify breakpoints by clicking the editors source code.
Another example, if we print the nested data structure, it can be huge

output.
I don't want to see a thousand lines of output from nested data structures.
I want to expand the output by clicking the output like JavaScript console.

Usually, IDEs provide such interfaces, and it is more natural to utilize these features.

Another reason is Ractor. Ruby 3.0 introduced new Ractor mechanism which isolates the object spaces and existing debuggers don't support it. Unfortunately, current debug gem dose not support Ractors yet, but it is one motivation.

One more little but biggest motivation is I like to make this kind of tool. It's technically difficult a bit and it's useful to have.

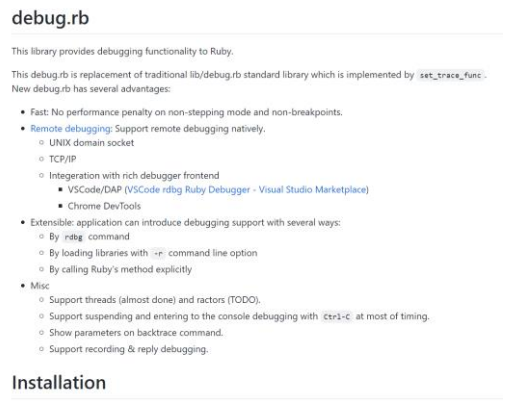
Introduction of “debug.gem”

https://github.com/ruby/debug

7

OK. Let’s start to introduce the debug gem.

All information are explained in https://github.com/ruby/debug



debug.rb

This library provides debugging functionality to Ruby.

This debug.rb is replacement of traditional lib/debug.rb standard library which is implemented by `set_trace_func`.
New debug.rb has several advantages:

- Fast: No performance penalty on non-stepping mode and non-breakpoints.
- Remote debugging: Support remote debugging natively.
  - UNIX domain socket
  - TCP/IP
  - Integeration with rich debugger frontend
    - VSCode/DAP (VSCode rdbg Ruby Debugger - Visual Studio Marketplace)
    - Chrome DevTools
- Extensible: application can introduce debugging support with several ways:
  - By `rdbg` command
  - By loading libraries with `-r` command line option
  - By calling Ruby's method explicitly
- Misc
  - Support threads (almost done) and ractors (TODO).
  - Support suspending and entering to the console debugging with `Ctrl-C` at most of timing.
  - Show parameters on backtrace command.
  - Support recording & reply debugging.

Installation

8

This talk doesn not explain how to use new debugger but explains what kind of features new debugger provides.
Please check out the documentation on the GitHub to know HOW TO USE.
We wrote everything on a README in about a thousand lines.

debug.gem

- Created from scratch (2021 Feb~)
- Supports Ruby 2.6 and later
  - Utilize recent introduced APIs
- Ruby 3.1 (2021/Dec) will be shipped with debug.gem
  - Replacement with old lib/debug.rb
- Like other libraries (lib/debug.rb, byebug, gdb, lldb, …)
  debug.gem provides REPL to execute debug commands

9

“debug gem” was created from scratch this year.
Gem is already released so you can use it by “gem install debug”.
It supports Ruby 2.6 and later because it uses recent introduced API.
Ruby 3.1 released soon will be shipped with this debug.gem to replace old lib/debug.rb.
Like other debuggers such as byebug, gdb and so on, debug.gem provides REPL which accepts debug commands to investigate the bugs interactively.

Use debug.gem

1. Use “rdbg” command
   - `rdbg target.rb`
   - `rdbg -c -- bin/rails`
   - `rdbg -c -- bundle exec rake`
2. Load “debug.gem” in your application
   - `require "debug"` (or “debug/…”, see doc)
   - `gem 'debug'` in Gemfile (and Bundler.require)
3. Use with IDE
   - (VSCode) .vscode/launch.json (ruby-rdbg extension will make) and push “Start debugging” button

10

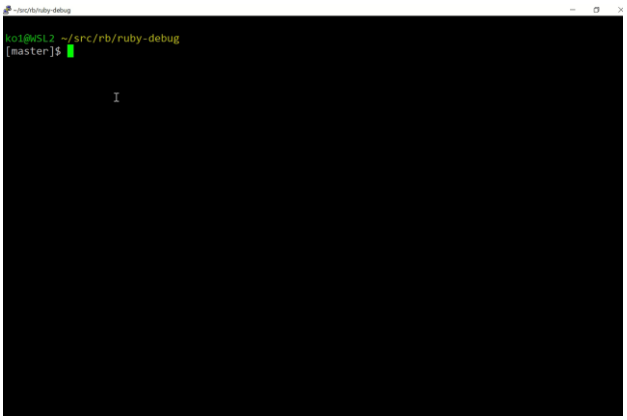
To use ‘debug.gem’, mainly there are 3 ways.

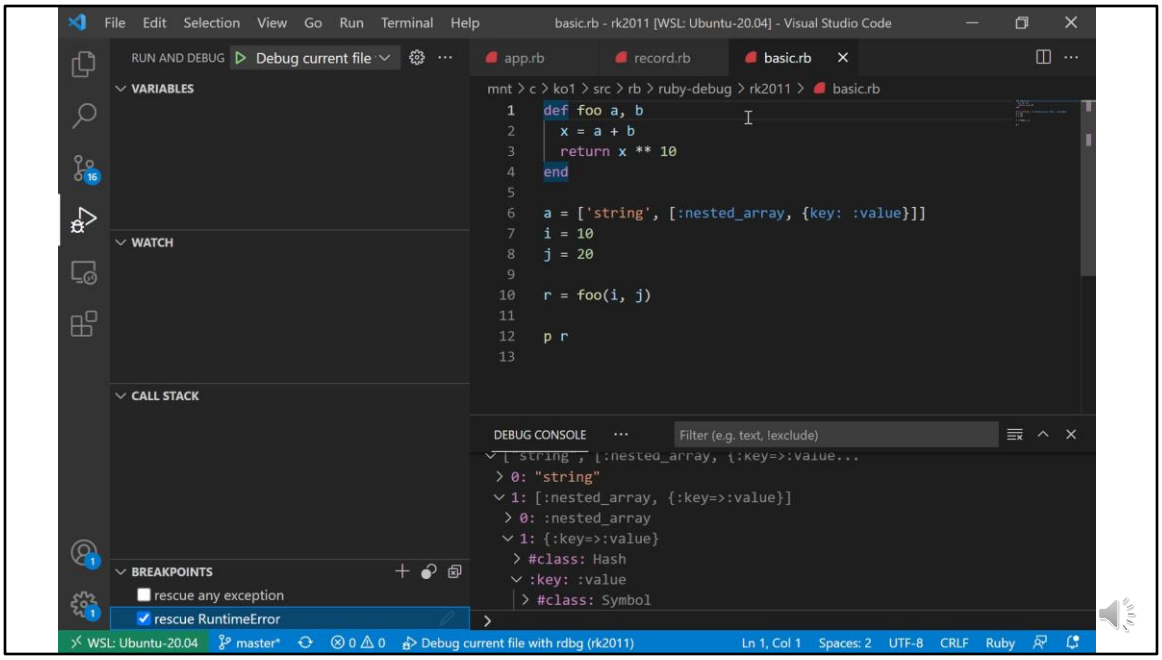
The first way is use “rdbg” command like byebug or gdb commands.
You can specify ruby script or execution command with “-c” option.

The second way is rewrite your application to require ‘debug’ or rewrite your Gemfile.
I think Rails developers like this way.

The third way is to use IDE. For example, on the VSCode, you need to setup the lancuh.json file in your workspace.
Fortunately, ruby-rdbg extension made a default setting so you don’t see the details.
After that, you only need to push the “Start debugging” button to start debugging with debug.gem.

Demo: Basic usage





This demonstration shows debug gem and VSCode integration.
You can start debugging by menu, you can specify what kind of program you want to run.
And the program runs with the debugger.
In this case, there are no breakpoints, so the program terminates.

Let's specify the breakpoint with typing F9 key on VSCode, and run debugger again, you can see the program stops at the breakpoint.
As you can see, there are local variables view and you can expand the nested data structure by clicking it.
You can control the execution by pushing the buttons.

Basic features

- Control the program execution
  - Set breakpoints
  - Step execution (step-in/over/out)
- Query the program status
  - See the source code at breakpoint
  - See the backtrace
  - Select the frame in backtrace
  - Access to variables of the specific frames
  - Evaluate an expression on the specific frame

13

As you can see, debug gem provides basic features which are provided by other debuggers.

Set a breakpoint

- Use “break” command at the beginning
  - `break 10` # break at 10 line on current file
  - `break foo.rb:10` # break at the location
  - `break MyClass#my_method` # break at the method
  - `catch FooException` # break at FooException is raised
  - `break … if foo == bar` # break if foo == bar
- Write “`binding.break`” line in your program
  - You can insert it like “`binding.irb`”
  - “`binding.b`” for short and “`debugger`” like JavaScript
- Use IDEs/editors breakpoint support

14 

To specify the breakpoints, there are three ways.

One is to use “break” command on the debugger’s console.
You can specify files, lines, method names, exceptions.
Also, you can specify conditions you want to break the execution.

Another way is to write “binding.break” in your application like “binding.pry” or “binding.irb”.
There is shorter “binding.b” or simply “debugger” method like JavaScript.
If you can modify the source code, it is easy way to specify the breakpoints before execute the program.

The last way is use IDE’s breakpoint feature. Now we only supports VSCode but we can increase the support platforms like vim editors.

Set a breakpoint (cont.)

- Use “break” command at the beginning (and IDE)
  - Do not need to modify the source code
  - Cooperation with IDE/Editor (e.g. set it with F9 on VSCode)
- Write “`binding.break`” method in your program
  - Straight forward for some Ruby users

15

Using “break” command or using IDEs support we don’t need to modify the source code.
I think it is easiest way to use IDEs supports to do it.

Writing “binding.break” method in your application is well-known style like “binding.pry”.
You can control breakpoints with your application code.

Control debugger from the program by `binding.break do: expr`

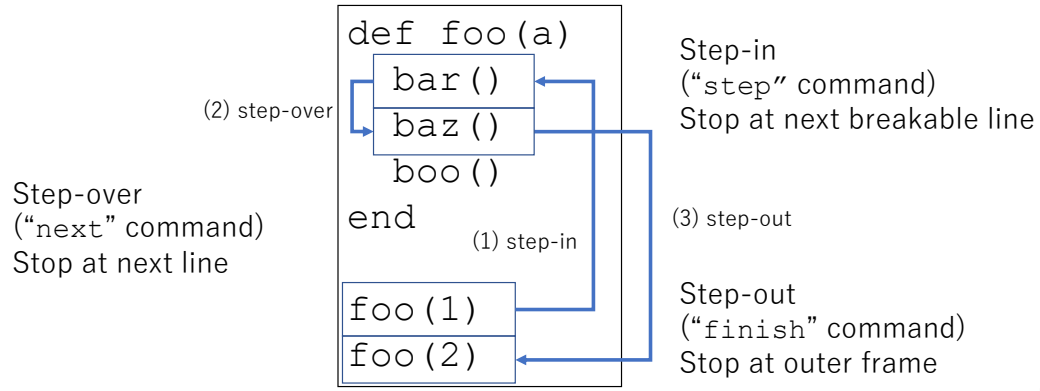
```
# enable "trace line" feature while bar()
def foo
  binding.break do: 'trace line'
  bar()
  biding.break do: 'trace off line'
end
```

16

Another interesting idea to use “binding.break” method is using “do:” keyword.
If “do” keyword is given, it doesn’t stop the execution, but execute the debug command.
In this case, it enables “trace” feature before “bar()” method and disable it after “bar()” method.
You can see the trace information for “bar()” method.

In general, a debug target program can not communicate with an underlying debugger.
This feature achieves it and you can use debugger’s feature in your application.

Step execution
Step-in, Step-over, Step-out



debug.gem provides normal execution control features, step-in, step-over and step-out.
step-in, by "step" command stops after next breakable line, so use "step-in" stops in foo's line.
steo-over, by "next" command stops at next line. So it doesn't stop in bar() method.
step-out, by "finish" command stops when the scope is finished. so it stops just after returning the "foo()" method.

Access to the local variables in the specific frame

- See the backtrace with “`backtrace`”
- Select the frame
  - “`frame <num>`”
  - “`up`” / “`down`” to select upper/lower
- Access to the frame local variables
  - “`outline`” command and “`info`” command for overview
  - “`p <expr>`” and “`pp <expr>`”

```
(rdbg) p a ** 10    # command
=> 25937424601
```

```
[1, 6] in target.rb
    1|
    2| def foo(a) = bar(a+1)
=>  3| def bar(a) = a * 2
    4| foo 10
    5|
    6| __END__
=>#0    Object#bar(a=11) at target.rb:3 #=> 22
  #1    Object#foo(a=10) at target.rb:2
  # and 1 frames (use `bt' command for all frames)
```

```
(rdbg) outline    # command
Object.methods: inspect  to_s
locals: a
```

```
(rdbg) info    # c
%self = main
%return = 22
a = 11
```


When the program stops at the breakpoint, you can see the backtrace with given parameters like that.
With binding.irb, you can see only the specified binding.
On the other hands, with the debugger, you can access any bindings in the backtrace by specifying with “frame” command.
And you can check the variables with “info” command or “outline” command like that. “outline” command is same as pry’s ls command.
Of course, you can evaluate the Ruby expression in the debug console.

Advanced features

- [demo] Remote debugging
- [demo] VSCode/Chrome browser seamless integration
- [demo] Postmortem debugging
- [demo] Record and replay debugging
- Event tracing
- Multi-process debugging
- Pause with “Ctrl-C” or when attaching the debugger

19

I introduced basic features which are supported by many other debuggers.
Next, I want to introduce advanced features debug.gem provides.

Today I’ll show demonstarations about 4 features.

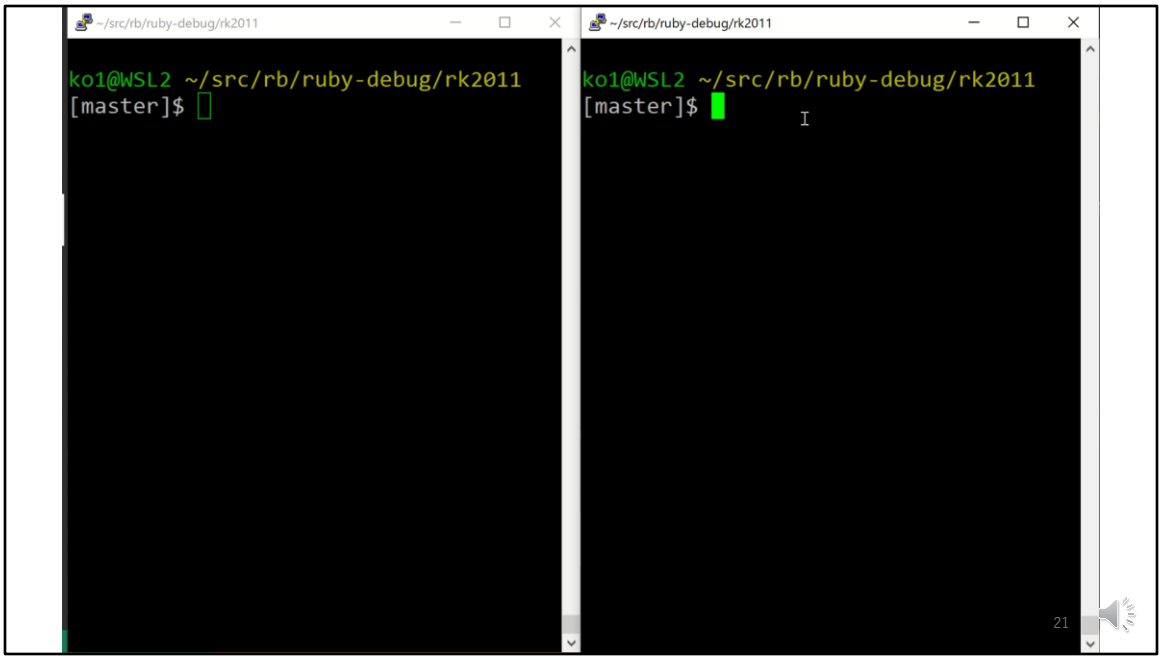
Demo: Remote debugging
Connect over network

- Easy to open remote debug port and attach
  - `rdbg --open script.rb` (or `rdbg -O`)
    - Run program with opening debug port
    - `require 'debug/open' # in script`
  - `rdbg -attach` (or `rdbg -A`)
    - Access to debug port
- Debug no TTY attached processes
  - Daemon processes
  - Redirecting by shell's pipe
- Query the process status like sigdump but more details

20

At first, debug.gem supports remote debugging feature.
You can open debug port by using "rdbg" command with "open" option.
After that, you can attach to the port with "rdbg --attach" command, and the attached program will pause the execution.
TCP/IP and UNIX domain socket are supported to communicate.

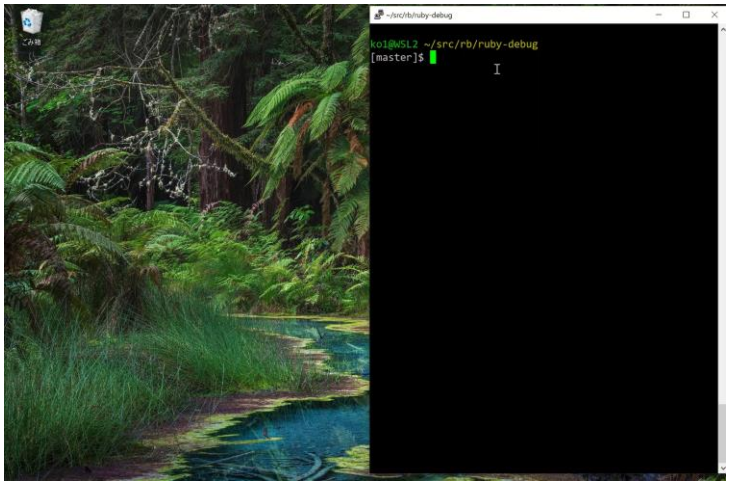
I think it helps to debug no TTY processes like daemon process, redirected processes with shell's pipe and remote machine processes.
If you open the debug port, anytime you can attach to the processes and check the program state so it will help the development.



Let's demonstrate the remote debugging.

Start the program with "-O" option on "rdbg" command, and debug port is opened and the program stops at the beginning.
On the next terminal, start "rdbg" command with "-A" option, it attached to the opening debug port and you can see the debug console.
You can control the target programs with any debug commands remotely.

Demo: Seamless integration with VSCode/Chrome browser

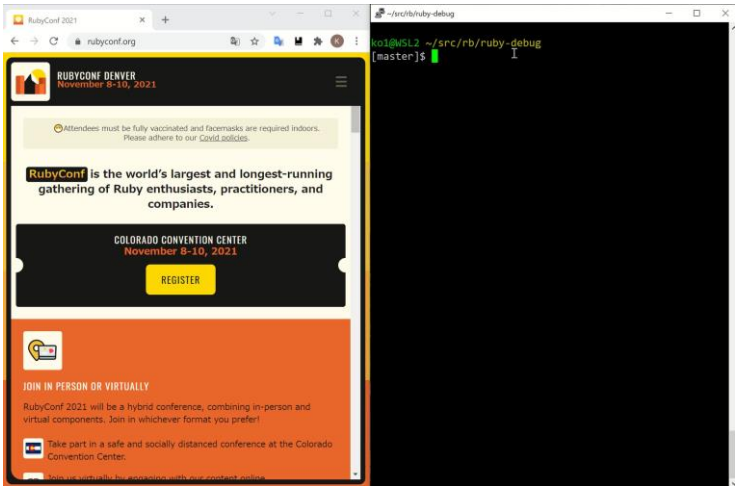


22

Next advanced topic is integration with VSCode and Chrome browser.

If you start the debugging with the debug console, but you want to brows it on the VSCode, you can open the VSCode with “open vscode” command. Like that, wait a seconds, VSCode launched automatically, and debugging can be continued on VSCode.

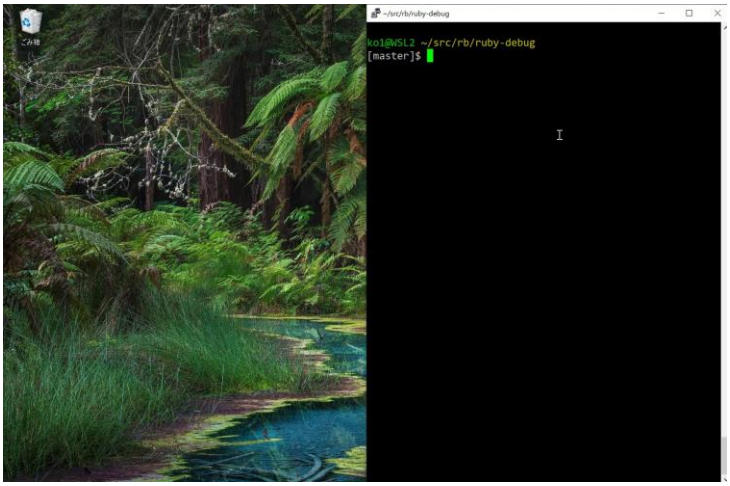
Demo: Seamless integration with VSCode/Chrome browser



23

You can also use Chrome browser as a debugger frontend.
Same as “open vscode” command, you can use “open chrome” and we can see the URL.
Put this URL into the chrome browser, and you can invoke the Chrome’s debug frontend and continue the debugging.

Demo: Start VSCode for debugger frontend



24

You can specify vscode or chrome for the debugger frontend at first with “open” option.
If you are not using VSCode, but you want to use VSCode as a debugger front end, this feature will help you.

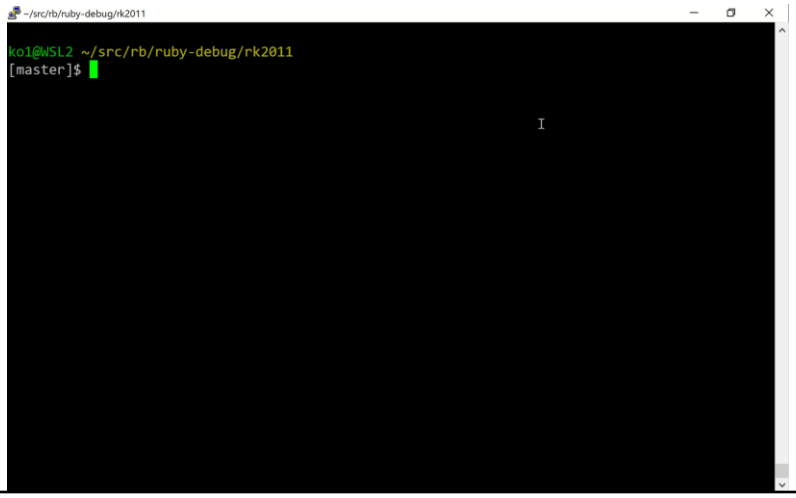
Demo: Postmortem debugging
Debug dead Ruby process

25

Next example is postmortem debugging.

This program stops because raising the ZeroDivideError.
But we can enables postmortem feature and you can enter the debug console when the process terminates with the exception.
You can see the variables at the exception is raised.

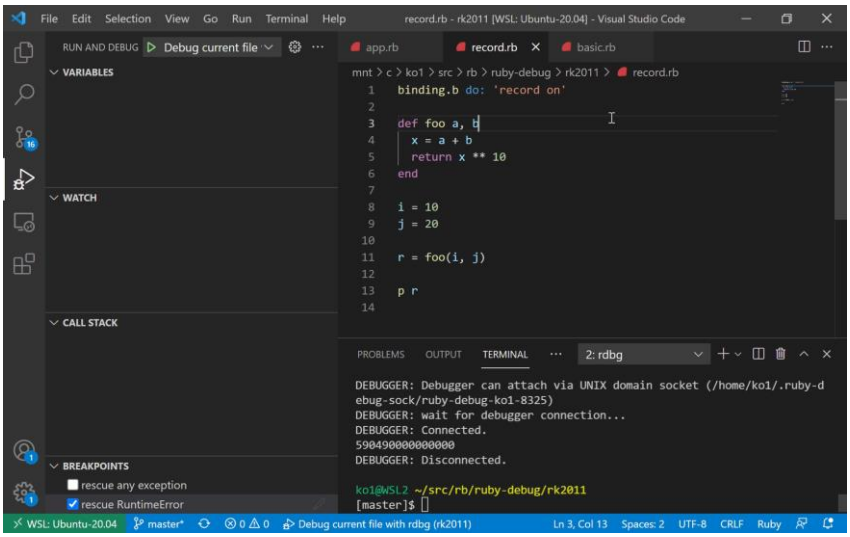
Demo: Record and replay debugging Backward stepping execution



The last advanced feature today I want to introduce is record and replay debugging.

If we enables this feature, we can record the execution information and step back the execution by “step back” command.
This feature helps if you want to check the last status before the breakpoint.

This feature is not optimized so it consumes time and memory and it is not feasible to use it on a entire execution, but we can use it with some limited lines.



27

You can use this record and reply debugging with VSCode.

Performance

```
def fib n
  if n < 0
    raise # breakpoint
  elsif n<2
    n
  else
    fib(n-1)+fib(n-2)
  end
end

require 'benchmark'
Benchmark.bm{|x|
  x.report{ fib(35) }
}
```

	Without breakpoint	With breakpoint
ruby	0.93	N/A
rdbg (debug.gem)	0.92 (**x0.98**)	0.92 (**x0.98**)
byebug	1.23 (x1.32)	75.15 (x80.80)
RubyMine	0.97 (x1.04)	22.66 (x24.36)
old lib/debug.rb	221.88 (x238.58)	285.99 (x307.51)

Execution time in sec (ratio with ruby result (smaller is better))

ruby 3.0.1p64
rdbg 1.0.0.rc2
byebug 11.1.3
RubyMine 2021.2.1 w/ debase 0.2.5.beta2

Intel(R) Core(TM) i7-10810U CPU, Windows 10, WSL2

28

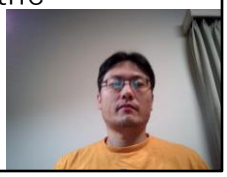
At the start of this presentation, I talked that one of the motivations is performance.
This figure shows how this program becomes slower with debugger when we set the breakpoints.

With byebug, 80 times slower than normal Ruby program. RubyMine also make it 24 times slower.
Old lib/debug.rb slows it about 300 times.

However, as you can see, with debug.gem we don't see any performance penalty on this case.

Acknowledgements

- Naoto Ono san (@ono-max) implements test-frameworks for the debugger and Chrome browser support. The part of works were done in GSoC project.
- Stan Lo san (@st0012) submits tremendous patches to improve the debugger usability such as coloring and so on based on his debugger trials. Also, he makes many tests for the debugger.
- Ruby committers helps me to design and implement the debugger



At last, let me shows acknowledgements.

Ono-san helps me to make test-framework and chrome browser integration. The test framework is achievements of the Google Summer of Coe.
Stan Lo-san submits many patches to improve the debug.gem. He suggest many useful ideas from the Rails programmer.
And many people helps us.

Thank you so much.

Conclusion

- “debug.gem” is newly created Ruby debugger from scratch
  - Faster.
  - Modern UI.
  - Many useful features.
- “`gem install debug`” now!
  - And give us your feedback.
  - I love to introduce the debugger on your meetup, please contact me.
- Ractor supports is not available, now working on.

Conclusion.

“debug.gem” is newly created Ruby debugger from scratch. It is fast, it has modern UI and it provides many features.
You can use now by installing this gem.
This gem is not matured so your feedback will help us.
If you need an advice or requests, feel free to contact us.

Thank you for your listening!

debug.gem:
Ruby's new debug functionality

Koichi Sasada
<ko1@cookpad.com>

cookpad