“My bad opinions”

2015/01/15

Awk in 20 Minutes

What's Awk

Awk is a tiny programming language and a command line tool. It's particularly appropriate for log parsing on servers, mostly because Awk will operate on files, usually structured in lines of human-readable text.

I say it's useful on servers because log files, dump files, or whatever text format servers end up dumping to disk will tend to grow large, and you'll have many of them per server. If you ever get into the situation where you have to analyze gigabytes of files from 50 different servers without tools like Splunk or its equivalents, it would feel fairly bad to have and download all these files locally to then drive some forensics on them.

This personally happens to me when some Erlang nodes tend to die and leave a crash dump of 700MB to 4GB behind, or on smaller individual servers (say a VPS) where I need to quickly go through logs, looking for a common pattern.

In any case, Awk does more than finding data (otherwise, `grep` or `ack` would be enough) — it also lets you process the data and transform it.

Code Structure

An Awk script is structured simply, as a sequence of patterns and actions:

```
# comment
Pattern1 { ACTIONS; }

# comment
Pattern2 { ACTIONS; }

# comment
Pattern3 { ACTIONS; }

# comment
Pattern4 { ACTIONS; }
```

Every line of the document to scan will have to go through each of the patterns, one at a time. So if I pass in a file that contains the following content:

```
this is line 1
this is line 2
```

Then the content `this is line 1` will match against `Pattern1`. If it matches, `ACTIONS` will be executed. Then `this is line 1` will match against `Pattern2`. If it doesn't match, it skips to `Pattern3`, and so on.

Once all patterns have been cleared, `this is line 2` will go through the same process, and so on for other lines, until the input has been read entirely.

This, in short, is Awk's execution model.

Data Types

Awk only has two main data types: strings and numbers. And even then, Awk likes to convert them into each other. Strings can be interpreted as numerals to convert their values to numbers. If the string doesn't look like a numeral, it's `0`.

Both can be assigned to variables in `ACTIONS` parts of your code with the `=` operator. Variables can be declared anywhere, at any time, and used even if they're not initialized: their default value is `""`, the empty string.

Finally, Awk has arrays. They're unidimensional associative arrays that can be started dynamically. Their syntax is just `var[key] = value`. Awk can simulate multidimensional arrays, but it's all a big hack anyway.

Patterns

The patterns that can be used will fall into three broad categories: regular expressions, Boolean expressions, and special patterns.

Regular and Boolean Expressions

The Awk regular expressions are your run of the mill regexes. They're not PCRE under `awk` (but `gawk` will support the fancier stuff — it depends on the implementation! See with `awk --version`), though for most usages they'll do plenty:

```
/admin/ { ... }     # any line that contains 'admin'
/^admin/ { ... }    # lines that begin with 'admin'
/admin$/ { ... }    # lines that end with 'admin'
/^[0-9.]+ / { ... } # lines beginning with series of numbers and periods
/(POST|PUT|DELETE)/ # lines that contain specific HTTP verbs
```

And so on. Note that the patterns *cannot* capture specific groups to make them available in the `ACTIONS` part of the code. They are specifically to match content.

Boolean expressions are similar to what you would find in PHP or Javascript. Specifically, the operators `&&` ("and"), `||` ("or"), and `!` ("not") are available. This is also what you'll find in pretty much all C-like languages. They'll operate on any regular data type.

What's specifically more like PHP and Javascript is the comparison operator, `==`, which will do fuzzy matching, so that the string `"23"` compares equal to the number `23`, such that `"23" == 23` is *true*. The operator `!=` is also available, without forgetting the other common ones: `>`, `<`, `>=`, and `<=`.

You can also mix up the patterns: Boolean expressions can be used along with regular expressions. The pattern `/admin/ || debug == true` is valid and will match when a line that contains either the word 'admin' is met, or whenever the variable `debug` is set to `true`.

Note that if you have a specific string or variable you'd want to match against a regex, the operators `~` and `!~` are what you want, to be used as `string ~ /regex/` and `string !~ /regex/`.

Also note that all patterns are *optional*. An Awk script that contains the following:

```
{ ACTIONS }
```

Would simply run `ACTIONS` for every line of input.

Special Patterns

There are a few special patterns in Awk, but not that many.

The first one is `BEGIN`, which matches only *before* any line has been input to the file. This is basically where you can initiate variables and all other kinds of state in your script.

There is also `END`, which as you may have guessed, will match *after* the whole input has been handled. This lets you clean up or do some final output before exiting.

Finally, the last kind of pattern is a bit hard to classify. It's halfway between variables and special values, and they're called *Fields*, which deserve a section of their own.

Fields

Fields are best explained with a visual example:

```
# According to the following line
#
# $1         $2    $3
# 00:34:23   GET   /foo/bar.html
# \_____________  _____________/
#               $0

# Hack attempt?
/admin.html$/ && $2 == "DELETE" {
  print "Hacker Alert!";
}
```

The fields are (by default) separated by white space. The field `$0` represents the entire line on its own, as a string. The field `$1` is then the first bit (before any white space), `$2` is the one after, and so on.

A fun fact (and a thing to avoid in most cases) is that you can *modify the line* by assigning to its field. For example, if you go `$0 = "HAHA THE LINE IS GONE"` in one block, the next patterns will now operate on that line instead of the original one, and similarly for any other field variable!

Actions

There's a bunch of possible actions, but the most common and useful ones (in my experience) are:

```
{ print $0; }  # prints $0. In this case, equivalent to 'print' alone
{ exit; }      # ends the program
{ next; }      # skips to the next line of input
{ a=$1; b=$0 } # variable assignment
{ c[$1] = $2 } # variable assignment (array)

{ if (BOOLEAN) { ACTION }
  else if (BOOLEAN) { ACTION }
  else { ACTION }
}
{ for (i=1; i<x; i++) { ACTION } }
{ for (item in c) { ACTION } }
```

This alone will contain a major part of your Awk toolbox for casual usage when dealing with logs and whatnot.

The *variables are all global*. Whatever variables you declare in a given block will be visible to other blocks, for each line. This severely limits how large your Awk scripts can become before they're unmaintainable horrors. Keep it minimal.

Functions

Functions can be called with the following syntax:

```
{ somecall($2) }
```

There is a somewhat restricted set of built-in functions available, so I like to point to regular documentation for these.

User-defined functions are also fairly simple:

```
# function arguments are call-by-value
function name(parameter-list) {
     ACTIONS; # same actions as usual
}

# return is a valid keyword
function add1(val) {
     return val+1;
}
```

Special Variables

Outside of regular variables (global, instantiated anywhere), there is a set of special variables acting a bit like configuration entries:

```
BEGIN { # Can be modified by the user
  FS = ",";   # Field Separator
  RS = "\n";  # Record Separator (lines)
  OFS = " ";  # Output Field Separator
  ORS = "\n"; # Output Record Separator (lines)
}
{ # Can't be modified by the user
  NF          # Number of Fields in the current Record (line)
  NR          # Number of Records seen so far
  ARGV / ARGC # Script Arguments
}
```

I put the modifiable variables in BEGIN because that's where I tend to override them, but that can be done anywhere in the script to then take effect on follow-up lines.

Examples

That's it for the core of the language. I don't have a whole lot of examples there because I tend to use Awk for quick one-off tasks.

I still have a few files I carry around for some usage and metrics, my favorite one being a script used to parse Erlang crash dumps shaped like this:

```
=erl_crash_dump:0.3
Tue Nov 18 02:52:44 2014
Slogan: init terminating in do_boot ()
System version: Erlang/OTP 17 [erts-6.2] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-poll:false]
Compiled: Fri Sep 19 03:23:19 2014
Taints:
Atoms: 12167
=memory
total: 19012936
processes: 4327912
processes_used: 4319928
```

```
system: 14685024
atom: 339441
atom_used: 331087
binary: 1367680
code: 8384804
ets: 382552
=hash_table:atom_tab
size: 9643
used: 6949
...
=allocator:instr
option m: false
option s: false
option t: false
=proc:<0.0.0>
State: Running
Name: init
Spawned as: otp_ring0:start/2
Run queue: 0
Spawned by: []
Started: Tue Nov 18 02:52:35 2014
Message queue length: 0
Number of heap fragments: 0
Heap fragment data: 0
Link list: [<0.3.0>, <0.7.0>, <0.6.0>]
Reductions: 29265
Stack+heap: 1598
OldHeap: 610
Heap unused: 656
OldHeap unused: 468
Memory: 18584
Program counter: 0x00007f42f956620O (init:boot_loop/2 + 64)
CP: 0x0000000000000000 (invalid)
=proc:<0.3.0>
State: Waiting
...
=port:#Port<0.0>
Slot: 0
Connected: <0.3.0>
Links: <0.3.0>
Port controls linked-in driver: efile
=port:#Port<0.14>
Slot: 112
Connected: <0.3.0>
...
```

To yield the following result:

```
$ awk -f queue_fun.awk $PATH_TO_DUMP
MESSAGE QUEUE LENGTH: CURRENT FUNCTION
======================================
10641: io:wait_io_mon_reply/2
12646: io:wait_io_mon_reply/2
32991: io:wait_io_mon_reply/2
2183837: io:wait_io_mon_reply/2
730790: io:wait_io_mon_reply/2
80194: io:wait_io_mon_reply/2
...
```

Which is a list of functions running in Erlang processes that caused mailboxes to be too large. Here's the script:

```
# Parse Erlang Crash Dumps and correlate mailbox size to the currently running
# function.
#
# Once in the procs section of the dump, all processes are displayed with
# =proc:<0.M.N> followed by a list of their attributes, which include the
# message queue length and the program counter (what code is currently
# executing).
#
# Run as:
#
#    $ awk -v threshold=$THRESHOLD -f queue_fun.awk $CRASHDUMP
#
# Where $THRESHOLD is the smallest mailbox you want inspects. Default value
# is 1000.
BEGIN {
    if (threshold == "") {
        threshold = 1000 # default mailbox size
    }
    procs = 0 # are we in the =procs entries?
    print "MESSAGE QUEUE LENGTH: CURRENT FUNCTION"
    print "======================================"
}

# Only bother with the =proc: entries. Anything else is useless.
procs == 0 && /^=proc/ { procs = 1 } # entering the =procs entries
procs == 1 && /^=/ && !/^=proc/ { exit 0 } # we're done


# Message queue length: 1210
# 1       2     3       4
/^Message queue length: / && $4 >= threshold { flag=1; ct=$4 }
/^Message queue length: / && $4 <  threshold { flag=0 }

# Program counter: 0x00007f5fb8cb2238 (io:wait_io_mon_reply/2 + 56)
# 1       2        3                  4                       5 6
flag == 1 && /^Program counter: / { print ct ":", substr($4,2) }
```

crash-dump.awk hosted with ❤ by **GitHub** **view raw**

Can you follow along? If so, you can understand Awk. Congratulations.

- Fred T-H
- @mononcqc
- RSS

- Property-Based Testing with PropEr
- Erlang in Anger
- Learn You Some Erlang