Scan - Making string manipulation efficient

Colin Walker, 2008-24-04

One of the many things that makes iRules so powerful is the ability to inspect data and break it up into custom variables. You can store whatever information you need in your desired format and put it into memory on the BIG-IP for use later. Whether you decide to use it for the same request, the same connection or whenever you see the next connection with similar data come through, that's completely up to you. To make variables so valuable, however, you need to be able to not only pull data off the wire, but to break it up and manipulate it into usable formats and pieces.

TCL, which iRules is based off of, offers many ways of performing this type of data manipulation. Depending on your needs you can split, format, sort, concat, increment, join, and list data. There's even the powerful string command and its many sub-commands that allow you to manipulate strings to great effect. An extremely useful and powerful command that is often overlooked, however, is the **scan** command. Not only is this command full of utility, but it's efficient as well, making it a great choice in many situations once you understand how to use it.

The scan command's definition over on sourceforge says - "*scan* - Parse string using conversion specifiers in the style of sscanf" - so what does that mean? It means that you can use the scan command in lieu of other commands, depending on the situation, to do things like string conversions and splitting data into substrings. Think of it almost like a limited regular expression for a single string, but much, much faster.

The way the scan command works is by taking a string and breaking it down based on user supplied **"conversion characters"**. A conversion character is a character specified for the scan command that dictates what type of data is being read in, and the type of output the command is going to have. These characters always begin with %, and can take some supplied arguments such as maximum string length, which you can read more about on sourceforge. Here are the available conversion characters:

> **d**
> The input substring must be a decimal integer. It is read in and the integer value is stored in the variable, truncated as required by the size modifier value.
>
> **o**
> The input substring must be an octal integer. It is read in and the integer value is stored in the variable, truncated as required by the size modifier value.
>
> **x**
> The input substring must be a hexadecimal integer. It is read in and the integer value is stored in the variable, truncated as required by the size modifier value.
>
> **u**
> The input substring must be a decimal integer. The integer value is truncated as required by the size modifier value, and the corresponding unsigned value for that truncated range is computed and stored in the variable as a decimal string. The conversion makes no sense without reference to a truncation range, so the size modifier ll is not permitted in combination with conversion character u.
>
> **i**
> The input substring must be an integer. The base (i.e. decimal, binary, octal, or hexadecimal) is determined in the same fashion as described in expr. The integer value is stored in the variable, truncated as required by the size modifier value.
>
> **c**
> A single character is read in and its Unicode value is stored in the variable as an integer value. Initial white space is not skipped in this case, so the input substring may be a white-space character.
>
> **s**
> The input substring consists of all the characters up to the next white-space character; the characters are copied to the variable.

> **e or f or g**
> The input substring must be a floating-point number consisting of an optional sign, a string of decimal digits possibly containing a decimal point, and an optional exponent consisting of an e or E followed by an optional sign and a string of decimal digits. It is read in and stored in the variable as a floating-point value.
>
> **[chars]**
> The input substring consists of one or more characters in chars. The matching string is stored in the variable. If the first character between the brackets is a] then it is treated as part of chars rather than the closing bracket for the set. If chars contains a sequence of the form a-b then any character between a and b (inclusive) will match. If the first or last character between the brackets is a -, then it is treated as part of chars rather than indicating a range.
>
> **[^chars]**
> The input substring consists of one or more characters not in chars. The matching string is stored in the variable. If the character immediately following the ^ is a] then it is treated as part of the set rather than the closing bracket for the set. If chars contains a sequence of the form a-b then any character between a and b (inclusive) will be excluded from the set. If the first or last character between the brackets is a -, then it is treated as part of chars rather than indicating a range value.
>
> **n**
> No input is consumed from the input string. Instead, the total number of characters scanned from the input string so far is stored in the variable.

That's a lot of options, I know, but we won't try and cover them all here. You can get the official sourceforge documentation for the scan command here. I'll let you read up on the entirety of the command options there. We'll just go over a few here to get you started and show you how the scan command can be useful in your iRules.

Let's start with something simple and go through a few examples. Perhaps I have a string that I want to separate into multiple parts. Maybe it's a three part host name like you'd often see on the web, such as "www.mydomain.com". If I wanted to break this apart into three separate variables, I could use scan like this:

```
when HTTP_REQUEST {
  scan [HTTP::host] %s.%s.%s prefix host suffix
  log "Prefix = $prefix; Host = $host; Suffix = $suffix"
}
```

In the above example we're using the conversion character s, which reads in all characters until the next whitespace, and separating the hostname into three separate values. We're then logging those values, which would result in an entry in /var/log/ltm like:

```
Apr  1 16:18:50 tmm tmm[293245]: Rule rule_log_hosts : client Prefix = www; Host = mydomain; Suffix =
```

As you can see, the string was split apart much like using the split command, which would have resulted in a TCL list. The scan command, however, is more efficient, not to mention you get to specify your own variable names, which can be helpful at times, and save you from having to perform further commands to dump list values into single variables later, if that's the format you want them in.

Another thing that you can do with scan, which it excels at, is number manipulation. In this example let's split apart a date string. We'll take the string from the above log entry - "Apr 1 16:18:50". What if we wanted that separated into individual strings containing each date part? It would look something like:

```
when CLIENT_DATA {
  set myString "Apr  1 16:18:50"
  scan $myString %s%s%d:%d:%d month date hour minute second
  log "Month = $month; Date = $date; Hour = $hour; Minute = $minute; Second = $second"
}
```

Notice how we use a single command to extract many different values in multiple data types separated by different characters. This would take many split and string operations to achieve, or a more expensive regular expression. The above log statement should output:

```
Apr  24 11:24:50 tmm tmm[142512]: Rule rule_log_dateparts : Month = Apr; Date = 1; Hour = 16; Minute
```

As you can see we now have the individual values stored in their respective variables, ready for use in whatever manner we need them. Now that you've seen a couple of ways to split data up with scan, for this last example I'll show you a way that you can verify string formatting with scan, another handy use case. In the below example we're going to use the hostname example from earlier and modify it a little. In this case we want to ensure that the host that's being requested always has a three part domain name, like outlined in the example above. Again, we could do this with string and split commands, but scan makes it straight-forward and efficient, like this:

```
when HTTP_REQUEST {
  if { [scan [HTTP::host] %s.%s.%s prefix host suffix] != 3 } {
    HTTP::respond 200 content "The host name you supplied was invalied, please try again."
  } else {
    pool httpPool
  }
}
```

As you can see the scan command has many uses ranging from very simple to far more advanced. Hopefully this de-mystifies it enough to get you started. For some more advanced examples of using the scan command, check out some of these great examples in our codeshare that make use of it in some more interesting and advanced ways:

- DNS Decoding
- SOCK5 SSL Persistence
- Validate String Characters in a Cookie
- RADIUS Load Balancing with iRules

Get the Flash Player to see this player.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com