


原创

自动化运维工具Ansible实战（五）Playbooks剧本使用

 小左先森 [关注](#)

2018-05-04 15:40:07 219759人阅读 0人评论

一、Playbook 简介

Playbooks与Ad-Hoc相比，是一种完全不同的运用Ansible的方式，而且是非常之强大的；也是系统ansible命令的集合，其利用yaml语言编写，运行过程，ansible-playbook命令根据自上而下的顺序依次执行。

简单来说，Playbooks 是一种简单的配置管理系统与多机器部署系统的基础。与现有的其他系统有不同之处，且非常适合于复杂应用的部署。

同时，Playbooks开创了很多特性，它可以允许你传输某个命令的状态到后面的指令，如你可以从一台机器的文件中抓取内容并附为变量，然后在另一台机器中使用，这使得你可以实现一些复杂的部署机制，这是ansible命令无法实现的。

Playbooks可用于声明配置，更强大的地方在于，在Playbooks中可以编排有序的执行过程，甚至于做到在多组机器间，来回有序的执行特别指定的步骤。并且可以同步或异步的发起任务。

我们使用Ad-Hoc时，主要是使用 /usr/bin/ansible 程序执行任务.而使用Playbooks时，更多是将之放入源码控制之中，用之推送你的配置或是用于确认你的远程系统的配置是否符合配置规范。

在如右的链接中：[ansible-examples repository](#)，有一些整套的Playbooks，它们阐明了上述的这些技巧。



二、Playbook 语言的示例

playbooks 的格式是yaml，语法做到最小化，意在避免 playbooks 成为一种编程语言或是脚本，但它也并不是一个配置模型或过程的模型。

playbook是由一个或多个“play”组成的列表。play的主要功能在于将事先归并为一组的主机装扮成事先通过Ansible中的tasks定义好的角色（play的内容被称为tasks，即任务）。从根本上来讲所谓tasks无非是调用Ansible的一个module。将多个“play”组织在一个playbook中即可以让它们联同起来按事先编排的机制一同工作。

“plays”算是一个类比，可以通过多个plays告诉系统做不同的事情，不仅是定义一种特定的状态或模型。也可以在不同时间运行不同的plays。

对于初学者，这里有一个playbook，其中仅包含一个play：

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
    - name: ensure apache is at the latest version
      yum: pkg=httpd state=latest
    - name: write the apache config file
      template: src=/srv/httpd.j2 dest=/etc/httpd.conf
      notify:
        - restart_anache
```

[在线客服](#)

```
- name: restart apache
  service: name=httpd state=restarted
```

第一行中，文件开头为 `---`；这是YAML将文件解释为正确的文档的要求。YAML允许多个“文档”存在于一个文件中，每个“文档”由 `---` 符号分割，但Ansible只需要一个文件存在一个文档即可，因此这里需要存在于文件的开始行第一行。

YAML对空格非常敏感，并使用空格来将不同的信息分组在一起，在整个文件中应该只使用空格而不使用制表符，并且必须使用一致的间距，才能正确读取文件。相同缩进级别的项目被视为同级元素。

以 `-` 开头的项目被视为列表项目。作为散列或字典操作，它具有 `key: value` 格式的项。YAML文档基本上定义了一个分层的树结构，其中位于左侧是包含的元素。YAML文件扩展名通常为 `.yaml` 或者 `.yml`。

接下来，我们将分别讲解 `playbook` 语言的多个特性

三、Playbook 构成

Playbook主要有以下四部分构成：

1. target section：定义将要执行playbook的远程主机组
2. variable section：定义playbook运行时需要使用的变量
3. task section：定义将要在远程主机上执行的任务列表
4. handler section：定义task执行完成以后需要调用的任务

而Playbook对应的目录层有五个，分别如下：

一般所需的目录层有：(视情况可变化)

1. vars 变量层
2. tasks 任务层
3. handlers 触发条件
4. files 文件
5. template 模板

接下来，我们将分别介绍构成playbook的四层结构。



1、Hosts（主机）与Users（用户）

我们可以为playbook中的每一个play，个别的选择操作的目标机器是哪些，以哪个用户身份去完成要执行的步骤（called tasks）

```
---
- hosts: webservers
  remote_user: root
  tasks:
    - name: test connection
      remote_user: yourname
      sudo: yes
```

playbook中的每一个play的目的都是为了让某个或某些主机以某个指定的用户身份执行任务。

hosts：用于指定要执行指定任务的主机其可以是一个或多个,由逗号为分隔符分隔主机组

remote_user：用于指定远程主机上的执行任务的用户。不过remote_user也可用于各tasks中。也可以通过指定其通过sudo的方式在远程主机上执行任务其可用于play全局或某任务。此外甚至可以在sudo时使用sudo_user指定sudo时切换的用户

user：与remote_user相同

sudo：如果设置为yes，执行该任务组的用户在执行任务的时候，获取root权限

sudo_user：如果设置user为bob，sudo为yes，sudo_user为tom时，则bob用户在执行任务时会获得tom用户的权限

connection：通过什么方式连接到远程主机，默认为ssh

gather_facts：除非明确说明不需要在远程主机上执行setup模块，否则默认自动执行。如果确实不需要setup模块传递过来的变量，则可以将该选项设置为False

在线
客服

- 参数remote_user以前写做user，在Ansible 1.4以后才改为remote_user。主要为了不跟user模块混淆（user模块用于在远程系统上创建用户）
- 如果我们需要在使用sudo时指定密码，可在运行ansible-playbook命令时加上选项 --ask-sudo-pass (-K)。如果使用sudo时，playbook疑似被挂起，可能是在sudo prompt处被卡住，这时可执行 Control-C(正文结束字符)杀死卡住的任务，再重新运行一次。
- 当使用sudo_user切换到非root用户时，模块的参数会暂时写入/tmp目录下的一个随机临时文件。当命令执行结束后，临时文件立即删除。这种情况发生在普通用户的切换时，比如从“bob”切换到“tom”，切换到root账户时，不会发生，如从“bob”切换到“root”，直接以普通用户或root身份登录也不会发生。如果不希望这些数据在短暂的时间内可以被读取（不可写），请避免在sudo_user中传递未加密的密码。其它情况下，“/tmp”目录不被使用，这种情况不会发生。Ansible 也有意识的在日志中不记录密码参数

2、tasks 列表和 action

每一个play包含了一个tasks列表（任务列表）。

任务列表中的各任务按次序逐个在hosts中指定的所有主机上执行即在所有主机上完成第一个任务后再开始第二个。在自上而下运行某playbook时如果中途发生错误，所有已执行任务都将回滚，因此在更正playbook后重新执行即可。

每一个tasks必须有一个名称name，这样在运行playbook时，从其输出的任务执行信息中可以很好的辨别出是属于哪一个tasks的。如果没有定义name，“action”的值将会用作输出信息中标记特定的tasks。

tasks的目的是使用指定的参数执行模块，而在模块参数中可以使用变量。模块执行是幂等的，这意味着多次执行是安全的，因为其结果均一致。每个task都应该有其name用于playbook的执行结果输出，建议其内容尽可能清晰地描述任务执行步骤。如果未提供name则action的结果将用于输出。

如果要声明一个tasks，以前有一种格式：“action: module options”（可能在一些老的playbooks中还能见到）。现在推荐使用更常见的格式：“module: options”。

下面是一种基本的task的定义，service module使用key=value格式的的参数，这也是大多数module使用的参数格式：

```
tasks:
  - name: make sure apache is running
    service: name=httpd state=running

# 在众多模块中，只有command和shell模块仅需要给定一个列表而无需使用“key=value”格式如下：
tasks:
  - name: disable selinux
    command: /sbin/setenforce 0

# 使用command module和shell module时，我们需要关心返回码信息，如果有一条命令，它的成功执行的返回码不是
tasks:
  - name: run this command and ignore the result
    shell: /usr/bin/somecommand || /bin/true
# 或者使用ignore_errors来忽略错误信息
tasks:
  - name: run this command and ignore the result
    shell: /usr/bin/somecommand
    ignore_errors: True

# 如果action行看起来太长，可以使用space（空格）或者indent（缩进）隔开连续的一行：
tasks:
  - name: Copy ansible inventory file to client
    copy: src=/etc/ansible/hosts dest=/etc/ansible/hosts
      owner=root group=root mode=0644
```



3、Handlers 在发生改变时执行的操作

上面我们曾提到过，module具有“幂等”性，所以当远程主机被人改动时，可以重放playbooks达到恢复的目的。playbooks本身可以识别这种改动，并且有一个基本的事件系统（事件系统），可以响应这种改动。

（当发生改动时）“notify”这个actions会在playbook的每一个tasks结束时被触发，而且即使有多个不同的tasks通知改动的发生，“notify” actions只会被触发一次。这样可以避免多次有改变发生时每次都执行指定的

在线
客服

说明：

在“notify”中定义内容一定要和tasks中定义的 - name 内容一样，这样才能达到触发的效果，否则会不生效。

举例来说，比如多个resources指出因为一个配置文件被改动，所以apache需要重新启动，但是重新启动的操作只会被执行一次。

这里有一个例子，当一个文件的内容被改动时，重启两个services：

```
- name: template configuration file
  template: src=template.j2 dest=/etc/foo.conf
  notify:
    - restart memcached
    - restart apache
```

“notify”下列出的即是 handlers。

handlers也是一些tasks的列表，通过名字来引用，它们和一般的tasks并没有什么区别。handlers是由通知者进行notify，如果没有被notify，handlers不会执行。不管有多少个通知者进行了notify，等到play中的所有tasks执行完成之后，handlers也只會被执行一次。

这里是一个 handlers 的示例：

```
handlers:
  - name: restart memcached
    service: name=memcached state=restarted
  - name: restart apache
    service: name=apache state=restarted
```

Handlers最佳的应用场景是用来重启服务，或者触发系统重启操作。除此以外很少用到了。

说明：handlers会按照声明的顺序执行。

4、tags

tags用于让用户选择运行或略过playbook中的部分代码。ansible具有幂等性，因此会自动跳过没有变化的部分，即便如此，有些代码为测试其确实没有发生变化的时间依然会非常的长。此时如果确信其没有变化就可以通过tags跳过这些代码片断。



5、示例

通过playbook添加用户示例

(1) 给远程主机添加用户test

```
[root@Ansible ~]# vim user.yml
---
- name: create user
  hosts: all
  user: root
  gather_facts: False
  vars:
    - user: test
  tasks:
    - name: create user
      user: name={{ user }}
```

上面的playbook 实现的功能是新增一个用户：

name：对该playbook实现的功能做一个概述，后面执行过程中，会打印name变量的值

hosts：指定了对哪些主机进行参作

user：指定了使用什么用户登录远程主机操作

gather_facts：指定了在以下任务部分执行前，是否先执行setup模块获取主机相关信息，这在后面的task会使用到setup获取的信息时用到

vars：指定了变量，这里指定了一个user变量，其值为test，需要注意的是，变量值一定要用引号引起

在线
客服

tasks: 指定了一个任务，其下面的name参数同样是对任务的描述，在执行过程中会打印出来。user指定了调用user模块，name是user模块里的一个参数，而增加的用户名字调用了上面user变量的值

查看执行结果如下：

```
[root@Ansible ~]# ansible-playbook user.yml
```

```
[root@Ansible ~]# ansible-playbook user.yml

PLAY [create user] *****

TASK [create user] *****
changed: [192.168.8.66]

PLAY RECAP *****
192.168.8.66 : ok=1 changed=1 unreachable=0 failed=0

[root@Ansible ~]# ansible all -m command -a "id test"
192.168.8.66 | SUCCESS | rc=0 >>
uid=1003(test) gid=1003(test) 组=1003(test) 创建test用户
```

@51CTO博客

(2) 删除远程主机test的账号

只需user: name="{{ user }}" state=absent remove=yes 即可

```
[root@Ansible ~]# vim user.yml
---
- name: create user
  hosts: all
  user: root
  gather_facts: False
  vars:
    - user: test
  tasks:
    - name: create user
      user: name={{ user }} state=absent remove=yes
[root@Ansible ~]# ansible all -m command -a "id test"
```

```
[root@Ansible ~]# ansible-playbook user.yml

PLAY [create user] *****

TASK [create user] *****
changed: [192.168.8.66]

PLAY RECAP *****
192.168.8.66 : ok=1 changed=1 unreachable=0 failed=0

[root@Ansible ~]# ansible all -m command -a "id test"
192.168.8.66 | FAILED | rc=1 >>
id: test: no such usernon-zero return code
```



@51CTO博客

(3) 通过playbook 安装apache

安装 httpd 服务，将本地准备好的配置文件 copy 过去，并且启动服务

```
[root@Ansible ~]# vim apache.yml
---
- hosts: all
  remote_user: root
  gather_facts: False
  tasks:
    - name: install apache on CentOS 7
      yum: name=httpd state=present
    - name: copy httpd conf
      copy: src=/etc/httpd/conf/httpd.conf dest=/etc/httpd/conf/httpd.conf
    - name: start apache service
      service: name=httpd state=started
[root@Ansible ~]# ansible-playbook apache.yml
[root@Ansible ~]# ansible all -m shell -a "lsof -i:80"
```

在线
客服

```
[root@Ansible ~]# vim apache.yml
[root@Ansible ~]# ansible-playbook apache.yml

PLAY [all] *****

TASK [install apache on CentOS 7] *****
changed: [192.168.8.66]

TASK [copy httpd conf] *****
ok: [192.168.8.66]

TASK [start apache] *****
changed: [192.168.8.66]

PLAY RECAP *****
192.168.8.66 : ok=3 changed=2 unreachable=0 failed=0

[root@Ansible ~]# ansible all -m shell -a "lsof -i:80"
192.168.8.66 | SUCCESS | rc=0 >>
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
httpd 52285 root 4u IPv6 948202 0t0 TCP *:http (LISTEN)
httpd 52286 apache 4u IPv6 948202 0t0 TCP *:http (LISTEN)
httpd 52287 apache 4u IPv6 948202 0t0 TCP *:http (LISTEN)
httpd 52288 apache 4u IPv6 948202 0t0 TCP *:http (LISTEN)
httpd 52289 apache 4u IPv6 948202 0t0 TCP *:http (LISTEN)
httpd 52290 apache 4u IPv6 948202 0t0 TCP *:http (LISTEN)
```

@51CTO博客

(4) 通过playbook 安装apache (修改端口, 并带有vars变量)

将httpd.conf监听的端口改为8080, 然后重新覆盖配置文件, 当这个配置文件发生改变时, 就触发handler进行服务重启

notify 这个 action可用于在每个play的最后被触发, 这样可以避免多次有改变发生时每次都执行指定的操作, notify中列出的操作称为handler

```
[root@Ansible ~]# vim apache.yml
---
- hosts: all
  remote_user: root
  gather_facts: False
  vars:
    src_http_dir: /etc/httpd/conf
    dest_http_dir: /etc/httpd/conf
  tasks:
    - name: install apache on CentOS 7
      yum: name=httpd state=present
    - name: copy httpd conf
      copy: src={{ src_http_dir }}/httpd.conf dest={{ dest_http_dir }}/httpd.conf
      notify:
        - systemctl restart httpd
    - name: start apache service
      service: name=httpd state=restarted
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
[root@Ansible ~]# vim /etc/httpd/conf/httpd.conf
42 Listen 80 --》 修改为8080
[root@Ansible ~]# ansible-playbook apache.yml
[root@Ansible ~]# ansible all -m shell -a "lsof -i:8080"
```



```
[root@Ansible ~]# vim apache.yml
[root@Ansible ~]# vim /etc/httpd/conf/httpd.conf
[root@Ansible ~]# ansible-playbook apache.yml

PLAY [all] *****

TASK [install apache on CentOS 7] *****
ok: [192.168.8.66]

TASK [copy httpd conf] *****
ok: [192.168.8.66]

TASK [start apache service] *****
changed: [192.168.8.66]

PLAY RECAP *****
192.168.8.66 : ok=3 changed=1 unreachable=0 failed=0

[root@Ansible ~]# ansible all -m shell -a "lsof -i:8080"
192.168.8.66 | SUCCESS | rc=0 >>
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
httpd 53724 root 4u IPv6 957249 0t0 TCP *:webcache (LISTEN)
httpd 53725 apache 4u IPv6 957249 0t0 TCP *:webcache (LISTEN)
httpd 53726 apache 4u IPv6 957249 0t0 TCP *:webcache (LISTEN)
httpd 53727 apache 4u IPv6 957249 0t0 TCP *:webcache (LISTEN)
httpd 53728 apache 4u IPv6 957249 0t0 TCP *:webcache (LISTEN)
httpd 53729 apache 4u IPv6 957249 0t0 TCP *:webcache (LISTEN)
```

@51CTO博客

在线
客服

四、Playbook 常用模块

Playbook的模块与在Ansible命令行下使用的模块有一些不同。这主要是因为playbook中会使用到一些facts变量和一些通过setup模块从远程主机上获取到的变量。有些模块没法在命令行下运行，就是因为它们需要这些变量。而且即使那些可以在命令行下工作的模块也可以通过playbook的模块获取一些更高级的功能。

具体模块可参考官网 (http://docs.ansible.com/ansible/latest/list_of_all_modules.html)。

这里从官方分类的模块里选择最常用的一些模块进行介绍。

1、template模块

(1) 简介

- 在实际应用中，我们的配置文件有些地方可能会根据远程主机的配置的不同而有稍许的不同，template可以使用变量来接收远程主机上setup收集到的facts信息，针对不同配置的主机，定制配置文件。用法大致与copy模块相同
- template_host包含模板机器的节点名称
- template_uid所有者的数字用户标识
- template_path是模板的路径
- template_fullpath是模板的绝对路径
- template_run_date是模板呈现的日期

(2) 参数

attributes(2.3后新增): 文件或目录的属性
backup: 如果原目标文件存在，则先备份目标文件
block_end_string(2.4后新增): 标记块结束的字符串
block_start_string(2.4后新增): 标记块的开始的字符串
dest: 目标文件路径
follow(2.4后新增): 是否遵循目标中的文件链接
force: 是否强制覆盖，默认为yes
group: 目标文件或目录的所属组
owner: 目标文件或目录的所属主
mode: 目标文件的权限。对于那些习惯于/usr/bin/chmod的记住，模式实际上是八进制数字（如0644或01777）。离开前导零可能会有意想不到的结果。从版本1.8开始，可以将模式指定为符号模式（例如u+rw或u=rw,g=r,o=r）
newline_sequence(2.4后新增): 指定用于模板文件的换行符序列（\n、\r、\r\n）
src: 源模板文件路径
trim_blocks(2.4后新增): 如果这设置为True，则删除块后的第一个换行符
validate: 在复制之前通过命令验证目标文件，如果验证通过则复制
variable_end_string(2.4后新增): 标记打印语句结束的字符串
variable_start_string(2.4后新增): 标记打印语句开头的字符串



(3) 示例

```
# 官方简单示例
- template: src=/mytemplates/foo.j2 dest=/etc/file.conf owner=bin group=wheel mode=0644
- template: src=/mytemplates/foo.j2 dest=/etc/file.conf owner=bin group=wheel mode="u=rw,g=r,o=r"
- template: src=/mine/sudoers dest=/etc/sudoers validate='sudo -cf %s'

playbook的引用该模板配置文件的方法示例:
- name: Setup BIND
  host: all
  tasks:
    - name: configure BIND
      template: src=/etc/httpd/conf/httpd.conf dest=/etc/httpd/conf/httpd.conf owner=root group=root
# 或者是这样
- template:
  src: /etc/httpd/conf/httpd.conf
```

在线
客服

mode: 0644

创建DOS样式文本文件

```
- template:
  src: config.ini.j2
  dest: /share/windows/config.ini
  newline_sequence: '\r\n'
```

2、set_fact模块

(1) 简介

- set_fact模块可以自定义facts，这些自定义的facts可以通过template或者变量的方式在playbook中使用
- 如果你想要获取一个进程使用的内存的百分比，则必须通过set_fact来进行计算之后得出其值，并将其值在playbook中引用

(2) 参数

cacheable(2.4后新增): 可缓存

key_value: 该set_fact模块将key=value作为变量，设置在剧本范围中

(3) 示例

配置mysql innodb buffer size的示例

```
[root@Ansible ~]# echo "# Configure the buffer pool" >> /etc/my.cnf
[root@Ansible ~]# echo "innodb_buffer_pool_size = {{ innodb_buffer_pool_size_mb|int }}M" >> /etc/my.cnf
[root@Ansible ~]# vim set_fact.yml
---
- name: Configure Mariadb
  hosts: all
  tasks:
    - name: install Mariadb
      yum: name=mariadb-server state=installed

    - name: Calculate InnoDB buffer pool size
      set_fact: innodb_buffer_pool_size_mb={{ ansible_memtotal_mb / 2 }}
    - name: Configure Mariadb
      template: src=/etc/my.cnf dest=/etc/my.cnf owner=root group=root mode=0644

    - name: Start Mariadb
      service: name=mariadb state=started enabled=yes
  handlers:
    - name: restart Mariadb
      service: name=mariadb state=restarted
[root@Ansible ~]# ansible-playbook set_fact.yml
```



```
[root@Ansible ~]# echo "# Configure the buffer pool" >> /etc/my.cnf
[root@Ansible ~]# echo "innodb_buffer_pool_size = {{ innodb_buffer_pool_size_mb|int }}M" >> /etc/my.cnf
[root@Ansible ~]# vim set_fact.yml
[root@Ansible ~]# ansible-playbook set_fact.yml

PLAY [Configure Mariadb] *****

TASK [Gathering Facts] *****
ok: [192.168.8.66]

TASK [install Mariadb] *****
changed: [192.168.8.66]

TASK [Calculate InnoDB buffer pool size] *****
ok: [192.168.8.66]

TASK [Configure Mariadb] *****
changed: [192.168.8.66]

TASK [Start Mariadb] *****
changed: [192.168.8.66]

PLAY RECAP *****
192.168.8.66 : ok=5 changed=3 unreachable=0 failed=0 @51CTO博客
```

在线
客服

在远程主机上查看该配置

```
[root@Client ~]# vim /etc/my.cnf
```



```

[mysqld]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock
# Disabling symbolic-links is recommended to prevent assorted security risks
symbolic-links=0
# Settings user and group are ignored when systemd is used.
# If you need to run mysqld under a different user or group,
# customize your systemd unit file for mariadb according to the
# instructions in http://fedoraproject.org/wiki/Systemd

[mysqld_safe]
log-error=/var/log/mariadb/mariadb.log
pid-file=/var/run/mariadb/mariadb.pid

#
# include all files from the config directory
#
!includedir /etc/my.cnf.d
# Configure the buffer pool
innodb_buffer_pool_size = 1504M
~
~
~

```

@51CTO博客

3、pause模块

(1) 简介

- 在playbook执行的过程中暂停一定时间或者提示用户进行某些操作
- 要为每个主机暂停、等待、休眠，可以使用wait_for模块
- 如果您想提前暂停而不是设置为过期，或者您需要完全中止剧本运行，则可以使用Ctrl + c。要继续提前按ctrl + c然后c。要中止一个剧本按ctrl + c然后a
- 该模块也支持Windows目标

(2) 参数

echo(2.5后增加)：控制键入时是否显示键盘输入。如果设置了“秒”或“分钟”，则不起作用

minutes：暂停多少分钟

seconds：暂停多少秒

prompt：打印一串信息提示用户操作



(3) 示例

暂停5分钟以建立应用程序缓存。

```
- pause:
  minutes: 5
```

有用的提醒，提醒您注意更新后的内容。

```
- pause:
  prompt: "Make sure org.foo.FooOverload exception is not present"
```

暂停以后获得一些信息。

```
- pause:
  prompt: "Enter a secret"
  echo: no
```

4、wait_for模块

(1) 简介

- wait_for模块是在playbook的执行过程中，等待某些操作完成以后再进行后续操作

(2) 参数

active_connection_states(2.3后新增)：被计为活动连接的TCP连接状态列表

connect_timeout：在下一个任务执行之前等待连接的超时时间

delay：等待一个端口或者文件或者连接到指定的状态时，默认超时时间为300秒，在这等待的300s的时间里，wait_for模块会一直轮询指定的对象是否到达指定的状态，delay即为多长时间轮询一次状态

exclude_hosts(1.8后新增)：在查找状态的活动TCP连接时要忽略的主机或IP的列表drained

在线
客服

port: wait_for模块等待的主机的端口
path: 文件路径, 只有当这个文件存在时, 下一任务才开始执行, 即等待该文件创建完成
search_regex(1.4后新增): 可以用来匹配文件或套接字连接中的字符串。默认为多行正则表达式
sleep(2.3后新增): 检查之间睡眠的秒数, 在2.3之前, 这被硬编码为1秒
state: 等待的状态, 即等待的文件或端口或者连接状态达到指定的状态时, 下一个任务开始执行。当等的对象为端口时, 状态有started, stoped, 即端口已经监听或者端口已经关闭; 当等待的对象为文件时, 状态有present或者started, absent, 即文件已创建或者删除; 当等待的对象为一个连接时, 状态有drained, 即连接已建立。默认为started
timeout: wait_for的等待的超时时间, 默认为300秒

(3) 示例

```
- name: create task
  hosts: all
  tasks:
    # 等待8080端口已正常监听, 才开始下一个任务, 直到超时
    - wait_for: port=8080 state=started
    # 等待8000端口正常监听, 每隔10s检查一次, 直至等待超时
    - wait_for: port=8081 delay=10
    # 等待8000端口直至有连接建立
    - wait_for: host=0.0.0.0 port=8000 delay=10 state=drained
    # 等待8000端口有连接建立, 如果连接来自192.168.8.66或者192.168.8.8, 则忽略
    - wait_for: host=0.0.0.0 port=8000 state=drained exclude_hosts=192.168.8.66,192.168.8.8
    # 等待/tmp/foo文件已创建
    - wait_for: path=/tmp/foo
    # 等待/tmp/foo文件已创建, 而且该文件中需要包含completed字符串
    - wait_for: path=/tmp/foo search_regex=completed
    # 等待/var/lock/file.lock被删除
    - wait_for: path=/var/lock/file.lock state=absent
    # 等待指定的进程被销毁
    - wait_for: path=/proc/3466/status state=absent
    # 等待openssh启动, 10s检查一次
    - local_action: wait_for port=22 host="{{ ansible_ssh_host | default(inventory_hostname) }}" sea
```

5、assemble模块

(1) 简介

- assemble模块用于组装文件, 即将多个零散的文件(称为碎片), 合并一个目标文件

(2) 参数

attributes(2.3后新增): 文件或目录的属性
backup: 创建一个备份文件(如果yes), 包括时间戳信息
decrypt(2.4后新增): 控制使用保管库对源文件进行自动解密
delimiter(1.4后新增): 分隔文件内容的分隔符
dest: 使用所有源文件的连接创建的文件, 合并后的大文件路径
group: 合并后的大文件的所属组
owner: 合并后的大文件的所属主
ignore_hidden(2.0后新增): 组装时, 是否忽略隐藏文件, 默认为no
mode: 合并后的大文件的权限。对于那些习惯于/usr/bin/chmod的记住, 模式实际上是八进制数字(如0644或01777)。离开前导零可能会有意想不到的结果。从版本1.8开始, 可以将模式指定为符号模式(例如u+rw或u=rw,g=r,o=r)
regexp: 在regex匹配文件名时汇编文件。如果未设置, 则所有文件都被组合。所有\"(反斜杠)必须转义为\"才能符合yaml语法
remote_src(1.4后新增): 如果为False, 它将在本地主机上搜索src, 如果为True, 它将转到src的远程主机。默认值为True
src: 源文件(即零散文件)的路径
validate(2.0后新增): 与template的validate相同, 指定命令验证文件



在线
客服

```
# 指定源文件（即零散文件）的路径，合并一个目标文件someapp.conf
- assemble:
  src: /etc/someapp/fragments
dest: /etc/someapp/someapp.conf

# 指定一个分隔符，将在每个片段之间插入
- assemble:
  src: /etc/someapp/fragments
  dest: /etc/someapp/someapp.conf
delimiter: '### START FRAGMENT ###'

# 在SSHD传递验证后，将新的'sshd_config'文件复制到目标地址
- assemble:
  src: /etc/ssh/conf.d/
  dest: /etc/ssh/sshd_config
  validate: '/usr/sbin/sshd -t -f %s'
```

6、add_host模块

1、简介

- add_host模块使用变量在清单中创建新的主机组，以便在以后的相同剧本中使用。获取变量以便我们可以更充分地定义新主机
- add_host模块在playbook执行的过程中，动态的添加主机到指定的主机组中

2、参数

groups: 要添加主机至指定的组，以逗号分隔
name: 要添加的主机名或IP地址，包含冒号和端口号

3、示例

```
# 添加主机到webservers组中，主机的变量foo的值为42
- name: add host to group 'just_created' with variable foo=42
  add_host:
    name: "{{ ip_from_ec2 }}"
    groups: just_created
    foo: 42

# 将主机添加到多个组
- name: add host to multiple groups
  add_host:
    hostname: "{{ new_ip }}"
    groups:
      - group1
      - group2

# 向主机添加一个非本地端口的主机
- name: add a host with a non-standard port local to your machines
  add_host:
name: "{{ new_ip }}:{{ new_port }}"

# 添加一个通过隧道到达的主机别名 (Ansible <= 1.9)
- name: add a host alias that we reach through a tunnel (Ansible <= 1.9)
  add_host:
    hostname: "{{ new_ip }}"
    ansible_ssh_host: "{{ inventory_hostname }}"
ansible_ssh_port: "{{ new_port }}"

# 添加一个通过隧道到达的主机别名 (Ansible >= 2.0)
- name: add a host alias that we reach through a tunnel (Ansible >= 2.0)
  add_host:
    hostname: "{{ new_ip }}"
    ansible_host: "{{ inventory_hostname }}"
    ansible_port: "{{ new_port }}"
```



在线
客服

7、group_by模块

- group_by模块在playbook执行的过程中，动态的创建主机组

(2) 参数

key: 其值将被用作组的变量
parents(2.4后新增): 父组的列表

(3) 示例

创建主机组

```
- group_by:
  key: machine_{{ ansible_machine }}
```

创建类似“kvm-host”的主机组

```
- group_by:
  key: virt_{{ ansible_virtualization_type }}_{{ ansible_virtualization_role }}
```

创建嵌套主机组

```
- group_by:
  key: el{{ ansible_distribution_major_version }}-{{ ansible_architecture }}
  parents:
    - el{{ ansible_distribution_major_version }}
```

8、debug模块

(1) 简介

- debug模块在执行过程中打印语句，可用于调试变量或表达式中输出信息
- 用于与“when:”指令一起调试

(2) 参数

msg: 调试输出的消息，打印自定义消息

var: 将某个任务执行的输出作为变量传递给debug模块，debug会直接将其打印输出

verbosity(2.1后新增): debug的运行调试级别



(3) 示例

为每个主机打印IP地址和网关

```
- debug:
  msg: "System {{ inventory_hostname }} has uuid {{ ansible_product_uuid }}"

- debug:
  msg: "System {{ inventory_hostname }} has gateway {{ ansible_default_ipv4.gateway }}"
  when: ansible_default_ipv4.gateway is defined

- shell: /usr/bin/uptime
  register: result

- debug:
  var: result          # 直接将上一条指令的结果作为变量传递给var，由debug打印出result的值
  verbosity: 2

- name: Display all variables/facts known for a host
  debug:
  var: hostvars[inventory_hostname]
  verbosity: 4
```

9、fail模块

(1) 简介

- fail模块用于终止当前playbook的执行，通常与条件语句组合使用，当满足条件时，终止当前play的运行
- 可以直接由failed_when取代

在线
客服

msg: 终止前打印出信息, 用于执行失败的自定义消息

(3) 示例

```
# 执行失败时打印出自定义信息
- fail:
    msg: "The system may not be provisioned according to the CMDB status."
    when: cmdb_status != "to-be-staged"
```

五、Playbook 循环

在使用Ansible做自动化运维的时候, 免不了的要重复执行某些操作, 如: 添加几个用户, 创建几个MySQL用户并为之赋予权限, 操作某个目录下所有文件等等。好在playbook支持循环语句, 可以使得某些需求很容易而且很规范的实现。

以下主要来自[他人分享](#)。

1、with_items

with_items是playbooks中最基本也是最常用的循环语句。

示例:

```
tasks:
- name: Secure config files
  file: path=/etc/{{ item }} mode=0600 owner=root group=root
  with_items:
    - my.cnf
    - shadow
    - fstab
# 或
with_items: "{{ somelist }}"
```

上面的例子说明在/etc下创建权限级别为0600, 属主属组都是root三个文件, 分别为my.cnf、shadow、fstab

使用with_items迭代循环的变量可以是单纯的列表, 也可以是一个较为复杂的数据结果, 如字典类型:

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

2、with_nested嵌套循环

示例:

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

item[0]是循环的第一个列表的值['alice','bob']。

item[1]是第二个列表的值。表示循环创建alice和bob两个用户, 并且为其赋予在三个数据库上的所有权限。

也可以将用户列表事先赋值给一个变量:

```
tasks:
```

```
- "{{users}}"
- [ 'clientdb', 'employeeedb', 'providerdb' ]
```

3、with_dict

with_dict可以遍历更复杂的数据结构：

假如有如下变量内容：

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

现在需要输出每个用户的用户名和手机号：

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

4、with_fileglob文件匹配遍历

可以指定一个目录，使用with_fileglob可以循环这个目录中的所有文件，示例如下：

```
tasks:
- name: Make key directory
  file: path=/root/.sshkeys ensure=directory mode=0700 owner=root group=root
- name: Upload public keys
  copy: src={{ item }} dest=/root/.sshkeys mode=0600 owner=root group=root
  with_fileglob:
    - keys/*.pub
- name: Assemble keys into authorized_keys file
  assemble: src=/root/.sshkeys dest=/root/.ssh/authorized_keys mode=0600 owner=root group=
```



5、with_subelement遍历子元素

假如现在需要遍历一个用户列表，并创建每个用户，而且还需要为每个用户配置以特定的SSH key登录。变量文件内容如下：

```
users:
- name: alice
  authorized:
    - /tmp/alice/onekey.pub
    - /tmp/alice/twokey.pub
  mysql:
    password: mysql-password
    hosts:
      - "%"
      - "127.0.0.1"
      - "::1"
      - "localhost"
    privs:
      - "*.*.SELECT"
      - "DB1.*.ALL"
- name: bob
  authorized:
    - /tmp/bob/id_rsa.pub
  mysql:
    password: other-mysql-password
    hosts:
```

在线
客服

```

- "DB2.*:ALL"

# playbook中定义如下:
- 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

# 也可以遍历嵌套的子列表:
- name: Setup MySQL users
  mysql_user: name={{ item.0.name }} password={{ item.0.mysql.password }} host={{ item.1
  with_subelements:
    - users
- mysql.hosts

```

6、with_sequence循环整数序列

with_sequence可以生成一个自增的整数序列，可以指定起始值和结束值，也可以指定增长步长。参数以key=value的形式指定，format指定输出的格式。数字可以是十进制、十六进制、八进制：

```

- 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%02d
    # 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 # stride用于指定步长
    # a simpler way to use the sequence plugin
    # create 4 groups
    - group: name=group{{ item }} state=present
      with_sequence: count=4

```



7、with_random_choice随机选择

从列表中随机取一个值：

```

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

```

8、do-Util循环

示例：

```

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

```

重复执行shell模块，当shell模块执行的命令输出内容包含“all systems go”的时候停止。重试5次，延迟时间10秒。retries默认值为3，delay默认值为5。任务的返回值为最后一次循环的返回结果。

9、循环注册变量

在线
客服


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

变量echo内容如下：

```
{
  "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"
    }
  ]
}
```



遍历注册变量的结果：

```
- 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`"
```

10、with_together遍历数据并行集合

示例：

```
- hosts: webservers
  remote_user: root
  vars:
    alpha: [ 'a', 'b', 'c', 'd' ]
    numbers: [ 1, 2, 3, 4 ]
  tasks:
    - debug: msg="{{ item.0 }} and {{ item.1 }}"
      with_together:
        - "{{ alpha }}"
        - "{{ numbers }}"
```

在线
客服

输出的结果为：

```

ok: [192.168.1.65] => (item=['a', 1]) => {
  "item": [
    "a",
    1
  ],
  "msg": "a and 1"
}
ok: [192.168.1.65] => (item=['b', 2]) => {
  "item": [
    "b",
    2
  ],
  "msg": "b and 2"
}
ok: [192.168.1.65] => (item=['c', 3]) => {
  "item": [
    "c",
    3
  ],
  "msg": "c and 3"
}
ok: [192.168.1.65] => (item=['d', 4]) => {
  "item": [
    "d",
    4
  ],
  "msg": "d and 4"
}

```

loop模块一般在下面的场景中使用：

1. 类似的配置模块重复了多遍
2. fact是一个列表
3. 创建多个文件,然后使用assemble聚合成一个大文件
4. 使用with_fileglob匹配特定的文件管理

六、Playbook 条件语句



在有的时候play的结果依赖于变量、fact或者是前一个任务的执行结果，从而需要使用到条件语句。

以下主要来自[他人分享](#)。

1、when

有的时候在特定的主机需要跳过特定的步骤，例如在安装包的时候，需要指定主机的操作系统类型，或者是当操作系统的硬盘满了之后，需要清空文件等,可以使用when语句来做判断。when关键字后面跟着的是python的表达式,在表达式中你能够使用任何的变量或者fact,当表达式的结果返回的是false,便会跳过本次的任务。

(1) 基本用法

```

---
- name: Install VIM
  hosts: all tasks:
    - name: Install VIM via yum
      yum: name=vim-enhanced state=installed
      when: ansible_os_family == "RedHat"
    - name: Install VIM via apt
      apt: name=vim state=installed
      when: ansible_os_family == "Debian"
    - name: Unexpected OS family
      debug: msg="OS Family {{ ansible_os_family }} is not supported" fail=yes
      when: not ansible_os_family == "RedHat" or ansible_os_family == "Debian"

```

在线
客服

条件语句还有一种用法,它还可以让你当达到一定的条件的时候暂停下来,等待你的输入确认。一般情况下,当ansible遭遇到error时,它会直接结束运行。那其实你可以当遭遇到不是预期的情况的时候给使用pause模块,这

```
- name: pause for unexpected conditions
  pause: prompt="Unexpected OS"
  when: ansible_os_family != "RedHat"
```

(2) 在when中使用jinja2的语法

```
tasks:
  - command: /bin/false
    register: result
    ignore_errors: True
  - command: /bin/something
    when: result|failed
  - command: /bin/something_else
    when: result|success
  - command: /bin/still/something_else
    when: result|skipped
  - command: /bin/foo
    when: result|changed
- hosts: all
  user: root
  vars:
    epic: true
  tasks:
    - shell: echo "This certainly is epic!"
      when: epic
    - shell: echo "This certainly is not epic!"
      when: not epic
```

(3) 如果变量不存在，则可以通过jinja2的'defined'命令跳过

```
tasks:
  - shell: echo "I've got '{{ foo }}' and am not afraid to use it!"
    when: foo is defined
  - fail: msg="Bailing out. this play requires 'bar'"
    when: bar is not defined
```

(4) when在循环语句中的使用方法

```
tasks:
  - command: echo {{ item }}
    with_items: [ 0, 2, 4, 6, 8, 10 ]
    when: item > 56
# 在include中使用的示例: - include: tasks/sometasks.yml
# 在roles中使用的示例: - hosts: webbservers
  roles:
    - { role: debian_stock_config, when: ansible_os_family == 'Debian' }
```



2、条件导入

有时候，你也也许想在一个Playbook中以不同的方式做事，比如说在debian和centos上安装apache，apache的包名不同，除了when语句，还可以使用下面的示例来解决：

```
---
- hosts: all
  remote_user: root
  vars_files:
    - "vars/common.yml"
    - [ "vars/{{ ansible_os_family }}.yml", "vars/os_defaults.yml" ]
  tasks:
    - name: make sure apache is running
      service: name={{ apache }} state=running
```

很多不同的yml文件只是包含键和值，如下：

```
[root@Ansible ~]# for vars/CentOS.yml
apache: httpd
somethingelse: 42
```

在线
客服

查看的将是“vars/Debian.yml”而不是“vars/CentOS.yml”，如果没找到，则寻找默认文件“vars/os_defaults.yml”。

3、with_first_found

有时候，我们想基于不同的操作系统，选择不同的配置文件，及配置文件的存放路径，可以借助with_first_found来解决：

```
- name: template a file
  template: src={{ item }} dest=/etc/myapp/foo.conf
  with_first_found:
    - files:
      - {{ ansible_distribution }}.conf
      - default.conf
    paths:
      - search_location_one/somedir/
      - /opt/other_location/somedir/
```

4、failed_when

failed_when其实是ansible的一种错误处理机制，是由fail模块使用了when条件语句的组合效果。示例如下：

```
- name: this command prints FAILED when it fails
  command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'FAILED' in command_result.stderr"
```

我们也可以直接通过fail模块和when条件语句，写成如下：

```
- name: this command prints FAILED when it fails
  command: /usr/bin/example-command -x -y -z
  register: command_result
  ignore_errors: True

- name: fail the play if the previous command did not succeed
  fail: msg="the command failed"
  when: "'FAILED' in command_result.stderr"
```



5、changed_when

当我们控制一些远程主机执行某些任务时，当任务在远程主机上成功执行，状态发生更改时，会返回changed状态响应，状态未发生更改时，会返回OK状态响应，当任务被跳过时，会返回skipped状态响应。我们可以通过changed_when来手动更改changed响应状态。示例如下：

```
- shell: /usr/bin/billybass --mode="take me to the river"
  register: bass_result
  changed_when: "bass_result.rc != 2" # 只有该条task执行以后，bass_result.rc的值不为2时，才会返回chan

# 永远不会报告“改变”的状态
- shell: wall 'beep'
  changed_when: False # 当changed_when为false时，该条task在执行以后，永远不会返回changed状态
```



©著作权归作者所有：来自51CTO博客作者小左先森的原创作品，如需转载，请注明出处，否则将追究法律责任

Ansible 自动化运维 Playbook

自动化运维

在线客服

3

收藏 分享

上一篇：自动化运维工具Ansible实战... 下一篇：ELK实时日志分析平台环境部署

3

4

分享



小左先森

关注



小左先森

27篇文章, 144W+人气, 29粉丝

关注



提问和评论都可以, 用心的回复会被更多人看到和认可

Ctrl+Enter 发布

取消

发布

推荐专栏

更多



基于Python的DevOps实战

自动化运维开发新概念

共20章 | 抚琴煮酒

¥51.00 387人订阅

订 阅



全局视角看大型园区网

路由交换+安全+无线+优化+运维

共40章 | 51CTO夏杰

¥51.00 1214人订阅

订 阅



网工2.0晋级攻略 —— 零基础入门Python/A...

网络工程师2.0进阶指南

共30章 | 姜汁啤酒

¥51.00 1414人订阅

订 阅



负载均衡高手炼成记

高并发架构之路

共15章 | sery

¥51.00 473人订阅

订 阅



带你玩转高可用

前百度高级工程师的架构高可用实战

共15章 | 曹林华

¥51.00 444人订阅

订 阅



猜你喜欢

Flume学习之路 (三) Flume的配置方式

iptables交互配置脚本【Linux运维之道之脚本案例】

自动化运维工具Ansible详细部署

Flume学习之路 (二) Flume的Source类型

运维自动化-Ansible (一)

使用Prometheus+grafana打造高逼格监控平台

在线
客服



Jenkins与Docker的自动化CI/CD实战

CentOS7 搭建企业级NFS网络文件服务器

Django+Django-Celery+Celery的整合实战

Nginx10m+高并发内核优化详解

部署SaltStack及批量安装httpd服务

ansible基础学习，常用模块概述

k8s实践6:从解决报错开始入门RBAC

Linux 性能测试工具 sysbench 的安装与简单使用

架构师的操作系统

Kubernetes共享存储之Glusterfs+Heketi



在线
客服

