Loops

Often you'll want to do many things in one task, such as create a lot of users, install a lot of packages, or repeat a polling step until a certain result is reached.

This chapter is all about how to use loops in playbooks.

Topics

Standard Loops

To save some typing, repeated tasks can be written in short-hand like so:

```
- name: add several users
  user: name={{ item }} state=present groups=wheel
  with_items:
     - testuser1
     - testuser2
```

If you have defined a YAML list in a variables file, or the ‘vars’ section, you can also do:

```
with_items: somelist
```

The above would be the equivalent of:

```
- name: add user testuser1
  user: name=testuser1 state=present groups=wheel
```

```
- name: add user testuser2
  user: name=testuser2 state=present groups=wheel
```

The yum and apt modules use with_items to execute fewer package manager transactions.

Note that the types of items you iterate over with 'with_items' do not have to be simple lists of strings. If you have a list of hashes, you can reference subkeys using things like:

```
- name: add several users
  user: name={{ item.name }} state=present groups={{ item.groups }}
  with_items:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

Also be aware that when combining *when* with *with_items* (or any other loop statement), the *when* statement is processed separately for each item. See The When Statement for an example.

Nested Loops

Loops can be nested as well:

```
- name: give users access to multiple databases
  mysql_user: name={{ item[0] }} priv={{ item[1] }}.*:ALL append_privs=yes password=foo
  with_nested:
    - [ 'alice', 'bob' ]
    - [ 'clientdb', 'employeedb', 'providerdb' ]
```

As with the case of 'with_items' above, you can use previously defined variables.

Just specify the variable's name without templating it with '{{ }}':

```
- name: here, 'users' contains the above list of employees
  mysql_user: name={{ item[0] }} priv={{ item[1] }}.*:ALL append_privs=yes password=foo
  with_nested:
    - users
    - [ 'clientdb', 'employeedb', 'providerdb' ]
```

Looping over Hashes

New in version 1.5.

Suppose you have the following variable:

```
---
users:
  alice:
    name: Alice Appleworth
    telephone: 123-456-7890
  bob:
    name: Bob Bananarama
    telephone: 987-654-3210
```

And you want to print every user's name and phone number. You can loop through the elements of a hash using `with_dict` like this:

```
tasks:
  - name: Print phone records
    debug: msg="User {{ item.key }} is {{ item.value.name }} ({{ item.value.telephone }})"
    with_dict: users
```

Looping over Fileglobs

`with_fileglob` matches all files in a single directory, non-recursively, that match a pattern. It can be used like this:

```
---
- hosts: all

  tasks:

    # first ensure our target directory exists
    - file: dest=/etc/fooapp state=directory

    # copy each file over that matches the given pattern
    - copy: src={{ item }} dest=/etc/fooapp/ owner=root mode=600
      with_fileglob:
        - /playbooks/files/fooapp/*
```

> **Note**
> When using a relative path with `with_fileglob` in a role, Ansible resolves the path relative to the *roles/<rolename>/files* directory.

Looping over Parallel Sets of Data

> **Note**
> This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

Suppose you have the following variable data was loaded in via somewhere:

```
---
alpha: [ 'a', 'b', 'c', 'd' ]
numbers:  [ 1, 2, 3, 4 ]
```

And you want the set of '(a, 1)' and '(b, 2)' and so on. Use 'with_together' to get this:

```
tasks:
    - debug: msg="{{ item.0 }} and {{ item.1 }}"
      with_together:
        - alpha
        - numbers
```

Looping over Subelements

Suppose you want to do something like loop over a list of users, creating them, and allowing them to login by a certain set of SSH keys.

How might that be accomplished? Let's assume you had the following defined and loaded in via "vars_files" or maybe a "group_vars/all" file:

```
---
users:
  - name: alice
    authorized:
      - /tmp/alice/onekey.pub
      - /tmp/alice/twokey.pub
  - name: bob
    authorized:
      - /tmp/bob/id_rsa.pub
```

It might happen like so:

```
- user: name={{ item.name }} state=present generate_ssh_key=yes
  with_items: users

- authorized_key: "user={{ item.0.name }} key='{{ lookup('file', item.1) }}'"
  with_subelements:
     - users
     - authorized
```

Subelements walks a list of hashes (aka dictionaries) and then traverses a list with a given key inside of those records.

The authorized_key pattern is exactly where it comes up most.

Looping over Integer Sequences

`with_sequence` generates a sequence of items in ascending numerical order. You can specify a start, end, and an optional step value.

Arguments should be specified in key=value pairs. If supplied, the 'format' is a printf style string.

Numerical values can be specified in decimal, hexadecimal (0x3f8) or octal (0600). Negative numbers are not supported. This works as follows:

```
---
- hosts: all

  tasks:
```

```
    # create groups
    - group: name=evens state=present
    - group: name=odds state=present

    # create some test users
    - user: name={{ item }} state=present groups=evens
      with_sequence: start=0 end=32 format=testuser%02x

    # create a series of directories with even numbers for some reason
    - file: dest=/var/stuff/{{ item }} state=directory
      with_sequence: start=4 end=16 stride=2

    # a simpler way to use the sequence plugin
    # create 4 groups
    - group: name=group{{ item }} state=present
      with_sequence: count=4
```

Random Choices

The ‘random_choice’ feature can be used to pick something at random. While it’s not a load balancer (there are modules for those), it can somewhat be used as a poor man’s loadbalancer in a MacGyver like situation:

```
- debug: msg={{ item }}
  with_random_choice:
     - "go through the door"
     - "drink from the goblet"
     - "press the red button"
     - "do nothing"
```

One of the provided strings will be selected at random.

At a more basic level, they can be used to add chaos and excitement to otherwise predictable automation environments.

Do-Until Loops

Sometimes you would want to retry a task until a certain condition is met. Here's an example:

```
- action: shell /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

The above example run the shell module recursively till the module's result has "all systems go" in its stdout or the task has been retried for 5 times with a delay of 10 seconds. The default value for "retries" is 3 and "delay" is 5.

The task returns the results returned by the last task run. The results of individual retries can be viewed by -vv option. The registered variable will also have a new key "attempts" which will have the number of the retries for the task.

Finding First Matched Files

> **Note**
> This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

This isn't exactly a loop, but it's close. What if you want to use a reference to a file based on the first file found that matches a given criteria, and some of the filenames are determined by variable names? Yes, you can do that as follows:

```
- name: INTERFACES | Create Ansible header for /etc/network/interfaces
  template: src={{ item }} dest=/etc/foo.conf
  with_first_found:
    - "{{ansible_virtualization_type}}_foo.conf"
    - "default_foo.conf"
```

This tool also has a long form version that allows for configurable search paths. Here's an example:

```
- name: some configuration template
  template: src={{ item }} dest=/etc/file.cfg mode=0444 owner=root group=root
  with_first_found:
    - files:
       - "{{inventory_hostname}}/etc/file.cfg"
      paths:
       - ../../../templates.overwrites
       - ../../../templates
    - files:
        - etc/file.cfg
      paths:
        - templates
```

Iterating Over The Results of a Program Execution

> **Note**
> This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

Sometimes you might want to execute a program, and based on the output of

that program, loop over the results of that line by line. Ansible provides a neat way to do that, though you should remember, this is always executed on the control machine, not the local machine:

```
- name: Example of looping over a command result
  shell: /usr/bin/frobnicate {{ item }}
  with_lines: /usr/bin/frobnications_per_host --param {{ inventory_hostname }}
```

Ok, that was a bit arbitrary. In fact, if you're doing something that is inventory related you might just want to write a dynamic inventory source instead (see *Dynamic Inventory*), but this can be occasionally useful in quick-and-dirty implementations.

Should you ever need to execute a command remotely, you would not use the above method. Instead do this:

```
- name: Example of looping over a REMOTE command result
  shell: /usr/bin/something
  register: command_result

- name: Do something with each result
  shell: /usr/bin/something_else --param {{ item }}
  with_items: command_result.stdout_lines
```

Looping Over A List With An Index

> **Note**
> This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

If you want to loop over an array and also get the numeric index of where you are in the array as you go, you can also do that. It's uncommonly used:

```
- name: indexed loop demo
  debug: msg="at array position {{ item.0 }} there is a value {{ item.1 }}"
  with_indexed_items: some_list
```

Flattening A List

> **Note**
> This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

In rare instances you might have several lists of lists, and you just want to iterate over every item in all of those lists. Assume a really crazy hypothetical datastructure:

```
---
# file: roles/foo/vars/main.yml
packages_base:
  - [ 'foo-package', 'bar-package' ]
packages_apps:
  - [ ['one-package', 'two-package' ]]
  - [ ['red-package'], ['blue-package']]
```

As you can see the formatting of packages in these lists is all over the place. How can we install all of the packages in both lists?:

```
- name: flattened loop demo
  yum: name={{ item }} state=installed
  with_flattened:
     - packages_base
     - packages_apps
```

That's how!

Using register with a loop

When using `register` with a loop the data structure placed in the variable during a loop, will contain a `results` attribute, that is a list of all responses from the module.

Here is an example of using `register` with `with_items`:

```
- shell: echo "{{ item }}"
  with_items:
    - one
    - two
  register: echo
```

This differs from the data structure returned when using `register` without a loop:

```
{
    "changed": true,
    "msg": "All items completed",
    "results": [
        {
            "changed": true,
            "cmd": "echo \"one\" ",
            "delta": "0:00:00.003110",
            "end": "2013-12-19 12:00:05.187153",
            "invocation": {
```

```
                "module_args": "echo \"one\"",
                "module_name": "shell"
            },
            "item": "one",
            "rc": 0,
            "start": "2013-12-19 12:00:05.184043",
            "stderr": "",
            "stdout": "one"
        },
        {
            "changed": true,
            "cmd": "echo \"two\" ",
            "delta": "0:00:00.002920",
            "end": "2013-12-19 12:00:05.245502",
            "invocation": {
                "module_args": "echo \"two\"",
                "module_name": "shell"
            },
            "item": "two",
            "rc": 0,
            "start": "2013-12-19 12:00:05.242582",
            "stderr": "",
            "stdout": "two"
        }
    ]
}
```


Subsequent loops over the registered variable to inspect the results may look like:

```
- name: Fail if return code is not 0
  fail:
    msg: "The command ({{ item.cmd }}) did not have a 0 return code"
  when: item.rc != 0
  with_items: echo.results
```

Writing Your Own Iterators

While you ordinarily shouldn't have to, should you wish to write your own ways to loop over arbitrary datastructures, you can read *Developing Plugins* for some starter information. Each of the above features are implemented as plugins in ansible, so there are many implementations to reference.

See also

Playbooks
An introduction to playbooks

Playbook Roles and Include Statements
Playbook organization by roles

Best Practices
Best practices in playbooks

Conditionals
Conditional statements in playbooks

Variables
All about variables

User Mailing List
Have a question? Stop by the google group!

irc.freenode.net
#ansible IRC chat channel

Previous Next