Examining R Profiling Data: The proftools Package

Luke Tierney  
University of Iowa

Riad Jarjour  
University of Iowa

Abstract

This note introduces the proftools package for examining data collected by R’s sampling profiler. proftools includes facilities for summarizing results at the function, call, and source line level; for filtering to narrow the focus to functions of primary interest; and for visualizing profiling data. Use of the package is illustrated with a small running example.

Keywords: proftools, profiling, R, Rprof.

1. Introduction

Profiling is a program analysis method for determining where a program run spends most of its execution time, and can be very helpful in guiding programmer effort for improving program performance. R includes a sampling based profiling mechanism that records information about calls on the stack at specified time intervals. If available, information about the specific source code lines active at the sampling point is recorded as well. Information about time spent in the garbage collector can also be collected. The collected profiling data is written to a file, by default the file Rprof.out in the current working directory. The function summaryRprof, available in the base package in R, provides a simple interface for examining this data. The proftools package provides a much more extensive set of tools for summarizing, visualizing, and filtering this data.

2. Collecting Profile Data

The proftools package includes an example script bootlm.R in the samples directory that runs several examples from the boot package (Canty and Ripley 2015) and then fits a simple linear model. The file path can be obtained as

```
srcfile <- system.file("samples", "bootlmEx.R", package = "proftools")
```

The traditional way to collect profiling data in R is to call Rprof to start profiling, run the code to be profiled, and then call Rprof again to end profiling. For example, to profile the code in bootlm.R, and collect both source line and GC information, you could use

```
profout <- tempfile()  
Rprof(file = profout, gc.profiling = TRUE, line.profiling = TRUE)  
source(srcfile)
```
Examining R Profiling Data: The \texttt{proftools} Package

```
Rprof(NULL)
pd <- readProfileData(profout)
unlink(profout)
```

The \texttt{proftools} package provides the simpler alternative

```
pd <- profileExpr(source(srcfile))
```

By default \texttt{profileExpr} enables GC and source information to be collected. It also trims off stack information leading up to the \texttt{profileExpr} call.

### 3. Summary Functions

The most basic summary function is \texttt{funSummary} for summarizing profile results at the function level. It produces information similar to the result returned by R’s \texttt{summaryRprof} but in a more usable form:

```
head(funSummary(pd), 10)
```

```
## total.pct gc.pct self.pct gcself.pct
## withVisible 100.00 9.00 0.00 0.00
## source 100.00 9.00 0.00 0.00
## eval 100.00 9.00 1.66 0.00
## FUN 77.73 3.79 2.13 0.47
## statistic 77.73 3.79 0.24 0.24
## lapply 77.49 3.79 1.66 0.00
## glm (bootlmEx.R:33) 50.47 2.61 0.95 0.00
## boot (bootlmEx.R:39) 37.68 2.37 0.24 0.24
## boot (bootlmEx.R:44) 36.49 1.42 0.00 0.00
## sapply 29.15 1.42 0.24 0.00
```

The result returned by \texttt{funSummary}, and most summary functions, is a data frame, so \texttt{head} can be used to focus on the top entries.

By default, when source information is available results are summarized at the call level, so multiple calls to \texttt{boot} from different source lines are shown separately. This can be suppressed by supplying the \texttt{srclines = FALSE} argument:

```
head(funSummary(pd, srclines = FALSE), 10)
```

```
## total.pct gc.pct self.pct gcself.pct
## withVisible 100.00 9.00 0.00 0.00
## source 100.00 9.00 0.00 0.00
## eval 100.00 9.00 1.66 0.00
## boot 77.73 4.03 0.24 0.24
## FUN 77.73 3.79 2.13 0.47
## statistic 77.49 3.79 0.24 0.24
## lapply 77.49 3.79 1.66 0.00
## glm 50.47 2.61 0.95 0.00
## sapply 29.15 1.42 0.24 0.00
## model.frame.default 28.91 2.13 0.71 0.24
```
Data can also be summarized by call:

```
head(callSummary(pd), 10)
```

<table>
<thead>
<tr>
<th>path</th>
<th>total.pct</th>
<th>gc.pct</th>
<th>self.pct</th>
<th>gcself.pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>withVisible -&gt; eval</td>
<td>100.00</td>
<td>9.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>source -&gt; withVisible</td>
<td>100.00</td>
<td>9.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>eval -&gt; eval</td>
<td>100.00</td>
<td>9.00</td>
<td>0.47</td>
<td>0.00</td>
</tr>
<tr>
<td>lapply -&gt; FUN</td>
<td>77.49</td>
<td>3.79</td>
<td>1.66</td>
<td>0.47</td>
</tr>
<tr>
<td>FUN -&gt; statistic</td>
<td>77.49</td>
<td>3.79</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>statistic -&gt; glm (bootlmEx.R:33)</td>
<td>50.47</td>
<td>2.61</td>
<td>0.95</td>
<td>0.00</td>
</tr>
<tr>
<td>eval -&gt; boot (bootlmEx.R:39)</td>
<td>37.68</td>
<td>2.37</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>boot (bootlmEx.R:39) -&gt; lapply</td>
<td>37.44</td>
<td>2.13</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>eval -&gt; boot (bootlmEx.R:44)</td>
<td>36.49</td>
<td>1.42</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>boot (bootlmEx.R:44) -&gt; lapply</td>
<td>36.49</td>
<td>1.42</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

When source information is available in the profile data the `srcSummary` function can be used to summarize at the source line level; only lines appearing in the sample are included:

```
srcSummary(pd)
```

<table>
<thead>
<tr>
<th>path</th>
<th>total.pct</th>
<th>gctotal.pct</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>bootlmEx.R:7</td>
<td>0.24</td>
<td>0.00</td>
<td>{ n &lt;- nrow(d)</td>
</tr>
<tr>
<td>bootlmEx.R:8</td>
<td>1.66</td>
<td>0.00</td>
<td>gp1 &lt;- 1:table(as.numeric(d ...</td>
</tr>
<tr>
<td>bootlmEx.R:9</td>
<td>0.71</td>
<td>0.24</td>
<td>m1 &lt;- sum(d[gp1,1] * f[gp1] ...</td>
</tr>
<tr>
<td>bootlmEx.R:10</td>
<td>0.24</td>
<td>0.00</td>
<td>m2 &lt;- sum(d[-gp1,1] * f[-gp ...</td>
</tr>
<tr>
<td>bootlmEx.R:11</td>
<td>0.47</td>
<td>0.00</td>
<td>ss1 &lt;- sum(d[gp1,1]^2 * f[g ...</td>
</tr>
<tr>
<td>bootlmEx.R:13</td>
<td>0.24</td>
<td>0.00</td>
<td>c(m1 - m2, (ss1 + ss2)/(sum ...</td>
</tr>
<tr>
<td>bootlmEx.R:16</td>
<td>3.79</td>
<td>0.24</td>
<td>boot(grav1, diff.means, R = 999 ...</td>
</tr>
<tr>
<td>bootlmEx.R:21</td>
<td>0.24</td>
<td>0.00</td>
<td>nuke.diag &lt;- glm.diag(nuke.lm)</td>
</tr>
<tr>
<td>bootlmEx.R:33</td>
<td>50.47</td>
<td>2.61</td>
<td>lm.b &lt;- glm(fit+resid[inds] ...</td>
</tr>
<tr>
<td>bootlmEx.R:35</td>
<td>22.99</td>
<td>0.71</td>
<td>pred.b &lt;- predict(lm.b, x.pred)</td>
</tr>
<tr>
<td>bootlmEx.R:36</td>
<td>0.47</td>
<td>0.24</td>
<td>c(coef(lm.b), pred.b - (fit ...</td>
</tr>
<tr>
<td>bootlmEx.R:39</td>
<td>37.68</td>
<td>2.37</td>
<td>nuke.boot &lt;- boot(nuke.data, nu ...</td>
</tr>
<tr>
<td>bootlmEx.R:44</td>
<td>36.49</td>
<td>1.42</td>
<td>nuke.boot &lt;- boot(nuke.data, nu ...</td>
</tr>
<tr>
<td>bootlmEx.R:50</td>
<td>6.87</td>
<td>0.95</td>
<td>X &lt;- matrix(rnorm(n * p), n, p)</td>
</tr>
<tr>
<td>bootlmEx.R:51</td>
<td>0.47</td>
<td>0.00</td>
<td>y &lt;- rnorm(n) + X[,1]</td>
</tr>
<tr>
<td>bootlmEx.R:52</td>
<td>14.45</td>
<td>4.03</td>
<td>fit &lt;- lm(y ~ X)</td>
</tr>
</tbody>
</table>

The function `annotateSource` can show the full files with profiling annotations.

A useful way to examine profile data is to look for hot execution paths. This approach sorts functions called at top level, i.e. at the bottom of the call stack, by the total amount of time spent in their top level calls; within each top level call to a function f the functions called by f are sorted by the amount of time spent in them within the top level call to f; and the process continues for higher level calls. The function `hotPaths` produces a hot path summary; the `total.pct` argument causes leaf functions in stack traces to be pruned back until the execution time percentage for each stack trace is at least `total.pct`:

```
hotPaths(pd, total.pct = 10.0)
```

<table>
<thead>
<tr>
<th>path</th>
<th>total.pct</th>
<th>self.pct</th>
</tr>
</thead>
</table>

## source
# . withVisible 100.00 0.00
# .. eval 100.00 0.00
# ... eval 100.00 0.00
# ... boot (bootlmEx.R:39) 37.68 0.24
# ... lapply 37.44 0.00
# ... FUN 37.44 0.00
# .... statistic 37.44 0.00
# .... glm (bootlmEx.R:33) 26.54 0.24
# ... eval 13.74 13.74
# ... predict (bootlmEx.R:35) 10.66 0.00
# ... predict.glm 10.66 10.66
# ... FUN 36.49 0.00
# ... statistic 36.49 0.00
# ... glm (bootlmEx.R:33) 23.93 0.47
# ... eval 10.66 10.66
# ... predict (bootlmEx.R:35) 12.32 0.24
# ... predict.glm 12.09 12.09
# ... lm (bootlmEx.R:52) 14.45 0.00

Examining the result of hotPaths starting with high values of total.pct and then moving to lower values is a useful way to explore where the computational effort is concentrated. An alternative for limiting the depth to which stack traces are followed is provided by the maxdepth argument.

### 4. Filtering Profile Data

The hot path summary shows information associated with the source command itself that is not directly relevant to our analysis. The filterProfileData function can be used to select or omit certain functions, drop functions with small self or total times, narrow to a particular time interval, among others. For example, by selecting only stack traces that include calls to withVisible and then trimming off the leading four calls we can focus just on the work done in the sourced file:

```r
filteredPD <- filterProfileData(pd, select = "withVisible", skip = 4)
```

The hot path summary for this reduced profile is

```r
hotPaths(filteredPD, total.pct = 10.0)
```
We can use the `focus` filter to further narrow our examination to stack frames containing calls to `glm` and also remove all calls preceding the first `glm` call from the selected stack frames. For this reduced data it also makes sense to follow the paths further by lowering `total.pct` to 5%:

```
glmPD <- filterProfileData(filteredPD, focus = "glm")
hotPaths(glmPD, total.pct = 5.0)
```

## 5. Visualizing Profile Data

Call graphs annotated with profile information are a very popular way to view profiling results. `plotProfileCallGraph` uses the `graph` (Gentleman, Whalen, Huber, and Falcon 2015) and `Rgraphviz` (Gentry, Long, Gentleman, Falcon, Hahne, Sarkar, and Hansen 2015) packages from Bioconductor to render an annotated call graph. The default style for the graph is
Examine R Profiling Data: The proftools Package

Based on the style used in Google’s profiling library but can be customized in a number of ways. A call graph for the full profile data is produced by

```
plotProfileCallGraph(pd)
```

and is shown in Figure 1.

By default the call graph size is limited to at most 30 nodes; nodes with lower total hit counts are dropped. The maxnodes argument can be used to adjust this limit. Also by default color is used to encode the total hit score for the function nodes. This can be suppressed with the score = "none" argument.

We can obtain a more readable graph by filtering. For example, to examine the glm.fit call and its callees we can use

```
plotProfileCallGraph(filterProfileData(pd, focus = "glm.fit"))
```

The result is shown in Figure 2.

A printed version of the call graph, similar to the call graph produced by gprof (Graham, Kessler, and Mckusick 1982), can be obtained with printProfileCallGraph. For the subgraph of the glm.fit calls, for example,

```
printProfileCallGraph(filterProfileData(pd, focus = "glm.fit"))
```

produces the printed representation shown in Figure 3.

Another visualization sometimes used is a flame graph. A flame graph for the original data is produced by

```
flameGraph(pd)
```

The result is shown in Figure 4. A flame graph of the filtered data is produced by

```
flameGraph(filteredPD)
```

and shown in Figure 5.

The vertical positions of rectangles in these flame graphs represent call depth on the stack. The widths of the rectangles represent the amount of time spent in a call at a particular call or set of calls at a particular depth. The order argument determines the ordering of call rectangles at a particular level within a call at the lower level. The default order is "hot"; it uses the hot path ordering with the call with the largest amount of time first. This produces a visual representation of the hot path summary. The "alpha" ordering orders the calls alphabetically.

Specifying order = "time" shows the calls in the order in which they occurred:

```
flameGraph(pd, order = "time")
```

Figure 6 shows the boot calls preceding the data generation for the lm call and the lm call itself.
Figure 1: Full call graph of profile data.
Figure 2: Call graph for `glm.fit` call.
Call graph

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>% self</th>
<th>% children</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.35</td>
<td>3.55</td>
<td>3.79</td>
<td>glm.fit [1]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.95</td>
<td>%*% [4]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.95</td>
<td>.Call [2]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>any [6]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.24</td>
<td>0.24</td>
<td>colnames&lt;- [7]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>dev.resids [8]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>eval [9]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>is.function [11]</td>
</tr>
<tr>
<td></td>
<td>0.71</td>
<td>0.24</td>
<td>0.24</td>
<td>structure [3]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>sum [13]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.24</td>
<td>0.24</td>
<td>variance [14]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>2</td>
<td>0.95</td>
<td>0.00</td>
<td></td>
<td>glm.fit [1]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>3</td>
<td>0.95</td>
<td>0.71</td>
<td>0.24</td>
<td>structure [3]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.95</td>
<td>.*% [5]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>4</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>structure [3]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>5</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>%*% [4]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>6</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>any [6]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>7</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>is.data.frame [10]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>colnames&lt;- [7]</td>
</tr>
<tr>
<td></td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>is.data.frame [10]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>8</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>dev.resids [8]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>9</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>glm.fit [1]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>10</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>is.data.frame [10]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>11</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>is.function [11]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>12</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>variance [14]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>13</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>rep.int [12]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>14</td>
<td>0.24</td>
<td>0.00</td>
<td>0.24</td>
<td>glm.fit [1]</td>
</tr>
</tbody>
</table>

Figure 3: Printed call graph.
Exercising R Profiling Data: The proftools Package

Call Graph

Figure 4: Flame graph visualizing hot paths for the full profile data.

Default colors for flame graphs are based on the rainbow palette. Alternative colors can be specified by providing a colormap function. If space permits, the call labels are printed within the rectangles representing the calls. A method for the identify function is available that can be used to identify the individual calls when they are not visible. An option to this method is to request that all instance of the identified calls be outlined on the plot. The identify method uses the value returned by flameGraph; thus it would be used as

```r
fg <- flameGraph(pd)
identify(fg)
```

A third visualization that is sometimes used is a callee tree map. This is produced by

```r
calleeTreeMap(pd)
```

and shown in Figure 7. A callee tree map shows a tree map (Shneiderman 1998) of the calls in the call tree. The tiling algorithm used depends on the squarify argument. If squarify is TRUE then the squarified algorithm (Bruls, Huizing, and van Wijk 2000) is used; otherwise, the longer side is partitioned. Again a method for identify is provided calleeTreeMap objects. Clicking on rectangles in the plot returns a list of the call stacks for the identified rectangles.

6. Graphical User Interfaces

The function writeCallgrindFile can be used to write the profile data in Valgrind’s callgrind format for use with the kcachegrind or qcachegrind graphical user interfaces available on
Linux and Mac OS X. Figure 8 shows the kcachegrind interface for the example profile data written out using writeCallgrindFile. By default writeCallgrindFile for the common case where, as here, the profiled code is in a file run using source and source code information is retained. If this is the case, writeCallgrindFile removes calls associated with the source call and adds a <TOP> entry; this persuades kcachegrind to show the top level line information.

Graphical user interfaces within R will be made available in the proftoolsGUI package. The current development version provides two interfaces, one based on gWidgets2 (Verzani 2015) and one on shiny (Chang, Cheng, Allaire, Xie, and McPherson 2015).

References


Examining R Profiling Data: The proftools Package

Figure 6: Time graph of the full profile data.


Callee Tree Map

Figure 7: Call tree map of the full profile data.

Figure 8: kcachegrind interface for examining profile data for the example.
Affiliation:

Luke Tierney
Department of Statistics and Actuarial Science
Faculty of Statistics
University of Iowa
Iowa City, USA
E-mail: luke-tierney@uiowa.edu
URL: http://homepage.stat.uiowa.edu/~luke/

Riad Jarjour
Department of Statistics and Actuarial Science
Faculty of Statistics
University of Iowa
Iowa City, USA
E-mail: riad-jarjour@uiowa.edu