ANSIBLE 2.4

文档作者：马龙帅
博客地址：*https://www.cnblogs.com/f-ck-need-u/*

目录

1.基本配置和使用 5

1.1 安装Ansible 5

1.2 配置ansible 7

1.2.1 环境配置 7

1.2.2 SSH互信配置 8

1.2.3 简单测试 9

1.3 inventory 10

2.选项和常用模块 12

2.1 ansible命令解释 12

2.2 常用模块 15

shell和command 15

复制模块copy 16

template模块 17

文件模块file 18

fetch拉取文件模块 18

rsync模块synchronize 19

包管理模块yum 19

配置yum源模块yum_repository 20

服务管理模块service 20

systemd模块 21

用户管理模块user 21

authorzied_key模块 22

debug模块 22

定时任务模块cron 23

归档模块archive 24

解包模块unarchive 24

下载模块get_url 25

wait_for模块 26

script模块 26

3.YAML语法和playbook写法 28

3.1 初步说明 28

3.2 列表 28

3.3 字典 29

3.4 分行写 31

3.5 向模块传递参数 32

3.6 playbook和play的关系 33

3.7 playbook中什么时候使用引号 34

4.playbook应用和示例 36

4.1 yaml简单示例 36

4.2 ansible-playbook命令说明及playbook书写简单示例 36
4.3 playbook的内容 37
4.3.1 hosts和remoter_user 37
4.3.2 task list 38
4.3.3 notify和handler 40
4.3.4 标签tag 41
4.4 include和roles 41
4.4.1 include 41
4.4.2 roles 43
4.5 roles示例：批量自动化安装 45
5.各种变量定义方式和变量引用 52
5.1 ansible facts 52
5.2 变量引用json数据的方式 53
5.2.1 引用json字典数据的方式 54
5.2.2 引用json数组数据的方式 54
5.2.3 引用facts数据 55
5.3 设置本地facts 56
5.4 输出和引用变量 56
5.5 注册和定义变量的各种方式 57
5.5.1 register注册变量 57
5.5.2 set_fact定义变量 57
5.5.3 vars定义变量 58
5.5.4 vars_files定义变量 58
5.5.5 roles中的变量 58
5.5.6 命令行传递变量 59
5.5.7 借助with_items叠加变量 59
5.5.8 inventory中主机变量和主机组变量 60
5.5.9 内置变量 61
6.条件判断和循环 65
6.1 循环 65
6.1.1 with_items迭代列表 65
6.1.2 with_dict迭代字典项 66
6.1.3 with_fileglob迭代文件 66
6.1.4 with_lines迭代行 67
6.1.5 with_nested嵌套迭代 67
6.2 条件判断 67
7.执行过程分析、异步模式和速度优化 69
7.1 ansible执行过程分析 69
7.2 ansible并发和异步 71
7.3 ansible的-t选项妙用 72
7.4 优化ansible速度 73
7.4.1 设置ansible开启ssh长连接 74

7.4.2 开启pipelining 74
7.4.3 修改ansible执行策略 76
7.4.4 设置facts缓存 77
8.playbook杂项 78
8.1 指定运行play的主机delegate_to和local_action 78
8.2 只运行一次run_once 78
8.3 分批执行play 79
8.4 分批执行play时的最大失败百分比 80
8.5 错误处理——忽略错误 80
8.6 错误处理——自定义错误判断条件 80
9.playbook示例：编译安装httpd 82

1.基本配置和使用

Ansible是一种批量、自动部署工具，不仅可以批量，还可以自动。它主要基于ssh进行通信，不要求客户端(被控制端)安装ansible。

1.1 安装Ansible

安装方法有多种，可以下载源码后编译安装，可以从git上获取资源安装，也可以rpm包安装。rpm安装需要配置epel源。

```
cat <<eof>>/etc/yum.repos.d/my.repo
[epel]
name=epel
baseurl=http://mirrors.aliyun.com/epel/7Server/x86_64/
enable=1
gpgcheck=0
eof
```

后面几篇文章用到的环境。

主机描述	IP地址	主机名	操作系统
ansible6_server1	192.168.100.150	server1.longshuai.com	CentOS 6.6
ansible6_node1	192.168.100.59	node1.longshuai.com	CentOS 6.6
ansible6_node2	192.168.100.60	node2.longshuai.com	CentOS 6.6
ansible6_node3	192.168.100.61	node3.longshuai.com	CentOS 6.6
ansible7_server2	192.168.100.62	server2.longshuai.com	CentOS 7.2
ansible7_node1	192.168.100.63	anode1.longshuai.com	CentOS 7.2
ansible7_node2	192.168.100.64	anode2.longshuai.com	CentOS 7.2
ansible7_node3	192.168.100.65	anode3.longshuai.com	CentOS 7.2

经测试，CentOS 6上安装ansible 2.3版本有可能会非常慢，需要将ansible执行的结果使用重定向或者-t选项保存到文件中，下次执行才会快。

```
shell> yum -y install ansible
/etc/ansible/ansible.cfg
/etc/ansible/hosts
/etc/ansible/roles
/usr/bin/ansible
/usr/bin/ansible-2
/usr/bin/ansible-2.6
/usr/bin/ansible-connection
/usr/bin/ansible-console
/usr/bin/ansible-console-2
/usr/bin/ansible-console-2.6
/usr/bin/ansible-doc
/usr/bin/ansible-doc-2
/usr/bin/ansible-doc-2.6
/usr/bin/ansible-galaxy
/usr/bin/ansible-galaxy-2
/usr/bin/ansible-galaxy-2.6
/usr/bin/ansible-playbook
/usr/bin/ansible-playbook-2
/usr/bin/ansible-playbook-2.6
/usr/bin/ansible-pull
/usr/bin/ansible-pull-2
/usr/bin/ansible-pull-2.6
/usr/bin/ansible-vault
/usr/bin/ansible-vault-2
/usr/bin/ansible-vault-2.6
```

使用ansible-doc可以列出相关的帮助。

```
ansible-doc -h
Usage: ansible-doc [options] [module...]

Options:
  -a, --all             Show documentation for all modules
  -h, --help            show this help message and exit
  -l, --list            List available modules
  -M MODULE_PATH, --module-path=MODULE_PATH
                        specify path(s) to module library (default=None)
  -s, --snippet         Show playbook snippet for specified module(s)
  -v, --verbose         verbose mode (-vvv for more, -vvvv to enable
                        connection debugging)
  --version             show program's version number and exit
```

其中"-l"选项用于列出ansible的模块，通常结合grep来筛选。例如找出和yum相关的可用模块。

```
ansible-doc -l | grep yum
yum                                Manages packages with the `yum' package manager
yum_repository                     Add or remove YUM repositories
```

再使用"-s"选项可以获取指定模块的使用帮助。例如，获取yum模块的使用语法。

```
ansible-doc -s yum
- name: Manages packages with the `yum' package manager
  action: yum
      conf_file           # The remote yum configuration file to use for the transaction.
      disable_gpg_check   # Whether to disable the GPG checking of signatures of packages being
                            installed. Has an effect only if state is `present' or  `latest'.
      disablerepo         # `Repoid' of repositories to disable for the install/update operation.
                            These repos will not persist beyond the transaction. When specifying
                            multiple repos, separate them with a ",".
      enablerepo          # `Repoid' of repositories to enable for the install/update operation.
                            These repos will not persist beyond the transaction. When specifying
                            multiple repos, separate them with a ",".
      exclude             # Package name(s) to exclude when state=present, or latest
      installroot         # Specifies an alternative installroot, relative to which all packages
                            will be installed.
      list                # Package name to run the equivalent of yum list <package> against.
      name=               # Package name, or package specifier with version, like `name-1.0'.
                            When using state=latest, this can be '*' which means run: yum -y update.
                            You can also pass a url or a local path to a rpm file(using state=present).
                            To operate on several packages this can accept a comma separated list of
                            packages or (as of 2.0) a list of packages.
      skip_broken         # Resolve depsolve problems by removing packages that are causing problems
                            from the trans- action.
      state               # Whether to install (`present' or `installed', `latest'), or remove
                            (`absent' or `removed') a package.
      update_cache        # Force yum to check if cache is out of date and redownload if needed. Has
                            an effect only if state is `present' or `latest'.
      validate_certs      # This only applies if using a https url as the source of the rpm.
                            e.g. for localinstall. If set to `no', the SSL certificates will not be
                            validated. This should only set to `no' used on personally controlled
                            sites using self-signed certificates as it avoids verifying the source
                            site. Prior to 2.1 the code worked as if this was set to `yes'.
```

例如使用yum安装unix2dos包。

```
ansible 192.168.100.60 -m yum -a "name=unix2dos state=present"
```

其中192.168.100.60是被ansible远程控制的机器，即要在此机器上安装unix2dos，下一小节将说明如何指定待控制主机。"-m"指定模块名称，"-a"用于为模块指定各模块参数，例如name和state。

ansible命令选项和各模块的使用方法见：Ansible系列(二)：选项和常用模块。

1.2 配置ansible

1.2.1 环境配置

Ansible配置以ini格式存储配置数据，在Ansible中几乎所有配置都可以通过Ansible的Playbook或环境变量来重新赋值。在运行Ansible命令时，命令将会按照以下顺序查找配置文件。

- `ANSIBLE_CONFIG`：首先，Ansible命令会检查环境变量，及这个环境变量指向的配置文件。
- `./ansible.cfg`：其次，将会检查当前目录下的ansible.cfg配置文件。
- `~/.ansible.cfg`：再次，将会检查当前用户home目录下的.ansible.cfg配置文件。
- `/etc/ansible/ansible.cfg`：最后，将会检查在用软件包管理工具安装Ansible时自动产生的配置文件。

1、使用化境变量方式来配置

大多数的Ansible参数可以通过设置带有`ANSIBLE_`开头的环境变量进行配置，参数名称必须都是大写字母，如下配置：

```
export ANSIBLE_SUDO_USER=root
```

设置了环境变量之后，`ANSIBLE_SUDO_USER`就可以在后续操作中直接引用。

2、设置ansible.cfg配置参数

Ansible有很多配置参数，以下是几个默认的配置参数：

```
inventory = /root/ansible/hosts
library = /usr/share/my_modules/
forks = 5
sudo_user = root
remote_port = 22
host_key_checking = False
timeout = 20
log_path = /var/log/ansible.log
```

`inventory`：该参数表示inventory文件的位置，资源清单(inventory)就是Ansible需要连接管理的一些主机列表。
`library`：Ansible的所有操作都使用模块来执行实现，这个library参数就是指向存放Ansible模块的目录。
`forks`：设置默认情况下Ansible最多能有多少个进程同时工作，默认5个进程并行处理。具体需要设置多少个，可以根据控制端性能和被管理节点的数量来确定。
`sudo_user`：设置默认执行命令的用户，也可以在playbook中重新设置这个参数。
`remote_port`：指定连接被管理节点的管理端口，默认是22，除非设置了特殊的SSH端口，否则不需要修改此参数。
`host_key_checking`：设置是否检查SSH主机的密钥。可以设置为True或False。即ssh的主机再次验证。
`timeout`：设置SSH连接的超时间隔，单位是秒。
`log_path`：Ansible默认不记录日志，如果想把Ansible系统的输出记录到日志文件中，需要设置log_path。需要注意，模块将会调用被管节点的(r)syslog来记录，执行Ansible的用户需要有写入日志的权限。

1.2.2 SSH互信配置

将ansible server的ssh公钥分发到各被管节点上。

在ansible6_server1和ansible_server2上：

```
ssh-keygen -t rsa -f ~/.ssh/id_rsa -N ''
ssh-copy-id root@192.168.100.59
ssh-copy-id root@192.168.100.60
ssh-copy-id root@192.168.100.61
ssh-copy-id root@192.168.100.62
ssh-copy-id root@192.168.100.63
ssh-copy-id root@192.168.100.64
ssh-copy-id root@192.168.100.65
ssh-copy-id root@192.168.100.150
```

也可以使用ansible自身来批量添加密钥到被控节点上。使用ansible的authorized_key模块即可。见后文常见模块介绍部分。

以下是借助expect工具实现非交互式的ssh-copy-id，免得总是询问远程用户的登录密码。

```
# 安装expect
[root@server2 ~]# yum -y install expect

# expect脚本
[root@server2 ~]# cat auto_sshcopyid.exp
#!/usr/bin/expect

set timeout 10
set user_hostname [lindex $argv 0]
set password [lindex $argv 1]

spawn ssh-copy-id $user_hostname
expect {
        "(yes/no)?"
        {
                send "yes\n"
                expect "*password: " { send "$password\n" }
        }
        "*password: " { send "$password\n" }
}

expect eof

# 批量调用expect的shell脚本
[root@server2 ~]# cat sshkey.sh
#!/bin/bash

ip=`echo -n "$(seq -s "," 59 65),150" | xargs -d "," -i echo 192.168.100.{}`
password="123456"
#user_host=`awk '{print $3}' /root/.ssh/id_rsa.pub`

for i in $ip;do
        /root/auto_sshcopyid.exp root@$i $password &>>/tmp/a.log
        ssh root@$i "echo $i ok"
done

# 执行shell脚本配置互信
[root@server2 ~]# chmod +x /root/{sshkey.sh,auto_sshcopyid.exp}
[root@server2 ~]# ./sshkey.sh
```

1.2.3 简单测试

向默认的inventory文件/etc/ansible/hosts中添加上几个被管节点清单。

在ansible6_server1上：

```
cat >>/etc/ansible/hosts<<eof
192.168.100.59
192.168.100.60
192.168.100.61
192.168.100.62
192.168.100.63
192.168.100.64
192.168.100.65
[centos6]
192.168.100.59
192.168.100.60
192.168.100.61
[centos7]
192.168.100.63
192.168.100.64
192.168.100.65
[centos:children]
centos6
centos7
eof
```

在ansible7_server2上：

```
cat >>/etc/ansible/hosts<<eof
192.168.100.150
192.168.100.59
192.168.100.60
192.168.100.61
192.168.100.63
192.168.100.64
192.168.100.65
[centos6]
192.168.100.59
192.168.100.60
192.168.100.61
[centos7]
192.168.100.63
192.168.100.64
192.168.100.65
[centos:children]
centos6
centos7
eof
```

使用ping模块测试被管节点。能成功，说明ansible能控制该节点。

```
ansible 192.168.100.59 -m ping
ansible centos6 -m ping
192.168.100.59 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
192.168.100.60 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
192.168.100.61 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

如果要指定非root用户运行ansible命令，则加上"--sudo"或"-s"来提升为sudo_user配置项所指定用户的权限。

```
ansible webservers -m ping -u ansible --sudo
```

或者使用become提升权限。

```
ansible webservers -m ping -b --become-user=root --become-method=sudo
```

1.3 inventory

inventory用于定义ansible要管理的主机列表，可以定义单个主机和主机组。上面的/etc/ansible/hosts就是默认的inventory。下面展示了inventory常用的定义规则。

```
cat -n /etc/ansible/hosts
1 192.168.100.59:22
2 192.168.100.60 ansible_ssh_pass=123456 ansible_ssh_port=22
3 [nginx]
4 192.168.100.5[7:9]
5 [nginx:vars]
6 ansible_ssh_pass='123456'
7 [webservers:children]
8 nginx
```

第一行和第二行单独定义主机，第一行带上了连接被管节点的端口，第二行带上了单独传递给ssh的参数，分别是ssh连接时的登录远程用户的密码参数和ssh的连接端口。
第三行和第四行定义的是nginx主机组，该组中包含了192.168.100.57到59这3台主机。还支持字母的扩展，

如"web[a-d]"。
第五行和第六行定义了要传递给nginx主机组的变量。若定义为"[all:vars]"或"[*:vars]"则表示传递给所有主机的变量。
第七和第八行定义了一个新的主机组webservers，改组的组成员有nginx组。

可以指定多个inventory配置文件，只需在ansible的配置文件如/etc/ansible/ansible.cfg中将inventory指令设置为对应的文件或目录即可，如果是目录，那么此目录下的所有文件都是inventory文件。

inventory文件中可以使用一些内置变量，绝大多数ansible的连接和权限变量都可以在此使用，见ansible命令解释。常见的有：

- `ansible_ssh_host`： ansible使用ssh要连接的主机。
- `ansible_ssh_port`： ssh的端口。默认为22。
- `ansible_ssh_user`： ssh登录的用户名。默认为root。
- `ansible_ssh_pass`： ssh登录远程用户时的认证密码。
- `ansible_ssh_private_key_file`： ssh登录远程用户时的认证私钥。(?)
- `ansible_connection`： 使用何种模式连接到远程主机。默认值为smart(智能)，表示当本地ssh支持持久连接(controlpersist)时采用ssh连接，否则采用python的paramiko ssh连接。
- `ansible_shell_type`： 指定远程主机执行命令时的shell解析器，默认为sh(不是bash，它们是有区别的，也不是全路径)。
- `ansible_python_interpreter`： 远程主机上的python解释器路径。默认为/usr/bin/python。
- `ansible_*_interpreter`：使用什么解释器。例如，sh、bash、awk、sed、expect、ruby等等。

其中有几个参数可以在配置文件ansible.cfg中指定，但指定的指令不太一样，以下是对应的配置项：

- remote_port： 对应于ansible_ssh_port。
- remote_user： 对应于ansible_ssh_user。
- private_key_file： 对应于ansible_ssh_private_key_file。
- excutable： 对应于ansible_shell_type。但有一点不一样，excutable必须指定全路径，而后者只需指定basename。

如果定义了"ansible_ssh_host"，那么其前面的主机名就称为别名。例如，以下inventory文件中nginx就是一个别名，真正连接的对象是192.168.100.65。

```
nginx ansible_ssh_host=192.168.100.65 ansible_ssh_port=22
```

当inventory中有任何一台有效主机时，ansible就默认隐式地可以使用"localhost"作为本机，但inventory中没有任何主机时是不允许使用它的，且"all"或"*"所代表的所有主机也不会包含localhost。例如：

```
anisble localhost -i /path/to/inventory_file -m MODULE -a "ARGS"
anisble all -i /path/to/inventory_file -m MODULE -a "ARGS"
anisble * -i /path/to/inventory_file -m MODULE -a "ARGS"
```

`inventory_hostname`是ansible中可以使用的一个变量，该变量代表的是每个主机在inventory中的主机名称。例如"192.168.100.59"。这是目前遇到的第一个变量。

2.选项和常用模块

本章内容做速查手册

2.1 ansible命令解释

通过ansible命令执行的任务称为ad-hoc命令(任务)，其实它是相对playbook而言的。通常，命令行用来实现ansible的批量管理功能，playbook用来实现批量自动化功能。

【以下为普通选项：】

`-a MODULE_ARGS`

`--args=MODULE_ARGS`

传递参数给模块

`--ask-vault-pass`

询问vault的密码

`-B SECONDS`

`--background=SECONDS`

异步后台方式执行任务，并在指定的秒数后超时，超时会杀掉任务的进程。默认是同步，即保持长连接，它会等待所有执行完毕（即阻塞模式）。但有时候是没必要这样的，比如某些命令的执行时间比ssh的超时时间还长。如果不指定超时秒数，将以同步方式运行任务

`-P POLL_INTERVAL`

`--poll=POLL_INTERVAL`

异步模式下轮训任务的时间间隔，默认10秒

`-C`

`--check`

不对远程主机做出一些改变，而是预测某些可能发生的改变

`-D`

`--diff`

当文件或模板发生了改变，显示出不同之处，和-C选项配合使用更佳

`-e EXTRA_VARS`

`--extra-vars=EXTRA_VARS`

配置额外的配置变量(key=value或者YAML/JSON格式)

`-f FORKS`

`--forks=FORKS`

指定并行处理的进程数量，默认为5个

`-h`

`--help`

显示帮助信息

`-i INVENTORY`

`--inventory-file=INVENTORY`

指定inventory文件，多个文件使用逗号分隔。默认为/etc/ansible/hosts

`-l SUBSET`

`--limit=SUBSET`

使用额外的匹配模式来筛选目标主机列表。此处的匹配模式是在已有匹配模式下进行的，所以是更严格的筛选。例如指定了主机组的情况下，使用-l选项从中只选一台主机进行控制

`--list-hosts`

不会执行任何操作，而是列出匹配到的主机列表

`-m MODULE_NAME`

`--module-name=MODULE_NAME`

指定要执行的模块名，默认的模块为"command"

`-M MODULE_PATH`

`--module-path=MODULE_PATH`

指定模块目录，默认未设置

`--new-vault-password-file=NEW_VAULT_PASSWORD_FILE`

new vault password file for rekey

`-o`

`--one-line`

简化输出(一行输出模式)

`--output=OUTPUT_FILE`

output file name for encrypt or decrypt; use - for stdout

`--syntax-check`

检查playbook的语法，不会执行

`-t TREE`

`--tree=TREE`

记录输出到此目录中（测试时以每个host名如IP地址为文件名记录，结果记录到对应的文件中）。

此选项在ansible巨慢的时候(如瞬间应该返回的命令还需要10多秒才完成)有奇用，或者将ansible的结果重定向到某个文件中也能解决，为什么如此，我也不明白(表面看来和输出方式有关系)，多次亲测有效。

`--vault-password-file=VAULT_PASSWORD_FILE`

指定vault密码文件

`-v`

`--verbose`

输出详细信息，-vvv和-vvvv会输出更多新

`--version`

显示ansible的版本

【以下是连接选项，用于控制谁以及如何连接主机：】

`-k`

`--ask-pass`

询问连接时的密码

`--private-key=KEY_FILE`

`--key-file=KEY_FILE`

使用文件来认证SSH连接过程。

`-u REMOTE_USER`

`--user=REMOTE_USER`

使用指定的用户名进行连接

`-c CONNECTION`

`--connection=CONNECTION`

连接类型，默认为ssh。paramiko (SSH), ssh, winrm and local. local is mostly useful for crontab or kickstarts.

`-T TIMEOUT, --timeout=TIMEOUT`

连接的超时时间，单位为秒，默认为10

`--ssh-common-args=SSH_COMMON_ARGS`

指定传递给sftp/scp/ssh等工具的通用额外参数

`--sftp-extra-args=SFTP_EXTRA_ARGS`

指定只传递给sftp的额外参数，如-f

`--scp-extra-args=SCP_EXTRA_ARGS`

指定只传递给scp的额外参数，如-l

`--ssh-extra-args=SSH_EXTRA_ARGS`

指定只传递给ssh的额外参数，如-R

【以下是权限控制选项：控制在目标主机上以什么身份和权限运行任务：】

`-s`

`--sudo`

为运行ansible命令的用户提升权限为sudo_user的权限，此命令已废弃，使用become代替

`-U SUDO_USER`

`--sudo-user=SUDO_USER`

期望的sudo_user，默认为root，已废弃，使用become替代

`-S`

`--su`

使用su的方式执行操作，已废弃，使用become替代

`-R SU_USER`

`--su-user=SU_USER`

使用此user的su执行操作，默认为root，已废弃，使用become替代

`-b`

`--become`

使用become的方式升级权限

`--become-method=BECOME_METHOD`

指定提升权限的方式，可选以下几种：sudo/su/pbrun/pfexec/doas/dzdo/ksu/runas值

`--become-user=BECOME_USER`
要提升为哪个user的权限，默认为root

`--ask-sudo-pass`
询问sudo密码，已废弃，使用become替代

`--ask-su-pass`
询问su的密码，已废弃，使用become替代

`-K`
`--ask-become-pass`
询问become提升权限时的密码

2.2 常用模块

可以从ansible-doc -l | grep来找出想要的模块。再使用ansible-doc -s module_name来查看此模块的用法。官方模块列表和说明：https://docs.ansible.com/ansible/latest/modules_by_category.html

关于模块的使用方法，需要注意的是"state"。很多模块都会有该选项，且其值几乎都包含有"present"和"absent"，表示肯定和否定的意思。

ansible绝大多数模块都天然具有幂等特性，只有极少数模块如shell和command模块不具备幂等性。所谓的幂等性是指多次执行同一个操作不会影响最终结果。例如，ansible的yum模块安装rpm包时，如果待安装的包已经安装过了，则再次或多次执行安装操作都不会真正的执行下去。再例如，copy模块拷贝文件时，如果目标主机上已经有了完全相同的文件，则多次执行copy模块不会真正的拷贝。ansible具有幂等性的模块在执行时，都会自动判断是否要执行。

shell和command

默认ansible使用的模块是command，即可以执行一些shell命令。shell和command的用法基本一样，实际上shell模块执行命令的方式是在远程使用/bin/sh来执行的，如/bin/sh ping。

command不能解析变量(如$HOME)和某些操作符("<", ">", "|", ";"以及"&")，所以明确要使用这些不可解析的操作符时，使用shell模块来代替command。

```
ansible-doc -s shell
- name: Execute commands in nodes.
  action: shell
      chdir       # 在执行命令前，先cd到指定的目录下
      creates     # 用于判断命令是否要执行。如果指定的文件(可以使用通配符)存在，则不执行。
      removes     # 用于判断命令是否要执行。如果指定的文件(可以使用通配符)不存在，则不执行。
      executable  # 不再使用默认的/bin/sh解析并执行命令，而是使用此处指定的命令解析。
                  # 例如使用expect解析expect脚本。必须为绝对路径。
```

在ansible中使用shell或command模块一定要注意，它们默认不满足幂等性，很多操作会重复执行，但有些操作是不允许重复执行的。例如mysql的初始化命令mysql_install_db，它只能在第一次配置的过程中初始化一次，其他任何时候如非需要则不允许执行。这时候要实现幂等性，可以通过模块的creates和removes选项进行判断，但无论如何，在执行这两个模块的时候都需要考虑要执行的命令是否应该实现幂等性。

例如：

```
tasks:
  - shell: touch helloworld.txt creates=/tmp/hello.txt
```

但建议，在参数可能产生歧义的情况下，使用args来传递ansible的参数。如：

```
- shell: touch helloworld.txt
  args:
    creates: /tmp/hello.txt
```

```
# You can use shell to run other executables to perform actions inline
- name: Run expect to wait for a successful PXE boot via out-of-band CIMC
  shell: |
    set timeout 300
    spawn ssh admin@{{ cimc_host }}

    expect "password:"
    send "{{ cimc_password }}\n"

    expect "\n{{ cimc_name }}"
    send "connect host\n"

    expect "pxeboot.n12"
    send "\n"

    exit 0
  args:
    executable: /usr/bin/expect
  delegate_to: localhost
```

复制模块copy

```
ansible-doc -l | grep copy
copy                    Copies files to remote locations.
ec2_ami_copy            copies AMI between AWS regions, return new image id
netapp_e_volume_copy    Create volume copy pairs
nxos_file_copy          Copy a file to a remote NXOS device over SCP.
unarchive               Unpacks an archive after(optionally) copying it from the local machine.
vsphere_copy            Copy a file to a vCenter datastore
win_copy                Copies files to remote locations on windows hosts.
win_robocopy            Synchronizes the contents of two directories using Robocopy.
```

使用方法：

```
ansible-doc -s copy
- name: Copies files to remote locations.
action: copy
    backup=[yes|no]   # 拷贝的同时也创建一个包含时间戳信息的备份文件，默认为no
    dest=             # 目标路径，只能是绝对路径，如果拷贝的文件是目录，则目标路径必须也是目录
    content           # 直接以content给定的字符串或变量值作为文件内容保存到远程主机上，它会替代src选项
    directory_mode    # 当对目录做递归拷贝时，设置了directory_mode将会使得只拷贝新建文件，
                      # 旧文件不会被拷贝。默认未设置
    follow=[yes|no]   # 是否追踪到链接的源文件。
    force=[yes|no]    # 设置为yes(默认)时，将覆盖远程同名文件。设置为no时，忽略同名文件的拷贝。
    group             # 设置远程文件的所属组
    owner             # 设置远程文件的所有者
    mode=             # 设置远程文件的权限。使用数值表示时不能省略第一位，如0644。
                      # 也可以使用'u+rwx'或'u=rw,g=r,o=r'等方式设置。
    src=              # 拷贝本地源文件到远程，可使用绝对路径或相对路径。如果路径是目录，且目录后加了
                      # 斜杠"/"，则只会拷贝目录中的内容到远程，如果目录后不加斜杠，则拷贝目录本身和
                      # 目录内的内容到远程。
```

默认情况下，ansible copy会检查文件md5查看是否需要拷贝，相同则不会拷贝，否则会拷贝。如果设置force=yes，则当文件md5不同时(即文件内容不同)才覆盖拷贝，设置force=no时，则只拷贝对方没有的文件。

```
ansible centos -m copy -a "src=/tmp/copy.txt dest=/tmp mode=0770 owner=sshd group=sshd backup=yes" -o -f 6

ll /tmp
-rwxrwx--- 1 sshd sshd   24 May 28 16:45 copy.txt
-rwxrwx--- 1 root root   22 May 28 16:39 copy.txt.10915.2017-05-28@16:45:14~  #这是备份文件
```

如果拷贝的是目录，则目标路径必须是目录路径。如果使用"/"结尾，则拷贝的是目录中的文件，如果不以斜杠结尾，则拷贝的是目录加目录中的文件。

```
mkdir /tmp/a
touch /tmp/a/{1..10}.txt
ansible centos -m copy -a "src=/tmp/a dest=/tmp" -o -f 6
```

template模块

template模块用法和copy模块用法基本一致，它主要用于复制配置文件。

```
ansible-doc -s template
- name: Templates a file out to a remote server.
  action: template
      backup    # 拷贝的同时也创建一个包含时间戳信息的备份文件，默认为no
      dest=     # 目标路径
      force     # 设置为yes (默认)时，将覆盖远程同名文件。设置为no时，忽略同名文件的拷贝
      group     # 设置远程文件的所属组
      owner     # 设置远程文件的所有者
      mode      # 设置远程文件的权限。使用数值表示时不能省略第一位，如0644。
                # 也可以使用'u+rwx' or 'u=rw,g=r,o=r'等方式设置
      src=      # ansible控制器上Jinja2格式的模板所在位置，可以是相对或绝对路径
      validate  # 在复制到目标主机后但放到目标位置之前，执行此选项指定的命令。
                # 一般用于检查配置文件语法，语法正确则保存到目标位置。
                # 如果要引用目标文件名，则使用%s，下面的示例中的s%即表示目标机器上的/etc/nginx/nginx.conf。
```

```
ansible centos -m template -a "src=/tmp/nginx.conf.j2 dest=/etc/nginx/nginx.conf mode=0770 owner=root group=root ba
ckup=yes validate='nginx -t -c %s'" -o -f 6
```

虽然template模块可以按需求修改配置文件内容来复制模板到被控主机上，但是有一种情况它是不能解决的：不同被控节点所需的配置文件差异很大，并非修改几个变量就可以满足。例如在centos 6和centos 7上通过yum安装的nginx，它们的配置文件内容相差非常大，且centos 6上的nginx的默认就有一个/etc/nginx/conf.d/default.conf。如果直接复制同一个模板的nginx配置文件到centos 6和centos 7上，很可能导致某一版本的nginx不能启动。

这时就有必要在复制模板时挑选对应发行版的模板文件进行配对复制，例如要复制到 centos 6上的源模板是nginx6.conf.j2，复制到centos 7上的源模板是nginx7.conf.j2。这种行为可以称之为**"基于变量选择文件或模板"**。

```
---
 - tasks:
     - name: template file based var
       template: src=/templates/nginx{{ ansible_distribution_major_version }}.conf.j2 dest=/etc/nginx/nginx.conf va
lidate="/usr/sbin/nginx -t -c %s"
```

还可以在文件内容中指定jinja2的替代变量，在ansible执行时首先会根据变量内容进行渲染，渲染后再执行相关模块。例如，此处的template模块，复制一个基于发行版本号的yum源配置文件。以下是某个repo文件模板base.repo.j2的内容。

```
[epel]
name=epel
baseurl=http://mirrors.aliyun.com/epel/{{ ansible_distribution_major_version }}Server/x86_64/
enable=1
gpgcheck=0
```

再复制即可。

```
---
 - tasks:
     - template: src=my.repo.j2 dest=/etc/yum.repos.d/my.repo
```

文件模块file

管理文件、目录的属性，也可以创建文件或目录。

```
ansible-doc -s file
- name: Sets attributes of files
  action: file
      group       # file/directory的所属组
      owner       # file/directory的所有者
      mode        # 修改权限，格式可以是0644、'u+rwx'或'u=rw,g=r,o=r'等
      path=       # 指定待操作的文件，可使用别名'dest'或'name'来替代path
      recurse     # (默认no)递归修改文件的属性信息，要求state=directory
      src         # 创建链接时使用，指定链接的源文件
      state       # directory:如果目录不存在则递归创建
                  # file:文件不存在时，不会被创建(默认值)
                  # touch:touch由path指定的文件，即创建一个新文件，或修改其mtime和atime
                  # link:修改或创建软链接
                  # hard:修改或创建硬链接
                  # absent:目录和其中的文件会被递归删除，文件或链接将取消链接状态
```

需要注意的是，file模块可以递归创建目录，但是不能在不存在的目录中创建文件，只能先创建目录，再在此目录中创建文件。

创建目录，并递归修改目录的属性。

```
ansible 192.168.100.63 -m file -a 'path=/tmp/xyz state=directory owner=root group=root mode=0755 recurse=yes'
```

修改目录中属性。

```
ansible 192.168.100.63 -m file -a 'path=/tmp/xyz state=touch mode=0644'
```

创建或修改文件属性。

```
ansible 192.168.100.63 -m file -a 'path=/tmp/xyz/a.txt state=touch mode=0644'
```

fetch拉取文件模块

和copy工作方式类似，只不过是从远程主机将文件拉取到本地端，存储时使用主机名作为目录树，且只能拉取文件不能拉取目录。

```
ansible-doc -s fetch
- name: Fetches a file from remote nodes
  action: fetch
      dest=               # 本地存储拉取文件的目录。例如dest=/data，src=/etc/fstab，
                          # 远程主机名host.exp.com，则保存的路径为/data/host.exp.com/etc/fstab。
      fail_on_missing     # 当设置为yes时，如果拉取的源文件不存在，则此任务失败。默认为no。
      flat                # 改变拉取后的路径存储方式。如果设置为yes，且当dest以"/"结尾时，将直接把源文件
                          # 的basename存储在dest下。显然，应该考虑多个主机拉取时的文件覆盖情况。
      src=                # 远程主机上的源文件。只能是文件，不支持目录。在未来的版本中可能会支持目录递归拉取。
      validate_checksum   # fetch到文件后，检查其md5和源文件是否相同。
```

```
# 存储为/tmp/localhost/etc/fstab
ansible localhost -m fetch -a "src=/etc/fstab dest=/tmp"

# 存储为/tmp/fstab
ansible localhost -m fetch -a "src=/etc/fstab dest=/tmp/ flat=yes"

# 存储为/tmp/fstab-localhost
ansible localhost -m fetch -a "src=/etc/fstab dest=/tmp/fstab-{{inventory_hostname}} flat=yes"
```

rsync模块synchronize

synchronize模块用于实现rsync的简单版常用功能，它无法实现完整版的rsync，毕竟rsync功能太多太细致。如果要使用rsync，还是应该使用command或shell模块来调用rsync命令。

完整的rsync功能见rsync命令中文手册。

```
ansible-doc -s synchronize
- name: A wrapper around rsync to make common tasks in your playbooks quick and easy.
  action: synchronize
      src=           # 指定待传输的源文件。可以是相对路径，也可以是绝对路径。
      dest=          # 目标路径。可以是绝对路径，也可以是相对路径。
      mode           # 指定推(push)还是拉(pull)的传输模式。
                     # push时，本地为sender端，pull时，远程为sender端。默认为push。
      archive        # 等价于rsync的"-a"选项，即使用归档模式。它等价于rsync的"-rtopgDl"选项。值为yes/no。
      times          # 保留mtime属性。
      group          # 保留所属组属性。
      owner          # 保留所有者属性。
      links          # 拷贝链接文件自身。
      perms          # 保留权限属性。
      recursive      # 递归到目录中的文件。
      compress       # 传输过程中压缩传输。应该总是开启，除非遇到问题。即rsync的"-z"选项。
      copy_links     # 拷贝软链接的文件名和其指向的文件的内容。即a指向b文件时，将在目标端生成a普通
                     # 文件，但此文件中的内容是b中的内容。
      dirs           # 非递归方式传输目录。
      delete         # 目标端如果比源端文件多，则删除这些多出来的文件，要求recursive=yes。
      checksum       # 等价于"-c"选项，将基于文件的checksum来判断是否同步，而不是默认的quick check
                     # 算法，该算法基于文件大小和最近的mtime来判断是否要同步。该选项会大幅降低效率，
                     # 应谨慎使用。注意，它无法影响archive，即archive仍会启用。
      existing_only  # receiver端没有的文件不同步。但仍会传输，只是临时文件重组后不重命名而已。
      partial        # 等价于"--partial"选项。默认rsync在传输中断时会删除传输了一半的文件，指定该选
                     # 项将保留这部分不完整的文件，使得下次传输时可以直接从未完成的数据块开始传输。
      dest_port      # ssh的连接端口。
      rsync_opts     # 指定额外的rsync选项。使用数组的方式传递这些选项。
      rsync_path     # 等价于"--rsync-path"选项，目的是启动远程rsync。
                     # 例如可以指定[--rsync-path=rsync]，甚至[--rsync-path=cd /tmp/c && rsync]。
                     # 当不指定rsync路径时，默认为/usr/bin/rysnc。
      rsync_timeout  # 指定rsync在多久时间内还没有数据传输就超时退出。
      verify_host    # 对目标主机进行ssh的host key验证。
```

包管理模块yum

```
ansible-doc -s yum
- name: Manages packages with the `yum' package manager
  action: yum
    disable_gpg_check   # 安装包时禁止gpgcheck，仅在state=present或latest时生效。
    disablerepo         # 禁用指定的repoid，多个repoid使用逗号分隔。
    enablerepo          # 明确使用该repoid
    exclude             # 排除哪些包不安装，仅在state=present或latest时生效。
    list                # 类似于yum list。
    name=               # 指定安装的包名，可带上版本号。多个包可使用逗号分隔。
    state               # 状态。('present'、'installed','latest')用于安装包，
                        # ('absent'、'removed')用于移除已安装包。
    update_cache        # 强制更新yum的cache。
```

name需要配合state来使用，如果state指定为present/installed/latest将安装包，其中latest是安装最新包，默认为present。如果指定为absent/removed则用于卸载包。

在ansible中，很多地方都会出现present和absent的状态，它们一般都表示目标是否应该存在还是不存在，也就是要进行的动作是创建和删除。

列出和ansible相关的包。

```
ansible centos -m yum -a "list=ansible" -f 6
```

安装包。

```
ansible centos -m yum -a "name=dos2unix state=installed" -o -f 6
```

安装本地的包，且排除某些包不安装。

```
ansible centos -m yum -a "name=/tmp/*.rpm exclude=*unix* state=present"
```

卸载包。

```
ansible centos -m yum -a "name=dos2unix state=removed" -o -f 6
```

配置yum源模块yum_repository

用于配置yum源。可以实现非常完整的yum仓库配置。但是一般只需简单的添加yum源即可。所以，以下是简单版的用法和示例。

```
ansible-doc -s yum_repository
- name: Add or remove YUM repositories
  action: yum_repository
      baseurl      # 地址
      mirrorlist   # 设置mirrorlist地址
      description  # 描述信息
      enabled      # 是否启用该仓库，默认为yes
      file         # 保存此仓库的文件，不设置该项的话则默认以name选项中的名称命名，将自动以".repo"后缀结尾。
      gpgcheck     # 是否要进行gpgcheck
      name=        # 仓库的名称，要保证名称的唯一性
      reposdir     # 保存.repo文件的目录，默认/etc/yum.repos.d/
      state        # repo文件的状态，present/absent，默认present。
```

例如：

```
- name: Add repository
  yum_repository:
    name: aliyun_epel
    description: EPEL YUM repo
    baseurl: http://mirrors.aliyun.com/epel/7/$basearch/

- name: Add multiple repositories into a file
  yum_repository:
    name: epel
    description: EPEL YUM repo
    file: sohu_epel
    baseurl: http://mirrors.sohu.com/fedora-epel/7/$basearch/
    gpgcheck: no
```

```
ansible 192.168.100.63 -m yum_repository -a 'name=aliyun_epel description="epel repo" baseurl=http://mirrors.aliyun
.com/epel/7/$basearch/ gpgcheck=no enabled=yes'
```

查看生成的repo文件。

```
[root@server2 ~]# ansible 192.168.100.63 -m shell -a "cat /etc/yum.repos.d/aliyun_epel.repo"
192.168.100.63 | SUCCESS | rc=0 >>
[aliyun_epel]
baseurl = http://mirrors.aliyun.com/epel/7/$basearch/
enabled = 1
gpgcheck = 0
name = epel repo
```

服务管理模块service

```
ansible-doc -s service
- name: Manage services.
  action: service
      enabled   # 设置服务为开机自启动，默认为no
      name=     # 服务名
      state     # 'started'和'stoped'分别启动和停止服务，它们是幂等操作，多次启动或停止服务的结果是一样的，
                # 也就是说对于运行中的服务不会再执行启动操作，同理停止也是一样。'restarted'总是重启服务，
                # 'reloaded'总是重读配置文件，如果服务是未运行状态，则'reloaded会启动服务。
                # (state和enabled两者至少要给一个)
```

设置httpd开机自启动。

```
ansible centos -m service -a 'name=httpd enabled=yes' -f 6 -o
```

启动httpd服务。

```
ansible centos -m service -a 'name=httpd state=started' -f 6 -o
```

systemd模块

管理systemd风格的服务。

```
ansible-doc -s systemd
- name: Manage services.
  action: systemd
      daemon_reload   # 在执行所有动作之前，先确定是否要reload一次。值为yes/no。
      enabled         # 是否设置开机自启动。
      masked          # 是否将此unit做mask(隐藏、掩盖)处理。mask后的unit将无法启动。
      name=           # 待操作服务名。可以是name，也可以是name.service。
      state           # 'started'/'stopped'具有幂等性。但restarted和reloaded总是会执行。
```

用户管理模块user

同理还有组管理模块group，就不多做说明了。同样，创建用户时，默认会创建同名group。

```
ansible-doc -s user
- name: Manage user accounts
  action: user
      name=             # 要创建、修改、移除的用户名。
      password          # 设置用户密码。此处只能使用加密密码作为值。
      system            # 设置为yes表示创建一个系统用户，只能用于创建，不能用于修改已有用户为系统用户。
      state             # 创建用户(present)还是删除用户(absent)。默认为present。
      createhome        # 创建家目录，或者已有的用户但家目录不存在也会创建。设置为no则不创建家目录。
      home              # 指定要创建的家目录路径
      move_home         # 如果设置为yes，则"home="则表示将家目录移动到此选项指定的路径下。
      uid               # 设置用户的uid
      group             # 设置用户的primary group
      groups            # 将用户加入到辅助组列表中。如果设置"groups="，则会将此用户从所有辅助组中移除。
      shell             # 设置用户的shell。
      force             # 配合'state=absent'时，等价于'userdel --force'，即强制删除用户、家目录和邮件列表。
      remove            # 配合'state=absent'时，等价于'userdel --remove'，即删除家目录和邮件列表。
      update_password   # user是幂等模块，"always"将总是修改密码。"on_create"将只在创建用户时设置密码。
```

创建系统用户，并指定shell。

```
ansible centos -m user -a "name=longshuai system=yes shell=/sbin/nologin"
```

删除用户。

```
ansible centos -m user -a "name=longshuai state=absent"
```

指定`update_password=always`将总是修改用户的密码，不管该用户是否已存在。而`update_password=on_create`则只有新创建用户时才设置密码，如果用户已存在，则不会修改该用户的密码。默认值就是always。

```
openssl passwd -1 123456
$1$9jwmFoVU$MVz7ywscpoPS5WXC.srcP/

ansible centos -m user -a 'name=longshuai3 password="$1$9jwmFoVU$MVz7ywscpoPS5WXC.srcP/" update_password=always'
```

创建用户并指定密码，但如果用户已存在则不修改密码。

```
openssl passwd -1 234567
$1$J9Nt0scL$R9Db5Pi1AJ7FQv4Xzia0w/

ansible centos -m user -a 'name=longshuai3 password="$1$J9Nt0scL$R9Db5Pi1AJ7FQv4Xzia0w/" update_password=on_create'
```

authorzied_key模块

```
ansible-doc -s authorized_key
- name: Adds or removes an SSH authorized key
  action: authorized_key
    key=         # 公钥路径，可以是本地文件，可以是url地址。
                 # 本地文件时使用{{ lookup('file', '~/.ssh/id_rsa.pub') }}，
                 # url使用https://github.com/username.keys。
    manage_dir   # 是否创建或修改目标authorized_keys所在目录的所有者和权限。默认为yes。
                 # 使用自定义的目标路径时，必须设置为no或false
    path         # authorized_keys所在的目录，默认为家目录下的.ssh目录中
    state        # present/absent，是否将密钥添加到目标authorized_keys文件中。
    user=        # 添加到远程哪个用户下的authorized_keys文件
```

例如将公钥添加到centos组中的机器中。使用-k选项用于询问ssh连接的密码。

```
ansible centos -m authorized_key -a "key={{lookup('file','~/.ssh/id_rsa.pub')}} state=present user=root" -k
```

要想使用该模块实现非交互，需要在inventory文件中指定主机或主机组中加上"ansible_ssh_pass='PASSWORD'"参数。例如：

```
192.168.100.65 ansible_ssh_pass='123456'
[centos6]
host1
host2
host3
[centos6:vars]
ansible_ssh_pass='123456'
```

但这样不会将主机密钥添加到主控制端的known_hosts文件中，因此下次执行ansible时会失败。可以在配置文件中做如下设置：

```
host_key_checking = False
```

或者干脆使用expect非交互更方便。

debug模块

用于输出自定义的信息，类似于echo、print等输出命令。ansible中的debug主要用于输出变量值、表达式值，以及用于when条件判断时。使用方式非常简单。

```
ansible-doc -s debug
- name: Print statements during execution
  action: debug
      msg         # 输出自定义信息。如果省略，则输出普通字符。
      var         # 指定待调试的变量。只能指定变量，不能指定自定义信息，且变量不能加{{}}包围，而是直接的变量名。
      verbosity   # 控制debug运行的调试级别，有效值为一个数值N。
```

例如：

```
ansible centos -m debug -a 'msg="i want to print this messages"'
ansible centos -m debug -a 'var=ansible_eth0.ipv4.address'
```

可以输出变量值，不过一般使用到变量的时候都会在playbook中使用debug模块，以下是一个示例：

```
tasks:
  - name: print any messages
      debug: msg="you name is {{ name }}"
```

定时任务模块cron

cron模块用于设置定时任务，也用于管理定时任务中的环境变量。

```
ansible-doc -s cron
- name: Manage cron.d and crontab entries.
  action: cron
      backup          # (yes/no)如果设置了，则会在修改远程cron_file前备份这些文件
      cron_file       # 自定义cron_file的文件名，使用相对路径则表示在/etc/cron.d中。必须同时指定user选项
      user            # 指定哪个用户的crontab将要被修改，默认为root
      disabled        # 禁用crontab中的某个job，要求state=present
      env             # (yes/no)设置一个环境变量，将添加在crontab的顶端。使用name和value定义变量名和值
      job             # 需要执行的命令。如果设置了env，则表示环境变量的值，此时job="XXXX"等价于value="XXXX"。
                      # 要求state=present
      minute          # 分(0-59, *, */N)，不写时，默认为*
      hour            # 时(0-23, *, */N)，不写时，默认为*
      day             # 日(1-31, *, */N)，不写时，默认为*
      month           # 月(1-12, *, */N)，不写时，默认为*
      weekday         # 周(0-6 for Sunday-Saturday, *)，不写时，默认为*
      name            # 描述crontab任务的字符串。但如果设置的是env，则name为环境变量的名称。要求state=absent
                      # 注意，若未设置name，且state=present，则总会创建一个新job条目，即使cron_file中已经存在
                      # 同样的条目
      special_time    # 定时任务的别称，用于定义何时运行job条目。
                      # 有效值有reboot/hourly/daily/weekly/monthly/yearly/annually。
      state           # job或者env的状态是present(默认)还是absent。present用于创建，absent用于移除
```

除了cron模块本身可以管理cron的环境变量，另一个模块cronvar也可以定义定时任务的环境变量。

```
ansible-doc -s cronvar
- name: Manage variables in crontabs
  action: cronvar
      backup         # (yes/no)如果设置了，则会在修改远程cron_file前备份这些文件
      cron_file      # 自定义cron_file的文件名，使用相对路径则表示在/etc/cron.d中
      state          # present用于创建变量，absent用于移除变量
      user           # 指定哪个用户的crontab将要被修改，默认为root
      value          # 环境变量的值，要求state=present
```

例如：创建一个job，每2分钟进行一次时间同步，并且自定义cron_file。

```
ansible centos7 -m cron -a 'name="ntpdate" job="/usr/sbin/ntpdate ntp1.aliyun.com" cron_file=ntpdate_cron user=root
minute=*/2' -o
```

验证是否添加正确。

```
ansible centos7 -m shell -a 'cat /etc/cron.d/ntpdate_cron'
192.168.100.65 | SUCCESS | rc=0 >>
#Ansible: ntpdate
*/2 * * * * root /usr/sbin/ntpdate ntp1.aliyun.com

192.168.100.63 | SUCCESS | rc=0 >>
#Ansible: ntpdate
*/2 * * * * root /usr/sbin/ntpdate ntp1.aliyun.com

192.168.100.64 | SUCCESS | rc=0 >>
#Ansible: ntpdate
*/2 * * * * root /usr/sbin/ntpdate ntp1.aliyun.com
```

移除一个job，要求name必须匹配。如有必要，需要同时指定cron_file和user。

```
ansible centos7 -m cron -a 'name="ntpdate" state=absent cron_file=ntpdate_cron user=root' -o
```

设置crontab环境变量，并定义一个job。

```
ansible centos7 -m cron -a 'env=yes name=app_home value=/tmp' -o
ansible centos7 -m cron -a 'name="ntpdate" job="/usr/sbin/ntpdate ntp1.aliyun.com" minute=*/2' -o
```

验证设置结果。

```
ansible centos7 -m shell -a 'crontab -l'
192.168.100.65 | SUCCESS | rc=0 >>
app_home="/tmp"
#Ansible: ntpdate
*/2 * * * * /usr/sbin/ntpdate ntp1.aliyun.com

192.168.100.63 | SUCCESS | rc=0 >>
app_home="/tmp"
#Ansible: ntpdate
*/2 * * * * /usr/sbin/ntpdate ntp1.aliyun.com

192.168.100.64 | SUCCESS | rc=0 >>
app_home="/tmp"
#Ansible: ntpdate
*/2 * * * * /usr/sbin/ntpdate ntp1.aliyun.com
```

归档模块archive

用于在远端压缩文件。当然，前提是在远端主机上要有对应的压缩工具。支持zip/gz/tar/bz2。

```
ansible-doc -s archive
- name: Creates a compressed archive of one or more files or trees.
  action: archive
      dest          # 目标归档文件名。除非path指定要压缩的是单文件，否则需要dest选项
      format        # 指定压缩格式，默认为gz格式
      group         # 文件/目录的所属组
      owner         # 文件/目录的所有者
      mode          # 设置文件/目录的的权限，支持'0644'或'u+rwx'或'u=rw,g=r,o=r'等格式
      path=         # 要压缩的文件，可以是绝对路径，也可以是glob统配的路径，还可以是文件列表
      remove        # 压缩后删除源文件
```

例如：

```
# 将目录/path/to/foo/压缩为/path/to/foo.tgz
- archive:
    path: /path/to/foo
    dest: /path/to/foo.tgz

# 压缩普通文件/path/to/foo为/path/to/foo.gz并删除源文件，由于压缩的是单文件，所以可以省略dest选项
- archive:
    path: /path/to/foo
    remove: True

# 将单文件/path/to/foo压缩为zip格式
- archive:
    path: /path/to/foo
    format: zip

# 将给定的文件列表压缩为bz2格式，压缩包路径为/path/file.tar.bz2
- archive:
    path:
        - /path/to/foo
        - /path/wong/foo
    dest: /path/file.tar.bz2
    format: bz2
```

解包模块unarchive

默认复制ansible端的归档文件到被控主机，然后在被控主机上进行解包。如果设置选项remote_src=yes，则表示解包被控主机上的归档文件。

要求在被控主机上有对应的解包命令。unzip命令用于解压".zip"文件，gtar(tar包提供)命令用于解压".tar"、".tar.gz"、".tar.bz2"和".tar.xz"。

```
ansible-doc -s unarchive
- name: Unpacks an archive after (optionally) copying it from the local machine.
  action: unarchive
      creates      # 如果指定的文件存在则不执行该任务。可用于实现幂等性
      dest=        # 远程机器上需要被解包的归档文件，要求是绝对路径
      exclude      # 列出解包过程中想要忽略的目录和文件
      group        # 文件/目录的所属组
      owner        # 文件/目录的所有者
      mode         # 设置文件/目录的的权限，支持'0644'或'u+rwx'或'u=rw,g=r,o=r'等格式
      keep_newer   # 在解包过程中，如果目标路径中和包中有同名文件，且比包中的文件更新，则保留新的文件
      list_files   # 设置为true时，将返回归档文件中的文件列表
      remote_src   # 设置为yes表示远程主机上已有目标归档文件，即不再从本地复制归档文件到远端，直接在远端解包。
                   # 默认为no
      src=         # 如果remote_src=no,将复制本地归档文件到远端，可相对路径也可绝对路径.
                     如果remote_src=yes，将解包远程已存在的归档文件
                     如果remote_src=yes且src中包含了"://",将指挥远程主机从url中下载文件并解包
```

例如：

```
# 复制ansible端的foo.tgz文件到远端并解包
- unarchive:
    src: foo.tgz
    dest: /var/lib/foo

# 直接解包远端已经存在的文件- unarchive:
    src: /tmp/foo.zip
    dest: /usr/local/bin
    remote_src: True

# 从url上下载压缩包，然后进行解压
- unarchive:
    src: https://example.com/example.zip
    dest: /usr/local/bin
    remote_src: True
```

下载模块get_url

```
ansible-doc -s get_url
- name: Downloads files from HTTP, HTTPS, or FTP to node
  action: get_url
      backup    # 下载文件时同时创建一个名称中包含时间戳的备份文件
      dest=     # 文件保存路径，必须为绝对路径。
                # 如果dest是一个目录，则使用url的base name作为文件名
                # 如果dest是一个目录，则'force'选项不生效
                # 如果dest是一个目录，则总是会下载目标文件，但只在已存在的文件变化了才会替换旧文件
      force     # 如果设置为yes，且dest不是一个目录时，则总是会下载文件，但只在已存在的文件变化了才会替换旧文件
                # 如果设置为no(默认)，则只会在目录路径下不存在该文件时才会进行下载。
      tmp_dest  # 下载时临时存放目录，在任务执行完成前会删除下载的临时文件
      group     # 文件/目录的所属组
      owner     # 文件/目录的所有者
      mode      # 设置文件/目录的的权限，支持'0644'或'u+rwx'或'u=rw,g=r,o=r'等格式
      timeout   # 请求url时的超时时间，默认10秒钟
      url=      # 要下载的url路径，(http|https|ftp)://[user[:pass]]@host.domain[:port]/path
                # 还支持file格式的路径，实现复制功能。file:///path/to/file
```

注意，dest为目录或者force=yes时，总是会下载文件到临时存放目录中，只不过不一定会替换旧文件。只有force=no(默认)且dest是一个文件时，在文件已存在时才不会下载文件。

例如：

```
# 下载foo.conf，若/etc/foo.conf已存在，则不下载该文件
  get_url:
    url: http://example.com/path/file.conf
    dest: /etc/foo.conf
    mode: 0440

# 下载foo.conf到/tmp目录下，若/tmp/foo.conf已存在则检查文件是否相同，不相同则进行替换
  get_url:
    url: http://example.com/path/file.conf
    dest: /tmp/

# 复制一个本地文件
  get_url:
    url: file:///tmp/afile.txt
    dest: /tmp/afilecopy.txt
```

wait_for模块

有些时候任务之间对状态、文件、端口等资源是有依赖关系的，只有满足了前提，任务才会继续。wait_for模块就是用于判断任务在满足什么条件的情况下会继续。主要用来判断端口是否开启、文件是否存在、文件中是否存在某些字符串。

```
ansible-doc -s wait_for
- name: Waits for a condition before continuing.
  action: wait_for
      delay          # 在检查操作进行之前等待的秒数
      host           # 等待这个主机处于启动状态，默认为127.0.0.1
      port           # 等待这个端口已经开放
      path           # 这个文件是否已经存在
      search_regex   # 在文件中进行正则匹配
      state          # present/started/stopped/absent/drained.默认started
                       当检查的是一个端口时:
                         started:保证端口是开放的
                         stopped:保证端口是关闭的
                        当检查的是一个文件时:
                         present/started:在检查到文件存在才会继续
                         absent:检查到文件被移除后才会继续
      sleep          # 两次检查之间sleep的秒数，默认1秒
      timeout        # 检查的等待超时时间(秒数，默认300)
```

例如：

```
# 连接上主机后10秒后才检查8000端口是否处于开放状态，300秒(默认值)内未开放则超时。
- wait_for:
    port: 8000
    delay: 10

# 直到/tmp/foo文件存在才会继续
- wait_for:
    path: /tmp/foo

# 直到/tmp/foo文件中能匹配"completed"字符串才继续
- wait_for:
    path: /tmp/foo
    search_regex: completed

# 直到/var/lock/file.lock这个锁文件被移除了才继续
- wait_for:
    path: /var/lock/file.lock
    state: absent

# 直到/proc/3466/status文件被移除才继续，可用来判断进程是启动还是停止，pid文件是存在还是被移除等
- wait_for:
    path: /proc/3466/status
    state: absent
```

script模块

script模块用于控制远程主机执行脚本。在执行脚本前，ansible会将本地脚本传输到远程主机，然后再执行。在执

行脚本的时候，其采用的是远程主机上的shell环境。

```
ansible-doc -s script
- name: Runs a local script on a remote node after transferring it
  action: script
      chdir          # 在远程执行脚本前先切换到此目录下。
      creates        # 当此文件存在时，不执行脚本。可用于实现幂等性。
      removes        # 当此文件不存在时，不执行脚本。可用于实现幂等性。
      free_form=     # 本地待执行的脚本路径、选项、参数。之所以称为free_form，是因为它是脚本名+选项+参数。
```

例如，将ansible端/tmp/a.sh发送到各被控节点上执行，但如果被控节点的/tmp下有hello.txt，则不执行。

```
---
    - hosts: centos
      remote_user: root
      tasks:
        - name: execute /tmp/a.sh,but only /tmp/hello.txt is not yet created
          script: /tmp/a.sh hello
          args:
            creates: /tmp/hello.txt
```

3.YAML语法和playbook写法

ansible的playbook采用yaml语法，它简单地实现了json格式的事件描述。yaml之于json就像markdown之于html一样，极度简化了json的书写。在学习ansible playbook之前，很有必要把yaml的语法格式、引用方式做个梳理。

3.1 初步说明

以一个简单的playbook为例，说明yaml的基本语法。

```
---
    - hosts: 192.168.100.59,192.168.100.65
      remote_user: root
      pre_tasks:
        - name: set epel repo for Centos 7
          yum_repository:
            name: epel7
            description: epel7 on CentOS 7
            baseurl: http://mirrors.aliyun.com/epel/7/$basearch/
            gpgcheck: no
            enabled: True

      tasks:
# install nginx and run it
        - name: install nginx
          yum: name=nginx state=installed update_cache=yes
        - name: start nginx
          service: name=nginx state=started

      post_tasks:
        - shell: echo "deploy nginx over"
          register: ok_var
        - debug: msg="{{ ok_var.stdout }}"
```

1. yaml文件以`---`开头，以表明这是一个yaml文件，就像xml文件在开头使用`<?xml version="1.0" encoding="utf-8"?>`宣称它是xml文件一样。但即使没有使用`---`开头，也不会有什么影响。

2. yaml中使用"#"作为注释符，可以注释整行，也可以注释行内从"#"开始的内容。

3. yaml中的字符串通常不用加任何引号，即使它包含了某些特殊字符。但有些情况下，必须加引号，最常见的是在引用变量的时候。具体见后文。

4. 关于布尔值的书写格式，即true/false的表达方式。其实playbook中的布尔值类型非常灵活，可分为两种情况：

- 模块的参数： 这时布尔值作为字符串被ansible解析。接受yes/on/1/true/no/off/0/false，这时被ansible解析。例如上面示例中的`update_cache=yes`。
- 非模块的参数： 这时布尔值被yaml解释器解析，完全遵循yaml语法。接受不区分大小写的true/yes/on/y/false/no/off/n。例如上面的`gpgcheck=no`和`enabled=True`。

  建议遵循ansible的官方规范，模块的布尔参数采用yes/no，非模块的布尔参数采用True/False。

3.2 列表

使用"- "(减号加一个或多个空格)作为列表项，也就是json中的数组。yaml的列表在playbook中极重要，必须得搞

清楚它的写法。

例如：

```
- zhangsan
- lisi
- wangwu
```

还支持内联写法：使用中括号。

```
[zhangsan,lisi,wangwu]
```

它们等价于json格式的：

```
[
    "zhangsan",
    "lisi",
    "wangwu"
]
```

再例如：

```
- 班名：初中1班
  人数：35
  班主任：隔壁老张
  今天的任务：扫操场

- 班名：初中2班
  人数：38
  班主任：隔壁老王
  今天的任务：搬桌子
```

具体在ansible playbook中，列表所描述的是**局部环境**，它不一定要有名称，不一定要从同一个属性开始，只要使用"- "，它就表示圈定一个范围，范围内的项都属于该列表。例如：

```
---
    - name: list1               # 列表1，同时给了个名称
      hosts: localhost          # 指出了hosts是列表1的一个对象
      remote_user: root         # 列表1的属性
      tasks:                    # 还是列表1的属性

    - hosts: 192.168.100.65     # 列表2，但是没有为列表命名，而是直入主题
      remote_user: root
      sudo: yes
      tasks:
```

唯一要注意的是，每一个playbook中必须包含"hosts"和"tasks"项。更严格地说，是**每个play**的顶级列表必须包含这两项。就像上面的例子中，就表示该playbook中包含了**两个play**，每个play的顶级列表都包含了hosts和tasks。其实绝大多数情况下，一个playbook中都只定义一个play，所以只有一个顶级列表项。**顶级列表的各项，其实可以将其看作是ansible-playbook运行时的选项**。

另外，playbook中某项是一个动作、一个对象或一个实体时，一般都定义成列表的形式。见下文。

3.3 字典

官方手册上这么称呼，其实就是key=value的另一种写法。使用"冒号+空格"分隔，即`key: value`。它一般当作列表项的属性。

例如：

```
- 班名: 初中1班
  人数:
    总数: 35
    男: 19
    女: 16
  班主任:
    大名: 隔壁老张
    这厮多大: 39
    这厮任教多少年: 15
  今天的任务: 扫操场

- 班名: 初中2班
  人数:
    总数: 38
    男: 19
    女: 19
  班主任:
    大名: 隔壁老王
    这厮多大: 30
    喜调戏女老师: True
  今天的任务: 搬桌子
     未完成任务怎么办:
        - 继续搬，直到完成
        - 写检讨
```

具体到playbook中，一般"虚拟性"的内容都可以通过字典的方式书写，而实体化的、动作性的、对象性的内容则应该定义为列表形式。

```
---
    - hosts: localhost              # 列表1
      remote_user: root
      tasks:
        - name: test1               # 子列表，下面是shell模块，是一个动作，所以定义为列表，只不过加了个name
          shell: echo /tmp/a.txt
          register: hi_var
        - debug: var=hi_var.stdout  # 调用模块，这是动作，所以也是列表
        - include: /tmp/nginx.yml   # 同样是动作，包含文件
        - include: /tmp/mysql.yml
        - copy:                     # 调用模块，定义为列表。但模块参数是虚拟性内容，应定义为字典而非列表
            src: /etc/resolv.conf   # 模块参数1
            dest: /tmp              # 模块参数2

    - hosts: 192.168.100.65           # 列表2
      remote_user: root
      vars:
        nginx_port: 80                # 定义变量，是虚拟性的内容，应定义为字典而非列表
        mysql_port: 3306
      vars_files:
        - nginx_port.yml              # 无法写成key/value格式，且是实体文件，因此定义为列表
      tasks:
        - name: test2
          shell: echo /tmp/a.txt
          register: hi_var            # register是和最近一个动作绑定的
        - debug: var=hi_var.stdout
```

从上面示例的copy模块可以得出，模块的参数是虚拟性内容，也能使用字典的方式定义。

字典格式的key/value，也支持内联格式写法：使用大括号。

```
{大名: 隔壁老王,这厮多大: 30,喜调戏女老师: True}
{nginx_port: 80,mysql_port: 3306}
```

这等价于json格式的：

```
{
    "大名": "隔壁老王",
    "这厮多大": 30,
    "喜调戏女老师": "True"
}
{
    "nginx_port": 80,
    "mysql_port": 3306
}
```

再结合其父项，于是转换成json格式的内容：

```
"班主任": {
    "大名": "隔壁老王",
    "这厮多大": 30,
    "喜调戏女老师": "True"
}

"vars": {
    "nginx_port": 80,
    "mysql_port": 3306
}
```

再加上列表项(使用中括号)，于是：

```
[
  {
    "hosts": "192.168.100.65",
    "remote_user": "root",
    "vars": {
      "nginx_port": 80,
      "mysql_port": 3306
    },
    "vars_files": [
      "nginx_port.yml"
    ],
    "tasks": [
      {
        "name": "test2",
        "shell": "echo /tmp/a.txt",
        "register": "hi_var"
      },
      {
        "debug": "var=hi_var.stdout"
      }
    ]
  }
]
```

3.4 分行写

playbook中有3种方式进行续行。

- 在"key:"的后面使用大于号。
- 在"key:"的后面使用竖线。这种方式可以像脚本一样写很多行语句。
- 多层缩进。

例如，下面的3中方法。

```
---
    - hosts: localhost
      tasks:
        - shell: echo 2 >>/tmp/test.txt
            creates=/tmp/haha.txt          # 比模块shell缩进更多
        - shell: >                         # 在"key: "后使用大于号
            echo 2 >>/tmp/test.txt
            creates=/tmp/haha.txt
        - shell: |                         # 指定多行命令
            echo 2 >>/tmp/test.txt
            echo 3 >>/tmp/test.txt
          args:
            creates: /tmp/haha.txt
```

3.5 向模块传递参数

模块的参数一般来说是key=value格式的，有3种传递的方式：

- 直接写在模块后，此时要求使用"key=value"格式。这是让ansible内部去解析字符串。因为可分行写，所以有多种写法。
- 写成字典型，即"key: value"。此时要求多层缩进。这是让yaml去解析字典。
- 使用内置属性args，然后多层缩进定义参数列表。这是让ansible明确指定用yaml来解析。

例如：

```
---
    - hosts: localhost
      tasks:
        - yum: name=unix2dos state=installed    # key=value直接传递
        - yum:
            name: unxi2dos
            state: installed            # "key: value"字典格式传递
        - yum:
          args:                                 # 使用args传递
            name: unix2dos
            state:installed
```

但要注意，当模块的参数是`free_form`时，即格式不定，例如shell和command模块指定要执行的命令，它无法写成key/value格式，此时不能使用上面的第二种方式。也就是说，下面第一个模块是正确的，第二个模块是错误的，因为shell模块的命令"echo haha"是自由格式的，无法写成key/value格式。

```
---
    - hosts: localhost
      tasks:
        - yum:
            name: unxi2dos
            state: installed
        - shell:
            echo haha
            creates: /tmp/haha.txt
```

所以，调用一个模块的方式就有了多种形式。例如：

```
---
    - hosts: localhost
      tasks:
        - shell: echo 1 >/tmp/test.txt creates=/tmp/haha.txt
        - shell: echo 2 >>/tmp/test.txt
            creates=/tmp/haha.txt
        - shell: echo 3 >>/tmp/test.txt
          args:
             creates: /tmp/haha.txt
        - shell: >
            echo 4 >>/tmp/test.txt
            creates=/tmp/haha.txt
        - shell: |
            echo 5.1 >>/tmp/test.txt
            echo 5.2 >>/tmp/test.txt
          args:
            creates: /tmp/haha.txt
        - yum:
            name: dos2unix
            state: installed
```

3.6 playbook和play的关系

一个playbook中可以包含多个play。每个play都至少包含有tasks和hosts这两项，还可以包含其他非必须项，如vars,vars_files,remote_user等。tasks中可以通过模块调用定义一系列的action。只不过，绝大多数时候，一个playbook都只定义一个play。

所以，大致关系为：

- playbook: [play1,play2,play3]
- play: [hosts,tasks,vars,remote_user...]
- tasks: [module1,module2,...]

也就是说，每个顶级列表都是一个play。例如，下面的playbook中包含了两个play。

```
---
    - name: list1
      hosts: localhost
      remote_user: root
      tasks:

    - hosts: 192.168.100.65
      remote_user: root
      sudo: yes
      tasks:
```

需要注意，有些时候play中使用了role，可能看上去没有tasks，这是因为role本身就是整合playbook的，所以没有也没关系。但没有使用role的时候，必须得包含hosts和tasks。例如：

```
---
  - hosts: centos
    remote_user: root
    pre_tasks:
        - name: config the yum repo for centos 7
          yum_repository:
              name: epel
              description: epel
              baseurl: http://mirrors.aliyun.com/epel/7/$basearch/
              gpgcheck: no
          when: ansible_distribution_major_version == "7"

        - name: config the yum repo for centos 6
          yum_repository:
              name: epel
              description: epel
              baseurl: http://mirrors.aliyun.com/epel/6/$basearch/
              gpgcheck: no
          when: ansible_distribution_major_version == "6"

    roles:
        - nginx

    post_tasks:
      - shell: echo 'deploy nginx/mysql over'
        register: ok_var
      - debug: msg='{{ ok_var.stdout }}'
```

3.7 playbook中什么时候使用引号

playbook中定义的都是些列表和字典。绝大多数时候，都不需要使用引号，但有两个特殊情况需要考虑使用引号。

- 出现大括号"{}"。
- 出现冒号加空格时": "。

大括号要使用引号包围，是因为不使用引号时会被yaml解析成内联字典。例如要使用大括号引用变量的时候，以及想输出大括号符号的时候。

```
---
    - hosts: localhost
      tasks:
        - shell: echo "{{inventory_hostname}}:haha"
```

冒号尾随空格时要使用引号包围，是因为它会被解析为"key: value"的形式。而且包围冒号的引号还更严格。例如下面的debug模块中即使使用了引号也是错误的。

```
---
    - hosts: localhost
      tasks:
        - shell: echo "{{inventory_hostname}}:haha"
          register: hello
        - debug: msg="{{hello.stdout}}: heihei"
```

因为它把{{...}}当成key，heihei当成value了。因此，必须将整个debug模块的参数都包围起来，显式指定这一段是模块的参数。但这样会和原来的双引号冲突，因此使用单引号。

```
---
    - hosts: localhost
      tasks:
        - shell: echo "{{inventory_hostname}}:haha"
          register: hello
        - debug: 'msg="{{hello.stdout}}: heihei"'
```

但是，如果将shell模块中的冒号后也尾随上空格，即写成`echo "{{inventory_hostname}}: haha"`，那么shell模块也会

报错。因此也要使用多个引号，正确的如下：

```
---
    - hosts: localhost
      tasks:
        - shell: 'echo "{{inventory_hostname}}: haha"'
          register: hello
        - debug: 'msg="{{hello.stdout}}: heihei"'
```

4.playbook应用和示例

playbook是ansible实现批量自动化最重要的手段。在其中可以使用变量、引用、循环等功能，相比ad-hoc而言，其功能要强大的多。

4.1 yaml简单示例

ansible的playbook采用yaml语法。以下是一个yaml格式的文件：

```
---
# Members in Bob's family
    name: Bob
    age: 30
    gender: Male
    wife:
        name: Alice
        age: 27
        gender: Female
    children:
      - name: Jim
        age: 6
        gender: Male
      - name: Lucy
        age: 3
        gender: Female
```

4.2 ansible-playbook命令说明及playbook书写简单示例

以下是一个简单的playbook示例。该示例执行两个任务，第一个任务是执行一个/bin/date命令，第二个任务是复制/etc/fstab文件到目标主机上的/tmp下，它们分别使用了ansible的command模块和copy模块。

```
cat /tmp/test.yaml
---
    - hosts: centos7
      tasks:
        - name: execute date cmd
          command: /bin/date
        - name: copy fstab to /tmp
          copy: src=/etc/fstab dest=/tmp
```

书写好playbook后，使用ansible-playbook命令来执行。ansible-playbook命令的选项和ansible命令选项绝大部分都相同。但也有其特有的选项。以下是截取出来的帮助信息。

```
ansible-playbook --help
Usage: ansible-playbook playbook.yml

Options:
  -e EXTRA_VARS,--extra-vars=EXTRA_VARS # 设置额外的变量，格式为key/value。-e "key=KEY",
                                        # 如果是文件方式传入变量，则-e "@param_file"
  --flush-cache          # 清空收集到的fact信息缓存
  --force-handlers       # 即使task执行失败，也强制执行handlers
  --list-tags            # 列出所有可获取到的tags
  --list-tasks           # 列出所有将要被执行的tasks
  -t TAGS,--tags=TAGS    # 以tag的方式显式匹配要执行哪些tag中的任务
  --skip-tags=SKIP_TAGS  # 以tag的方式忽略某些要执行的任务。被此处匹配的tag中的任务都不会执行
  --start-at-task=START_AT_TASK # 从此task开始执行playbook
  --step                 # one-step-at-a-time:在每一个任务执行前都进行交互式确认
  --syntax-check         # 检查playbook语法
```

现在执行上面的test.yaml。

```
ansible-playbook /tmp/test.yaml

PLAY [centos7] *****************************************************************

TASK [Gathering Facts] *********************************************************
ok: [192.168.100.64]
ok: [192.168.100.65]
ok: [192.168.100.63]

TASK [execute date cmd] ********************************************************
changed: [192.168.100.63]
changed: [192.168.100.64]
changed: [192.168.100.65]

TASK [copy fstab to /tmp] ******************************************************
changed: [192.168.100.64]
changed: [192.168.100.63]
changed: [192.168.100.65]

PLAY RECAP *********************************************************************
192.168.100.63             : ok=3    changed=2    unreachable=0    failed=0
192.168.100.64             : ok=3    changed=2    unreachable=0    failed=0
192.168.100.65             : ok=3    changed=2    unreachable=0    failed=0
```

从以上结果中，可以看出：(1)默认情况下，ansible-playbook和ansible是一样的，都是同步阻塞模式，需要先在所有主机上执行完一个任务，才会继续下一个任务；(2)在执行前会自动收集fact信息；(3)从显示结果中可以判断出任务是否真的执行了，抑或者是因为幂等性而没有执行。(4)每一个play都包含数个task，且都有响应信息play recap。

4.3 playbook的内容

4.3.1 hosts和remoter_user

对于playbook中的每一个play，使用hosts选项可以定义要执行这些任务的主机或主机组，还可以使用remote_user指定在远程主机上执行任务的用户，实际上remote_user是ssh连接到被控主机上的用户，自然而然执行命令的身份也将是此用户。

例如：

```
---
    - hosts: centos6,centos7,192.168.100.59
      remote_user: root
      tasks: XXXXXX
```

虽然在hosts处可以使用","分隔主机或主机组，但官方手册上并没有介绍该方法。除此之外，有以下几种指定主机和主机组的方式：

- `all`或`*`：表示inventory中的所有主机。
- `:`：取并集。例如"host1:host2:group1"表示2台主机加一个主机组。
- `:&`：取交集。例如"group1:&group2"表示两个主机组中都有的主机。
- `:!`：排除。例如"group1:!host1"表示group1中排除host1主机的剩余主机。
- `通配符`：例如"web*.baidu.com"。
- `数字范围`：例如"web[0-5].baidu.com"。
- `字母范围`：例如"web[a-d].baidu.com"。
- `正则表达式`：以"~"开头。例如"~web\d\.baidu\.com"。

此外，在ansible命令行或ansible-playbook命令行中，可以使用"-l"选项来限定执行任务的主机。例如：

```
ansile centos -l host[1:5] -m ping
```

表示centos主机组中只有host1到host5才执行ping模块。

还可以在某个task上单独定义执行该task的身份，这将覆盖全局的定义。

```
---
    - hosts: centos6,centos7,192.168.100.59
      remote_user: root
      tasks:
        - name: run a command
          shell: /bin/date
        - name: copy a file to /tmp
          copy: src=/etc/fstab dest=/tmp
          remote_user: myuser
```

也支持权限升级的方式。

```
---
    - hosts: centos6,centos7,192.168.100.59
      remote_user: yourname
      tasks:
        - name: run a command
          shell: /bin/date
        - name: copy a file to /tmp
          copy: src=/etc/fstab dest=/tmp
          become: yes
          become_method: sudo
          become_user: root    # 此项默认值就是为root，所以可省
```

从上面的示例可以看出remote_user实际上并不是执行任务的绝对身份，它只是ssh连接过去的身份，只不过没有指定become的时候，它正好就用此身份来运行任务。

4.3.2 task list

1.特性

每个play都包含一个hosts和一个tasks，hosts定义的是inventory中待控制的主机，tasks下定义的是一系列task任务列表，比如调用各个模块。这些task按顺序一次执行一个，直到所有被筛选出来的主机都执行了这个task之后才会移动到下一个task上进行同样的操作。

需要注意的是，虽然只有被筛选出来的主机会执行对应的task，但是所有主机(此处的所有主机表示的是，hosts选项所指定的那些主机)都会收到相同的task指令，所有主机收到指令后，ansible主控端会筛选某些主机，并通过ssh在远程执行任务。也就是说，如果查看`ansible-playbook -vvvv`的信息，将会发现临时任务文件会通过sftp发送到所有的被控主机上，但是只有一部分被筛选(如果进行了筛选)的主机才会ssh过去并远程执行命令。

当某一台被控主机执行某个任务出错或失败时，它将会被移除出任务轮询列表。也就是说，对于某主机来说，某任务执行失败，后续的所有任务都不会再去执行。当然，这不会影响其他的主机执行任务(除非主机的任务之间有依赖关系)。

最重要的是，ansible中的task是幂等性的，多次执行不会影响那些成功执行过的任务。另外幂等性还表现在执行失败后如果修正了playbook再次执行，将不会影响那些原本已经执行成功的任务，即使是不同主机也不会影响。仅这方面而言，ansible对于排错来说是极其友好的。当然，某些特殊的模块或者特殊定义的task并不一定总是幂等的，例如最简单的，执行一个command或者shell模块的命令，它会重复执行。但也有办法使其变得幂等，以command和shell模块为例，它们有两个选项：creates和removes，它们分别表示被控主机上指定的文件存在(不存在)时就不执行命令。

2.定义task的细节

- (1).可以为每个task加上name项，也可以多个task依赖于一个name。

例如下面的两个例子。从两个示例中可以看出，两个task其实都是属于一个name的，第二个task无需再使用name命名。

示例一：

```
tasks:
    - name: do something to initialize mariadb
      file: path=/mydata/data state=directory owner=mysql group=mysql mode=0755
    - shell: /usr/bin/mysql_install_db --datadir=/mydata/data --user=mysql creates=/mydata/data/ibdata1
```

示例二：

```
tasks:
    - name: echo var passed by nginx
      shell: echo "{{ hi_var }}"
      register: var_result
    - debug: msg="{{ var_result.stdout }}"
```

实际上，name只是一种描述性语句，它可以定义在任何地方。例如，定义在play的顶端。

```
---
- name: start a play
  hosts: localhost
  tasks:
```

- (2).既然是task，那么必然会有其要执行的一个或多个任务，其本质是加载并执行ansible对应的模块。在playbook中，每调用的一个模块都称为一个action。

例如，定义一个确保服务是启动状态的task，有以下三种方法传递模块参数：

```
tasks:
  - name: be sure the sshd is running
    service: name=sshd state=started     # 方法一：  定义为key=value，直接传递参数给模块

    service:                             # 方法二：  定义为key: value方式
      name: sshd
      state: started

    service:                  # 方法三：  使用args内置关键字，然后定义为key: value方式
    args:
      name: sshd
      state: started
```

但要注意，ping模块、command和shell模块是不需要 `key=value` 格式的。对于ping命令，可以直接省略 `key=value`。对于command和shell，只需要给定命令路径和要接上去的选项或参数即可，且无法使用上面的方法二。例如下面定义的是一个ntpdate命令，只需给定它的参数即可。

```
tasks:
    - name: execute command ntpdate
      shell: /usr/sbin/ntpdate ntp1.aliyun.com
    - name: ping host
      ping:
```

对于command或shell模块来说，有时候要考虑命令的返回状态码。如果要忽略非0状态码继续执行任务，可以使用以下两种方式：

```
tasks:
    - name: ignore non_zero return code
      shell: /usr/sbin/ntpdate ntp1.aliyun.com || /bin/true
```

或者

```
tasks:
    - name: another way to ignore the non_zero return code
      shell: /usr/sbin/ntpdate ntp1.aliyun.com
      ignore_errors: true
```

- (3).如果action的key=value太多，导致内容太长，可以在上一行的缩进级别基础上继续缩进表示续行。

例如，下面的owner比src多缩进了4个空格。

```
tasks:
  - name: Copy ansible inventory file to client
    copy: src=/etc/fstab dest=/tmp
              owner=root group=root mode=0644
```

- (4).在action的value部分，可以引用已经定义的变量，可以是已定义好的自定义的变量，也可以是内置变量。变量相关内容见后文。

- (5).使用include指令，可以将其他的playbook文件包含到此playbook文件中。include的方式见下文。

4.3.3 notify和handler

ansible中几乎所有的模块都具有幂等性，这意味着被控主机的状态是否发生改变是能被捕捉的，即每个任务的changed=true或changed=false。ansible在捕捉到changed=true时，可以触发notify组件(如果定义了该组件)。

notify是一个组件，并非一个模块，它可以直接定义action，其主要目的是调用handler。例如：

```
tasks:
    - name: copy template file to remote host
      template: src=/etc/ansible/nginx.conf.j2 dest=/etc/nginx/nginx.conf
      notify:
        - restart nginx
        - test web page
      copy: src=nginx/index.html.j2 dest=/usr/share/nginx/html/index.html
      notify:
        - restart nginx
```

这表示当执行template模块的任务时，如果捕捉到changed=true，那么就会触发notify，如果分发的index.html改变了，那么也重启nginx(当然这是没必要的)。notify下定义了两个待调用的handler。handler主要用于重启服务或者触发系统重启，除此之外很少使用handler。以下是这两个handler的内容：

```
handlers:
    - name: restart nginx
      service: name=nginx state=restarted
    - name: test web page
      shell: curl -I http://192.168.100.10/index.html | grep 200 || /bin/false
```

handler的定义和tasks的定义完全一样，唯一需要限定的是handler中task的name必须和notify中定义的名称相同。

注意，notify是在执行完一个play中所有task后被触发的，在一个play中也只会被触发一次。意味着如果一个play中有多个task出现了changed=true，它也只会触发一次。例如上面的示例中，向nginx复制配置文件和复制index.html时如果都发生了改变，都会触发重启apache操作。但是只会在执行完play后重启一次，以避免多余的重

启。

4.3.4 标签tag

可以为playbook中的每个任务都打上标签，标签的主要作用是可以在ansible-playbook中设置只执行哪些被打上tag的任务或忽略被打上tag的任务。

```
tasks:
   - name: make sure apache is running
     service: name=httpd state=started
     tags: apache
   - name: make sure mysql is running
     service: name=mysqld state=started
     tags: mysql
```

以下是ansible-playbook命令关于tag的选项。

```
  --list-tags           # list all available tags
  -t TAGS, --tags=TAGS  # only run plays and tasks tagged with these values
  --skip-tags=SKIP_TAGS # only run plays and tasks whose tags do not match these values
```

4.4 include和roles

如果将所有的play都写在一个playbook中，很容易导致这个playbook文件变得臃肿庞大，且不易读。因此，可以将多个不同任务分别写在不同的playbook中，然后使用include将其包含进去即可。而role则是整合playbook的方式。无论是include还是role，其目的都是分割大playbook以及复用某些细化的play甚至是task。

4.4.1 include

可以将task列表和handlers独立写在其他的文件中，然后在某个playbook文件中使用include来包含它们。除此之外，还可以写独立的playbook文件，使用include来包含这个文件。

也即是说，include可以导入两种文件：导入task、导入playbook。

1.一种是任务列表式的文件(没有tasks或handlers指令)，它只能在tasks或handlers指令的子选项处使用include包含。这种方式可以传递变量到被包含的文件中。

假设某个task列表文件/yaml/a.yaml内容如下：

```
---
    - name: execute ntpdate
      shell: /usr/sbin/ntpdate ntp1.aliyun.com
```

在同目录/yaml下有一个名为test.yaml的playbook(除了role，playbook中所有相对路径都是基于playbook的)，在此playbook中使用include来包含它，如果使用相对路径将会包含同目录下的文件。

```
---
    - hosts: centos7
      tasks:
       - include: a.yaml
```

可以在include的时候传递变量给对应的文件，这样在被包含的文件中就可以引用该变量的值。

```
---
  - hosts: centos7
    tasks:
      - include: a.yaml sayhi="hello world"
```

或者

```
---
  - hosts: centos7
    tasks:
       - include: a.yaml
         vars:
           sayhi: "hello world"
```

然后可以在被包含的文件a.yaml中使用该变量。例如：

```
---
    - name: execute ntpdate
      shell: /usr/sbin/ntpdate ntp1.aliyun.com
    - name: say hi to world
      debug: msg="{{ sayhi }}"
```

执行该test.yaml，将会输出对应的变量值。

```
ansible-playbook /yaml/test.yaml

PLAY [centos7] *****************************************************************

TASK [Gathering Facts] *********************************************************
ok: [192.168.100.64]
ok: [192.168.100.63]
ok: [192.168.100.65]

TASK [execute ntpdate] *********************************************************
changed: [192.168.100.64]
changed: [192.168.100.65]
changed: [192.168.100.63]

TASK [sayhi to world] **********************************************************
ok: [192.168.100.63] => {
    "changed": false,
    "msg": "hello world"
}
ok: [192.168.100.65] => {
    "changed": false,
    "msg": "hello world"
}
ok: [192.168.100.64] => {
    "changed": false,
    "msg": "hello world"
}

PLAY RECAP *********************************************************************
192.168.100.63             : ok=3    changed=1    unreachable=0    failed=0
192.168.100.64             : ok=3    changed=1    unreachable=0    failed=0
192.168.100.65             : ok=3    changed=1    unreachable=0    failed=0
```

2.另一种是include整个playbook文件，即include的动作是加载一个或多个play，所以写在顶级列表的层次。

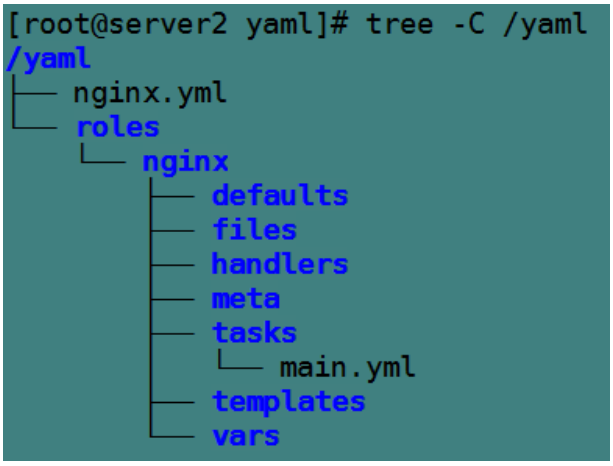
```
- name: this is a play at the top level of a file
  hosts: all
  remote_user: root

  tasks:

  - name: say hi
    tags: foo
    shell: echo "hi..."

- include: load_balancers.yml  sayhi="hello world"
- include: webservers.yml
- include: dbservers.yml

    any other operations
```

需要说明的是，在ansible 2.4版本中，添加了includes和imports两种导入的方式，它们对静态和动态导入支持的更细化，而ansible 2.3及以前的include语句已经废弃，但仍可用。相关官方手册地址。

4.4.2 roles

roles意为角色，主要用于封装playbook实现复用性。在ansible中，roles通过文件的组织结构来展现。

对于一个role，它的文件组织结构如下图所示。

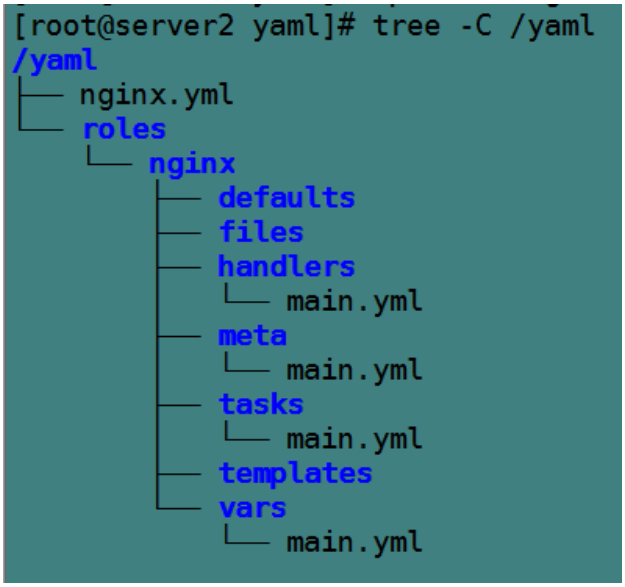


首先需要有一个roles目录。同时，在roles目录所在目录中，还要有一个playbook文件，此处为nginx.yml，nginx.yml文件是ansible-playbook需要执行的文件，在此文件中定义了角色，当执行到角色时，将会到roles中对应的角色目录中寻找相关文件。

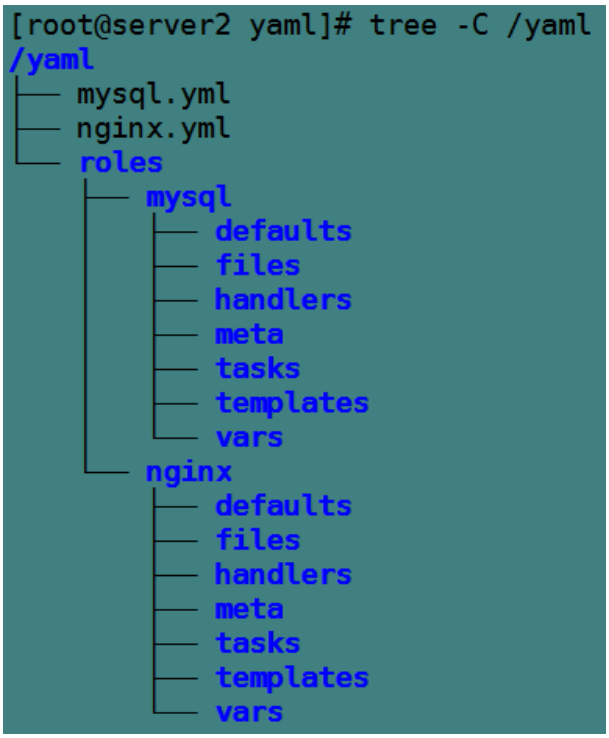
roles目录中的子目录是即是各个role。例如，此处只有一个名为nginx的role，在role目录中，有几个固定名称的目录(如果没有则忽略)。在这些目录中，还要有一些固定名称的文件，除了固定名称的文件，其他的文件可以随意命名。以下是各个目录的含义：

- tasks目录：存放task列表。若role要生效，此目录必须要有一个主task文件main.yml，在main.yml中可以使用include包含同目录(即tasks)中的其他文件。
- handlers目录：存放handlers的目录，若要生效，则文件必须名为main.yml文件。
- files目录：在task中执行copy或script模块时，如果使用的是相对路径，则会到此目录中寻找对应的文件。
- templates目录：在task中执行template模块时，如果使用的是相对路径，则会到此目录中寻找对应的模块文件。
- vars目录：定义专属于该role的变量，如果要有var文件，则必须为main.yml文件。
- defaults目录：定义角色默认变量，角色默认变量的优先级最低，会被任意其他层次的同名变量覆盖。如果要有var文件，则必须为main.yml文件。
- meta目录：用于定义角色依赖(dependencies)，如果要有角色依赖关系，则文件必须为main.yml。

所以，相对完整的role的文件组织结构如下图。



如果是多个role，则在roles同级目录下定义多个入站(作用类似于C语言的main函数)文件(如上面的nginx.yml)，并在roles目录下创建对应的role目录即可。



当然，如果不是使用相对路径，那么role的文件结构就无所谓了，但是roles功能开发出来，就是为了解决文件混乱和playbook臃肿问题的。所以如果可以，尽量使用推荐的role文件结构。

另外，如果role中出现的task、var、handler等和单独定义的对象同名冲突了，则优先执行role中的内容。

以下是nginx role的入站文件nginx.yml的内容。

```
---
  - hosts: centos7
    roles:
      - nginx
```

更多更详细的role用法以及组织结构，见下面的示例。

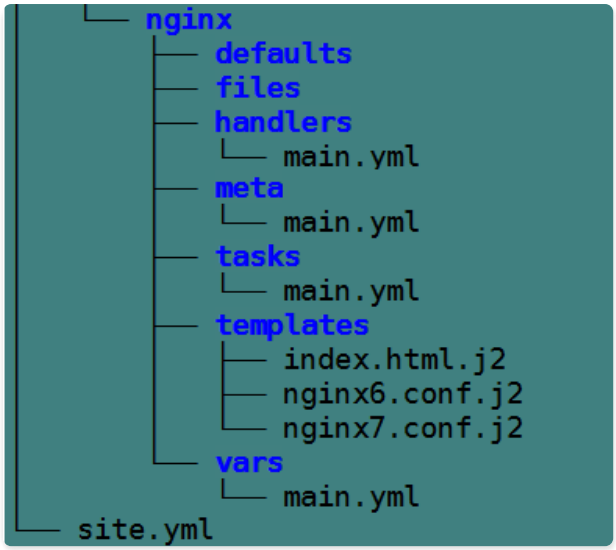
4.5 roles示例：批量自动化安装

下面演示的是使用role批量自动安装nginx和mysql(CentOS 6)或maridb(CentOS 7)的示例。由于被控节点有CentOS 6和CentOS 7两种发行版的操作系统，因此除了要挑选对应的数据库，还要让nginx的配置文件适应各操作系统，因为nginx在这两个版本的系统上配置内容有所不同。在此，nginx、mysql和mariadb是3个role，且让nginx role依赖于mysql或mariadb role。

下面的例子中，有些地方是不太合理或多余的行为，但是作为学习示例，可以很好的理解roles之间的组织结构和相关的操作。

首先是文件的结构。

```
[root@server2 yaml]# tree -C /yaml/
/yaml/
├── roles
│   ├── mariadb
│   │   ├── defaults
│   │   ├── files
│   │   │   └── my.cnf
│   │   ├── handlers
│   │   │   └── main.yml
│   │   ├── meta
│   │   ├── tasks
│   │   │   └── main.yml
│   │   ├── templates
│   │   └── vars
│   │       └── main.yml
│   ├── mysql
│   │   ├── defaults
│   │   ├── files
│   │   │   └── my.cnf
│   │   ├── handlers
│   │   │   └── main.yml
│   │   ├── meta
│   │   ├── tasks
│   │   │   └── main.yml
│   │   ├── templates
│   │   └── vars
│   │       └── main.yml
```



其中site.yml是入站文件，用于调用nginx、mysql和mariadb这3个role，这个文件中的内容如下。它有3个作用：(1)在调用roles之前，先根据发行版配置好yum源(pre_tasks);(2)调用nginx role，此处没有调用mysql和mariadb这两个role，因为在nginx role的meta/main.yml文件中定义了nginx role依赖于这两个role，所以此处可以不用定义;(3)在执行完nginx role之后，输出一个提示信息(post_tasks)。

```
cat /yaml/site.yml
---
  - hosts: centos
    remote_user: root

 # 根据发行版配置好yum源，使用when进行条件判断
    pre_tasks:
        - name: config the yum repo for centos 7
          yum_repository:
              name: epel
              description: epel
              baseurl: http://mirrors.aliyun.com/epel/7/$basearch/
              gpgcheck: no
          when: ansible_distribution_major_version == "7"

        - name: config the yum repo for centos 6
          yum_repository:
              name: epel
              description: epel
              baseurl: http://mirrors.aliyun.com/epel/6/$basearch/
              gpgcheck: no
          when: ansible_distribution_major_version == "6"

    roles:
        - nginx

# 输出over消息
    post_tasks:
      - shell: echo 'deploy nginx/mysql over'
        register: ok_var
      - debug: msg='{{ ok_var.stdout }}'
```

以下是nginx role中的各文件内容。其中template复制的源文件都是从centos 6 nginx和centos 7 nginx上提取的，只不过是重新命名了而已。

```
/yaml/roles/nginx/tasks/main.yml
---
   - name: make sure nginx state is installed
     yum: name=nginx state=installed

   - name: template nginx.conf
# 基于变量赋值配置文件模板，检查配置文件语法，并在必要的时候触发handler
     template: src=nginx{{ ansible_distribution_major_version }}.conf.j2
                 dest=/etc/nginx/nginx.conf
                 validate="/usr/sbin/nginx -t -c %s"
     notify:
        - restart nginx

# 基于jinja2渲染模板文件，且改变时也触发重启操作
   - name: copy index.html
     template: src=index.html.j2 dest=/usr/share/nginx/html/index.html
     notify:
        - restart nginx

   - name: make sure nginx service is running
     service: name=nginx state=started

# 引用变量 nginx_port，在vars/main.yml中定义了
   - name: make sure port is open
     wait_for: port="{{ nginx_port }}"

/yaml/roles/nginx/handlers/main.yml
---
   - name: restart nginx
     service: name=nginx state=restarted

/yaml/roles/nginx/vars/main.yml
nginx_port: 80

# 定义nginx依赖于MySQL或mariadb，具体依赖于哪个，是通过条件进行判断的，centos 6表示依赖于mysql，centos 7表示依赖于mariadb
# 同时传递了两个值给变量hi_var，由于是在依赖的时候传递的，所以这两个变量可直接在依赖的role(mysql role或mariadb role)的play
book中引用
/yaml/roles/nginx/meta/main.yml
---
   dependencies:
     - { role: mysql,hi_var: "hello mysql",when: "ansible_distribution_major_version == '6'" }
     - { role: mariadb,hi_var: "hello mariadb",when: "ansible_distribution_major_version == '7'" }
```

上面拷贝了index.html.j2，其内容为：

```
shell> cat /yaml/roles/nginx/templates/index.html.j2
<h1>hello from {{ ansible_default_ipv4.address }}<h1>
```

在template执行时，它会使用jinja2引擎对文件中的变量进行替换，使得在拷贝到不同主机时，该index.html的内容是基于远程主机ip的。此处使用的变量是收集到的facts中的变量"ansible_default_ipv4.address"。

以下是mysql role中各文件的内容。注意，MySQL的my.cnf和mariadb的my.cnf默认情况下并不一样(mariadb的my.cnf默认多了一项配置"!includedir /etc/my.cnf.d"，MySQL需要取消该项)，所以需要分别提供。

```
/yaml/roles/mysql/tasks/main.yml
---
    - name: make sure mysql is installed
      yum: name=mysql-server state=installed

# 特别需要注意下面的初始化命令，由于执行的是shell模块，所以要考虑其幂等性，显然初始化动作是一定要实现幂等性的
    - name: do something to initialize mysql
      file: path=/mydata/data state=directory owner=mysql group=mysql mode=0755
    - shell: /usr/bin/mysql_install_db --datadir=/mydata/data --user=mysql creates=/mydata/data/ibdata1

    - name: copy my.cnf
      copy: src=my.cnf dest=/etc/my.cnf
      notify:
         - restart mysql

    - name: make sure mysql is running
      service: name=mysqld state=started

    - name: make sure mysql port is open
      wait_for:
         port: "{{ mysql_port }}"
         timeout: 10

# 这里输出了nginx/meta/main.yml中传递的变量
    - name: echo var passed by nginx
      shell: echo "{{ hi_var }}"
      register: var_result
    - debug: msg="{{ var_result.stdout }}"

/yaml/roles/mysql/handlers/main.yml
---
   - name: restart mysql
     service: name=mysqld state=restarted

/yaml/roles/mysql/vars/main.yml
---
    mysql_port: 3306
```

以下是mariadb role中各文件的内容，和mysql大体上是一致的。注意，MySQL的my.cnf和mariadb的my.cnf默认情况下并不一样，所以需要分别提供。

```
/yaml/roles/mariadb/tasks/main.yml
---
    - name: make sure mariadb is installed
      yum: name=mariadb-server state=installed

    - name: do something to initialize mariadb
      file: path=/mydata/data state=directory owner=mysql group=mysql mode=0755
    - shell: /usr/bin/mysql_install_db --datadir=/mydata/data --user=mysql creates=/mydata/data/ibdata1

    - name: copy my.cnf
      copy: src=my.cnf dest=/etc/my.cnf
      notify:
         - restart mariadb

    - name: make sure mariadb is running
      service: name=mariadb state=started

    - name: make sure mariadb port is open
      wait_for:
         port: "{{ mariadb_port }}"
         timeout: 10

    - name: echo var passed by nginx
      shell: echo "{{ hi_var }}"
      register: var_result
    - debug: msg="{{ var_result.stdout }}"

/yaml/roles/mariadb/handlers/main.yml
---
    - name: restart mariadb
      service: name=mariadb state=restarted

/yaml/roles/mariadb/vars/main.yml
---
    mariadb_port: 3306
```

以下是两台机器的测试结果，一台是centos 7(192.168.100.54)，一台是centos 6(192.168.100.70)。

```
PLAY [newhosts] ***************************************************************

TASK [Gathering Facts] ********************************************************
ok: [192.168.100.70]
ok: [192.168.100.54]

TASK [config the yum repo for centos 7] ***************************************
skipping: [192.168.100.70]
changed: [192.168.100.54]

TASK [config the yum repo for centos 6] ***************************************
skipping: [192.168.100.54]
changed: [192.168.100.70]

TASK [mysql : make sure mysql is installed] ***********************************
skipping: [192.168.100.54]
changed: [192.168.100.70]

TASK [mysql : do something to initialize mysql] *******************************
skipping: [192.168.100.54]
changed: [192.168.100.70]

TASK [mysql : command] ********************************************************
skipping: [192.168.100.54]
changed: [192.168.100.70]

TASK [mysql : copy my.cnf] ****************************************************
skipping: [192.168.100.54]
changed: [192.168.100.70]

TASK [mysql : make sure mysql is running] *************************************
skipping: [192.168.100.54]
changed: [192.168.100.70]

TASK [mysql : make sure mysql port is open] ***********************************
skipping: [192.168.100.54]
ok: [192.168.100.70]
```

```
TASK [mysql : echo var passed by nginx] ************************************************
skipping: [192.168.100.54]
changed: [192.168.100.70]

TASK [mysql : debug] *******************************************************************
skipping: [192.168.100.54]
ok: [192.168.100.70] => {
    "changed": false,
    "msg": "hello mysql"
}

TASK [mariadb : make sure mariadb is installed] ****************************************
skipping: [192.168.100.70]
changed: [192.168.100.54]

TASK [mariadb : do something to initialize mariadb] ************************************
skipping: [192.168.100.70]
changed: [192.168.100.54]

TASK [mariadb : command] ***************************************************************
skipping: [192.168.100.70]
changed: [192.168.100.54]

TASK [mariadb : copy my.cnf] ***********************************************************
skipping: [192.168.100.70]
changed: [192.168.100.54]

TASK [mariadb : make sure mariadb is running] ******************************************
skipping: [192.168.100.70]
changed: [192.168.100.54]

TASK [mariadb : make sure mariadb port is open] ****************************************
skipping: [192.168.100.70]
ok: [192.168.100.54]

TASK [mariadb : echo var passed by nginx] **********************************************
skipping: [192.168.100.70]
changed: [192.168.100.54]

TASK [mariadb : debug] *****************************************************************
skipping: [192.168.100.70]
ok: [192.168.100.54] => {
    "changed": false,
    "msg": "hello mysql"
}

TASK [nginx : make sure nginx state is installed] **************************************
changed: [192.168.100.54]
changed: [192.168.100.70]

TASK [nginx : template nginx.conf] *****************************************************
ok: [192.168.100.70]
changed: [192.168.100.54]

TASK [nginx : copy index.html] *********************************************************
changed: [192.168.100.70]
changed: [192.168.100.54]

TASK [nginx : make sure nginx service is running] **************************************
changed: [192.168.100.54]
changed: [192.168.100.70]

TASK [nginx : make sure port is open] **************************************************
ok: [192.168.100.70]
ok: [192.168.100.54]

RUNNING HANDLER [mysql : restart mysql] ************************************************
changed: [192.168.100.70]

RUNNING HANDLER [mariadb : restart mariadb] ********************************************
changed: [192.168.100.54]

RUNNING HANDLER [nginx : restart nginx] ************************************************
changed: [192.168.100.54]

TASK [command] *************************************************************************
changed: [192.168.100.54]
```

```
changed: [192.168.100.70]

TASK [debug] *******************************************************************************
ok: [192.168.100.54] => {
    "changed": false,
    "msg": "deploy nginx/mysql over"
}
ok: [192.168.100.70] => {
    "changed": false,
    "msg": "deploy nginx/mysql over"
}

PLAY RECAP *********************************************************************************
192.168.100.54             : ok=19   changed=14   unreachable=0    failed=0
192.168.100.70             : ok=18   changed=12   unreachable=0    failed=0
```

相信看过上面的roles组织示例，对roles的用法和playbook就有了较深的认识。其实，ansible有一个网站专门存放了一大堆的playbook，算是playbook仓库吧。可以下载下来稍作修改就能使用，即使不使用，借鉴他们的写法也是很值得的。地址：ansible galaxy

另外，根据不同标准组织role可能会让playbook写起来更容易，例如上面的示例中，按照发行版来划分role比上面按照安装软件类型划分可能会更简单些。当然，如果在inventory中就划分好centos 6和centos 7也是可以的。哪种更方便、复用性更好就需要自行考虑了。

5.各种变量定义方式和变量引用

5.1 ansible facts

facts组件是用来收集被管理节点信息的，使用setup模块可以获取这些信息。

```
ansible-doc -s setup
- name: Gathers facts about remote hosts
```

以下是某次收集的信息示例。由于收集的信息项非常多，所以截取了部分内容项。

```
ansible 192.168.100.64 -m setup
192.168.100.64 | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "192.168.100.64"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::20c:29ff:fe03:a452"
        ],
        "ansible_apparmor": {
            "status": "disabled"
        },
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "07/02/2015",
        "ansible_bios_version": "6.00",
        "ansible_cmdline": {
            "BOOT_IMAGE": "/vmlinuz-3.10.0-327.el7.x86_64",
            "LANG": "en_US.UTF-8",
            "biosdevname": "0",
            "crashkernel": "auto",
            "net.ifnames": "0",
            "quiet": true,
            "ro": true,
            "root": "UUID=b2a70faf-aea4-4d8e-8be8-c7109ac9c8b8"
        },
.....................................
        "ansible_default_ipv6": {},
        "ansible_devices": {
            "sda": {
                "holders": [],
                "host": "SCSI storage controller: LSI Logic / Symbios Logic 53c1030 PCI-X Fusion-MPT Dual Ultra320
SCSI (rev 01)",
                "model": "VMware Virtual S",
                "partitions": {
                    "sda1": {
                        "holders": [],
                        "sectors": "512000",
                        "sectorsize": 512,
                        "size": "250.00 MB",
                        "start": "2048",
                        "uuid": "367d6a77-033b-4037-bbcb-416705ead095"
                    },
                    "sda2": {
                        "holders": [],
                        "sectors": "37332992",
                        "sectorsize": 512,
                        "size": "17.80 GB",
                        "start": "514048",
                        "uuid": "b2a70faf-aea4-4d8e-8be8-c7109ac9c8b8"
                    },
                    "sda3": {
                        "holders": [],
                        "sectors": "4096000",
                        "sectorsize": 512,
                        "size": "1.95 GB",
                        "start": "37847040",
                        "uuid": "d505113c-daa6-4c17-8b03-b3551ced2305"
                    }
                },
```

```
.............................
        "ansible_system_capabilities_enforced": "True",
        "ansible_system_vendor": "VMware, Inc.",
        "ansible_uptime_seconds": 33022,
        "ansible_user_dir": "/root",
        "ansible_user_gecos": "root",
        "ansible_user_gid": 0,
        "ansible_user_id": "root",
        "ansible_user_shell": "/bin/bash",
        "ansible_user_uid": 0,
        "ansible_userspace_architecture": "x86_64",
        "ansible_userspace_bits": "64",
        "ansible_virtualization_role": "guest",
        "ansible_virtualization_type": "VMware",
        "module_setup": true
    },
    "changed": false
}
```

使用filter可以筛选指定的facts信息。例如：

```
ansible 192.168.100.64 -m setup -a "filter=changed"
192.168.100.64 | SUCCESS => {
    "ansible_facts": {},
    "changed": false
}

ansible localhost -m setup -a "filter=*ipv4"
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_default_ipv4": {
            "address": "192.168.100.62",
            "alias": "eth0",
            "broadcast": "192.168.100.255",
            "gateway": "192.168.100.2",
            "interface": "eth0",
            "macaddress": "00:0c:29:d9:0b:71",
            "mtu": 1500,
            "netmask": "255.255.255.0",
            "network": "192.168.100.0",
            "type": "ether"
        }
    },
    "changed": false
}
```

facts收集的信息是json格式的，其内任一项都可以当作变量被直接引用(如在playbook、jinja2模板中)引用。见下文。

5.2 变量引用json数据的方式

在ansible中，任何一个模块都会返回json格式的数据，即使是错误信息都是json格式的。

在ansible中，json格式的数据，其内每一项都可以通过变量来引用它。当然，引用的前提是先将其注册为变量。

例如，下面的playbook是将shell模块中echo命令的结果注册为变量，并使用debug模块输出。

```
---
    - hosts: 192.168.100.65
      tasks:
        - shell: echo hello world
          register: say_hi
        - debug: var=say_hi
```

debug输出的结果如下：

```
TASK [debug] *******************************************
ok: [192.168.100.65] => {
    "say_hi": {
        "changed": true,
        "cmd": "echo hello world",
        "delta": "0:00:00.002086",
        "end": "2017-09-20 21:03:40.484507",
        "rc": 0,
        "start": "2017-09-20 21:03:40.482421",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "hello world",
        "stdout_lines": [
            "hello world"
        ]
    }
}
```

可以看出，结果是一段json格式的数据，最顶端的key为`say_hi`，其内是一大段的字典(即使用大括号包围的)，其中的`stdout_lines`还包含了一个json数组，也就是所谓的yaml列表项(即使用中括号包围的)。

5.2.1 引用json字典数据的方式

如果想要输出json数据的某一字典项，则应该使用"key.dict"或"key['dict']"的方式引用。例如最常见的stdout项"hello world"是想要输出的项，以下两种方式都能引用该字典变量。

```
---
    - hosts: 192.168.100.65
      tasks:
        - shell: echo hello world
          register: say_hi
        - debug: var=say_hi.stdout
        - debug: var=sya_hi['stdout']
```

ansible-playbook的部分输出结果如下：

```
TASK [debug] *******************************************
ok: [192.168.100.65] => {
    "say_hi.stdout": "hello world"
}

TASK [debug] *******************************************
ok: [192.168.100.65] => {
    "say_hi['stdout']": "hello world"
}
```

"key.dict"或"key['dict']"的方式都能引用，但在dict字符串本身就包含"."的时候，应该使用中括号的方式引用。例如：

```
anykey['192.168.100.65']
```

5.2.2 引用json数组数据的方式

如果想要输出json数据中的某一数组项(列表项)，则应该使用"key[N]"的方式引用数组中的第N项，其中N是数组的index，从0开始计算。如果不使用index，则输出的是整个数组列表。

例如想要输出上面的`stdout_lines`中的"hello world"，它是数组`stdout_lines`中第一项所以使用`stdout_lines[0]`来引用，再加上`stdout_lines`上面的`say_hi`，于是引用方式如下：

```
---
    - hosts: 192.168.100.65
      tasks:
        - shell: echo hello world
          register: say_hi
        - debug: var=say_hi.stdout_lines[0]
```

由于`stdout_lines`中仅有一项，所以即使不使用index的方式即`say_hi.stdout_lines`也能得到期望的结果。输出结果如下：

```
TASK [debug] *******************************************
ok: [192.168.100.65] => {
    "say_hi.stdout_lines[0]": "hello world"
}
```

再看下面一段json数据。

```
"ipv6": [
   {
       "address": "fe80::20c:29ff:fe26:1498",
       "prefix": "64",
       "scope": "link"
   }
]
```

其中key=ipv6，其内有且仅有是一个列表项，但该列表内包含了数个字典项。要引用列表内的字典，例如上面的address项。应该如下引用：

```
ipv6[0].address
```

5.2.3 引用facts数据

既然已经了解了json数据中的字典和列表列表项的引用方式，显然facts中的一大堆数据就能引用并派上用场了。例如以下是一段facts数据。

```
shell> ansible localhost -m setup -a "filter=*eth*"
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_eth0": {
            "active": true,
            "device": "eth0",
            "features": {
                "busy_poll": "off [fixed]",
                "fcoe_mtu": "off [fixed]",
                "generic_receive_offload": "on",
                ........................
            },
            "ipv4": {
                "address": "192.168.100.62",
                "broadcast": "192.168.100.255",
                "netmask": "255.255.255.0",
                "network": "192.168.100.0"
            },
            "macaddress": "00:0c:29:d9:0b:71",
            "module": "e1000",
             ........................
        }
    },
    "changed": false
}
```

显然，facts数据的顶级key为`ansible_facts`，在引用时应该将其包含在变量表达式中。但自动收集的facts比较特殊，它以`ansible_facts`作为key，ansible每次收集后会自动将其注册为变量，所以facts中的数据都可以直接通过变量引用，甚至连顶级key `ansible_facts`都要省略。

例如引用上面的ipv4的地址address项。

```
ansible_eth0.ipv4.address
```

而不能写成：

```
ansible_facts.ansible_eth0.ipv4.address
```

但其他任意时候，都应该带上所有的key。

5.3 设置本地facts

在ansible收集facts时，还会自动收集/etc/ansible/facts.d/*.fact文件内的数据到facts中，且以`ansible_local`做为key。目前fact支持两种类型的文件：ini和json。当然，如果fact文件的json或ini格式写错了导致无法解析，那么肯定也无法收集。

例如，在/etc/ansible/facts.d目录下存在一个my.fact的文件，其内数据如下：

```
shell> cat /etc/ansible/facts.d/my.fact
{
    "family": {
        "father": {
            "name": "Zhangsan",
            "age": "39"
        },
        "mother": {
            "name": "Lisi",
            "age": "35"
        }
    }
}
```

ansible收集facts后的本地facts数据如下：

```
shell> ansible localhost -m setup -a "filter=ansible_local"
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_local": {
            "my": {
                "family": {
                    "father": {
                        "age": "39",
                        "name": "Zhangsan"
                    },
                    "mother": {
                        "age": "35",
                        "name": "Lisi"
                    }
                }
            }
        }
    },
    "changed": false
}
```

可见，如果想要引用本地文件中的某个key，除了带上ansible_local外，还必须得带上fact文件的文件名。例如，引用father的name。

```
ansible_local.my.family.father.name
```

5.4 输出和引用变量

上文已经展示了一种变量的引用方式：使用debug的var参数。debug的另一个参数msg也能输出变量，且msg可以输出自定义信息，而var参数只能输出变量。

另外，msg和var引用参数的方式有所不同。例如：

```
---
    - hosts: 192.168.100.65
      tasks:
        - debug: 'msg="ipv4 address: {{ansible_eth0.ipv4.address}}"'
        - debug: var=ansible_eth0.ipv4.address
```

msg引用变量需要加上双大括号包围，既然加了大括号，为了防止被解析为内联字典，还得加引号包围。这里使用了两段引号，因为其内还包括了一个":"，加引号可以防止它被解析为"key: "的格式。而var参数引用变量则直接指定变量名。

这就像bash中引用变量的方式是一样的，有些时候需要加上`$`，有些时候不能加`$`。也就是说，当引用的是变量的值，就需要加双大括号，就像加`$`一样，而引用变量本身，则不能加双大括号。其实双大括号是jinja2中的分隔符。

执行的部分结果如下：

```
TASK [debug] *******************************************************
ok: [192.168.100.65] => {
    "msg": "ipv4 address: 192.168.100.65"
}

TASK [debug] *******************************************************
ok: [192.168.100.65] => {
    "ansible_eth0.ipv4.address": "192.168.100.65"
}
```

几乎所有地方都可以引用变量，例如循环、when语句、信息输出语句、template文件等等。只不过有些地方不能使用双大括号，有些地方需要使用。

5.5 注册和定义变量的各种方式

ansible中定义变量的方式有很多种，大致有：(1)将模块的执行结果注册为变量；(2)直接定义字典类型的变量；(3)role中文件内定义变量；(4)命令行传递变量；(5)借助with_items迭代将多个task的结果赋值给一个变量；(6)inventory中的主机或主机组变量；(7)内置变量。

5.5.1 register注册变量

使用register选项，可以将当前task的输出结果赋值给一个变量。例如，下面的示例中将echo的结果"haha"赋值给say_hi变量。注意，模块的输出结果是json格式的，所以，引用变量时要指定引用的对象。

```
---
    - hosts: localhost
      tasks:
        - shell: echo haha
          register: say_hi
        - debug: var=say_hi.stdout
```

5.5.2 set_fact定义变量

set_fact和register的功能很相似，也是将值赋值给变量。它更像shell中变量的赋值方式，可以将某个变量的值赋值给另一个变量，也可以将字符串赋值给变量。

例如：

```
---
    - hosts: 192.168.100.65
      tasks:
        - shell: echo haha
          register: say_hi
        - set_fact: var1="{{say_hi.stdout}}"
        - set_fact: var2="your name is"
        - debug: msg="{{var2}} {{var1}}"
```

5.5.3 vars定义变量

可以在play或task层次使用vars定义字典型变量。如果同名，则task层次的变量覆盖play层次的变量。

例如：

```
---
    - hosts: localhost
      vars:
        var1: value1
        var2: value2
      tasks:
         - debug: msg="{{var1}} {{var2}}"
           vars:
             var2: value2.2
```

输出结果为：

```
TASK [debug] *******************************************
ok: [localhost] => {
    "msg": "value1 value2.2"
}
```

5.5.4 vars_files定义变量

和vars一样，只不过它是将变量以字典格式定义在独立的文件中，且`vars_files`不能定义在task层次，只能定义在play层次。

```
---
    - hosts: localhost
      vars_files:
        - /tmp/var_file1.yml
        - var_file2.yml
      tasks:
         - debug: msg="{{var1}} {{var2}}"
```

上面var_file2.yml使用的是相对路径，基于playbook所在的路径。例如该playbook为/tmp/x.yml，则var_file2.yml也应该在/tmp下。当然，完全可以使用绝对路径。

5.5.5 roles中的变量

由于role是整合playbook的，它有默认的文件组织结构。其中有一个目录vars，其内的main.yml用于定义变量。还有defaults目录内的main.yml则是定义role默认变量的，默认变量的优先级最低。

```
shell> tree /yaml
/yaml
├── roles
│   └── nginx
│       ├── defaults
│       │   └── main.yml
│       ├── files
│       ├── handlers
│       ├── meta
│       ├── tasks
│       ├── templates
│       └── vars
│           └── main.yml
└── site.yml
```

main.yml中变量定义方式也是字典格式，例如：

```
---
    mysql_port: 3306
```

5.5.6 命令行传递变量

ansible和ansible-playbook命令的"-e"选项都可以传递变量，传递的方式有两种：`-e key=value`和`-e @var_file`。注意，当key=value方式传递变量时，如果变量中包含特殊字符，必须防止其被shell解析。

例如：

```
ansible localhost -m shell -a "echo {{say_hi}}" -e 'say_hi="hello world"'
ansible localhost -m shell -a "echo {{say_hi}}" -e @/tmp/var_file1.yml
```

其中/tmp/var_file1.yml中的内容如下：

```
---
    say_hi: hello world
```

5.5.7 借助with_items叠加变量

ansible中可以借助with_items实现列表迭代的功能，作用于变量注册的行为上，就可以实现将多个结果赋值给同一个变量。

例如下面的playbook中，给出了3个item列表，并在shell模块中通过固定变量"{{item}}"分别迭代，第一次迭代的是haha，第二次迭代的是heihei，第三次迭代的是hehe，也就实现了3次循环。最后，将结果注册为变量hi_var。

```
---
    - hosts: localhost
      remote_user: root
      tasks:
        - name: test #
          shell: echo "{{item}}"
          with_items:
            - haha
            - heihei
            - hehe
          register: hi_var
        - debug: var=hi_var.results[0].stdout
        - debug: var=hi_var.results[1].stdout
        - debug: var=hi_var.results[2].stdout
```

每次迭代的过程中，调用item的模块都会将结果保存在一个key为results的数组中。因此，引用迭代后注册的变量时，需要在变量名中加上results，并指定数组名。例如上面的`hi_var.results[N].stdout`。

还可以使用for循环遍历列表。例如：

```
- debug: msg="{% for i in hi_var.results %} {{i.stdout}} {% endfor %}"
```

其实，看一下hi_var的输出就很容易理解了。以下是hi_var的第一个列表的输出。

```
"hi_var": {
    "changed": true,
    "msg": "All items completed",
    "results": [
        {
            "_ansible_item_result": true,
            "_ansible_no_log": false,
            "_ansible_parsed": true,
            "changed": true,
            "cmd": "echo \"haha\"",
            "delta": "0:00:00.001942",
            "end": "2017-09-21 04:45:57.032946",
            "invocation": {
                "module_args": {
                    "_raw_params": "echo \"haha\"",
                    "_uses_shell": true,
                    "chdir": null,
                    "creates": null,
                    "executable": null,
                    "removes": null,
                    "warn": true
                }
            },
            "item": "haha",
            "rc": 0,
            "start": "2017-09-21 04:45:57.031004",
            "stderr": "",
            "stderr_lines": [],
            "stdout": "haha",
            "stdout_lines": [
                "haha"
            ]
        }
```

5.5.8 inventory中主机变量和主机组变量

在inventory文件中可以为主机和主机组定义变量，不仅包括内置变量赋值，还包括自定义变量赋值。例如以下inventory文件。

```
192.168.100.65 ansible_ssh_port=22 var1=1
[centos7]
192.168.100.63
192.168.100.64
192.168.100.65 var1=2
[centos7:vars]
var1=2.2
var2=3
[all:vars]
var2=4
```

其中`anisble_ssh_port`是主机内置变量，为其赋值22，这类变量是设置类变量，不能被引用。此外还在多处为主机192.168.100.65进行了赋值。其中`[centos7:vars]`和`[all:vars]`表示为主机组赋值，前者是伪centos7这个组赋值，后者是为所有组赋值。

以下是执行语句：

```
shell> ansible 192.168.100.65 -i /tmp/hosts -m shell -a 'echo "{{var1}} {{var2}}"'
192.168.100.65 | SUCCESS | rc=0 >>
2 3
```

从结果可知，主机变量优先级高于主机组变量，给定的主机组变量优先级高于all特殊组。

除了在inventory文件中定义主机、主机组变量，还可以将其定义在host_vars和group_vars目录下的独立的文件中，但要求这些host_vars或group_vars这两个目录和inventory文件或playbook文件在同一个目录下，且变量的文件以对应的主机名或主机组名命名。

例如，inventory文件路径为/etc/ansible/hosts，playbook文件路径为/tmp/x.yml，则主机192.168.100.65和主机组centos7的变量文件路径可以为以下几种：

- /etc/ansible/host_vars/192.168.100.65
- /etc/ansible/group_vars/centos7
- /tmp/host_vars/192.168.100.65
- /tmp/group_vars/centos7

以下为几个host_vars和group_vars目录下的文件内容。

```
shell> cat /etc/ansible/{host_vars/192.168.100.65,group_vars/centos7} \
           /tmp/{host_vars/192.168.100.65,group_vars/centos7}
var1: 1
var2: 2
var3: 3
var4: 4
```

以下为/tmp/x.yml的内容。

```
---
    - hosts: 192.168.100.65
      tasks:
        - debug: msg='{{var1}} {{var2}} {{var3}} {{var4}}'
```

执行结果如下：

```
TASK [debug] *******************************************
ok: [192.168.100.65] => {
    "msg": "1 2 3 4"
}
```

5.5.9 内置变量

ansible除了inventory中内置的一堆不可被引用的设置类变量，还有几个全局都可以引用的内置变量，主要有以下几个：
inventory_hostname、inventory_hostname_short、groups、group_names、hostvars、play_hosts、inventory_dir和ansible_version。

1.inventory_hostname和inventory_hostname_short

分表代表的是inventory中被控节点的主机名和主机名的第一部分，如果定义的是主机别名，则变量的值也是别名。

例如inventory中centos7主机组定义为如下：

```
[centos7]
192.168.100.63
host1 ansible_ssh_host=192.168.100.64
www.host2.com ansible_ssh_host=192.168.100.65
```

分别输出它们的`inventory_hostname`和`inventory_hostname_short`。

```
shell> ansible centos7 -m debug -a 'msg="{{inventory_hostname}} & {{inventory_hostname_short}}"'
192.168.100.63 | SUCCESS => {
    "msg": "192.168.100.63 & 192"
}
host1 | SUCCESS => {
    "msg": "host1 & host1"
}
www.host2.com | SUCCESS => {
    "msg": "www.host2.com & www"
}
```

2.groups和group_names

group_names返回的是主机所属主机组，如果该主机在多个组中，则返回多个组，如果它不在组中，则返回ungrouped这个特殊组。

例如，某个inventory文件如下：

```
192.168.100.60
192.168.100.63
192.168.100.64
192.168.100.65
[centos6]
192.168.100.60
[centos7]
192.168.100.63
host1 ansible_ssh_host=192.168.100.64
www.host2.com ansible_ssh_host=192.168.100.65
[centos:children]
centos6
centos7
```

其中100.60定义在centos6和centos中，所以返回这两个组。同理100.63返回centos7和centos。100.64和100.65则返回ungrouped，虽然它们在centos7中都定义了别名，但至少将100.64和100.65作为主机名时，它们是不在任何主机中的。另一方面，host1和www.host2.com这两个别名主机都返回centos7和centos两个组。

groups变量则是返回其所在inventory文件中所有组和其内主机名。注意，该变量对每个控制节点都返回一次，所以返回的内容可能非常多。例如，上面的inventory中，如果指定被控节点为centos7，则会重复返回3次(因为有3台被控主机)该inventory文件。其中的第三台主机www.host2.com的返回结果为：

```
www.host2.com | SUCCESS => {
    "msg": {
        "all": [
            "192.168.100.60",
            "192.168.100.63",
            "192.168.100.64",
            "192.168.100.65",
            "host1",
            "www.host2.com"
        ],
        "centos": [
            "192.168.100.60",
            "192.168.100.63",
            "host1",
            "www.host2.com"
        ],
        "centos6": [
            "192.168.100.60"
        ],
        "centos7": [
            "192.168.100.63",
            "host1",
            "www.host2.com"
        ],
        "ungrouped": [
            "192.168.100.60",
            "192.168.100.63",
            "192.168.100.64",
            "192.168.100.65"
        ]
    }
}
```

3.hostvars

该变量用于引用其他主机上收集的facts中的数据，或者引用其他主机的主机变量、主机组变量。其key为主机名或主机组名。

举个例子，假如使用ansible部署一台php服务器host1，且配置文件内需要指向另一台数据库服务器host2的ip地址ip2，可以直接在配置文件中指定ip2，但也可以在模板配置文件中直接引用host2收集的facts数据中的ansible_eth0.ipv4.address变量。

例如，centos7主机组中包含了192.168.100.[63:65]共3台主机。playbook内容如下：

```
---
    - hosts: centos7
      tasks:
        - debug: msg="{{hostvars['192.168.100.63'].ansible_eth0.ipv4.address}}"
```

执行结果如下：

```
TASK [debug] *******************************************************
ok: [192.168.100.63] => {
    "msg": "192.168.100.63"
}
ok: [192.168.100.64] => {
    "msg": "192.168.100.63"
}
ok: [192.168.100.65] => {
    "msg": "192.168.100.63"
}
```

但注意，**在引用其他主机facts中数据时，要求被引用主机进行了facts收集动作，或者有facts缓存**。否则都没收集，当然无法引用其facts数据。也就是说，当被引用主机没有facts缓存时，ansible的控制节点中必须同时包含引用主机和被引用主机。

除了引用其他主机的facts数据，还可以引用其他主机的主机变量和主机组变量，且不要求被引用主机有facts数据，因为主机变量和主机组变量是在ansible执行任务前加载的。

例如，inventory中格式如下：

```
192.168.100.59
[centos7]
192.168.100.63 var63=63
192.168.100.64
192.168.100.65
[centos7:vars]
var64=64
```

playbook内容如下：

```
---
    - hosts: 192.168.100.59
      tasks:
        - debug: msg="{{hostvars['192.168.100.63'].var63}} & {{hostvars['192.168.100.65'].var64}}"
```

执行结果如下：

```
TASK [debug] *************************************
ok: [192.168.100.59] => {
    "msg": "63 & 64"
}
```

4.play_hosts和inventory_dir

play_hosts代表的是当前play所涉及inventory内的所有主机名列表。

例如，inventory内容为：

```
192.168.100.59
[centos6]
192.168.100.62
192.168.100.63
[centos7]
192.168.100.64
192.168.100.65
```

那么，该inventory内的任意一或多台主机作为ansible或ansible-playbook的被控节点时，都会返回整个inventory内的所有主机名称。

inventory_dir是所使用inventory所在的目录。

5.ansible_version

代表的是ansible软件的版本号。变量返回的内容如下：

```
{
        "full": "2.3.1.0",
        "major": 2,
        "minor": 3,
        "revision": 1,
        "string": "2.3.1.0"
    }
```

最后，不得不说ansible的变量定义方式太丰富了，但是ansible的官方手册真的恶心到吐，太烂了。

6.条件判断和循环

6.1 循环

ansible中的循环都是借助迭代来实现的。基本都是以"with_"开头。以下是常见的几种循环。

6.1.1 with_items迭代列表

ansibel支持迭代功能。例如，有一大堆要输出的命令、一大堆要安装的软件包、一大堆要copy的文件等等。

例如，要安装一堆软件包。

```
---
    - hosts: localhost
      tasks:
        - yum: name="{{item}}" state=installed
          with_items:
            - pkg1
            - pkg2
            - pkg3
```

它会一个一个迭代到特殊变量"{{item}}"处。

再例如，指定一堆文件列表，然后使用grep搜索出给定文件列表中包含"www.example.com"字符串的文件：

```
---
    - hosts: localhost
      tasks:
        - shell: grep -Rl "www\.example\.com" "{{item}}"
          with_items:
            - file1
            - file2
            - file3
          register: match_file
        - debug: msg="{% for i in match_file.results %} {{i.stdout}} {% endfor %}"
```

注意，将with_items迭代后的结果注册为变量时，其注册结果也是列表式的，且其key为"results"。具体的结果比较长，可以使用debug模块的var或msg参数观察match_file变量的结果。

在上面，是使用for循环进行引用的。如果不使用for循环，那么就需要使用数组格式。例如，引用match_file中的第一和第二个结果。

```
 - debug: var=match_file.results[0].stdout
 - debug: var=match_file.results[1].stdout
```

显然，不如循环引用更好，因为不知道match_file中到底有几个匹配文件，也就不能确定match_file中的列表数量。

每个列表项中可能都包含一个或多个字典，既然with_items迭代的是列表项，那么肯定也能迭代列表中的各字典。

例如：

```
tasks:
    - command: echo {{ item }}
      with_items: [ 0, 2, 4, 6, 8, 10 ]
      register: num
    - debug: msg="{% for i in num.results %} {{i.stdout}} {% endfor %}"
```

再例如：

```
---
    - hosts: localhost
      tasks:
        - shell: echo "name={{item.name}},age={{item.age}}"
          with_items:
            - {name: zhangsan,age: 32}
            - {name: lisi,age: 33}
            - {name: wangwu,age: 35}
          register: who
        - debug: msg="{% for i in who.results %} {{i.stdout}} {% endfor %}"
```

6.1.2 with_dict迭代字典项

使用"with_dict"可以迭代字典项。迭代时，使用"item.key"表示字典的key，"item.value"表示字典的值。

例如：

```
---
    - hosts: localhost
      tasks:
        - debug: msg="{{item.key}} & {{item.value}}"
          with_dict: { address: 1,netmask: 2,gateway: 3 }
```

另一种情况，字典是已存储好的。例如ansible facts中的ansible_eth0.ipv4，其内容如下：

```
"ipv4": {
    "address": "192.168.100.65",
    "netmask": "255.255.255.0",
    "gateway": "192.168.100.2"
}
```

这种情况下，with_dict处可以直接指定该字典的key。即：

```
---
    - hosts: localhost
      tasks:
        - debug: msg="{{item.key}} & {{item.value}}"
          with_dict: ansible_eth0.ipv4
```

再例如，直接引用playbook中定义的vars。

```
---
 - hosts: 192.168.100.65
   gather_facts: False
   vars:
     user:
        longshuai_key:
           name: longshuai
           gender: Male
        xiaofang_key:
           name: xiaofang
           gender: Female
   tasks:
      - name: print hash loop var
        debug: msg="{{ item.key }} & {{ item.value.name }} & {{ item.value.gender }}"
        with_dict: "{{ user }}"
```

6.1.3 with_fileglob迭代文件

例如，拷贝一堆用通配符匹配出来的文件到各远程主机上。

```
---
    - hosts: centos
      tasks:
        - copy: src="{{item}}" dest=/tmp/
          with_fileglob:
            - /tmp/*.sh
            - /tmp/*.py
```

注意，通配符无法匹配"/"，因此无法递归到子目录中，也就无法迭代子目录中的文件。

6.1.4 with_lines迭代行

with_lines很好用，可以将命令行的输出结果按行迭代。

例如，find一堆文件出来，copy走。

```
---
    - hosts: localhost
      tasks:
        - copy: src="{{item}}" dest=/tmp/yaml
          with_lines:
            - find /tmp -type f -name "*.yml"
```

6.1.5 with_nested嵌套迭代

嵌套迭代是指多次迭代列表项。例如：

```
---
    - hosts: localhost
      tasks:
        - debug: msg="{{item[0]}} & {{item[1]}}"
          with_nested:
            - [a,b]
            - [1,2,3]
```

结果将得到"a & 1"、"a & 2"、"a & 3"、"b & 1"、"b & 2"和"b & 3"共6个结果。

6.2 条件判断

在ansible中，只有when可以实现条件判断。

```
tasks:
  - name: config the yum repo for centos 6
    yum_repository:
       name: epel
       description: epel
       baseurl: http://mirrors.aliyun.com/epel/6/$basearch/
       gpgcheck: no
    when: ansible_distribution_major_version == "6"
```

注意两点：

- when判断的对象是task，所以和task在同一列表层次。它的判断结果决定它所在task是否执行，而不是它下面的task是否执行。
- when中引用变量的时候不需要加{{ }}符号。

此外，还支持各种逻辑组合。

```
tasks:

# 逻辑或
  - command: /sbin/shutdown -h now
    when: (ansible_distribution == "CentOS" and ansible_distribution_major_version == "6") or
          (ansible_distribution == "Debian" and ansible_distribution_major_version == "7")

# 逻辑与
  - command: /sbin/shutdown -t now
    when:
      - ansible_distribution == "CentOS"
      - ansible_distribution_major_version == "6"

# 取反
  - command: /sbin/shutdown -t now
    when: not ansible_distribution == "CentOS"
```

还可以直接直接引用布尔值的变量。

```
---
    - hosts: localhost
      vars:
        epic: False

      tasks:
        - debug: msg="This certainly is epic!"
          when: not epic
```

此外，可以使用jinja2的defined来测试变量是否已定义，使用undefined可以取反表示未定义。例如：

```
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 undefined
```

7.执行过程分析、异步模式和速度优化

7.1 ansible执行过程分析

使用ansible的`-vvv`或`-vvvv`分析执行过程。以下是一个启动远程192.168.100.61上httpd任务的执行过程分析。其中将不必要的信息都是用"....."替换了。

```
# 读取配置文件，然后开始执行对应的处理程序。
Using /etc/ansible/ansible.cfg as config file
META: ran handlers

# 第一个任务默认都是收集远程主机信息的任务。
# 第一个收集任务，加载setup模块
Using module file /usr/lib/python2.7/site-packages/ansible/modules/system/setup.py

# 建立连接，获取被控节点当前用户的家目录，用于存放稍后的临时任务文件，此处返回值为/root。
# 在-vvv的结果中，第一行属于描述性信息，第二行为代码执行段，第三行类似此处的<host_node>(, , , , , )为上一段代码的返回结果。后同
<192.168.100.61> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.61> SSH: EXEC ssh -C ..........................
<192.168.100.61> (0, '/root\n', '')

# 再次建立连接，在远端创建临时任务文件的目录，临时目录由配置文件中的remote_tmp指令控制
<192.168.100.61> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.61> SSH: EXEC ssh -C .......................... '/bin/sh -c '"'"'( umask 77 && mkdir -p "` echo /root/.ansible/tmp/ansible-tmp-1495977564.58-40718671162202 `" && echo ansible-tmp-1495977564.58-40718671162202="` echo /root/.ansible/tmp/ansible-tmp-1495977564.58-40718671162202 `" ) && sleep 0'"'"''
<192.168.100.61> (0, 'ansible-tmp-1495977564.58-40718671162202=/root/.ansible/tmp/ansible-tmp-1495977564.58-40718671162202\n', '')

# 将要执行的任务放到临时文件中，并使用sftp将任务文件传输到被控节点上
<192.168.100.61> PUT /tmp/tmpY5vJGX TO /root/.ansible/tmp/ansible-tmp-1495977564.58-40718671162202/setup.py
<192.168.100.61> SSH: EXEC sftp -b - -C ............. '[192.168.100.61]'
<192.168.100.61> (0, 'sftp> put /tmp/tmpY5vJGX /root/.ansible/tmp/ansible-tmp-1495977564.58-40718671162202/setup.py\n', '')

# 建立连接，设置远程任务文件其所有者有可执行权限
<192.168.100.61> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.61> SSH: EXEC ssh -C .......................... '/bin/sh -c '"'"'chmod u+x /root/.ansible/tmp/ansible-tmp-1495977564.58-40718671162202/ /root/.ansible/tmp/ansible-tmp-1495977564.58-40718671162202/setup.py && sleep 0'"'"''
<192.168.100.61> (0, '', '')

# 建立连接，执行任务，执行完成后立即删除任务文件，并返回收集到的信息给ansible。到此为止，setup收集任务结束，关闭共享连接
<192.168.100.61> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.61> SSH: EXEC ssh -C ............ '/bin/sh -c '"'"'/usr/bin/python /root/.ansible/tmp/ansible-tmp-1495977564.58-40718671162202/setup.py; rm -rf "/root/.ansible/tmp/ansible-tmp-1495977564.58-40718671162202/" > /dev/null 2>&1 && sleep 0'"'"''
<192.168.100.61> (0, '\r\n{"invocation": {"..........', 'Shared connection to 192.168.100.61 closed.\r\n')


# 进入下一个任务，此处为服务管理任务，所以加载service模块
Using module file /usr/lib/python2.7/site-packages/ansible/modules/system/service.py

# 建立连接，获取被控节点当前用户的家目录，用于存放稍后的临时任务文件，此处返回值为/root
<192.168.100.61> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.61> SSH: EXEC ssh -C ..........................
<192.168.100.61> (0, '/root\n', '')

# 建立连接，将要执行的任务放入到临时文件中，并传输到远程目录
<192.168.100.61> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.61> SSH: EXEC ssh -C .............. '/bin/sh -c '"'"'( umask 77 && mkdir -p "` echo /root/.ansible/tmp/ansible-tmp-1495977564.97-137863382080241 `" && echo ansible-tmp-1495977564.97-137863382080241="` echo /root/.ansible/tmp/ansible-tmp-1495977564.97-137863382080241 `" ) && sleep 0'"'"''
<192.168.100.61> (0, 'ansible-tmp-1495977564.97-137863382080241=/root/.ansible/tmp/ansible-tmp-1495977564.97-137863382080241\n', '')

<192.168.100.61> PUT /tmp/tmpn5uZhP TO /root/.ansible/tmp/ansible-tmp-1495977564.97-137863382080241/service.py
<192.168.100.61> SSH: EXEC sftp -b - -C ............. '[192.168.100.61]'
<192.168.100.61> (0, 'sftp> put /tmp/tmpn5uZhP /root/.ansible/tmp/ansible-tmp-1495977564.97-137863382080241/service.py\n', '')
```

```
# 建立连接，设置远程任务文件其所有者有可执行权限
<192.168.100.61> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.61> SSH: EXEC ssh -C ..................... '/bin/sh -c '"'"'chmod u+x /root/.ansible/tmp/ansible-tmp-1495977564.97-137863382080241/ /root/.ansible/tmp/ansible-tmp-1495977564.97-137863382080241/service.py && sleep 0'"'"''
<192.168.100.61> (0, '', '')

# 建立连接，执行任务，执行完成后立即删除任务文件，并将执行的结果返回到ansible端。到此为止，service模块任务执行结束，关闭共享连接
<192.168.100.61> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.61> SSH: EXEC ssh -C ............. '/bin/sh -c '"'"'/usr/bin/python /root/.ansible/tmp/ansible-tmp-1495977564.97-137863382080241/service.py; rm -rf "/root/.ansible/tmp/ansible-tmp-1495977564.97-137863382080241/" > /dev/null 2>&1 && sleep 0'"'"''
<192.168.100.61> (0, '\r\n{"msg": "............', 'Shared connection to 192.168.100.61 closed.\r\n')
```

将上面的进行总结，执行过程将是这样的：

- 读取配置文件
- 加载inventory文件。包括主机变量和主机组变量
- 执行第一个任务：收集远程被控节点的信息
  - 建立连接，获取家目录信息
  - 将要执行的收集任务放到临时文件中
  - 将临时任务文件传输到被控节点的临时目录中
  - ssh连接到远端执行收集任务
  - 删除任务文件
  - 将收集信息返回给ansible端
  - 关闭连接
- 执行第二个任务，此为真正的主任务
  - 建立连接，获取家目录信息
  - 将要执行的任务放到临时文件中
  - 将临时任务文件传输到被控节点的临时目录中
  - ssh连接到远端执行任务
  - 删除任务文件
  - 将执行结果返回给ansible端，ansible输出到屏幕或指定文件中
  - 关闭连接
- 执行第三个任务
- 执行第四个任务

如果是多个被控节点，那么将同时在多个节点上并行执行每一个任务，例如同时执行信息收集任务。不同节点之间的任务没有先后关系，主要依赖于性能。每一个任务执行完毕都会立即将结果返回给ansible端，所以可以**通过ansible端结果的输出顺序和速度判断执行完毕的先后顺序。**

如果节点数太多，ansible无法一次在所有远程节点上执行任务，那么将先在一部分节点上执行一个任务(每一批节点的数量取决于fork进程数量)，直到这一批所有节点上该任务完全执行完毕才会接入下一个批节点(数量取决于fork进程数量)，直到所有节点将该任务都执行完毕，然后重新回到第一批节点开始执行第二个任务。依次类推，直到所有节点执行完所有任务，ansible端才会释放shell。这是默认的同步模式，也就是说在未执行完毕的时候，ansible是占用当前shell的，任务执行完毕后，释放shell了才可以输入其他命令做其他动作。

如果是异步模式，假如fork控制的并发进程数为5，远程控制节点为24个，则ansible一开始会将5个节点的任务扔在后台，并每隔一段时间去检查这些节点的任务完成情况，**当某节点完成不会立即返回，而是继续等待直到5个进程都空闲了**，才会将这5个节点的结果返回给ansible端，ansible会继续将下一批5个节点的任务扔在后台并每隔一段时间进行检查，依次类推，直到完成所有任务。

在异步模式下，如果设置的检查时间间隔为0，在将每一批节点的任务丢到后台后都会立即返回ansible，并立即将下一批节点的任务丢到后台，直到所有任务都丢到后台完成后，会返回ansible端，ansible会立即释放占用的shell。也就是说，此时ansible是不会管各个节点的任务执行情况的，不管执行成功还是失败。

因此，在轮训检查时间内，ansible仍然正在运行(尽管某批任务已经被放到后台执行了)，当前shell进程仍被占用处于睡眠状态，只有指定的检查时间间隔为0，才会尽快将所有任务放到后台并释放shell。

需要注意3点：

1 .按批(例如每次5台全部完成一个任务才进入下一批的5台)完成任务的模式在ansible 2.0版本之后可以通过修改ansible的执行策略来改变(见后文)，改变后会变成"前赴后继"的执行模式：当一个节点执行完一个任务会立即接入另一个节点，不再像默认情况一样等待这一批中的其他节点完成该任务。
2 .上面执行过程是默认的执行过程，如果开启了pipelining加速anisble执行效率，会省去sftp到远端的过程。
3 .信息收集任务是默认会执行的，但是可以设置禁用它。

7.2 ansible并发和异步

上面已经对ansible的执行过程进行了很详细的分析，也解释了同步和异步的模式是如何处理任务的。所以此处简单举几个例子。

ansible默认只会创建5个进程并发执行任务，所以一次任务只能同时控制5台机器执行。如果有大量的机器需要控制，例如20台，ansible执行一个任务时会先在其中5台上执行，执行成功后再执行下一批5台，直到全部机器执行完毕。使用-f选项可以指定进程数，指定的进程数量多一些，不仅会实现全并发，对异步的轮训poll也会有正面影响。

ansible默认是同步阻塞模式，它会等待所有的机器都执行完毕才会在前台返回。可以采取异步执行模式。

异步模式下，ansible会将节点的任务丢在后台，每台被控制的机器都有一个job_id，ansible会根据这个job_id去轮训该机器上任务的执行情况，例如某机器上此任务中的某一个阶段是否完成，是否进入下一个阶段等。 **即使任务早就结束了，但只有轮训检查到任务结束后才认为该job结束**。可以指定任务检查的时间间隔，默认是10秒。除非指定任务检查的间隔为0，否则会等待所有任务都完成后，ansible端才会释放占用的shell。

如果指定时间间隔为0，则ansible会立即返回(至少得连接上目标主机，任务发布成功之后立即返回)，并不会去检查它的任务进度。

```
ansible centos -B200 -P 0 -m yum -a "name=dos2unix" -o -f 6
192.168.100.61 | SUCCESS => {"ansible_job_id": "986026954359.9166", "changed": true, "finished": 0, "results_file":
"/root/.ansible_async/986026954359.9166", "started": 1}
192.168.100.59 | SUCCESS => {"ansible_job_id": "824724696770.9431", "changed": true, "finished": 0, "results_file":
"/root/.ansible_async/824724696770.9431", "started": 1}
192.168.100.60 | SUCCESS => {"ansible_job_id": "276581152579.10006", "changed": true, "finished": 0, "results_file"
: "/root/.ansible_async/276581152579.10006", "started": 1}
192.168.100.64 | SUCCESS => {"ansible_job_id": "237326453903.72268", "changed": true, "finished": 0, "results_file"
: "/root/.ansible_async/237326453903.72268", "started": 1}
192.168.100.63 | SUCCESS => {"ansible_job_id": "276700021098.73070", "changed": true, "finished": 0, "results_file"
: "/root/.ansible_async/276700021098.73070", "started": 1}
192.168.100.65 | SUCCESS => {"ansible_job_id": "877427488272.72032", "changed": true, "finished": 0, "results_file"
: "/root/.ansible_async/877427488272.72032", "started": 1}
```

关于同步、异步以及异步时的并行数、轮训间隔对ansible的影响，通过以下示例说明：

当有6个节点时，仅就释放shell的速度而言，以下几种写法：

```
# 同步模式，大约10秒返回。
ansible centos -m command -a "sleep 5" -o
# 异步模式，分两批执行。大约10秒返回。
ansible centos -B200 -P 1 -m command -a "sleep 5" -o -f 5
# 异步模式，和上一条命令时间差不多，但每次检查时间长一秒，所以可能会稍有延迟。大约11-12秒返回。
ansible centos -B200 -P 2 -m command -a "sleep 5" -o -f 5
# 异步模式，一批就执行完，大约5-6秒返回。
ansible centos -B200 -P 1 -m command -a "sleep 5" -o -f 6
# 异步模式，一批就完成，大约5-6秒完成。
ansible centos -B200 -P 2 -m command -a "sleep 5" -o -f 6
# 异步模式，分两批，且检查时间过长。即使只睡眠5秒，但仍需要10秒才能判断该批执行结束。所以大约20秒返回。
ansible centos -B200 -P 10 -m command -a "sleep 5" -o -f 5
# 异步模式，一批执行完成，但检查时间超过睡眠时间，因此大约10秒返回。
ansible centos -B200 -P 10 -m command -a "sleep 5" -o -f 6
```

在异步执行任务时，需要注意那些有依赖性的任务。对于那些对资源要求占有排它锁的任务，如yum，不应该将Poll的间隔设置为0。如果设置为0，很可能会导致资源阻塞。

总结来说,大概有以下一些场景需要使用到ansible的异步特性：

- 某个task需要运行很长的时间,这个task很可能会达到ssh连接的timeout
- 没有任务是需要等待它才能完成的，即没有任务依赖此任务是否完成的状态
- 需要尽快返回当前shell

当然也有一些场景不适合使用异步特性：

- 这个任务是需要运行完后才能继续另外的任务的
- 申请排它锁的任务

当然，对于有任务依赖性的任务，也还是可以使用异步模式的，只要检查它所依赖的主任务状态已完成就可以。例如，要配置nginx，要求先安装好nginx，在配置nginx之前先检查yum安装的状态。

```
- name: 'YUM - fire and forget task'
  yum: name=nginx state=installed
  async: 1000
  poll: 0
  register: yum_sleeper

- name: 'YUM - check on fire and forget task'
  async_status: jid={{ yum_sleeper.ansible_job_id }}
  register: job_result
  until: job_result.finished
  retries: 30
```

7.3 ansible的-t选项妙用

ansible的"-t"或"--tree"选项是将ansible的执行结果按主机名保存在指定目录下的文件中。

有些时候，ansible执行起来的速度会非常慢，这种慢体现在即使执行的是一个立即返回的简单命令(如ping模块)，也会耗时很久，且不是因为ssh连接慢导致的。如果使用-t选项，将第一次执行得到的结果按inventory中定义的主机名保存在文件中，下次执行到同一台主机时速度将会变快很多，即使之后不再加上-t选项，也可以在一定时间内保持迅速执行。即使执行速度正常（如执行一个Ping命令0.7秒左右），使用-t选项也可以在此基础上变得更快。

除了使用-t选项，使用重定向将结果重定向到某个文件中也是一样的效果。至于为何会如此，我也不知道，是在无

意中测试出来的。有必要指出：我在CentOS 6.6上遇到过这样的问题，但并不是总会如此，且在CentOS 7上正常。因此，如果你也出现了这样的问题，可以参考这种偏方。

以CentOS 6.6安装的ansible 2.3为例，正常执行ansible会非常慢，使用-t可以解决这个问题。如下。

没有使用-t时：移除dos2unix包所需时间为13秒多。

```
time ansible centos -B200 -P 0 -m yum -a "name=dos2unix state=removed" -o -f 6
192.168.100.60 | SUCCESS => {"ansible_job_id": "987125400759.10653", "changed": true, "finished": 0, "results_file"
: "/root/.ansible_async/987125400759.10653", "started": 1}
192.168.100.63 | SUCCESS => {"ansible_job_id": "735153954362.74074", "changed": true, "finished": 0, "results_file"
: "/root/.ansible_async/735153954362.74074", "started": 1}
192.168.100.61 | SUCCESS => {"ansible_job_id": "192721090554.9813", "changed": true, "finished": 0, "results_file":
"/root/.ansible_async/192721090554.9813", "started": 1}
192.168.100.64 | SUCCESS => {"ansible_job_id": "494724112239.73269", "changed": true, "finished": 0, "results_file"
: "/root/.ansible_async/494724112239.73269", "started": 1}
192.168.100.59 | SUCCESS => {"ansible_job_id": "2259915341.10078", "changed": true, "finished": 0, "results_file":
"/root/.ansible_async/2259915341.10078", "started": 1}
192.168.100.65 | SUCCESS => {"ansible_job_id": "755223232484.73025", "changed": true, "finished": 0, "results_file"
: "/root/.ansible_async/755223232484.73025", "started": 1}

real    0m13.746s
user    0m1.288s
sys     0m1.417s
```

使用-t选项后：安装dos2unix只需1.9秒左右。

```
time ansible centos -B200 -P 0 -m yum -a "name=dos2unix state=installed" -o -f 6 -t /tmp/a


real    0m1.933s
user    0m0.398s
sys     0m0.900s
```

之后即使不再使用-t选项，对同样的主机进行操作，速度也会变得非常快。

```
time ansible centos -B200 -P 0 -m yum -a "name=dos2unix state=removed" -o -f 6

real    0m1.730s
user    0m0.892s
sys     0m0.572s
```

至于保存的内容为何？实际上仅仅只是保存了普通的输出内容而已。

```
ll /tmp/a/
total 24
-rw-r--r-- 1 root root 145 May 28 15:54 192.168.100.59
-rw-r--r-- 1 root root 145 May 28 15:54 192.168.100.60
-rw-r--r-- 1 root root 143 May 28 15:54 192.168.100.61
-rw-r--r-- 1 root root 143 May 28 15:54 192.168.100.63
-rw-r--r-- 1 root root 145 May 28 15:54 192.168.100.64
-rw-r--r-- 1 root root 145 May 28 15:54 192.168.100.65

cat /tmp/a/192.168.100.59
{"ansible_job_id": "659824383578.10145", "changed": true, "finished": 0, "results_file": "/root/.ansible_async/6598
24383578.10145", "started": 1}
```

7.4 优化ansible速度

最初，ansible的执行效率和saltstack(基于zeromq消息队列的方式)相比要慢的多的多，特别是被控节点量很大的时候。但是ansible发展到现在，它的效率得到了极大的改善。在被控节点不太多的时候，默认的设置已经够快，即使被控节点数量巨大的时候，也可以通过一些优化，极大的提高其执行效率。

前面"-t"选项也算是一种提速方式，但算是"bug"式的问题，所以没有通用性。

7.4.1 设置ansible开启ssh长连接

ansible天然支持openssh，默认连接方式下，它对ssh的依赖性非常强。所以优化ssh连接，在一定程度上也在优化ansible。其中一点是开启ssh的长连接，即长时间保持连接状态。

要开启ssh长连接，要求ansible端的openssh版本高于或等于5.6。使用`ssh -V`可以查看版本号。然后设置ansible使用ssh连接被控端的连接参数，此处修改/etc/ansible/ansible.cfg，在此文件中启动下面的连接选项，其中ControlPersist=5d是控制ssh连接会话保持时长为5天。

```
ssh_args = -C -o ControlMaster=auto -o ControlPersist=5d
```

除此之外直接设置/etc/ssh/ssh_config(不是sshd_config，因为ssh命令是客户端命令)中对应的长连接项也是可以的。
开启长连接后，在会话过期前会一直建立连接，在netstat的结果中会看到ssh连接是一直established状态，且会在当前用户家目录的".ansible/cp"目录下生成一些socket文件，每个会话一个文件。
例如：执行一次ad-hoc操作。

```
ansible centos -m ping
```

查看netstat，发现ssh进程的会话一直是established状态。

```
shell> netstat -tnalp

Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address      State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*            LISTEN      1143/sshd
tcp        0      0 127.0.0.1:25            0.0.0.0:*            LISTEN      2265/master
tcp        0      0 192.168.100.62:58474    192.168.100.59:22    ESTABLISHED 31947/ssh: /root/.a
tcp        0     96 192.168.100.62:22       192.168.100.1:8189   ESTABLISHED 29869/sshd: root@pt
tcp        0      0 192.168.100.62:37718    192.168.100.64:22    ESTABLISHED 31961/ssh: /root/.a
tcp        0      0 192.168.100.62:38894    192.168.100.60:22    ESTABLISHED 31952/ssh: /root/.a
tcp        0      0 192.168.100.62:48659    192.168.100.61:22    ESTABLISHED 31949/ssh: /root/.a
tcp        0      0 192.168.100.62:33546    192.168.100.65:22    ESTABLISHED 31992/ssh: /root/.a
tcp        0      0 192.168.100.62:54824    192.168.100.63:22    ESTABLISHED 31958/ssh: /root/.a
tcp6       0      0 :::22                   :::*                 LISTEN      1143/sshd
tcp6       0      0 ::1:25                  :::*                 LISTEN      2265/master
```

且家目录下"~/.ansible/cp/"下会生成对应的socket文件。

```
ls -l ~/.ansible/cp/
total 0
srw------- 1 root root 0 Jun  3 18:26 5c4a6dce87
srw------- 1 root root 0 Jun  3 18:26 bca3850113
srw------- 1 root root 0 Jun  3 18:26 c89359d711
srw------- 1 root root 0 Jun  3 18:26 cd829456ec
srw------- 1 root root 0 Jun  3 18:26 edb7051c84
srw------- 1 root root 0 Jun  3 18:26 fe17ac7eed
```

7.4.2 开启pipelining

pipeline也是openssh的一个特性。在ansible执行每个任务的流程中，有一个过程是将临时任务文件put到一个ansible端的一个临时文件中，然后sftp传输到远端，然后通过ssh连接过去远程执行这个任务。如果开启了pipelining，一个任务的所有动作都在一个ssh会话中完成，也会省去sftp到远端的过程，它会直接将要执行的任务在ssh会话中进行。

开启pipelining的方式是配置文件(如ansible.cfg)中设置pipelining=true，默认是false。

```
shell> grep '^pipelining' /etc/ansible/ansible.cfg
pipelining = True
```

但是要注意，如果在ansible中使用sudo命令的话(ssh user@host sudo cmd)，需要在被控节点的/etc/sudoers中禁用"requiretty"。

之所以要设置/etc/sudoers中的requiretty，是因为ssh远程执行命令时，它的环境是非登录式非交互式shell，默认不会分配tty，没有tty，ssh的sudo就无法关闭密码回显(使用"-tt"选项强制SSH分配tty)。所以出于安全考虑，/etc/sudoers中默认是开启requiretty的，它要求只有拥有tty的用户才能使用sudo，也就是说ssh连接过去不允许执行sudo。可以通过visudo编辑配置文件，注释该选项来禁用它。

```
grep requiretty /etc/sudoers
# Defaults    requiretty
```

修改设置/etc/sudoers是在被控节点上进行的(或者ansible连接过去修改)，其实在ansible端也可以解决sudo的问题，只需在ansible的ssh参数上加上"-tt"选项即可。

```
ssh_args = -C -o ControlMaster=auto -o ControlPersist=5d -tt
```

以下是开启pipelining前ansible执行过程，其中将很多不必要的信息使用.....来替代了。

```
######## 开启pipelining前，执行ping模块的过程 ########
Using /etc/ansible/ansible.cfg as config file
Loading callback plugin minimal of type stdout, v2.0 from /usr/lib/python2.7/site-packages/ansible/plugins/callback
/__init__.pyc
META: ran handlers
Using module file /usr/lib/python2.7/site-packages/ansible/modules/system/ping.py

# 首先建立一次ssh连接，获取远端当前用户家目录，用于存放稍后的临时任务文件
<192.168.100.65> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.65> SSH: EXEC ssh -vvv -C ............... 192.168.100.65 '/bin/sh -c '"'"'echo ~ && sleep 0'"'"''
<192.168.100.65> (0, '/root\n', ................)

# 再次建立ssh连接，创建临时文件目录
<192.168.100.65> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.65> SSH: EXEC ssh -vvv -C ..................... 192.168.100.65 '/bin/sh -c '"'"'( umask 77 && mkdir -p
"......." ) && sleep 0'"'"''
<192.168.100.65> (0, 'ansible-tmp-1496489511.13-10633592020239=/root/.ansible/tmp/ansible-tmp-1496489511.13-1063359
2020239\n', '.............')

# 将任务放入到本地临时文件中，然后使用sftp传输到远端
<192.168.100.65> PUT /tmp/tmp2_VKGo TO /root/.ansible/tmp/ansible-tmp-1496489511.13-10633592020239/ping.py
<192.168.100.65> SSH: EXEC sftp -b - -vvv -C ................. '[192.168.100.65]'
<192.168.100.65> (0, 'sftp> put /tmp/tmp2_VKGo /root/.ansible/tmp/ansible-tmp-1496489511.13-10633592020239/ping.py\
n', '......................')

# 又一次建立ssh连接，对任务文件进行授权
<192.168.100.65> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.65> SSH: EXEC ssh -vvv -C ............. 192.168.100.65 '/bin/sh -c '"'"'chmod u+x .......... /ping.py
&& sleep 0'"'"''
<192.168.100.65> (0, '', '........................')

# 最后执行任务，完成任务后删除任务文件，并返回ansible端信息，注意ssh -tt选项，它强制为ssh会话分配tty，这样可以执行sudo命令
<192.168.100.65> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.65> SSH: EXEC ssh -vvv -C .............. -tt 192.168.100.65 '/bin/sh -c '"'"'/usr/bin/python .........
../ping.py; rm -rf "..........." > /dev/null 2>&1 && sleep 0'"'"''
<192.168.100.65> (0, '\r\n{"invocation": {"...............')
以下是开启pipelining后，ansible执行过程。
########### 开启pipelining后，执行ping模块的过程 ########
Using /etc/ansible/ansible.cfg as config file
Loading callback plugin minimal of type stdout, v2.0 from /usr/lib/python2.7/site-packages/ansible/plugins/callback
/__init__.pyc
META: ran handlers
Using module file /usr/lib/python2.7/site-packages/ansible/modules/system/ping.py

# 只建立一次ssh连接，所有动作都在这一个ssh连接中完成，由于没有使用-tt选项，所以需要在被控主机上禁用requiretty选项
<192.168.100.65> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.100.65> SSH: EXEC ssh -vvv -C ............ 192.168.100.65 '/bin/sh -c '"'"'/usr/bin/python && sleep 0'"'"'
'
<192.168.100.65> (0, '\n{".............')
```

从上面的过程对比中可以看到，开启pipelining后，每次执行任务时都大量减少了ssh连接次数(只需要一次ssh连接)，且省去了sftp传输任务文件的过程，因此在管理大量节点时能极大提升执行效率。

7.4.3 修改ansible执行策略

默认ansible在远程执行任务是按批并行执行的，一批控制多少台主机由命令行的"-f"或"--forks"选项控制。例如，默认的并行进程数是5，如果有20台被控主机，那么只有在每5台全部执行完一个任务才继续下一批的5台执行该任务，即使中间某台机器性能较好，完成速度较快，它也会空闲地等待在那，直到所有20台主机都执行完该任务才会以同样的方式继续下一个任务。如下所示：

h1 h2 h3 h4 h5(T1)-->h6 h7 h8 h9 h10(T1)...-->h16 h17 h18 h19 h20(T1)-->h1 h2 h3 h4 h5(T2)-->.....

在ansible 2.0中，添加了一个策略控制选项strategy，默认值为"linear"，即上面按批并行处理的方式。还可以设置strategy的值为"free"。
在free模式下，ansible会尽可能快的切入到下一个主机。同样是上面的例子，首先每5台并行执行一个任务，当其中某一台机器由于性能较好提前完成了该任务，它不会等待其他4台完成，而是会跳出该任务让ansible切入到下一台机器来执行该任务。也就是说，这种模式下，一台主机完成一个任务后，另一台主机会立即执行任务，它是"前赴

后继"的方式。如下所示：

h1 h2 h3 h4 h5(T1)-->h1 h2 h3 h4 h6(T1)-->h1 h3 h4 h6 h7(T1)-->.....-->h17 h18 h19 h20(T1) h1(T2)-->h18 h19 h20(T1) h1 h2(T2)-->...

设置的方式如下：

```
- hosts: all
  strategy: free
  tasks:
  ...
```

7.4.4 设置facts缓存

ansible或ansible-playbook默认总是先收集facts信息。在被控主机较少的情况下，收集信息还可以容忍，如果被控主机数量非常大，收集facts信息会消耗掉非常多时间。

可以设置"gather_facts: no"来禁止ansible收集facts信息，但是有时候又需要使用facts中的内容，这时候可以设置facts的缓存。例如，在空闲的时候收集facts，缓存下来，在需要的时候直接读取缓存进行引用。

ansible的配置文件中可以修改'gathering'的值为'smart'、'implicit'或者'explicit'。smart表示默认收集facts，但facts已有的情况下不会收集，即使用缓存facts；implicit表示默认收集facts，要禁止收集，必须使用`gather_facts: False`；explicit则表示默认不收集，要显式收集，必须使用`gather_facts: Ture`。

在使用facts缓存时(即设置为smart)，ansible支持两种facts缓存：redis和jsonfile。

例如，以下是/etc/ansible/ansible.cfg中jsonfile格式的缓存配置方法。

```
[defaults]
gathering = smart
fact_caching_timeout = 86400
fact_caching = jsonfile
fact_caching_connection = /path/to/cachedir
```

这里设置的缓存过期时间为86400秒，即缓存一天。缓存的json文件放在/path/to/cachedir目录下，各主机的缓存文件以主机名命名。缓存文件是一个json文件，要查看缓存文件，如/path/to/cachedir/192.168.100.59中的内容，使用如下语句即可。

```
cat /path/to/cachedir/192.168.100.59 | python -m json.tool
```

8.playbook杂项

实在是因为ansibel的官方文档太烂太不人性化了，一大堆的知识点乱七八糟的分布。所以，借此文收集一些乱七八糟的知识点，所以，此文会不断更新。

8.1 指定运行play的主机delegate_to和local_action

默认情况下，ansible会在指定的hosts的所有主机上运行整个play。但有些时候，有些任务只想在某些主机上运行，例如，在haproxy主机上运行haproxy相关配置任务，在后端web主机上运行nginx或apache相关配置任务。

这时可以使用delegate_to选项来分配任务，且**分配的主机对象可以不用在hosts列表中，只要它在inventory中即可**。例如：

```
---
    - hosts: 192.168.100.59,192.168.100.150
      tasks:
        - shell: echo "{{inventory_hostname}}" >/tmp/hello.txt
          delegate_to: localhost
```

上面的playbook中指定了两个被控节点192.168.100.59和192.168.100.150，但是它们不会执行这个任务，而是在ansible的本地执行。但注意，**这个任务在本地会执行两次**，因为有两个被控节点，**且变量"inventory_hostname"的值不是localhost，而是被控节点的主机名**。要想让某个任务只运行一次，可以使用run_once选项，见下文。

如果要将任务分配给多个主机，可以使用with_items的方式进行迭代。例如：

```
---
    - hosts: 192.168.100.59,192.168.100.150
      tasks:
        - shell: echo "{{inventory_hostname}}" >/tmp/hello.txt
          delegate_to: "{{item}}"
          with_items:
             - localhost
             - 192.168.100.66
```

如果是分配给ansible本地端执行任务，则还有一个更简介的方式：使用local_action。例如，下面两个task是等价的。

```
tasks:
    - shell: echo haha >>/tmp/hello.txt
      delegate_to: localhost
    - local_action: shell echo haha >>/tmp/hello.txt
```

实际上，local_action的方式是延续了老版本anisble的action组件。在以前的版本中，任何一个模块都可以使用action来指定。例如下面两个task是等价的。

```
tasks:
    - shell: echo haha >>/tmp/hello.txt
    - action: shell echo haha >>/tmp/hello.txt
```

8.2 只运行一次run_once

使用delegate_to或local_action都可以分配任务给指定主机，但如果被控节点有多台时，这些分配后的任务会多次执行。再例如，在备份某个数据库时，由于主从复制结构下的可能会有多台数据库服务器，但实际上只需备份其中

一台就够了，这时可以指定run_once，让任务仅只执行一次。

```
---
    - hosts: mysql_rep
      tasks:
        - name: backup mysql data
          shell: mysqldump ......
          run_once: True
```

既然是只执行一次任务，那么肯定会挑其中一台被控节点。挑选时是按照hosts中的顺序挑选的。

8.3 分批执行play

最典型的案例是：修改负载均衡后端服务器时，要先摘除后端server，再修改后重启，最后加入到负载server的后端列表中。默认情况下，ansible会将后端服务器一次性全部摘除(假如hosts指定了所有后端主机)，再依次全部修改、重启、添加会后端列表。这样就会导致修改后端服务器的过程中无法向外提供服务，显然这是不合理的。当然，也可以使用hosts限定被控主机列表。

ansible支持使用serial选项实现分批执行play的功能。例如：

```
---
    - hosts: 192.168.100.[59:68]
      serial: 5
      tasks:
        - task1
        - task2
        - task3
```

在上面的示例中，指定了10台被控节点，如果不使用serial，那么ansible会先控制10台节点上执行task1，执行完后再控制这10台节点执行task2，同理task3。但上面指定了serial，那么ansible将先控制5台主机执行task1、task2、task3，执行完后再控制5台主机执行task1、task2、task3，直到所有主机都执行结束。

这和ansible的"--forks"选项的并行不一样，"--forks"选项指定的并行是每个task内分批执行，例如默认情况下的"--forks=5"，ansible会先控制5台主机执行task1，这5台执行结束后再控制另外5台继续执行task1，同理task2和task3也如此。

以下面的过程来描述serial和"--forks"的区别，假如有10台主机0-9，serial和"--forks"都指定为5。

- serial的并行方式：

  01234：task1、task2、task3
  56789：task1、task2、task3

- "--forks"的并行方式：

  01234：task1
  56789：task1
  01234：task2
  56789：task2
  01234：task3
  56789：task3

回到serial。serial的指定方式有多种，还支持百分比、列表的形式。例如：

```
---
    - hosts: host[1:10]
      serial: 30%
```

表示每批有hosts的10分之3主机执行play。

```
---
    - hosts: host[1:10]
      serial:
        - 1
        - 3
        - 60%
```

表示第一批只有一台主机执行play，第2批有3台主机执行play，第三批有60%的主机执行play。

8.4 分批执行play时的最大失败百分比

假如所有负载均衡的后端server做为被控节点，且所有被控节点执行任务都失败，那么即使分批执行play，最终的结果也依然是无法提供服务

ansible也考虑到了这种情况。可以使用`max_fail_percentage`指定最大失败百分比，例如指定30%，那么分批执行时，如果有30%的主机执行play失败时，自动退出该play(不是退出ansibel，而是play)。

```
---
    - hosts: 192.168.100.[59:68]
      serial: 5
      max_fail_percentage: 30%
      tasks:
        - task1
        - task2
        - task3
```

8.5 错误处理—忽略错误

Ansible默认会检查命令和模块的返回状态，并进行相应的错误处理，**默认是遇到错误就中断playbook的执行**，这些默认行为都是可以改变的。

对于command和shell模块，如果命令返回非零状态码，则ansible判定这2个模块执行失败，可以通过ignore_errors忽略返回状态码。如下：

```
- name: this will not be counted as a failure
 command: /bin/false
 ignore_errors: yes
```

8.6 错误处理—自定义错误判断条件

命令不依赖返回状态码来判定是否执行失败，而是要查看命令返回内容来决定，比如返回内容中包括FAILED字符串，则判定为失败。示例如下：

```
- name: command returns FAILED when it fails
 command: /usr/bin/test_cmd
 register: rst
 failed_when: "'FAILED' in rst.stderr"
```

此外ansible会自动判断模块执行状态，也就是changed=True/False，但可以自定义达到changed=True的条件，示例如下：

```
- name: copy in nginx conf
template: src=nginx.conf.j2 dest=/etc/nginx/nginx.conf

- name: validate nginx conf
 shell: "/usr/local/nginx/sbin/nginx -t"
 register: command_result
 changed_when: command_result.stdout.find('successful')
```

上面的例子中，当shell模块的命令返回中有"successful"字符串时，则为changed=True状态，下面这个设定将永远也不会达到changed=True状态。

```
- name: validate nginx conf
 shell: "/usr/local/nginx/sbin/nginx -t"
 changed_when: false
```

9.playbook示例：编译安装httpd

以下是playbook的内容。它的处理流程是：

1. 先在本地下载apr,apr-util,httpd共3个.tar.gz文件。
2. 解压这3个文件。
3. 安装pcre和pcre-devel依赖包。
4. 编译安装apr。
5. 编译安装apr-util。
6. 编译安装httpd。

```
---
        - hosts: all
          tasks:
                - name: download apr,apr-util,httpd
                  get_url: url="{{item}}" dest=/root/pkg
                  with_items:
                        - https://mirrors.tuna.tsinghua.edu.cn/apache/apr/apr-1.6.2.tar.gz
                        - https://mirrors.tuna.tsinghua.edu.cn/apache/apr/apr-util-1.6.0.tar.gz
                        - https://mirrors.tuna.tsinghua.edu.cn/apache/httpd/httpd-2.4.27.tar.gz
                  delegate_to: localhost
                  run_once: True
                - unarchive: src="/root/pkg/{{item}}" dest=/root/
                  with_items:
                        - httpd-2.4.27.tar.gz
                        - apr-1.6.2.tar.gz
                        - apr-util-1.6.0.tar.gz
                  tags: unarchive

                - name: install pcre and pcre-devel and expat-devel
                  yum: name="{{item}}" state=installed
                  with_items:
                        - pcre
                        - pcre-devel
                        - expat-devel

                - name: complie apr
                  shell: cd /root/apr-1.6.2 && ./configure --prefix=/usr/local/apr && make && make install

                - name: complie apr-util
                  shell: |
                        cd /root/apr-util-1.6.0
                        ./configure --prefix=/usr/local/apr-util --with-apr=/usr/local/apr
                        make && make install

                - name: complie httpd
                  shell: |
                        cd /root/httpd-2.4.27
                        ./configure --prefix=/usr/local/apache --sysconfdir=/etc/apache \
                        --enable-mpms-shared=all \
                        --with-z --with-pcre \
                        --with-apr=/usr/local/apr \
                        --with-apr-util=/usr/local/apr-util \
                        --with-mpm=event
                        make && make install
```

编译完成后，还有一系列操作，比如设置PATH环境变量、设置man路径、修改配置文件、启动httpd等。这些就懒得放进去了。