Script tuning in Bash

SCRIPT WORKOUT

In the old days, shells were capable of little more than calling external programs and executing basic, internal commands. With all the bells and whistles in the latest versions of Bash, however, you hardly need the support of external tools. **BY MIRKO DÖLLE**

Many scripts treat Bash as if it were capable of little more than calling external programs. This is surprising, since version 2 of the default shell has a command set that covers everything from complex string manipulation, through regular expressions, to arrays; useful functions that make expensive program calls redundant.

The main advantage of internal functions is that the shell does not need to spawn a new process, which saves processing time and memory. This capability can be important, particularly if you need to launch a program like *grep* or *cut* in a loop, as the completion time and memory consumption of a script can explode if you're not careful. This article describes some simple techiques for speeding up your Bash scripts.

Benchmarking

The following scripts evaluate an Apache logfile on a website for comparison's sake. If you are interested in which pages have been requested, you need to isolate the GET string from the log, as follows:

```
84.57.16.30 - - ➥
[21/Oct/2005:04:18:26 +0200] ➥
"GET /favicon.ico HTTP/1.1" ➥
404 209 "-" "Mozilla/5.0 ➥
(X11; U; Linux i686; de-DE; ➥
rv:1.7.5) Gecko/20041122 ➥
Firefox/1.0"
```

Listing 1: External Log Evaluation

```
01 #!/bin/bash
02 IFS=$'\n'
03 for l in `cat access.log`; do
04   IFS=" "
05   echo "${l}" | cut -d" " -f7
06 done
```

Listing 2: Internal Log Evaluation

```
01 #!/bin/bash
02 exec 3<access.log
03 while read -u 3 Request; do
04   Request="${Request##*GET }"
05   echo "${Request%% HTTP/*}"
06 done
```

Listing 1 shows an approach that many Bash scripts use. The call to *cat* in Line 3 first reads the whole logfile, and hands it to the *for* loop as a parameter list, meaning that Bash has to cache the contents of the file first. In Line 5, Listing 1 then goes on to call the external *cut* program for every single log entry. Running on 750 MHz Pentium III machine, the script took over 18.5 seconds to parse a 600 Kbyte Apache logfile.

In contrast to this, Listing 2 took just 3.3 seconds, which is almost six times as fast: it uses file descriptor 3 to open the file, and it uses the *Request* variable to process one line at a time in the loop, meaning that Bash will only ever need to

Listing 3: Internal grep

```
01 #!/bin/bash
02 exec 3<$2
03 while read -u 3 line; do
04   if [ -z "${line/*${1}*}" ];
   then
05     echo "$line"
06   fi
07 done
```

cache a single log entry. The script then removes the characters from the start of the line, up to and including *GET,* and all the characters from the end of the line up to and including *HTTP/*.

You could save another tenth of a second by removing all the characters up to and including the blank from the end of the request, as this simplifies the string comparison performed by the internal Bash function.

Replacement Functions

The *basename* and *dirname* tools are easily replaced using Bash functions. You need the *${Variable%Pattern}* string function, which removes the shortest matching pattern from the end of the string, and *${Variable##Pattern}*, which removes the longest matching pattern from the start of the string, to do this. A simple *alias* is actually all it takes to replace *dirname*:

```
alias dirname=echo ${1%/*};
```

The functionality for *basename* is not much more complex, although you do need to consider the fact that *basename* can remove file extensions, which means combining both string functions:

```
function basename() {
  B=${1##*/}
  echo ${B%$2}
}
```

It is more efficient to store the result of the string truncation in the *B* variable than to *set* the first parameter and pass the second parameter.

It doesn't make sense to replace every single program call with internal Bash commands. Listing 3 is a good example of this: it implements a rudimentary *grep*. Although the script comprises just a few lines of code, and may look efficient at first, it takes over two minutes to search for a filename in the 600 Kbyte web server log – *grep* turned up with the goodies in less than 0.1 seconds.

The search pattern is the biggest performance killer. If a match is found, the line parsed from the file is completely deleted by a search and replaced in Line 4. The search pattern, **${1}**, tells Bash to search every single character in the line for a match. If you use *#*${1}** as your search pattern, telling Bash to search at the start of the line only, Bash only does one comparison per line, and this reduces the script runtime to less than three seconds.

Dissecting IPs

Although the string functions were not the bottleneck in the previous example, there may be more elegant, and quicker, ways of dissecting strings in some scenarios. For example, if you need to sort the IP addresses from which requests originated, you can't use *sort* or a simple lexical sort function. This would put *217.83.13.152* before *62.104.118.59*. Instead, you need to extract the individual bytes in the IP address, convert them to a sortable format, sort them, and finally display the results without duplicates.

Listings 4 and 5 show two possible approaches with completely different performance characteristics. The script in Listing 4 starts by parsing the logfile line by line (Line 3), and then extracts the first IP address in Line 4. Lines 6 through 10 remove one octet at a time from the rear of the IP address, and store the address byte as a decimal in the *IP* array.

In Line 11, the call to *printf*, which is also an internal Bash command, outputs the four octets in the IP address as dot-separated, three-digit decimals with zero padding. The last line pipes the output to the external *sort* program, before *uniq* removes the duplicates. Listing 4 takes about 2.6 seconds to process a 600 Kbyte Apache logfile.

Listing 4: String Functions

```
01 #!/bin/bash
02 function GetIP() {
03   while read -u $1 Request; do
04     tmp="${Request%% *}"
05     IP[1]="${tmp%%.*}"
06     IP[4]="${tmp##*.}"
07     tmp="${tmp%.*}"
08     IP[3]="${tmp##*.}"
09     tmp="${tmp%.*}"
10     IP[2]="${tmp##*.}"
11     printf
   "%03d.%03d.%03d.%03d\n"
   ${IP[1]} ${IP[2]} ${IP[3]}
   ${IP[4]}
12   done
13 }
14 exec 3<access.log
15 GetIP 3 | sort | uniq
```

Listing 5: String Functions

```
01 #!/bin/bash
02 function GetIP() {
03   IFS=". "
04   while read -u $1 -a IP; do
05     printf
   "%03d.%03d.%03d.%03d\n"
   ${IP[0]} ${IP[1]} ${IP[2]}
   ${IP[3]}
06   done
07 }
08 exec 3<access.log
09 GetIP 3 | sort | uniq
```

Compacting Functions

The program in Listing 5 does the same job as Listing 4, but it takes only 1.6 seconds, an improvement of almost 40 percent. The string functions in Lines 4 through 10 in Listing 4 are what cause the difference in runtimes: instead of extracting the IP address first, and then using seven function calls to dissect it, Listing 5 calls Bash's internal *read* function with the special *IFS* variable. Bash treats the characters stored in *IFS* as parameter separators – these default to the space, tab, and newline characters.

Line 3 of Listing 5 defines the dot and blank as separators. Calling *read* with the *-a* switch tells the function not to store a whole line in a variable, but to use the separator from *IFS*, and to write the input to the *IP* array variable one element at a time. The octets that make up the IP address go straight to the *IP[0]* through *IP[3]* variables on calling *read*. Thus, a single function call in Listing 5 replaces Lines 3 through 10 in Listing 4.

Incidentally, you could replace the external program calls to *sort* and *uniq* using Bash functions, but you can't expect Bash to match *sort*, a C program, for efficiency. Just as in real life, some of your Bash tweaking may go unrewarded, but coding for efficiency still pays off in the long run. ■

INFO

[1] Sample programs: *http://www.linux-magazine.com/Magazine/Downloads/64/bash*