README
README
Last Update: Sat Nov 29 22:47:07 -0700 2008

ruby-prof

Overview

ruby-prof is a fast code profiler for Ruby. Its features include:

- Speed - it is a C extension and therefore many times faster than the standard Ruby profiler.
- Modes - Ruby prof can measure a number of different parameters, including

```
call times, memory usage and object allocations.
```

- Reports - can generate text and cross-referenced html reports
  - Flat Profiles - similar to the reports generated by the standard Ruby profiler
  - Graph profiles - similar to GProf, these show how long a method runs, which methods call it and which methods it calls.
  - Call tree profiles - outputs results in the calltree format suitable for the KCacheGrind profiling tool.
- Threads - supports profiling multiple threads simultaneously
- Recursive calls - supports profiling recursive method calls

Requirements

ruby-prof requires Ruby 1.8.4 or higher.

If you are running Linux or Unix you'll need a C compiler so the extension can be compiled when it is installed.

If you are running Windows, then install the Windows specific RubyGem which includes an already built extension.

Install

The easiest way to install ruby-prof is by using Ruby Gems. To install:

```
gem install ruby-prof
```

If you are running Windows, make sure to install the Win32 RubyGem which includes a pre-built binary. Due to a bug in ruby-gems, you cannot install the gem to a path that contains spaces (see rubyforge.org/tracker/?func=detail&aid=23003&group_id=126&atid=577).

ruby-prof is also available as a tarred gzip archive and zip archive.

Usage

There are three ways of running ruby-prof.

ruby-prof executable

The first is to use ruby-prof to run the Ruby program you want to profile. For more information refer to the ruby-prof documentation.

ruby-prof API

The second way is to use the ruby-prof API to profile particular segments of code.

```
require 'ruby-prof'

# Profile the code
RubyProf.start
...
[code to profile]
...
result = RubyProf.stop

# Print a flat profile to text
printer = RubyProf::FlatPrinter.new(result)
printer.print(STDOUT, 0)
```

Alternatively, you can use a block to tell ruby-prof what to profile:

```
require 'ruby-prof'

# Profile the code
result = RubyProf.profile do
  ...
  [code to profile]
  ...
end

# Print a graph profile to text
printer = RubyProf::GraphPrinter.new(result)
printer.print(STDOUT, 0)
```

Starting with the 0.6.1 release, ruby-prof also supports pausing and resuming profiling runs.

```
require 'ruby-prof'

# Profile the code
RubyProf.start
[code to profile]
RubyProf.pause
[other code]
RubyProf.resume
[code to profile]
result = RubyProf.stop
```

Note that resume will automatically call start if a profiling run has not yet started. In addition, resume can also take a block:

```
require 'ruby-prof'

# Profile the code
RubyProf.resume do
  [code to profile]
end

data = RubyProf.stop
```

With this usage, resume will automatically call pause at the end of the block.

require unprof

The third way of using ruby-prof is by requiring unprof.rb:

```
require 'unprof'
```

This will start profiling immediately and will output the results using a flat profile report.

This method is provided for backwards compatibility. Using ruby-prof provides more flexibility.

Profiling Tests

Starting with the 0.6.1 release, ruby-prof supports profiling tests cases written using Ruby's built-in unit test framework (ie, test derived from Test::Unit::TestCase). To enable profiling simply add the following line of code to your test class:

```
include RubyProf::Test
```

Each test method is profiled separately. ruby-prof will run each test method once as a warmup and then ten additional times to gather profile data. Note that the profile data will **not** include the class's setup or teardown methods.

Separate reports are generated for each method and saved, by default, in the test process's working directory. To change this, or other profiling options, modify your test class's PROFILE_OPTIONS hash table. To globally change test profiling options, modify RubyProf::Test::PROFILE_OPTIONS.

Profiling Rails

To profile a Rails application it is vital to run it using production like settings (cache classes, cache view lookups, etc.). Otherwise, Rail's dependency loading code will overwhelm any time spent in the application itself (our tests show that Rails dependency loading causes a roughly 6x slowdown). The best way to do this is create a new Rails environment, profile.rb.

So to profile Rails:

1. Create a new profile.rb environment - or simply copy the example file in ruby-prof/rails/environment/profile.rb
2. Copy the file:

   ```
   ruby-prof/rails/profile_test_helper.rb
   ```

   To:

   ```
   your_rails_app/test/profile_test_helper.rb
   ```

3. Create a new test directory for profiling:

   ```
   your_rails_app/test/profile
   ```

4. Write unit, functional or integration tests specifically designed to profile some part of your Rails application. At the top of each test, replace this line:

   ```
   require File.dirname(__FILE__) + '/../test_helper'
   ```

   With:

   ```
   require File.dirname(__FILE__) + '/../profile_test_helper'
   ```

   For example:

   require File.dirname(*FILE*) + '/../profile_test_helper'

```
class ExampleTest < Test::Unit::TestCase

  include RubyProf::Test
  fixtures ....

  def test_stuff
    puts "Test method"
  end

end
```

5. Now run your tests. Results will be written to:

```
your_rails_app/tmp/profile
```

Reports

ruby-prof can generate a number of different reports:

- Flat Reports
- Graph Reports
- HTML Graph Reports
- Call graphs

Flat profiles show the overall time spent in each method. They are a good of quickly identifying which methods take the most time. An example of a flat profile and an explanation can be found in examples/flat.txt.

Graph profiles also show the overall time spent in each method. In addition, they also show which methods call the current method and which methods its calls. Thus they are good for understanding how methods gets called and provide insight into the flow of your program. An example text graph profile is located at examples/graph.txt.

HTML Graph profiles are the same as graph profiles, except output is generated in hyper-linked HTML. Since graph profiles can be quite large, the embedded links make it much easier to navigate the results. An example html graph profile is located at examples/graph.html.

HTML Graph profiles are the same as graph profiles, except output is generated in hyper-linked HTML. Since graph profiles can be quite large, the embedded links make it much easier to navigate the results. An example html graph profile is located at examples/graph.html.

Call graphs output results in the calltree profile format which is used by KCachegrind. Call graph support was generously donated by Carl Shimer. More information about the format can be found at the KCachegrind site.

Printers

Reports are created by printers. Supported printers include:

- RubyProf::FlatPrinter - Creates a flat report in text format
- RubyProf::GraphPrinter - Creates a call graph report in text format
- RubyProf::GraphHtmlPrinter - Creates a call graph report in HTML (separate files per thread)
- RubyProf::CallTreePrinter - Creates a call tree report compatible with KCachegrind.

To use a printer:

```
result = RubyProf.end
printer = RubyProf::GraphPrinter.new(result)
printer.print(STDOUT, 0)
```

The first parameter is any writable IO object such as STDOUT or a file. The second parameter, which has a default value of 0, specifies the minimum percentage a method must take to be printed. Percentages should be specified as integers in the range 0 to 100. For more information please see the documentation for the different printers.

Measurements

Depending on the mode and platform, ruby-prof can measure various aspects of a Ruby program. Supported measurements include:

- process time (RubyProf::PROCESS_TIME)
- wall time (RubyProf::WALL_TIME)
- cpu time (RubyProf::CPU_TIME)
- object allocations (RubyProf::ALLOCATIONS)
- memory usage (RubyProf::MEMORY)
- garbage collections runs (RubyProf::GC_RUNS)
- garbage collection time (RubyProf::GC_TIME)

Process time measures the time used by a process between any two moments. It is unaffected by other processes concurrently running on the system. Note that Windows does not support measuring process times - therefore, all measurements on Windows use wall time.

Wall time measures the real-world time elapsed between any two moments. If there are other processes concurrently running on the system that use significant CPU or disk time during a profiling run then the reported results will be too large.

CPU time uses the CPU clock counter to measure time. The returned values are dependent on the correctly setting the CPU's frequency. This mode is only supported on Pentium or PowerPC platforms.

Object allocation reports show how many objects each method in a program allocates. This support was added by Sylvain Joyeux and requires a patched Ruby interpreter. For more information and the patch, please see: rubyforge.org/tracker/index.php?func=detail&aid=11497&group_id=426&atid=1700

Memory usage reports show how much memory each method in a program uses. This support was added by Alexander Dymo and requires a patched Ruby interpreter. For more information, see: rubyforge.org/tracker/index.php?func=detail&aid=17676&group_id=1814&atid=7062

Garbage collection runs report how many times Ruby's garbage collector is invoked during a profiling session. This support was added by Jeremy Kemper and requires a patched Ruby interpreter. For more information, see: rubyforge.org/tracker/index.php?func=detail&aid=17676&group_id=1814&atid=7062

Garbage collection time reports how much time is spent in Ruby's garbage collector during a profiling session. This support was added by Jeremy Kemper and requires a patched Ruby interpreter. For more information, see: rubyforge.org/tracker/index.php?func=detail&aid=17676&group_id=1814&atid=7062

To set the measurement:

- RubyProf.measure_mode = RubyProf::PROCESS_TIME
- RubyProf.measure_mode = RubyProf::WALL_TIME
- RubyProf.measure_mode = RubyProf::CPU_TIME
- RubyProf.measure_mode = RubyProf::ALLOCATIONS
- RubyProf.measure_mode = RubyProf::MEMORY
- RubyProf.measure_mode = RubyProf::GC_RUNS
- RubyProf.measure_mode = RubyProf::GC_TIME

The default value is RubyProf::PROCESS_TIME.

You may also specify the measure_mode by using the RUBY_PROF_MEASURE_MODE environment variable:

- export RUBY_PROF_MEASURE_MODE=process
- export RUBY_PROF_MEASURE_MODE=wall
- export RUBY_PROF_MEASURE_MODE=cpu
- export RUBY_PROF_MEASURE_MODE=allocations
- export RUBY_PROF_MEASURE_MODE=memory
- export RUBY_PROF_MEASURE_MODE=gc_runs
- export RUBY_PROF_MEASURE_MODE=gc_time

Note that these values have changed since ruby-prof-0.3.0.

On Linux, process time is measured using the clock method provided by the C runtime library. Note that the clock method does not report time spent in the kernel or child processes and therefore does not measure time spent in methods such as Kernel.sleep method. If you need to measure these values, then use wall time. Wall time is measured using the gettimeofday kernel method.

On Windows, timings are always wall times. If you set the clock mode to PROCESS_TIME, then timing are read using the clock method provided by the C runtime library. Note though, these values are wall times on Windows and not process times like on Linux. Wall time is measured using the GetLocalTime API.

If you use wall time, the results will be affected by other processes running on your computer, network delays, disk access, etc. As result, for the best results, try to make sure your computer is only performing your profiling run and is otherwise quiescent.

On both platforms, cpu time is measured using the RDTSC assembly function provided by the Pentium and PowerPC platforms. CPU time is dependent on the cpu's frequency. On Linux, ruby-prof attempts to read this value from "/proc/cpuinfo." On Windows, you must specify the clock frequency. This can be done using the RUBY_PROF_CPU_FREQUENCY environment variable:

```
export RUBY_PROF_CPU_FREQUENCY=<value>
```

You can also directly set the cpu frequency by calling:

```
RubyProf.cpu_frequency = <value>
```

Recursive Calls

Recursive calls occur when method A calls method A and cycles occur when method A calls method B calls method C calls method A. ruby-prof detects both direct recursive calls and cycles. Both are indicated in reports by a dash and number following a method name. For example, here is a flat profile from the test method RecursiveTest#test_recursive:

%self total self wait child calls name 100.00 2.00 2.00 0.00 0.00 2 Kernel#sleep

```
0.00      2.00      0.00      0.00      2.00         0  RecursiveTest#test_cycle
0.00      0.00      0.00      0.00      0.00         2  Fixnum#==
0.00      0.00      0.00      0.00      0.00         2  Fixnum#-
0.00      1.00      0.00      0.00      1.00         1  Object#sub_cycle-1
0.00      2.00      0.00      0.00      2.00         1  Object#sub_cycle
0.00      2.00      0.00      0.00      2.00         1  Object#cycle
0.00      1.00      0.00      0.00      1.00         1  Object#cycle-1
```

Notice the presence of Object#cycle and Object#cycle-1. The -1 means the method was either recursively called (directly or indirectly).

However, the self time values for recursive calls should always be accurate. It is also believed that the total times are

accurate, but these should be carefully analyzed to verify their veracity.

Multi-threaded Applications

Unfortunately, Ruby does not provide an internal api for detecting thread context switches. As a result, the timings ruby-prof reports for each thread may be slightly inaccurate. In particular, this will happen for newly spanned threads that immediately go to sleep. For instance, if you use Ruby's timeout library to wait for 2 seconds, the 2 seconds will be assigned to the foreground thread and not the newly created background thread. These errors can largely be avoided if the background thread performs an operation before going to sleeep.

Performance

Significant effort has been put into reducing ruby-prof's overhead as much as possible. Our tests show that the overhead associated with profiling code varies considerably with the code being profiled. Most programs will run approximately twice as slow while highly recursive programs (like the fibonacci series test) will run three times slower.

Windows Binary

The Windows binary is built with the latest version of MinGW. The source repository also includes a Microsoft VC++ 2005 solution. If you wish to run a debug version of ruby-prof on Windows, then it is highly recommended you use VC++.

License

Hanna RDoc template hand-crafted by **Mislav**