Parsing Text Printouts within Ansible Playbooks

From Building Network Automation Solutions

Ansible can take the data produced by an external script (or **show** command executed on a network device) and use it in subsequent tasks in an Ansible playbook – an ideal solution if you need collect device data to generate a report, verify device state (example: *are BGP neighbors up*) or check device state before configuring it (example: *is another customer configured on this interface?*)

Ansible can use *structured data* in JSON or YAML format directly. There are Ansible plug-ins that can process data returned in XML format. But what if the networking device cannot do anything else but return text printout in tabular or free-form format (I'm looking at you, Cisco IOS)?

Try to Get the Structured Data First

Before starting to build a kludge that would turn a **show** command printout into something that Ansible could use explore all other options:

- Your device might have REST API that returns the data you need in JSON or XML format (example: late Cisco IOS XE, Junos);
- Your device might support NETCONF and return the required data in XML format (Junos, Nexus OS, few show commands in Cisco IOS);
- **Show** commands executed on the device could display the data in JSON or XML format (Nexus OS, Junos).

It might be that other people already solved the problem:

- NAPALM (https://napalm.readthedocs.io/en/latest/) has tons of getters (https://napalm.readthedocs.io/en/latest/support/index.html) (routines that return data from the device in structured format) that can be called with napalm_get_facts (https://github.com/napalm-automation/napalm-ansible) Ansible module.
- NTC-Ansible (https://github.com/networktocode/ntc-ansible) is an extensible multi-vendor framework that uses TextFSM to parse printouts returned by **show** commands. It might be easier to extend NTC-Ansible than to try to hack your way around within Ansible.

Also, it's usually easier to write an Ansible module (in Python) or even an external script (if you're a Perl aficionado) than to do heavy lifting within an Ansible playbook.

Worst Case: Use a Hack

If you can't get the structured data from your device, can't use NAPALM or NTC-Ansible and don't have any programming skills (or can't find someone who would write a short Python or Perl program for you), try to use the hack described in the rest of this article.

Generate a printout on the network device

You have to generate a printout on your network device that has one line for every object you're interested in, and no extra headers/footers.

For example, to get a list of IP-enabled interfaces on Cisco IOS you could use **show ip interface brief**, but that printout includes extraneous information, including a header line and interfaces without IP addresses.

```
rtr#show ip interface brief
Interface        IP-Address      OK? Method Status          Protocol
FastEthernet0    unassigned      YES NVRAM  up              down
FastEthernet1    unassigned      YES NVRAM  up              up
Vlan1            10.0.0.1        YES NVRAM  up              up
Dialer3          unassigned      YES NVRAM  up              up
Dialer2          10.1.2.3        YES IPCP   up              up
```

To display only the interfaces with IP addresses configured, use an **include** filter that matches a sequence of digits (IP address) followed by one or more whitespaces followed by YES (interface is OK):

```
rtr#show ip interface brief | inc [0-9]+[ ]+YES
Vlan1            10.0.0.1        YES NVRAM  up              up
Dialer2          10.1.2.3        YES IPCP   up              up
```

Matching on a combination of digits and dots (IP address) is not good enough – that would also match subinterface names like FastEthernet0.1

After figuring out what **show** command and filter you need to use to get the desired printout execute the show command from an Ansible playbook and store the printouts in an Ansible variable with the **register** parameter.

```
- hosts: all
  name:  Check DHCP pools configured on the device
  tasks:
  - name: "Get interface list"
    ios_command:
      username:  "{{ansible_user}}"
      password:  "{{ansible_ssh_password}}"
      host:      "{{inventory_hostname}}"
      commands:
        - "show ip interface brief | inc \\.[0-9]+[ ]+YES"
    register: printout
```

If you're not familiar with Ansible playbook format or Ansible networking modules watch the Ansible for Networking Engineers (http://www.ipspace.net/Ansible_for_Networking_Engineers) webinar.

Finally, we need to parse the one-line-per-object printout to extract the information we're looking for (interface names). We'll use the Ansible regular expression Jinja2 filters to remove extraneous text or to extract the bits you need.

Using Regular Expression Filters (New in Ansible 2.2)

Ansible 2.2 added **regex_findall** and **regex_search** Jinja2 filters. The **regex_findall** returns a list of all matched regular expressions whereas **regex_search** takes additional arguments that allow you to specify which match groups you want to get (see Python Regular Expression Syntax (https://docs.python.org/2/library/re.html) for more details).

You can use the **regex** filters in Jinja2 templates or in **map** filter. For example, to extract interface names from **show ip interface brief** printout, use this Jinja2 expression:

```
- set_fact:
    intf: |
      {{printout.stdout_lines[0] |
        map('regex_findall','^([A-Za-z]+[0-9./]+)') |
        map('join') | list }}
```

Sounds complex? Let's figure out what the above expression is doing:

- **printout** is the variable that contains Cisco IOS command printouts (registered by the **ios_command** module);
- **printout.stdout_lines** is a list of printouts lines, one per executed command;
- **printout.stdout_lines[0]** is the list of lines of the first executed command (**show ip interfaces** in our example);
- **map** filter passes each element of the input list through a specified filter (**regex_findall**) using additional parameters (regular expression);
- **regex_findall** filter matches a string passed to it with a regular expression and returns a list of matches;
- **map** combined with **regex_findall** thus returns a list of lists – one list of matches per input line;
- **join** filter merges a list into a string. **join** used in a **map** merges every element of a list. In our example a list of lists is merged into a list of strings;
- **map** doesn't really return a list but a *generator* (code that can return a list). The **list** filter transforms a generator returned by **map** into a list.

End result: **intf** fact (variable) contains a list of interface names.

Here's another example using **regex_search** to extract DHCP pool names from **show ip dhcp pool** printout:

```
- set_fact:
    pool: |
      {{printout.stdout_lines[1] |
        map('regex_search','Pool\\s+(?P<id>.+)\\s+:','\\g<id>') |
        map('join') | list }}
```

The only difference between **regex_findall** and **regex_search** are the extra arguments that you can specify in **regex_search** after the regular expression. These arguments specify which parts of the matched regular expression you want to get in the output list (in our example, we wanted to get *named group* **id**).

For a complete example, explore the DHCP-Pools directory (https://github.com/ipspace/ansible-examples/tree/master/DHCP-Pools) in my Ansible examples repository (https://github.com/ipspace/ansible-examples)

Get Rid of Useless Parts of the Printout

Sometimes it's easiest to remove the extraneous parts of the printout by replacing them with an empty string.

For example, this is how you would extract the interface names from the printout by replacing everything after the first whitespace with an empty string:

```
- set_fact:
    intf: |
      {{printout.stdout_lines[0] |
        map('regex_replace','\\s+.*','') |
        list }}
```

Notes:

- **printout** is the variable in which we stored the output of the **show** command(s) with the **register** parameter.
- **printout.stdout_lines[0]** contains the printout from the first **show** command (**ios_command** module can execute multiple **show** commands at once).
- **printout.stdout_lines[0]** is really a list containing one line of printout in each element.
- **map** is a Jinja2 filter that executes the same operation (another Jinja2 filter) on every element of the list – our **map** filter will execute **regex_replace** on every line of the **show** command printout.
- **regex_replace** is a Jinja2 filter that replaces whatever its first argument matches with its second argument. In our case the first argument matches a whitespace followed by anything (double backslash has to be used due to YAML encoding rules) and replaces that with an empty string.
- **map** filter returns a *generator* which is useless for our purposes. **List** filter converts a generator into a list.

End result: the **intf** variable contains a list of interface names.

Extracting Bits of Data with regex_replace

The **regex_replace** filter allows you to grab something matched with a regular expression and use that in the replacement string. For example, this is how you would extract the interface IP addresses from the same printout:

```
- set_fact:
    addr: |
    {{printout.stdout_lines[0] |
      map('regex_replace','.*\\s+(?P<ip>[0-9.]+)\\s+.*','\\g<ip>') |
      list }}
```

Notes:

- The regular expression matches anything (beginning of line) followed whitespace followed by any sequence of digits and dots followed by another whitespace followed by anything (till the end of the line). It's important that your regular expression matches the whole line.
- The sequence of digits and dots is stored in a regular expression *group* (yes, you will have to read the Python Regular Expressions (https://docs.python.org/2/library/re.html) page to understand what's going on).
- The whole string (one line of the printout) is replaced by the value of group *ip* – the IP address we extracted with the regular expression.

Got this far? Congratulations. Why don't you take an extra step and register for the network automation online course (http://www.ipspace.net/Building_Network_Automation_Solutions)?

Also, you've seen how complex regular expressions can get. Try to use any other method before resorting to this particular Swiss Army Chainsaw (http://www.catb.org/jargon/html/S/Swiss-Army-chainsaw.html).