O'REILLY

Запускаем Ansible

Простой способ автоматизации управления конфигурациями и развертыванием приложений

Третье издание

Бас Мейер
Лорин Хохштейн
Рене́ Мозер

BOOKS.KZ

Бас Мейер, Лорин Хохштейн и Рене Мозер

Запускаем Ansible

ТРЕТЬЕ ИЗДАНИЕ

THIRD EDITION

Ansible: Up and Running

Automating Configuration Management and Deployment the Easy Way

Bas Meijer, Lorin Hochstein, and René Moser

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

ТРЕТЬЕ ИЗДАНИЕ

Запускаем Ansible

Простой способ автоматизации управления конфигурациями и развертыванием приложений

Бас Мейер, Лорин Хохштейн и Рене Мозер

BOOKS.KZ
2023

УДК 004.4
ББК 32.372
М42

М42 Бас Мейер, Лорин Хохштейн и Рене Мозер

Запускаем Ansible. Простой способ автоматизации управления конфигурациями и развертыванием приложений. 3-е изд. / пер. с англ. А. Н. Киселева – М.: ДМК Пресс, 2023. – 482 с.: ил.

ISBN 978-6-01763-867-2

Среди множества имеющихся инструментов управления конфигурациями Ansible выделяется своими преимуществами, такими как небольшой объем, отсутствие необходимости устанавливать что-либо на управляемые хосты и простота в изучении и освоении.

Наиболее существенное отличие этого издания от предыдущего – добавление шести новых глав, охватывающих применение контейнеров, фреймворка Molecule, платформы автоматизации Ansible Automation Platform и коллекций Ansible; приемы создания образов, поддержки облачной инфраструктуры и реализации конвейеров CI/CD.

Книга предназначена разработчикам инструментов infrastructure as a code для автоматизации задач по подготовке и конфигурированию инфраструктуры.

ISBN 978-1-098-10915-8 (англ.)
ISBN 978-6-01763-867-2 (казах.)

Оглавление

Предисловие к третьему изданию

Со времени публикации второго издания этой книги в 2017 году многое изменилось в мире Ansible и Python, включая выход нескольких новых версий. Немало изменений произошло и за пределами проекта: например, Red Hat, компания-основательница проекта Ansible, была куплена корпорацией IBM. Однако это никак не повлияло не проект Ansible: он все так же продолжает развиваться и привлекать новых пользователей. Развитие облачных и контейнерных технологий тоже значительно повлияло на общий ландшафт.

Мы внесли множество изменений в это издание. Наиболее существенное – добавление шести новых глав, охватывающих контейнеры, Molecule, коллекции Ansible, создание образов, поддержку облачной инфраструктуры и CI/CD. Мы также обновили и дополнили другие главы, уделив больше внимания передовым приемам разработки программного обеспечения и фреймворкам тестирования, помогающим проверить код и придать дополнительную уверенность в нем. Мы обновили все примеры кода для совместимости с последней версией Ansible, а также все сведения, что связаны с Python. Мы постарались отразить все важные изменения, произошедшие в период между 2017 и 2022 годами. Мы могли бы продолжать и дальше, но не будем этого делать, потому что вы сами, погрузившись в книгу, сможете увидеть, насколько далеко продвинулся Ansible.

Обозначения и соглашения, принятые в этой книге

В книге действуют следующие типографские соглашения.

Курсив

Используется для обозначения новых терминов, имен файлов и их расширений.

`Моноширинный шрифт`

Используется для оформления листингов программ, а также в обычном тексте для обозначения элементов программы, таких как имена переменных или функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный полужирный шрифт`

Используется для выделения команд и другого текста, который должен быть набран самим пользователем.

`Моноширинный курсив`

Используется для выделения текста, который нужно заменить данными пользователя или значениями, определяемыми контекстом.

Так обозначаются примечания общего характера.

Так обозначаются предупреждения и предостережения.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** на странице с описанием соответствующей книги.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются имя автора, название книги, издательство и ISBN, например: «*Мейер Б., Хохштейн Л., Мозер Р.* Запускаем Ansible. М.: O'Reilly; ДМК Пресс, 2023. Copyright © 2023 O'Reilly Media, Inc., 978-1-098-10915-8 (англ.), 978-5-97060-513-4 (рус.)».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры, для того чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Благодарности

От Лорин

Мои благодарности Яну-Пит Менсу (Jan-Piet Mens), Мэтту Джейнсу (Matt Jaynes) и Джону Джарвису (John Jarvis) за отзывы в процессе написания книги. Спасибо Айзаку Салдана (Isaac Saldana) и Майку Рован (Mike Rowan) из SendGrid за поддержку этого начинания. Благодарю Майкла ДеХаана (Michael DeHaan) за создание Ansible и поддержку сообщества, которое разрослось вокруг продукта, а также за отзывы о книге, включая объяснения, почему было выбрано название Ansible. Спасибо моему редактору Брайану Андерсону (Brian Anderson) за его безграничное терпение в работе со мной.

Спасибо маме и папе за их неизменную поддержку; моему брату Эрику (Eric), настоящему писателю в нашей семье; двум моим сыновьям Бенджамину (Benjamin) и Джулиану (Julian). И наконец, спасибо моей жене Стейси (Stacy) за все.

От Рене

Спасибо моей семье, моей жене Симоне (Simone) за любовь и поддержку, моим трем деткам, Джил (Gil), Сарине (Sarina) и Лиан (Léanne), за свет и радость, что они привнесли в мою жизнь; спасибо всем, кто внес свой вклад в развитие Ansible, спасибо вам за ваш труд; и особое спасибо Маттиасу Блейзеру (Matthias Blaser), познакомившему меня с Ansible.

От Баса

Спасибо Хенку де Йонгу (Henk de Jongh), открывшему мне двери в издательство O'Reilly в начале девяностых. Спасибо Джорди Клементу (Jordi Clement), познакомившему меня с Ansible. Спасибо всем, кто внес свой вклад в развитие Ansible, спасибо вам за ваш труд. Спасибо всем потрясающим командам, в которых я формировался и рос как специалист: Antraciet, Integration and Engineering at IMC, iWelcome, CD@GS, Vendora, CDaaS, Spitfire, Colibri, Wilbur, Duck Tape, Purple, ICC. Спасибо Фрэнку Безему (Frank Bezema) и Вернеру Дейкерману (Werner Dijkerman). Спасибо Джири Хугланду (Jiri Hoogland) и Vola Dynamics за поддержку развития свободного программного обеспечения. Большое спасибо Тону Керстену (Ton Kersten) и Кериму Сатирли (Kerim Satirli)! Отдельное спасибо Яну-Пит Менсу (Jan-Piet Mens), Марику Ветте (Marek Vette) и Джону Каннифу (John Cunniff) за отзывы! Спасибо Серджу ван Джиндерахтеру (Serge van Ginderachter), Люку Мерфи (Luke Murphy), Роберту де Боку (Robert de Bock), Винсенту ван дер Кассену (Vincent van der Kussen), Дагу Вирсу (Dag Wieers), Арнабу Синху (Arnab Sinha), Ананду Буддефу (Anand Buddhef) и все остальным участникам встреч на Ansible Benelux Meetup: без них я не смог бы стать одним из авторов этой книги. Спасибо Саре Грей (Sarah Grey) за редактирование этой книги. И спасибо моей семье за любовь и поддержку.

Глава 1

Введение

Сейчас интересное время для работы в ИТ-индустрии. Мы не поставляем нашим клиентам программное обеспечение, установив его на одну-единственную машину и совершая дежурные звонки раз в день. Вместо этого мы медленно превращаемся в облачных инженеров.

Сейчас мы развертываем программные приложения, связывая воедино службы, которые работают в распределенной компьютерной сети и взаимодействуют по разным сетевым протоколам. Типичное приложение может включать веб-серверы, серверы приложений, систему кеширования данных в оперативной памяти, очереди задач, очереди сообщений, базы данных SQL, NoSQL-хранилища и балансировщики нагрузки.

Мы также должны убедиться в наличии достаточного количества ресурсов, и в случае ошибок в системе (а они будут случаться) мы элегантно выйдем из ситуации. Также имеются второстепенные службы, которые нужно разворачивать и поддерживать, такие как служба журналирования, мониторинга и анализа. Имеются и внешние службы, с которыми нужно устанавливать взаимодействие, например с интерфейсами «инфраструктура как сервис» (Infrastructure-as-a-Service, IaaS) для управления экземплярами виртуальных машин[1].

Мы можем связать эти службы вручную: запустить нужные серверы, зайти на каждый из них, установить пакеты приложений, отредактировать конфигурационные файлы и т. д. Но это серьезный труд. Такой процесс требует много времени, способствует появлению множества ошибок и просто утомляет, особенно в третий или четвертый раз. А работа вручную над более сложными задачами, как, например, установка облака OpenStack для вашего приложения, – так и просто сумасшествие. Есть способ лучше.

Если вы читаете эту книгу, значит, уже загорелись идеей управления конфигурациями и теперь рассматриваете Ansible как средство управ-

[1] Рекомендую превосходные книги Томаса А. Лимонцелли (Thomas A. Limoncelli), Страта Р. Чалупа (Strata R. Chalup) и Кристины Дж. Хоган (Christina J. Hogan) «The Practice of Cloud System Administration», тома 1 и 2 (Addison-Wesley), и книгу Мартина Клеппмана (Martin Kleppman) «Designing Data-Intensive Applications» (O'Reilly).

ления. Кем бы вы ни были, разработчиком, развертывающим свой код в промышленном окружении, или системным администратором, ищущим лучшие средства автоматизации, я думаю, вы найдете в лице Ansible превосходное решение ваших проблем.

Примечание о версиях

Все примеры кода в этой книге были протестированы в версии Ansible 2.9.0, которая на момент написания книги являлась самой свежей. Ansible Tower включает версию 2.9.27 в последнем выпуске. Версия Ansible 2.8 завершила свой жизненный путь выпуском 2.8.20 в апреле 2021 года.

Многие годы сообщество Ansible активно разрабатывало новые роли и модули, в результате чего на свет появились тысячи модулей и более 20 000 ролей. Сложности, неизбежно возникающие при управлении такими масштабными проектами, привели создателей к необходимости реорганизовать Ansible и разделить его на три части:

- *компоненты ядра*, созданные командой Ansible;
- *сертифицированные* разработки, созданные бизнес-партнерами Red Hat;
- разработки *сообщества*, созданные тысячами энтузиастов по всему миру.

Ansible 2.9 имеет массу встроенных возможностей, но последующие версии будут более модульными. Эта новая организация проекта упрощает управление ими.

Примеры, представленные в этой книге, должны работать в разных версиях Ansible, но вообще смена версии предполагает тестирование, о чем мы поговорим в главе 14.

Откуда взялось название «Ansible»?

Название заимствовано из области научной фантастики. *Ansible* – это устройство связи, способное передавать информацию быстрее скорости света. Писатель Урсула Ле Гуин впервые представила эту идею в своем романе «Планета Роканнона», а остальные писатели-фантасты подхватили ее. Если быть более точным, Майкл ДеХаан, сооснователь проекта, позаимствовал название Ansible из книги Орсона Скотта Карда «Игра Эндера». В этой книге ansible использовался для одновременного контроля большого числа кораблей, удаленных на огромные расстояния. Подумайте об этом как о метафоре контроля удаленных серверов.

Ansible: область применения

Ansible часто описывают как *инструмент управления конфигурациями*, и обычно он упоминается в том же контексте, что и Chef, Puppet и Salt. Когда мы говорим об *управлении конфигурациями*, то часто подразумеваем описание состояния серверов в некотором виде, а затем применение специальных средств для приведения серверов в это состояние: установку необходимых пакетов приложений, копирование конфигурационных файлов с определенными разрешениями в файловой системе, запуск необходимых служб и т. д. Подобно другим средствам управления, Ansible предоставляет *предметно-ориентированный язык (Domain Specific Language, DSL)*, который используется для описания состояний серверов.

Эти инструменты также можно использовать для развертывания программного обеспечения. Под развертыванием мы часто подразумеваем процесс получения двоичного кода из исходного (если необходимо), копирования необходимых файлов на сервер(ы), добавление конфигурационных свойств и переменных окружения и запуск служб в определенном порядке. Capistrano и Fabric – два примера инструментов с открытым кодом для развертывания приложений. Ansible тоже является превосходным инструментом как для развертывания, так и для управления конфигурациями программного обеспечения. Использование единой системы управления конфигурациями и развертыванием значительно упрощает жизнь системным администраторам.

Некоторые специалисты отмечают необходимость согласования развертывания, когда в процесс вовлечено несколько удаленных серверов и операции должны осуществляться в определенном порядке. Например, базу данных нужно установить до установки веб-серверов или выводить веб-серверы из-под управления балансировщика нагрузки только по одному, чтобы система не прекращала работу во время обновления. Система Ansible хороша и в этом, поскольку изначально создавалась для проведения манипуляций сразу на нескольких серверах. Ansible имеет удивительно простую модель управления порядком действий.

Наконец, вы услышите, как люди говорят о подготовке и наполнении (provisioning) новых серверов. В контексте облачных услуг, таких как Amazon EC2, под *подготовкой и наполнением* подразумевается развертывание новых экземпляров виртуальной машины или облачных служб «программное обеспечение как услуга» (Software as a Service, SaaS). Ansible охватывает и эту область, предоставляя несколько модулей поддержки облаков, включая EC2, Azure[1], Digital Ocean, Google Compute Engine, Linode и Rackspace[2], а также любые облака, поддерживающие OpenStack API.

[1] Да, Azure поддерживает серверы на Linux.

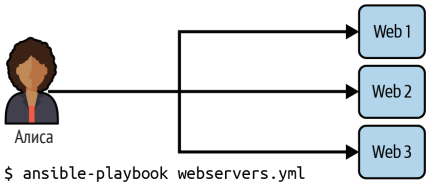
[2] Например, смотрите презентацию Джесса Китинга (Jesse Keating) «Using Ansible at Scale to Manage a Public Cloud» (https://oreil.ly/djLsk), ранее работавшего в Rackspace.

Несколько сбивает с толку использование термина *провайдер* (provisioner) в документации к утилите Vagrant, которую мы обсудим далее в этой главе, в отношении системы управления конфигурациями. Так, Vagrant называет Ansible своего рода провайдером там, где, как мне кажется, провайдером является сам Vagrant, поскольку именно он отвечает за запуск виртуальных машин.

Как работает Ansible

На рис. 1.1 показан простой пример использования Ansible. Пользователь, которого мы будем звать Алиса, использует Ansible для настройки трех веб-серверов NGINX, действующих под управлением Ubuntu. Она написала для Ansible сценарий *webservers.yml*. В терминологии Ansible сценарии называются *playbook*. Сценарий описывает, какие *хосты* (в Ansible они называются удаленными серверами) подлежат настройке и упорядоченный список *задач*, которые должны быть выполнены на этих хостах. В этом примере хосты носят имена web1, web2 и web3, и для настройки каждого из них требуется выполнить следующие задачи:

- установить Nginx;
- сгенерировать конфигурационные файлы для Nginx;
- скопировать сертификат безопасности;
- запустить Nginx.



Playbook: webservers.yml

```
---
- name: Configure webservers
  hosts: webservers
  become: True
  tasks:
    - name: Install nginx
      package: name=nginx
    - name: Install config file
      template:
        src: nginx.config.j2
        dest: /etc/nginx/nginx.conf
      notify: Restart nginx
  handlers:
    - name: Restart nginx
      service:name=nginx state=restarted
```

Рис. 1.1. Ansible выполняет сценарий настройки трех веб-серверов

В следующей главе мы обсудим, что в действительности входит в этот сценарий. Алиса запускает сценарий командой `ansible-playbook`. В примере сценарий называется *webservers.yml* и запускается вводом команды в терминале:

```
$ ansible-playbook webservers.yml
```

Ansible устанавливает параллельные SSH-соединения с хостами web1, web2 и web3 и выполняет первую задачу из списка на всех хостах одновременно. В этом примере первая задача – установка пакета NGINX, которая в сценарии выглядит так:

```
- name: Install nginx
  package:
    name: nginx
```

Выполняя ее, Ansible проделает следующие действия.

1. Сгенерирует сценарий на языке Python, который установит пакет NGINX.
2. Скопирует его на хосты web1, web2 и web3.
3. Запустит на хостах web1, web2 и web3.
4. Дождется, пока сценарий завершится на всех хостах.

Затем Ansible перейдет к следующей задаче в списке и повторит эти же четыре шага.

Важно отметить, что:

1) каждая задача выполняется на всех хостах одновременно;
2) Ansible ожидает завершения задачи на всех хостах, прежде чем приступить к выполнению следующей;
3) задачи выполняются в установленном вами порядке.

Какие преимущества дает Ansible?

Существует несколько систем управления конфигурациями с открытым исходным кодом, так почему мы выбираем Ansible? Ниже перечисляется 21 причина, подталкивающая нас к этому выбору. Но основными являются простота, широта возможностей и защищенность.

Простота

Разработчики стремились максимально упростить процесс установки и освоение Ansible.

Простота синтаксиса

Сценарии Ansible определяются в файлах формата YAML с использованием синтаксиса шаблонов Jinja2. Напомним, что в терминоло-

гии Ansible сценарии управления конфигурацией называются *playbook* (сценарий, пьеса). Фактически синтаксис сценариев Ansible основан на YAML, языке описания данных, который создавался специально, чтобы легко восприниматься человеком. В некотором роде YAML для JSON – то же, что Markdown для HTML.

Простота аудита

Сценарии Ansible легко поддаются исследованию – например, можно легко получить список всех действий и вовлеченных хостов. Для выполнения пробных прогонов мы часто используем команду `ansible-playbook --check`. Встроенная поддержка журналирования позволяет увидеть, кто, что и где делал. Механизм журналирования реализован как подключаемый модуль, а созданные им журналы легко получить с помощью сборщиков журналов.

Практически ничего не нужно устанавливать на удаленных хостах

Для управления серверами с помощью Ansible на серверах Linux должна быть установлена поддержка SSH и Python, а на серверах Windows должен быть включен WinRM. В Windows Ansible использует PowerShell вместо Python, что избавляет от необходимости предварительно устанавливать на хосте какого-либо агента или любое другое программное обеспечение.

На *управляющей машине* (той, что используется для управления удаленными машинами) должен быть установлен Python версии 3.8 или выше. В зависимости от ресурсов, которыми требуется управлять с помощью Ansible, может потребоваться установить дополнительные сторонние библиотеки. Загляните в документацию, чтобы узнать, имеются ли у модуля особые требования.

Возможность масштабирования вниз

Да, Ansible можно использовать для управления сотнями и даже тысячами узлов. Но что нас особенно зацепило, так это его масштабируемость вниз. Ansible можно запустить на очень скромном оборудовании, таком как Raspberry Pi или старом ПК, и даже использовать для управления единственным узлом – нужно лишь написать один сценарий. Ansible подтверждает принцип Алана Кея: «Простое должно оставаться простым, а сложное – возможным».

Простота распространения

Мы не думаем, что вам понадобится повторно использовать сценарии Ansible в разных контекстах. В главе 7 мы обсудим роли, предлагающие возможность организации сценариев, и Ansible Galaxy, онлайн-репози-

торий для хранения этих ролей.

Основной единицей повторного использования в сообществе Ansible в настоящее время является *коллекция*. Вы можете упаковать свои модули, плагины, библиотеки, роли и даже сборники игр в коллекцию и поделиться ею с другими через Ansible Galaxy. Также можно организовать распространение внутри организации с помощью инструмента Automation Hub, входящего в состав Ansible Tower. Роли могут использоваться совместно как отдельные репозитории.

На практике, однако, каждая организация настраивает свои серверы немного не так, как другие, поэтому лучше писать свои сборники сценариев для своей организации, а не пытаться повторно использовать те, что находятся в общем доступе. Мы считаем, что основная ценность изучения чужих схем заключается в возможности увидеть, как все работает, если только вы не работаете с конкретным продуктом, производитель которого является сертифицированным партнером или членом сообщества Ansible.

Простота абстракций

Ansible работает с простыми *абстракциями* системных ресурсов, таких как файлы, каталоги, пользователи, группы, службы, пакеты и веб-сервисы.

Для сравнения давайте посмотрим, как настроить каталог в командной оболочке. Для этого используются три команды:

```
mkdir -p /etc/skel/.ssh
chown root:root /etc/skel/.ssh
chmod go-wrx /etc/skel/.ssh
```

Ansible, в свою очередь, предлагает абстракцию – модуль `file`, с помощью которого определяются параметры желаемого состояния. Следующее единственное действие дает тот же эффект, что и три команды выше:

```
- name: Ensure .ssh directory in user skeleton
  file:
    path: /etc/skel/.ssh
    mode: '0700'
    owner: root
    group: root
    state: directory
```

Этот слой абстракции позволяет использовать одни и те же сценарии для управления конфигурациями серверов с Linux. Например, вместо использования конкретного диспетчера пакетов, такого как `dnf`, `yum` или `apt`, Ansible предлагает абстракцию «пакет» (просто имейте в виду, что

имена пакетов могут отличаться). Но при желании можно также использовать системные абстракции.

Если вы действительно этого хотите, то можете написать свои сценарии Ansible для выполнения различных действий в разных операционных системах на удаленных серверах. Но Бас, один из авторов этой книги, старается избегать этого по возможности, предпочитая писать сценарии для реально используемых систем.

Выполнение задач сверху вниз

В книгах по управлению конфигурациями часто упоминается идея *конвергенции* (сходимости), или *последовательного приведения к конечному состоянию*, которая нередко ассоциируется с именем Марка Бургесса (Mark Burgess) и его системой управления конфигурациями CFEngine. Если система управления конфигурациями конвергентна, то она может многократно выполнять управляющие воздействия, с каждым разом приводя сервер все ближе к желаемому состоянию.

Идея конвергенции неприменима к Ansible из-за отсутствия понятия многоэтапных воздействий на конфигурацию серверов. Модули Ansible устроены так, что единственный запуск сценария Ansible сразу приводит каждый сервер в желаемое состояние.

Широта возможностей

Ansible помогает значительно повысить производительность в нескольких областях управления системами. Абстракции высокого уровня, предоставляемые Ansible (например, роли), дают возможность устанавливать и настраивать программное обеспечение быстрее и потенциально безопаснее.

Встроенные модули

Ansible можно использовать для выполнения произвольных команд оболочки на удаленных серверах, но по-настоящему сильной его стороной является набор встроенных модулей. Модули необходимы для выполнения таких задач, как установка пакетов приложений, перезапуск службы или копирование конфигурационных файлов.

Как мы увидим позже, модули Ansible несут *декларативную* функцию и используются для описания требуемого состояния серверов. Например, вы могли бы вызвать модуль `user`, чтобы убедиться в существовании учетной записи `deploy` в группе `web`:

```
- name: Ensure deploy user exists
  user:
    name: deploy
    group: web
```

Использование технологии принудительной настройки

Некоторые системы управления конфигурациями, использующие агентов, такие как Chef и Puppet, по умолчанию основаны на технологии *добровольной настройки*. Агенты, установленные на серверах, периодически подключаются к центральной службе и читают информацию о конфигурации. Управление изменениями конфигурации серверов в этом случае выглядит так:

4) вы: вносите изменения в сценарий управления конфигурациями,
5) вы: передаете изменения центральной службе,
6) агент на сервере: периодически включается по таймеру,
7) агент на сервере: подключается к центральной службе,
8) агент на сервере: читает новые сценарии управления конфигурациями,
9) агент на сервере: запускает полученные сценарии локально, обновляя состояние сервера.

Ansible, напротив, по умолчанию использует технологию *принудительной настройки*. Внесение изменений выглядит так:

1) вы: вносите изменения в сценарий,
2) вы: запускаете новый сценарий,
3) Ansible: подключается к серверам и запускает модули, обновляя состояние серверов.

Как только вы запустите команду `ansible-playbook`, Ansible подключится к удаленным серверам и выполнит всю работу; это снижает риск выхода из строя случайных серверов, когда запланированные на них задачи не могут успешно изменить их состояние. Принудительная настройка дает важное преимущество – вы контролируете время обновления серверов. Вам не приходится ждать. Каждое действие в сценарии может быть нацелено на один или группу серверов. Вы можете выполнять больше операций автоматически, не выполняя вход на серверы вручную.

Многоуровневая оркестрация

Технология принудительной настройки позволяет также реализовать с помощью Ansible *многоуровневую оркестрацию* – управление отдельными группами компьютеров для выполнения различных операций, таких как обновление ПО. Вы можете организовать управление системами мониторинга, балансировщиками нагрузки, базами данных и веб-серверами с помощью конкретных инструкций и обеспечить их согласованную работу. Это очень сложно сделать с системой, основанной на технологии добровольной настройки.

Отсутствие ведущего узла

Сторонники добровольной настройки утверждают, что их подход лучше масштабируется на большое число серверов и удобнее, когда новые серверы могут появиться в любой момент. Однако централизованная система управления конфигурацией может испытывать значительную нагрузку, когда тысячи агентов одновременно попытаются извлечь свою конфигурацию, особенно если им требуется выполнить несколько циклов для конвергенции. Для сравнения: Ansible официально поддерживает особый режим, называемый `ansible-pull`, в котором сценарии извлекаются из репозитория, такого как GitHub. Ansible не нуждается в ведущем узле, но при желании вы можете использовать централизованную систему для запуска сценариев.

Поддержка плагинов

Значительная часть функциональности Ansible реализована в виде подключаемых модулей – плагинов, из которых наиболее часто используются плагины Lookup и Filter. Плагины расширяют базовые возможности Ansible логикой и функциями, доступными для всех модулей. Модули вводят в язык Ansible новые «глаголы». Вы тоже можете писать свои плагины (глава 10) и модули (глава 12) на Python .

Ansible можно интегрировать с другими продуктами. Примерами успешной интеграции могут служить Kubernetes и Ansible Tower. Ansible Runner – это «инструмент и библиотека для Python, помогающая организовать взаимодействие с Ansible напрямую или в составе другой системы, например через интерфейс образа контейнера, как автономный инструмент или как модуль на Python, который можно импортировать»[1].

С помощью библиотеки *ansible-runner* можно запустить сценарий Ansible из программы на Python:

```
#!/usr/bin/env python3
import ansible_runner

r = ansible_runner.run(private_data_dir='./playbooks', playbook='playbook.yml')

print("{}: {}".format(r.status, r.rc))
print("Final status:")
print(r.stats)
```

Поддержка решения широкого круга задач

Модули Ansible предназначены для решения широкого круга задач системного администрирования. В списке ниже перечислены катего-

[1] Цитата из документации к Ansible Runner (https://oreil.ly/sZwPY).

рии доступных модулей. Эти ссылки ведут в список модулей (**https://oreil.ly/0Xel7**) в документации:

- Cloud (**https://oreil.ly/0xeNu**);
- Files (**https://oreil.ly/3cq87**);
- Monitoring (**https://oreil.ly/z6dde**);
- Source Control (**https://oreil.ly/WEMHZ**);
- Clustering (**https://oreil.ly/b31cn**);
- Identity (**https://oreil.ly/39yJA**);
- Net Tools (**https://oreil.ly/Pb137**);
- Storage (**https://oreil.ly/IZBGX**);
- Commands (**https://oreil.ly/wyyJZ**);
- Infrastructure (**https://oreil.ly/XhW90**);
- Network (**https://oreil.ly/UFHZo**);
- System (**https://oreil.ly/mn569**);
- Crypto (**https://oreil.ly/puZGg**);
- Inventory (**https://oreil.ly/zBvdF**);
- Notification (**https://oreil.ly/ulrdH**);
- Utilities (**https://oreil.ly/veSG4**);
- Database (**https://oreil.ly/iEv9l**);
- Messaging (**https://oreil.ly/aTOvP**);
- Packaging (**https://oreil.ly/71GLO**);
- Windows (**https://oreil.ly/c8NwK**).

Настоящая масштабируемость

Крупные предприятия успешно используют Ansible для настройки десятков тысяч узлов и отлично поддерживают окружения, в которых серверы появляются и исчезают динамически. Организации с сотнями групп разработчиков программного обеспечения обычно используют AWX или комбинацию Ansible Tower и Automation Hub для аудита и защиты с контролем доступа на основе ролей.

Вас волнует масштабируемость SSH? Ansible использует мультиплексирование SSH для оптимизации производительности и реальные примеры управления тысячами узлов с помощью Ansible (глава 12).

Защищенность

Автоматизация с помощью Ansible помогает повысить защищенность системы до базовых уровней безопасности и стандартов соответствия.

Самодокументирующийся код

Авторам книги нравится думать о сценариях Ansible как о выполняемой документации. Они сродни файлам README, которые описывают действия, необходимые для развертывания программного обеспечения, но, в отличие от них, сценарии всегда содержат актуальные инструкции, поскольку сами являются выполняемым кодом. Эксперты могут создавать сценарии, отражающие передовой опыт, а новички – использовать их как учебники и пребывать в уверенности, что получат хороший результат.

Воспроизводимость

Если всю свою систему вы настроите с помощью Ansible, то она пройдет то, что Стив Трауготт (Steve Traugott) называет «тестированием десятым этажом» (**https://oreil.ly/AMf1S**): «Могу ли я взять случайную машину, для которой никогда не выполнялось резервного копирования, выкинуть ее из окна десятого этажа и при этом не потерять работу системного администратора?»

Эквивалентность создаваемых окружений

Ansible поддерживает определенный способ организации контента, помогающий определить конфигурацию на надлежащем уровне. Вы с легкостью сможете определить настройки для различных окружений: разработки, тестирования, обкатки и промышленной эксплуатации. Окружение обкатки обычно делается максимально похожим на промышленное окружение, чтобы разработчики могли выявить любые проблемы до того, как изменения попадут в промышленное окружение.

Шифрование переменных

При необходимости хранить конфиденциальные данные, такие как пароли или токены, можно использовать эффективный инструмент `ansible-vault`. Мы используем его для шифрования переменных в Git. Более подробно этот вопрос обсуждается в главе 8.

Защищенный транспорт

Ansible просто использует Secure Shell (SSH) для Linux и WinRM для Windows. Обычно мы защищаем и укрепляем эти широко используемые протоколы управления системами с помощью защищенных настроек конфигурации и брандмауэра.

Если вы предпочитаете модель, основанную на приемах добровольной настройки, то для вас Ansible официально поддерживает особый режим, называемый `ansible-pull`. В этой книге не раскрываются особенности этого режима, но вы можете узнать больше об этом из официальной документации (**https://docs.ansible.com/**).

Идемпотентность

Модули также являются идемпотентными[1]. Идемпотентность – замечательное свойство и означает, что сценарий Ansible можно применить к одному и тому же серверу много раз без всякого ущерба для конфигурации последнего. Давайте рассмотрим пример, когда нам нужно создать пользователя `deploy`:

```
- name: Ensure deploy user exists
  user:
    name: deploy
    group: web
```

Если пользователя `deploy` не существует, то Ansible создаст его. Если он существует, то Ansible просто перейдет к следующему шагу. То есть сценарии Ansible можно запускать на сервере много раз. Это важное отличие от сценариев командной оболочки, потому что повторный запуск таких сценариев может привести к незапланированным – и хорошо, если безобидным – последствиям[2].

Отсутствие демонов

В Ansible нет агента, прослушивающего некоторый порт. Поэтому у злоумышленников нет цели для атаки на Ansible. (Однако существуют другие цели для атаки – элементы цепочки доставки программного обеспечения, такого как библиотеки Python и другие импортируемые компоненты.)

Связь между Ansible и Ansible, Inc.

Название *Ansible* относится как к программному обеспечению, так и к компании, управляющей проектом. Майкл ДеХаан, создатель программного обеспечения Ansible, является бывшим техническим директором компании Ansible. Во избежание путаницы хочу уточнить, что для обозначения продукта я использую Ansible, а компании – Ansible, Inc.

Ansible, Inc. проводит обучение и предоставляет консультационные услуги по Ansible, а также собственной веб-системе управления Ansible Tower, о которой рассказывается в главе 19. В октябре 2015 года Red Hat купила Ansible Inc., а в 2019 году IBM купила Red Hat.

---

[1] Идемпоте́нтность – свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при одинарном. – *Прим. перев.*

[2] Если вам интересно, что думает автор Ansible об идемпотентности и конвергенции, прочтите публикацию Майкла ДеХаана «Idempotence, convergence, and other silly fancy words we use too often» («Идемпотентность, конвергенция и другие причудливые слова, которые мы используем слишком часто») на странице группы Ansible Project (https://oreil.ly/pNSNr).

Не слишком ли проста система Ansible?

В период работы над книгой редактор сказал Лорин, что «некоторые специалисты, использующие систему управления конфигурациями XYZ, называют Ansible циклом for по сценариям, запускаемым через SSH». Планируя переход с другой системы управления конфигурациями на Ansible, действительно могут возникнуть сомнения в его эффективности.

Однако, как скоро будет показано, Ansible имеет гораздо более широкие возможности, чем сценарии командной оболочки. Как уже упоминалось, модули Ansible гарантируют идемпотентность, Ansible имеет превосходную поддержку шаблонов и переменных с разными областями видимости. Любой, кто считает, что суть Ansible заключается в работе со сценариями командной оболочки, никогда не занимался поддержкой нетривиальных программ на языке оболочки. Если есть выбор, я предпочту Ansible сценариям командной оболочки.

Что я должен знать?

Для эффективной работы с Ansible необходимо знать основы администрирования операционной системы Unix/Linux. Ansible позволяет автоматизировать процессы, но не является волшебным инструментом, способным выполнять операции, которые вы не знаете, как выполнить.

Читатели данной книги должны быть знакомы по крайней мере с одним из дистрибутивов Linux (Ubuntu, RHEL/CentOS, SUSE и пр.) и понимать, как:

- подключиться к удаленной машине через SSH;
- работать в командной строке Bash (каналы и перенаправление);
- устанавливать пакеты приложений;
- использовать команду *sudo*;
- проверять и устанавливать разрешения для файлов;
- запускать и останавливать службы;
- устанавливать переменные окружения;
- писать сценарии (на любом языке).

Если все это вам известно, то можете смело приступать к работе с Ansible.

Я не предполагаю, что вы знаете какой-то определенный язык программирования. Например, вам не нужно знать Python, если вы не собираетесь самостоятельно писать модули.

О чем не рассказывается в этой книге

Эта книга не является исчерпывающим руководством по работе с Ansible. Она позволяет подготовиться к использованию Ansible в кратчайшие сроки и дает описание некоторых задач, которые недостаточно полно описываются в официальной документации.

Книга не описывает использования официальных модулей Ansible. Их более 3500, и они достаточно хорошо представлены в официальной документации. Для просмотра справочной документации и списка модулей, упоминавшихся выше, можно использовать инструмент командной строки `ansible-doc`.

Глава 8 охватывает только основные возможности механизма шаблонов Jinja2, главным образом потому, что авторы используют только самые основные функции Jinja2 при работе с Ansible. Для более глубокого знакомства Jinja2 обращайтесь к официальной документации Jinja2 (**https://oreil.ly/LAXa7**).

Книга не дает детального описания некоторых возможностей Ansible, используемых в основном для поддержки ранних версий Linux.

Наконец, некоторые особенности Ansible мы не будем рассматривать, просто чтобы не увеличивать и без того немалый объем книги. Для знакомства с этими особенностями обращайтесь к официальной документации (**https://docs.ansible.com/**).

Поехали!

В этой вводной главе мы в общих чертах рассмотрели основные понятия Ansible, в том числе особенности взаимодействий с удаленными серверами и отличия от других инструментов управления конфигурациями. В следующих главах обсуждается практическое использование Ansible.

Глава 2

Установка и настройка

Система Ansible написана на Python и предназначена для использования в операционных системах Linux/macOS/BSD. С другой стороны, она может управлять конфигурацией всех типов операционных систем и, как правило, не требует ничего устанавливать в целевые системы при условии, что в системах Linux/macOS/BSD установлен Python, а в Windows установлен PowerShell. Обычно многие устанавливают Ansible на свою рабочую станцию, на которой должен быть установлен Python 3.8.

Установка Ansible

В настоящее время все основные дистрибутивы Linux включают пакет Ansible. Поэтому, использующие Linux смогут установить Ansible, используя встроенный диспетчер пакетов. Но имейте в виду, что это может быть не самая последняя версия Ansible. Если вы используете macOS, то я рекомендую использовать для установки Ansible замечательный диспетчер пакетов Homebrew:

```
$ brew install ansible
```

На любом компьютере с Unix/Linux/macOS можно также установить Ansible с помощью одного из диспетчеров пакетов Python и с его же помощью добавить инструменты и библиотеки на Python, которые могут вам пригодиться, при условии, что вы добавите `~/.local/bin` в список путей в переменной окружения `PATH`. Если вы предпочтете Ansible Tower или AWX, то установите соответствующую версию `ansible-core`.

```
$ pip3 install --user ansible==2.9.27
```

При установке версии выше 2.10 (например, 5.9.0), pip3 также установит все стандартные коллекции, следуя принципу «все включено».

При работе с несколькими проектами удобно установить Ansible в виртуальное окружение Python (*virtualenv*). Это избавит вас от конфликтов с системной установкой Python и от загромождения пользовательского окружения. Используя модуль Python venv и pip3, можно установить в каждый проект только то, что действительно необходимо:

```
$ python3 -m venv .venv --prompt A
$ source .venv/bin/activate
(A)
```

После активации виртуального окружения приглашение командной оболочки сменится на (A) для напоминания. Выйти из виртуального окружения можно командой **deactivate**.

Возможность запуска Ansible в Windows официально не поддерживается, но вы сможете управлять системами Windows удаленно с помощью Ansible, используя PowerShell поверх WinRM[1].

Запустить Ansible на хосте с Windows (т. е. использовать машину с Windows в роли управляющей машины) все же возможно, но при этом запускать Ansible следует в подсистеме Windows для Linux (WSL2) . На практике это означает, что вы будете запускать Ubuntu рядом с Windows на одном компьютере. WSL2 – это подсистема, которую можно активировать в Windows 10 Home Edition (и более поздних версиях). Она не поддерживается Ansible и поэтому не должна использоваться для управления промышленными системами. Для установки Ansible в WSL2 выполните следующие команды:

```
sudo apt-get update
sudo apt-get install python3-pip git libffi-dev
libssl-dev -y
pip3 install --user ansible
```

Дополнительные зависимости

Плагины и модули Ansible могут потребовать установить дополнительные библиотеки Python. Например, для управления системами Windows и Docker нужно установить следующие две библиотеки для Python:

```
(A) pip3 install pywinrm docker
```

[1] Чтобы узнать, почему официально не поддерживается возможность запуска Ansible в Windows, прочитайте статью «Why No Ansible Controller for Windows?» (https://oreil.ly/xrtnD) в блоге Мэтта Дэвиса (Matt Davis).

В некотором смысле виртуальное окружение Python было предшественником контейнеров: оно позволяет изолировать библиотеки и избежать «ада зависимостей».

Запуск Ansible в контейнерах

В комплект Ansible входит `ansible-builder` – инструмент, помогающий создать среду выполнения и осуществлять запуск Ansible в контейнере для автоматизации узкоспециализированных рабочих процессов. Он основан на структуре каталогов `ansible-runner`. Это сложная тема, и она выходит за рамки данной главы. Однако мы вернемся к ней в главе 23 .

Версия Ansible для разработчиков

При наличии желания поэкспериментировать с последними возможностями Ansible вы можете получить новейшую версию из ветки разработки на GitHub :

```
$ python3 -m venv .venv --prompt S
$ source .venv/bin/activate
(S) python3 -m pip install --upgrade pip
(S) pip3 install wheel
(S) git clone https://github.com/ansible/ansible.git --recursive
(S) pip3 install -r ansible/requirements.txt
```

Однако, используя версию Ansible для разработчиков, вам придется каждый раз запускать следующие команды, чтобы настроить переменные окружения, включая переменную `PATH`, чтобы ваша командная оболочка знала, где находятся программы `ansible` и `ansible-playbook`:

```
(S) cd ./ansible
(S) source ./hacking/env-setup
```

Подготовка сервера для экспериментов

Для выполнения примеров, приведенных в книге, вам необходимо иметь SSH-доступ и права пользователя *root* на сервере Linux. К счастью, сегодня легко получить недорогой доступ к виртуальной машине Linux в общедоступных облачных службах.

Использование Vagrant для подготовки сервера

Если вы предпочитаете не тратиться на облачные услуги, то я предложил бы установить Vagrant – отличный инструмент с открытым исходным кодом для управления виртуальными машинами. С его помощью можно запустить виртуальную машину с Linux на ноутбуке, которая и послужит вам сервером для экспериментов.

Vagrant – отличное окружение для тестирования сценариев Ansible. Мы часто используем Vagrant при разработке наших собственных сценариев Ansible и поэтому будем использовать его на протяжении всей книги. Но тестирование сценариев управления конфигурацией не единственное предназначение Vagrant; изначально этот инструмент разрабатывался для создания воспроизводимых окружений разработки. Если вам доводилось присоединяться к новой команде разработчиков программного обеспечения и тратить несколько дней на выяснение того, какое программное обеспечение нужно установить на свой ноутбук, чтобы запустить версию для разработчиков, то вы наверняка знаете, насколько болезненным может быть этот процесс. Vagrant создавался для облегчения этой боли. Сценарии Ansible – отличный способ настроить машину Vagrant, чтобы новички в вашей команде могли приступить к работе в первый же день.

Vagrant требует установки гипервизора, такого как VirtualBox. Скачайте VirtualBox, а затем Vagrant. Vagrant имеет встроенную поддержку Ansible, которой мы и воспользуемся в этой главе для настройки машин Vagrant.

Рекомендую создать отдельный каталог для сценариев Ansible и прочих файлов. В следующем примере я создал такой каталог с именем *playbooks*. Структура каталогов важна для Ansible: если разместить файлы в правильных местах, то мозаика будет складываться из отдельных кусочков без всяких проблем.

Выполните следующие команды, чтобы создать конфигурационный файл Vagrant (Vagrantfile) для 64-битого образа виртуальной машины Ubuntu/Focal и запустить его:

```
$ mkdir playbooks
$ cd playbooks
$ vagrant init ubuntu/focal64
$ vagrant up
```

При первом запуске команда `vagrant up` загрузит файл образа виртуальной машины. На это может потребоваться некоторое время в зависимости от качества соединения с интернетом.

В случае успеха вы увидите, как в окне терминала побегут следующие строки:

```
$ vagrant up default
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/focal64'...
==> default: Matching MAC address for NAT networking...
```

```
==> default: Checking if box 'ubuntu/default64' version is up to date...
==> default: Setting the name of the VM: default
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Setting hostname...
==> default: Configuring and enabling network interfaces...
==> default: Mounting shared folders...
    default: /vagrant => C:/Users/basme/ansiblebook/ch02/playbooks
```

Теперь можно попробовать зайти по SSH на вашу новую виртуальную машину Ubuntu 20.04, выполнив следующую команду:

```
$ vagrant ssh
```

Если все прошло благополучно, то вы увидите приветствие на экране:

```
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-72-generic x86_64)
  * Documentation: https://help.ubuntu.com
  * Management: https://landscape.canonical.com
  * Support: https://ubuntu.com/advantage
  System information as of Sun Apr 18 14:53:23 UTC 2021
  System load: 0.08 Processes: 118
  Usage of /: 3.2% of 38.71GB Users logged in: 0
  Memory usage: 20% IPv4 address for enp0s3: 10.0.2.15
  Swap usage: 0%

1 update can be installed immediately.
0 of these updates are security updates.
To see these additional updates run: apt list --upgradable

vagrant@ubuntu-focal:~$
```

Выполнив вход с помощью команды `vagrant ssh`, вы сможете взаимодействовать с командной оболочкой Bash, но Ansible подключается к виртуальной машине с помощью обычного клиента SSH. Дайте Vagrant команду вывести конфигурацию SSH:

```
$ vagrant ssh-config
```

Вот как выглядит вывод этой команды на компьютере Баса с Windows:

```
Host default
   HostName 127.0.0.1
   User vagrant
   Port 2222
   UserKnownHostsFile /dev/null
   StrictHostKeyChecking no
   PasswordAuthentication no
   IdentityFile C:/Users/basme/.vagrant.d/insecure_private_key
IdentitiesOnly yes
LogLevel FATAL
```

Вот самые важные строки:

```
HostName 127.0.0.1
  User vagrant
  Port 2222
  IdentityFile C:/Users/basme/.vagrant.d/insecure_private_key
```

Начиная с версии 1.7, в Vagrant изменился порядок работы с закрытыми SSH-ключами. Начиная с этой версии, Vagrant генерирует новый закрытый ключ для каждой машины. Более ранние версии использовали один и тот же ключ, который по умолчанию хранился в каталоге *$HOME/.vagrant.d/insecure_private_key*. Примеры в этой книге основаны на Vagrant 2.2.

У вас строки должны выглядеть похоже, за исключением места хранения файла идентификации.

Проверьте, сможете ли вы запустить новый SSH-сеанс из командной строки, используя эту информацию. Команда SSH также правильно работает, если ей передать относительный путь при запуске в каталоге *playbooks*:

```
$ ssh vagrant@127.0.0.1 -p 2222 \
  -i .vagrant/machines/default/virtualbox/private_key
```

Вы должны увидеть приглашение к вводу в Ubuntu. Введите `exit`, чтобы завершить SSH-сеанс.

Передача информации о сервере в Ansible

Ansible может управлять только известными ей серверами. Передать информацию о серверах в Ansible можно в файле *реестра* (inventory). Мы обычно создаем каталог *inventory*, в который сохраняем эту информацию.

```
$ mkdir inventory
```

Каждому серверу должно быть присвоено имя для идентификации в Ansible. С этой целью можно использовать имя хоста или выбрать другой псевдоним. С именем также должны определяться дополнительные параметры подключения. Присвоим нашему серверу псевдоним *testserver*.

Создайте в каталоге *inventory* текстовый файл. Если в роли тестового сервера вы используете виртуальную машину Vagrant, то дайте файлу имя *vagrant.ini*, если вы используете машины Amazon EC2, то назовите файл *ec2.ini*. Имейте в виду, что, несмотря на расширение *.ini* в именах этих файлов реестра, они не следуют правилам оформления INI-файлов, определенным в Microsoft. В частности, INI-файлы всегда состоят из пар *ключ/значение*, что не всегда верно для файлов реестра.

Файлы *.ini* будут служить реестром для Ansible. Они определяют инфраструктуру для управления в группах, обозначенных именами в квадратных скобках. Если вы используете Vagrant, то содержимое вашего файла должно выглядеть как в примере 2.1. Группа `[webservers]` включает один хост: `testserver`. Здесь можно заметить один из недостатков использования Vagrant: необходимость передачи в Ansible дополнительных данных – переменных *vars*, – определяющих параметры подключения к группе. В реальных окружениях эти переменные обычно не нужны. С другой стороны, если вы используете окружения для обкатки с разными параметрами безопасности, то реестр – отличное место для определения этих различий.

Пример 2.1. *inventory/vagrant.ini*

```
[webservers]
testserver ansible_port=2222

[webservers:vars]
ansible_host=127.0.0.1
ansible_user=vagrant
ansible_private_key_file=.vagrant/machines/default/virtualbox/private_key
```

Если предположить, что у вас есть Ubuntu-машина в облаке Amazon EC2 с именем хоста *ec2-203-0-113-120.compute-1.amazonaws.com*, то содержимое файла реестра будет выглядеть так:

```
[webservers]
testserver ansible_host=ec2-203-0-113-120.compute- 1.amazonaws.com

[webservers:vars]
ansible_user=ec2-user
ansible_private_key_file=/path/to/keyfile.pem
```

Ansible поддерживает программу `ssh-agent`, поэтому нет необходимости явно указывать файлы SSH-ключей в реестре. Если вы входите в систему со своим собственным идентификатором пользователя, то вам тоже не придется указывать их.

Чтобы проверить способность Ansible подключиться к серверу, используем утилиту командной строки `ansible`. Мы будем пользоваться ею лишь изредка, в основном для решения специфических задач.

Попросим Ansible установить соединение с сервером *testserver*, указанным в файле реестра *vagrant.ini*, и вызвать модуль `ping`:

```
$ ansible testserver -i inventory/vagrant.ini -m ping
```

Если на локальном SSH-клиенте включена проверка ключей хоста, вы увидите нечто похожее на первую попытку Ansible подключиться к серверу:

```
The authenticity of host '[127.0.0.1]:2222 ([127.0.0.1]:2222)' can't be
established.
ED25519 key fingerprint is SHA256:6l2Lg8/EBqMFstGNPqFtLychVkxRxqdvRhvLlv/Tj1E.
Are you sure you want to continue connecting (yes/no)?
```

Просто введите `yes`.

В случае успеха появится следующий результат:

```
testserver | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

Если Ansible сообщит об ошибке, то добавьте в команду флаг `-vvvv`, чтобы получить больше информации об ошибке:

```
$ ansible testserver -i inventory/vagrant.ini -m ping -vvvv
```

Мы видим, что команда выполнилась успешно. Часть ответа `"changed": false` говорит о том, что выполнение модуля не изменило состояния сервера. Текст `"ping":"pong"` является характерной особенностью модуля `ping`.

Модуль `ping` не производит никаких изменений. Он лишь проверяет способность Ansible начать SSH-сеанс с сервером и может пригодиться в начале большого сценария.

Упрощение задачи с помощью файла ansible.cfg

Нам пришлось ввести много текста в файл реестра, чтобы проверить возможность подключения к тестовому серверу. К счастью, Ansible поддерживает несколько способов передачи такой информации, и мы не обязаны группировать ее в одном месте. Сейчас мы воспользуемся одним из таких способов – файлом *ansible.cfg* – и определим в нем некоторые настройки по умолчанию, чтобы потом нам не пришлось набирать так много текста.

Где лучше хранить файл ansible.cfg?

Ansible будет искать файл *ansible.cfg* в следующих местоположениях в указанном порядке:

- файл, указанный в переменной окружения ANSIBLE_CONFIG;
- *./ansible.cfg* (*ansible.cfg* в текущем каталоге);
- *~/.ansible.cfg* (*.ansible.cfg* в вашем домашнем каталоге);
- */etc/ansible/ansible.cfg* (Linux) или */usr/local/etc/ansible/ansible.cfg* (BSD).

Я обычно храню *ansible.cfg* в текущем каталоге вместе со сценариями. Это позволяет хранить его в том же репозитории, где хранятся мои сценарии, а также дает возможность создавать конфигурационные файлы отдельно для каждого проекта.

В примере 2.2 показан файл *ansible.cfg*, определяющий местоположение файла реестра (*inventory*) и параметры, влияющие на работу Ansible, например на форматирование вывода.

Учетная запись для входа и соответствующий закрытый ключ SSH могут зависеть от используемого реестра, поэтому блок *vars* с параметрами подключения лучше добавлять в файл реестра, а не в файл *ansible.cfg*. Однако добавление имени файла закрытого ключа в файл *ansible.cfg* или в файлы реестра сделает конфигурацию менее гибкой и уменьшит возможность совместного использования вашего проекта несколькими пользователями. Альтернативное решение – неявное использование конфигурации SSH.

В нашем примере конфигурации в *ansible.cfg* проверка SSH-ключей хоста отключена. Это удобно при работе с Vagrant, потому что иначе потребовалось бы вносить изменения в файл *~/.ssh/known_hosts* каждый раз, когда удаляется имеющийся или создается новый Vagrant-сервер. Одна-

ко отключение проверки ключей для серверов в сети несет определенные риски.

Пример 2.2. *ansible.cfg*

```
[defaults]
inventory = inventory/vagrant.ini
host_key_checking = False
stdout_callback = yaml
callback_enabled = timer
```

Ansible и система управления версиями

Ansible по умолчанию хранит реестр в файле */etc/ansible/hosts*. Хранение реестра в одном каталоге со сценариями и другими артефактами дает возможность использовать конкретный реестр для каждого проекта, а не только глобальный. Но если отделить проект от реестра, то его будет проще повторно использовать на машинах, принадлежащих другим.

Хотя работа с системами управления версиями не затрагивается в этой книге, я настоятельно рекомендую использовать для управления сценариями систему, подобную Git. Если вы разработчик программного обеспечения, то наверняка знакомы с системами управления версиями. Если вы системный администратор и прежде не пользовались ими, тогда это хороший повод начать знакомство.

С настройками по умолчанию можно запускать Ansible без ключа `-i` с именем хоста:

```
$ ansible testserver -m ping
```

Нам нравится использовать инструмент командной строки `ansible` для запуска произвольных команд на удаленных серверах. Произвольные команды также можно выполнять с помощью модуля `command`. При запуске модуля необходимо указать аргумент `-a` с запускаемой командой.

Например, вот как можно проверить время работы сервера с момента последнего запуска:

```
$ ansible testserver -m command -a uptime
```

Результат должен выглядеть примерно так:

```
testserver | CHANGED | rc=0 >>
  10:37:28 up 2 days, 14:11, 1 user, load average: 0.00, 0.00, 0.00
```

Модуль `command` используется настолько часто, что сделан модулем по умолчанию, т. е. его имя можно опустить в команде:

```
$ ansible testserver -a uptime
```

Если команда в аргументе `-a` содержит пробелы, то ее необходимо заключить в кавычки, чтобы командная оболочка передала Ansible всю строку как единый аргумент. Например, вот как выглядит извлечение нескольких последних строк из журнала */var/log/dmesg*:

```
$ ansible testserver -a "tail /var/log/dmesg"
```

Вывод, возвращаемый машиной Vagrant, выглядит примерно так:

```
testserver | CHANGED | rc=0 >>
[ 9.940870] kernel: 14:48:17.642147 main   VBoxService 6.1.16_Ubuntu r140961
(verbosity: 0) linux.amd64 (Dec 17 2020 22:06:23) release log
                    14:48:17.642148 main   Log opened 2021-04-18T14:48:17.642143000Z
[ 9.941331] kernel: 14:48:17.642623 main   OS Product: Linux
[ 9.941419] kernel: 14:48:17.642718 main   OS Release: 5.4.0-72-generic
[ 9.941506] kernel: 14:48:17.642805 main   OS Version: #80-Ubuntu SMP Mon Apr 12
17:35:00 UTC 2021
[ 9.941602] kernel: 14:48:17.642895 main   Executable: /usr/sbin/VBoxService
                    14:48:17.642896 main   Process ID: 751
                    14:48:17.642896 main   Package type: LINUX_64BITS_GENERIC
                    (OSE)
[ 9.942730] kernel: 14:48:17.644030 main   6.1.16_Ubuntu r140961 started.
Verbose level = 0
[ 9.943491] kernel: 14:48:17.644783 main   vbglR3GuestCtrlDetectPeekGetCancelSupport:
Supported (#1)
```

Чтобы выполнить команду с привилегиями root, нужно передать параметр `-b` или `--become`. В этом случае Ansible выполнит команду *от лица* (*become*) пользователя root. В Unix/Linux для этого обычно используется такой инструмент, как *sudo*, который необходимо настроить. В примерах Vagrant в этой книге это было сделано автоматически.

Например, для доступа к */var/log/syslog* требуются привилегии root:

```
$ ansible testserver -b -a "tail /var/log/syslog"
Результат будет выглядеть примерно так:
testserver | CHANGED | rc=0 >>
Apr 23 10:39:41 ubuntu-focal multipathd[471]: sdb: failed to get udev uid:
Invalid argument
Apr 23 10:39:41 ubuntu-focal multipathd[471]: sdb: failed to get sysfs uid: No
data available
Apr 23 10:39:41 ubuntu-focal multipathd[471]: sdb: failed to get sgio uid: No
data available
Apr 23 10:39:42 ubuntu-focal multipathd[471]: sda: add missing path
Apr 23 10:39:42 ubuntu-focal multipathd[471]: sda: failed to get udev uid:
Invalid argument
Apr 23 10:39:42 ubuntu-focal multipathd[471]: sda: failed to get sysfs uid: No
data available
Apr 23 10:39:42 ubuntu-focal multipathd[471]: sda: failed to get sgio uid: No
```

```
data available
Apr 23 10:39:43 ubuntu-focal systemd[1]: session-95.scope: Succeeded.
Apr 23 10:39:44 ubuntu-focal systemd[1]: Started Session 97 of user vagrant.
Apr 23 10:39:44 ubuntu-focal python3[187384]: ansible-command Invoked with
_raw_params=tail /var/log/syslog warn=True _uses_shell=False stdin_add_newline=True
strip_empty_ends=True argv=None chdir=None executable=None creates=None
removes=None stdin=None
```

Как видите, Ansible фиксирует свои действия в syslog.

Утилита `ansible` не ограничивается модулями `ping` и `command`: вы можете использовать любой модуль по желанию. Например, следующей командой можно установить NGINX в Ubuntu:

```
$ ansible testserver -b -m package -a name=nginx
```

Если установить NGINX не удалось, то, возможно, нужно обновить список пакетов. Чтобы Ansible выполнила эквивалент команды `apt-get update` перед установкой пакета, замените аргумент `name=nginx` на `name=nginx update_cache=yes`.

Перезапустить Nginx можно так:

```
$ ansible testserver -b -m service -a "name=nginx state=restarted"
```

Поскольку только пользователь root может установить пакет NGINX и перезапустить службы, необходимо указать аргумент `-b`.

Остановка тестового сервера

В этой книге мы будем совершенствовать настройку тестового сервера, поэтому не привязывайтесь к своей первой виртуальной машине. Просто остановите ее командой:

```
$ vagrant destroy -f
```

Удобные настройки Vagrant

Vagrant поддерживает множество конфигурационных параметров для настройки виртуальных машин, но два из них особенно полезны при использовании Vagrant для тестирования: установка определенного IP-адреса и включение переадресации агента.

Переадресация портов и частные IP-адреса

Когда вы создаете новый Vagrantfile командой `vagrant init`, сетевая конфигурация по умолчанию позволяет получить доступ к виртуальной машине Vagrant только через порт SSH, который переадресуется с локального хоста. Для машины Vagrant, запускаемой первой, назначается

порт 2222, а для каждой последующей будет назначаться другой порт. Как результат, единственный способ получить доступ к машине Vagrant с конфигурацией по умолчанию – это подключиться по SSH к `localhost` через порт 2222. Vagrant переадресует этот порт в порт 22 внутри виртуальной машины Vagrant.

Эта конфигурация по умолчанию не очень удобна для тестирования веб-приложений, потому что веб-приложение будет прослушивать порт, к которому у нас нет доступа.

Есть два способа решить эту проблему. Один из них: настроить в Vagrant переадресацию дополнительных портов. Например, если ваше веб-приложение прослушивает порт 80 внутри машины Vagrant, то вы можете настроить переадресацию порта 8040 локального хоста в порт 80 на машине Vagrant. Точно так же можно переадресовать локальный порт 8443 в порт 443 гостевой системы.

Как показано на рис. 2.1, мы настроим Vagrant так, чтобы запросы браузера, поступающие в порты 8080 и 8443, наша локальная машина переадресовывала в порты 80 и 443 на машине Vagrant. Это позволит нам получить доступ к веб-серверу, работающему внутри Vagrant, обратившись по URL *http://localhost:8080* и *https://localhost:8443*.

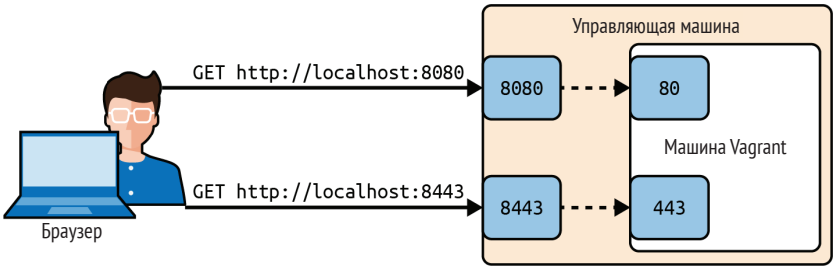


Рис. 2.1. Экспорт портов на машине Vagrant

В примере 2.3 показано, как настроить переадресацию портов в файле Vagrantfile.

Пример 2.3. Переадресация локального порта 8000 в порт 80 машины Vagrant

```
# Vagrantfile
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Другие конфигурационные параметры не показаны
  config.vm.network :forwarded_port, host: 8000, guest: 80
  config.vm.network :forwarded_port, host: 8443, guest: 443
end
```

Переадресация портов для других машин в локальной сети тоже будет выполняться, поэтому мы считаем более полезным назначать каждой

машине Vagrant свой IP-адрес. При таком подходе взаимодействие с ними становится больше похожим на взаимодействие с частными удаленными серверами: вы можете напрямую подключиться к порту 80 машины с указанным IP-адресом, а не к локальному порту 8000, и только вы сможете это сделать, если не станете настраивать переадресацию портов.

Простейший способ – назначить машине частный IP-адрес. В примере 2.4 показано, как назначить IP-адрес *192.168.33.10* виртуальной машине в файле Vagrantfile.

Пример 2.4. Назначение частного IP-адреса машине Vagrant

```
# Vagrantfile
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Другие конфигурационные параметры не показаны

  config.vm.network "private_network", ip: "192.168.33.10"
end
```

Если на машине Vagrant запустить веб-сервер, прослушивающий порт 80, то обратиться к нему можно будет по URL *http://192.168.33.10*.

В этой конфигурации используется *частная сеть* Vagrant. Виртуальная машина будет доступна только с машины, где работает Vagrant. Вы не сможете подключиться к этому IP-адресу с другой физической машины, даже если она находится в той же сети, что и машина, на которой работает Vagrant. Однако разные машины Vagrant могут подключаться друг к другу.

Дополнительную информацию о различных параметрах настройки сети в Vagrant вы найдете в документации (**https://oreil.ly/EXvBL**).

Включение переадресации агента

Если соединяетесь с удаленным репозиторием Git через SSH и используете переадресацию агента, то вам нужно также настроить виртуальную машину Vagrant так, чтобы Vagrant включал переадресацию агента при подключении к агенту через SSH (пример 2.5). Дополнительные сведения о переадресации агентов вы найдете в главе 20.

Пример 2.5. Включение переадресации агента

```
# Vagrantfile
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
```

```
  # Другие конфигурационные параметры не показаны
  # включение переадресации агента ssh
  config.ssh.forward_agent = true
end
```

Подготовка Docker

Иногда бывает нужно сравнить контейнеры, выполняющиеся в разных вариантах Linux, и разные среды выполнения контейнеров. Vagrant может создать виртуальную машину с нуля, установить Docker или Podman и автоматически запустить образ контейнера за один раз:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/focal64"
  config.vm.provision "docker" do |d|
    d.run "nginx"
  end
end
```

Подготовка локальной версии Ansible

Для Vagrant есть внешние инструменты, называемые *провайдерами (provisioners)*, которые он использует для настройки виртуальной машины после ее запуска. Помимо Ansible, Vagrant также может предоставлять сценарии командной оболочки, устанавливать Chef, Puppet, Salt и CFEngine.

В примере 2.6 показан файл Vagrantfile с настройкой `ansible_local`, согласно которой на виртуальную машину устанавливается система Ansible и используется в качестве провайдера, в частности, с помощью сценария Ansible с именем *playbook.yml*.

Пример 2.6. *Vagrantfile*

```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/xenial64"
  config.vm.provision "ansible_local" do |ansible|
    ansible.compatibility_mode = "2.0"
    ansible.galaxy_role_file = "roles/requirements.yml"
    ansible.galaxy_roles_path = "roles"
    ansible.playbook = "playbook.yml"
    ansible.verbose = "vv"
  end
end
```

Благодаря этому нет необходимости вручную устанавливать Ansible на свой компьютер. Если в вашем файле Vagrantfile имеется настройка

`config.vm.provision "ansible_local"`, то система будет установлена и запущена в виртуальной машине. При использовании настройки `config.vm.provision "ansible"` в Vagrantfile провайдер будет использовать версию Ansible, уже установленную на вашем компьютере.

Когда запускаются сценарии провайдеров

Когда в первый раз запускается команда `vagrant up`, Vagrant выполнит сценарий, осуществляющий подготовку и наполнение виртуальной машины, и зафиксирует факт своего запуска. После остановки и повторного запуска виртуальной машины Vagrant «вспомнит», что сценарий провайдера уже выполнялся, и не будет повторно запускать его.

При желании можно принудительно запустить сценарий наполнения на запущенной виртуальной машине:

```
$ vagrant provision
```

Можно также перезагрузить виртуальную машину и запустить сценарий наполнения после перезагрузки:

```
$ vagrant reload --provision
```

Аналогично можно запустить остановленную виртуальную машину с принудительным запуском сценария наполнения:

```
$ vagrant up --provision
```

Мы часто используем эти команды для запуска сценариев Ansible из командной строки с некоторым тегом или ограничением.

Плагины Vagrant

Возможности Vagrant можно расширять с помощью механизма плагинов. В последних версиях достаточно просто перечислить нужные плагины. Давайте рассмотрим два примера: `vagrant-hostmanager` и `vagrant-vbguest`:

```
config.vagrant.plugins = ["vagrant-hostmanager", "vagrant-vbguest"]
```

vagrant-hostmanager

Плагин `vagrant-hostmanager` помогает обращаться к нескольким виртуальным машинам по именам хостов. Он изменит имена хостов и добавит гостевые системы в */etc/hosts*, а иногда и сам хост, в зависимости от конфигурации:

```
# управление файлом /etc/hosts
config.hostmanager.enabled = true
```

```
config.hostmanager.include_offline = true
config.hostmanager.manage_guest = true
config.hostmanager.manage_host = true
```

vagrant-vbguest

Плагин `vagrant-vbguest` работает в VirtualBox и может автоматически устанавливать или обновлять дополнения для гостевой системы (Guest Additions) в гостевых виртуальных машинах. Бас обычно отключает эти функции в macOS, потому что обмен файлами между гостевыми системами и macOS недостаточно быстр и не всегда надежен. Более того, обмен файлами между хостом и гостевой системой не имитирует порядок развертывания программного обеспечения в окружениях разработки, тестирования, обкатки и промышленной эксплуатации. Но он отлично подходит для изучения Ansible в Windows:

```
# обновление дополнений гостевых систем
if Vagrant.has_plugin?("vagrant-vbguest")
  config.vbguest.auto_update = true
end
```

Настройка VirtualBox

При желании можно определить свойства виртуальной машины и ее внешний вид в VirtualBox. Например:

```
host_config.vm.provider "virtualbox" do |vb|
  vb.name = "web"
  virtualbox.customize ["modifyvm", :id,
       "--audio", "none",
       "--cpus", 2,
       "--memory", 2048,
       "--graphicscontroller", "VMSVGA",
       "--vram", "64"
    ]
end
```

Vagrantfile – это Ruby

Файлы Vagrantfile выполняются интерпретатором Ruby. Это знание может вам пригодиться хотя бы для настройки подсветки синтаксиса в текстовом редакторе. В Vagrantfile можно объявлять переменные, использовать управляющие структуры и циклы и т. д. В примерах исходного кода, прилагаемых к этой книге, есть более сложный пример файла Vagrantfile (**https://oreil.ly/h1jTF**), который мы используем для работы с 15 различными вариантами Linux, как показано на рис. 2.2.

Рис. 2.2. Запуск различных дистрибутивов Linux в VirtualBox

Для настройки гостевых систем мы используем файл JSON с такими элементами, как:

```
[
    {
       "name": "centos8",
       "cpus": 1,
       "distro": "centos",
       "family": "redhat",
       "gui": false,
       "box": "centos/stream8",
       "ip_addr": "192.168.56.6",
       "memory": "1024",
       "no_share": false,
       "app_port": "80",
```

```
        "forwarded_port": "8006"
    },
    {
        "name": "focal",
        "cpus": 1,
        "distro": "ubuntu",
        "family": "debian",
        "gui": false,
        "box": "ubuntu/focal64",
        "ip_addr": "192.168.56.8",
        "memory": "1024",
        "no_share": false,
        "app_port": "80",
        "forwarded_port": "8008"
    }
]
```

И в файле Vagrantfile у нас есть пара конструкций для создания одной гостевой системы по имени при входе, например:

```
$ vagrant up focal
```

Вот сам файл Vagrantfile:

```
Vagrant.require_version ">= 2.0.0"
# Подключить модуль JSON
require 'json'
# Прочитать файл JSON с настройками
f = JSON.parse(File.read(File.join(File.dirname(__FILE__), 'config.json')))
# Локальная переменная PATH_SRC для монтирования
$PathSrc = ENV['PATH_SRC'] || "."
Vagrant.configure(2) do |config|
  config.vagrant.plugins = ["vagrant-hostmanager", "vagrant-vbguest"]
  # проверить обновления базового образа
  config.vm.box_check_update = true
  # небольшая задержка
  config.vm.boot_timeout = 1200
  # запретить обновление дополнений гостевой системы
  if Vagrant.has_plugin?("vagrant-vbguest")
    config.vbguest.auto_update = false
  end
  # включить переадресацию агента ssh
  config.ssh.forward_agent = true
  # использовать стандартный для vagrant ключ ssh
  config.ssh.insert_key = false
  # управление файлом /etc/hosts
  config.hostmanager.enabled = true
  config.hostmanager.include_offline = true
```

```
  config.hostmanager.manage_guest = true
  config.hostmanager.manage_host = true
  # Цикл по элементам в файле JSON
  f.each do |g|
    config.vm.define g['name'] do |s|
      s.vm.box = g['box']
      s.vm.hostname = g['name']
      s.vm.network 'private_network', ip: g['ip_addr']
      s.vm.network :forwarded_port,
        host: g['forwarded_port'],
        guest: g['app_port']
      # установить значение no_share равным false,
      # чтобы разрешить совместное использование файлов
      s.vm.synced_folder ".", "/vagrant", disabled: g['no_share']
      s.vm.provider :virtualbox do |virtualbox|
        virtualbox.customize ["modifyvm", :id,
           "--audio", "none",
           "--cpus", g['cpus'],
           "--memory", g['memory'],
           "--graphicscontroller", "VMSVGA",
           "--vram", "64"
        ]
        virtualbox.gui = g['gui']
        virtualbox.name = g['name']
      end
    end
  end
  config.vm.provision "ansible_local" do |ansible|
    ansible.compatibility_mode = "2.0"
    ansible.galaxy_role_file = "roles/requirements.yml"
    ansible.galaxy_roles_path = "roles"
    ansible.playbook = "playbook.yml"
    ansible.verbose = "vv"
  end
end
```

Свойства всех виртуальных машин настраиваются в файле *config.json*.

Настройка промышленного окружения

Для подключения к машинам Linux/macOS/BSD Ansible использует SSH, а для подключения к машинам Windows – WinRM. Сетевыми устройствами можно управлять через HTTPS или SSH. Никакого дополнительного программного обеспечения на целевых хостах устанавливать не требуется (при условии, что на машинах Linux/macOS/BSD установлен Python, а на машинах Windows – PowerShell).

Традиционные системные администраторы проявляют здоровую осторожность при внедрении инструментов, требующих системных привилегий, потому что обычно только сами системные администраторы имеют такие разрешения. В Unix принято делегировать разработчикам доступ только к определенным командам с помощью `sudo` с тщательно выверенными файлами в */etc/sudoers.d/*.

Этот подход не работает ни с Ansible, ни с такой ограничительной оболочкой, как `rbash`. Ansible создает временные каталоги со случайными именами для различных сценариев на Python, тогда как `sudo` нужны точные команды. Альтернативой является смещение фокуса на содержание изменений в системе управления версиями в окружении обкатки и наличие файла с настройками `sudo` для группы `ansible`, например:

```
%ansible    ALL=(ALL) ALL
```

Заключение

В этой главе был представлен обзор создания и настройки тестового окружения с помощью VirtualBox и Vagrant для изучения Ansible. Vagrant поддерживает множество настроек, которые не рассматривались в этой главе. Дополнительные сведения вы найдете в официальной документации Vagrant. Описание всех возможностей Vagrant выходит за рамки этой книги. Для более близкого знакомства с Vagrant мы рекомендуем прочитать книгу «*Vagrant: Up and Running*» (O'Reilly) Митчелла Хашимото (Mitchell Hashimoto), создателя Vagrant.

Глава 3

Сценарии: начало

Приступая к использованию Ansible, вы начинаете с написания сценариев. *Сценарием* (playbook) в Ansible называется файл, описывающий порядок управления конфигурациями. Рассмотрим, например, установку веб-сервера NGINX и его настройку для поддержки защищенных соединений.

К концу этой главы у вас появятся следующие файлы и каталоги:

```
.
├── Vagrantfile
├── ansible.cfg
├── files
│   ├── index.html
│   ├── nginx.conf
│   ├── nginx.crt
│   └── nginx.key
├── inventory
│   └── vagrant.ini
├── requirements.txt
├── templates
│   ├── index.html.j2
│   └── nginx.conf.j2
├── webservers-tls.yml
├── webservers.yml
└── webservers2.yml
```

Подготовка

Измените содержимое Vagrantfile, как показано ниже:

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/focal64"
  config.vm.hostname = "testserver"
  config.vm.network "forwarded_port",
    id: 'ssh', guest: 22, host: 2202, host_ip: "127.0.0.1", auto_correct: false
  config.vm.network "forwarded_port",
    id: 'http', guest: 80, host: 8080, host_ip: "127.0.0.1"
```

```
  config.vm.network "forwarded_port",
    id: 'https', guest: 443, host: 8443, host_ip: "127.0.0.1"
  # запретить обновление дополнений гостевой системы
  if Vagrant.has_plugin?("vagrant-vbguest")
    config.vbguest.auto_update = false
  end
  config.vm.provider "virtualbox" do |virtualbox|
    virtualbox.name = "ch03"
  end
end
```

Эти настройки отобразят порты 8080 и 8443 локальной машины в порты 80 и 443 машины Vagrant, а также зарезервируют переадресацию локального порта 2202 в порт 22 этой конкретной виртуальной машины, как мы делали это в главе 1. После сохранения изменений дайте команду применить их:

```
$ vagrant up
```

В результате на экране должны появиться следующие строки:

```
==> default: Forwarding ports...
     default: 22 (guest) => 2202 (host) (adapter 1)
     default: 80 (guest) => 8080 (host) (adapter 1)
     default: 443 (guest) => 8443 (host) (adapter 1)
```

Теперь ваш тестовый сервер запущен и готов к экспериментам.

Очень простой сценарий

В нашем первом примере сценария мы настроим хост для запуска простого веб-сервера. Сначала посмотрим, что получится, если запустить сценарий *webservers.yml* (пример 3.1), а затем детально изучим его содержимое. Это простейший из возможных сценариев для решения такой задачи, однако в процессе обсуждения мы будем постепенно улучшать его.

Пример 3.1. *webservers.yml*

```
---

- name: Configure webserver with nginx
  hosts: webservers
  become: True
  tasks:
    - name: Ensure nginx is installed
      package: name=nginx update_cache=yes

    - name: Copy nginx config file
```

```
        copy:
          src: nginx.conf
          dest: /etc/nginx/sites-available/default

      - name: Enable configuration
        file: >
          dest=/etc/nginx/sites-enabled/default
          src=/etc/nginx/sites-available/default
          state=link

      - name: Copy index.html
        template: >
          src=index.html.j2
          dest=/usr/share/nginx/html/index.html

      - name: Restart nginx
        service: name=nginx state=restarted
...
```

Файл конфигурации NGINX

Данному сценарию необходим дополнительный файл конфигурации NGINX.

NGINX распространяется с готовым конфигурационным файлом, настраивающим сервер на обслуживание только статичных файлов, и очень часто его приходится адаптировать под конкретные нужды. Поэтому мы изменим файл конфигурации по умолчанию в рамках данного примера. Позднее мы также добавим в файл конфигурации поддержку TLS. В примере 3.2 приводится стандартный конфигурационный файл NGINX. Сохраните его в файле с именем *playbooks/files/nginx.conf*[1].

Пример 3.2. *nginx.conf*

```
server {
        listen 80 default_server;
        listen [::]:80 default_server ipv6only=on;

        root /usr/share/nginx/html;
        index index.html index.htm;

        server_name localhost;

        location / {
                try_files $uri $uri/ =404;
        }
}
```

[1] Наш файл *nginx.conf*, несмотря на данное ему имя, заменит файл *sites-enabled/default*, а не основной конфигурационный файл */etc/nginx.conf*.

Создание веб-страницы

Теперь добавим простую веб-страницу. Ansible включает систему генерирования HTML-страниц на основе файлов-шаблонов. Сохраните код из примера 3.3 в файле *playbooks/templates/index.html.j2*.

```
<html>
  <head>
    <title>Welcome to ansible</title>
  </head>
  <body>
  <h1>Nginx, configured by Ansible</h1>
  <p>If you can see this, Ansible successfully installed nginx.</p>

  <p>Running on {{ inventory_hostname }}</p>
  </body>
</html>
```

В этом шаблоне используется специальная переменная Ansible `inventory_hostname`. Обрабатывая шаблон, Ansible заменит ссылку на нее именем хоста, указанным в реестре (рис. 3.1). Полученная разметка HTML подскажет браузеру, как отобразить страницу.

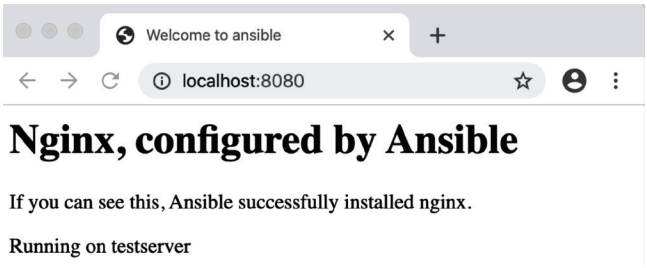


Рис. 3.1. Вид получившейся страницы

В соответствии с соглашениями Ansible копирует файлы из каталога *files*, а шаблоны Jinja2 ищет в подкаталоге *templates*. Поиск в этих каталогах система Ansible выполняет автоматически. Мы будем следовать этому соглашению на протяжении всей книги.

Создание группы веб-серверов

Теперь создадим группу `webservers` в файле реестра, чтобы получить возможность сослаться на нее в сценарии. Пока в эту группу войдет только наш тестовый сервер `testserver`.

Файлы реестра имеют формат *.ini*. Подробнее этот формат мы рассмотрим далее в книге. Откройте файл *playbooks/inventory/vagrant.ini* в редакторе и добавьте строку `[webservers]` над строкой `testserver`, как показано в примере 3.4. Это означает, что `testserver` включен в группу `webservers`.

Пример 3.4. *playbooks/inventory/vagrant.ini*

```
[webservers]
testserver ansible_port=2202

[webservers:vars]
ansible_user = vagrant
ansible_host = 127.0.0.1
ansible_private_key_file = .vagrant/machines/default/virtualbox/private_key
```

В главе 1 мы создали файл *ansible.cfg* со ссылкой на реестр, поэтому нет необходимости использовать параметр командной строки `-i`. Теперь проверим группы в реестре с помощью следующей команды:

```
$ ansible-inventory --graph
```

Она должна вывести:

```
@all:
  |--@ungrouped:
  |--@webservers:
  | |--testserver
```

Запуск сценария

Сценарии запускаются командой `ansible-playbook`, например:

```
$ ansible-playbook webservers.yml
```

В примере 3.5 показано, как должен выглядеть результат.

Пример 3.5. Результат запуска сценария командой *ansible-playbook*

```
PLAY [Configure webserver with nginx] *****************************************
TASK [Gathering Facts] ********************************************************
ok: [testserver]

TASK [Ensure nginx is installed] **********************************************
changed: [testserver]

TASK [Copy nginx config file] *************************************************
changed: [testserver]

TASK [Enable configuration] ***************************************************
ok: [testserver]

TASK [Copy index.html] ********************************************************
changed: [testserver]

TASK [Restart nginx] **********************************************************
changed: [testserver]

PLAY RECAP ********************************************************************
```

```
testserver : ok=6 changed=4 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
Playbook run took 0 days, 0 hours, 0 minutes, 18 seconds
```

Если в процессе работы сценария не возникло никаких ошибок, то по его завершении можно запустить веб-браузер и открыть страницу *http://localhost:8080*, которая должна выглядеть, как показано на рис. 3.1[1].

Ни одна книга O'Reilly с такой обложкой не была бы полной без описания программы `cowsay`. Если на вашей локальной машине установлена программа `cowsay`, вывод Ansible будет выглядеть так:

```
< PLAY [Configure webserver with nginx] >
 -------------------------------------
        \  ^__^
         \ (oo)\_______
           (__)\       )\/\
               ||----w |
               ||     ||
```

Если вы не хотите видеть говорящих коров, то добавьте в файл `ansible.cfg` следующие строки:

```
[defaults]
cow_selection = random
cowsay_enabled_stencils=cow,bunny,kitty,koala,moose,sheep,tux
```

Также отключить использование программы `cowsay` можно настройкой переменной окружения `ANSIBLE_NOCOWS`:

```
$ export ANSIBLE_NOCOWS=1
```

Сценарии пишутся на YAML

Все сценарии Ansible пишутся на YAML. YAML – это язык разметки, напоминающий JSON, но намного проще для восприятия человеком. Прежде чем перейти к сценарию, рассмотрим основные понятия YAML, наиболее важные при написании сценариев.

Допустимый файл в формате JSON является также допустимым файлом в формате YAML, потому что YAML допускает заключение строк в кавычки, воспринимает значения `true` и `false` как действительные логические выражения, а также синтаксис определения списков и словарей, аналогичный синтаксису массивов и объектов в JSON. Но я не советую писать сценарии на JSON, поскольку человеку гораздо проще читать YAML.

[1] Если у вас возникли ошибки, то перейдите к главе 8, где описываются приемы отладки.

Начало файла

Документы YAML начинаются с трех дефисов, обозначающих его начало. Каждый файл Ansible может содержать только один документ YAML.

```
---
```

Сценарии Ansible принято начинать с трех дефисов (чтобы явно обозначить начало). Однако Ansible не посчитает ошибкой, если вы забудете указать эти дефисы.

Конец файла

Файлы YAML принято заканчивать тремя точками, чтобы явно отметить конец документа. Но многие не следуют этой практике.

```
...
```

Однако Ansible не посчитает ошибкой, если вы забудете поставить эти три точки в конце своего файла.

Комментарии

Комментарии начинаются со знака решетки (#) и продолжаются до конца строки, как в сценариях на языке командной оболочки, Python и Ruby. Отступы в комментариях принято устанавливать вровень с кодом.

```
# Это комментарий на языке YAML
```

Отступы и пробельные строки

Как и в языке Python, в документах YAML принято оформлять отступы пробелами, чтобы уменьшить количество знаков пунктуации. Мы используем два пробела. А для большей удобочитаемости мы предпочитаем добавлять пробельные строки между задачами в сценарии и между разделами в файлах.

Строки

Обычно строки в YAML не заключаются в кавычки, даже если они включают пробелы. Хотя это не возбраняется. Например, вот строка на языке YAML:

```
это пример предложения
```

Аналог в JSON выглядит так:

```
"это пример предложения"
```

Иногда Ansible требует заключать строки в кавычки. Хорошей практикой считается просто заключать в кавычки все строки. В двойные кавыч-

ки обычно принято заключать имена переменных в выражениях интерполяции. Одинарные кавычки принято использовать для литеральных значений, которые не должны интерполироваться, таких как номера версий и числа с плавающей точкой или строки с зарезервированными символами, такими как двоеточия, круглые или фигурные скобки. Но об этом чуть позже.

Никогда, никогда не заключайте в кавычки логические значения! Помните: `NO` – это строка (аббревиатура, обозначающая Норвегию [Norway]).

Почему в одном случае используется «True», а в другом «Yes»?

Внимательный читатель заметит, что в *webservers.yml* в одном случае используется `True` (для получения привилегий root с помощью `become`) и `yes` в другом случае (для обновления кеша `apt`).

Ansible – достаточно гибкая система в отношении обозначения в сценариях значений «истина» и «ложь». Строго говоря, аргументы модуля (такие как `update_cache=yes`) интерпретируются иначе, чем значения где-либо еще в сценарии (такие как `become: True`). Эти и другие значения обрабатываются синтаксическим анализатором YAML и, следовательно, подчиняются обозначениям значений «истина» и «ложь» YAML:

- истина в YAML: `true, True, TRUE, yes, Yes, YES, on, On, ON`;
- ложь в YAML: `false, False, FALSE, no, No, NO, off, Off, OFF`.

Аргументы передаются модулям в виде строк и подчиняются внутренним соглашениям в Ansible:

- истина в аргументе модуля: `yes, on, 1, true`;
- ложь в аргументе модуля: `no, off, 0, false`.

Рекомендуется проверять все файлы YAML с помощью инструмента командной строки `yamllint`. С настройками по умолчанию он выдаст следующее предупреждение:

```
warning truthy value should be one of [false, true] (truthy)
```

Придерживаясь этого правила обозначения истинности, Бас использует только `true` и `false` (без кавычек).

Булевы выражения

В YAML есть собственный логический тип. Он предлагает широкий выбор строк, которые могут интерпретироваться как «истина» и «ложь». Вот примеры истинных значений в YAML:

```
true, True, TRUE, yes, Yes, YES, on, On, ON
```

В JSON истинное значение выглядит так:

```
true
```

А это примеры ложных значений в YAML:

```
false, False, FALSE, no, No, NO, off, Off, OFF
```

В JSON же используется только одно значение:

```
false
```

Бас использует только строчные буквы `true` и `false`. Одна из причин таких предпочтений заключается в том, что эти два значения являются возвращаемыми; например, они выводятся в режиме отладки, даже при использовании любого другого разрешенного варианта. Поскольку `true` и `false` также являются допустимыми логическими значениями в JSON, их использование упрощает использование динамических данных, потому что действия Ansible возвращают результаты в виде данных в формате JSON.

Списки

Списки в YAML похожи на массивы в JSON и Ruby или списки в Python. Строго говоря, в YAML они называются *последовательностями*, но мы называем их *списками*, чтобы избежать противоречий с официальной документацией Ansible.

Списки оформляются с помощью отступов и дефиса. Определение каждого списка начинается с его имени и следующего за ним двоеточия:

```
shows:
   - My Fair Lady
   - Oklahoma
   - The Pirates of Penzance
```

Аналог в JSON:

```
{
   "shows": [
     "My Fair Lady",
     "Oklahoma",
     "The Pirates of Penzance"
     ]
}
```

Как видите, списки в YAML легче читаются, потому что при их оформлении используется меньше лишних символов. Еще раз обратите внимание, что в YAML не нужно заключать строки в кавычки даже при наличии в них пробелов. YAML также поддерживает формат встроенных

списков. Такие списки заключаются в квадратные скобки, а элементы списка разделяются запятой, как показано ниже:

```
shows: [ My Fair Lady , Oklahoma , The Pirates of Penzance ]
```

Словари

Словари в YAML подобны объектам в JSON, словарям в Python, хеш-массивам в Ruby или ассоциативным массивам в PHP. Технически в YAML они называются *отображениями* (mapping), но мы называем их *словарями*, чтобы избежать противоречий с официальной документацией Ansible. Они выглядят так:

```
address:
  street: Main Street
  appt: 742
  city: Logan
  state: Ohio
```

Аналог в JSON:

```
{
  "address": {
    "street": "Main Street",
    "appt": 742,
    "city": "Logan",
    "state": "Ohio"
  }
}
```

YAML также поддерживает формат встроенных словарей. Такие словари заключаются в фигурные скобки, а элементы словаря разделяются запятой, как показано ниже:

```
address: { street: Main Street, appt: '742', city: Logan, state: Ohio}
```

Многострочные строковые значения

YAML поддерживает многострочные строковые значения, распознавая так называемые *операторные скобки* (| и >), символы, обозначающие начало многострочного текста (+ и -), и даже отступы (от 1 до 9). Например, чтобы задать предварительно отформатированный текстовый блок, можно использовать вертикальную черту со знаком плюс (|+):

```
---
visiting_address: |+
  Department of Computer Science

  A.V. Williams Building
  University of Maryland
```

```
city: College Park
state: Maryland
```

Синтаксический анализатор YAML сохранит разрывы строк, как вы указали их.

JSON не поддерживает использование многострочных строковых значений. Поэтому все разрывы строк нужно заменить на `\n` или использовать массив:

```
{
   "visiting_address": ["Department of Computer Science",
       "A.V. Williams Building",
       "University of Maryland"],
   "city": "College Park",
   "state": "Maryland"
}
```

Чистый YAML вместо строковых аргументов

При разработке сценариев Ansible вы часто будете сталкиваться с необходимостью передать модулю множество аргументов. Для эстетики их можно поместить в несколько строк. Кроме того, желательно, чтобы Ansible интерпретировал аргументы как словарь YAML, потому что в YAML можно использовать `yamllint` для поиска опечаток , которые трудно отыскать при использовании строкового формата. Этот стиль также позволяет вводить более короткие строки, что упрощает сравнение версий.

Лорин нравится такой стиль:

```
- name: Ensure nginx is installed
  package: name=nginx update_cache=true
```

Бас предпочитает стиль, принятый в YAML, потому что его корректность можно проанализировать с помощью `yamllint`:

```
- name: Ensure nginx is installed
  package:
    name: nginx
    update_cache: true
```

Структура сценария

Применив все правила, описанные выше, к нашему сценарию, получим вторую версию (пример 3.6):

Пример 3.6. *webservers2.yml*

```
---
- name: Configure webserver with nginx
```

```
  hosts: webservers
  become: true
  tasks:
    - name: Ensure nginx is installed
      package:
        name: nginx
        update_cache: true

    - name: Copy nginx config file
      copy:
        src: nginx.conf
        dest: /etc/nginx/sites-available/default

    - name: Enable configuration
      file:
        src: /etc/nginx/sites-available/default
        dest: /etc/nginx/sites-enabled/default
        state: link

    - name: Copy home page template
      template:
        src: index.html.j2
        dest: /usr/share/nginx/html/index.html

    - name: Restart nginx
      service:
        name: nginx
        state: restarted
...
```

Операции

В любом формате – YAML или JSON – сценарий является списком словарей, или списком *операций* (play). Наш примерный сценарий содержит список с единственной операцией с именем `Configure webserver with nginx` (Настройка веб-сервера nginx).

Вот эта операция:

```
- name: Configure webserver with nginx
  hosts: webservers
  become: true

  tasks:
    - name: Ensure nginx is installed
      package:
        name: nginx
        update_cache: true
```

```
    - name: Copy nginx config file
      copy:
        src: nginx.conf
        dest: /etc/nginx/sites-available/default

    - name: Enable configuration
      file:
        src: /etc/nginx/sites-available/default
        dest: /etc/nginx/sites-enabled/default
        state: link

    - name: Copy index.html
      template:
        src: index.html.j2
        dest: /usr/share/nginx/html/index.html

    - name: Restart nginx
      service:
        name: nginx
        state: restarted
...
```

Каждая операция должна содержать переменную `hosts`, определяющую список хостов или группу, такую как `webservers` или волшебную группу `all` (все хосты в реестре), к которым будет применяться эта операция. Воспринимайте операцию как нечто, связывающее хосты и задачи. Иногда вам придется определять разные операции для разных групп хостов, и вы будете определять по несколько операций в сценариях.

Кроме хостов и задач, операции также могут содержать параметры. Мы рассмотрим этот вопрос позднее, а сейчас познакомимся с тремя основными параметрами.

`name:`

Комментарий, описывающий операцию. Ansible выведет его перед запуском операции. Считается хорошим тоном начинать комментарий с заглавной буквы.

`become:`

Если имеет значение «истина», то Ansible выполнит каждую задачу, предварительно приобретя привилегии пользователя, объявленного в параметре `become_user`. Это может пригодиться для управления серверами Linux, поскольку многие версии этой системы не позволяют устанавливать SSH-соединение с привилегиями root. При необходимости параметр `become` можно указать для каждой задачи или для каждой операции, а в `become_user` можно

указать пользователя root (этот пользователь подразумевается по умолчанию, если он опущен) или другого пользователя, но имейте в виду, что become подчиняется политикам вашей системы. Возможно, потребуется изменить файл *sudoers*, чтобы назначенный пользователь мог получить привилегии root.

`vars:`

Список переменных и значений. Мы увидим назначение этого параметра позднее в данной главе.

Задачи

Наш пример сценария содержит одну операцию с пятью задачами. Вот первая задача:

```
- name: Ensure nginx is installed
  package:
    name: nginx
    update_cache: true
```

В этом примере задача вызывает модуль с именем `package` и передает ему аргументы `name: nginx` и `update_cache: yes`. Эти аргументы сообщают модулю `package`, что он должен установить пакет с именем `nginx` и обновить кеш пакетов (эквивалентно выполнению команды `apt-get update` в Ubuntu) перед установкой пакета.

Даже притом, что имена задач можно не указывать, я рекомендую использовать их, поскольку они служат хорошими напоминаниями их целей. Имена будут особенно полезны для тех, кто попытается разобраться в вашем сценарии, в том числе и вам через полгода. Как мы уже видели, Ansible выводит имя задачи перед ее запуском. Наконец, как вы увидите в главе 16, можно также использовать флаг `--start-at-task <имя задачи>`, чтобы с помощью `ansible-playbook` запустить сценарий с середины операции. В этом случае необходимо сослаться на задачу по имени.

Аргументы для модуля можно передать команде `ansible` в виде одной строки в параметре `-a` и использовать параметр `-m` для передачи имени модуля:

```
$ ansible webservers -b -m package -a 'name=nginx update_cache=true'
```

Однако важно помнить, что с точки зрения парсера Ansible аргументы воспринимаются как одна строка, а не словарь. В командной строке это не вызывает никаких проблем, но в сценариях является частым источником ошибок, особенно когда используются сложные модули с большим количеством необязательных аргументов. Для облегчения управления версиями Бас предпочитает разбивать аргументы на несколько строк. Поэтому он всегда использует синтаксис YAML:

```
- name: Ensure nginx is installed
  package:
    name: nginx
    update_cache: true
```

Модули

Модули – это сценарии, которые поставляются с Ansible и производят определенные действия на хосте. Правда, надо признать, что это довольно общее описание, но среди модулей Ansible встречается множество вариантов. Как рассказывалось в главе 1, Ansible выполняет задачу на хосте, генерируя сценарий, исходя из имени модуля и аргументов, а затем копирует этот сценарий на хост и запускает его. Модули для Unix/Linux, которые поставляются с Ansible, написаны на Python, а модули для Windows написаны на PowerShell. Вы же можете писать свои модули на любом языке.

В этой главе используются следующие модули.

package
: Устанавливает или удаляет пакеты с использованием диспетчера пакетов хоста.

copy
: Копирует файл с локальной машины на хосты.

file
: Устанавливает атрибуты файла, символической ссылки или каталога.

service
: Запускает, останавливает или перезапускает службу.

template
: Создает файл на основе шаблона и копирует его на хосты.

Документация по модулям Ansible

Ansible поставляется с утилитой командной строки `ansible-doc`, которая выводит документацию по модулям Ansible. Используйте ее как man-страницы для модулей Ansible. Например, для вывода документации к модулю `service` выполните команду:

```
$ ansible-doc service
```

Чтобы найти более конкретные модули, связанные с диспетчером пакетов `apt` в Ubuntu, попробуйте выполнить команду:

```
$ ansible-doc -l | grep ^apt
```

Резюме

Итак, сценарий содержит одну или несколько операций. Операции назначаются неупорядоченному множеству хостов и содержат упорядоченные списки задач. Каждая задача использует ровно один модуль. Диаграмма на рис. 3.2 изображает взаимосвязи между сценариями, операциями, хостами, задачами и модулями.

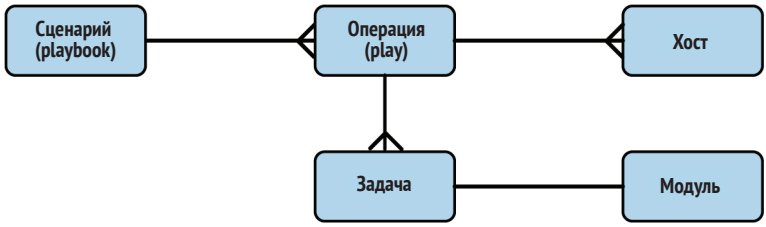


Рис. 3.2. Диаграмма взаимосвязей

Есть изменения? Отслеживание состояния хоста

Когда вы запускаете команду `ansible-playbook`, она выводит информацию о состоянии каждой задачи, выполняемой в рамках операции.

Вернитесь к примеру 3.5 и обратите внимание, что состояние некоторых задач указано как `changed` (изменено), а других – `ok`. Например, задача «Ensure nginx is installed task» (Проверить, установлен ли nginx) имеет статус `changed`. На моем терминале он выделен желтым.

```
TASK: [Ensure nginx is installed] ***********************************************
changed: [testserver]
```

С другой стороны, задача «Enable configuration» (Включить конфигурацию) имеет статус `ok`, на моем терминале он выделен зеленым:

```
TASK: [Enable configuration] ****************************************************
ok: [testserver]
```

Любая запущенная задача потенциально может изменить состояние хоста. Перед тем как совершить какое-либо действие, модули проверяют, требуется ли изменить состояние хоста. Если состояние хоста соответствует значениям аргументов модуля, то Ansible не предпринимает никаких действий и сообщает, что статус `ok`.

Если между состоянием хоста и значениями аргументов модуля есть разница, то Ansible вносит изменения в состояние хоста и сообщает, что статус был изменен (`changed`).

Как показано в примере выше, задача «Ensure nginx is installed» внесла изменения, а это значит, что до запуска сценария пакет `nginx` не был установлен. Задача «Enable configuration» не внесла изменений, значит, на сервере уже был сохранен файл конфигурации, и он идентичен тому,

который предполагалось скопировать. Из вышесказанного следует, что в сценарии могут иметься «пустые операции» (ничего не делающие), которые мы удалим. Старайтесь запускать сценарии чаще и проверяйте, что при последующих запусках задачи возвращают статус ok.

Позже в этой главе мы увидим, что способность Ansible определять изменение состояния можно использовать для выполнения дополнительных действий с помощью обработчиков. Но даже без обработчиков полезно иметь в своем распоряжении информацию об изменении состояния хостов в результате выполнения сценария.

Становимся знатоками: поддержка TLS

Теперь рассмотрим более сложный пример. Добавим в предыдущий сценарий настройку поддержки TLSv1.2 веб-сервером. Полный сценарий приводится в примере 3.9, а в этом разделе мы кратко представим некоторые возможности Ansible:

- переменные;
- циклы;
- обработчики;
- тесты;
- проверки.

TLS и SSL

Возможно, вам более знакома аббревиатура SSL (Secure Sockets Layer – слой защищенных сокетов), чем TLS. SSL – это более старый протокол, используемый для обеспечения безопасности взаимодействий браузеров и веб-серверов; именно его использование отмечается добавлением символа «S» в HTTPS. SSL продолжает развиваться, и самая современная его версия имеет номер v1.3. Несмотря на то что многие продолжают использовать аббревиатуру SSL, подразумевая более новый протокол, в этой книге я буду использовать точное название: TLS.

Создание сертификата TLS

Мы должны вручную создать сертификат TLS. Для промышленной эксплуатации сертификат TLS необходимо приобрести в центре сертификации. Но мы будем использовать «самоподписанный» (self-signed) сертификат, поскольку его можно создать бесплатно. Выполните следующую команду в каталоге *ansiblebook/ch03/playbooks*:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
     -subj /CN=localhost \
     -keyout files/nginx.key -out files/nginx.crt
```

Она создаст файлы *nginx.key* и *nginx.crt* в подкаталоге *files* внутри каталога *playbooks*. Срок действия сертификата ограничен одним годом (365 дней) со дня его создания.

Переменные

Теперь операция в нашем сценарии включает раздел `vars:`. Этот раздел определяет пять переменных и каждой присваивает значение:

```
vars:
  tls_dir: /etc/nginx/ssl/
  key_file: nginx.key
  cert_file: nginx.crt
  conf_file: /etc/nginx/sites-available/default
  server_name: localhost
```

В нашем примере каждое значение – это строка (например, */etc/nginx/sites-available/default*), но вообще значением переменной может служить любое выражение, допустимое в YAML. В дополнение к строкам и булевым выражениям можно использовать списки и словари.

Переменные можно использовать в задачах и в файлах шаблонов. Для ссылки на переменные используются скобки, например `{{ mustache }}`. Ansible заменит это выражение `{{ mustache }}` значением переменной `mustache`.

В нашем сценарии имеется следующая задача:

```
- name: Manage nginx config template
  template:
    src: nginx.conf.j2
    dest: "{{ conf_file }}"
    mode: '0644'
  notify: Restart nginx
```

При ее выполнении Ansible заменит `"{{ conf_file }}"` на */etc/nginx/sites-available/default*.

Когда использовать кавычки в строках Ansible

Если ссылка на переменную следует сразу после имени модуля, то парсер YAML ошибочно воспримет ее как начало встроенного словаря. Например:

```
- name: Perform some task
  command: {{ myapp }} -a foo
```

Ansible попытается интерпретировать первую часть выражения `{{ myapp }} -a foo` не как строку, а как словарь и выдаст ошибку. В данном случае необходимо заключить аргументы в кавычки:

```
- name: Perform some task
  command: "{{ myapp }} -a foo"
```

Похожая ошибка возникает при наличии двоеточия в аргументе. Например:

```
- name: Show a debug message
  debug:
    msg: The debug module will print a message: neat, eh?
```

Двоеточие в аргументе `msg` сбивает с толку синтаксический анализатор YAML. Чтобы избежать этого, необходимо заключить в кавычки все выражение аргумента. Для этого можно использовать и одинарные, и двойные кавычки; Бас предпочитает использовать двойные кавычки, когда в строке есть переменные:

```
- name: Show a debug message
  debug:
    msg: "The debug module will print a message: neat, eh?"
```

Теперь синтаксический анализатор YAML не ошибется. Ansible поддерживает чередование одинарных и двойных кавычек, поэтому можно поступить так:

```
- name: Show escaped quotes
  debug:
    msg: '"The module will print escaped quotes: neat, eh?"'

- name: Show quoted quotes
  debug:
    msg: "'The module will print quoted quotes: neat, eh?'""
```

Это дает ожидаемый результат:

```
TASK [Show escaped quotes] ******************************************************
ok: [localhost] ==> {
    "msg": "\"The module will print escaped quotes: neat, eh?\""
}
TASK [Show quoted quotes] *******************************************************
ok: [localhost] ==> {
    "msg": "'The module will print quoted quotes: neat, eh?'"
}
```

Создание шаблона с конфигурацией NGINX

Если вы занимались веб-программированием, то, вероятно, сталкивались с системой шаблонов для создания разметки HTML. Если нет, то поясню, что шаблон – это простой текстовый файл, где с использованием специального синтаксиса определяются переменные, которые должны заменяться фактическими значениями. Если вы когда-либо получали автоматически сгенерированное электронное письмо от какой-либо компании, то наверняка заметили, что в письме используется шаблон, аналогичный приведенному в примере 3.7.

Пример 3.7. Шаблон электронного письма

```
Dear {{ name }},
You have {{ random_number }} Bitcoins in your account, please click: {{ phishing_url }}.
```

В случае с Ansible это не HTML-страницы или электронные письма, а файлы конфигурации. Если можно избежать редактирования файлов конфигурации вручную, то лучше так и поступить. Это особенно полезно, если используются одни и те же конфигурационные данные (например, IP-адрес сервера очереди или учетные сведения для базы данных) в нескольких файлах. Гораздо разумнее поместить информацию о конкретном окружении в одном месте, а затем создавать все файлы, требующие этой информации, на основе шаблона.

Для поддержки шаблонов в Ansible используется механизм Jinja2 , как и в замечательном веб-фреймворке Flask. Если вы когда-либо пользовались библиотеками шаблонов, такими как Mustache, ERB или Django, то Jinja2 покажется вам знакомым инструментом.

В файл конфигурации NGINX необходимо добавить информацию о месте хранения ключа и сертификата TLS. Чтобы исключить использование жестко заданных значений, которые могут изменяться со временем, мы воспользуемся поддержкой шаблонов в Ansible.

В каталоге *playbooks* создайте подкаталог *templates* и файл *templates/nginx.conf.j2*, как показано в примере 3.8.

Пример 3.8. *templates/nginx.conf.j2*

```
server {
        listen 80 default_server;
        listen [::]:80 default_server ipv6only=on;

        listen 443 ssl;
        ssl_protocols TLSv1.2;
        ssl_prefer_server_ciphers on;
        root /usr/share/nginx/html;
        index index.html;
```

```
        server_tokens off;
        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;

        server_name {{ server_name }};
        ssl_certificate {{ tls_dir }}{{ cert_file }};
        ssl_certificate_key {{ tls_dir }}{{ key_file }};

        location / {
            try_files $uri $uri/ =404;
        }
}
```

Мы используем расширение файла *.j2*, чтобы показать, что файл является шаблоном Jinja2. Однако вы можете использовать любое другое расширение. Для Ansible это неважно.

В нашем шаблоне используются четыре переменные.

`server_name`
: Имя хоста веб-сервера (например, *www.example.com*).

`cert_file`
: Путь к файлу сертификата TLS.

`key_file`
: Путь к файлу закрытого ключа TLS.

`tls_dir`
: Каталог со всеми этими файлами.

Мы определим эти переменные в сценарии.

Ansible также использует механизм шаблонов Jinja2 для определения переменных в сценариях. Вспомните: мы уже встречали выражение `{{ conf_file }}` в самом сценарии. Вы можете использовать все возможности Jinja2 в своих шаблонах, но мы не будем подробно рассматривать их здесь. За дополнительной информацией о шаблонах Jinja2 обращайтесь к официальной документации (**https://oreil.ly/Je0rA**). Впрочем, вам едва ли потребуются все продвинутые возможности. Но вы почти наверняка *будете* пользоваться фильтрами; мы рассмотрим их в последующей главе.

Циклы

Если потребуется запустить задачу для каждого элемента из списка, то для этого можно использовать цикл `loop`. Цикл выполняет задачу не-

сколько раз, каждый раз заменяя элемент `item` разными значениями из указанного списка:

```
- name: Copy TLS files
  copy:
    src: "{{ item }}"
    dest: "{{ tls_dir }}"
    mode: '0600'
  loop:
    - "{{ key_file }}"
    - "{{ cert_file }}"
  notify: Restart nginx
```

Обработчики

А теперь вернемся к нашему сценарию *webservers-tls.yml* (пример 3.9). Он содержит раздел `handlers` с определениями обработчиков:

```
handlers:
  - name: Restart nginx
    service:
      name: nginx
      state: restarted
```

Также в некоторых задачах можно увидеть инструкцию `notify`:

```
- name: Manage nginx config template
  template:
    src: nginx.conf.j2
    dest: "{{ conf_file }}"
    mode: '0644'
  notify: Restart nginx
```

Обработчики – это одна из форм условного выполнения, поддерживаемых в Ansible. *Обработчик* схож с задачей, но запускается только после получения уведомления от задачи. Задача посылает уведомление, если обнаруживается изменение состояния системы после ее выполнения.

Задача уведомляет обработчик с именем, переданным ей в аргументе. В предыдущем примере имя обработчика `Restart nginx`. Сервер NGINX нужно перезапустить, если изменится любой из компонентов:

- ключ TLS;
- сертификат TLS;
- файл конфигурации;
- содержимое каталога *sites-enabled*.

Мы добавляем инструкцию `notify` в каждую задачу, чтобы обеспечить перезапуск NGINX, если выполняется одно из этих условий.

Несколько фактов об обработчиках, которые необходимо помнить

Обработчики обычно выполняются после завершения всех задач и только один раз, даже если было получено несколько уведомлений. Чтобы запустить обработчик в середине операции, нужно добавить следующие две строки:

```
- name: Restart nginx
  meta: flush_handlers
```

Если сценарий содержит несколько обработчиков, то они всегда выполняются в порядке следования в разделе `handlers`, а не в порядке поступления уведомлений.

В официальной документации Ansible говорится, что обработчики в основном используются для перезапуска служб и перезагрузки. Лорин использует их исключительно для перезапуска служб – он считает такой подход небольшой оптимизацией, когда службы перезапускаются только один раз и при наличии изменений вместо безоговорочного перезапуска всех служб в конце сценария, потому что перезапуск одной службы обычно не занимает много времени. Но при перезапуске NGINX есть риск повлиять на сеансы пользователей; уведомления обработчиков помогают избежать ненужных перезапусков. Бас любит проверять конфигурацию перед перезапуском, особенно если речь идет о критически важной службе, такой как sshd. Он всегда использует обработчики, уведомляющие другие обработчики.

Тестирование

Одна из проблем, связанных с обработчиками, заключается в том, что они могут затруднять отладку сценариев. Проблема обычно разворачивается примерно так:

- я запускаю сценарий;
- одна из задач с уведомлением изменяет состояние;
- в следующей задаче возникает ошибка, прерывающая работу Ansible;
- я исправляю ошибку в сценарии;
- запускаю Ansible снова;
- ни одна из задач не сообщает об изменении состояния во второй раз, Ansible не запускает обработчик.

В таких случаях полезно включить тест в сценарий. В Ansible есть модуль `uri`, который может выполнять HTTPS-запросы для проверки работы веб-сервера:

```
- name: "Test it! https://localhost:8443/index.html"
  delegate_to: localhost
  become: false
  uri:
    url: 'https://localhost:8443/index.html'
    validate_certs: false
    return_content: true
  register: this
  failed_when: "'Running on ' not in this.content"
```

Проверка

Ansible замечательно умеет генерировать осмысленные сообщения об ошибках, если вы забудете поставить кавычки в нужных местах и в итоге получите недопустимый YAML, а `yamlint` помогает найти еще менее заметные проблемы. Кроме того, `ansible-lint` – это инструмент на языке Python, помогающий находить потенциальные проблемы в сценариях.

Обязательно проверяйте синтаксис Ansible ваших сценариев перед запуском. Вот как можно это сделать:

```
$ ansible-playbook --syntax-check webservers-tls.yml
$ ansible-lint webservers-tls.yml
$ yamllint webservers-tls.yml
$ ansible-inventory --host testserver -i inventory/vagrant.ini
$ vagrant validate
```

Сценарий

Если вы следовали за примерами в этой главе, то теперь ваш сценарий должен выглядеть, как показано в примере 3.9.

Пример 3-9. *playbooks/webservers-tls.yml*

```
---
- name: Configure webserver with Nginx and TLS
  hosts: webservers
  become: true
  gather_facts: false

  vars:
    tls_dir: /etc/nginx/ssl/
```

```
    key_file: nginx.key
    cert_file: nginx.crt
    conf_file: /etc/nginx/sites-available/default
    server_name: localhost

  handlers:
    - name: Restart nginx
      service:
        name: nginx
        state: restarted

  tasks:
    - name: Ensure nginx is installed
      package:
        name: nginx
        update_cache: true
      notify: Restart nginx

    - name: Create directories for TLS certificates
      file:
        path: "{{ tls_dir }}"
        state: directory
        mode: '0750'
      notify: Restart nginx

    - name: Copy TLS files
      copy:

        src: "{{ item }}"
        dest: "{{ tls_dir }}"
        mode: '0600'
      loop:
        - "{{ key_file }}"
        - "{{ cert_file }}"
      notify: Restart nginx

    - name: Manage nginx config template
      template:
        src: nginx.conf.j2
        dest: "{{ conf_file }}"
        mode: '0644'
      notify: Restart nginx

    - name: Enable configuration
      file:
        src: /etc/nginx/sites-available/default
        dest: /etc/nginx/sites-enabled/default
```

```
        state: link

    - name: Install home page
      template:
        src: index.html.j2
        dest: /usr/share/nginx/html/index.html
        mode: '0644'

    - name: Restart nginx
      meta: flush_handlers

    - name: "Test it! https://localhost:8443/index.html"
      delegate_to: localhost
      become: false
      uri:
        url: 'https://localhost:8443/index.html'
        validate_certs: false
        return_content: true
      register: this
      failed_when: "'Running on ' not in this.content"
      tags:
        - test
...
```

Запуск сценария

Запуск сценария выполняется командой `ansible-playbook`:

```
$ ansible-playbook webservers-tls.yml
```

Вывод должен выглядеть примерно так:

```
PLAY [Configure webserver with Nginx and TLS] **********************************

TASK [Ensure nginx is installed] ***********************************************
ok: [testserver]

TASK [Create directories for TLS certificates] *********************************
changed: [testserver]

TASK [Copy TLS files] **********************************************************
changed: [testserver] => (item=nginx.key)
changed: [testserver] => (item=nginx.crt)

TASK [Manage nginx config template] ********************************************
changed: [testserver]

TASK [Install home page] *******************************************************
```

```
ok: [testserver]

RUNNING HANDLER [Restart nginx] ************************************************
changed: [testserver]

TASK [Test it! https://localhost:8443/index.html] ******************************
ok: [testserver]

PLAY RECAP *********************************************************************
testserver : ok=7 changed=4 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

Откройте в браузере страницу *https://localhost:8443* (не забудьте «s» в конце *https*). Если вы используете Chrome, то, как и я, получите неприятное сообщение о том, что «установленное соединение не защищено» (рис. 3.3).

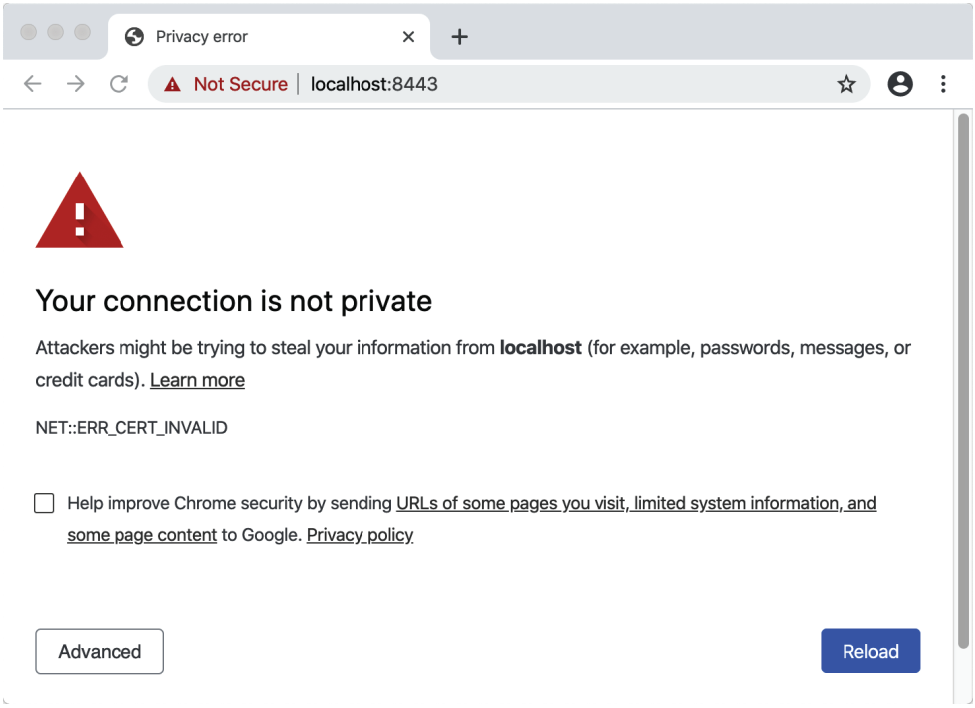


Рис. 3.3. Некоторые браузеры, такие как Chrome, не доверяют «самоподписанным» сертификатам TLS

Не беспокойтесь. Ошибка ожидаема, поскольку мы создали «самоподписанный» сертификат TLS. А такие браузеры, как Chrome, доверяют только сертификатам, выпущенным доверенным центром сертификации.

Shebang

Когда в Unix-подобной операционной системе текстовый файл имеет разрешение на выполнение, то мы называем его сценарием. Если первая строка в файле начинается с двух символов `#!`, то механизм загрузки программы интерпретирует оставшуюся часть первой строки как директиву, определяющую интерпретатор, который следует вызвать для обработки сценария. Он запустит интерпретатор и передаст ему сценарий как аргумент. Мы изменили разрешения для нашего сценария (*webservers-tls.yml*), объявив содержащий его файл выполняемым, и запустили файл со следующей строкой *shebang*. (Символ `#` без `!` – это просто комментарий.)

```
#!/usr/bin/env ansible-playbook
# Этот сценарий теперь будет выполняться с помощью ansible-playbook.
---
```

Заключение

В этой главе мы изучили многое из того, *что* делает Ansible с хостами. Обработчики – лишь одна из форм управления потоком выполнения, поддерживаемых в Ansible. В главе 9 мы рассмотрим циклическое и условное выполнение задач на основе значений переменных. В следующей главе мы также поговорим об аспекте *кто*. Другими словами, как описать хосты, на которых должны выполняться сценарии.

Глава 4

Реестр: описание серверов

До настоящего момента мы рассматривали работу лишь с одним сервером (или *хостом* в терминологии Ansible). Реестр в простейшем виде – это список имен хостов, перечисленных через запятую, который может даже не содержать внешних серверов:

```
$ ansible all -i 'localhost,' -a date
```

В действительности вам предстоит управлять многими хостами. Группа хостов, данными о которых располагает Ansible, называется *реестром* (inventory). В этой главе вы узнаете, как составить реестр, описывающий группу хостов.

В настоящий момент наш файл *ansible.cfg* должен выглядеть, как показано в примере 4.1, и включать все плагины поддержки реестра.

Пример 4.1. ansible.cfg

```
[defaults]
inventory = inventory

[inventory]
enable_plugins = host_list, script, auto, yaml, ini, toml
```

В этой главе все примеры реестров мы будем сохранять в каталоге *inventory*. Реестр Ansible – очень гибкий объект: это может быть текстовый файл (в нескольких форматах), каталог или выполняемый файл, причем некоторые выполняемые файлы поставляются в виде плагинов. Плагины поддержки реестра позволяют указать источник данных, например поставщика облачных услуг, для составления реестра.

Серж ван Гиндерахтер (Serge van Ginderachter) – самый авторитетный специалист по реестрам Ansible. Мы настоятельно рекомендуем почитать его блог (**https://oreil.ly/tUABr**).

Реестр может храниться отдельно от сценариев Ansible. Это означает, что можно создать единый каталог реестров для использования с Ansible в командной строке, содержащих хосты, работающих в Vagrant, Amazon EC2, Google Cloud Platform, Microsoft Azure и вообще где угодно!

Файл реестра

Самый простой способ описать имеющиеся хосты – перечислить их в текстовом файле, который принято называть *файлом реестра хостов*. В простейшем случае реестр – это файл *hosts*, содержащий список имен хостов, как показано в примере 4.2.

Пример 4.2. Простейший файл реестра

```
frankfurt.example.com
helsinki.example.com
hongkong.example.com
johannesburg.example.com
london.example.com
newyork.example.com
seoul.example.com
sydney.example.com
```

Система Ansible автоматически добавляет в реестр хост *localhost*. Она понимает, что имя `localhost` ссылается на локальную машину, поэтому будет взаимодействовать с ней напрямую, минуя SSH-соединение.

Вводная часть: несколько машин Vagrant

Для обсуждения приемов работы с реестром нам потребуется несколько хостов. Давайте настроим в Vagrant три хоста и назовем их `vagrant1`, `vagrant2` и `vagrant3`.

Прежде чем вносить изменения в файл Vagrantfile, не забудьте удалить существующую виртуальную машину, выполнив команду:

```
$ vagrant destroy --force
```

Если запустить эту команду без флага `--force`, то Vagrant предложит подтвердить удаление каждой виртуальной машины из перечисленных в Vagrantfile.

После этого измените файл Vagrantfile, как показано в примере 4.3.

Пример 4.3. Vagrantfile с тремя серверами

```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Использовать один и тот же ключ для всех машин
  config.ssh.insert_key = false
```

```
  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/focal64"
    vagrant1.vm.network "forwarded_port", guest: 80, host: 8080
    vagrant1.vm.network "forwarded_port", guest: 443, host: 8443
  end
  config.vm.define "vagrant2" do |vagrant2|
    vagrant2.vm.box = "ubuntu/focal64"
    vagrant2.vm.network "forwarded_port", guest: 80, host: 8081
    vagrant2.vm.network "forwarded_port", guest: 443, host: 8444
  end
  config.vm.define "vagrant3" do |vagrant3|
    vagrant3.vm.box = "centos/stream8"
    vagrant3.vm.network "forwarded_port", guest: 80, host: 8082
    vagrant3.vm.network "forwarded_port", guest: 443, host: 8445
  end
end
```

Начиная с версии 1.7, Vagrant по умолчанию использует для каждого хоста свой SSH-ключ. В примере 4.3 присутствует строка, которая возвращает Vagrant к использованию одного SSH-ключа для всех хостов:

```
config.ssh.insert_key = false
```

Использование одного и того же ключа для всех хостов упрощает настройку Ansible, поскольку в этом случае требуется указать в конфигурации только один SSH-ключ.

Теперь предположим, что каждый из этих серверов потенциально может быть веб-сервером, поэтому в примере 4.3 порты 80 и 443 на каждой машине Vagrant отображены в порты локальной машины.

Виртуальные машины запускаются командой:

```
$ vagrant up
```

Если все в порядке, она выведет следующее:

```
Bringing machine 'vagrant1' up with 'virtualbox' provider...
Bringing machine 'vagrant2' up with 'virtualbox' provider...
Bringing machine 'vagrant3' up with 'virtualbox' provider...
...
    vagrant1: 80 (guest) => 8080 (host) (adapter 1)
    vagrant1: 443 (guest) => 8443 (host) (adapter 1)
    vagrant1: 22 (guest) => 2222 (host) (adapter 1)
==> vagrant1: Running 'pre-boot' VM customizations...
==> vagrant1: Booting VM...
==> vagrant1: Waiting for machine to boot. This may take a few minutes...
    vagrant1: SSH address: 127.0.0.1:2222
    vagrant1: SSH username: vagrant
    vagrant1: SSH auth method: private key
```

```
==> vagrant1: Machine booted and ready!
==> vagrant1: Checking for guest additions in VM...
==> vagrant1: Mounting shared folders...
    vagrant1: /vagrant => /Users/bas/code/ansible/ansiblebook/ansiblebook/ch03
```

Далее давайте посмотрим, какие порты локальной машины отображены в порт SSH (22) каждой виртуальной машины. Напомним, что эти данные можно получить командой:

```
$ vagrant ssh-config
```

Результат должен выглядеть примерно так:

```
Host vagrant1
  HostName 127.0.0.1
  User vagrant
  Port 2222
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /Users/lorin/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
Host vagrant2
  HostName 127.0.0.1
  User vagrant
  Port 2200
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /Users/lorin/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
Host vagrant3
  HostName 127.0.0.1
  User vagrant
  Port 2201
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /Users/lorin/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

Большая часть информации в выводе `ssh-config` повторяется, и ее можно сократить. Отличаются только номера портов на локальной машине, в которые отображаются порты виртуальных машин. Так, для `vagrant1` используется порт 2222, для `vagrant2` – порт 2200 и для `vagrant3` – порт 2201.

По умолчанию Ansible использует локального клиента SSH, т. е. она будет понимать любые псевдонимы, настроенные в файле конфигурации SSH. Поэтому мы используем подстановочный знак в файле *~/.ssh/config*:

```
Host vagrant*
  Hostname 127.0.0.1
  User vagrant
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile ~/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

Измените файл *hosts*, как показано ниже:

```
vagrant1 ansible_port=2222
vagrant2 ansible_port=2200
vagrant3 ansible_port=2201
```

Теперь проверим доступность этих машин. Например, получить информацию о сетевом интерфейсе в `vagrant2` можно командой:

```
$ ansible vagrant2 -a "ip addr show dev enp0s3"
```

Она должна вывести примерно следующее:

```
vagrant2 | CHANGED | rc=0 >>
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
group default qlen 1000
    link/ether 02:1e:de:45:2c:c8 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
       valid_lft 86178sec preferred_lft 86178sec
    inet6 fe80::1e:deff:fe45:2cc8/64 scope link
       valid_lft forever preferred_lft forever
```

Поведенческие параметры хостов в реестре

Для описания машин Vagrant в файле реестра Ansible необходимо явно указать порт (2222, 2200 или 2201), к которому будет подключаться SSH-клиент системы Ansible. В Ansible эти переменные называются *поведенческими параметрами (behavioral parameters)*. Некоторые из них можно использовать для изменения значений по умолчанию (табл. 4.1).

Таблица 4.1. Поведенческие параметры

Имя	Значение по умолчанию	Описание
`ansible_host`	Имя хоста	Имя хоста или IP-адрес
`ansible_port`	22	Порт для подключения по протоколу SSH

Имя	Значение по умолчанию	Описание
ansible_user	$USER	Пользователь для подключения по протоколу SSH
ansible_password	(нет)	Пароль для подключения по протоколу SSH
ansible_connection	smart	Как Ansible будет подключаться к хосту (см. следующий раздел)
ansible_ssh_private_key_file	(нет)	Закрытый SSH-ключ для аутентификации по протоколу SSH
ansible_shell_type	sh	Командная оболочка для выполнения команд (см. следующий раздел)
ansible_python_interpreter	/usr/bin/python	Путь к интерпретатору Python на хосте (см. следующий раздел)
ansible_*_interpreter	(нет)	Аналоги ansible_python_interpreter для других языков (см. следующий раздел)

Назначение некоторых параметров очевидно из их названий, другие требуют дополнительных пояснений.

`ansible_connection`

Ansible поддерживает несколько *транспортов* – механизмов подключения к хостам. По умолчанию используется транспорт `smart`. Он проверяет поддержку локальным SSH-клиентом функции `ControlPersist`. Если SSH-клиент поддерживает ее, то Ansible будет использовать локального SSH-клиента. Если локальный клиент не поддерживает `ControlPersist`, тогда транспорт `smart` будет использовать библиотеку SSH-клиента на Python с названием *Paramiko*.

`ansible_shell_type`

Ansible устанавливает SSH-соединения с удаленными машинами и затем запускает на них сценарии. По умолчанию Ansible считает, что на удаленных машинах используется командная оболочка Bourne Shell, доступная как */bin/sh*, и создает параметры командной строки, соответствующие оболочке Bourne Shell.

В этом параметре можно также передать значение `csh`, `fish` или `powershell` (при работе с Windows). При этом имейте в виду, что Ansibly не поддерживает ограниченные командные оболочки.

`ansible_python_interpreter`

Модули, входящие в состав Ansible, реализованы на Python, поэтому Ansible должна знать местоположение интерпретатора Python на удаленной машине. Вам может потребоваться изменить эту переменную, чтобы указать путь к требуемой версии интерпре-

татора. Самый простой способ запустить Ansible под управлением Python 3 – установить ее с помощью `pip3` и настроить данный параметр, как показано ниже:

```
ansible_python_interpreter="/usr/bin/env python3"
```

`ansible_*_interpreter`

Если вы собираетесь использовать нестандартные модули, написанные не на Python, используйте этот параметр, чтобы определить путь к интерпретатору (например, */usr/bin/ruby*). Подробнее об этом мы поговорим в главе 12.

Переопределение значений по умолчанию в поведенческих параметрах

Переопределить значения по умолчанию некоторых поведенческих параметров можно в секции `[defaults]` файла *ansible.cfg* (табл. 4.2). Подумайте, где лучше сделать это. Являются ли изменения личным выбором или они касаются всей вашей команды? Потребуется ли часть вашего реестра для настройки другого окружения? Напомним, что настроить параметры SSH можно в файле *~/.ssh/config*.

Таблица 4.2. Значения по умолчанию, которые можно изменить в *ansible.cfg*

Поведенческий параметр	Параметр в файле ansible.cfg
`ansible_port`	`remote_port`
`ansible_user`	`remote_user`
`ansible_ssh_private_key_file`	`ssh_private_key_file`
`ansible_shell_type`	`executable` (см. следующий абзац)

Параметр `executable` в файле *ansible.cfg* не совсем то же самое, что поведенческий параметр `ansible_shell_type`. Параметр `executable` определяет полный путь к используемой командной оболочке на удаленной машине (например, */usr/local/bin/fish*). Ansible выбирает имя в конце этого пути (для */usr/local/bin/fish* это будет имя *fish*) и использует его как значение по умолчанию для `ansible_shell_type`.

Группы, группы и еще раз группы

Занимаясь настройками, мы обычно совершаем действия не с одним хостом, а с их группой. Ansible автоматически определяет группу `all` (или `*`). Она включает все хосты, перечисленные в реестре. Например, мы можем примерно оценить синхронность хода часов на машинах с помощью команды:

```
$ ansible all -a "date"
```

или

```
$ ansible '*' -a "date"
```

На компьютере Баса она вывела следующее:

```
vagrant2 | CHANGED | rc=0 >>
Wed 12 May 2021 01:37:47 PM UTC
vagrant1 | CHANGED | rc=0 >>
Wed 12 May 2021 01:37:47 PM UTC
vagrant3 | CHANGED | rc=0 >>
Wed 12 May 2021 01:37:47 PM UTC
```

В файле реестра можно определять свои группы. Файлы реестра в Ansible оформляются в формате *.ini*, в котором параметры группируются в секции.

Вот как можно объединить в группу `vagrant` наши Vagrant-хосты наряду с другими хостами из примера, приводившегося в начале главы:

```
frankfurt.example.com
helsinki.example.com
hongkong.example.com
johannesburg.example.com
london.example.com
newyork.example.com
seoul.example.com
sydney.example.com

[vagrant]
vagrant1 ansible_port=2222
vagrant2 ansible_port=2200
vagrant3 ansible_port=2201
```

Также можно было бы перечислить Vagrant-хосты в начале файла и потом объединить их в группу:

```
frankfurt.example.com
helsinki.example.com
hongkong.example.com
johannesburg.example.com
london.example.com
newyork.example.com
seoul.example.com
sydney.example.com
vagrant1 ansible_port=2222
vagrant2 ansible_port=2200
vagrant3 ansible_port=2201
```

```
[vagrant]
vagrant1
vagrant2
vagrant3
```

Группы можно определять, как вам заблагорассудится: они могут пересекаться или быть вложенными. Порядок следования групп не имеет значения, главный критерий – удобочитаемость.

Пример: развертывание приложения Django

Представьте, что вы отвечаете за развертывание веб-приложения, реализованного на основе фреймворка Django и выполняющего продолжительные операции. Чтобы развернуть приложение, на хосте должны также присутствовать:

- последняя версия самого веб-приложения Django, выполняемого HTTP-сервером Gunicorn;
- веб-сервер NGINX, находящийся перед сервером Gunicorn и обслуживающий статические ресурсы;
- очередь задач Celery, выполняющая продолжительные операции от лица веб-сервера;
- диспетчер очередей сообщений RabbitMQ, обеспечивающий работу Celery;
- база данных Postgres, используемая в качестве хранилища.

В последующих главах мы подробно рассмотрим пример развертывания Django-приложения такого типа. Но в том примере не будут использоваться Celery и RabbitMQ. Также представьте, что данное приложение необходимо развернуть в разных окружениях: промышленном (для реального использования), тестовом (для тестирования на хостах, к которым члены нашей команды имеют доступ) и в Vagrant (для локального тестирования).

В промышленном окружении необходимо обеспечить быстрый и надежный отклик системы, поэтому мы:

- запустим веб-приложение на нескольких хостах и поставим перед ними балансировщик нагрузки;
- запустим серверы очередей задач на нескольких хостах;
- установим Gunicorn, Celery, RabbitMQ и Postgres на отдельных серверах;
- используем два хоста для размещения основной базы данных Postgres и ее копии.

Допустим, что у нас имеются один балансировщик нагрузки, три веб-сервера, три очереди задач, один сервер RabbitMQ и два сервера баз данных, т. е. всего 10 хостов (рис. 4.1).

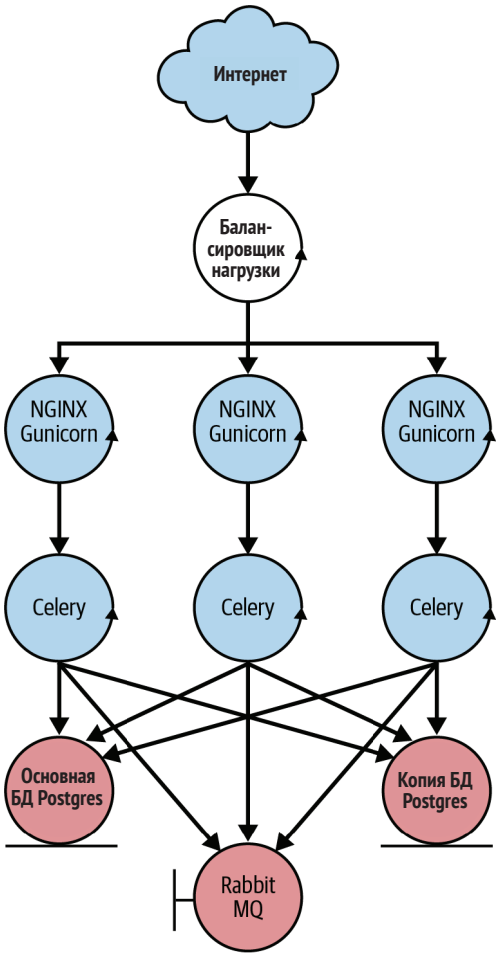


Рис. 4.1. Десять хостов, на которых предполагается развернуть приложение Django

Представим также, что в окружении для обкатки (staging) мы решили использовать меньше хостов, чем в промышленном окружении. Это позволит сократить издержки, поскольку нагрузка на окружение обкатки будет существенно ниже. Допустим, мы решили настроить в тестовом окружении всего два хоста. Мы установим веб-сервер и диспетчер очереди задач на один хост, а RabbitMQ и Postgres – на другой.

В локальном окружении Vagrant мы решили использовать три сервера: один – для веб-приложения, второй – для диспетчера очереди задач, третий – для установки RabbitMQ и Postgres.

В примере 4.4 представлен вариант возможного файла реестра, в котором наши серверы сгруппированы по принадлежности к окружению (промышленному, тестовому, Vagrant) и по функциональности (веб-сервер, диспетчер очереди задач и т. д.).

Пример 4.4. Файл реестра для развертывания приложения Django

```
[production]
frankfurt.example.com
helsinki.example.com
hongkong.example.com
johannesburg.example.com
london.example.com
newyork.example.com
seoul.example.com
sydney.example.com
tokyo.example.com
toronto.example.com

[staging]
amsterdam.example.com
chicago.example.com

[lb]
helsinki.example.com

[web]
amsterdam.example.com
seoul.example.com
sydney.example.com
toronto.example.com
vagrant1

[task]
amsterdam.example.com
hongkong.example.com
johannesburg.example.com
newyork.example.com
vagrant2

[rabbitmq]
chicago.example.com
tokyo.example.com
vagrant3

[db]
chicago.example.com
frankfurt.example.com
london.example.com
vagrant3
```

Мы могли бы сначала перечислить все серверы в начале файла, не определяя группы, но в этом нет необходимости, и это сделало бы файл только длиннее.

Обратите внимание, что нам понадобилось только один раз указать поведенческие параметры для экземпляров Vagrant.

Псевдонимы и порты

Мы описали наши хосты Vagrant так:

```
[vagrant]
vagrant1 ansible_port=2222
vagrant2 ansible_port=2200
vagrant3 ansible_port=2201
```

Имена `vagrant1`, `vagrant2`, `vagrant3` – это *псевдонимы*. Они не настоящие имена серверов, но их удобно использовать для обозначения хостов. Разрешение имен в Ansible осуществляется с использованием реестра, конфигурационного файла SSH, файла */etc/hosts* и DNS. Эта гибкость полезна при разработке, но может вызвать путаницу.

Ansible поддерживает синтаксис `<hostname>:<port>` описания хостов. То есть строку с описанием `vagrant1` *можно* заменить объявлением `127.0.0.1:2222` (пример 4.5).

Пример 4.5. Этот реестр не работает

```
[vagrant]
127.0.0.1:2222
127.0.0.1:2200
127.0.0.1:2201
```

Однако нам не удастся задействовать гипотетический реестр, представленный в примере 4.5, потому что с IP-адресом *127.0.0.1* можно определить только один хост, поэтому группа `vagrant` в этом случае может содержать лишь один хост.

Группировка групп

Ansible позволяет также определять группы, состоящие из других групп. Например, на веб-серверы и на серверы очередей требуется установить фреймворк Django и его зависимости. Поэтому будет полезно определить группу `django`, включающую обе вышеуказанные группы. Для этого достаточно добавить следующие строки в файл реестра:

```
[django:children]
web
task
```

Обратите внимание, что для определения группы групп используется другой синтаксис, отличный от синтаксиса определения группы хостов. Благодаря этому Ansible поймет, что `web` и `task` – это группы, а не хосты.

Имена хостов с номерами (домашние питомцы и стадо)

Файл реестра в примере 4.4 выглядит достаточно сложным. На самом деле он описывает всего лишь 15 разных хостов. А это количество не так уж и велико в нашем облачном безразмерном мире. Тем не менее даже 15 хостов в файле реестра могут вызывать затруднения, потому что каждый хост имеет свое, уникальное имя.

Билл Бейкер (Bill Baker) из Microsoft выделил отличительные особенности управления серверами, которые интерпретируются как *домашние питомцы* и как *стадо*[1]. Своим домашним питомцам мы даем отличительные имена и работаем с ними в индивидуальном порядке, но животных в стаде мы часто идентифицируем по их номерам.

Подход к именованию серверов с использованием нумерации более масштабируемый, и Ansible с легкостью поддерживает его посредством числовых шаблонов. Например, если у вас имеется 20 серверов с именами *web1.example.com*, *web2.example.com* и т. д., то вы можете описать их в файле реестра так:

```
[web]
web[1:20].example.com
```

Если вы предпочитаете использовать ведущие нули (например, *web01.example.com*), укажите их в определении диапазона :

```
[web]
web[01:20].example.com
```

Ansile поддерживает также возможность определения диапазонов букв. Если вы предпочитаете использовать условные обозначения *web-a.example.com*, *web-b.example.com* и т. д., тогда поступите так (для тех же 20 серверов):

```
[web]
web-[a-t].example.com
```

Переменные хостов и групп: внутренняя сторона реестра

Вспомните, как мы определили поведенческие параметры для хостов Vagrant:

[1] Этот термин предложил Рэнди Биас (Randy Bias) из Cloudscaling (https://oreil.ly/Zsvdf).

```
vagrant1 ansible_host=127.0.0.1 ansible_port=2222
vagrant2 ansible_host=127.0.0.1 ansible_port=2200
vagrant3 ansible_host=127.0.0.1 ansible_port=2201
```

Эти параметры являются переменными, имеющими особое значение для Ansible. Точно так же можно задать переменные с произвольными именами и соответствующие значения для разных хостов. Например, можно определить переменную `color` и присвоить ей уникальное значение для каждого сервера:

```
amsterdam.example.com color=red
seoul.example.com color=green
sydney.example.com color=blue
toronto.example.com color=purple
```

Эту переменную затем можно использовать в сценарии как любую другую. Авторы книги редко закрепляют переменные за отдельными хостами, но часто связывают переменные с группами.

В примере с Django веб-приложению и диспетчеру очереди необходимо взаимодействовать с RabbitMQ и Postgres. Предположим, доступ к базе данных Postgres защищен на сетевом уровне (только веб-приложение и диспетчер очереди задач могут использовать базу данных) и на уровне учетных данных. Доступ к RabbitMQ защищен при этом только на сетевом уровне.

Для приведения такой системы в рабочее состояние необходимо настроить:

- на веб-серверах: имя хоста, порт, имя пользователя и пароль основного сервера Postgres, а также имя базы данных;
- на сервере диспетчера очереди: имя хоста, порт, имя пользователя и пароль основного сервера Postgres, а также имя базы данных;
- на веб-серверах: имя хоста и порт сервера RabbitMQ;
- на сервере диспетчера очереди: имя хоста и порт сервера RabbitMQ;
- на основном сервере Postgres: имя хоста, порт, имя пользователя и пароль сервера копии базы данных Postgres (только в промышленном окружении).

Информация о конфигурации зависит от окружения, поэтому имеет смысл определить групповые переменные для промышленной, тестовой и Vagrant групп. В примере 4.6 показан один из вариантов объявления этой информации в виде переменных групп в файле реестра. (В главе 8 будет показан более удачный способ хранения паролей.)

Пример 4.6. Определение переменных групп в реестре

```
[all:vars]
ntp_server=ntp.ubuntu.com
[production:vars]
db_primary_host=frankfurt.example.com
db_primary_port=5432
db_replica_host=london.example.com
db_name=widget_production
db_user=widgetuser
db_password=pFmMxcyD;Fc6)6
rabbitmq_host=johannesburg.example.com
rabbitmq_port=5672

[staging:vars]
db_primary_host=chicago.example.com
db_primary_port=5432
db_name=widget_staging
db_user=widgetuser
db_password=L@4Ryz8cRUXedj
rabbitmq_host=chicago.example.com
rabbitmq_port=5672

[vagrant:vars]
db_primary_host=vagrant3
db_primary_port=5432
db_name=widget_vagrant
db_user=widgetuser
db_password=password
rabbitmq_host=vagrant3
rabbitmq_port=5672
```

Обратите внимание, что переменные групп объединяются в секции с именами `[<имя группы>:vars]`. Также отметьте, что мы воспользовались группой `all`, которую Ansible создает автоматически, чтобы определить переменные для всех хостов.

Переменные хостов и групп: создание собственных файлов

Если у вас не слишком много хостов, переменные можно поместить в файл реестра. Но с увеличением объема информации становится все сложнее управлять переменными таким способом. Кроме того, хотя переменные Ansible могут хранить логические и строковые значения, списки и словари, в файле реестра допускается задавать только логические и строковые значения.

Ansible поддерживает более масштабируемый подход к управлению переменными. Вы можете создать отдельный файл с переменными для каждого хоста и каждой группы. Такие файлы переменных должны иметь формат YAML.

Ansible проверяет наличие файлов переменных хостов в каталоге *host_vars* и файлов переменных групп в каталоге *group_vars*. Эти каталоги должны находиться в каталоге со сценарием или в каталоге с реестром. Если имеются оба каталога, то каталог со сценарием просматривается первым, а каталог с реестром – вторым.

К примеру, допустим, что Лорин хранит сценарии в каталоге */home/lorin/playbooks/*, а файл реестра – в каталоге */home/lorin/inventory/hosts*, тогда он должен был бы сохранить переменные для хоста *amsterdam.example.com* в файле */home/lorin/inventory/host_vars/amsterdam.example.com*, а переменные для группы хостов в промышленном окружении – в файле */home/lorin/inventory/group_vars/production* (пример 4.7).

Пример 4.7. *group_vars/production*

```
---
db_primary_host: frankfurt.example.com
db_primary_port: 5432
db_replica_host: london.example.com
db_name: widget_production
db_user: widgetuser
db_password: 'pFmMxcyD;Fc6)6'
rabbitmq_host: johannesburg.example.com
rabbitmq_port: 5672
...
```

Для представления этих значений также можно использовать словари YAML, как показано в примере 4.8.

Пример 4.8. *group_vars/production*, со словарями

```
---
db:
  user: widgetuser
  password: 'pFmMxcyD;Fc6)6'
  name: widget_production
  primary:
    host: frankfurt.example.com
    port: 5432
  replica:
    host: london.example.com
    port: 5432
rabbitmq:
  host: johannesburg.example.com
```

```
  port: 5672
...
```

При использовании словарей YAML доступ к переменным должен производиться с помощью точечной нотации:

```
"{{ db.primary.host }}"
```

Также к переменным в словаре можно обращаться так:

```
"{{ db['primary']['host'] }}"
```

Сравните эти два приема с тем, как мы должны были бы обращаться к переменным в противном случае:

```
"{{ db_primary_host }}"
```

При желании можно продолжить разбивку информации. Ansible позволяет определить *group_vars/production* как каталог и поместить сюда несколько файлов YAML с определениями переменных. Например, переменные, описывающие базу данных, можно поместить в один файл, а переменные, описывающие RabbitMQ, – в другой, как показано в примерах 4.9 и 4.10.

Пример 4.9. *group_vars/production/db*

```
---
db:
  user: widgetuser
  password: 'pFmMxcyD;Fc6)6'
  name: widget_production
  primary:
    host: frankfurt.example.com
    port: 5432
  replica:
    host: london.example.com
    port: 5432
...
```

Пример 4.10. *group_vars/production/rabbitmq*

```
---
rabbitmq:
  host: johannesburg.example.com
  port: 6379
...
```

В общем и целом лучше не усложнять и не разбивать переменные на слишком большое количество файлов. Однако в больших командах и проектах ценность отдельных файлов возрастает, так как многим может потребоваться извлекать и работать с файлами одновременно.

Динамический реестр

До настоящего момента мы явно описывали наши хосты в файле реестра. Однако вам может понадобиться хранить всю информацию о хостах во внешней системе. Например, если хосты располагаются в облаке Amazon EC2, то вся информация о них будет храниться в EC2, и вы сможете извлекать ее посредством веб-интерфейса EC2, Query API или с помощью инструмента командной строки, такого как `awscli`. Другие облачные провайдеры поддерживают похожие интерфейсы.

Если вы управляете своими собственными серверами, используя автоматизированную систему инициализации, такую как Cobbler или Ubuntu Metal as a Service (MAAS), то она уже отслеживает ваши серверы. Или, может быть, вся ваша информация хранится в одной из тех причудливых баз данных управления конфигурациями (Configuration Management DataBases, CMDB).

В этом случае вам не придется вручную копировать информацию в файл реестра, поскольку в конечном счете этот файл не будет соответствовать содержимому внешней системы – подлинного источника данных о ваших хостах. Ansible поддерживает функцию *динамического реестра*, которая позволяет избежать копирования.

Если файл реестра отмечен как выполняемый, то Ansible будет интерпретировать его как сценарий динамического реестра и запускать его вместо чтения.

Сделать файл выполняемым можно командой `chmod +x`. Например:

```
$ chmod +x vagrant.py *
```

Плагины поддержки реестров

В состав Ansible входит несколько выполняемых файлов, которые могут подключаться к различным облачным системам при условии, что вы установите все необходимое ПО и настроите аутентификацию. Этим плагинам обычно требуется конфигурационный файл YAML в каталоге *inventory*, а также некоторые переменные окружения или файлы аутентификации.

Получить список доступных плагинов можно командой:

```
$ ansible-doc -t inventory -l
```

А документацию с описанием конкретного плагина – командой:

```
$ ansible-doc -t inventory <имя плагина>
```

Амазон EC2

Если вы используете Amazon EC2, то установите необходимые зависимости:

```
$ pip3 install boto3 botocore
```

Создайте файл *inventory/aws_ec2.yml*, содержащий хотя бы одну строку:

```
plugin: aws_ec2
```

Диспетчер ресурсов Azure

Установите следующие зависимости в виртуальное окружение Python (virtualenv) с Ansible 2.9.xx:

```
$ pip3 install msrest msrestazure
```

Создайте файл *inventory/azure_rm.yml*, содержащий хотя бы следующие строки:

```
plugin: azure_rm
platform: azure_rm
auth_source: auto
plain_host_names: true
```

Интерфейс сценария динамического реестра

Сценарий динамического реестра должен поддерживать два параметра командной строки:

- `--host=<имя хоста>` для вывода информации о хостах;
- `--list` для вывода информации о группах.

Также он должен возвращать вывод в формате JSON со структурой, которую Ansible сможет проанализировать.

Вывод информации о хосте

Чтобы получить данные о конкретном хосте, Ansible вызывает сценарий динамического реестра с аргументом `--host=`:

```
$ ansible-inventory -i inventory/hosts --host=vagrant2
```

В составе Ansible имеется сценарий `ansible-inventory`, который действует как сценарий динамического реестра, но извлекает информацию из статического реестра, переданного в аргументе командной строки `-i`.

Вывод сценария должен содержать переменные для заданного хоста, включая поведенческие параметры, например:

```
{
    "ansible_host": "127.0.0.1",
    "ansible_port": 2200,
    "ansible_ssh_private_key_file": "~/.vagrant.d/insecure_private_key",
    "ansible_user": "vagrant"
}
```

Результаты выводятся в виде объекта JSON, имена свойств в котором соответствуют именам переменных, а значения – значениям этих переменных.

Вывод списка членов групп

Сценарий динамического реестра должен уметь выводить списки членов всех групп, а также данные об отдельных хостах. В репозитории GitHub (**https://oreil.ly/vselj**) с примерами для этой книги есть сценарий реестра для хостов Vagrant, который называется *vagrant.py*. Чтобы получить содержимое всех групп, Ansible вызовет его командой:

```
$ ./vagrant.py --list
```

Результат должен выглядеть так:

```
{"vagrant": ["vagrant1", "vagrant2", "vagrant3"]}
```

Результат выводится в виде единого объекта JSON, имена свойств в котором соответствуют именам групп, а значения – это массивы с именами хостов.

Для оптимизации команда `--list` должна поддерживать вывод всех переменных всех хостов. Это освобождает Ansible от необходимости повторно вызывать сценарий с параметром `--host`, чтобы получить переменные отдельных хостов.

Для этого команда `--list` должна возвращать ключ `_meta` с переменными всех хостов, как показано ниже:

```
"_meta": {
    "hostvars": {
      "vagrant1": {
        "ansible_user": "vagrant",
        "ansible_host": "127.0.0.1",
        "ansible_ssh_private_key_file": "~/.vagrant.d/insecure_private_key",
        "ansible_port": "2222"
      },
      "vagrant2": {
        "ansible_user": "vagrant",
        "ansible_host": "127.0.0.1",
```

```
    "ansible_ssh_private_key_file": "~/.vagrant.d/insecure_private_key",
    "ansible_port": "2200"
  },
  "vagrant3": {
    "ansible_user": "vagrant",
    "ansible_host": "127.0.0.1",
    "ansible_ssh_private_key_file": "~/.vagrant.d/insecure_private_key",
    "ansible_port": "2201"
  }
}
```

Написание сценария динамического реестра

Одной из удобных функций Vagrant является возможность получить список запущенных виртуальных машин командой `vagrant status`. Допустим, у нас имеется файл Vagrantfile, как показано в примере 4.3. Если запустить команду `vagrant status`, то результат будет выглядеть, как в примере 4.11:

Пример 4.11. Вывод команды vagrant status

```
$ vagrant status
Current machine states:

vagrant1                  running (virtualbox)
vagrant2                  running (virtualbox)
vagrant3                  running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run 'vagrant status NAME'.
```

Поскольку Vagrant уже хранит информацию о состоянии машин, нет необходимости вносить их список в файл реестра. Вместо этого можно написать сценарий динамического реестра, который запрашивает у Vagrant данные о запущенных на данный момент машинах. В этом случае нам не нужно будет вносить обновления в файл реестра, даже если число машин в Vagrantfile изменится.

Рассмотрим пример создания сценария динамического реестра, который извлекает данные о хостах из Vagrant. Наш сценарий будет получать необходимую информацию, выполняя команду `vagrant status`. Ее вывод, который приводится в примере 4.11, предназначен для людей, а не машин. Чтобы получить список запущенных хостов в формате, подходящем для анализа машиной, нужно добавить в команду параметр `--machine-readable`:

```
$ vagrant status --machine-readable
```

Результат выглядит так:

```
1620831617,vagrant1,metadata,provider,virtualbox
1620831617,vagrant2,metadata,provider,virtualbox
1620831618,vagrant3,metadata,provider,virtualbox
1620831619,vagrant1,provider-name,virtualbox
1620831619,vagrant1,state,running
1620831619,vagrant1,state-human-short,running
1620831619,vagrant1,state-human-long,The VM is running. To stop this
VM%!(VAGRANT_COMMA) you can run `vagrant halt` to\nshut it down
forcefully%!(VAGRANT_COMMA) or you can run `vagrant suspend` to
simply\nsuspend the virtual machine. In either case%!(VAGRANT_COMMA)
to restart it again%!(VAGRANT_COMMA)\nsimply run `vagrant up`.
1620831619,vagrant2,provider-name,virtualbox
1620831619,vagrant2,state,running
1620831619,vagrant2,state-human-short,running
1620831619,vagrant2,state-human-long,The VM is running. To stop this
VM%!(VAGRANT_COMMA) you can run `vagrant halt` to\nshut it down
forcefully%!(VAGRANT_COMMA) or you can run `vagrant suspend` to
simply\nsuspend the virtual machine. In either case%!(VAGRANT_COMMA)
to restart it again%!(VAGRANT_COMMA)\nsimply run `vagrant up`.
1620831620,vagrant3,provider-name,virtualbox
1620831620,vagrant3,state,running
1620831620,vagrant3,state-human-short,running
1620831620,vagrant3,state-human-long,The VM is running. To stop this
VM%!(VAGRANT_COMMA) you can run `vagrant halt` to\nshut it down
forcefully%!(VAGRANT_COMMA) or you can run `vagrant suspend` to
simply\nsuspend the virtual machine. In either case%!(VAGRANT_COMMA)
to restart it again%!(VAGRANT_COMMA)\nsimply run `vagrant up`.
1620831620,,ui,info,Current machine states:\n\nvagrant1
running (virtualbox)\nvagrant2         running (virtualbox)\nvagrant3
running (virtualbox)\n\nThis environment represents multiple VMs. The VMs
are all listed\nabove with their current state. For more information about
a specific\nVM%!(VAGRANT_COMMA) run `vagrant status NAME`
```

Получить информацию об отдельно взятой машине Vagrant, например `vagrant2`, можно командой

```
$ vagrant ssh-config vagrant2
```

Она выведет следующий результат:

```
Host vagrant2
  HostName 127.0.0.1
  User vagrant
  Port 2200
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
```

```
    IdentityFile /Users/lorin/.vagrant.d/insecure_private_key
    IdentitiesOnly yes
    LogLevel FATAL
```

Нашему сценарию динамического реестра необходимо вызвать эти команды, проанализировать результаты и вывести соответствующий текст в формате JSON. Для анализа результата команды `vagrant ssh-config` можно воспользоваться библиотекой Paramiko. Но сначала установите ее командой `pip`:

```
$ pip3 install --user paramiko
```

Ниже приводится интерактивный сеанс Python, объясняющий, как использовать Paramiko :

```
$ python3
>>> import io
>>> import subprocess
>>> import paramiko
>>> cmd = ["vagrant", "ssh-config", "vagrant2"]
>>> ssh_config = subprocess.check_output(cmd).decode("utf-8")
>>> config = paramiko.SSHConfig()
>>> config.parse(io.StringIO(ssh_config))
>>> host_config = config.lookup("vagrant2")
>>> print (host_config)
{'hostname': '127.0.0.1', 'user': 'vagrant', 'port': '2200', 'userknownhostsfile':
'/dev/null', 'stricthostkeychecking': 'no', 'passwordauthentication': 'no',
'identityfile': ['/Users/bas/.vagrant.d/insecure_private_key'], 'identitiesonly':
'yes', 'loglevel': 'FATAL'}
```

В примере 4.12 приводится полный исходный код сценария *vagrant.py*.

Пример 4.12. vagrant.py

```
#!/usr/bin/env python3
""" Сценарий динамического реестра Vagrant """
# Основан на реализации Марка Мандела (Mark Mandel)
# https://github.com/markmandel/vagrant_ansible_example

import argparse
import io
import json
import subprocess
import sys

import paramiko

def parse_args():
```

```
    """параметры командной строки"""
    parser = argparse.ArgumentParser(description="Vagrant inventory script")
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument('--list', action='store_true')
    group.add_argument('--host')
    return parser.parse_args()

def list_running_hosts():
    """Функция vagrant.py --list"""
    cmd = ["vagrant", "status", "--machine-readable"]
    status = subprocess.check_output(cmd).rstrip().decode("utf-8")
    hosts = []
    for line in status.splitlines():
        (_, host, key, value) = line.split(',')[:4]
        if key == 'state' and value == 'running':
            hosts.append(host)
    return hosts

def get_host_details(host):
    """Функция vagrant.py --host <имя хоста>"""
    cmd = ["vagrant", "ssh-config", host]
    ssh_config = subprocess.check_output(cmd).decode("utf-8")
    config = paramiko.SSHConfig()
    config.parse(io.StringIO(ssh_config))
    host_config = config.lookup(host)
    return {'ansible_host': host_config['hostname'],
            'ansible_port': host_config['port'],
            'ansible_user': host_config['user'],
            'ansible_private_key_file': host_config['identityfile'][0]}

def main():
    """главная функция"""
    args = parse_args()
    if args.list:
        hosts = list_running_hosts()
        json.dump({'vagrant': hosts}, sys.stdout)
    else:
        details = get_host_details(args.host)
        json.dump(details, sys.stdout)

if __name__ == '__main__':
    main()
```

Деление реестра на несколько файлов

Если вам необходим обычный файл реестра и сценарий динамического реестра (или их комбинация), то просто поместите их в один каталог и

настройте систему Ansible так, чтобы она использовала этот каталог как реестр. Это можно сделать двумя способами – добавив параметр `inventory` в *ansible.cfg* или включив параметр командной строки `-i`. Ansible обработает все файлы и объединит результаты в единый реестр.

Это означает, что вы сможете создать единый каталог реестра, обслуживаемый системой Ansible, включающий хосты в Vagrant, Amazon EC2, Google Cloud Platform, Microsoft Azure и вообще где угодно!

Например, вот как могла бы выглядеть структура такого каталога:

inventory/aws_ec2.yml
inventory/azure_rm.yml
inventory/group_vars/vagrant
inventory/group_vars/staging
inventory/group_vars/production
inventory/hosts
inventory/vagrant.py

Добавление элементов во время выполнения с помощью add_host и group_by

Ansible позволяет добавлять хосты и группы в реестр прямо во время выполнения сценария. Эта возможность может пригодиться тем, кто управляет динамическими кластерами, такими как Redis Sentinel.

add_host

Модуль `add_host` добавляет хост в реестр. Этот модуль можно использовать, например, для создания и настройки новых экземпляров виртуальных машин в облаке IaaS.

Может ли пригодиться модуль add_host при использовании динамического реестра?

Даже если вы используете сценарии динамического реестра, вам все равно может пригодиться модуль `add_host`, например чтобы запустить и настроить новый экземпляр виртуальной машины в ходе выполнения сценария Ansible.

Если новый хост появится во время выполнения сценария Ansible, то сценарий динамического реестра не подхватит его. Это объясняется тем, что создание динамического реестра производится в начале выполнения сценария, поэтому Ansible не увидит новых хостов, появившихся после.

Мы рассмотрим пример работы использования модуля `add_host` в главе 14.

Запуск модуля выглядит так:

```
- name: Add the host
  add_host
    name: hostname
    groups: web,staging
    myvar: myval
```

Определение списка групп и дополнительных переменных можно опустить.

Ниже показано практическое применение модуля `add_host`. Здесь он добавляет новую машину Vagrant и настраивает ее:

```
---
- name: Provision a Vagrant machine
  hosts: localhost
  vars:
    box: centos/stream8

  tasks:
    - name: Create a Vagrantfile
      command: "vagrant init {{ box }}"
      args:
        creates: Vagrantfile

    - name: Bring up the vagrant machine
      command: vagrant up
      args:
        creates: .vagrant/machines/default/virtualbox/box_meta

    - name: Add the vagrant machine to the inventory
      add_host:
        name: default
        ansible_host: 127.0.0.1
        ansible_port: 2222
        ansible_user: vagrant
        ansible_private_key_file: >
          .vagrant/machines/default/virtualbox/private_key

- name: Do something to the vagrant machine
  hosts: default
  tasks:
    # Здесь находится список выполняемых задач
    - name: ping
      ping:
...
```

Модуль `add_host` добавляет хост только на время исполнения сценария. Он не вносит изменений в файл реестра.

Подготавливая свои сценарии Ansible, мы предпочитаем разбивать их на две операции. Первая выполняется на локальном хосте и подготавливает хосты, а вторая настраивает их.

Обратите внимание, что для этой задачи задан параметр `creates=Vagrantfile`:

```
- name: Create a Vagrantfile
  command: "vagrant init {{ box }}"
  args:
    creates: Vagrantfile
```

Он сообщает системе Ansible, что если файл Vagrantfile имеется, то хост уже находится в правильном состоянии и нет необходимости выполнять команду снова. Это один из способов достижения идемпотентности в сценариях Ansible, запускающих модуль `command`, благодаря которому команда (потенциально неидемпотентная) выполняется только один раз.

group_by

Посредством модуля `group_by` Ansible позволяет создавать новые группы во время выполнения сценария, основываясь на значении переменной, которая была установлена для каждого хоста и в терминологии Ansible называет *фактом* (подробнее о фактах рассказывается в главе 5).

Если сбор фактов включен, то Ansible определит для каждого хоста набор дополнительных переменных. Например, для 32-разрядных x86 машин будет определена переменная `ansible_machine` со значением `i386`, а для 64-разрядных x86 машин – со значением `x86_64`. Если Ansible используется для управления хостами с разной аппаратной архитектурой, то можно создать группы `i386` и `x86_64` с отдельными задачами.

Также можно воспользоваться фактом `ansible_distribution` для группировки хостов по названию дистрибутива Linux (например, Ubuntu или CentOS).

```
- name: Create groups based on Linux distribution
  group_by:
    key: "{{ ansible_facts.distribution }}"
```

Сценарий в примере 4.13 определяет отдельные группы для хостов с Ubuntu и CentOS, используя модуль `group_by`, а затем устанавливает пакеты – в Ubuntu с помощью модуля `apt` и в CentOS с помощью модуля `yum`.

Пример 4.13. Создание отдельных групп для разных дистрибутивов Linux

```
---

- name: Group hosts by distribution
  hosts: all
  gather_facts: true
  tasks:
    - name: Create groups based on distro
      group_by:
        key: "{{ ansible_facts.distribution }}"

- name: Do something to Ubuntu hosts
  hosts: Ubuntu
  become: true
  tasks:
    - name: Install jdk and jre
      apt:
        update_cache: true
        name:
          - openjdk-11-jdk-headless
          - openjdk-11-jre-headless
- name: Do something else to CentOS hosts
  hosts: CentOS
  become: true
  tasks:
    - name: Install jdk
      yum:
        name:
          - java-11-openjdk-headless
          - java-11-openjdk-devel
```

Заключение

На этом мы заканчиваем обсуждение реестра Ansible. Реестр – очень гибкий объект, помогающий описать имеющуюся инфраструктуру и порядок ее использования. Реестр может быть простым текстовым файлом и сложным сценарием, какой только вам удастся написать.

В следующей главе мы поближе познакомимся с переменными.

Глава 5

Переменные и факты

Ansible не является полноценным языком программирования, но в ней присутствуют некоторые черты, присущие языкам программирования. Одна из таких черт – *подстановка переменных*. В этой главе мы подробнее рассмотрим поддержку переменных в Ansible, включая специальный тип переменных, который в терминах Ansible называется *фактом*.

Определение переменных в сценариях

Самый простой способ определить переменную – поместить в сценарий секцию `vars` с именами и значениями переменных. Мы уже использовали этот прием в примере 3.9, где определили несколько переменных конфигурации:

```
vars:
  tls_dir: /etc/nginx/ssl/
  key_file: nginx.key
  cert_file: nginx.crt
  conf_file: /etc/nginx/sites-available/default
  server_name: localhost
```

Определение переменных в отдельных файлах

Ansible позволяет также распределить объявления переменных по нескольким файлам, использовав секцию `vars_files` . Допустим, что в предыдущем примере нам понадобилось поместить переменные в файл *nginx.yml*, убрав их из сценария. Для этого достаточно заменить секцию `vars` секцией `vars_files`, как показано ниже:

```
vars_files:
  - nginx.yml
```

Файл *nginx.yml* будет выглядеть, как показано в примере 5.1.

Пример 5.1. *nginx.yml*

```
key_file: nginx.key
cert_file: nginx.crt
```

```
conf_file: /etc/nginx/sites-available/default
server_name: localhost
```

В главе 6 мы увидим пример использования секции `vars_files` для перемещения переменных с конфиденциальной информацией в отдельный файл.

Структура каталогов

Как уже обсуждалось в главе 4, Ansible позволяет определить переменные, связанные с хостами или группами, в файле реестра или в отдельных файлах, находящихся рядом с файлом реестра или сценарием. Для этого нужно создать каталоги рядом с файлом реестра или сценариями. Файлы и каталоги в подкаталоге *group_vars* должны иметь имена, совпадающие с именами соответствующих им групп в файле реестра, а файлы в каталоге *host_vars* – с именами соответствующих им хостов:

```
inventory/
  production/
    hosts
    group_vars/
      webservers.yml
      all.yml
    host_vars/
      hostname.yml
```

Вывод значений переменных

Для отладки часто удобно иметь возможность вывести значения переменных. В главе 3 мы видели, как использовать модуль `debug` для вывода произвольного сообщения. Его также можно использовать для вывода значений переменных:

```
- debug: var=myvarname
```

Эту сокращенную форму записи без атрибута `name` удобно использовать во время разработки, и мы еще не раз применим ее в этой главе.

Интерполяция переменных

Чтобы вывести отладочное сообщение с переменной, нужно заключить имя переменной в двойные кавычки и окружить двойными фигурными скобками:

```
- name: Display the variable
  debug:
    msg: "The file used was {{ conf_file }}"
```

Значения переменных можно объединять в двойных фигурных скобках с помощью оператора тильды `~`:

```
- name: Concatenate variables
  debug:
    msg: "The URL is https://{{ server_name ~'.'~ domain_name }}/"
```

Регистрация переменных

Часто требуется установить значение переменной в зависимости от результата задачи. Напомним, что все модули в Ansible возвращают результат в формате JSON. Чтобы сохранить этот результат, нужно создать *зарегистрированную переменную* при вызове модуля с помощью ключевого слова `register`. Пример 5.2 демонстрирует, как сохранить ввод команды `whoami` в переменной `login`.

Пример 5.2. Сохранение вывода команды в переменной

```
- name: Capture output of whoami command
  command: whoami
  register: login
```

Чтобы использовать переменную `login` позднее, мы должны знать тип ее значения. Значением переменных, объявленных с помощью ключевого слова `register`, всегда является словарь, однако ключи в словаре могут отличаться в зависимости от вызываемого модуля.

К сожалению, в официальной документации по модулям Ansible не указывается, как выглядят значения, возвращаемые каждым модулем. Но в документации к модулям часто приводятся примеры с ключевым словом `register`, что может оказаться полезным. Простейший способ узнать, какие значения возвращает модуль, – зарегистрировать переменную и вывести ее содержимое с помощью модуля `debug`.

Допустим, у нас есть сценарий, представленный в примере 5.3.

Пример 5.3. *whoami.yml*

```
---
- name: Show return value of command module
  hosts: fedora
  gather_facts: false
  tasks:
    - name: Capture output of id command
      command: id -un
      register: login

    - debug: var=login
    - debug: msg="Logged in as user {{ login.stdout }}"
...
```

Вот что выведет модуль `debug`:

```
TASK [debug] *******************************************************************
ok: [fedora] ==> {
    "login": {
        "changed": true,                          ❶
        "cmd": [                                  ❷
            "id",
            "-un"
        ],
        "delta": "0:00:00.002262",
        "end": "2021-05-30 09:25:41.696308",
        "failed": false,
        "rc": 0,                                  ❸
        "start": "2021-05-30 09:25:41.694046",
        "stderr": "",                             ❹
        "stderr_lines": [],
        "stdout": "vagrant",                      ❺
        "stdout_lines": [                         ❻
            "vagrant"
        ]
    }
}
```

❶ Ключ `changed` присутствует в возвращаемых значениях всех модулей, с его помощью Ansible сообщает, произошли ли изменения в состоянии. Модули `command` и `shell` всегда возвращают значение `true`, если оно не было изменено ключевым словом `changed_when`, которое будет рассматриваться в главе 8.

❷ Ключ `cmd` содержит выполненную команду в виде списка строк.

❸ Ключ `rc` содержит код возврата. Если он не равен нулю, то Ansible считает, что задача выполнилась с ошибкой.

❹ Ключ `stderr` содержит текст, записанный в стандартный вывод ошибок, в виде одной строки.

❺ Ключ `stdout` содержит текст, записанный в стандартный вывод, в виде одной строки.

❻ Ключ `stdout_lines` содержит текст, записанный в стандартный вывод, с разбивкой на строки по символу перевода строки. Это список, каждый элемент которого является одной строкой из стандартного вывода.

При использовании ключевого слова `register` с модулем `command` обычно требуется доступ к ключу `stdout`, как показано в примере 5.4.

Пример 5.4. Использование результата вывода команды в задаче

```
- name: Capture output of id command
  command: id -un
```

```
    register: login

- debug: msg="Logged in as user {{ login.stdout }}"
```

Иногда бывает желательно как-то обработать вывод задачи, потерпевшей ошибку. Однако если задача потерпела ошибку, то Ansible остановит ее выполнение, не дав возможности получить эту ошибку. Чтобы Ansible не останавливала работу после появления ошибки, можно использовать ключевое слово `ignore_errors`, как показано в примере 5.5.

Пример 5.5. Игнорирование ошибки при выполнении модуля

```
- name: Run myprog
  command: /opt/myprog
  register: result
  ignore_errors: true

- debug: var=result
```

Возвращаемое значение модуля `shell` имеет такую же структуру, что и возвращаемое значение модуля `command`, но другие модули возвращают отличающиеся ключи.

В примере 5.6 показано, что возвращается модуль `stat`, получающий атрибуты файла.

Пример 5.6. Фрагмент вывода модуля *stat*

```
TASK [Display result.stat] ************************************************************
ok: [ubuntu] ==> {
    "result.stat": {
        "atime": 1622724660.888851,
        "attr_flags": "e",
        "attributes": [
            "extents"
        ],
        "block_size": 4096,
        "blocks": 8,
        "charset": "us-ascii",
        "checksum": "7df51a4a26c00e5b204e547da4647b36d44dbdbf",
        "ctime": 1621374401.1193385,
        "dev": 2049,
        "device_type": 0,
        "executable": false,
        "exists": true,
        "gid": 0,
```

```
        "gr_name": "root",
        "inode": 784,
        "isblk": false,
        "ischr": false,
        "isdir": false,
        "isfifo": false,
        "isgid": false,
        "islnk": false,
        "isreg": true,
        "issock": false,
        "isuid": false,
        "mimetype": "text/plain",
        "mode": "0644",
        "mtime": 1621374219.5709288,
        "nlink": 1,
        "path": "/etc/ssh/sshd_config",
        "pw_name": "root",
        "readable": true,
        "rgrp": true,
        "roth": true,
        "rusr": true,
        "size": 3287,
        "uid": 0,
        "version": "1324051592",
        "wgrp": false,
        "woth": false,
        "writeable": true,
        "wusr": true,
        "xgrp": false,
        "xoth": false,
        "xusr": false
    }
}
```


Модуль `stat` сообщает всю информацию о файле, какую только можно получить.

Если вы собираетесь использовать зарегистрированные переменные в своих сценариях, то обязательно узнайте, что возвращается в них в обоих случаях – когда состояние хоста изменяется и когда оно не изменяется. В противном случае ваш сценарий может потерпеть неудачу, попытавшись обратиться к отсутствующему ключу зарегистрированной переменной.

Доступ к ключам словаря в переменной

Если переменная содержит словарь, то получить доступ к его ключам можно при помощи точки (.) или индекса ([]). В примере 5.6 был представлен способ ссылки на переменную с использованием точки:

```
{{ result.stat }}
```

Однако точно так же можно было бы использовать индекс:

```
{{ result['stat'] }}
```

Это правило применимо к любому уровню вложенности, т. е. все следующие выражения эквивалентны:

```
result['stat']['mode']
result['stat'].mode
result.stat['mode']
result.stat.mode
```

Бас предпочитает пользоваться точкой (точечной нотацией), кроме случаев, когда ключ содержит символы, которые нельзя использовать в качестве имени переменной, такие как точка, пробел или дефис.

Главное преимущество формы доступа с индексом – возможность использовать переменные в квадратных скобках (не заключая их в кавычки):

```
- name: Display result.stat detail
  debug: var=result['stat'][stat_key]
```

Для разыменования переменных Ansible использует Jinja2. За дополнительной информацией обращайтесь к документации Jinja2 (**https://oreil.ly/8hKiE**).

Факты

Как было показано выше, когда Ansible выполняет сценарий, до запуска первой задачи происходит следующее:

```
TASK [Gathering Facts] *********************************************************
ok: [debian]
ok: [fedora]
ok: [ubuntu]
```

На этапе сбора фактов (`GATHERING FACTS`) Ansible подключается к хосту и запрашивает у него всю информацию: аппаратную архитектуру, название операционной системы, IP-адреса, объем памяти и диска и др. Получить доступ ко всем этим данным можно через переменную `ansible_facts`. По умолчанию к некоторым фактам Ansible можно также обращаться как к переменным верхнего уровня, добавляя префикс `ansible_`. Эти переменные ведут себя точно так же, как любые другие переменные. Это поведение можно отключить с помощью параметра `INJECT_FACTS_AS_VARS`.

В примере 5.7 показан сценарий, который выводит сведения об операционной системе каждого сервера.

Пример 5.7. Сценарий, сообщающий сведения об операционной системе

```
---
- name: 'Ansible facts.'
  hosts: all
  gather_facts: true
  tasks:
    - name: Print out operating system details
      debug:
        msg: >-
          os_family:
          {{ ansible_facts.os_family }},
          distro:
          {{ ansible_facts.distribution }}
          {{ ansible_facts.distribution_version }},
          kernel:
          {{ ansible_facts.kernel }}
...
```

Так выглядит вывод для серверов с Debian, Fedora и Ubuntu:

```
PLAY [Ansible facts.] ************************************************************
TASK [Gathering Facts] ***********************************************************
ok: [debian]
ok: [fedora]
ok: [ubuntu]
TASK [Print out operating system details] *****************************************
ok: [ubuntu] ==> {
    "msg": "os_family: Debian, distro: Ubuntu 20.04, kernel: 5.4.0-73-generic"
}
ok: [fedora] ==> {
    "msg": "os_family: RedHat, distro: Fedora 34, kernel: 5.11.12-300.fc34.x86_64"
}
ok: [debian] ==> {
    "msg": "os_family: Debian, distro: Debian 10, kernel: 4.19.0-16-amd64"
}
PLAY RECAP ***********************************************************************
debian : ok=2  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
fedora : ok=2  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
ubuntu : ok=2  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

Просмотр всех фактов, доступных для сервера

Ansible осуществляет сбор фактов с помощью специального модуля `setup`. Вам не нужно запускать этот модуль в сценариях, потому что

Ansible делает это автоматически на этапе сбора фактов. Однако если вручную запустить его с помощью утилиты `ansible`, например:

```
$ ansible ubuntu -m setup
```

то Ansible выведет все факты, как показано в примере 5.8.

Пример 5.8. Результат запуска модуля *setup*

```
ubuntu | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "192.168.4.10",
            "10.0.2.15"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fef1:d47",
            "fe80::a6:4dff:fe77:e100"
        ],
(множество других фактов)
```

Обратите внимание, что модуль возвращает словарь с ключом `ansible_facts`, значением которого является словарь с именами и значения актуальных фактов.

Вывод подмножества фактов

Поскольку Ansible собирает большое количество фактов, модуль `setup` поддерживает параметр `filter` для фильтрации фактов по именам с поддержкой шаблонных символов (*шаблонные символы* используются командными оболочками для выбора файлов по шаблону, такому как *.txt.) Параметр `filter` фильтрует только по ключам верхнего уровня в словаре `ansible_facts`. Например, команда

```
$ ansible all -m setup -a 'filter=ansible_all_ipv6_addresses'
```

выведет:

```
debian | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fe8d:c04d",
            "fe80::a00:27ff:fe55:2351"
        ]
    },
    "changed": false
}
fedora | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv6_addresses": [
            "fe80::505d:173f:a6fc:3f91",
```

```
            "fe80::a00:27ff:fe48:995"
        ]
    },
    "changed": false
}
ubuntu | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fef1:d47",
            "fe80::a6:4dff:fe77:e100"
        ]
    },
    "changed": false
}
```

Использование параметра `filter` помогает найти интересующие детали настройки сервера. Фильтр `filter=ansible_env` выведет значения переменных окружения на целевых хостах.

Любой модуль может возвращать факты

Если внимательно рассмотреть пример 5.8, то можно заметить, что результатом является словарь с ключом `ansible_facts`. Ключ `ansible_facts` в возвращаемом значении – это идиома Ansible. Если модуль вернет словарь, содержащий ключ `ansible_facts`, то Ansible создаст переменные с этими именами и значениями и ассоциирует их с активным хостом. Модули, возвращающие информацию об объектах, не являющихся уникальными для хоста, получают имена, оканчивающееся на `_info`.

Для модулей, возвращающих факты, нет необходимости регистрировать переменные, поскольку Ansible создает их автоматически. Например, задача в примере 5.9 использует модуль `service_facts` для извлечения фактов о службах, а затем выводит информацию, касающуюся демона SSH. (Обратите внимание, что здесь для обращения к ключу словаря используется индексная нотация. Это связано с наличием точки в имени ключа.)

Пример 5.9. Использование модуля *service_facts* для извлечения фактов

```
- name: Show a fact returned by a module
  hosts: debian
  gather_facts: false
  tasks:
    - name: Get services facts
      service_facts:

    - debug: var=ansible_facts['services']['sshd.service']
```

Эта задача выведет:

```
TASK [debug] *******************************************************************
ok: [debian] ==> {
    "ansible_facts['services']['sshd.service']": {
        "name": "sshd.service",
        "source": "systemd",
        "state": "active",
        "status": "enabled"
    }
}
```

Обратите внимание, что нет необходимости использовать ключевое слово `register` при вызове модуля `service_facts`, потому что он возвращает факты. В Ansible имеется несколько модулей, возвращающих факты.

Локальные факты

Ansible поддерживает также дополнительный механизм, позволяющий ассоциировать факты с хостом. Разместите один или несколько файлов на хосте в каталоге */etc/ansible/facts.d*, и Ansible обнаружит их, если они отвечают любому из следующих требований:

- имеют формат *.ini*;
- имеют формат JSON;
- являются выполняемыми файлами, не принимающими аргументов, и выводят результат в формате JSON в стандартный вывод.

Эти факты доступны в виде ключей особой переменной `ansible_local`. В примере 5.10 представлен файл факта в формате *.ini*.

Пример 5.10. */etc/ansible/facts.d/example.fact*

```
[book]
title=Ansible: Up and Running
authors=Meijer, Hochstein, Moser
publisher=O'Reilly
```

Если скопировать этот файл в */etc/ansible/facts.d/example.fact* на удаленном хосте, мы сможем обратиться к содержимому переменной `ansible_local` в сценарии:

```
- name: Print ansible_local
  debug: var=ansible_local

- name: Print book title
  debug: msg="The title of the book is {{ ansible_local.example.book.title }}"
```

Вот что получится в результате выполнения этих задач:

```
TASK [Print ansible_local] ******************************************************
ok: [fedora] ==> {
    "ansible_local": {
        "example": {
            "book": {
                "authors": "Meijer, Hochstein, Moser",
                "publisher": "O'Reilly",
                "title": "Ansible: Up and Running"
            }
        }
    }
}
TASK [Print book title] *********************************************************
ok: [fedora] ==> {
    "msg": "The title of the book is Ansible: Up and Running"
}
```

Обратите внимание на структуру значения переменной `ansible_local`. Поскольку файл факта называется *example.fact*, переменная `ansible_local` получит значение-словарь с ключом `example`.

Использование модуля set_fact для задания новой переменной

Ansible позволяет устанавливать факты (по сути, создавать новые переменные) в задачах с помощью модуля `set_fact`. Лорин часто использует `set_fact` непосредственно после вызова `service_facts`, чтобы упростить ссылки на переменные. Пример 5.11 демонстрирует, как использовать `set_fact`, чтобы к переменной можно было обращаться по имени `nginx_state` вместо `ansible_facts.services.nginx.state`.

Пример 5.11. Использование *set_fact* для упрощения ссылок на переменные

```
- name: Set nginx_state
  when: ansible_facts.services.nginx.state is defined
  set_fact:
    nginx_state: "{{ ansible_facts.services.nginx.state }}"
```

Встроенные переменные

Ansible определяет несколько переменных, всегда доступных в сценариях. Они перечислены в табл. 5.1.

Переменная	Описание
`hostvars`	Словарь, ключами которого являются имена хостов Ansible, а значениями – словари, отображающие имена переменных в их значения

Переменная	Описание
`inventory_hostname`	Имя текущего хоста, как оно задано в Ansible. Может быть полным доменным именем (например, `myhost.example.com`)
`inventory_hostname_short`	Имя текущего хоста, как оно задано в Ansible, без имени домена (например, `myhost`)
`group_names`	Список всех групп, в которые входит текущий хост
`groups`	Словарь, ключи которого – имена групп в Ansible, а значения – списки имен хостов, входящих в группы. Включает группы `all` и `ungrouped`: `{"all": [...], "web": [...], "ungrouped": [...]}`
`ansible_check_mode`	Логическая переменная, принимающая истинное значение, когда сценарий выполняется в режиме проверки (см. раздел «Режим проверки» в главе 8)
`ansible_play_batch`	Логическая переменная, принимающая истинное значение, когда сценарий выполняется в тестовом режиме (см. раздел «Пакетная обработка хостов» в главе 11)
`ansible_play_hosts`	Список имен хостов из реестра, участвующих в текущей операции
`ansible_version`	Словарь с информацией о версии Ansible: `{"full": "2.3.1.0", "major": 2, "minor": 3, "revision": 1, "string": "2.3.1.0"}`

Переменные `hostvars`, `inventory_hostname` и `groups` заслуживают отдельного обсуждения.

hostvars

В Ansible область видимости переменных ограничивается хостами. Рассуждать о значении переменной имеет смысл только в контексте заданного хоста.

Идея соответствия переменных заданному хосту может показаться странной, поскольку Ansible позволяет определять переменные для групп хостов. Например, если объявить переменную в секции `vars` операции, она будет определена для набора хостов в этой операции. Но на самом деле Ansible создаст копию этой переменной для каждого хоста в группе.

Иногда задача, запущенная на одном хосте, требует значения переменной, определяемого на другом хосте. Например, вам может понадобиться создать на веб-сервере конфигурационный файл, содержащий IP-адрес интерфейса *eth1* сервера базы данных, который заранее неизвестен. IP-адрес доступен как факт `ansible_eth1.ipv4.address` сервера базы данных.

Решить проблему можно с помощью переменной `hostvars`. Это словарь, содержащий все переменные, объявленные на всех хостах, ключами которого являются имена хостов, как они заданы в реестре Ansible. Если Ansible еще не собрала фактов о хосте, тогда вы не сможете получить доступа к его фактам с использованием переменной `hostvars`, кроме случая, когда включено кеширование фактов[1].

Продолжим наш пример. Если сервер базы данных имеет имя *db.example.com*, тогда мы можем добавить в шаблон конфигурации следующую ссылку:

```
{{ hostvars['db.example.com'].ansible_eth1.ipv4.address }}
```

На ее место будет подставлено значение факта `ansible_eth1.ipv4.address`, связанного с хостом *db.example.com*.

Имейте в виду, что значение `hostvars` вычисляется при запуске Ansible, а `host_vars` – это каталог, в котором можно определить переменные для конкретной системы.

inventory_hostname

`inventory_hostname` – это имя текущего хоста, как оно задано в реестре Ansible. Если вы определили псевдоним для хоста, тогда это – псевдоним. Например, если реестр содержит строку

```
ubuntu ansible_host=192.168.4.10
```

тогда переменная `inventory_hostname` получит значение `ubuntu`.

Вот как с помощью `hostvars` и `inventory_hostname` можно вывести все переменные, связанные с текущим хостом:

```
- debug: var=hostvars[inventory_hostname]
```

groups

Переменная `groups` может пригодиться для доступа к переменным, определенным для группы хостов. Допустим, мы настраиваем хост балансировщика нагрузки, и требуется добавить в конфигурационный файл IP-адреса всех серверов в группе `web`. Тогда мы можем добавить в шаблон конфигурации следующий фрагмент:

```
backend web-backend
{% for host in groups.web %}
```

[1] Информация о кешировании данных приводится в главе 11.

```
    server {{ hostvars[host].inventory_hostname }} \
    {{ hostvars[host].ansible_default_ipv4.address }}:80
{% endfor %}
```

и получить такой результат:

```
backend web-backend
  server georgia.example.com 203.0.113.15:80
  server newhampshire.example.com 203.0.113.25:80
  server newjersey.example.com 203.0.113.38:80
```

С помощью переменной `groups` в шаблоне файла конфигурации можно перебирать хосты в группе, используя только имя группы, или изменять хосты в группе, не изменяя шаблон файла конфигурации.

Установка переменных из командной строки

Переменные, установленные передачей параметра `-e var=value` команде `ansible-playbook`, имеют наивысший приоритет и могут заменять ранее определенные переменные. В примере 5.12 показано, как установить переменную `greeting` со значением `hiya`.

Пример 5.12. Установка переменной в командной строке

```
$ ansible-playbook 4-12-greet.yml -e greeting=hiya
```

Метод `ansible-playbook -e var=value` лучше всего использовать, когда сценарий Ansible предполагается применять подобно сценарию командной оболочки, принимающему аргумент командной строки. Параметр `-e` позволяет передавать переменные как аргументы.

В примере 5.13 демонстрируется очень простой сценарий, который выводит сообщение, определяемое переменной.

Пример 5.13. Вывод сообщения, определяемого переменной

```
---
- name: Pass a message on the command line
  hosts: localhost
  gather_facts: false

  vars:
    greeting: "you didn't specify a message"

  tasks:
    - name: Output a message
      debug:
        msg: "{{ greeting }}"
...
```

Если запустить его, как показано ниже:

```
$ ansible-playbook greet.yml -e greeting=hiya
```

то он выведет:

```
PLAY [Pass a message on the command line] ***************************************
TASK [Gathering Facts] **********************************************************
ok: [localhost]
TASK [Output a message] *********************************************************
ok: [localhost] ==> {
    "msg": "hiya"
}
PLAY RECAP **********************************************************************
localhost : ok=2  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

Чтобы включить пробел в значение переменной, используйте кавычки:

```
$ ansible-playbook greet.yml -e 'greeting="hi there"'
```

Данное значение необходимо целиком заключить в одинарные кавычки `'greeting="hi there"'`, чтобы командная оболочка интерпретировала его как один аргумент. Кроме того, строку `"hi there"` нужно заключить в двойные кавычки, чтобы Ansible интерпретировала сообщение как единую строку.

Вместо отдельных переменных Ansible позволяет передать ей файл с определениями переменных, для чего в параметре `-e` следует передать имя файла в виде `@filename.yml`. Например, допустим, что у нас имеется файл, как показано в примере 5.14.

Пример 5.14. *greetvars.yml*

```
greeting: hiya
```

Этот файл можно передать сценарию, как показано ниже:

```
$ ansible-playbook 5-12-greet.yml -e @5-14-greetvars.yml
```

В примере 5.15 показан простой способ вывода любой переменной, заданной флагом `-e` в командной строке.

Пример 5.15. Вывод переменной, заданной флагом *-e*

```
---
- name: Show any variable during debugging.
  hosts: all
  gather_facts: true
  tasks:
    - debug: var="{{ variable }}"
...
```

Этот прием фактически позволяет получить «изменчивую переменную», которую можно использовать для отладки:

```
$ ansible-playbook 5-15-variable-variable.yml -e variable=ansible_python
```

Приоритет

Мы рассмотрели несколько способов определения переменных, и может случиться так, что вам потребуется задавать одну и ту же переменную для хоста множество раз, используя разные значения. По возможности избегайте этого. Но если сделать это не получается, то вспомните правила определения приоритета в Ansible. Когда одна и та же переменная определяется в нескольких местах, правила приоритета определяют, какое из значений она получит в конце концов.

Знание правил приоритета, которые применяет Ansible, может вам очень пригодиться[1]. Вот простое эмпирическое правило: чем ближе к хосту, тем выше приоритет. Соответственно, `group_vars` имеет приоритет над значениями по умолчанию, определяемыми ролями, а `host_vars` имеет приоритет перед `group_vars`. Ниже перечислены способы определения переменных в порядке возрастания приоритетов.

1. Значения командной строки (например, `-u my_user`; это не переменные).
2. Значения по умолчанию в ролях (определяются в *role/defaults/main.yml*).
3. Переменные, определяемые в реестре или в сценарии для групп хостов.
4. Секция *group_vars/all* в реестре.
5. Секция *group_vars/all* в сценарии.
6. Секция *group_vars/** в реестре.
7. Секция *group_vars/** в сценарии.
8. Переменные, определяемые в реестре или в сценарии для хостов.
9. Секция *host_vars/** в реестре.
10. Секция *host_vars/** в сценарии.
11. Факты хостов / кешированные факты из *set_facts*.
12. Переменные операций.
13. Секция *vars_prompt* в операции.
14. Секция *vars_files* в операции.
15. Переменные ролей (определяемые в файле *role/vars/main.yml*).
16. Блочные переменные (только для задач в блоке).

[1] Подробности ищите в официальной документации (https://oreil.ly/gqsfK).

17. Переменные задач (только для задач).
18. Секция *include_vars*.
19. Факты, возвращаемые модулем *set_facts* / зарегистрированные переменные.
20. Параметры ролей (и include_role).
21. Подключаемые параметры.
22. Дополнительные переменные (например, `-e "user=my_user"`).

Заключение

В этой главе мы рассмотрели разные способы определения и доступа к фактам и переменным. Отделение переменных от задач и создание реестров с правильными значениями переменных позволяет создавать окружения для тестирования и обкатки программного обеспечения. Ansible – очень гибкая система в том, что касается определения данных на том или ином уровне. В следующей главе мы сконцентрируемся на практических примерах развертывания приложений.

Глава 6

Введение в Mezzanine: тестовое приложение

В главе 3 мы рассмотрели основные правила написания сценариев. Но в реальной жизни все более запутано, чем во вводных главах книг по программированию. Поэтому в этой главе мы рассмотрим законченный пример развертывания нетривиального приложения, а в следующей исследуем реализацию развертывания с помощью Ansible.

В качестве примера приложения используем систему управления контентом (Content Management System, CMS) Mezzanine (**https://oreil.ly/xqgMN**), сходную по духу с WordPress. Mezzanine устанавливается поверх Django, свободно распространяемого фреймворка веб-приложений на Python.

Почему сложно развертывать приложения в промышленном окружении

Давайте немного отклонимся от темы и поговорим о различиях между запуском программного обеспечения в окружении разработки на вашем ноутбуке и в промышленном окружении. Mezzanine – отличный пример приложения, которое гораздо легче запустить в окружении разработки, чем развернуть в промышленном окружении. В примере 6.1 показано, что необходимо для запуска приложения в Ubuntu Focal/64[1].

Пример 6.1. Запуск Mezzanine в окружении разработки

```
$ sudo apt-get install -y python3-venv
$ python3 -m venv venv
$ source venv/bin/activate
$ pip3 install wheel
$ pip3 install mezzanine
$ mezzanine-project myproject
$ cd myproject
$ sed -i 's/ALLOWED_HOSTS = \[\]/ALLOWED_HOSTS = ["*"]/' myproject/settings.py
```

[1] В этой главе мы установим пакеты Python в виртуальное окружение. Но в репозитории вы также найдете пример развертывания на виртуальной машине Vagrant.

```
$ python manage.py migrate
$ python manage.py runserver 0.0.0.0:8000
```

В конечном итоге вы должны увидеть следующий результат на терминале:

```
            .....
          _d^^^^^^^^^b_
       .d''           ``b.
     .p'                `q.
    .d'                   `b.
   .d'                     `b.   * Mezzanine 4.3.1
   ::                       ::   * Django 1.11.29
  ::    M E Z Z A N I N E    ::  * Python 3.8.5
   ::                       ::   * SQLite 3.31.1
   `p.                     .q'   * Linux 5.4.0-74-generic
    `p.                   .q'
     `b.                 .d'
       `q..           ..p'
          ^q........p^
              ''''
Performing system checks...
System check identified no issues (0 silenced).
June 15, 2021 - 19:24:35
Django version 1.11.29, using settings 'myproject.settings'
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
```

Введя в браузере адрес *http://127.0.0.1:8000/*, вы должны увидеть веб-страницу, как показано на рис. 6.1. (Этот сервер принимает запросы с любого IP-адреса, согласно указанному в сообщении адресу 0.0.0.0.)

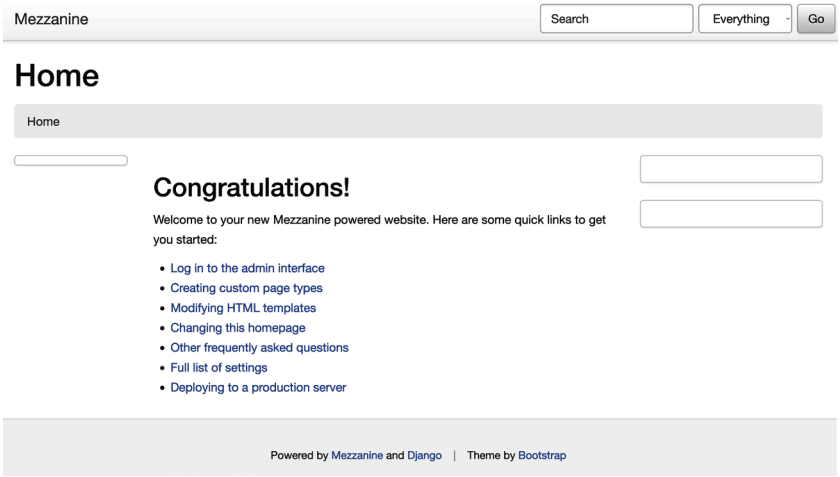


Рис. 6.1. Главная страница Mezzanine сразу после установки

Совсем другое дело – развертывание приложения в промышленном окружении. Когда вы запустите команду `mezzanine-project`, Mezzanine сгенерирует сценарий развертывания Fabric (**http://www.fabfile.org/**) в файле *myproject/fabfile.py*, который можно использовать для развертывания проекта на промышленном сервере. (Fabric – это инструмент, написанный на Python, позволяющий автоматизировать выполнение задач через SSH.) Сценарий содержит почти 700 строк кода без учета подключаемых им файлов конфигурации, также участвующих в развертывании.

Почему развертывание в промышленном окружении настолько сложнее? Я рад, что вы спросили. В окружении разработки Mezzanine допускает следующие упрощения (см. рис. 6.2):

- в качестве базы данных система использует SQLite и создает файл базы данных, если он отсутствует;
- HTTP-сервер разработки обслуживает и статический контент (изображения, файлы *.css*, JavaScript), и динамически сгенерированную разметку HTML;
- HTTP-сервер разработки использует незащищенный протокол HTTP, а не HTTPS (защищенный);
- процесс HTTP-сервера разработки запускается на переднем плане, занимая окно терминала;
- имя хоста HTTP-сервера всегда 127.0.0.1 (`localhost`).

Теперь посмотрим, что происходит при развертывании в промышленном окружении.

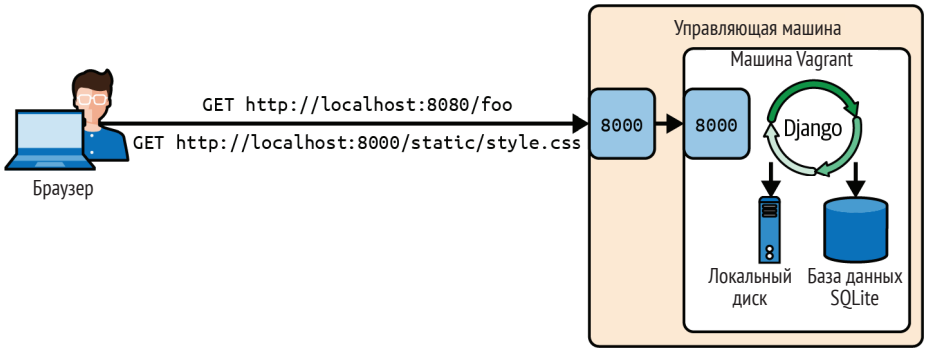


Рис. 6.2. Приложение Django в режиме разработки

База данных PostgreSQL

SQLite – встраиваемая база данных. В промышленном окружении лучше использовать базу данных промышленного уровня, обеспечивающую лучшую поддержку многочисленных одновременных запросов и позволяющую запускать несколько HTTP-серверов для балансировки

нагрузки. А это значит, что необходимо развернуть систему управления базами данных, такую как MySQL или PostgreSQL (или просто Postgres). Установка одного из упомянутых серверов баз данных создает дополнительные трудности. Мы должны:

1) установить сервер базы данных;
2) убедиться в его работоспособности;
3) создать базу данных;
4) создать пользователя базы данных с соответствующими правами доступа к ней;
5) настроить приложение Mezzanine на использование учетных данных пользователя базы данных и информации о соединении.

Сервер приложений Gunicorn

Поскольку Mezzanine является Django-приложением, его можно запускать под управлением HTTP-сервера Django, называемого в документации Django *сервером разработки*. Вот что сказано о сервере разработки (**https://oreil.ly/vBIFd**) в документации к Django 1.11:

> «Не используйте этот сервер в промышленном окружении. Он предназначен только для разработки. (Мы делаем веб-фреймворки, а не веб-серверы.)»

Django реализует стандарт Web Server Gateway Interface (WSGI)[1]. То есть для запуска Django-приложений, таких как Mezzanine, подойдет любой HTTP-сервер. Мы будем использовать Gunicorn – один из популярных HTTP-серверов с поддержкой WSGI, который использует сценарий развертывания Mezzanine. Также обратите внимание, что Mezzanine использует незащищенную версию Django, которая больше не поддерживается.

Веб-сервер NGINX

Gunicorn выполняет Django-приложение в точности как север разработки. Однако Gunicorn не обслуживает статических ресурсов приложения, таких как файлы изображений, *.css* и JavaScript. Их называют статическими, потому что они никогда не изменяются, в отличие от динамически генерируемых веб-страниц, которые обслуживает Gunicorn.

Несмотря на то что Gunicorn прекрасно справляется с шифрованием TLS, для работы с шифрованием обычно настраивают NGINX[2].

---

[1] Описание протокола WSGI можно найти в Python Enhancement Proposal (PEP) 3333 (https://oreil.ly/yyMcf).

[2] Поддержка шифрования TLS была добавлена в Gunicorn 0.17. До этого для поддержки шифрования приходилось использовать отдельное приложение, такое как NGINX.

Для обработки статических объектов и поддержки шифрования TLS мы будем использовать NGINX, как показано на рис. 6.3.

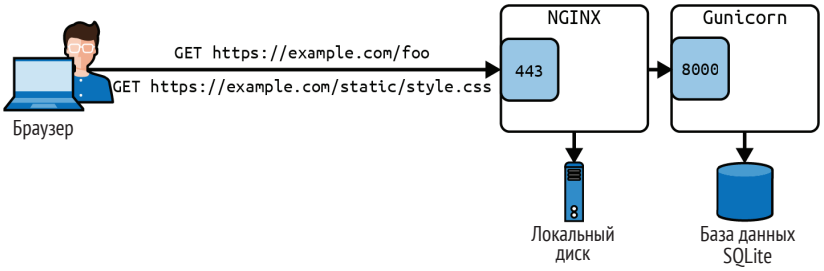


Рис. 6.3. NGINX как реверсивный прокси

Мы должны настроить NGINX как *реверсивный прокси* для Gunicorn. Если поступит запрос на получение статического объекта, например файл *.css*, то NGINX вернет его клиенту, взяв непосредственно из локальной файловой системы. Иначе NGINX передаст запрос Gunicorn, отправив HTTP-запрос службе Gunicorn, действующей на этой же машине. Какое из этих действий выполнить, NGINX определяет по URL.

Обратите внимание, что запросы извне поступают в NGINX по протоколу HTTPS (т. е. зашифрованы), а все запросы из NGINX в Gunicorn передаются в открытом, нешифрованном виде (по протоколу HTTP).

Диспетчер процессов Supervisor

В окружении разработки мы запускаем сервер приложений в терминале как приложение переднего плана. Закрытие терминала в этом случае приводит к автоматическому завершению программы. В промышленном окружении сервер приложений должен запускаться в фоновом режиме, чтобы он не завершался по окончании сеанса в терминале, в котором запущен процесс.

В просторечии такие процессы называют *демонами* (daemon), или *службами* (service). Мы должны запустить Gunicorn как демон, и еще нам нужна возможность останавливать и перезапускать его. Существует много диспетчеров задач, способных выполнить эту работу. Мы будем использовать Supervisor, потому что именно его используют сценарии развертывания Mezzanine.

Заключение

Теперь вы должны понимать, что требуется для развертывания веб-приложения в промышленном окружении. В главе 7 мы перейдем к реализации этой задачи с помощью Ansible.

Глава 7

Развертывание Mezzanine с помощью Ansible

Пришло время написать сценарий Ansible для развертывания Mezzanine на сервере. Мы проделаем это шаг за шагом. Но если вы относитесь к тому типу людей, которые начинают читать с конца книги, чтобы узнать, чем все закончится, то в конце главы в примере 7.27 вы увидите сценарий полностью. Он также доступен в репозитории GitHub. Прежде чем запустить его, прочитайте файл README.

Мы старались оставаться как можно ближе к оригинальным сценариям, которые написал Стефан МакДональд (Stephen McDonald), автор Mezzanine[1].

Вывод списка задач в сценарии

Прежде чем углубиться в недра нашего сценария, давайте взглянем на него с высоты. Утилита `ansible-playbook` поддерживает параметр `--list-tasks`. Он позволяет получить список всех задач, объявленных в сценарии. Вот как можно использовать этот параметр:

```
$ ansible-playbook --list-tasks mezzanine.yml
```

Пример 7.1 демонстрирует вывод этой команды для сценария *mezzanine.yml*, приведенного в примере 7.27.

Пример 7.1. Список задач в сценарии Mezzanine

```
playbook: mezzanine.yml
  play #1 (web): Deploy mezzanine    TAGS: []
    tasks:
      Install apt packages       TAGS: []
      Create project path        TAGS: []
      Create a logs directory    TAGS: []
      Check out the repository on the host    TAGS: []
```

[1] В состав дистрибутива Mezzanine больше не входит сценарий Fabric для автоматизации развертывания.

```
Create python3 virtualenv TAGS: []
Copy requirements.txt to home directory   TAGS: []
Install packages listed in requirements.txt   TAGS: []
Create project locale      TAGS: []
Create a DB user  TAGS: []
Create the database       TAGS: []
Ensure config path exists TAGS: []
Create tls certificates   TAGS: []
Remove the default nginx config file    TAGS: []
Set the nginx config file TAGS: []
Enable the nginx config file      TAGS: []
Set the supervisor config file    TAGS: []
Install poll twitter cron job     TAGS: []
Set the gunicorn config file      TAGS: []
Generate the settings file        TAGS: []
Apply migrations to create the database, collect static content  TAGS: []
Set the site id   TAGS: []
Set the admin password    TAGS: []
```

Это простой способ выяснить, какие действия производятся сценарием.

Организация устанавливаемых файлов

Как уже говорилось, Mezzanine развертывается поверх Django. В терминологии Django веб-приложение называется *проектом*, и мы должны дать ему имя. Пусть это будет *mezzanine_example*.

Наш сценарий производит установку на машину Vagrant и помещает файлы в домашний каталог пользователя Vagrant.

Пример 7.2. Структура каталогов в */home/vagrant*

```
.
|---- logs
|---- mezzanine
|     |___ mezzanine_example
|____ .virtualenvs
      |___ mezzanine_example
```

В примере 7.2 показана соответствующая структура каталогов внутри */home/vagrant*:

- */home/vagrant/mezzanine_example* – каталог верхнего уровня, куда будет копироваться исходный код из репозитория в GitHub;
- */home/vagrant/.virtualenvs/mezzanine_example* – каталог виртуального окружения Python (virtualenv), куда будут устанавливаться все дополнительные пакеты на языке Python;

- */home/vagrant/logs* – каталог для хранения журналов, создаваемых приложением Mezzanine.

Переменные и скрытые переменные

Как показано в примере 7.3, сценарий определяет довольно много переменных.

Пример 7.3. Определения переменных

```
vars:
  user: "{{ ansible_user }}"
  proj_app: mezzanine_example
  proj_name: "{{ proj_app }}"
  venv_home: "{{ ansible_env.HOME }}/.virtualenvs"
  venv_path: "{{ venv_home }}/{{ proj_name }}"
  proj_path: "{{ ansible_env.HOME }}/mezzanine/{{ proj_name }}"
  settings_path: "{{ proj_path }}/{{ proj_name }}"
  reqs_path: requirements.txt
  manage: "{{ python }} {{ proj_path }}/manage.py"
  live_hostname: 192.168.33.10.nip.io
  domains:
    - 192.168.33.10.nip.io
    - www.192.168.33.10.nip.io
  repo_url: git@github.com:ansiblebook/mezzanine_example.git
  locale: 'en_US.UTF-8'
  # Переменные ниже отсутствуют в сценарии fabfile.py установки Mezzanine
  # но мы добавили их для удобства
  conf_path: /etc/nginx/conf
  tls_enabled: true
  python: "{{ venv_path }}/bin/python"
  database_name: "{{ proj_name }}"
  database_user: "{{ proj_name }}"
  database_host: localhost
  database_port: 5432
  gunicorn_procname: gunicorn_mezzanine

vars_files:
  - secrets.yml
```

В большинстве случаев мы старались использовать те же имена переменных, что и в Fabric-сценарии установки Mezzanine. Мы также добавили несколько переменных, чтобы сделать процесс более прозрачным. Например, сценарии Fabric используют переменную `proj_name` для хранения имени базы данных и имени пользователя базы данных. Мы предпочитаем задавать вспомогательные переменные, такие как `database_name` и `database_user`, и определять их через `proj_name`.

Отметим несколько важных моментов. Во-первых, обратите внимание, как можно определить одну переменную на основе другой. Например, переменная `venv_path` определяется на основе переменных `venv_home` и `proj_name`.

Во-вторых, обратите внимание, как можно сослаться на факты Ansible в этих переменных. Например, переменная `venv_home` определена на основе факта `ansible_env`, получаемого из каждого хоста.

И наконец, обратите внимание, что мы определили несколько переменных в отдельном файле *secrets.yml*:

```
vars_files:
  - secrets.yml
```

Этот файл содержит такие данные, как пароли и токены, и они должны оставаться конфиденциальными. В репозитории на GitHub этот файл отсутствует. Вместо него имеется файл *secrets.yml.example*. Вот как он выглядит:

```
db_pass: e79c9761d0b54698a83ff3f93769e309
admin_pass: 46041386be534591ad24902bf72071B
secret_key: b495a05c396843b6b47ac944a72c92ed
nevercache_key: b5d87bb4e17c483093296fa321056bdc

# Вы должны создать приложение Twitter по адресу: https://dev.twitter.com
# чтобы получить учетные данные для интеграции Mezzanine с Twitter.
#
## Подробности об интеграции Mezzanine с Twitter приводятся по адресу:
# https://mezzanine.readthedocs.io/en/latest/twitter-integration.html
twitter_access_token_key: 80b557a3a8d14cb7a2b91d60398fb8ce
twitter_access_token_secret: 1974cf8419114bdd9d4ea3db7a210d90
twitter_consumer_key: 1f1c627530b34bb58701ac81ac3fad51
twitter_consumer_secret: 36515c2b60ee4ffb9d33d972a7ec350a
```

Чтобы воспользоваться им, скопируйте файл *secrets.yml.example* в *secrets.yml* и измените его так, чтобы он содержал данные вашего сайта.

Обратите внимание, что *secrets.yml* перечислен в файле *.gitignore* репозитория Git, чтобы предотвратить случайное сохранение этих данных в публичном репозитории. Лучше всего воздержаться от копирования незашифрованных данных в репозиторий, чтобы избежать рисков, связанных с безопасностью. Это всего лишь один из способов обеспечения секретности данных. Их также можно передавать через переменные окружения. Другой способ, описанный в главе 8, заключается в использовании версии файла *secrets.yml*, зашифрованной при помощи `ansible-vault`.

Установка большого количества пакетов

Нам потребуется установить два типа пакетов, чтобы развернуть Mezzanine: системные пакеты и несколько пакетов для Python. Поскольку мы собираемся развертывать приложение в Ubuntu, будем использовать для установки системных пакетов диспетчер `apt`, а для установки пакетов Python – диспетчер `pip`.

Устанавливать системные пакеты обычно проще, чем пакеты Python, потому что они созданы для непосредственного использования операционной системой. Однако в репозиториях системных пакетов зачастую отсутствуют новейшие версии библиотек для Python, которые нам необходимы. Поэтому их мы будем устанавливать отдельно. Это компромисс между стабильностью и использованием новейших и самых лучших версий.

В примере 7.4 показана задача, которую мы используем для установки системных пакетов.

Пример 7.4. Установка системных пакетов

```
- name: Install apt packages
  become: true
  apt:
    update_cache: true
    cache_valid_time: 3600
    pkg:
      - acl
      - git
      - libjpeg-dev
      - libpq-dev
      - memcached
      - nginx
      - postgresql
      - python3-dev
      - python3-pip
      - python3-venv
      - python3-psycopg2
      - supervisor
```

Когда устанавливается несколько пакетов, Ansible передает весь список модулю `apt`, а модуль вызовет программу `apt` только один раз, тоже передав ей весь список устанавливаемых пакетов целиком. Модуль `apt` способен обрабатывать такие списки.

Добавление выражения become в задачу

В примерах сценариев в главе 3 нам требовалось, чтобы сценарий целиком выполнялся с привилегиями пользователя root, поэтому мы добав-

ляли в операции выражение `become: true`. При развертывании Mezzanine большинство задач будет выполняться с привилегиями пользователя, от лица которого устанавливается SSH-соединение с хостом, а не root. Поэтому мы должны приобретать привилегии root не для *всей* операции, а только для определенных задач.

Для этого можно добавить выражение `become: true` в задачи, которые *необходимо* выполнить с привилегиями root, как в примере 7.4. Для большей наглядности Бас предпочитает добавлять `become: true` прямо под `- name:`.

Обновление кеша диспетчера пакетов apt

Ubuntu поддерживает кеш имен всех *apt*-пакетов, доступных в архиве пакетов Ubuntu. Представьте, что вы пытаетесь установить пакет с именем *libssl-dev*. Вы можете использовать программу `apt-cache`, чтобы запросить из кеша информацию об известной версии этой программы:

```
$ apt-cache policy libssl-dev
```

Все примеры команд в этом разделе выполняются на удаленном хосте (Ubuntu), а не на управляющей машине.

Результат показан в примере 7.5.

Пример 7.5. Вывод *apt-cache*

```
libssl-dev:
  Installed: (none)
  Candidate: 1.1.1f-1ubuntu2.4
  Version table:
     1.1.1f-1ubuntu2.4 500
        500 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages
     1.1.1f-1ubuntu2.3 500
        500 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages
     1.1.1f-1ubuntu2 500
        500 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages
```

Как видите, этот пакет не был установлен. Согласно информации из кеша, на локальной машине новейшая версия – 1.1.1f-1ubuntu2.4. Мы также получили информацию о местонахождении архива пакета.

В некоторых случаях, когда проект Ubuntu выпускает новую версию пакета, он удаляет устаревшую версию из архива. Если локальный кеш *apt* на сервере Ubuntu не был обновлен, он попытается установить пакет, которого нет в архиве.

Продолжая пример, предположим, что мы решили установить пакет *libssl-dev*:

```
$ sudo apt-get install libssl-dev
```

Если версия 1.1.1f-1ubuntu2.4 больше не доступна в архиве пакетов, мы увидим сообщение об ошибке.

Привести локальный кеш пакетов *apt* в актуальное состояние можно командой `apt-get update`. Вызывая модуль `apt` в Ansible, ему необходимо передать аргумент `update_cache: true`, чтобы обеспечить поддержание локального кеша apt в актуальном состоянии, как это показано в примере 7.4.

Обновление кеша занимает некоторое время, а мы можем запускать сценарий много раз подряд для отладки, поэтому, чтобы избежать ненужных затрат времени на обновление кеша, можно передать модулю аргумент `cache_valid_time`. Он разрешает обновление кеша, только если тот старше установленного порогового значения. В примере 7.4 используется аргумент `cache_valid_time: 3600`, который разрешает обновление кеша, только если он старше 3600 с (1 час).

Извлечение проекта из репозитория Git

Хотя Mezzanine можно использовать, не написав ни строчки кода, одной из сильных сторон этой системы является то, что она написана с использованием фреймворка Django, который, в свою очередь, служит прекрасной платформой для веб-приложений на Python. Если вам просто нужна система управления контентом (CMS), тогда обратите внимание на что-нибудь вроде WordPress. Но если вы пишете специализированное приложение, включающее функциональность CMS, то вам как нельзя лучше подойдет Mezzanine.

В ходе развертывания вам потребуется получить из репозитория Git код вашего Django-приложения. Выражаясь языком Django, репозиторий должен хранить *проект*. Я создал репозиторий в GitHub (**https://oreil.ly/HtoNP**) с проектом Django, содержащий все необходимые файлы. Этот проект и будет развертывать наш сценарий.

С помощью программы `mezzanine-project`, которая поставляется вместе с Mezzanine, мы создали файлы проекта, как показано ниже:

```
$ mezzanine-project mezzanine_example
$ chmod +x mezzanine_example/manage.py
```

В нашем репозитории нет никаких конкретных Django-приложений. Там содержатся только файлы, необходимые для проекта. В реальных условиях этот репозиторий содержал бы подкаталоги с дополнительными Django-приложениями.

В примере 7.6 показано, как использовать модуль `git` для извлечения проекта из удаленного репозитория Git.

Пример 7.6. Извлечение проекта из репозитория Git

```
- name: Check out the repository on the host
  git:
    repo: "{{ repo_url }}"
    dest: "{{ proj_path }}"
    version: master
    accept_hostkey: true
```

Мы открыл доступ к репозиторию для всех желающих, чтобы читатели смогли обращаться к нему, но в реальной жизни вам придется обращаться к закрытым репозиториям Git по SSH. Поэтому мы настроили переменную `repo_url` для использования схемы, которая клонирует репозиторий по SSH:

```
repo_url: git@github.com:ansiblebook/mezzanine_example.git
```

Если вы собираетесь опробовать примеры на своем компьютере, то для запуска сценария вы должны создать учетную запись на GitHub (**https://github.com/signup**):

1) добавить открытый ключ SSH в свою учетную запись на GitHub (**https://github.com/settings/keys**);
2) запустить SSH-агента на управляющей машине:

```
$ eval $(ssh-agent)
```

3) добавить свой закрытый ключ SSH в SSH-агента:

```
$ ssh-add  <путь к закрытому ключу>
```

В случае успеха следующая команда выведет открытый ключ SSH, только что добавленный вами:

```
$ ssh-add -L
```

Вывод должен выглядеть примерно так:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIN1/YRlI7Oc+KyM6NFZt7fb7pY+btItKHMLbZhdbwhj2
```

Чтобы включить агента перенаправления, добавьте следующие строки в файл *ansible.cfg*:

```
[ssh_connection]
ssh_args = -o ForwardAgent=yes
```

Проверить работоспособность агента перенаправления можно с помощью Ansible, как показано ниже:

```
$ ansible web -a "ssh-add -L"
```

Эта команда должна вывести то же самое, что команда `ssh-add -L` на вашей локальной машине.

Нелишним также будет убедиться в достижимости сервера GitHub по SSH:

```
$ ansible web -a "ssh -T git@github.com"
```

В случае успеха ее вывод должен выглядеть примерно так:

```
web | FAILED | rc=1 >>
Hi bbaassssiiee! You've successfully authenticated, but GitHub does not provide
shell access.
```

Пусть вас не смущает слово `FAILED` в выводе – сам факт получения сообщения от сервера GitHub уже говорит о том, что все в порядке.

Кроме URL репозитория в параметре `repo` и пути к репозиторию в параметре `dest`, нужно также передать дополнительный параметр `accept_hostkey`, связанный с *проверкой ключей хоста*. (Агента перенаправления SSH и проверку ключей хоста мы подробно рассмотрим в главе 20.)

Установка Mezzanine и других пакетов в virtualenv

Мы можем устанавливать пакеты Python от лица пользователя root на уровне всей системы, но лучше устанавливать их в изолированное окружение, чтобы избежать конфликтов с системными пакетами Python. В Python подобные изолированные окружения называют *virtualenv*. Пользователь может создать большое количество окружений virtualenv и установить в них пакеты без использования привилегий пользователя root. (Напомню, что мы собираемся установить некоторые пакеты для Python, чтобы получить самые свежие версии.)

Модуль `pip` в Ansible позволяет создавать такие изолированные окружения virtualenv и устанавливать в них пакеты.

В примере 7.7 демонстрируется использование модуля `pip` для создания virtualenv в Python 3 и установки последних версий инструментов поддержки.

Пример 7.7. Создание *virtualenv* в Python 3

```
- name: Create python3 virtualenv
  pip:
    name:
      - pip
      - wheel
      - setuptools
    state: latest
    virtualenv: "{{ venv_path }}"
    virtualenv_command: /usr/bin/python3 -m venv
```

В примере 7.8 показаны две задачи, устанавливающие пакеты Python в изолированное окружение virtualenv.

Пример 7.8. Установка пакетов Python

```
- name: Copy requirements.txt to home directory
  copy:
    src: requirements.txt
    dest: "{{ reqs_path }}"
    mode: '0644'

- name: install packages listed in requirements.txt
  pip:
    virtualenv: "{{ venv_path }}"
    requirements: "{{ reqs_path }}"
```

На практике в проектах Python принято перечислять зависимости пакетов в файле с именем *requirements.txt*. И действительно, репозиторий в нашем примере с системой Mezzanine содержит файл *requirements.txt*. Его содержимое приводится в примере 7.9.

Пример 7.9. *requirements.txt*

```
Mezzanine==4.3.1
```

Обратите внимание, что в файле *requirements.txt* указана конкретная версия пакета Python Mezzanine (4.3.1). В этом файле отсутствуют другие пакеты Python, которые мы должны установить, поэтому мы явно перечислим их в файле *requirements.txt* в каталоге со сценариями, который затем скопируем на хост.

Ansible позволяет указать разрешения для файлов, используемых несколькими модулями, включая `file`, `copy` и `template`. Разрешения можно задавать в символическом виде (например: `'u+rwx'` или `'u=rw,g=r,o=r'`). Для тех, кто имеет опыт использования `/usr/bin/chmod`, напомним, что на самом деле разрешения являются восьмеричными числами, поэтому, задавая разрешения в числовом виде, обязательно добавляйте начальный ноль, чтобы YAML-парсер в Ansible понял, что это восьмеричное число (например, `0644` или `01777`), или заключайте число в кавычки (например, `'644'` или `'1777'`), чтобы Ansible получила строку, которую можно преобразовать в число. Если передать Ansible число без соблюдения одного из этих правил, то она интерпретирует его как десятичное число, что приведет к неожиданным результатам. Лучшей практикой, помогающей избежать неоднозначности, считается явное определение наборов разрешений для каждого файла в одинарных кавычках и со сброшенными специальными битами (suid, segid), таких как `'0755'`.

Для всех остальных зависимостей будут установлены самые последние версии.

С другой стороны, если бы понадобилось зафиксировать версии всех пакетов, то мы могли бы организовать это несколькими способами, например можно создать файл *requirements.txt* и перечислить в нем пакеты с требуемыми версиями. Пример такого файла приводится в примере 7.10.

Пример 7.10. Пример файла *requirements.txt*

```
beautifulsoup4==4.9.3
bleach==3.3.0
certifi==2021.5.30
chardet==4.0.0
Django==1.11.29
django-appconf==1.0.4
django-compressor==2.4.1
django-contrib-comments==2.0.0
filebrowser-safe==0.5.0
future==0.18.2
grappelli-safe==0.5.2
gunicorn==20.1.0
idna==2.10
Mezzanine==4.3.1
oauthlib==3.1.1
packaging==21.0
Pillow==8.3.1
pkg-resources==0.0.0
psycopg2==2.9.1
pyparsing==2.4.7
python-memcached==1.59
pytz==2021.1
rcssmin==1.0.6
requests==2.25.1
requests-oauthlib==1.3.0
rjsmin==1.1.0
setproctitle==1.2.2
six==1.16.0
soupsieve==2.2.1
tzlocal==2.1
urllib3==1.26.6
webencodings==0.5.1
```

Если бы у нас уже имелось готовое изолированное окружение virtualenv с установленными в него пакетами, то мы могли бы воспользоваться командой `pip freeze`, чтобы вывести список установленных па-

кетов. Например, если окружение virtualenv находится в *~/.virtualenvs/mezzanine_example*, то активировать его и сохранить список установленных пакетов в файл *requirements.txt* можно было бы так:

```
$ source .virtualenvs/mezzanine_example/bin/activate
$ pip freeze > requirements.txt
```

В примере 7.11 показано, как можно задать имена пакетов и их версии в виде списка словарей. В этом примере модуль `pip` выполнит обход списка словарей и с помощью `with_items` получит отдельные элементы в виде `item.name` и `item.version`.

Пример 7.11. Определение имен пакетов и их версий

```
- name: Install python packages with pip
  pip:
    virtualenv: "{{ venv_path }}"
    name: "{{ item.name }}"
    version: "{{ item.version }}"
  with_items:
    - {name: mezzanine, version: '4.3.1' }
    - {name: gunicorn, version: '20.1.0' }
    - {name: setproctitle, version: '1.2.2' }
    - {name: psycopg2, version: '2.9.1' }
    - {name: django-compressor, version: '2.4.1' }
    - {name: python-memcached, version: '1.59' }
```

Обратите внимание на одинарные кавычки вокруг номеров версий: они гарантируют интерпретацию номеров как литералов без округления в некоторых пограничных случаях.

Короткое отступление: составные аргументы задач

Вызывая модуль, ему можно передать аргумент в виде строки (отлично подходит для особых случаев). Так, в примере 7.11 мы могли бы передать модулю `pip` строковый аргумент:

```
- name: Install package with pip
  pip: virtualenv={{ venv_path }} name={{ item.name }} version={{ item.version }}
```

Если вам не нравятся длинные строки, то строку аргумента можно разбить на несколько строк с помощью функции свертки строк в YAML:

```
- name: Install package with pip
  pip: >
    virtualenv={{ venv_path }}
```

```
    name={{ item.name }}
    version={{ item.version }}
```

Ansible поддерживает еще один способ разбиения команды вызова модуля на несколько строк. Вместо строкового аргумента можно передать словарь, в котором ключи соответствуют именам переменных. То есть пример 7.11 мог бы выглядеть так:

```
- name: Install package with pip
  pip:
    virtualenv: "{{ venv_path }}"
    name: "{{ item.name }}"
    version: "{{ item.version }}"
```

Подход на основе словарей также очень удобно использовать для вызова модулей, принимающих *составные аргументы*. Составной аргумент – это аргумент, включающий список или словарь. Хорошим примером модуля с составными аргументами может служить модуль `uri`, который посылает веб-запросы. В примере 7.12 показано, как можно вызвать модуль, принимающий список в параметре `body`.

Пример 7.12. Вызов модуля с составными аргументами

```
- name: Login to a form based webpage
  uri:
    url: 'https://your.form.based.auth.example.com/login.php'
    method: POST
    body_format: form-urlencoded
    body:
      name: your_username
      password: 'your_password'
      enter: Sign in
    status_code: 302
  register: login
```

Передача аргументов в виде словарей вместо строк – широко распространенная практика, позволяющая избежать ошибок с пробелами, которые могут возникнуть при передаче необязательных аргументов, и она очень хорошо зарекомендовала себя при работе с системами управления версиями. Но самое главное преимущество такой формы записи – это чистый синтаксис YAML, и все парсеры и линтеры YAML понимают ее. Форма записи со знаком равенства (`=`) считается устаревшей и нежелательной.

Если вы хотите разбить аргументы на несколько строк без передачи составных аргументов, то можете самостоятельно выбрать, в какой форме это сделать. Это дело вкуса. Бас обычно предпочитает словари, но в книге используются оба варианта.

Настройка базы данных

Когда среда Django действует в режиме разработки, то в качестве базы данных она использует SQLite. В этом случае автоматически создается файл базы данных, если такового не существует.

Чтобы задействовать систему управления базами данных, такую как Postgres, сначала нужно создать учетную запись пользователя, владеющего базой данных, а затем саму базу данных внутри Postgres. Чуть позже мы настроим Mezzanine, используя данные этого пользователя.

Ansible поставляется с модулями `postgresql_user` и `postgresql_db` для создания учетных записей пользователей и баз данных внутри Postgres. В примере 7.13 показано, как пользоваться этими модулями в сценариях.

При создании базы данных мы настраиваем региональные настройки `lc_ctype` и `lc_collate`. А чтобы гарантировать установку региональных настроек в системе, мы используем модуль `locale_gen`.

Пример 7.13. Создание базы данных и пользователя базы данных

```
- name: Create project locale
  become: true
  locale_gen:
    name: "{{ locale }}"

- name: Create a DB user
  become: true
  become_user: postgres
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"

- name: Create the database
  become: true
  become_user: postgres
  postgresql_db:
    name: "{{ database_name }}"
    owner: "{{ database_user }}"
    encoding: UTF8
    lc_ctype: "{{ locale }}"
    lc_collate: "{{ locale }}"
    template: template0
```

Обратите внимание на выражения `become: true` и `become_user: postgres` в двух последних задачах. Когда выполняется установка Postgres в Ubuntu, в ее процессе создается пользователь с именем `postgres`, обладающий привилегиями администратора для данной установки. Отметьте также, что по умолчанию пользователь root не обладает привилегиями

администратора в Postgres. По этой причине в сценарии необходимо выполнить команду `become` для пользователя Postgres, чтобы выполнять административные задачи, такие как создание пользователей и баз данных.

При создании базы данных мы устанавливаем кодировку (UTF8) и определяем региональные настройки (`LC_CTYPE`, `LC_COLLATE`) для базы данных. Поскольку в сценарии определяются региональные настройки, мы использовали шаблон `template0`[1].

Создание файла local_settings.py из шаблона

Все настройки проекта Django должны находиться в файле *settings.py*. Mezzanine, следуя общему правилу, разбивает их на две группы:

- настройки, одинаковые для всех установок (*settings.py*);
- настройки, разные для разных установок (*local_settings.py*).

Мы определили настройки, общие для всех установок, в файле *settings.py* в репозитории проекта (**https://oreil.ly/HtoNP**).

Файл *settings.py* содержит код на Python, который загружает файл *local_settings.py* с настройками, зависящими от установки. Файл *.gitignore* настроен так, чтобы игнорировать *local_settings.py*, потому что разработчики часто создают свои версии этого файла с настройками для их окружений разработки.

Нам тоже нужно создать файл *local_settings.py* и выгрузить его на удаленный хост. В примере 7.14 приводится шаблон Jinja2, который мы используем.

Пример 7.14. *local_settings.py.j2*

```
# Эти настройки уникальные и никому не должны передаваться.
SECRET_KEY = "{{ secret_key }}"
NEVERCACHE_KEY = "{{ nevercache_key }}"
ALLOWED_HOSTS = [{% for domain in domains %}"{{ domain }}",{% endfor %}]

DATABASES = {
    "default": {
        # Может завершаться на "postgresql_psycopg2", "mysql", "sqlite3" или
"oracle".
        "ENGINE": "django.db.backends.postgresql_psycopg2",
        # Имя БД или путь к файлу БД, если используется sqlite3.
        "NAME": "{{ proj_name }}",
        # Не используется с sqlite3.
        "USER": "{{ proj_name }}",
```

[1] За более подробной информацией о шаблонах баз данных обращайтесь к документации Postgres (https://oreil.ly/GhjeJ).

```
        # Не используется с sqlite3.
        "PASSWORD": "{{ db_pass }}",
        # Для локального хоста можно указать пустую строку. Не используется с sqlite3.
        "HOST": "127.0.0.1",
        # Пустая строка соответствует порту по умолчанию. Не используется с sqlite3.
        "PORT": "",
    }
}

CACHE_MIDDLEWARE_KEY_PREFIX = "{{ proj_name }}"
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.MemcachedCache",
        "LOCATION": "127.0.0.1:11211",
    }
}

SESSION_ENGINE = "django.contrib.sessions.backends.cache"
```

Большая часть этого шаблона проста и понятна. Он использует синтаксис `{{ variable }}` для вставки значений переменных, таких как `secret_key`, `nevercache_key`, `proj_name` и `db_pass`. Единственная неочевидная вещь – это строка, приведенная в примере 7.15:

Пример 7.15. Использование цикла for в шаблоне Jinja2

```
ALLOWED_HOSTS = [{% for domain in domains %}"{{ domain }}",{% endfor %}]
```

Если вернуться к определению переменной, то можно увидеть, что переменная `domains` определена так:

```
domains:
  - 192.168.33.10.nip.io
  - www.192.168.33.10.nip.io
```

Система Mezzanine будет отвечать *только* на запросы к серверам, перечисленным в списке в переменной `domains`, в нашем случае *http://192.168.33.10.nip.io* и *http://www.192.168.33.10.nip.io*. Если в Mezzanine поступит запрос к другому хосту, сайт вернет ошибку «Bad Request (400)».

Нам нужно, чтобы эта строка в сгенерированном файле выглядела так:

```
ALLOWED_HOSTS = ["192.168.33.10.nip.io", "www.192.168.33.10.nip.io"]
```

Для этого можно использовать цикл for, как показано в примере 7.15. Но обратите внимание, что результат получается не совсем тот, которого мы добиваемся, – получающаяся строка содержит завершающую запятую:

```
ALLOWED_HOSTS = ["192.168.33.10.nip.io", "www.192.168.33.10.nip.io",]
```

Однако Python вполне устраивает наличие завершающей запятой в списке, и мы можем оставить все как есть.

Рассмотрим синтаксис цикла `for` в Jinja2. Чтобы было удобнее, разобьем его на несколько строк:

```
ALLOWED_HOSTS = [
{% for domain in domains %}
                "{{ domain }}",
{% endfor %}
               ]
```

Что такое nip.io?

Вероятно, вы заметили, что используемые доменные имена выглядят немного странно: *192.168.33.10.nip.io* и *www.192.168.33.10.nip.io*. Они включают также IP-адреса.

Переходя на сайт, вы практически всегда вводите в адресную строку браузера доменное имя, например *http://www.ansiblebook.com*, вместо его IP-адреса *http://151.101.192.133*. Когда мы создаем сценарий развертывания Mezzanine в Vagrant, то должны настроить доступные имена или доменные имена.

Проблема заключается в том, что у нас нет DNS-записи, отображающей имя виртуальной машины Vagrant в IP-адрес (в нашем случае *192.168.33.10*). Но ничто не мешает нам создать DNS-запись. Например, можно создать DNS-запись *mezzanine-internal.ansiblebook.com*, указывающую на *192.168.33.10*.

Однако, чтобы создать DNS-имя, которое разрешается в определенный IP-адрес, можно воспользоваться удобной службой *nip.io*. Она предоставляется бесплатно, и нам не придется создавать собственных DNS-записей. Если *AAA.BBB.CCC.DDD* – это IP-адрес, тогда *AAA.BBB.CCC.DDD.nip.io* – это DNS-запись, разрешающаяся в адрес *AAA.BBB.CCC.DDD*. Например, *192.168.33.10.nip.io* разрешается в *192.168.33.10*. Кроме того, *www.192.168.33.10.nip.io* тоже разрешается в *192.168.33.10*.

Мне кажется, что *nip.io* – очень удобный инструмент для развертывания веб-приложений с закрытыми IP-адресами с целью тестирования. С другой стороны, вы можете просто добавить записи в файл */etc/hosts* на локальной машине. Этот прием будет работать даже в отсутствие подключения к интернету.

Сгенерированный конфигурационный файл, все еще корректный с точки зрения Python, будет выглядеть, как показано ниже:

```
ALLOWED_HOSTS = [
                "192.168.33.10.nip.io",
```

```
    "www.192.168.33.10.nip.io",
]
```

Обратите внимание, что цикл `for` должен завершаться выражением `{% endfor %}`. Также отметьте, что инструкции `for` и `endfor` заключены в операторные скобки `{% %}`. Они отличаются от скобок `{{ }}`, которые мы используем для подстановки переменных.

Все переменные и факты, заданные в сценарии, доступны внутри шаблона Jinja2, т. е. нет необходимости явно передавать переменные в шаблон.

Выполнение команд django-manage

Приложения Django используют особый сценарий *manage.py* (**https://oreil.ly/BrUy8**) для выполнения следующих административных действий:

- создания таблиц в базе данных;
- выполнения миграций баз данных;
- загрузки начальных данных в базу из файла;
- записи данных из базы в файл;
- копирования статических данных в соответствующий каталог.

В дополнение к встроенным командам, которые поддерживает *manage.py*, приложения Django могут добавлять свои команды. Mezzanine, например, добавляет свою команду `createdb`, которая используется для приведения базы данных в исходное состояние и копирования статических ресурсов в надлежащее место. Официальные сценарии Fabric поддерживают аналогичные действия:

```
$ manage.py createdb --noinput --nodata
```

В состав Ansible входит модуль `django_manage`, который запускает команды `manage.py`. Мы можем использовать его так:

```
- name: Initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
```

К сожалению, команда `createdb`, которую добавляет Mezzanine, не является идемпотентной. При повторном запуске она завершится ошибкой:

```
TASK [initialize the database] ******************************************
fatal: [web]: FAILED! => {"changed": false, "cmd": "./manage.py createdb --
noinput --nodata", "msg": "\n:stderr: CommandError: Database already create
d, you probably want the migrate command\n", "path": "/home/vagrant/.virtua
lenvs/mezzanine_example/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/b
in:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin", "syspath": ["/tmp/ans
```

```
ible_django_manage_payload_4xfy5e7i/ansible_django_manage_payload.zip", "/u
sr/lib/python38.zip", "/usr/lib/python3.8", "/usr/lib/python3.8/lib-dynload
", "/usr/local/lib/python3.8/dist-packages", "/usr/lib/python3/dist-package
s"]}
```

К счастью, команда `createdb` эквивалентна двум идемпотентным встроенным командам из `manage.py`.

`migrate`
: Создает и обновляет таблицы базы данных для моделей Django.

`collectstatic`
: Копирует статические ресурсы в надлежащие каталоги.

Используя эти команды, можно реализовать идемпотентную задачу:

```
- name: Apply migrations to create the database, collect static content
  django_manage:
    command: "{{ item }}"
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
  loop:
    - syncdb
    - collectstatic
```

Запуск своих сценариев на Python в контексте приложения

Для инициализации нашего приложения необходимо внести два изменения в базу данных:

1) создать объект модели Site (**https://oreil.ly/COd8x**), содержащий доменное имя сайта (в нашем случае *192.168.33.10.nip.io*);
2) задать имя пользователя с правами администратора и пароль.

Несмотря на то что все это можно сделать с помощью простых SQL-команд, обычно это делается из кода на Python. Именно так решают эту задачу сценарии Fabric в Mezzanine, и мы тоже пойдем этим путем.

Здесь есть два подводных камня. Сценарии на Python должны запускаться в контексте созданного изолированного окружения, и окружение Python должно быть настроено так, чтобы сценарий импортировал файл *settings.py* из каталога *~/mezzanine/mezzanine_example/mezzanine_example*.

Когда нам требуется выполнить свой код на Python, мы обычно пишем свой модуль для Ansible. Однако, насколько нам известно, Ansible не позволяет запускать модули в контексте virtualenv. Поэтому данный вариант исключается.

Вместо этого мы используем модуль `script`. Он копируется поверх нестандартного сценария и выполняет его. Я написал два сценария: один для создания записи Site, а другой для создания учетной записи пользователя с правами администратора.

Модулю `script` можно передавать аргументы командной строки и анализировать их. Но мы решили передавать аргументы через переменные окружения. Нам не хотелось передавать пароли через командую строку (их можно увидеть в списке процессов, который выводит команда `ps`), а кроме того, переменные окружения легче проанализировать в сценариях, чем аргументы командной строки.

Ansible позволяет устанавливать переменные окружения посредством выражения `environment`, которое принимает словарь с именами и значениями переменных. Выражение `environment` можно добавить в любую задачу, если это не `script`.

Для запуска сценариев в контексте изолированного окружения virtualenv также необходимо установить переменную `path`, чтобы первый найденный выполняемый сценарий на Python оказался внутри virtualenv. В примере 7.16 показан пример запуска двух сценариев.

Пример 7.16. Использование модуля *script* для запуска кода на Python

```
- name: Set the site id
  script: scripts/setsite.py
  environment:
    PATH: "{{ venv_path }}/bin"
    PROJECT_DIR: "{{ proj_path }}"
    PROJECT_APP: "{{ proj_app }}"
    WEBSITE_DOMAIN: "{{ Uve_hostname }}"

- name: Set the admin password
  script: scripts/setadmin.py
  environment:
    PATH: "{{ venv_path }}/bin"
    PROJECT_DIR: "{{ proj_path }}"
    PROJECT_APP: "{{ proj_app }}"
    ADMIN_PASSWORD: "{{ admin_pass }}"
```

Сами сценарии приводятся в примерах 7.17 и 7.18. Вы найдете их в каталоге *scripts*.

Пример 7.17. *scripts/setsite.py*

```
#!/usr/bin/env python3
""" Сценарий настраивает домен сайта """
```

```
# Предполагается наличие трех переменных окружения
#
# PROJECT_DIR: корневой каталог проекта
# PROJECT_APP: имя проекта приложения
# WEBSITE_DOMAIN: домен сайта (например, www.example.com)
import os
import sys

# Добавить путь к каталогу проекта в переменную окружения PATH
proj_dir = os.path.expanduser(os.environ['PROJECT_DIR'])
sys.path.append(proj_dir)

proj_app = os.environ['PROJECT_APP']
os.environ['DJANGO_SETTINGS_MODULE'] = proj_app + '.settings'
import django
django.setup()
from django.conf import settings
from django.contrib.sites.models import Site
domain = os.environ['WEBSITE_DOMAIN']
Site.objects.filter(id=settings.SITE_ID).update(domain=domain)
Site.objects.get_or_create(domain=domain)
```

Пример 7.18. *scripts/setadmin.py*

```
#!/usr/bin/env python3
""" Сценарий настраивает учетную запись администратора """
# Предполагается наличие трех переменных окружения
#
# PROJECT_DIR: каталог проекта (например, ~/projname)
# PROJECT_APP: Имя проекта приложения
# ADMIN_PASSWORD: пароль администратора
import os
import sys

# добавить путь к каталогу проекта в переменную окружения PATH
proj_dir = os.path.expanduser(os.environ['PROJECT_DIR'])
sys.path.append(proj_dir)

proj_app = os.environ['PROJECT_APP']
os.environ['DJANGO_SETTINGS_MODULE'] = proj_app + '.settings'
import django
django.setup()
from django.contrib.auth import get_user_model
User = get_user_model()
u, _ = User.objects.get_or_create(username='admin')
u.is_staff = u.is_superuser = True
u.set_password(os.environ['ADMIN_PASSWORD'])
u.save()
```

Перед импортом `django` необходимо установить переменную окружения `DJANGO_SETTINGS_MODULE`.

Настройка конфигурационных файлов служб

Далее настроим конфигурационный файл для Gunicorn (сервера приложений), NGINX (веб-сервера) и Supervisor (диспетчер процессов), как показано в примере 7.19. Шаблон конфигурационного файла для Gunicorn показан в примере 7.21, а шаблон конфигурационного файла для Supervisor – в примере 7.22.

Пример 7.19. Настройка конфигурационных файлов

```
- name: Set the gunicorn config file
  template:
    src: templates/gunicorn.conf.py.j2
    dest: "{{ proj_path }}/gunicorn.conf.py"
    mode: '0750'

- name: Set the supervisor config file
  become: true
  template:
    src: templates/supervisor.conf.j2
    dest: /etc/supervisor/conf.d/mezzanine.conf
    mode: '0640'
  notify: Restart supervisor

- name: Set the nginx config file
  become: true
  template:
    src: templates/nginx.conf.j2
    dest: /etc/nginx/sites-available/mezzanine.conf
    mode: '0640'
  notify: Restart nginx
```

Во всех трех случаях конфигурационные файлы генерируются из шаблонов. Процессы Supervisor и NGINX запускаются с привилегиями пользователя root (хотя они тут же и понижают свои привилегии), поэтому нужно выполнить команду `become`, чтобы получить право доступа к конфигурационным файлам.

Если конфигурационный файл Supervisor изменится, то Ansible запустит обработчик `restart supervisor`. Если изменится конфигурационный файл NGINX, то Ansible запустит обработчик `restart nginx`, как показано в примере 7.20.

Пример 7.20. Обработчики

```
handlers:

  - name: Restart supervisor
    become: true
    supervisorctl:
      name: "{{ gunicorn_procname }}"
      state: restarted

  - name: Restart nginx
    become: true
    service:
      name: nginx
      state: restarted
```

Gunicorn имеет конфигурационный файл на языке Python; в нем мы передаем значения некоторых переменных.

Пример 7.21. *templates/gunicorn.conf.py.j2*

```
from multiprocessing import cpu_count

bind = "unix:{{ proj_path }}/gunicorn.sock"
workers = cpu_count() * 2 + 1
errorlog = "/home/{{ user }}/logs/{{ proj_name }}_error.log"
loglevel = "error"
proc_name = "{{ proj_name }}"
```

Конфигурационный файл Supervisor тоже просто определяет переменные.

Пример 7.22. *templates/supervisor.conf.j2*

```
[program:{{ gunicorn_procname }}]
command={{ venv_path }}/bin/gunicorn -c gunicorn.conf.py -p gunicorn.pid \
    {{ proj_app }}.wsgi:application
directory={{ proj_path }}
user={{ user }}
autostart=true
stdout_logfile = /home/{{ user }}/logs/{{ proj_name }}_supervisor
autorestart=true
redirect_stderr=true
environment=LANG="{{ locale }}",LC_ALL="{{ locale }}",LC_LANG="{{ locale }}"
```

В примере 7.23 приводится единственный шаблон, в котором используется дополнительная логика (кроме подстановки переменных). Он основан на логике выполнения по условию – если переменная `tls_enabled` имеет значение `true`, то включается поддержка TLS. В шаблоне в разных местах можно увидеть операторы `if`:

```
{% if tls_enabled %}
...
{% endif %}
```

В нем также используется Jinja2-фильтр `join`:

```
server_name {{ domains|join(", ") }};
```

Этот фрагмент кода ожидает, что переменная `domains` содержит список. Он сгенерирует строку с элементами из `domains`, перечислив их через запятую. В нашем случае список `domains` определен так:

```
domains:
  - 192.168.33.10.nip.io
  - www.192.168.33.10.nip.io
```

После применения шаблона получаем следующее:

```
server_name 192.168.33.10.nip.io, www.192.168.33.10.nip.io;
```

Пример 7.23. *templates/nginx.conf.j2*

```
upstream {{ proj_name }} {
    server unix:{{ proj_path }}/gunicorn.sock fail_timeout=0;
}
server {
    listen 80;
    {% if tls_enabled %}
    listen 443 ssl;
    {% endif %}
    server_name {{ domains|join(", ") }};
    server_tokens off;
    client_max_body_size 10M;
    keepalive_timeout    15;
    {% if tls_enabled %}
    ssl_certificate      conf/{{ proj_name }}.crt;
    ssl_certificate_key  conf/{{ proj_name }}.key;
    ssl_session_tickets off;
    ssl_session_cache    shared:SSL:10m;
    ssl_session_timeout  10m;
    ssl_protocols TLSv1.3;
    ssl_ciphers EECDH+AESGCM:EDH+AESGCM;
    ssl_prefer_server_ciphers on;
    {% endif %}
    location / {
        proxy_redirect      off;
        proxy_set_header    Host                    $host;
        proxy_set_header    X-Real-IP               $remote_addr;
        proxy_set_header    X-Forwarded-For         $proxy_add_x_forwarded_for;
        proxy_set_header    X-Forwarded-Protocol    $scheme;
```

```
        proxy_pass          http://{{ proj_name }};
    }
    location /static/ {
        root            {{ proj_path }};
        access_log      off;
        log_not_found   off;
    }
    location /robots.txt {
        root            {{ proj_path }}/static;
        access_log      off;
        log_not_found   off;
    }
    location /favicon.ico {
        root            {{ proj_path }}/static/img;
        access_log      off;
        log_not_found   off;
    }
}
```

Вы можете создавать шаблоны со структурами управления, такими как циклы `for` и условные инструкции `if/else`. Кроме того, Jinja2 поддерживает множество функций для преобразования данных из переменных, фактов и реестров в конфигурационные файлы.

Активация конфигурации NGINX

По соглашениям, принятым в Ubuntu для конфигурационных файлов NGINX, они должны помещаться в каталог */etc/nginx/sites-available* и активироваться символической ссылкой в каталоге */etc/nginx/sites-enabled*. (В системах Red Hat – в каталоге */etc/nginx/conf.d*.)

Сценарии Fabric для Mezzanine просто копируют конфигурационный файл непосредственно в *sites-enabled*. Но мы отклонимся от способа, принятого в Mezzanine, и используем модуль `file` для создания символической ссылки (пример 7.24). Также мы удалим конфигурационный файл, который пакет NGINX создает в */etc/nginx/sites-enabled/default*.

Пример 7.24. Активация конфигурации NGINX

```
- name: Remove the default nginx config file
  become: true
  file:
    path: /etc/nginx/sites-enabled/default
    state: absent
  notify: Restart nginx

- name: Set the nginx config file
  become: true
```

```
    template:
      src: templates/nginx.conf.j2
      dest: /etc/nginx/sites-available/mezzanine.conf
      mode: '0640'
    notify: Restart nginx

  - name: Enable the nginx config file
    become: true
    file:
      src: /etc/nginx/sites-available/mezzanine.conf
      dest: /etc/nginx/sites-enabled/mezzanine.conf
      state: link
      mode: '0777'
    notify: Restart nginx
```

Как показано в примере 7.24, мы использовали модуль `file`, чтобы создать символическую ссылку и удалить конфигурационный файл по умолчанию. Этот модуль удобно использовать для создания каталогов, символических ссылок и пустых файлов; удаления файлов, каталогов и символических ссылок; а также для настройки таких свойств, как разрешения и владение.

Установка сертификатов TLS

В нашем сценарии определяется переменная `tls_enabled`. Если она получает значение `true`, то сценарий установит сертификаты TLS. В нашем примере мы используем самоподписанный сертификат, поэтому сценарий создаст сертификат, если он не существует. В промышленном окружении необходимо скопировать существующий сертификат TLS, который вы получили от центра сертификации.

В примере 7.25 представлены две задачи, которые вовлечены в процесс настройки сертификатов TLS. Модуль `file` используется, чтобы при необходимости создать каталог для сертификатов TLS.

Пример 7.25. Установка сертификатов TLS

```
- name: Ensure config path exists
  become: true
  file:
    path: "{{ conf_path }}"
    state: directory
    mode: '0755'

- name: Create tls certificates
  become: true
  command: >
```

```
    openssl req -new -x509 -nodes -out {{ proj_name }}.crt
    -keyout {{ proj_name }}.key -subj '/CN={{ domains[0] }}' -days 365
  args:
    chdir: "{{ conf_path }}"
    creates: "{{ conf_path }}/{{ proj_name }}.crt"
  when: tls_enabled
  notify: Restart nginx
```

Обратите внимание, что обе задачи содержат выражение:

```
when: tls_enabled
```

Если `tls_enabled` имеет значение `false`, то Ansible пропустит задачу.

В Ansible нет модулей для создания сертификатов TLS, поэтому приходится использовать модуль `command` и с его помощью запускать команды `openssl` для создания самоподписанного сертификата. Поскольку команда очень длинная, мы используем YAML-синтаксис свертки строк с символом «>», чтобы разбить команду на несколько строк.

Параметр `chdir` изменяет каталог перед запуском команды. Параметр `creates` обеспечивает идемпотентность: Ansible сначала проверит наличие файла `{{ conf_path }}/{{ proj_name }}.crt` на хосте и, если он существует, пропустит эту задачу.

Установка задания cron для Twitter

Если выполнить команду `manage.py poll_twitter`, то Mezzanine извлечет твиты из настроенных учетных записей и поместит их на домашнюю страницу. Сценарии Fabric, поставляемые с Mezzanine, поддерживают актуальность сообщений с помощью задания `cron`, которое вызывается каждые пять минут.

Если в точности следовать за сценариями Fabric, мы должны скопировать сценарий с заданием `cron` в каталог */etc/cron.d*. Для этого можно бы использовать модуль `template`, но в состав Ansible входит модуль `cron`, который позволяет создавать и удалять задания `cron`, что, на наш взгляд, более изящно. В примере 7.26 представлена задача, которая устанавливает задание `cron`.

Пример 7.26. Установка задания *cron* для синхронизации с Twitter

```
- name: Install poll twitter cron job
  cron:
    name: "poll twitter"
    minute: "*/5"
    user: "{{ user }}"
    job: "{{ manage }} poll_twitter"
```

Если вручную подключиться к настраиваемой машине по SSH, то командой `crontab -l` можно проверить присутствие требуемого задания

в общем списке. Вот как выглядит это задание после установки на машине Vagrant:

```
#Ansible: poll twitter
*/5 * * * * /home/vagrant/.virtualenvs/mezzanine_example/bin/python3 \
/home/vagrant/mezzanine/mezzanine_example/manage.py poll_twitter
```

Обратите внимание на комментарий в первой строке. Благодаря таким комментариям модуль `cron` поддерживает удаление заданий по именам. Например:

```
- name: Remove cron job
  cron:
    name: "poll twitter"
    state: absent
```

Эта задача вызовет модуль `cron`, который отыщет строку комментария с указанным именем и удалит задание.

Сценарий целиком

В примере 7.27 представлен полный сценарий во всем своем великолепии.

Пример 7.27. *mezzanine.yml*: сценарий целиком

```
---
- name: Deploy mezzanine
  hosts: web

  vars:
    user: "{{ ansible_user }}"
    proj_app: 'mezzanine_example'
    proj_name: "{{ proj_app }}"
    venv_home: "{{ ansible_env.HOME }}/.virtualenvs"
    venv_path: "{{ venv_home }}/{{ proj_name }}"
    proj_path: "{{ ansible_env.HOME }}/mezzanine/{{ proj_name }}"
    settings_path: "{{ proj_path }}/{{ proj_name }}"
    reqs_path: '~/requirements.txt'
    manage: "{{ python }} {{ proj_path }}/manage.py"
    live_hostname: 192.168.33.10.nip.io
    domains:
      - 192.168.33.10.nip.io
      - www.192.168.33.10.nip.io
    repo_url: 'git@github.com:ansiblebook/mezzanine_example.git'
    locale: 'en_US.UTF-8'
    # Переменные ниже отсутствуют в сценарии fabfile.py установки Mezzanine
    # но я добавил их для удобства
    conf_path: /etc/nginx/conf
```

```
    tls_enabled: true
    python: "{{ venv_path }}/bin/python3"
    database_name: "{{ proj_name }}"
    database_user: "{{ proj_name }}"
    database_host: localhost
    database_port: 5432
    gunicorn_procname: gunicorn_mezzanine

  vars_files:
    - secrets.yml

  tasks:
    - name: Install apt packages
      become: true
      apt:
        update_cache: true
        cache_valid_time: 3600
        pkg:
          - acl
          - git
          - libjpeg-dev
          - libpq-dev
          - memcached
          - nginx
          - postgresql
          - python3-dev
          - python3-pip
          - python3-venv
          - python3-psycopg2
          - supervisor

    - name: Create project path
      file:
        path: "{{ proj_path }}"
        state: directory
        mode: '0755'

    - name: Create a logs directory
      file:
        path: "{{ ansible_env.HOME }}/logs"
        state: directory
        mode: '0755'

    - name: Check out the repository on the host
      git:
        repo: "{{ repo_url }}"
        dest: "{{ proj_path }}"
```

```
    version: master
    accept_hostkey: true

- name: Create python3 virtualenv
  pip:
    name:
      - pip
      - wheel
      - setuptools
    state: latest
    virtualenv: "{{ venv_path }}"
    virtualenv_command: /usr/bin/python3 -m venv

- name: Copy requirements.txt to home directory
  copy:
    src: requirements.txt
    dest: "{{ reqs_path }}"
    mode: '0644'

- name: Install packages listed in requirements.txt
  pip:
    virtualenv: "{{ venv_path }}"
    requirements: "{{ reqs_path }}"

- name: Create project locale
  become: true
  locale_gen:
    name: "{{ locale }}"

- name: Create a DB user
  become: true
  become_user: postgres
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"

- name: Create the database
  become: true
  become_user: postgres
  postgresql_db:
    name: "{{ database_name }}"
    owner: "{{ database_user }}"
    encoding: UTF8
    lc_ctype: "{{ locale }}"
    lc_collate: "{{ locale }}"
    template: template0

- name: Ensure config path exists
```

```
    become: true
    file:
      path: "{{ conf_path }}"
      state: directory
      mode: '0755'

  - name: Create tls certificates
    become: true
    command: >
      openssl req -new -x509 -nodes -out {{ proj_name }}.crt
      -keyout {{ proj_name }}.key -subj '/CN={{ domains[0] }}' -days 365
    args:
      chdir: "{{ conf_path }}"
      creates: "{{ conf_path }}/{{ proj_name }}.crt"
    when: tls_enabled
    notify: Restart nginx

  - name: Remove the default nginx config file
    become: true
    file:
      path: /etc/nginx/sites-enabled/default
      state: absent
    notify: Restart nginx

  - name: Set the nginx config file
    become: true
    template:
      src: templates/nginx.conf.j2
      dest: /etc/nginx/sites-available/mezzanine.conf
      mode: '0640'
    notify: Restart nginx

  - name: Enable the nginx config file
    become: true
    file:
      src: /etc/nginx/sites-available/mezzanine.conf
      dest: /etc/nginx/sites-enabled/mezzanine.conf
      state: link
      mode: '0777'
    notify: Restart nginx

  - name: Set the supervisor config file
    become: true
    template:
      src: templates/supervisor.conf.j2
      dest: /etc/supervisor/conf.d/mezzanine.conf
      mode: '0640'
```

```
    notify: Restart supervisor

  - name: Install poll twitter cron job
    cron:
      name: "poll twitter"
      minute: "*/5"
      user: "{{ user }}"
      job: "{{ manage }} poll_twitter"

  - name: Set the gunicorn config file
    template:
      src: templates/gunicorn.conf.py.j2
      dest: "{{ proj_path }}/gunicorn.conf.py"
      mode: '0750'

  - name: Generate the settings file
    template:
      src: templates/local_settings.py.j2
      dest: "{{ settings_path }}/local_settings.py"
      mode: '0750'

  - name: Apply migrations to create the database, collect static content
    django_manage:
      command: "{{ item }}"
      app_path: "{{ proj_path }}"
      virtualenv: "{{ venv_path }}"
    with_items:
      - migrate
      - collectstatic

  - name: Set the site id
    script: scripts/setsite.py
    environment:
      PATH: "{{ venv_path }}/bin"
      PROJECT_DIR: "{{ proj_path }}"
      PROJECT_APP: "{{ proj_app }}"
      DJANGO_SETTINGS_MODULE: "{{ proj_app }}.settings"
      WEBSITE_DOMAIN: "{{ live_hostname }}"

  - name: Set the admin password
    script: scripts/setadmin.py
    environment:
      PATH: "{{ venv_path }}/bin"
      PROJECT_DIR: "{{ proj_path }}"
      PROJECT_APP: "{{ proj_app }}"
      ADMIN_PASSWORD: "{{ admin_pass }}"

handlers:
```

```
    - name: Restart supervisor
      become: true
      supervisorctl:
        name: "{{ gunicorn_procname }}"
        state: restarted

    - name: Restart nginx
      become: true
      service:
        name: nginx
        state: restarted
...
```

Сценарии могут получаться длиннее, чем хотелось бы, и такие сценарии, в которых все действия и переменные перечислены в одном файле, сложнее поддерживать. Поэтому этот сценарий следует рассматривать лишь как промежуточный шаг в процессе обучении работе с Ansible. В следующей главе будет представлен более удобный способ организации сценариев.

Запуск сценария на машине Vagrant

Переменные `live_hostname` и `domains` в нашем сценарии предполагают, что хост, на котором должна быть развернута система, доступен по адресу *192.168.33.10*. Файл Vagrantfile, что приводится в примере 7.28, настраивает машину Vagrant с этим IP-адресом.

Пример 7.28. *Vagrantfile*

```
Vagrant.configure("2") do |this|
  # Агент перенаправления ssh для клонирования из Github.com
  this.ssh.forward_agent = true
  this.vm.define "web" do |web|
    web.vm.box = "ubuntu/focal64"
    web.vm.hostname = "web"
    # Этот IP используется в сценарии
    web.vm.network "private_network", ip: "192.168.33.10"
    web.vm.network "forwarded_port", guest: 80, host: 8000
    web.vm.network "forwarded_port", guest: 443, host: 8443
    web.vm.provider "virtualbox" do |virtualbox|
      virtualbox.name = "web"
    end
  end
  this.vm.provision "ansible" do |ansible|
    ansible.playbook = "mezzanine.yml"
    ansible.verbose = "v"
    ansible.compatibility_mode = "2.0"
```

```
    ansible.host_key_checking = false
  end
end
```

Развертывание Mezzanine на новой машине Vagrant автоматически выполняется блоком `provision` после запуска команды:

```
$ vagrant up
```

После развертывания нового сайта Mezzanine он будет доступен по любому из перечисленных ниже адресов:

- *http://192.168.33.10.nip.io*;
- *https://192.168.33.10.nip.io*;
- *http://www.192.168.33.10.nip.io*;
- *https://www.192.168.33.10.nip.io*.

Устранение проблем

При попытке выполнить сценарий на локальной машине вы можете столкнуться с несколькими проблемами. В этом разделе описываются некоторые типичные проблемы и способы их преодоления.

Не получается извлечь файлы из репозитория Git

Вы можете увидеть, как задача с именем «check out the repository on the host» завершается со следующей ошибкой:

```
fatal: Could not read from remote repository.
```

Для ее исправления удалите предопределенный элемент для *192.168.33. 10* из файла *~/.ssh/known_hosts*.

Недоступен хост с адресом 192.168.33.10.nip.io

Некоторые маршрутизаторы WiFi имеют встроенный сервер DNS, который не распознает имя хоста *192.168.33.10.nip.io*. Проверить это можно следующей командой:

```
dig +short 192.168.33.10.nip.io
```

Она должна вывести:

```
192.168.33.10
```

Если выводится пустая строка, значит, ваш сервер DNS не распознает имена хостов *nip.io*. В этом случае добавьте в свой файл */etc/hosts* следующую строку:

```
192.168.33.10 192.168.33.10.nip.io
```

Bad Request (400)

Если ваш браузер вывел сообщение об ошибке «Bad Request (400)», это, скорее всего, связано с попыткой достичь сайта Mezzanine с использованием имени хоста или IP-адреса, который не включен в список `ALLOWED_HOSTS` в конфигурационном файле Mezzanine. Этот список заполняется по содержимому переменной `domains`, объявленной в сценарии Ansible:

```
domains:
  - 192.168.33.10.nip.io
  - www.192.168.33.10.nip.io
```

Заключение

В этом сценарии мы полностью развернули Mezzanine на одной машине. Теперь вы знаете, как осуществляется развертывание обычного приложения с поддержкой Mezzanine.

В следующей главе мы рассмотрим более продвинутые функции Ansible, не использовавшиеся в нашем примере. Мы покажем сценарий, который устанавливает базу данных и веб-службы на разные хосты, что часто бывает нужно в реальной ситуации.

Глава 8

Отладка сценариев Ansible

Давайте признаем – ошибки случаются. Ошибка ли это в сценарии, или же неверное значение в файле конфигурации, в любом случае, что-то идет не так. В этой главе мы рассмотрим приемы, позволяющие вылавливать эти ошибки.

Информативные сообщения об ошибках

Когда задача Ansible терпит неудачу, она выводит сообщение не в самом удобном формате для человека, пытающегося найти причину проблемы. Вот пример сообщения об ошибке, с которой мы столкнулись, работая над этой книгой:

```
TASK [mezzanine : check out the repository on the host]
**********************
fatal: [web]: FAILED! => {"changed": false, "cmd": "/usr/bin/git ls-remote
'' -h refs/heads/master", "msg": "Warning:********@github.com: Permission
denied (publickey).\r\nfatal: Could not read from remote
repository.\n\nPlease make sure you have the correct access rights\nand the
repository exists.", "rc": 128, "stderr": "Warning: Permanently added
'github.com,140.82.121.4' (RSA) to the list of known
hosts.\r\ngit@github.com: Permission denied (publickey).\r\nfatal: Could not
read from remote repository.\n\nPlease make sure you have the correct access
rights\nand the repository exists.\n", "stderr_lines": ["Warning:
Permanently added 'github.com,140.82.121.4' (RSA) to the list of known
hosts.", "git@github.com: Permission denied (publickey).", "fatal: Could not
read from remote repository.", "", "Please make sure you have the correct
access rights", "and the repository exists."], "stdout": "", "stdout_lines":
[]}
```

Как отмечается в главе 18, плагин `debug` может привести это сообщение к более удобочитаемому виду:

```
TASK [mezzanine : check out the repository on the host] **********************
fatal: [web]: FAILED! => {
    "changed": false,
    "cmd": "/usr/bin/git ls-remote '' -h refs/heads/master",
```

```
    "rc": 128
}
STDERR:
git@github.com: Permission denied (publickey).
fatal: Could not read from remote repository.
Please make sure you have the correct access rights
and the repository exists.
```

Чтобы включить плагин, достаточно добавить следующую строку в раздел `defaults` в файле *ansible.cfg*:

```
[defaults]
stdout_callback = debug
```

Но имейте в виду, что плагин `debug` выводит не всю информацию; более подробную информацию можно получить с помощью плагина YAML.

Отладка ошибок с SSH-подключением

Иногда системе Ansible не удается установить SSH-соединение с хостом. Давайте посмотрим, что она сообщает, если SSH-сервер вообще не отвечает на запросы:

```
$ ansible web -m ping
web | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh:
kex_exchange_identification: Connection closed by remote host",
    "unreachable": true
}
```

Такое поведение может быть обусловлено несколькими причинами:

- сервер SSH вообще не запущен;
- сервер SSH прослушивает нестандартный порт;
- порт, к которому вы пытаетесь подключиться, обслуживается каким-то другим сервером;
- порт может фильтроваться брандмауэром на вашем хосте;
- порт может фильтроваться брандмауэром на другом хосте;
- настройки контроля доступа TCP Wrappers, проверьте */etc/hosts.allow* и */etc/hosts.deny*;
- хост работает в гипервизоре с микросегметацией.

Убедившись в системной консоли, что SSH-сервер запущен и работает на хосте, можно попытаться подключиться удаленно с помощью `nc` или даже клиента *telnet*, чтобы проверить отклик:

```
$ nc hostname 2222
SSH-2.0-OpenSSH_8.2p1 Ubuntu-4ubuntu0.4
```

Затем можно попробовать удаленно подключиться с помощью SSH-клиента, используя флаг вывода подробной информации для отладки:

```
$ ssh -v user@hostname
```

Также полезно проверить, какие аргументы Ansible передает SSH-клиенту, и воспроизвести действие вручную в командной строке. Если вызвать `ansible` с аргументом `-vvv`, можно увидеть, как именно Ansible вызывает SSH. Это может пригодиться для отладки.

```
$ ansible web -vvv -m ping
```

В примере 8.1 показаны часть вывода этой команды.

Пример 8.1. Пример вывода команды *ansible* с аргументом *-vvv*

```
<127.0.0.1> SSH: EXEC ssh -vvv -4 -o PreferredAuthentications=publickey -o
        ForwardAgent=yes -o StrictHostKeyChecking=no -o Port=2200 -o
        'IdentityFile="/Users/bas/.vagrant.d/insecure_private_key"' -o
        KbdInteractiveAuthentication=no -o
        PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey -o
        PasswordAuthentication=no -o 'User="vagrant"' -o ConnectTimeout=10 127.0.0.1
        '/bin/sh -c '"'"'rm -f -r
        /home/vagrant/.ansible/tmp/ansible-tmp-1633182008.6825979-95820-
        137028099318259/ > /dev/null 2>&1 && sleep 0'"'"''
        <127.0.0.1> (0, b'', b'OpenSSH_8.1p1, LibreSSL 2.7.3\r\ndebug1: Reading
        configuration data /Users/bas/.ssh/config\r\ndebug3: kex names ok:
        [curve25519-sha256,diffie-hellman-group-exchange-sha256]\r\ndebug1: Reading
        configuration data /etc/ssh/ssh_config\r\ndebug1: /etc/ssh/ssh_config line
        20: Applying options for *\r\ndebug1: /etc/ssh/ssh_config line 47: Applying
        options for *\r\ndebug2: resolve_canonicalize: hostname 127.0.0.1 is
        address\r\ndebug1: auto-mux: Trying existing master\r\ndebug2: fd 3 setting
        O_NONBLOCK\r\ndebug2: mux_client_hello_exchange: master version 4\r\ndebug3:
        mux_client_forwards: request forwardings: 0 local, 0 remote\r\ndebug3:
        mux_client_request_session: entering\r\ndebug3: mux_client_request_alive:
        entering\r\ndebug3: mux_client_request_alive: done pid = 95516\r\ndebug3:
        mux_client_request_session: session request sent\r\ndebug3:
        mux_client_read_packet: read header failed: Broken pipe\r\ndebug2: Received
        exit status from master 0\r\n')
        web | SUCCESS => {
                "changed": false,
                "invocation": {
                        "module_args": {
                                "data": "pong"
                        }
                },
                "ping": "pong"
        }
```

Иногда при отладке проблем с подключением может даже понадобиться использовать флаг `-vvvv`, чтобы увидеть сообщение об ошибке, возвращаемое SSH-клиентом. Это равносильно добавлению флага `-v` в команду `ssh`, которую использует Ansible:

```
$ ansible all -vvvv -m ping
```

В примере 8.2 показано, какой большой объем отладочной информации можно получить в этом случае.

Пример 8.2. Пример вывода команды *ansible* с аргументом *-vvvv*

```
<192.168.56.10> ESTABLISH SSH CONNECTION FOR USER: vagrant
<192.168.56.10> SSH: EXEC ssh -vvv -4 -o PreferredAuthentications=publickey
-o ForwardAgent=yes -o StrictHostKeyChecking=no -o
'IdentityFile="/Users/bas/.vagrant.d/insecure_private_key"' -o
KbdInteractiveAuthentication=no -o
PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey -o
PasswordAuthentication=no -o 'User="vagrant"' -o ConnectTimeout=10
192.168.56.10 '/bin/sh -c '"'"'/usr/bin/python3 && sleep 0'"'"''
debug1: Reading configuration data /Users/bas/.ssh/config
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: /etc/ssh/ssh_config line 21: include /etc/ssh/ssh_config.d/* matched
no files
debug1: /etc/ssh/ssh_config line 54: Applying options for *
debug1: Authenticator provider $SSH_SK_PROVIDER did not resolve; disabling
debug1: Connecting to 192.168.56.10 [192.168.56.10] port 22.
debug1: fd 3 clearing O_NONBLOCK
debug1: Connection established.
debug1: identity file /Users/bas/.vagrant.d/insecure_private_key type -1
debug1: Local version string SSH-2.0-OpenSSH_8.6
debug1: Remote protocol version 2.0, remote software version OpenSSH_8.2p1
Ubuntu-4ubuntu0.5
debug1: compat_banner: match: OpenSSH_8.2p1 Ubuntu-4ubuntu0.5 pat OpenSSH*
compat 0x04000000
debug1: Authenticating to 192.168.56.10:22 as \'vagrant\'
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: algorithm: curve25519-sha256
debug1: kex: host key algorithm: ssh-ed25519
debug1: kex: server->client cipher: chacha20-poly1305@openssh.com MAC:
<implicit> compression: none
debug1: kex: client->server cipher: chacha20-poly1305@openssh.com MAC:
<implicit> compression: none
debug1: expecting SSH2_MSG_KEX_ECDH_REPLY
debug1: SSH2_MSG_KEX_ECDH_REPLY received
debug1: Server host key: ssh-ed25519
SHA256:BnlxL1InYlrSLQU10HFYzg6ZZkj1boxRSloEsK3bpxA
```

```
debug1: Host \'192.168.56.10\' is known and matches the ED25519 host key.
debug1: Found key in /Users/bas/.ssh/known_hosts:57
debug1: rekey out after 134217728 blocks
debug1: SSH2_MSG_NEWKEYS sent
debug1: expecting SSH2_MSG_NEWKEYS
debug1: SSH2_MSG_NEWKEYS received
debug1: rekey in after 134217728 blocks
debug1: Will attempt key: /Users/bas/.vagrant.d/insecure_private_key
explicit
debug1: SSH2_MSG_EXT_INFO received
debug1: kex_input_ext_info:
server-sig-algs=<ssh-ed25519,sk-ssh-ed25519@openssh.com,ssh-rsa,rsa-sha2-256
,rsa-sha2-512,ssh-dss,ecdsa-sha2-nistp256,ecdsa-sha2-nistp384,ecdsa-sha2-
nistp521,sk-ecdsa-sha2-nistp256@openssh.com>
debug1: SSH2_MSG_SERVICE_ACCEPT received
debug1: Authentications that can continue: publickey
debug1: Next authentication method: publickey
debug1: Trying private key: /Users/bas/.vagrant.d/insecure_private_key
debug1: Authentication succeeded (publickey).
Authenticated to 192.168.56.10 ([192.168.56.10]:22).
debug1: channel 0: new [client-session]
debug1: Requesting no-more-sessions@openssh.com
debug1: Entering interactive session.
debug1: pledge: filesystem full
debug1: client_input_global_request: rtype hostkeys-00@openssh.com
want_reply 0
debug1: client_input_hostkeys: searching /Users/bas/.ssh/known_hosts for
192.168.56.10 / (none)
debug1: client_input_hostkeys: no new or deprecated keys from server
debug1: Remote: /home/vagrant/.ssh/authorized_keys:1: key options:
agent-forwarding port-forwarding pty user-rc x11-forwarding
debug1: Requesting authentication agent forwarding.
debug1: Sending environment.
debug1: channel 0: setting env LC_TERMINAL_VERSION = "3.4.16"
debug1: channel 0: setting env LC_CTYPE = "UTF-8"
debug1: channel 0: setting env LC_TERMINAL = "iTerm2"
debug1: Sending command: /bin/sh -c \'/usr/bin/python3 && sleep 0\'
debug1: client_input_channel_req: channel 0 rtype exit-status reply 0
debug1: channel 0: free: client-session, nchannels 1
Transferred: sent 117208, received 1664 bytes, in 0.4 seconds
Bytes per second: sent 284246.0, received 4035.4
debug1: Exit status 0
')
web | SUCCESS => {
    "changed": false,
    "invocation": {
        "module_args": {
```

```
            "data": "pong"
        }
    },
    "ping": "pong"
}
META: ran handlers
META: ran handlers
```

Вы должны знать, что `"ping": "pong"` означает успешное соединение, даже если ему предшествуют отладочные сообщения.

Типичные проблемы с SSH

Для управления хостами Ansible подключается к ним через SSH нередко с правами администратора. Поэтому важно знать о проблемах с безопасностью, которые поначалу могут озадачить обычных пользователей.

PasswordAuthentication no

Параметр `PasswordAuthentication no` значительно повышает безопасность ваших серверов. По умолчанию Ansible предполагает, что подключение к удаленным машинам производится с использованием ключей SSH. Иметь пару ключей SSH недостаточно, необходимо также скопировать открытый ключ на машины, которыми вы собираетесь управлять. Традиционно это делается с помощью команды `ssh-copy-id`, но когда параметр `PasswordAuthentication` имеет значение `no`, администратор должен использовать учетную запись с открытыми ключами, чтобы скопировать ваш открытый ключ на серверы, желательно с помощью модуля `authorized_keys`:

```
- name: Install authorized_keys taken from file
  authorized_key:
    user: "{{ the_user }}"
    state: present
    key: "{{ lookup('file',the_pub_key)  }}"
    key_options: 'no-port-forwarding,from="93.184.216.34"'
    exclusive: true
```

Обратите внимание, что открытые ключи ed25519 достаточно короткие, и при необходимости их можно ввести в консоли.

Подключение по SSH с учетными данными другого пользователя

К разным хостам можно подключаться, используя учетные данные разных пользователей. По возможности старайтесь ограничивать вход

в систему с учетными данными пользователя root. Если подключаться к каждой машине нужно с учетными данными конкретного пользователя, то настройте переменную `ansible_user` в реестре:

```
[mezzanine]
web ansible_host=192.168.33.10 ansible_user=webmaster
db  ansible_host=192.168.33.11 ansible_user=dba
```

Обратите внимание, что при необходимости можно указать другого пользователя в командной строке:

```
$ ansible-playbook --user vagrant -i inventory/hosts mezzanine.yml
```

Также можно задать пользователя для каждого хоста в конфигурационном файле SSH. Наконец, в заголовке операции (play) можно задать переменную `remote_user` для каждой задачи.

Ошибка проверки ключа хоста

Иногда при попытке подключиться к машине можно получить такое сообщение об ошибке:

```
$ ansible -m ping web
web | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\r\n@    WARNING:
REMOTE HOST IDENTIFICATION HAS CHANGED!
@\r\n@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\r\nIT IS
POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!\r\nSomeone could be
eavesdropping on you right now (man-in-the-middle attack)!\r\nIt is also
possible that a host key has just been changed.\r\nThe fingerprint for the
ED25519 key sent by the remote host
is\nSHA256:+dX3jRW5eoZ+FzQP9jc6cIALXugh9bftvYvaQig+33c.\r\nPlease contact
your system administrator.\r\nAdd correct host key in
/Users/bas/.ssh/known_hosts to get rid of this message.\r\nOffending ED25519
key in /Users/bas/.ssh/known_hosts:2\r\nED25519 host key for 192.168.56.10
has changed and you have requested strict checking.\r\nHost key verification
failed.",
    "unreachable": true
}
```

В этом случае не отключайте `StrictHostKeyChecking` в конфигурации SSH, а просто удалите старый ключ хоста и добавьте новый:

```
ssh-keygen -R 192.168.33.10
ssh-keyscan 192.168.33.10 >> ~/.ssh/known_hosts
```

Частные сети

Поскольку по умолчанию Ansible использует клиента OpenSSH, вы можете использовать *хост-бастион*: центральную точку в DMZ для доступа к другим хостам в частной сети. В следующем примере все хосты, находящиеся в домене *private.cloud*, доступны через хост-бастион, указанный в `ProxyJump` в файле *~/.ssh/config*:

```
Host bastion
  Hostname 100.123.123.123
  User bas
  PasswordAuthentication no
Host *.private.cloud
  User bas
  CheckHostIP no
  StrictHostKeyChecking no
  ProxyJump bastion
```

Если бастион настроен с помощью VPN, то вам не нужно использовать SSH через интернет. Tailscale (**https://tailscale.com/**) – простой в использовании сервер VPN на основе WireGuard (**https://www.wireguard.com/**), который пропускает трафик от клиентов через бастион к другим хостам в частной подсети, не требуя выполнять дополнительные настройки на этих хостах.

Модуль debug

В этой книге мы уже использовали модуль `debug` несколько раз. Это аналог инструкции `print` в синтаксисе Ansible. Его можно использовать для вывода значений переменных и произвольных строк, как показано в примере 8.3.

Пример 8.3. Модуль debug в действии

```
- debug: var=myvariable
- debug: msg="The value of myvariable is {{ var }}"
```

Как уже говорилось в главе 5, можно вывести значения всех переменных, связанных с текущим хостом, как показано ниже:

```
- debug: var=hostvars[inventory_hostname]
```

Интерактивный отладчик сценариев

В Ansible 2.5 была добавлена поддержка интерактивного отладчика. Включить или отключить отладчик для конкретной операции, роли, блока или задачи можно с помощью ключевого слова `debugger`:

```
- name: deploy mezzanine on web
  hosts: web
  debugger: always
    ...
```

Если отладка включена, как в этом примере, то Ansible запустит отладчик, и вы сможете выполнять отдельные шаги в сценарии, вводя `c` (continue – продолжить):

```
PLAY [deploy mezzanine on web] *************************************************
TASK [mezzanine : install apt packages] ****************************************
changed: [web]
[web] TASK: mezzanine : install apt packages (debug)> c
TASK [mezzanine : create a logs directory] *************************************
changed: [web]
[web] TASK: mezzanine : create a logs directory (debug)> c
```

В табл. 8.1 перечислены команды, поддерживаемых отладчиком.

Таблица 8.1. Команды отладчика

Команда	Сокращение	Описание
`print`	p	Вывести значение переменой
`task.args[key] = value`	нет	Изменить аргументы задачи
`task_vars[key] = value`	нет	Изменить переменные задачи (после этой команды необходимо выполнить команду `update_task`, которая описывается ниже)
`update_task`	u	Создать задачу заново с обновленными переменными
`redo`	r	Повторно запустить задачу
`continue`	c	Продолжить выполнение, начиная со следующей задачи
`quit`	q	Выйти из отладчика

В табл. 8.2 перечислены переменные, поддерживаемые отладчиком .

Таблица 8.2. Переменные, поддерживаемые отладчиком

Переменная	Описание
`p task`	Имя задачи, где возникла ошибка
`p task.args`	Аргументы модуля
`p result`	Результат, который вернула задача, допустившая ошибку
`p vars`	Значения всех известных переменных
`p vars[key]`	Значение одной переменной *key*

Вот пример сеанса работы с отладчиком:

```
 TASK [mezzanine : install apt packages ************************************
ok: [web]
[web] TASK: mezzanine : install apt packages (debug)> p task.args
{'_ansible_check_mode': False,
 '_ansible_debug': False,
 '_ansible_diff': False,
 '_ansible_keep_remote_files': False,
 '_ansible_module_name': 'apt',
 '_ansible_no_log': False,
 '_ansible_remote_tmp': '~/.ansible/tmp',
 '_ansible_selinux_special_fs': ['fuse',
                                 'nfs',
                                 'vboxsf',
                                 'ramfs',
                                 '9p',
                                 'vfat'],
 '_ansible_shell_executable': '/bin/sh',
 '_ansible_socket': None,
 '_ansible_string_conversion_action': 'warn',
 '_ansible_syslog_facility': 'LOG_USER',
 '_ansible_tmpdir': '/home/vagrant/.ansible/tmp/ansible-tmp-1633193380-7157/',
 '_ansible_verbosity': 0,
 '_ansible_version': '2.11.0',
 'cache_valid_time': 3600,
 'pkg': ['git',
         'libjpeg-dev',
         'memcached',
         'python3-dev',
         'python3-pip',
         'python3-venv',
         'supervisor'],
 'update_cache': True}
```

Вывод значений переменных – одна из самых полезных возможностей, однако отладчик позволяет также изменять переменные и аргументы задач, потерпевших неудачу. За более подробной информацией обращайтесь к документации с описанием отладчика Ansible (**https://oreil.ly/IZSCl**).

Модуль assert

Модуль `assert` завершает сценарий с сообщением об ошибке при невыполнении заданного условия. Например, сценарий завершится с ошибкой, если не будет найден сетевой интерфейс `enp0s3`:

```
- name: Assert that the enp0s3 ethernet interface exists
  assert:
    that: ansible_enp0s3 is defined
```

Такая проверка тех или иных условий может очень пригодиться при отладке сценария.

В устаревших сценариях или ролях можно увидеть, что отладчик включается в виде стратегии. Такой способ включения отладчика может не поддерживаться в более новых версиях Ansible. При включенной стратегии по умолчанию линейного выполнения (linear) Ansible останавливает выполнение, пока активен отладчик, а затем запускает отлаживаемую задачу после ввода команды `redo`. Однако с включенной свободной (free) стратегией Ansible не ждет завершения отлаживаемой задачи на всех хостах и может поставить в очередь более поздние задачи на одном хосте, прежде чем задача завершится ошибкой на другом хосте. Пока отладчик активен, она не ставит в очередь и не выполняет никаких задач, однако все задачи, поставленные в очередь, остаются в очереди и запускаются сразу после выхода из отладчика. Узнать больше о стратегиях можно в документации (**https://oreil.ly/bLqah**).

К сожалению, движок Jinja2 не поддерживает функцию `len` из Python. Вместо нее следует использовать Jinja2-фильтр `length`:

```
assert:
  that: "ports|length == 1"
```

Имейте в виду, что код в выражении `assert` – это инструкции Jinja2, а не Python. Например, для проверки длины списка так соблазнительно использовать такой код:

```
# Недопустимый для Jinja2 код, который не будет работать!
assert:
  that: "len(ports) == 1"
```

Чтобы проверить статус файла в файловой системе хоста, можно сначала вызвать модуль `stat` и добавить проверку возвращаемого модулем значения:

```
- name: Stat /boot/grub
  stat:
    path: /boot/grub
```

```
    register: st

- name: Assert that /boot/grub is a directory
  assert:
    that: st.stat.isdir
```

Модуль `stat` собирает информацию о файле и возвращает словарь, содержащий поле `stat` со значениями, перечисленными в табл. 8.3.

Таблица 8.3. Возвращаемые значения модуля *stat* (некоторые платформы могут добавлять дополнительные поля)

Поле	Описание
`atime`	Время последнего доступа к файлу в формате меток времени Unix
`attributes`	Список атрибутов файла
`charset`	Кодировка символов в файле
`checksum`	Значение хеша файла
`ctime`	Время создания в формате меток времени Unix
`dev`	Числовой идентификатор устройства, где находится данный индексный узел
`executable`	`True`, если текущий пользователь имеет разрешение на выполнение файла
`exists`	`True`, если путь существует
`gid`	Числовой идентификатор группы-владельца
`gr_name`	Имя группы-владельца
`inode`	Номер индексного узла
`isblk`	`True`, если файл – специальный файл блочного устройства
`ischr`	`True`, если файл – специальный файл символьного устройства
`isdir`	`True`, если файл – каталог
`isfifo`	`True`, если файл – именованный канал
`isgid`	`True`, если идентификатор группы текущего пользователя совпадает с идентификатором группы-владельца
`islnk`	`True`, если файл – символическая ссылка
`isreg`	`True`, если файл – обычный файл
`issock`	`True`, если файл – сокет домена Unix
`isuid`	`True`, если идентификатор текущего пользователя совпадает с идентификатором владельца
`lnk_source`	Цель символической ссылки в удаленной файловой системе в нормализованном виде

Поле	Описание
`lnk_target`	Цель символической ссылки
`mimetype`	Тип файла
`mode`	Режим доступа к файлу в виде строки (например, «1177»)
`mtime`	Время последнего изменения в формате меток времени Unix
`nlink`	Количество жестких ссылок на файл
`pw_name`	Имя пользователя владельца файла
`readable`	`True`, если дано разрешение на чтение для текущего пользователя
`rgrp`	`True`, если дано разрешение на чтение для группы
`roth`	`True`, если дано разрешение на чтение для остальных
`rusr`	`True`, если дано разрешение на чтение для пользователя-владельца
`size`	Размер файла в байтах, если это обычный файл; объем данных для некоторых специальных файлов
`uid`	Числовой идентификатор пользователя владельца
`wgrp`	`True`, если дано разрешение на запись для группы
`woth`	`True`, если дано разрешение на запись для остальных
`writeable`	`True`, если дано разрешение на чтение для текущего пользователя
`wusr`	`True`, если дано разрешение на запись для пользователя-владельца
`xgrp`	`True`, если дано разрешение на выполнение для группы
`xoth`	`True`, если дано разрешение на выполнение для остальных
`xusr`	`True`, если дано разрешение на выполнение для пользователя

Проверка сценария перед запуском

Команда `ansible-playbook` поддерживает несколько флагов, позволяющих провести проверку сценария перед запуском. При использовании этих флагов сценарий *не* запускается.

Проверка синтаксиса

Как показано в примере 8.4, флаг `--syntax-check` включает проверку допустимости синтаксиса сценария.

Пример 8.4. Проверка синтаксиса

```
$ ansible-playbook --syntax-check playbook.yml
```

Список хостов

Как показано в примере 8.5, флаг `--list-hosts` выводит список хостов, на которых будет выполняться сценарий.

Пример 8.5. Список хостов

```
$ ansible-playbook --list-hosts playbook.yml
```

Иногда можно получить раздражающее предупреждение:

```
[WARNING]: provided hosts list is empty, only localhost
is available. Note that the implicit localhost does not
match 'all'
[WARNING]: Could not match supplied host pattern,
ignoring: db
[WARNING]: Could not match supplied host pattern,
ignoring: web
```

В реестре явно должен быть указан хотя бы один хост, иначе Ansible вернет это предупреждение, даже если сценарий выполняется только на локальном хосте. При пустом реестре (например, если используется сценарий динамической инвентаризации и в данный момент ни один хост не запущен) можно предотвратить появление этого сообщения, добавив в реестр группы:

```
ansible-playbook --list-hosts -i web,db playbook.yml
```

Список задач

Как показано в примере 8.6, флаг `--list-tasks` выводит список задач, которые запускает сценарий.

Пример 8.6. Список задач

```
$ ansible-playbook --list-tasks playbook.yml
```

Мы уже использовали этот флаг в примере 7.1 для вывода списка задач в нашем первом сценарии, развертывающем приложение Mezzanine. Напомню еще раз, что ни с одним из флагов, упоминаемых в этом разделе, команда `ansible-playbook` не запускает сценарий, а только проверяет его.

Режим проверки

Флаги `-C` и `--check` запускают Ansible в *режиме проверки* (также известном как *dry run* – сухой прогон), который показывает, изменила бы каж-

дая задача состояние хоста, но при этом никакие изменения фактически не выполняются.

```
$ ansible-playbook -C playbook.yml
$ ansible-playbook --check playbook.yml
```

Одна из сложностей использования режима проверки – правильная оценка успешности выполнения последующих частей сценария, зависящих от выполнения предыдущих. Если запустить в режиме проверки сценарий из примера 7.28, то он вернет признак ошибки, как показано в примере 8.7, потому что данная задача зависит от предыдущей, устанавливающей программу NGINX на хост. Еще одна сложность: модули, используемые в сценарии, должны поддерживать режим проверки, иначе проверка будет терпеть неудачу.

Пример 8.7. Ошибка при выполнении корректного сценария в режиме проверки

```
TASK [nginx : create ssl certificates] ***************************************
fatal: [web]: FAILED! => {
    "changed": false
}
MSG:
Unable to change directory before execution: [Errno 2] No such file or directory:
b'/etc/nginx/conf'
```

Подробнее о поддержке режима проверки модулями рассказывается в главе 19.

Вывод изменений в файлах

Флаги `-D` и `-diff` выводят информацию о любых изменениях, сделанных в файлах на удаленной машине. Этот флаг удобно использовать вместе с `--check`, чтобы увидеть, как Ansible изменит файл в нормальном режиме.

```
$ ansible-playbook -D --check playbook.yml
$ ansible-playbook --diff --check playbook.yml
```

Если Ansible внесет изменения в какой-то файл (например, используя такие модули, как `copy`, `file`, `template` и `lineinfile`), то она покажет их в формате *.diff*:

```
TASK [mezzanine : create a logs directory] ************************************
--- before
+++ after
@@ -1,4 +1,4 @@
 {
     "path": "/home/vagrant/logs",
```

```
-    "state": "absent"
+    "state": "directory"
 }

 changed: [web]
```

Теги

Ansible позволяет добавлять теги к задачам, ролям и операциям. Например, следующая операция отмечена тегами `mezzanine` и `nginx`. (Бас предпочитает использовать теги на уровне ролей из-за сложности их поддержки на уровне задач.)

```
- name: deploy postgres on db
  hosts: db
  debugger: on_failed
  vars_files:
    - secrets.yml
  roles:
    - role: database
      tags: database
      database_name: "{{ mezzanine_proj_name }}"
      database_user: "{{ mezzanine_proj_name }}"

- name: deploy mezzanine on web
  hosts: web
  debugger: always
  vars_files:
    - secrets.yml

  roles:
    - role: mezzanine
      tags: mezzanine
      database_host: "{{ hostvars.db.ansible_enp0s8.ipv4.address }}"
    - role: nginx
      tags: nginx
```

Добавив в команду флаг `-t имена_тегов` или `--tags имена_тегов`, можно потребовать от Ansible выполнить только операции и задачи, отмеченные определенными тегами, а добавив флаг `--skip-tags` – пропустить операции и задачи. Взгляните на пример 8.8.

Пример 8.8. Использование тегов

```
$ ansible-playbook -tnxinx playbook.yml
$ ansible-playbook --tags=xinx,database playbook.yml
$ ansible-playbook --skip-tags=mezzanine playbook.yml
```

Ограничение обслуживаемых хостов

Чтобы ограничить список хостов, на которых будет выполняться сценарий, можно использовать флаг `--limit`. С его помощью, например, можно организовать канареечное тестирование (**https://oreil.ly/seUXz**) с применением подробного журналирования. Флаг `--limit` ограничивает круг хостов, на которых будет выполняться сценарий в соответствии с указанным выражением. В простейшем случае это может быть имя одного хоста:

```
$ ansible-playbook -vv --limit db playbook.yml
```

Ограничения и теги – очень удобные инструменты отладки, но имейте в виду, что теги сложно поддерживать в больших масштабах. Ограничения также очень полезны для тестирования и развертывания отдельных частей инфраструктуры.

Заключение

В Ansible есть множество средств, помогающих в отладке. При правильном использовании они могут помочь сократить время, необходимое для тестирования каждого изменения. Они также пригодятся вам при изучении сценариев, представленных в следующих главах.

Глава 9

Роли: масштабирование сценариев

Роли в Ansible – это основной механизм деления сценария на отдельные файлы. Они упрощают написание сложных сценариев и их повторное использование. Думайте о роли как о чем-то, применяемом к одному или нескольким хостам. Например, хостам, которые будут выступать в роли серверов баз данных, можно присвоить роль `database`. Одной из особенностей Ansible, вызывающих у меня восхищение, является вертикальное масштабирование – вверх и вниз. Масштабирование вниз помогает упростить разработку отдельных задач, а масштабирование вверх – деление сложных задач на небольшие части. Роли обеспечивают возможность структурирования и не содержат никаких данных, специфичных для конкретной машины, поэтому ими можно делиться с коллегами, реализующими управление своими серверами и комбинирующими роли в своих собственных сценариях.

Здесь я имею в виду не количество хостов, а сложность автоматизируемых задач. В этой главе вы научитесь описывать и использовать роли!

Базовая структура роли

Роль в Ansible имеет имя, например `database`. Файлы, связанные с ролью `database`, хранятся в каталоге *roles/database*, содержащем следующие файлы и подкаталоги:

```
defaults/
  main.yml
files/
  pg_hba.conf
handlers/
  main.yml
meta/
  main.yml
tasks/
```

```
    main.yml
  templates/
    postgres.conf.j2
  vars/
    main.yml
```

tasks

В каталоге *tasks* находится файл *main.yml*, служащий точкой входа для действий, выполняемых ролью.

files

Содержит файлы и сценарии для выгрузки на хосты.

templates

Содержит файлы шаблонов Jinja2 для выгрузки на хосты.

handlers

В каталоге *handlers* имеется файл *main.yml*, описывающий действия, которые должны выполняться при получении уведомлений об изменениях.

vars

Переменные, которые обычно не должны переопределяться.

defaults

Переменные по умолчанию, которые можно переопределить.

meta

Информация о роли.

Ни один конкретный файл не является обязательным; например, если роль не имеет обработчиков, то нет необходимости создавать пустой файл *handlers/main.yml* и сохранять его в репозитории.

Где Ansible будет искать мои роли?

Ansible ищет роли в подкаталоге *roles*, находящемся в папке со сценарием. Системные роли можно поместить в каталог */etc/ansible/roles*. Местоположение системных ролей можно изменить, переопределив параметр `roles_path` в секции `defaults` файла *ansible.cfg*, как показано в примере 9.1. Такая организация помогает отделить роли, определяемые в проекте, от системных ролей.

Пример 9.1. *ansible.cfg*: изменение пути к каталогу с системными ролями

```
[defaults]
roles_path = galaxy_roles:roles
```

То же самое можно сделать, изменив переменную окружения `ANSIBLE_ROLES_PATH`.

Пример: развертывание Mezzanine с использованием ролей

Возьмем за основу наш сценарий развертывания Mezzanine и изменим его, реализовав роли. Можно было бы создать единственную роль с именем `mezzanine`, но мы пойдем дальше и дополнительно выделим развертывание базы данных Postgres и веб-сервера NGINX в отдельные роли с именами `database` и `nginx` соответственно. Это упростит развертывание базы данных на хосте, отличном от хоста для приложения Mezzanine, и отделит задачи, связанные с развертыванием веб-сервера.

Использование ролей в сценариях

Прежде чем погрузиться в детали определения ролей, посмотрим, как назначать роли хостам в сценариях. В примере 9.2 представлен наш сценарий для развертывания Mezzanine на единственном хосте после добавления ролей `database`, `nginx` и `mezzanine`.

Пример 9.2. *mezzanine-single-host.yml*

```
---
- name: Deploy mezzanine on vagrant
  hosts: web

  vars_files:
    - secrets.yml

  roles:
    - role: database
      database_name: "{{ mezzanine_proj_name }}"
      database_user: "{{ mezzanine_proj_name }}"
    - role: mezzanine
      database_host: '127.0.0.1'
    - role: nginx
...
```

Чтобы задействовать роли, в сценарии должна иметься секция `roles` со списком ролей. В нашем примере список содержит три роли – `database`, `nginx` и `mezzanine`.

Обратите внимание, как можно передавать переменные при вызове ролей. В нашем примере мы передаем роли `database` переменные `database_name` и `database_user`. Если эти переменные уже определены для роли (в *vars/main.yml* или *defaults/main.yml*), их значения будут переопределены значениями, указанными здесь.

Если ролям не передаются никакие переменные, то можно определить только имена ролей, как это сделано с ролью `nginx` в примере 9.2.

После определения ролей `database`, `nginx` и `mezzanine` писать сценарий для развертывания веб-приложения и базы данных на нескольких хостах становится намного проще. В примере 9.3 показан сценарий развертывания базы данных на хосте `db` и веб-службы на хосте `web`.

Пример 9.3. *mezzanine-across-hosts.yml*

```
---

- name: Deploy postgres on db
  hosts: db

  vars_files:
    - secrets.yml

  roles:
    - role: database
      database_name: "{{ mezzanine_proj_name }}"
      database_user: "{{ mezzanine_proj_name }}"

- name: Deploy mezzanine on web
  hosts: web

  vars_files:
    - secrets.yml

  roles:
    - role: mezzanine
      database_host: "{{ hostvars.db.ansible_enp0s8.ipv4.address }}"
    - role: nginx
...
```

Обратите внимание, что этот сценарий содержит две отдельные операции (play): «Deploy postgres on db» и «Deploy mezzanine on web». Каждая операция может применяться к целой группе хостов, но у нас только одна машина в каждой группе: сервер `db` и сервер `web`.

Предварительные и заключительные задачи

Иногда до или после запуска ролей требуется выполнить некоторые задачи. Допустим, необходимо обновить кеш диспетчера `apt` перед развертыванием Mezzanine, а после развертывания отправить уведомление в канал Slack.

Ansible выполняет сценарии в следующем порядке:

- до вызова любых ролей выполняются задачи в секции `pre_tasks`;
- затем выполняются роли в секции `roles`;
- и наконец, после вызова ролей выполняются задачи в секции `post_tasks`.

В примере 9.4 представлен сценарий развертывания Mezzanine с секциями `pre_tasks`, `roles` и `post_tasks`.

Пример 9.4. Списки задач для выполнения до и после вызова ролей

```
- name: Deploy mezzanine on web
  hosts: web
  vars_files:
    - secrets.yml

  pre_tasks:
    - name: Update the apt cache
      apt:
        update_cache: yes

  roles:
    - role: mezzanine
      database_host: "{{ hostvars.db.ansible_enp0s8.ipv4.address }}"
    - role: nginx

  post_tasks:
    - name: Notify Slack that the servers have been updated
      delegate_to: localhost
      slack:
        domain: acme.slack.com
        token: "{{ slack_token }}"
        msg: "web server {{ inventory_hostname }} configured."
...
```

Но хватит об использовании ролей; поговорим лучше об их написании.

Роль database для развертывания базы данных

Задача роли `database` – установить Postgres и создать базу данных и пользователя.

Все аспекты роли `database` определяются в следующих файлах:

- *roles/database/defaults/main.yml*;
- *roles/database/files/pg_hba.conf*;
- *roles/database/handlers/main.yml*;
- *roles/database/meta/main.yml*;
- *roles/database/tasks/main.yml*;
- *roles/database/templates/postgresql.conf.j2*;
- *roles/database/vars/main.yml*.

Эта роль включает два особых конфигурационных файла Postgres.

postgresql.conf.j2

Изменяет заданный по умолчанию параметр `listen_addresses`, чтобы сервер Postgres принимал соединения на любом сетевом интерфейсе. По умолчанию Postgres принимает соединения только от `localhost`, что нам не подходит, так как наша база данных развертывается на отдельном хосте.

pg_hba.conf

Настраивает режим аутентификации в Postgres по сети с использованием имени пользователя и пароля.

Я не привожу здесь этих файлов, поскольку они достаточно большие. Вы найдете их в примерах кода в каталоге *ch07* в репозитории GitHub (**https://oreil.ly/PddOX**).

В примере 9.5 показаны задачи, вовлеченные в процесс развертывания Postgres.

Пример 9.5. *roles/database/tasks/main.yml*

```
---

- name: Install apt packages
  become: true
  apt:
    update_cache: true
    cache_valid_time: 3600
    pkg: "{{ postgres_packages }}"

- name: Copy configuration file
  become: true
  template:
    src: postgresql.conf.j2
    dest: /etc/postgresql/12/main/postgresql.conf
    owner: postgres
    group: postgres
    mode: '0644'
  notify: Restart postgres

- name: Copy client authentication configuration file
  become: true
  copy:
    src: pg_hba.conf
    dest: /etc/postgresql/12/main/pg_hba.conf
    owner: postgres
    group: postgres
```

```
    mode: '0640'
  notify: Restart postgres
- name: Create project locale
  become: true
  locale_gen:
    name: "{{ locale }}"

- name: Create a DB user
  become: true
  become_user: postgres
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"

- name: Create the database
  become: true
  become_user: postgres
  postgresql_db:
    name: "{{ database_name }}"
    owner: "{{ database_user }}"
    encoding: UTF8
    lc_ctype: "{{ locale }}"
    lc_collate: "{{ locale }}"
    template: template0
...
```

В примере 9.6 показано содержимое файла с определениями обработчиков, вызываемых при получении уведомлений об изменениях.

Пример 9.6. *roles/database/handlers/main.yml*

```
---
- name: Restart postgres
  become: true
  service:
    name: postgresql
    state: restarted
...
```

Единственная переменная по умолчанию, которую мы определим, задает порт сервера базы данных. Она используется в шаблоне *postgresql.conf.j2*.

В примере 9.7 можно видеть список пакетов для установки. Он включает саму базу данных, клиентские библиотеки C и Python, а также `acl`.

Пример 9.7. *roles/database/defaults/main.yml*

```
---
postgres_packages:
```

```
  - acl  # для become_user: postgres
  - libpq-dev
  - postgresql
  - python3-psycopg2
...
```

Пакет `acl` необходим, когда для подключения и в `become_user` используются непривилегированные учетные записи. Модуль `file` сохраняет файлы на хосте с разрешениями пользователя, установившего соединение, но они должны быть доступны для чтения пользователю `become_user`. Чтобы разрешить доступ к файлам пользователю `become_user`, Ansible будет использовать команду `setfacl` из пакета `acl`.

Обратите внимание, что в списке задач имеются ссылки на переменные, которые не определены в роли:

- `database_name`;
- `database_user`;
- `db_pass`;
- `locale`.

Переменные `database_name` и `database_user` передаются в вызов роли в примерах 9.2 и 9.3. Переменная `db_pass` будет определена в файле *secrets.yml*, который включен в секцию `vars_files`. Переменная `locale`, вероятно, будет иметь одно и то же значение для всех хостов и может использоваться разными ролями или сценариями, поэтому мы определим ее в файле *group_vars/all*.

Зачем два разных способа определения переменных в ролях?

Когда в Ansible впервые появилась поддержка ролей, переменные для них можно было определить только в *vars/main.yml*. Переменные, объявленные в этом файле, имели более высокий приоритет, чем переменные в секции vars сценария. То есть их можно было переопределить, только передав в вызов роли в виде аргумента.

Позднее в Ansible появилось понятие переменных по умолчанию для ролей, определяемых в *defaults/main.yml*. Переменные этого типа определяются в ролях и имеют низкий приоритет – их можно переопределить, если объявить эти же переменные с другими значениями в сценарии.

Если вы считаете, что значение переменной в роли может понадобиться изменить, объявите ее как переменную по умолчанию. Если переменные не должны изменяться, объявляйте их как обычные переменные.

Роль mezzanine для развертывания Mezzanine

Задача роли `mezzanine` – установка Mezzanine. Сюда входят установка NGINX в качестве обратного прокси и Supervisor в качестве монитора процессов.

Ниже перечислены файлы, реализующие роль:

- *roles/mezzanine/files/setadmin.py*;
- *roles/mezzanine/files/setsite.py*;
- *roles/mezzanine/handlers/main.yml*;
- *roles/mezzanine/tasks/django.yml*;
- *roles/mezzanine/tasks/main.yml*;
- *roles/mezzanine/templates/gunicorn.conf.pyj2*;
- *roles/mezzanine/templates/local_settings.py.filters.j2*;
- *roles/mezzanine/templates/local_settings.py.j2*;
- *roles/mezzanine/templates/supervisor.conf.j2*;
- *roles/mezzanine/vars/main.yml*.

В примере 9.8 показаны переменные для данной роли. Обратите внимание, что мы изменили их имена так, чтобы они начинались с *mezzanine*. Это хорошее правило выбора имен переменных для ролей, поскольку в Ansible нет отдельного пространства имен для ролей. Это значит, что переменные, объявленные в других ролях или где-то еще в сценарии, будут доступны повсеместно, что может приводить к нежелательным последствиям, если случайно использовать одно и то же имя переменной в двух разных ролях.

Пример 9.8. *roles/mezzanine/vars/main.yml*

```
---
# файл с переменными для mezzanine
mezzanine_user: "{{ ansible_user }}"
mezzanine_venv_home: "{{ ansible_env.HOME }}/.virtualenvs"
mezzanine_venv_path: "{{ mezzanine_venv_home }}/{{ mezzanine_proj_name }}"
mezzanine_repo_url: git@github.com:ansiblebook/mezzanine_example.git
mezzanine_settings_path: "{{ mezzanine_proj_path }}/{{ mezzanine_proj_name }}"
mezzanine_reqs_path: '~/requirements.txt'
mezzanine_python: "{{ mezzanine_venv_path }}/bin/python"
mezzanine_manage: "{{ mezzanine_python }} {{ mezzanine_proj_path }}/manage.py"
mezzanine_gunicorn_procname: gunicorn_mezzanine
...
```

Поскольку список задач довольно длинный, мы решили разбить его на несколько файлов. В примере 9.9 показана задача верхнего уровня для роли `mezzanine`. Она устанавливает пакеты `apt`, а затем использует операторы `include` для вызова задач из двух других файлов, находящихся в том же каталоге и показанных в примерах 9.10 и 9.11.

Пример 9.9. *roles/mezzanine/tasks/main.yml*

```
---

- name: Install apt packages
  become: true
  apt:
    update_cache: true
    cache_valid_time: 3600
    pkg:
      - git
      - libjpeg-dev
      - memcached
      - python3-dev
      - python3-pip
      - python3-venv
      - supervisor

- include_tasks: setup.yml
- include_tasks: django.yml
...
```

Пример 9.10. *roles/mezzanine/tasks/setup.yml*

```
---
- name: Create a logs directory
  file:
    path: "{{ ansible_env.HOME }}/logs"
    state: directory
    mode: '0755'

- name: Check out the repository on the host
  git:
    repo: "{{ mezzanine_repo_url }}"
    dest: "{{ mezzanine_proj_path }}"
    version: master
    accept_hostkey: true
    update: false
  tags:
    - repo

- name: Create python3 virtualenv
  pip:
    name:
      - pip
      - wheel
      - setuptools
    state: latest
```

```
    virtualenv: "{{ mezzanine_venv_path }}"
    virtualenv_command: /usr/bin/python3 -m venv
  tags:
    - skip_ansible_lint

- name: Copy requirements.txt to home directory
  copy:
    src: requirements.txt
    dest: "{{ mezzanine_reqs_path }}"
    mode: '0644'

- name: Install packages listed in requirements.txt
  pip:
    virtualenv: "{{ mezzanine_venv_path }}"
    requirements: "{{ mezzanine_reqs_path }}"
```

Пример 9.11. *roles/mezzanine/tasks/django.yml*

```
---
- name: Generate the settings file
  template:
    src: templates/local_settings.py.j2
    dest: "{{ mezzanine_settings_path }}/local_settings.py"
    mode: '0750'

- name: Apply migrations to database, collect static content
  django_manage:
    command: "{{ item }}"
    app_path: "{{ mezzanine_proj_path }}"
    virtualenv: "{{ mezzanine_venv_path }}"
  with_items:
    - migrate
    - collectstatic

- name: Set the site id
  script: setsite.py
  environment:
    PATH: "{{ mezzanine_venv_path }}/bin"
    PROJECT_DIR: "{{ mezzanine_proj_path }}"
    PROJECT_APP: "{{ mezzanine_proj_app }}"
    DJANGO_SETTINGS_MODULE: "{{ mezzanine_proj_app }}.settings"
    WEBSITE_DOMAIN: "{{ live_hostname }}"

- name: Set the admin password
  script: setadmin.py
  environment:
    PATH: "{{ mezzanine_venv_path }}/bin"
    PROJECT_DIR: "{{ mezzanine_proj_path }}"
```

```
      PROJECT_APP: "{{ mezzanine_proj_app }}"
      ADMIN_PASSWORD: "{{ admin_pass }}"

  - name: Set the gunicorn config file
    template:
      src: templates/gunicorn.conf.py.j2
      dest: "{{ mezzanine_proj_path }}/gunicorn.conf.py"
      mode: '0750'

  - name: Set the supervisor config file
    become: true
    template:
      src: templates/supervisor.conf.j2
      dest: /etc/supervisor/conf.d/mezzanine.conf
      mode: '0640'
    notify: Restart supervisor

  - name: Install poll twitter cron job
    cron:
      name: "poll twitter"
      minute: "*/5"
      user: "{{ mezzanine_user }}"
      job: "{{ mezzanine_manage }} poll_twitter"
  ...
```

Есть существенная разница между задачами, объявленными в роли, и задачами, объявленными в сценарии как обычно. Она касается использования модулей `copy`, `script` и `template`. Когда модуль `copy` или `script` вызывается в задаче для роли, Ansible будет искать файлы в каталогах в том порядке, в каком они перечислены ниже, и использовать первый найденный. Пути к этим каталогам откладываются относительно каталога со сценарием верхнего уровня.

- *./roles/role_name/files/*;
- *./roles/role_name/*;
- *./roles/role_name/tasks/files/*;
- *./roles/role_name/tasks/*;
- *./files/*;
- *./*.

Аналогично, когда модуль `template` вызывается в задаче для роли, Ansible сначала проверит каталог *<имя_роли>/templates*, а затем *playbooks/templates* (наряду с менее очевидными каталогами). Таким способом роли определяют файлы по умолчанию в своих каталогах *files/* и *templates/*, но вы не можете просто заменить их файлами в подкаталогах *files/* и *templates/* проекта.

Это значит, что задача, которая раньше была определена в сценарии так:

```
- name: Copy requirements.txt to home directory
  copy:
    src: files/requirements.txt
    dest: "{{ mezzanine_reqs_path }}"
    mode: '0644'
```

теперь, когда она вызывается в роли, должна выглядеть так (обратите внимание на изменившийся параметр `src`):

```
- name: Copy requirements.txt to home directory
  copy:
    src: "{{ files_src_path | default() }}requirements.txt"
    dest: "{{ mezzanine_reqs_path }}"
    mode: '0644'
```

`files_src_path` – это переменная, хранящая путь, который можно переопределить. Эта переменная может также хранить пустое значение для реализации поведения по умолчанию. Такой вариант использования переменных с путями к файлам и шаблонам в ролях предложил (**https://oreil.ly/WgI9l**) Рамон де ла Фуэнте (Ramon de la Fuente).

В примере 9.12 показан файл обработчиков. Эти обработчики вызываются, когда поступают уведомления об изменениях.

Пример 9.12. *roles/mezzanine/handlers/main.yml*

```
---
- name: Restart supervisor
  become: true
  supervisorctl:
    name: gunicorn_mezzanine
    state: restarted
...
```

Мы не будем приводить здесь файлы шаблонов, поскольку они остались теми же, что и в предыдущей главе, хотя имена некоторых переменных изменились. За дополнительной информацией обращайтесь к примерам кода, прилагаемым к книге (**https://oreil.ly/PddOX**).

Создание файлов и каталогов ролей с помощью ansible-galaxy

В состав Ansible входит еще один инструмент командной строки, о котором мы пока не говорили. Это `ansible-galaxy`. Его основное назначение – загрузка ролей, которыми поделились члены сообщества Ansible (**https://galaxy.ansible.com/**), подробнее об этом чуть позже. Но с его помощью

также можно сгенерировать начальный набор файлов и каталогов для роли:

```
$ ansible-galaxy init --init-path playbooks/roles web
```

Параметр `--init-path` сообщает местоположение каталога *roles*. Если его опустить, то `ansible-galaxy` создаст файлы в текущем каталоге. Эта команда создаст следующие файлы и каталоги:

```
playbooks
|__ roles
     |__ web
          |— README.md
          |— defaults
          |  |__ main.yml
          |— files
          |— handlers
          |  |__ main.yml
          |— meta
          |  |__ main.yml
          |— tasks
          |  |__ main.yml
          |— templates
          |— tests
          |  |__ inventory
          |  |__ test.yml
          |__ vars
               |__ main.yml
```

Зависимые роли

Представьте, что у нас есть две роли – `web` и `database`, – и обе требуют установки сервера NTP[1]. Мы могли бы описать установку NTP-сервера в обеих ролях, но это привело бы к дублированию кода. Мы могли бы определить отдельную роль `ntp`, но тогда нам пришлось бы помнить, что, запуская роли `web` и `database`, мы также должны запустить роль `ntp`. Такой подход избавил бы от дублирования кода, но он чреват ошибками, поскольку можно забыть вызвать роль `ntp`. В действительности нам нужно, чтобы роль `ntp` всегда присваивалась хостам, которым присваиваются роли `web` и `database`.

Ansible поддерживает возможность определения *зависимостей* между ролями для подобных случаев. Определяя роль, можно указать, что она зависит от одной или нескольких других ролей, а Ansible позаботится о том, чтобы зависимые роли выполнялись первыми.

[1] NTP (Network Time Protocol) – протокол сетевого времени, используется для синхронизации времени.

Продолжая наш пример, допустим, что мы создали роль `ntp`, настраивающую хост для синхронизации часов с сервером NTP. Ansible позволяет передавать параметры зависимым ролям, поэтому представим, что мы передали адрес сервера NTP этой роли как параметр.

Укажем, что роль `web` зависит от роли `ntp`, создав файл *roles/web/meta/main.yml* и добавив в него роль `ntp` с параметром, как показано в примере 9.13.

Пример 9.13. *roles/web/meta/main.yml*

```
dependencies:
    - { role: ntp, ntp_server=ntp.ubuntu.com }
```

Таким способом можно определить несколько зависимых ролей. Например, если бы у нас была роль `django` для установки веб-сервера Django и мы хотели бы определить роли `nginx` и `memcached` как зависимости, тогда файл метаданных роли выглядел бы, как показано в примере 9.14.

Пример 9.14. *roles/django/meta/main.yml*

```
dependencies:
    - { role: web }
    - { role: memcached }
```

За более подробной информацией о зависимостях между ролями в Ansible обращайтесь к официальной документации (**https://oreil.ly/3nJ4K**).

Ansible Galaxy

Если вам понадобится установить на ваши хосты программное обеспечение с открытым исходным кодом, то вполне вероятно, что кто-то уже написал роль Ansible для этого. Хотя разработка сценариев для развертывания программного обеспечения не особенно сложна, некоторые системы действительно требуют сложных процедур развертывания.

Если вы захотите использовать роль, написанную кем-то другим, или просто посмотреть, как кто-то другой решил похожую задачу, Ansible Galaxy поможет вам в этом. *Ansible Galaxy* – это хранилище ролей Ansible с открытым исходным кодом, пополняемое членами сообщества Ansible. Сами роли хранятся на GitHub. **https://galaxy.ansible.com** – центральный веб-сайт для контента Ansible, а `ansible-galaxy` – инструмент интерфейса командной строки.

Веб-интерфейс

Исследовать доступные роли можно на сайте Ansible Galaxy (**http://galaxy.ansible.com**). Galaxy поддерживает обычный текстовый поиск, а также фильтрацию по категории или разработчику.

Интерфейс командной строки

Инструмент командной строки `ansible-galaxy` позволяет загружать роли с сайта Ansible Galaxy или создавать стандартную структуру каталогов для `ansible-role`.

Установка роли

Допустим, вы решили установить роль `ntp`, написанную пользователем GitHub с именем *oefenweb* (Миша тер Смиттен [Mischa ter Smitten], один из самых активных авторов Ansible Galaxy). Эта роль настраивает хост для синхронизации часов с сервером NTP.

Установите роль командой `ansible-galaxy install`.

```
$ ansible-galaxy install oefenweb.ntp
```

Программа `ansible-galaxy` по умолчанию устанавливает роли в первый каталог из перечисленных в *roles_path* (см. врезку «Где Ansible будет искать мои роли?» в начале главы), но вы можете изменить каталог установки, добавив параметр `-p` (при этом все необходимые подкаталоги будут созданы автоматически, если потребуется).

Результат должен выглядеть так:

```
Starting galaxy role install process
- downloading role 'ntp', owned by oefenweb
- downloading role from https://github.com/Oefenweb/ansible-ntp/archive/v1.1.33.
tar.gz
- extracting oefenweb.ntp to ./galaxy_roles/oefenweb.ntp
- oefenweb.ntp (v1.1.33) was installed successfully
```

Инструмент `ansible-galaxy` установит файлы роли в *galaxy_roles/oefenweb.ntp*.

Ansible поместит некоторые метаданные об установленной роли в файл *./galaxy_roles/oefenweb.ntp/meta/.galaxy_install_info*. На машине Баса этот файл содержит:

```
install_date: Tue Jul 20 12:13:44 2021
version: v1.1.33
```

Роль `oefenweb.ntp` имеет конкретный номер версии, поэтому он указан явно. Некоторые роли не имеют номера версии, поэтому для них вместо номера версии указывается имя ветки по умолчанию в GitHub, например `main`.

Вывод списка установленных ролей

Получить список установленных ролей можно командой:

```
$ ansible-galaxy list
```

Она выведет роли в порядке сортировки ключей `galaxy_info` в *meta/main.yml*, как показано ниже:

```
# /Users/bas/ansiblebook/ch07/playbooks/galaxy_roles
- oefenweb.ntp, v1.1.33
# /Users/bas/ansiblebook/ch07/playbooks/roles
- database, (unknown version)
- web, (unknown version)
```

Удаление роли

Удалить роль можно командой `remove`:

```
$ ansible-galaxy remove oefenweb.ntp
```

Требования к оформлению ролей на практике

Общепринятой практикой считается перечисление зависимостей в файле с именем *requirements.yml* в каталоге *roles*, расположенном в папке *<каталог_проекта>/roles/requirements.yml*. Если этот файл присутствует, то `ansible-galaxy` автоматически установит перечисленные в нем роли. Такой подход позволяет ссылаться на роли в Galaxy или в других репозиториях и извлекать их вместе с вашим собственным проектом. Добавление поддержки Ansible Galaxy устраняет необходимость создания подмодулей Git для достижения этого результата.

В следующем фрагменте первый параметр `src` определяет зависимость от роли `oefenweb.ntp` (если источник роли определен так, то по умолчанию она будет загружена из Galaxy). Второй параметр `src` инициирует загрузку роли `docker` непосредственно из GitHub; эту роль написал Джефф Гирлинг (Jeff Geerling) – известный в сообществе Ansible своей книгой «Ansible for DevOps, 2nd ed.» [LeanPub] и многими ролями в Galaxy. Третий загружает роль из локального репозитория Git. Параметр `name` в файле *requirements.yml* можно использовать для переименования ролей после загрузки.

```
---

- src: oefenweb.ntp

- src: https://github.com/geerlingguy/ansible-role-docker.git
  scm: git
  version: '4.0.0'
  name: geerlingguy.docker

- src: https://tools.example.intra/bitbucket/scm/ansible/install-nginx.git
  scm: git
  version: master
```

```
    name: web
...
```

Как поделиться своей ролью

Чтобы узнать, как поделиться своей ролью с другими членами сообщества, обращайтесь к разделу «Contributing Content» (**https://oreil.ly/lfLle**) на сайте Ansible Galaxy. Поскольку роли располагаются в репозитории GitHub, вам потребуется создать свою учетную запись.

Заключение

Теперь вы знаете, как использовать роли, создавать собственные роли и загружать роли, написанные другими. Роли – мощный инструмент организации сценариев. Мы пользуемся ими постоянно и настоятельно рекомендуем вам поступать так же. Если вы обнаружите, что конкретный ресурс, с которым вы работаете, не имеет роли в Galaxy, то напишите эту роль и поделитесь ею с другими!

Глава 10

Сложные сценарии

В предыдущей главе мы рассмотрели полноценный сценарий Ansible для развертывания Mezzanine CMS. В этом примере были использованы самые разные возможности Ansible, но далеко не все. Данная глава рассказывает о дополнительных возможностях, превращаясь в кладезь не менее полезной информации.

Решение проблем с неидемпотентными командами

В главе 7 мы предпочли отказаться от команды `createdb manage.py`, представленной в примере 10.1, потому что она не является идемпотентной.

Пример 10.1. Вызов команды *django manage.py* из *createdb*

```
- name: Initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
```

Мы решили эту проблему запуском нескольких идемпотентных команд `django manage.py`, которые в комплексе эквивалентны `createdb`. Но как быть, если нет модуля с эквивалентными командами? Решить эту проблему помогут выражения `changed_when` и `failed_when`, используемые в Ansible для обнаружения изменения состояния или ошибок.

Сначала нужно разобраться, что выводит команда в первый раз, а что во второй.

Как мы уже делали это в главе 5, добавим выражение `register` для сохранения в переменной вывода задачи, завершившейся с ошибкой, и выражение `failed_when: false`, чтобы исключить остановку сценария в случае ошибки. Следом добавим задачу `debug`, чтобы вывести на экран содержимое переменной. И наконец, используем выражение `fail` для остановки сценария, как показано в примере 10.2.

Пример 10.2. Вывод результата выполнения задачи

```
- name: Initialize the database
  django_manage:
```

```
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
  failed_when: false
  register: result

- debug: var=result

- fail:
```

В примере 10.3 показан вывод сценария после попытки запустить его второй раз.

Пример 10.3. Вывод сценария в случае, если база данных уже создана

```
TASK [debug] *******************************************************************
ok: [web] ==> {
    "result": {
        "changed": false,
        "cmd": "./manage.py createdb --noinput --nodata",
        "failed": false,
        "failed_when_result": false,
        "msg": "\n:stderr: CommandError: Database already created, you probably want
the migrate command\n",
        "path": "/home/vagrant/.virtualenvs/mezzanine_example/bin:/usr/local/sbin:/
usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/
games:/snap/bin",
        "syspath": [
            "/tmp/ansible_django_manage_payload_hb62e1ie/ansible_django_manage_pay
load.zip",
            "/usr/lib/python38.zip",
            "/usr/lib/python3.8",
            "/usr/lib/python3.8/lib-dynload",
            "/usr/local/lib/python3.8/dist-packages",
            "/usr/lib/python3/dist-packages"
        ]
    }
}
```

Это происходит при каждом повторном запуске задачи. Чтобы увидеть, что происходит при запуске в первый раз, удалите базу данных и позвольте сценарию воссоздать ее. Самый простой способ сделать это – запустить специальную задачу Ansible, которая удаляет базу данных:

```
$ ansible web -b --become-user postgres -m postgresql_db \
    -a  "name=mezzanine_example state=absent"
```

Если теперь запустить сценарий, он выведет строки, показанные в примере 10.4.

Пример 10.4. Вывод сценария при первом запуске

```
TASK [debug] *******************************************************************
ok: [web] ==> {
    "result": {
        "app_path": "/home/vagrant/mezzanine/mezzanine_example",
        "changed": false,
        "cmd": "./manage.py createdb --noinput --nodata",
        "failed": false,
        "failed_when_result": false,
        "out": "Operations to perform:\n  Apply all migrations: admin, auth, blog,
        conf, contenttypes, core, django_comments, forms, galleries, generic, pages,
        redirects, sessions, sites, twitter\nRunning migrations:\n  Applying
        contenttypes.0001_initial... OK\n  Applying auth.0001_initial... OK\n
        Applying admin.0001_initial... OK\n  Applying
        admin.0002_logentry_remove_auto_add... OK\n  Applying
        contenttypes.0002_remove_content_type_name... OK\n  Applying
        auth.0002_alter_permission_name_max_length... OK\n  Applying
        auth.0003_alter_user_email_max_length... OK\n  Applying
        auth.0004_alter_user_username_opts... OK\n  Applying
        auth.0005_alter_user_last_login_null... OK\n  Applying
        auth.0006_require_contenttypes_0002... OK\n  Applying
        auth.0007_alter_validators_add_error_messages... OK\n  Applying
        auth.0008_alter_user_username_max_length... OK\n  Applying
        sites.0001_initial... OK\n  Applying blog.0001_initial... OK\n  Applying
        blog.0002_auto_20150527_1555... OK\n  Applying blog.0003_auto_20170411_0504...
        OK\n  Applying conf.0001_initial... OK\n  Applying core.0001_initial... OK\n
        Applying core.0002_auto_20150414_2140... OK\n  Applying
        django_comments.0001_initial... OK\n  Applying
        django_comments.0002_update_user_email_field_length... OK\n  Applying
        django_comments.0003_add_submit_date_index... OK\n
        Applying pages.0001_initial... OK\n  Applying forms.0001_initial... OK\n
        Applying forms.0002_auto_20141227_0224... OK\n  Applying forms.0003_emailfield...
        OK\n  Applying forms.0004_auto_20150517_0510... OK\n  Applying
        forms.0005_auto_20151026_1600... OK\n  Applying forms.0006_auto_20170425_2225...
        OK\n  Applying galleries.0001_initial... OK\n  Applying
        galleries.0002_auto_20141227_0224... OK\n  Applying generic.0001_initial... OK\n
        Applying generic.0002_auto_20141227_0224... OK\n  Applying
        generic.0003_auto_20170411_0504... OK\n  Applying pages.0002_auto_20141227_0224...
        OK\n  Applying pages.0003_auto_20150527_1555... OK\n  Applying
        pages.0004_auto_20170411_0504... OK\n  Applying redirects.0001_initial... OK\n
        Applying sessions.0001_initial... OK\n  Applying sites.0002_alter_domain_unique...
        OK\n  Applying twitter.0001_initial... OK\n\nCreating default site record: web
        ...\n\nInstalled 2 object(s) from 1 fixture(s)\n",
        "pythonpath": null,
        "settings": null,
        "virtualenv": "/home/vagrant/.virtualenvs/mezzanine_example"
```

```
    }
}
```

Обратите внимание, что ключ `changed` получает значение `false`, хотя состояние базы данных изменилось. Это объясняется тем, что модуль `django_manage` всегда возвращает `"changed":false`, когда выполняет неизвестные ему команды.

Можно добавить выражение `changed_when`, отыскивающее подстроку `"Creating tables"` в возвращаемом значении `out`, как показано в примере 10.5.

Пример 10.5. Первая попытка добавить *changed_when*

```
- name: Initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
  register: result
  changed_when: '"Creating tables" in result.out'
```

Проблема этого подхода заключается в отсутствии переменной `out`, когда сценарий выполняется повторно. Это можно увидеть, вернувшись к примеру 10.3. Вместо нее объявлена переменная `msg`. Это означает, что, запустив сценарий во второй раз, он выведет следующее (не особенно информативное) сообщение об ошибке:

```
TASK: [Initialize the database] ***********************************************
fatal: [default] => error while evaluating conditional: "Creating tables" in
result.out
```

Значит, мы должны убедиться в присутствии переменной `result.out`, прежде чем обращаться к ней. Единственный способ сделать это:

```
changed_when: result.out is defined and "Creating tables" in result.out
```

Или, если `result.out` отсутствует, можно присвоить ей значение по умолчанию с помощью Jinja2-фильтра `default`:

```
changed_when: '"Creating tables" in result.out|default("")'
```

Окончательный вариант идемпотентной задачи показан в примере 10.6.

Пример 10.6. Идемпотентная задача *manage.py createdb*

```
- name: Initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
```

```
register: result
changed_when: '"Creating tables" in result.out|default("")'
```

Фильтры

Фильтры являются особенностью механизма шаблонов Jinja2. Поскольку Ansible использует Jinja2 для определения значений переменных и шаблонов, вы можете использовать фильтры внутри скобок `{{ }}` в ваших сценариях, а также в файлах шаблонов. Использование фильтров схоже с использованием конвейеров в Unix, где переменная передается через фильтр. Jinja2 поддерживает набор встроенных фильтров (**https://oreil.ly/7svtE**). Кроме того, Ansible добавляет свои фильтры (**https://oreil.ly/DlvWZ**), расширяя возможности фильтров Jinja2.

Далее мы рассмотрим несколько фильтров для примера, а чтобы получить полный их список, обращайтесь к официальной документации по Jinja2 и Ansible.

Фильтр default

Фильтр `default` – один из самых полезных. Его применение демонстрирует следующий пример:

```
host: "{{ database_host | default('localhost') }}"
```

Если переменная `database_host` определена, то на место фигурных скобок будет подставлено ее значение. Если она не определена, будет подставлена строка `localhost`. Некоторые фильтры принимают аргументы, некоторые – нет.

Фильтры для зарегистрированных переменных

Допустим, нам нужно запустить задачу и вывести ее результат, даже если она потерпит неудачу. Однако если задача выполнилась с ошибкой, необходимо, чтобы сценарий завершился сразу после вывода результата. В примере 10.7 показано, как этого добиться, передав фильтр `failed` в аргументе выражению `failed_when`.

Пример 10.7. Использование фильтра *failed*

```
- name: Run myprog
  command: /opt/myprog
  register: result
  ignore_errors: true

- debug: var=result

- debug:
    msg: "Stop running the playbook if myprog failed"
```

```
    failed_when: result|failed

# далее следуют другие задачи
```

В табл. 10.1 перечислены фильтры, которые можно использовать для проверки статуса зарегистрированных переменных.

Таблица 10.1. Фильтры для возвращаемых значений задач

Имя	Описание
`failed`	`True`, если задача завершилась неудачей
`changed`	`True`, если задача выполнила изменения
`success`	`True`, если задача завершилась успешно
`skipped`	`True`, если задача была пропущена

Фильтры для путей к файлам

В табл. 10.2 перечислены фильтры для работы с переменными, содержащими пути к файлам в файловой системе управляющей машины.

Таблица 10.2. Фильтры для работы с путями к файлам

Имя	Описание
`basename`	Базовое имя файла
`dirname`	Путь к файлу или каталогу
`expanduser`	Путь к файлу со знаком ~, обозначающим путь к домашнему каталогу
`realpath`	Канонический путь к файлу, разрешает символические ссылки

Рассмотрим следующий фрагмент сценария:

```
vars:
  homepage: /usr/share/nginx/html/index.html

tasks:
  - name: Copy home page
    copy:
      src: files/index.html
      dest: "{{ homepage }}"
```

Обратите внимание, что в нем дважды упоминается *index.html*: первый раз – в определении переменной `homepage`, второй – в определении пути к файлу на управляющей машине.

Фильтр `basename` дает возможность получить имя файла *index.html*, выделив его из полного пути, что позволит записать сценарий без повторения имени файла[1]:

[1] Спасибо Джону Джарвису (John Jarvis) за эту подсказку.

```
vars:
  homepage: /usr/share/nginx/html/index.html

tasks:

  - name: Copy home page
    copy:
      src: "files/{{ homepage | basename }}"
      dest: "{{ homepage }}"
```

Создание собственного фильтра

В нашем примере развертывания Mezzanine мы создали файл *local_settings.py* из шаблона, содержащего строку, показанную в примере 10.8.

Пример 10.8. Строка из файла *local_settings.py*, созданного из шаблона

```
ALLOWED_HOSTS = ["www.example.com", "example.com"]
```

У нас имеется переменная `domains` со списком имен хостов. Первоначально мы использовали цикл `for`, чтобы получить эту строку, но с фильтром шаблон будет выглядеть еще изящнее.

Существует встроенный фильтр Jinja2 с именем `join`, который объединяет строки из заданного списка, перечисляя их через разделитель, например через запятую. К сожалению, это не совсем тот результат, что нам нужен. Если применить его в шаблоне:

```
ALLOWED_HOSTS = [{{ domains|join(", ") }}]
```

то мы получим строки без кавычек, как показано в примере 10.9.

Пример 10.9. Имена хостов лишились кавычек

```
ALLOWED_HOSTS = [www.example.com, example.com]
```

Если бы у нас имелся фильтр (см. пример 10.10), заключающий строки в кавычки, тогда шаблон сгенерировал бы строку, как показано в примере 10.8.

Пример 10.10. Использование фильтра для заключения строк в кавычки

```
ALLOWED_HOSTS = [{{ domains|surround_by_quotes|join(", ") }}]
```

К сожалению, готового фильтра `surround_by_quotes` не существует. Но мы можем написать его сами. На самом деле Хэнфи Сан (Hanfei Sun) уже раскрыл этот вопрос на Stack Overflow (**https://oreil.ly/Y5kqL**).

Ansible ищет нестандартные фильтры в каталоге *filter_plugins*, находящемся в одном каталоге со сценариями.

В примере 10.11 показано, как выглядит реализация фильтра.

Пример 10.11. *filter_plugins/surround_by_quotes.py*

```
''' Взято по адресу: https://stackoverflow.com/a/68610557/571517 '''
class FilterModule():
    ''' Класс FilterModule должен иметь метод с именем filters '''
    @staticmethod
    def surround_by_quotes(a_list):
        ''' заключает в кавычки каждый элемент списка '''
        return ['"%s"' % an_element for an_element in a_list]
    def filters(self):
        ''' возвращает словарь с именами фильтров
            и соответствующими им реализациями '''
        return {'surround_by_quotes': self.surround_by_quotes}
```

Функция `surround_by_quotes` реализует фильтр Jinja2. Класс `FilterModule` определяет метод `filters`, возвращающий словарь с именами функций-фильтров и ссылками на соответствующие им реализации. Класс `FilterModule` обеспечивает доступность фильтров для Ansible.

Кроме того, в каталог *~/.ansible/plugins/filter* или */usr/share/ansible/plugins/filter* можно установить свои плагины фильтров. Также в переменной окружения `ANSIBLE_FILTER_PLUGINS` можно указать другой каталог, где хранятся ваши плагины.

Дополнительные примеры и документацию с описанием плагинов фильтров можно найти в репозитории GitHub (**https://oreil.ly/hGzbQ**).

Подстановки

В идеальном мире вся информация о конфигурации хранилась бы в переменных Ansible везде, где Ansible позволяет определять переменные (например, секция `vars` в сценарии; файлы, перечисленные в секции `vars_files`; файлы в каталогах *host_vars* или *group_vars*, которые мы обсуждали в главе 3).

Увы, мир несовершенен, и порой часть конфигурации должна храниться в других местах, например в текстовом файле или в файле *.csv*, и вам не хотелось бы копировать эти данные в переменные Ansible, поскольку в этом случае придется поддерживать две копии одних и тех же данных, а вы верите в принцип DRY[1]. Возможно, данные и вовсе хранятся не в файле, а в хранилище типа ключ/значение, таком как Redis. Ansible поддерживает функции *подстановки*, позволяющие читать настройки из разных источников, а затем использовать их в сценариях и шаблонах.

[1] DRY (от англ. *Don't Repeat Yourself*) – «не повторяйтесь». Этот термин был введен в замечательной книге «The Pragmatic Programmer: From Journeyman to Master» Эндрю Ханта и Дэвида Томаса (*Хант Э., Томас Д.* Программист-прагматик. Путь от подмастерья к мастеру. Лори, 2009. ISBN 5-85582-213-3, 0-201-61622-X. – *Прим. перев.*).

Получить полный список функций подстановки можно командой:

```
$ ansible-doc -t lookup -l
```

Перечень встроенных функций `ansible.builtin` приводится в табл. 10.3.

Таблица 10.3. Функции подстановки в *ansible.builtin*

Имя	Описание
config	Отыскивает текущие значения настроек Ansible
csvfile	Запись в файле *.csv*
dict	Возвращает пары ключ/значение из словарей
dnstxt	Запись в DNS типа TXT
env	Переменная окружения
file	Содержимое файла
fileglob	Список файлов с именами, соответствующими образцу
first_found	Возвращает первый найденный файл из списка
indexed_items	Перезаписывает список, чтобы вернуть «индексированные элементы»
ini	Читает данные из INI-файла
items	Список элементов
lines	Читает строки из вывода команды
list	Просто возвращает данные, полученные на входе
nested	Составляет список с вложенными элементами из других списков
password	Случайно сгенерированный пароль
pipe	Вывод команды, выполненной локально
random_choice	Возвращает случайный элемент из списка
redis	Значение ключа в Redis
sequence	Генерирует список из последовательных чисел
subelements	Обход ключей в списке словарей
template	Шаблон Jinja2 после обработки
together	Объединяет списки в синхронизированный список
unvault	Читает содержимое файлов, зашифрованных с помощью Vault
url	Возвращает содержимое URL
varnames	Выполняет поиск переменных с заданными именами
vars	Выполняет поиск значений переменных для подстановки в шаблон

Чтобы узнать, как пользоваться той или иной функцией, выполните команду:

```
$ ansible-doc -t lookup <имя плагина>
```

Все плагины подстановки выполняются на управляющей машине, а не на удаленном хосте.

Выполнить подстановку можно с помощью функции `lookup`, принимающей два аргумента. Первый аргумент – это строка с именем подстановки, второй – строка, содержащая один или несколько аргументов, которые передаются в подстановку. Например, подстановку `file` можно вызвать так:

```
lookup('file', '/path/to/file.txt')
```

В сценариях подстановки должны заключаться в фигурные скобки {{ }}, их также можно использовать в шаблонах.

В следующих разделах будет представлен только краткий обзор некоторых из доступных подстановок. Более подробную информацию можно найти в документации Ansible (**https://oreil.ly/tnCmt**).

file

Допустим, на управляющей машине имеется текстовый файл, содержащий открытый SSH-ключ, который необходимо скопировать на удаленный сервер. В примере 10.12 показано, как использовать подстановку `file` для чтения содержимого файла и его передачи модулю в параметре[1].

Пример 10.12. Использование подстановки *file*

```
- name: Add my public key for SSH
  authorized_key:
        user: vagrant
        key: "{{ lookup('file', item) }}"
  with_first_found:
        - ~/.ssh/id_ed25519.pub
        - ~/.ssh/id_rsa.pub
        - ~/.ssh/id_ecdsa.pub
```

Подстановки также можно использовать в шаблонах. Если потребуется использовать тот же прием для создания файла *authorized_keys* с содержимым файла открытого ключа, то можно создать шаблон Jinja2, выполняющий подстановку (пример 10.13), и затем вызвать модуль `template`, как показано в примере 10.14.

Пример 10.13. *authorized_keys.j2*

```
from="10.0.2.2" {{ lookup('file', '~/.ssh/id_ed25519.pub') }}
```

Пример 10.14. Задача, генерирующая файл *authorized_keys*

```
- name: Copy authorized_keys template
  template:
```

[1] Выполните команду `ansible-doc authorized_key`, чтобы узнать, как этот модуль может помочь защитить конфигурацию SSH.

```
    src: authorized_keys.j2
    dest: /home/vagrant/.ssh/authorized_keys
    owner: vagrant
    group: vagrant
    mode: '0600'
```

pipe

Подстановка `pipe` запускает внешнюю программу на управляющей машине и принимает ее вывод. Например, с помощью подстановки `pipe` можно установить открытый ключ по умолчанию для пользователя Vagrant. Все установленные экземпляры `vagrant` получают один и тот же файл *insecure_private_key*, благодаря чему любой разработчик сможет использовать машины Vagrant. Открытый ключ можно получить с помощью команды, которую здесь мы определили как переменную (чтобы избежать предупреждения о длине строки):

```
- name: Add default public key for vagrant user
  authorized_key:
    user: vagrant
    key: "{{ lookup('pipe', pubkey_cmd ) }}"
  vars:
    pubkey_cmd: 'ssh-keygen -y -f ~/.vagrant.d/insecure_private_key'
```

env

Подстановка `env` извлекает значение переменной окружения на управляющей машине. Например:

```
- name: Get the current shell
  debug: msg="{{ lookup('env', 'SHELL') }}"
```

Поскольку Бас использует командную оболочку `bash`, на его машине результат выглядит так:

```
TASK: [Get the current shell] *************************************************
ok: [web] ==> {
    "msg": "/bin/bash"
}
```

password

Подстановка `password` возвращает случайно сгенерированный пароль и записывает его в файл, указанный в аргументе. Например, если потребуется создать пользователя базы данных с именем `deploy` и случайным паролем, а затем записать пароль в файл *pw.txt* на управляющей машине, то это можно сделать так:

```
- name: Create deploy user, save random password in pw.txt
  become: true
  user:
    name: deploy
    password: "{{ lookup('password', 'pw.txt encrypt=sha512_crypt') }}"
```

template

Подстановка `template` позволяет получить результат применения шаблона Jinja2. Например, для шаблона, представленного в примере 10.15:

Пример 10.15. *message.j2*

```
This host runs {{ ansible_facts.distribution }}
```

Следующая задача:

```
- name: Output message from template
  debug:
    msg: "{{ lookup('template', 'message.j2') }}"
```

вернет такой результат:

```
TASK: [Output message from template] ***************************************
ok: [web] ==> {
    "msg": "This host runs Ubuntu\n"
}
```

csvfile

Подстановка `csvfile` читает запись из файла *.csv*. Допустим, у Лорина имеется файл *.csv*, который выглядит, как показано в примере 10.16.

Пример 10.16. *users.csv*

```
username,email
lorin,lorin@ansiblebook.com
john,john@example.com
sue,sue@example.org
```

Ему нужно получить электронный адрес Сью (Sue), используя плагин подстановки `csvfile`. Для этого можно использовать плагин, как показано ниже:

```
lookup('csvfile', 'sue file=users.csv delimiter=, col=1')
```

Подстановка `csvfile` – хороший пример подстановки, принимающей несколько аргументов. В данном случае плагину передаются четыре аргумента:

- `sue`;
- `file=users.csv`;
- `delimiter=,`;
- `col=1`.

Имя первого аргумента можно не указывать, но имена всех остальных должны указываться обязательно. Первый аргумент подстановки `csvfile` – это элемент, который должен присутствовать в столбце 0 (первый столбец, индексация начинается с 0) таблицы.

Остальные аргументы определяют имя файла *.csv*, символ-разделитель и какие столбцы необходимо вернуть. В данном примере мы должны:

- выполнить поиск в файле *users.csv*, в котором поля отделяются друг от друга запятыми;
- отыскать запись, в первом столбце которой хранится имя `sue`;
- вернуть значение второго столбца (столбец 1, индексация начинается с 0). В ответ плагин возвращает значение *sue@example.org*.

Если представить, что искомое имя пользователя хранится в переменной `username`, то можно сконструировать строку аргументов с помощью знака `+`, чтобы объединить строку из `username` с оставшейся частью строки с аргументами:

```
lookup('csvfile', username + ' file=users.csv delimiter=, col=1')
```

dig

Читатели этой книги наверняка знают, что делает система доменных имен (DNS), но на всякий случай напомним, что DNS – это служба, которая преобразует имена хостов, такие как *ansiblebook.com*, в соответствующие им IP-адреса, такие как *64.98.145.30*.

Чтобы использовать модуль `dig`, нужно установить пакет *dnspython* для Python на управляющую машину Ansible.

Служба DNS присваивает имени хоста от одной до нескольких записей. Наиболее распространенными типами записей DNS являются записи *A* и *CNAME*, связывающие имена хостов с IP-адресами (записи *A*) или сообщающие, что имя хоста является псевдонимом для другого имени хоста (запись *CNAME*).

Протокол DNS поддерживает также записи *TXT*. Каждая такая запись представлена произвольной строкой, которую можно связать с именем хоста, чтобы любой мог получить ее с помощью клиента DNS.

Например, Лорину принадлежит домен *ansiblebook.com*, поэтому он может создавать записи *TXT* для любых имен хостов в этом домене[1].

[1] Поставщики услуг DNS обычно предлагают веб-интерфейсы, с помощью которых можно выполнять задачи, связанные с DNS, например создавать записи *TXT*.

В частности, он создал запись *TXT* для имени хоста *ansiblebook.com*, содержащую номер ISBN этой книги. Получить запись *TXT* можно с помощью инструмента командной строки `dig`, как показано в примере 10.17.

Пример 10.17. Получение записи TXT с помощью инструмента *dig*

```
$ dig +short ansiblebook.com TXT
"isbn=978-1098109158"
```

Подстановка `dig` запрашивает у DNS-сервера записи, связанные с хостом. Вот как можно определить задачу в сценарии для запроса записей *TXT*:

```
- name: Look up TXT record
  debug:
    msg: "{{ lookup('dnstxt', 'ansiblebook.com', 'qtype=TXT') }}"
```

Она выведет следующее:

```
TASK: [Look up TXT record] ****************************************************
ok: [myserver] ==> {
    "msg": "isbn=978-1098109158"
}
```

Дополнительную информацию о плагине `dig` можно получить, выполнив команду:

```
$ ansible-doc -t lookup dig
```

redis

Redis – популярное хранилище типа ключ/значение, часто используемое как кеш, а также для хранения данных в службах очередей заданий, таких как Sidekiq. С помощью подстановки `redis` можно извлекать значения ключей. Ключ должен быть представлен строкой, поскольку модуль выполняет эквивалент команды `GET`. Эта подстановка организована иначе, чем большинство других, потому что поддерживает поиск списков переменной длины.

Модуль `redis` требует установки пакета `redis` для Python на управляющей машине.

Допустим, у нас имеется сервер Redis, запущенный на управляющей машине. Мы можем определить ключ `weather` со значением `sunny` и ключ `temp` со значением `25`, как показано ниже:

```
$ redis-cli SET weather sunny
$ redis-cli SET temp 25
```

Если определить в сценарии задачу извлечения этих ключей из хранилища Redis:

```
- name: Look up values in Redis
  debug:
    msg: "{{ lookup('redis', 'weather','temp') }}"
```

то она вернет следующее:

```
TASK: [Look up values in Redis] ***********************************************
ok: [localhost] ==> {
    "msg": "sunny,25"
}
```

Если имя хоста и порт не указаны, то по умолчанию модуль использует URL *redis://localhost:6379*. Чтобы задействовать другой сервер, в задаче можно использовать переменные окружения:

```
- name: Look up values in Redis
  environment:
    ANSIBLE_REDIS_HOST: redis1.example.com
    ANSIBLE_REDIS_PORT: 6379
  debug:
    msg: "{{ lookup('redis', 'weather','temp' ) }}"
```

или определить настройки в *ansible.cfg*:

```
[lookup_redis]
host: redis2.example.com
port: 6666
```

Кроме того, Redis можно настроить как кластер.

Написание собственного плагина подстановки

Если ни один из имеющихся плагинов вас не устраивает, всегда можно написать свой. Разработка собственных плагинов подстановок не является темой данной книги, но если вас действительно заинтересовал данный вопрос, то я предлагаю изучить исходный код плагинов входящих в состав Ansible (**https://oreil.ly/DbSU4**).

Написав свой плагин, поместите его в один из следующих каталогов:

- *lookup_plugins* в каталоге со сценарием;
- *~/.ansible/plugins/lookup*;
- */usr/share/ansible/plugins/lookup*;
- указанный в переменной окружения `ANSIBLE_LOOKUP_PLUGINS`.

Сложные циклы

До сих пор, описывая задачи, которые выполняют обход списка объектов, мы использовали выражение `with_items` и в нем определяли список объектов. Это самый распространенный способ выполнения операций в цикле, но Ansible поддерживает также другие механизмы итераций. Например, с помощью ключевого слова `until` можно повторять задачу снова и снова, пока она не завершится с признаком успеха:

```
- name: Unarchive maven
  unarchive:
    src: "{{  maven_url }}"
    dest: "{{  maven_location }}"
    copy: false
    mode: '0755'
  register: maven_download
  until: maven_download is success
  retries: 5
  delay: 3
```

Ключевое слово `loop` работает эквивалентно `with_items`, но список должен быть однородным, т. е. он не должен содержать разнотипные данные (такие как скаляры, массивы и словари). С помощью `loop` можно делать все, что угодно! В официальной документации (**https://oreil.ly/bgbdX**) эта тема рассматривается достаточно подробно, поэтому я приведу лишь несколько примеров, чтобы дать вам представление, как работают эти конструкции. Вот один из примеров сложных циклов:

```
- name: Iterate with loop
  debug:
    msg: "KPI: {{ item.kpi }} prio: {{ i + 1 }} goto: {{ item.dept }}"
  loop:
    - kpi: availability
      dept: operations
    - kpi: performance
      dept: development
    - kpi: security
      dept: security
  loop_control:
    index_var: i
    pause: 3
```

Вы можете передать список непосредственно большинству модулей, управляющих диспетчерами пакетов, таким как `apt`, `yum` и `package`. В старых сценариях часто можно встретить `with_items`, но в настоящее время эта конструкция почти не используется, и теперь принято определять задачи так:

```
- name: Install packages
  become: true
  package:
    name: "{{ list_of_packages }}"
    state: present
```

*Плагины with_**

Важно помнить, что `with_items` опирается на плагин подстановки; `items` – это лишь один из запросов. В табл. 10.4 перечислены другие доступные конструкции организации циклов, основанные на использовании плагина подстановки. Если понадобится, то вы сможете подключить свой плагин подстановки для выполнения итераций.

Таблица 10.4. Циклические конструкции

Имя	Вход	Способ выполнения цикла
`with_items`	Список	Цикл по списку элементов
`with_lines`	Команда для выполнения	Цикл по строкам вывода команды
`with_fileglob`	Шаблон поиска	Цикл по именам файлов
`with_first_found`	Список путей	Первый существующий файл
`with_dict`	Словарь	Цикл по элементам словаря
`with_flattened`	Список списков	Цикл по всем элементам вложенных списков
`with_indexed_items`	Список	Одна итерация
`with_nested`	Список	Вложенный цикл
`with_random_choice`	Список	Одна итерация
`with_sequence`	Последовательность целых чисел	Цикл по последовательности
`with_subelements`	Список словарей	Вложенный цикл
`with_together`	Список списков	Цикл по элементам объединенного списка
`with_inventory_hostnames`	Шаблон хоста	Цикл по хостам, соответствующим шаблону

Рассмотрим поближе наиболее важные из этих конструкций.

with_lines

Конструкция `with_lines` позволяет выполнять произвольные команды на управляющей машине и производить итерации по строкам в результатах.

Представьте, что у вас есть файл со списком имен и вы хотите, чтобы компьютер произнес их. Пусть содержимое файла выглядит так:

```
Ronald Linn Rivest
Adi Shamir
Leonard Max Adleman
Whitfield Diffie
Martin Hellman
```

В примере 10.18 показано, как использовать `with_lines` для чтения файла и выполнения итераций по строкам, содержащимся в нем.

Пример 10.18. Цикл с помощью *with_lines*

```
- name: Iterate over lines in a file
  say:
    msg: "{{ item }}"
  with_lines:
    - cat files/turing.txt
```

with_fileglob

Конструкция `with_fileglob` используется, когда нужно выполнить итерации по набору файлов на управляющей машине.

В примере 10.19 показано, как выполнить обход файлов с расширением *.pub* в каталоге */var/keys*, а также в подкаталоге *keys*, находящемся в одном каталоге со сценарием. Затем с помощью плагина `file` из каждого найденного файла извлекается его содержимое и передается модулю `author_key`.

Пример 10.19. Использование *with_fileglob* для добавления ключей

```
- name: Add public keys to account
  become: true
  authorized_key:
    user: deploy
    key: "{{ lookup('file', item) }}"
  with_fileglob:
    - /var/keys/*.pub
    - keys/*.pub
```

with_dict

Конструкция `with_dict` выполняет обход элементов словаря. При использовании этой конструкции переменная цикла `item` является словарем с двумя ключами:

- *key* – один из ключей в словаре;
- *value* – значение, соответствующее ключу *key*.

Например, если хост имеет интерфейс `enp0s8`, тогда в Ansible будет существовать факт с именем `ansible_enp0s8` и с ключом `ipv4`, содержащим примерно такой словарь:

```
{
 "address": "192.168.33.10",
 "broadcast": "192.168.33.255",
 "netmask": "255.255.255.0",
 "network": "192.168.33.0"
}
```

Можно обойти элементы этого словаря и вывести их по одному:

```
- name: Iterate over ansible_enp0s8
  debug:
    msg: "{{ item.key }}={{ item.value }}"
  with_dict: "{{ ansible_enp0s8.ipv4 }}"
```

Результат будет выглядеть так:

```
TASK [Iterate over ansible_enp0s8] *********************************************
ok: [web] => (item={'key': 'address', 'value': '192.168.33.10'}) => {
    "msg": "address=192.168.33.10"
}
ok: [web] => (item={'key': 'broadcast', 'value': '192.168.33.255'}) => {
    "msg": "broadcast=192.168.33.255"
}
ok: [web] => (item={'key': 'netmask', 'value': '255.255.255.0'}) => {
    "msg": "netmask=255.255.255.0"
}
ok: [web] => (item={'key': 'network', 'value': '192.168.33.0'}) => {
    "msg": "network=192.168.33.0"
}
```

Возможность итераций по словарям часто помогает уменьшить объем кода.

Циклические конструкции как плагины подстановок

Циклические конструкции реализованы в Ansible как плагины подстановок. Достаточно подставить `with` в начало имени плагина подстановки, чтобы использовать его в форме цикла. Так, пример 10.12 можно переписать с использованием формы `with_file`, как показано в примере 10.20.

Пример 10.20. Использование подстановки *file* в качестве конструкции цикла

```
- name: Add my public key for SSH
  authorized_key:
```

```
    user: vagrant
    key: "{{ item }}"
    key_options: 'from="10.0.2.2"'
    exclusive: true
  with_file: '~/.ssh/id_ed25519.pub'
```

Обычно плагины подстановок используются в роли циклических конструкций, только если требуется получить список. Именно поэтому мы отделили плагины в табл. 10.3 (возвращающие строки) от плагинов в табл. 10.4 (возвращающие списки).

Управление циклами

Ansible предоставляет пользователям более богатые возможности выполнения циклических операций, чем большинство языков программирования, но это не означает, что вы должны использовать все богатство вариантов. Старайтесь не усложнять свой код!

Выбор имени переменной цикла

Выражение `loop_var` позволяет дать переменной цикла другое имя, отличное от имени `item`, используемого по умолчанию, как показано в примере 10.21.

Пример 10.21. Использование имени user для переменной цикла

```
- name: Add users
  become: true
  user:
    name: "{{ user.name }}"
  with_items:
    - { name: gil }
    - { name: sarina }
    - { name: leanne }
  loop_control:
    loop_var: user
```

В примере 10.21 выражение `loop_var` дает лишь косметическое удобство, но вообще с его помощью можно определять гораздо более сложные циклы.

В примере 10.22 реализован цикл по нескольким задачам. Для этого в нем используется инструкция `include` с выражением `with_items`.

Однако файл *vhosts.yml* может включать также задачи, использующие выражение `with_items` для своих целей. Такая реализация могла бы породить конфликты из-за совпадения имен переменных цикла, используемых по умолчанию. Чтобы предотвратить такие конфликты, можно указать другое имя в выражении `loop_var` для внешнего цикла.

Пример 10.22. Использование имени *vhost* для переменной цикла

```
- name: Run a set of tasks in one loop
  include: vhosts.yml
  with_items:
    - { domain: www1.example.com }
    - { domain: www2.example.com }
    - { domain: www3.example.com }
  loop_control:
    loop_var: vhost
```

В подключаемой задаче (объявленной в файле *vhosts.yml*), которая представлена в примере 10.23, мы теперь без опаски можем использовать имя `item` по умолчанию.

Пример 10.23. Подключаемый файл может содержать циклы

```
- name: Create nginx directories
  file:
    path: "/var/www/html/{{ vhost.domain }}/{{ item }}"
  state: directory
  with_items:
    - logs
    - public_http
    - public_https
    - includes

- name: Create nginx vhost config
  template:
    src: "{{ vhost.domain }}.j2"
    dest: /etc/nginx/conf.d/{{ vhost.domain }}.conf
```

Мы оставили имя по умолчанию для переменной внутреннего цикла.

Управление выводом

В версии Ansible 2.2 появилось новое выражение `label`, помогающее до определенной степени управлять выводом цикла.

Следующий пример содержит обычный список словарей:

```
- name: Create nginx vhost configs
  become: true
  template:
    src: "{{ item.domain }}.conf.j2"
    dest: "/etc/nginx/conf.d/{{ item.domain }}.conf"
    mode: '0640'
  with_items:
    - { domain: www1.example.com, tls_enabled: true }
    - { domain: www2.example.com, tls_enabled: false }
```

```
    - { domain: www3.example.com, tls_enabled: false,
      aliases: [ edge2.www.example.com, eu.www.example.com ] }
```

По умолчанию Ansible выводит словари целиком. Если словари большие, читать вывод становится очень трудно:

```
TASK [Create nginx vhost configs] ******************************************
changed: [web] => (item={'domain': 'www1.example.com', 'tls_enabled': True})
changed: [web] => (item={'domain': 'www2.example.com', 'tls_enabled': False})
changed: [web] => (item={'domain': 'www3.example.com', 'tls_enabled': False,
'aliases': ['edge2.www.example.com', 'eu.www.example.com']})
```

Исправить эту проблему поможет выражение `label`.

Поскольку нас интересуют только доменные имена, мы можем просто добавить в раздел `loop_control` выражение `label`, описывающее, что именно должно выводиться при обходе элементов:

```
- name: Create nginx vhost configs
  become: true
  template:
    src: "{{ item.domain }}.conf.j2"
    dest: "/etc/nginx/conf.d/{{ item.domain }}.conf"
    mode: '0640'
  with_items:
    - { domain: www1.example.com, tls_enabled: true }
    - { domain: www2.example.com, tls_enabled: false }
    - { domain: www3.example.com, tls_enabled: false,
      aliases: [ edge2.www.example.com, eu.www.example.com ] }
  loop_control:
    label: "for domain {{ item.domain }}"
```

В результате вывод получится более удобочитаемым:

```
TASK [Create nginx vhost configs] ******************************************
ok: [web] => (item=for domain www1.example.com)
ok: [web] => (item=for domain www2.example.com)
ok: [web] => (item=for domain www3.example.com)
```

Имейте в виду, что, если используется флаг `-v` подробного вывода, словари будут выводиться целиком; не используйте этот флаг, чтобы скрыть пароли от посторонних глаз! Устанавливайте в критических задачах `no_log: true`.

Импортирование и подключение

Функции `import_*` позволяют подключать задачи и даже целые роли в разделе задач операции с помощью ключевых слов `import_tasks` и `import_`

role. Когда файлы *импортируются* в другие сценарии статически, Ansible запускает операции и задачи в каждом импортированном сценарии в том порядке, в каком они определены, как если бы они были определены непосредственно в основном сценарии.

Функции include_* позволяют динамически подключать задачи, переменные и даже целые роли с помощью ключевых слов include_tasks, include_vars и include_role. Эта возможность часто применяется в ролях для определения или группировки задач и их аргументов в отдельных подключаемых файлах. Подключаемые роли и задачи могут выполняться или не выполняться в зависимости от результатов других задач в сценарии. Когда цикл используется с include_tasks или include_role, подключаемые задачи или роли будут выполняться в каждой итерации цикла для каждого элемента.

Обратите внимание, что простое ключевое слово include устарело, и в настоящее время рекомендуется использовать include_tasks, include_vars и include_role.

Рассмотрим пример. В примере 10.24 определены две задачи, использующие идентичные аргументы become, when и tags.

Пример 10.24. Идентичные аргументы

```
- name: Install nginx
  become: true
  when: ansible_os_family == 'RedHat'
  package:
    name: nginx
  tags:
    - nginx

- name: Ensure nginx is running
  become: yes
  when: ansible_os_family == 'RedHat'
  service:
    name: nginx
    state: started
    enabled: yes
  tags:
    -nginx
```

Если выделить эти две задачи в отдельный файл, как показано в примере 10.25, и подключать его, как показано в примере 10.26, то можно упростить сценарий, определив аргументы только в include_tasks.

Пример 10.25. Выделение задач в отдельный файл

```
- name: Install nginx
  package:
    name: nginx

- name: Ensure nginx is running
  service:
    name: nginx
    state: started
    enabled: yes
```

Пример 10.26. Подключение задач и применение общих аргументов

```
- include_tasks: nginx_include.yml
  become: yes
  when: ansible_os_family == 'RedHat'
  tags: nginx
```

Динамическое подключение

Задачи , характерные для конкретной операционной системы, в ролях часто определяются в отдельных файлах. В зависимости от количества операционных систем, поддерживаемых ролью, для подключения задач может потребоваться масса шаблонного кода в `include_tasks`:

```
- include_tasks: Redhat.yml
  when: ansible_os_family == 'Redhat'

- include_tasks: Debian.yml
  when: ansible_os_family == 'Debian'
```

Начиная с версии 2.0, Ansible позволяет динамически подключать файлы, используя подстановку переменных. Этот прием называется *динамическим подключением*:

```
- name: Play platform specific actions
  include_tasks: "{{ ansible_os_family }}.yml"
```

Однако такое решение на основе динамического подключения имеет свой недостаток: команда `ansible-playbook --list-tasks` может не вывести задачи, подключаемые динамически, если Ansible не имеет информации для заполнения переменных, определяющих подключаемые файлы. Например, переменные-факты (см. главу 5) не заполняются, когда используется аргумент `--list-tasks`.

Подключение ролей

Выражение `include_role` – это особый вид операции подключения. В отличие от выражения `import_role`, которое статически импортирует

все компоненты роли, выражение include_role позволяет явно определить, какие компоненты подключаемой роли должны использоваться:

```
- name: Install nginx
  yum:
    pkg: nginx

- name: Install php
  include_role:
    name: php

- name: Configure nginx
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
```

Выражение include_role также открывает доступ к обработчикам, что позволяет, например, организовать обработку уведомления о перезапуске.

Поток управления роли

В каталоге задач роли можно определить отдельные файлы с задачами для разных вариантов использования, а в файле задач *main.yml* использовать include_tasks для каждого варианта. Однако выражение include_role может запускать отдельные компоненты ролей, указанные в выражении tasks_from. Представьте, что в зависимости от роли, которая выполняется перед ролью *main*, задача file меняет владельца файла. Но в этот момент соответствующая учетная запись еще не создана. Она будет создана позднее, в главной роли, во время установки пакета:

```
- name: Install nginx
  yum:
    pkg: nginx

- name: Install php
  include_role:
    name: php
    tasks_from: install          ❶

- name: Configure nginx
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf

- name: Configure php
```

```
include_role:
  name: php
  tasks_from: configure          ❷
```

❶ Подключает и выполняет *install.yml* из роли `php`.

❷ Подключает и выполняет *configure.yml* из роли `php`.

Блоки

Подобно выражениям `include_*`, выражение `block` реализует механизм группировки задач. Выражение `block` позволяет определять условия или аргументы сразу для всех задач в блоке:

```
- block:
  - name: Install nginx
    package:
      name: nginx

  - name: Ensure nginx is running
    service:
      name: nginx
      state: started
      enabled: yes

  become: yes
  when: "ansible_os_family == 'RedHat'
```

В отличие от `include_*` выражение `block` пока не поддерживает циклов.

Выражение `block` имеет еще одно, намного более интересное применение: обработку ошибок.

Обработка ошибок с помощью блоков

Обработка ошибок всегда была непростой задачей. Система Ansible изначально предусматривает возможность появления ошибок на хостах. Если возникает какая-то ошибка, она по умолчанию просто исключает хост из игры и продолжает настраивать другие хосты, где ошибок не наблюдалось.

Комбинацией выражений `serial` и `max_fail_percentage` Ansible дает возможность выполнить какие-то действия, когда операция объявляется потерпевшей неудачу. А благодаря выражению `block`, как показано в примере 10.27, Ansible поднимает обработку ошибок на уровень выше

и позволяет автоматизировать повторное выполнение или откат задач, потерпевших ошибку.

Пример 10.27. *app-upgrade.yml*

```
- block:    ❶
    - debug: msg="You will see a failed tasks right after this"

    - name: Returns 1
      command: /usr/bin/false

    - debug: msg="You never see this message"

  rescue:    ❷
    - debug: msg="You see this message in case of failure in the block"

 always:     ❸
    - debug: msg="This will be always executed"
```

❶ Начало выражения `block`.

❷ `rescue` перечисляет задачи, выполняемые, если в выражении `block` произойдет ошибка.

❸ Задачи, которые выполняются всегда.

Если у вас есть опыт программирования, то способ обработки ошибок в Ansible может напомнить вам парадигму try-except-finally , и она работает практически так же, как в следующей функции `division` на Pyhon:

```
def division(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

Для демонстрации возьмем самую обычную повседневную задачу: обновление приложения. Приложение распределяется в кластере виртуальных машин (ВМ) и развертывается в облаке IaaS (Apache CloudStack [**https://oreil.ly/zIDUh**]). Кроме того, облако CloudStack поддерживает возможность создания моментальных снимков ВМ. Упрощенный сценарий, выполняющий эту работу, действует по следующему алгоритму.

1. Забрать ВМ из-под управления балансировщиком нагрузки.
2. Создать снимок ВМ перед обновлением приложения.

3. Обновить приложение.
4. Выполнить тестирование.
5. Откатиться обратно, если что-то пошло не так.
6. Вернуть ВМ под управление балансировщиком нагрузки.
7. Удалить снимок ВМ.

Давайте реализуем этот алгоритм в виде сценария Ansible, максимально сохранив простоту (см. пример 10.28).

Пример 10.28. *app-upgrade.yml*

```
---

- hosts: app-servers
  serial: 1
  tasks:
    - name: Take VM out of the load balancer
    - name: Create a VM snapshot before the app upgrade
    - block:
        - name: Upgrade the application
        - name: Run smoke tests
      rescue:
        - name: Revert a VM to the snapshot after a failed upgrade
      always:
        - name: Re-add webserver to the loadbalancer
        - name: Remove a VM snapshot
...
```

Этот сценарий почти наверняка вернет действующую ВМ в кластер, под управление балансировщика нагрузки, даже если попытка обновления потерпит неудачу.

Задачи в выражении `always` будут выполняться всегда, даже в случае обнаружения ошибок при выполнении задач в выражении `rescue`! Тщательно отбирайте задачи, помещаемые в `always`.

Если под управление балансировщиком нагрузки должна возвращаться только обновленная ВМ, то сценарий нужно изменить, как показано в примере 10.29.

Пример 10.29. *app-upgrade.yml*

```
---
- hosts: app-servers
```

```
  serial: 1

  tasks:

    - name: Take VM out of the load balancer

    - name: Create a VM snapshot before the app upgrade

    - block:
        - name: Upgrade the application
        - name: Run smoke tests

      rescue:
        - name: Revert a VM to the snapshot after a failed upgrade

        - name: Re-add webserver to the loadbalancer
        - name: Remove a VM snapshot
...
```

В этой версии исчезло выражение `always`, а две его задачи помещены в конец операции. Они будут запущены, *только* если выражение `rescue` выполнится успешно. То есть под управление балансировщика нагрузки будут возвращаться только обновленные ВМ.

Окончательная версия сценария представлена в примере 10.30.

Пример 10.30. Сценарий обновления приложения с обработкой ошибок

```
 ---
- hosts: app-servers
  serial: 1
  tasks:

  - name: Take app server out of the load balancer
    delegate_to: localhost
    cs_loadbalancer_rule_member:
      name: balance_http
      vm: "{{ inventory_hostname_short }}"
      state: absent

  - name: Create a VM snapshot before an upgrade
    delegate_to: localhost
    cs_vmsnapshot:
      name: Snapshot before upgrade
      vm: "{{ inventory_hostname_short }}"
      snapshot_memory: true

  - block:
```

```
        - name: Upgrade the application
          script: upgrade-app.sh
        - name: Run smoke tests
          script: smoke-tests.sh
      rescue:
        - name: Revert the VM to a snapshot after a failed upgrade
          delegate_to: localhost
          cs_vmsnapshot:
            name: Snapshot before upgrade
            vm: "{{ inventory_hostname_short }}"
            state: revert

    - name: Re-add app server to the loadbalancer
      delegate_to: localhost
      cs_loadbalancer_rule_member:
        name: balance_http
        vm: "{{ inventory_hostname_short }}"
        state: present

    - name: Remove a VM snapshot after successful upgrade or successful rollback
      delegate_to: localhost
      cs_vmsnapshot:
        name: Snapshot before upgrade
        vm: "{{ inventory_hostname_short }}"
        state: absent
...
```

Шифрование конфиденциальных данных при помощи Vault

Сценарию установки Mezzanine требуется доступ к конфиденциальной информации, такой как пароли базы данных и администратора. Мы уже имели с этим дело в главе 6, где поместили все конфиденциальные данные в отдельный файл *secrets.yml*. Этот файл хранился вне системы управления версиями.

Ansible предлагает альтернативное решение: вместо хранения файла *secrets.yml* вне системы управления версиями можно создать его зашифрованную копию. В этом случае, если наша система управления версиями будет взломана, нарушитель не получит доступа к содержимому файла *secrets.yml*, если не располагает паролем для дешифрования.

Утилита командной строки `ansible-vault` позволяет создавать и редактировать зашифрованный файл, который `ansible-playbook` будет автоматически распознавать и расшифровывать с помощью пароля.

Шифрование в состоянии покоя

Этот инструмент способен шифровать только данные, находящиеся в состоянии покоя (т. е. на диске). Имейте в виду, что в задачах, использующих конфиденциальные данные, желательно устанавливать параметр `no_log: true`.

Вот как можно зашифровать имеющийся файл:

```
$ ansible-vault encrypt secrets.yml
```

Также можно создать новый зашифрованный файл в специальном каталоге *group_vars/all/next* в папке со сценарием. Бас хранит глобальные переменные в *group_vars/all/vars.yml*, а конфиденциальные данные – в *group_vars/all/vault* (без расширения, чтобы не сбивать с толку линтеры и редакторы).

```
$ mkdir -p group_vars/all/
$ ansible-vault create group_vars/all/vault
```

Вам будет предложено ввести пароль, а затем `ansible-vault` запустит текстовый редактор, чтобы вы могли заполнить файл. Для редактирования используется редактор, указанный в переменной окружения `$EDITOR`. Если эта переменная не определена в файле профиля командной оболочки (`export EDITOR=code`), то по умолчанию используется `vim`.

В примере 10.31 показано, как выглядит содержимое файла, зашифрованного с помощью `ansible-vault`.

Пример 10.31. Содержимое файла, зашифрованного с помощью *ansible-vault*

```
$ANSIBLE_VAULT;1.1;AES256
38626635666338393730353966303331643566646561363838333832623138613931363835363963
36383965386264333937633861366362353261396336666640a343437613564616635316532373635
...
35373564313132356663633633346136376332633665373634363234666363356530386562616463
35343436313638613837386661336366633832333938666532303931346434386433
```

К файлу, зашифрованному с помощью `ansible-vault`, можно обращаться в секции `vars_files` как к обычному файлу – вам не придется ничего менять в примере 7.28, если зашифровать файл *secrets.yml*.

Однако, чтобы не происходило ошибки при обращении к зашифрованному файлу, нужно подсказать утилите `ansible-playbook`, что она должна запросить пароль перед чтением зашифрованного файла. Для этого достаточно передать аргумент `--ask-vault-pass`:

```
$ ansible-playbook --ask-vault-pass playbook.yml
```

Также можно сохранить пароль в текстовом файле и сообщить `ansible-playbook`, где он находится, настроив переменную окружения `ANSIBLE_VAULT_PASSWORD_FILE` или добавив аргумент `--vault-password-file`:

```
$ ansible-playbook playbook.yml --vault-password-file ~/password.txt
```

Если аргумент параметра `--vault-password-file` представляет выполняемый файл, Ansible запустит его и использует содержимое стандартного вывода как пароль. Благодаря этому для передачи пароля в Ansible можно использовать сценарии.

В табл. 10.5 перечислены доступные команды `ansible-vault`.

Таблица 10.5. Команды *ansible-vault*

Команда	Описание
`ansible-vault encrypt` *file.yml*	Шифрует текстовый файл *file.yml*
`ansible-vault decrypt` *file.yml*	Дешифрует зашифрованный файл *file.yml*
`ansible-vault view` *file.yml*	Выводит содержимое зашифрованного файла *file.yml*
`ansible-vault create` *file.yml*	Создает новый зашифрованный файл *file.yml*
`ansible-vault edit` *file.yml*	Открывает в редакторе зашифрованный файл *file.yml*
`ansible-vault rekey` *file.yml*	Изменяет пароль к зашифрованному файлу *file.yml*

Шифрование с использованием разных паролей

Одного пароля может быть достаточно для небольшой команды, но иногда бывает желательно разделить задачи и использовать разные пароли в разных окружениях. В версии 2.4 появилась поддержка отдельного идентификатора шифрования Vault-ID для определенного зашифрованного файла. Такой идентификатор подобен имени конкретного пароля; например, для среды разработки можно определить идентификатор «dev», а для промышленной среды – идентификатор «prod».

В файле *ansible.cfg* в разделе `[defaults]` мы создаем ссылку на идентификаторы паролей и соответствующие им файлы паролей (эти файлы должны существовать):

```
[defaults]
vault_identity_list = dev@~/.vault_dev, prod@~/.vault_prod
```

Когда с помощью идентификатора шифруются данные для промышленной среды:

```
ansible-vault encrypt --encrypt-vault-id=prod group_vars/prod/vault
```

в заголовок файла помещается соответствующий идентификатор Vault-ID:

```
$ANSIBLE_VAULT;1.2;AES256;prod
```

Заключение

Ansible имеет множество функций, помогающих гибко обрабатывать пограничные случаи, будь то обработка ошибок и исключений, ввод и преобразование данных, итерации или использование конфиденциальных данных. В этой главе были представлены некоторые дополнительные возможности Ansible – вы можете вернуться к ней, когда они вам действительно понадобятся. Следующая глава будет более полезна для начинающих.

Глава 11

Управление хостами, задачами и обработчиками

Иногда поведение по умолчанию системы Ansible не в полной мере соответствует на шим желаниям. В этой главе мы познакомимся с инструментами Ansible, помогающими выбирать хосты для обслуживания, запускать задачи и использовать обработчики.

Шаблоны для выбора хостов

До сих пор параметр `host` в наших операциях определял единичный хост или группу, например:

```
hosts: web
```

Однако вместо единичного хоста или группы можно указать *шаблон* . Мы уже видели шаблон `all`, который позволяет запускать задачи на всех известных хостах:

```
hosts: all
```

Можно определить объединение двух групп с помощью двоеточия, например все машины в группах `dev` и `staging`:

```
hosts: dev:staging
```

С помощью двоеточия и знака амперсанда (`&`) можно определить пересечение. Например, все серверы баз данных в окружении обкатки (группа `staging`) можно выбрать так:

```
hosts: staging:&database
```

В табл. 11.1 перечислены шаблоны, поддерживаемые в Ansible. Обратите внимание, что регулярные выражения всегда начинаются со знака тильды (~).

Таблица 11.1. Поддерживаемые шаблоны

Действие	Пример использования
Все хосты	`all`
Все хосты	`*`
Объединение	`dev:staging`
Пересечение	`staging:&database`
Исключение	`dev:!queue`
Шаблон подстановки	`*.example.com`
Диапазон нумерованных серверов	`web[5:10]`
Регулярное выражение	`~web\d\.example\.(com|org)`

Ansible поддерживает также комбинации шаблонов. Например:

```
hosts: dev:staging:&database:!queue
```

Ограничение обслуживаемых хостов

Для ограничения перечня хостов, на которых будет выполняться сценарий, используется флаг `-l` или `--limit`, как показано в примере 11.1.

Пример 11.1. Ограничение перечня обслуживаемых хостов

```
$ ansible-playbook -l <образец> playbook.yml

$ ansible-playbook --limit <образец> playbook.yml
```

Для определения комбинаций хостов можно использовать только что описанный синтаксис шаблонов, например:

```
$ ansible-playbook -l 'staging:&database' playbook.yml
```

Запуск задачи на управляющей машине

Иногда необходимо выполнить конкретную задачу на управляющей машине. Для этого Ansible предлагает выражение `delegate_to: localhost`.

Серверы в большинстве организаций не имеют прямого доступа к интернету, но есть возможность загрузить необходимые файлы через прокси-сервер на управляющую машину. В таком случае можно делегировать загрузку файлов управляющей машине:

```
- name: Download goss binary
  delegate_to: localhost
  connection: local
  become: false
  get_url:
```

```
    url: "https://oreil.ly/RuRsL"
    dest: "~/Downloads/goss"
    mode: '0755'
  ignore_errors: true
```

Бас использует выражение `ignore_errors : true`, потому что в случае неудачи приходится использовать теневой ИТ-ресурс[1], чтобы получить файл и поместить его в каталог *Downloads* вручную. Goss – очень мощный инструмент тестирования серверов, основанный на спецификации YAML.

Сбор фактов вручную

В случаях, когда сервер SSH еще не запущен, полезно явно отключить сбор фактов. В противном случае Ansible попытается установить соединение с хостом и собрать факты еще до запуска первой задачи. Поскольку доступ к фактам необходим (напоминаю, что мы используем факт `ansible_env` в нашем сценарии), можно обратиться к модулю `setup` для инициации сбора фактов, как показано в примере 11.2.

Пример 11.2. Ожидание запуска SSH-сервера

```
---
- name: Chapter 9 playbook
  hosts: web
  gather_facts: false
  become: false
  tasks:
    - name: Wait for web ssh daemon to be running
      wait_for:
        port: 22
        host: "{{ inventory_hostname }}"
        search_regex: OpenSSH

    - name: Gather facts
      setup:
...
```

Получение IP-адреса хоста

В нашем сценарии несколько имен хостов искусственно создано из IP-адреса веб-сервера.

```
live_hostname: 192.168.33.10.xip.io
domains:
```

[1] Под использованием *теневого ИТ-ресурса* понимается практика, когда людям приходится прибегать к возможности получения требуемых файлов обходными путями, если (центральное) подразделение ИТ ограничивает доступ к коду, находящемуся в интернете. Например, двоичный файл можно упаковать в документ Microsoft Word в формате *uuencode* и отправить по электронной почте на свой рабочий компьютер.

```
  - 192.168.33.10.xip.io
  - www.192.168.33.10.xip.io
```

А если мы захотим использовать такую же схему, но не определять IP-адреса в переменных? В этом случае, если IP-адрес веб-сервера изменится, нам не придется вносить изменений в сценарий.

Ansible получает IP-адрес каждого хоста и сохраняет его в `ansible_facts`. Каждый сетевой интерфейс представлен связанным с ним фактом. Например, данные о сетевом интерфейсе `eth0` хранятся в факте `ansible_eth0`. Это показано в примере 11.4.

Пример 11.4. Факт *ansible_eth0*

```
"ansible_eth0": {
    "active": true,
    "device": "eth0",
    "ipv4": {
        "address": "10.0.2.15",
        "broadcast": "10.0.2.255",
        "netmask": "255.255.255.0",
        "network": "10.0.2.0"
    },
    "ipv6": [
        {
            "address": "fe80::5054:ff:fe4d:77d3",
            "prefix": "64",
            "scope": "link"
        }
    ],
    "macaddress": "52:54:00:4d:77:d3",
    "module": "e1000",
    "mtu": 1500,
    "promisc": false,
    "speed": 1000,
    "type": "ether"
}
```

Наша машина Vagrant имеет два интерфейса, `eth0` и `eth1`. Интерфейс `eth0` – приватный, с IP-адресом (*10.0.2.15*), недоступным для нас. Интерфейс `eth1` – тот самый, которому мы присвоили IP-адрес в нашем файле Vagrantfile (*192.168.33.10*).

Мы можем определить переменные следующим образом:

```
live_hostname: "{{ ansible_facts.eth1.ipv4.address }}.xip.io"
domains:
  - "{{ ansible_facts.eth1.ipv4.address }}.xip.io"
  - "www.{{ ansible_facts.eth1.ipv4.address }}.xip.io"
Running a Task on a Machine Other than the Host
```

Запуск задачи на сторонней машине

Иногда необходимо запустить задачу, связанную с хостом, но на другом сервере. Для этого можно использовать выражение `delegate_to`.

Обычно это требуется в двух случаях:

- для активации триггеров в системах мониторинга, таких как Nagios;
- для передачи хоста под управление балансировщика нагрузки, такого как HAProxy.

Представьте, например, что нам необходимо активировать триггеры Nagios для всех хостов в группе `web`. Допустим, у нас в реестре имеется запись *nagios.example.com*. На этом хосте запущена система мониторинга Nagios. В примере 11.5 показано, как можно было бы использовать выражение `delegate_to` в этом случае.

Пример 11.5. Использование *delegate_to* для настройки Nagios

```
- name: Enable alerts for web servers
  hosts: web
  tasks:
    - name: enable alerts
      delegate_to: nagios.example.com
      nagios:
        action: enable_alerts
        service: web
        host: "{{ inventory_hostname }}"
```

В этом примере Ansible выполняет задачу `nagios` на сервере *nagios.example.com*, но переменная `inventory_hostname`, используемая в операции, ссылается на хост `web`.

Более подробно о `delegate_to` рассказывается в *lamp_haproxy/rolling_update.yml*, в примерах проекта Ansible (**https://oreil.ly/XtkLO**).

Последовательное выполнение задачи на хостах по одному

По умолчанию Ansible выполняет каждую задачу на всех хостах параллельно. Но иногда требуется, чтобы задача выполнялась на хостах по очереди. Каноническим примером является обновление серверов приложений, которые действуют под управлением балансировщика нагрузки. Обычно сервер приложений выводится из-под управления балансировщиком нагрузки, обновляется и возвращается обратно. При этом не хотелось бы приостанавливать *все* серверы приложений сразу, потому что в этом случае служба станет недоступной.

Ограничить число хостов, на которых Ansible запускает сценарий, можно выражением `serial`. В примере 11.6 продемонстрирован последовательный вывод хостов из-под управления балансировщиком нагрузки Amazon EC2, обновление системных пакетов и возвращение хостов обратно. (Подробнее об Amazon EC2 рассказывается в главе 14.)

Пример 11.6. Вывод хостов из-под управления балансировщиком нагрузки и обновление пакетов

```
---
- name: Upgrade packages on servers behind load balancer
  hosts: myhosts
  serial: 1
  tasks:
    - name: Get the ec2 instance id and elastic load balancer id
      ec2_facts:

    - name: Take the host out of the elastic load balancer
      delegate_to: localhost
      ec2_elb:
        instance_id: "{{ ansible_ec2_instance_id }}"
        state: absent

    - name: Upgrade packages
      apt:
        update_cache: true
        upgrade: true

    - name: Put the host back in the elastic load balancer
      delegate_to: localhost
      ec2_elb:
        instance_id: "{{ ansible_ec2_instance_id }}"
        state: present
        ec2_elbs: "{{ item }}"
      with_items: ec2_elbs
...
```

В нашем примере мы передали выражению `serial` аргумент `1`, сообщив системе Ansible, что хосты должны обрабатываться последовательно. Если бы мы передали `2`, Ansible обрабатывала бы по два хоста сразу.

Обычно, когда задача терпит неудачу, Ansible прекращает обработку данного хоста, но продолжает обработку остальных. Если используется балансировщик нагрузки, то, возможно, практичнее будет отменить выполнение всей операции до того, как ошибка возникнет на всех хостах. Иначе может получиться так, что все хосты будут выведены из-под управления балансировщиком нагрузки и ему нечем будет управлять.

Определить максимальное количество хостов, находящихся в состоянии ошибки (в процентах), по достижении которого Ansible прекратит выполнение операции, можно с помощью выражения `max_fail_percentage` вместе с `serial`. Например, допустим, что мы указали максимальный процент неудач 25 %:

```
- name: Upgrade packages on servers behind load balancer
  hosts: myhosts
  serial: 1
  max_fail_percentage: 25
  tasks:
    # далее следуют задачи
```

Если бы у нас было 4 хоста и один потерпел неудачу при выполнении задачи, тогда Ansible продолжила бы выполнение операции, потому что порог в 25 % не превышен. Однако если на втором хосте задача также завершится с ошибкой, тогда Ansible остановит всю операцию, поскольку уже 50 % хостов будут находиться в состоянии ошибки, а это выше 25 %. Чтобы остановить операцию при первой же ошибке, установите `max_fail_percentage` равным `0`.

Пакетная обработка хостов

В выражение `serial` тоже можно передать проценты вместо числа хостов. В этом случае Ansible сама определит, сколько хостов из числа участвующих в операции соответствуют этому значению, как показано в примере 11.7.

Пример 11.7. Использование процентов в выражении serial

```
- name: Upgrade 50% of web servers
  hosts: myhosts
  serial: 50%
  tasks:
    # далее следуют задачи
```

Можно пойти еще дальше, например выполнить операцию сначала на одном хосте, убедиться, что все прошло благополучно, а затем последовательно выполнять операцию на большем числе хостов сразу. Это может пригодиться для управления большими логическими кластерами независимых хостов; например 30 хостами в сети доставки содержимого (Content Delivery Network, CDN).

Для реализации такого поведения, начиная с версии 2.2, Ansible позволяет задавать в выражении `serial` список с размерами пакетов. Элементами этого могут быть целые числа или проценты, как показано в примере 11.8.

Пример 11.8. Использование списка с размерами пакетов в выражении *serial*

```
- name: Configure CDN servers
  hosts: cdn
  serial:
    - 1
    - 30%
  tasks:
    # далее следуют задачи
```

Ansible будет ограничивать количество хостов в каждом пакете, следуя по списку в `serial`, пока не будет достигнут последний его элемент или не останется хостов для обработки. Это значит, что последний элемент в списке `serial` продолжит действовать до окончания операции, пока не будут обработаны все хосты.

Если предположить, что предыдущая операция охватывает 30 хостов сети CDN, тогда Ansible сначала выполнит операцию на одном хосте, а затем последовательно будет обрабатывать хосты пакетами по 30 % от общего числа хостов (т. е. 1, 10, 10, 9).

Однократный запуск

Иногда может потребоваться выполнить задачу однократно даже при наличии нескольких хостов. Например, представьте, что у вас есть несколько серверов приложений, запущенных за балансировщиком нагрузки, и вам необходимо осуществить миграцию базы данных, но только на одном из них.

Для этого можно воспользоваться выражением `run_once` и потребовать от Ansible выполнить задачу только один раз:

```
- name: Run the database migrations
  command: /opt/run_migrations
  run_once: true
```

Выражение `run_once` может также пригодиться при использовании `delegate_to: localhost`, если сценарий вовлекает несколько хостов и необходимо выполнить локальную задачу только один раз:

```
- name: Run the task locally, only once
  delegate_to: localhost
  command /opt/my-custom-command
  run_once: true
```

Выбор задач для запуска

Иногда желательно, чтобы Ansible выполнила не все задачи в сценарии, например во время разработки и отладки сценария. Для этого Ansible

поддерживает несколько параметров командной строки, позволяющих управлять выполнением задач.

step

Флаг `--step` заставляет Ansible запрашивать подтверждение на запуск каждой задачи:

```
$ ansible-playbook --step playbook.yml
Perform task: Install packages (y/n/c):
```

В ответ можно потребовать выполнить задачу (`y`), пропустить ее (`n`) или попросить Ansible выполнить оставшуюся часть сценария без дополнительных подтверждений (`c`).

start-at-task

Флаг `--start-at-task taskname` требует от Ansible выполнить сценарий, начиная с указанной задачи. Это удобно, если какая-то задача потерпела неудачу из-за ошибки в одной из предыдущих задач и вы хотите перезапустить сценарий с той задачи, которую только что исправили.

Запуск действий с тегами

Ansible позволяет добавлять теги к задачам, ролям и операциям. Добавив в команду флаг `-t имена_тегов` или `--tags имена_тегов`, можно потребовать от Ansible выполнить только операции, роли и задачи, отмеченные определенными тегами (пример 11.9).

Пример 11.9. Выполнение задач с указанными тегами

```
---
- name: Strategies
  hosts: strategies
  connection: local
  gather_facts: false

  tasks:

    - name: First task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false
      tags:
        - first

    - name: Second task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false
```

```
      tags:
        - second

    - name: Third task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false
      tags:
        - third
...
```

Если запустить этот сценарий с аргументом `--tags first`, то он выведет результаты, как показано в примере 11.10.

Пример 11.10. Запуск только задач с тегом *first*

```
$ ./playbook.yml --tags first
PLAY [Strategies] ***************************************************************
PLAY [Strategies] ***************************************************************
TASK [First task] ***************************************************************
ok: [one]
ok: [two]
ok: [three]
PLAY RECAP **********************************************************************
one   : ok=1  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
three : ok=1  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
two   : ok=1  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

Добавление тегов к задачам, ролям и операциям – это один из способов организовать точное управление выполняемыми действиями в сценариях.

Пропуск действий с тегами

Добавив в команду флаг `--skip-tags имена_тегов`, можно потребовать от Ansible выполнить только операции, роли и задачи, не имеющие указанных тегов.

Стратегии выполнения

Выражение `strategy` на уровне операции дает дополнительную возможность управления выполнением задач на всех хостах .

Мы уже знаем, что по умолчанию используется стратегия линейного выполнения `linear`. Согласно этой стратегии Ansible запускает задачу на всех хостах сразу, ждет ее завершения (успешного или с ошибкой) и затем запускает следующую задачу на всех хостах. Как результат, на выполнение каждой задачи уходит ровно столько времени, сколько для этого требуется самому медленному хосту.

Давайте используем сценарий, представленный в примере 11.9, для демонстрации применения разных стратегий. Мы используем минимальный файл `hosts`, представленный в примере 11.11, содержащий три хоста, для каждого из которых определена переменная `sleep_seconds` со своим значением секунд.

Пример 11.11. Файл *hosts* с тремя хостами и с разными значениями переменной *sleep_seconds*

```
[strategies]
one   sleep_seconds=1
two   sleep_seconds=6
three  sleep_seconds=10
```

linear

Сценарий в примере 11.12 выполняет операцию с тремя задачами локально, как того требует выражение `connection: local`. Каждая задача приостанавливается на время, указанное в переменной `sleep_seconds`.

Пример 11.12. Сценарий для проверки стратегии *linear*

```
---
- name: Strategies
  hosts: strategies
  connection: local
  gather_facts: false

  tasks:

    - name: First task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false

    - name: Second task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false

    - name: Third task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false
...
```

Если запустить этот сценарий со стратегией по умолчанию `linear`, он выведет результаты, показанные в примере 11.13.

Пример 11.13. Результаты выполнения сценария со стратегией *linear*

```
$ ./playbook.yml -l strategies
```

```
PLAY [Strategies] ***************************************************************
TASK [First task] ***************************************************************
Sunday 08 August 2021  16:35:43 +0200 (0:00:00.016)       0:00:00.016 *********
ok: [one]
ok: [two]
ok: [three]
TASK [Second task] **************************************************************
Sunday 08 August 2021  16:35:54 +0200 (0:00:10.357)       0:00:10.373 *********
ok: [one]
ok: [two]
ok: [three]
TASK [Third task] ***************************************************************
Sunday 08 August 2021  16:36:04 +0200 (0:00:10.254)       0:00:20.628 *********
ok: [one]
ok: [two]
ok: [three]
PLAY RECAP **********************************************************************
one   : ok=3  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
three : ok=3  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
two   : ok=3  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
Sunday 08 August 2021  16:36:14 +0200 (0:00:10.256)       0:00:30.884 *********
===============================================================================
First task ------------------------------------------------------------ 10.36s
Third task ------------------------------------------------------------ 10.26s
Second task ----------------------------------------------------------- 10.25s
```

Мы получили уже знакомый нам упорядоченный вывод. Обратите внимание на одинаковый порядок выполнения задач. Это объясняется тем, что хост `one` всегда выполняет задачи быстрее всех (так как для него установлена самая короткая задержка), а хост `three` – медленнее всех (для него установлена самая долгая задержка).

free

В Ansible доступна еще одна стратегия – стратегия `free`. Действуя в соответствии со стратегией `free`, Ansible не ждет результатов выполнения задачи на всех хостах. Вместо этого, как только каждый хост выполнит очередную задачу, ему тут же передается следующая.

В зависимости от быстродействия аппаратуры и задержек в сети один из хостов может справляться с задачами быстрее других, находящихся на другом краю света. Как результат, некоторые хосты могут оказаться уже настроенными, тогда как другие – находиться в середине операции.

Если определить для сценария стратегию `free`, как показано в примере 11.14, его вывод изменится.

Пример 11.14. Выбор стратегии *free* в сценарии

```
---
- name: Strategies
  hosts: strategies
  connection: local
  strategy: free
  gather_facts: false

  tasks:

    - name: First task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false

    - name: Second task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false

    - name: Third task
      command: sleep "{{ sleep_seconds }}"
      changed_when: false
...
```

Обратите внимание, что на этот раз мы выбрали стратегию `free` в третьей строке этой операции. Как показывает вывод в примере 11.15, хост `one` завершил операцию еще до того, как хост `three` успел выполнить первую задачу.

Пример 11.15. Результаты выполнения сценария со стратегией *free*

```
$ ./playbook.yml -l strategies
PLAY [Strategies] **************************************************************
Sunday 08 August 2021  16:40:35 +0200 (0:00:00.020)       0:00:00.020 *********
Sunday 08 August 2021  16:40:35 +0200 (0:00:00.008)       0:00:00.028 *********
Sunday 08 August 2021  16:40:35 +0200 (0:00:00.006)       0:00:00.035 *********
TASK [First task] **************************************************************
ok: [one]
Sunday 08 August 2021  16:40:37 +0200 (0:00:01.342)       0:00:01.377 *********
TASK [Second task] *************************************************************
ok: [one]
Sunday 08 August 2021  16:40:38 +0200 (0:00:01.225)       0:00:02.603 *********
TASK [Third task] **************************************************************
ok: [one]
TASK [First task] **************************************************************
ok: [two]
Sunday 08 August 2021  16:40:42 +0200 (0:00:03.769)       0:00:06.372 *********
ok: [three]
Sunday 08 August 2021  16:40:46 +0200 (0:00:04.004)       0:00:10.377 *********
```

```
TASK [Second task] ************************************************************
ok: [two]
Sunday 08 August 2021  16:40:48 +0200 (0:00:02.229)       0:00:12.606 *********
TASK [Third task] *************************************************************
ok: [two]
TASK [Second task] ************************************************************
ok: [three]
Sunday 08 August 2021  16:40:56 +0200 (0:00:07.998)       0:00:20.604 *********
TASK [Third task] *************************************************************
ok: [three]
PLAY RECAP ********************************************************************
one   : ok=3  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
three : ok=3  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
two   : ok=3  changed=0  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
Sunday 08 August 2021  16:41:06 +0200 (0:00:10.236)       0:00:30.841 *********
===============================================================================
Third task ------------------------------------------------------------ 10.24s
Second task ----------------------------------------------------------- 2.23s
First task ------------------------------------------------------------ 1.34s
```

Чтобы включить в вывод информацию о времени выполнения, мы добавили строку в *ansible.cfg* (обратные вызовы мы обсудим в следующей главе):

```
callback_whitelist = profile_tasks  ;
```

Выражение `callback_whitelist` будет преобразовано в `callback_enabled`.

Улучшенные обработчики

Иногда можно обнаружить, что поведение по умолчанию обработчиков в Ansible не соответствует желаемому. Этот подраздел описывает, как получить более полный контроль над моментом запуска обработчиков.

Обработчики в pre_tasks и post_tasks

Когда мы обсуждали обработчики, то узнали, что они обычно выполняются после всех задач, один раз и только после получения уведомлений. Но не забывайте, что кроме раздела `tasks` существуют еще `pre_tasks` и `post_tasks`.

Каждый раздел `tasks` в сценарии обрабатывается отдельно; любые обработчики, которым были отправлены уведомления из `pre_tasks`, `tasks` или `post_tasks`, выполняются в конце каждого раздела. Как результат, какой-то обработчик может выполниться несколько раз в ходе операции (пример 11.16).

Пример 11.16. *handlers.yml*

```
---
- name: Chapter 9 advanced handlers
  hosts: localhost

  handlers:
    - name: Print message
      command: echo handler executed

  pre_tasks:
    - name: Echo pre tasks
      command: echo pre tasks
      notify: Print message

  tasks:
    - name: Echo tasks
      command: echo tasks
      notify: Print message

  post_tasks:
    - name: Post tasks
      command: echo post tasks
      notify: Print message
```

Если запустить этот сценарий, он выведет результаты, показанные в примере 11.17.

Пример 11.17. Вывод *handlers.yml*

```
$ ./handlers.yml
PLAY [Chapter 9 advanced handlers] *********************************************
TASK [Gathering Facts] *********************************************************
ok: [localhost]
TASK [Echo pre tasks] **********************************************************
changed: [localhost]
RUNNING HANDLER [Print message] ************************************************
changed: [localhost]
TASK [Echo tasks] **************************************************************
changed: [localhost]
RUNNING HANDLER [Print message] ************************************************
changed: [localhost]
TASK [Post tasks] **************************************************************
changed: [localhost]
RUNNING HANDLER [Print message] ************************************************
changed: [localhost]
PLAY RECAP *********************************************************************
localhost : ok=7  changed=6  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

Как видите, обработчики получают уведомления из большего числа секций.

Принудительный запуск обработчиков

Возможно, вам показалось странным, что выше мы написали: *обычно* выполняются после всех задач. *Обычно*, потому что таково поведение по умолчанию. Однако Ansible позволяет управлять моментом выполнения обработчиков с помощью специального модуля `meta`.

В примере 11.18 показана часть операции, в которой используется модуль `meta` с выражением `flush_handlers` в середине. Сделано это, чтобы выполнить *дымовой тест* и убедиться, что обращение к некоторому URL возвращает OK. Но такая проверка не имеет большого смысла до перезапуска служб.

Пример 11.18. Дымовой тест для домашней страницы

```
- name: Install home page
  template:
    src: index.html.j2
    dest: /usr/share/nginx/html/index.html
    mode: '0644'
  notify: Restart nginx

- name: Restart nginx
  meta: flush_handlers

- name: "Test it! https://localhost:8443/index.html"
  delegate_to: localhost
  become: false
  uri:
    url: 'https://localhost:8443/index.html'
    validate_certs: false
    return_content: true
  register: this
  failed_when: "'Running on ' not in this.content"
  tags:
    - test
```

Добавив `flush_handlers`, мы принудительно послали уведомления обработчикам в середине операции.

Метакоманды

Метакоманды могут влиять на внутреннюю работу или состояние Ansible; их можно использовать в любом месте в сценарии. В качестве примера можно привести команду `flush_handlers`, которую мы только что

обсудили, другой пример – команда `refresh_inventory`, повторно читающая реестр (гарантированно не из кеша). Еще пара метакоманд: `clear_facts` и `clear_host_errors`. Также модуль `meta` предлагает команды управления потоком выполнения:

`end_batch` завершает обработку текущего пакета при использовании `serial`;

`end_host` завершает выполнение задач на текущем хосте без генерирования признака ошибки;

`end_play` завершает выполнение операции без генерирования признака ошибки.

Уведомление обработчиков из обработчиков

В файле *handlers* роли *roles/nginx/tasks/main.yml* выполняется проверка конфигурации перед ее перезагрузкой и перезапуском NGINX (пример 11.19). Этот шаг уменьшает вероятность простоя, если конфигурация вдруг окажется некорректной.

Пример 11.19. Проверка конфигурации перед перезапуском службы

```
---
- name: Restart nginx
  debug:
    msg: "checking config first"
  changed_when: true
  notify:
    - Check nginx configuration
    - Restart nginx - after config check

- name: Reload nginx
  debug:
    msg: "checking config first"
  changed_when: true
  notify:
    - Check nginx configuration
    - Reload nginx - after config check

- name: Check nginx configuration
  command: "nginx -t"
  register: result
  changed_when: "result.rc != 0"
  check_mode: false

- name: Restart nginx - after config check
  service:
```

```
      name: nginx
      state: restarted

  - name: Reload nginx - after config check
    service:
      name: nginx
      state: reloaded
```

Выражение `notify` позволяет уведомить перечисленные в нем обработчики; они будут выполняться в указанном порядке.

Выполнение обработчиков по событиям

До появления версии Ansible 2.2 поддерживался только один способ уведомления обработчиков: вызов `notify` с именем обработчика. Этот простой способ подходит для большинства ситуаций.

Прежде чем углубиться в рассуждения, как выполнение обработчиков по событиям может облегчить нам жизнь, рассмотрим короткий пример (пример 11.20).

Пример 11.20. Использование выражения *listen* в обработчиках

```
---
- hosts: mailservers
  tasks:

    - name: Copy postfix config file
      copy:
        src: main.conf
        dest: /etc/postfix/main.cnf
        mode: '0640'
      notify: Postfix config changed

  handlers:
    - name: Restart postfix
      service:
        name: postfix
        state: restarted
      listen: Postfix config changed
...
```

Выражение `listen` определяет то, что мы называем *событием*, появления которого должны дождаться обработчики. Таким способом можно отвязать уведомление, посылаемое задачей, от конкретного имени обработчика. Чтобы уведомить больше обработчиков об одном и том же событии, достаточно просто указать в требуемых обработчиках выражение `listen` с тем же событием.

Область видимости обработчиков ограничивается уровнем операции. Нельзя уведомить обработчики в другой операции ни с использованием, ни без использования выражения `listen`.

Выполнение обработчиков по событиям: случай SSL

Истинная ценность выражения `listen` в обработчиках проявляется при определении ролей или зависимостей между ролями. Один из очевидных случаев, с которыми мы сталкивались, – управление сертификатами SSL для разных служб.

Поскольку мы очень широко используем SSL в наших проектах, имеет смысл создать отдельную роль `ssl`. Это очень простая роль, единственное назначение которой – скопировать сертификаты SSL и ключи на удаленный хост. Для этого в файле *roles/ssl/tasks/main.yml* (см. пример 11.21) определяется несколько задач. Они предназначены для выполнения на хостах с операционной системой Red Hat Linux из-за конкретных путей к файлам, настроенным в переменных *roles/ssl/vars/RedHat.yml* (пример 11.22).

Пример 11.21. Задачи для роли *ssl*

```
---

- name: Include OS specific variables
  include_vars: "{{ ansible_os_family }}.yml"

- name: Copy SSL certs
  copy:
    src: "{{ item }}"
    dest: {{ ssl_certs_path }}/
    owner: root
    group: root
    mode: '0644'
  loop: "{{ ssl_certs }}"

- name: Copy SSL keys
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_keys_path }}/"
    owner: root
    group: root
    mode: '0640'
```

```
    with_items: "{{ ssl_keys }}"
    no_log: true
...
```

Пример 11.22. Переменные для систем на основе Red Hat

```
---
ssl_certs_path: /etc/pki/tls/certs
ssl_keys_path: /etc/pki/tls/private
...
```

В настройках по умолчанию для роли (пример 11.23) мы определили пустые списки сертификатов и ключей SSL, поэтому никакие сертификаты и ключи фактически обрабатываться не будут. У нас есть возможность переопределить эти значения по умолчанию, чтобы заставить роль копировать файлы.

Пример 11.23. Настройки по умолчанию для роли SSL

```
---
ssl_certs: []
ssl_keys: []
...
```

С этого момента у нас появляется возможность использовать роль `ssl` в других ролях в виде *зависимости*, как показано в примере 11.24, где определена роль `nginx` (файл *roles/nginx/meta/main.yml*). Все зависимые роли выполняются до родительской роли. То есть в нашем случае задачи из роли `ssl` выполнятся до задач из роли `nginx`. В результате сертификаты и ключи SSL уже будут находиться на месте и готовы к использованию ролью `nginx` (например, в конфигурации *vhost*).

Пример 11.24. Роль *nginx* зависит от SSL

```
---
dependencies:
  - role: ssl
...
```

Логически зависимости имеют однонаправленный характер: роль `nginx` зависит от роли `ssl`, как показано на рис. 11.1.

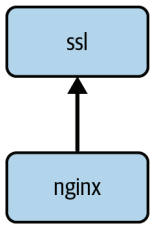


Рис. 11.1. Однонаправленная зависимость

Конечно, роль `nginx` могла бы обрабатывать все аспекты, касающиеся веб-сервера NGINX. Эта роль имеет задачу в файле *roles/nginx/tasks/main.yml* (пример 11.25) для развертывания шаблона с конфигурацией NGINX и перезапускает службу NGINX, посылая уведомление обработчику по его имени.

Пример 11.25. Задачи в роли *nginx*

```
- name: Configure nginx
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
  notify: Restart nginx
```

Последняя строка уведомляет обработчика о необходимости перезапустить веб-сервер NGINX.

Соответствующий обработчик для роли `nginx` определен в файле *roles/nginx/handlers/main.yml*, как показано в примере 11.26.

Пример 11.26. Обработчики для роли *nginx*

```
- name: Restart nginx
  service:
    name: nginx
    state: restarted
```

Так правильно?

Не совсем. Сертификаты SSL иногда требуется менять. И когда происходит замена сертификатов, все службы, использующие их, должны перезапускаться, чтобы взять в работу новые сертификаты.

И как это сделать? Известить обработчик `restart nginx` из роли `ssl` – вы именно это подумали, я угадал? Хорошо, давайте попробуем.

Исправим роль `ssh` в файле *roles/ssl/tasks/main.yml*, добавив в конец задачи копирования сертификатов и ключей выражение `notify` для перезапуска NGINX, как показано в примере 11.27.

Пример 11.27. Добавление выражения *notify* в задачу для перезапуска NGINX

```
---

- name: Include OS specific variables
  include_vars: "{{ ansible_os_family }}.yml"

- name: Copy SSL certs
  copy:
    src: "{{ item }}"
    dest: {{ ssl_certs_path }}/
    owner: root
```

```
    group: root
    mode: '0644'
  with_items: "{{ ssl_certs }}"
  notify: Restart nginx

- name: Copy SSL keys
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_keys_path }}/"
    owner: root
    group: root
    mode: '0644'
  with_items: "{{ ssl_keys }}"
  no_log: true
  notify: Restart nginx
...
```

Отлично, сработало! Но подождите, мы только что добавили новую зависимость в нашу роль `ssl`: зависимость от роли `nginx`, как показано на рис. 11.2.

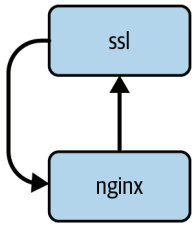


Рис. 11.2. Роль nginx зависит от роли ssl, а роль ssl зависит от роли nginx

И что из этого следует? Если теперь использовать такую роль `ssl` как зависимость в других ролях (таких как `postfix`, `dovecot` или `ldap`), Ansible будет жаловаться на попытку известить неизвестный обработчик, потому что `restart nginx` не будет определен в этих других ролях.

Версия Ansible 1.9 сообщала о попытке известить отсутствующий обработчик. Такое поведение повторно реализовано в версии Ansible 2.2, потому что было замечено как ошибка регресса. Однако его можно изменить с помощью параметра `error_on_missing_handler` в файле *ansible.cfg*, который по умолчанию имеет значение `error_on_missing_handler = true`.

Кроме того, нам могло бы понадобиться добавить в роль `ssl` больше имен обработчиков для уведомления. Однако такое решение очень плохо масштабируется.

Решить эту проблему поможет поддержка выполнения обработчиков по событиям! Вместо уведомления обработчика по имени мы можем послать событие – например, `ssl_certs_changed`, как показано в примере 11.28.

Пример 11.28. Уведомление обработчиков о наступлении события

```
---
- name: Include OS specific variables
  include_vars: "{{ ansible_os_family }}.yml"

- name: Copy SSL certs
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_certs_path }}/"
    owner: root
    group: root
    mode: '0644'
  with_items: "{{ ssl_certs }}"
  notify: ssl_certs_changed

- name: Copy SSL keys
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_keys_path }}/"
    owner: root
    group: root
    mode: '0644'
  with_items: "{{ ssl_keys }}"
  no_log: true
  notify: ssl_certs_changed
...
```

Как отмечалось, Ansible продолжит жаловаться на попытку уведомить неизвестный обработчик, однако, чтобы избавиться от назойливых жалоб, достаточно добавить пустой обработчик в роль `ssl`, как показано в примере 11.29.

Пример 11.29. Добавление пустого обработчика в роль SSL

```
---
- name: SSL certs changed
  debug:
    msg: SSL changed event triggered
  listen: ssl_certs_changed
...
```

Вернемся к нашей роли `nginx`, где мы должны в ответ на событие `ssl_certs_changed` перезапустить службу NGINX. Так как у нас уже есть тре-

буемый обработчик, мы просто добавим в него выражение `listen`, как показано в примере 11.30.

Пример 11.30. Добавление выражения *listen* в существующий обработчик в роли *nginx*

```
---
- name: restart nginx
  debug:
    msg: "checking config first"
  changed_when: true
  notify:
    - check nginx configuration
    - restart nginx - after config check
  listen: Ssl_certs_changed
...
```

Если теперь опять взглянуть на граф зависимостей, то можно заметить, что он изменился, как показано на рис. 11.3. Мы восстановили однонаправленный характер зависимости и получили возможность использовать роль `ssl` в других ролях.

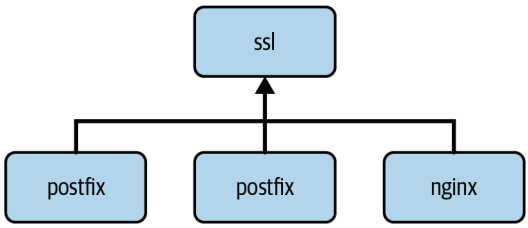


Рис. 11.3. Использование роли ssl в других ролях

И последнее замечание для создателей ролей, размещающих свои роли в Ansible Galaxy: добавляйте обработчики событий и отправку событий в свои роли, если это имеет смысл.

Заключение

Вы сделали это! Теперь вы знаете, как работает Ansible. В оставшейся части книги мы рассмотрим конкретные случаи использования Ansible и способы расширения и защиты автоматизации.

Глава 12

Управление хостами Windows

Ansible часто называют «системой управления конфигурациями на стероидах». По историческим причинам система Ansible имеет тесные связи с Unix и Linux, и свидетельства этому можно наблюдать повсюду, например в именах переменных (таких как `ansible_ssh_host`, `ansible_ssh_connection` и `sudo`). Однако с самого начала Ansible включает поддержку разных механизмов соединения.

Поддержка чужеродных операционных систем, отличных от Linux, таких как Windows, заключалась не только в реализации механизмов подключения к Windows, и в использовании более универсальных имен (например, в переименовании переменной `ansible_ssh_host` в `ansible_host` и выражения `sudo` в `become`).

Богатство библиотеки модулей для Windows уступает богатству библиотеки модулей для Linux. Если вы заинтересованы в использовании Ansible для управления системами Windows, то следите за сообщениями в блоге (**https://oreil.ly/s3zeS**) Джордана Бореана (Jordan Borean), специалиста по Windows в команде Ansible Core. Он создал образ VirtualBox, который мы используем в этой главе.

Подключение к Windows

Добавляя поддержку Windows, разработчики Ansible решили не отходить от своего правила и не стали добавлять специального агента для Windows – и это, как мне кажется, было верным решением. Внедрение нового агента, прослушивающего сеть, открыло бы новые возможности для атак извне. Ansible использует интегрированный механизм удаленного управления Windows Remote Management (WinRM), поддерживающий SOAP-подобный протокол, действующий поверх HTTPS.

WinRM – это наша главнейшая зависимость в Windows, и для взаимодействий с этим механизмом из Python нужно установить соответствующие пакеты в виртуальное окружение на управляющем хосте (для аутентификации в Active Directory требуется Kerberos):

```
$ python3 -mvenv py3
source py3/bin/activate
pip3 install --upgrade pip
pip3 install wheel
pip3 install pywinrm[kerberos]
```

По умолчанию Ansible пытается подключиться к удаленной машине по протоколу SSH, поэтому мы должны явно потребовать сменить механизм подключения. В большинстве случаев желательно включить все хосты с Windows в отдельную группу в реестре. Выбор конкретного имени для такой группы не имеет большого значения, но в последующих примерах сценариев мы будем использовать одно и то же имя группы и для разработки, и для промышленного окружения в отдельных файлах реестра. Причем для разработки используется файл *vagrant.ini*, определяющий среду разработки Vagrant/VirtualBox, описанную в этой главе:

```
[windows]
windows2022 ansible_host=127.0.0.1
```

Мы также добавим в файл реестра переменные с настройками соединения. Если помимо окружений разработки и эксплуатации имеются другие окружения, то имеет смысл установить переменные с настройками соединения в определенном реестре, потому что требования безопасности, такие как проверка сертификата, могут отличаться:

```
[windows:vars]
ansible_user=vagrant
ansible_password=vagrant
ansible_connection=winrm
ansible_port=45986
ansible_winrm_server_cert_validation=ignore
ansible_winrm_scheme=https
ansible_become_method=runas
ansible_become_user=SYSTEM
```

Как отмечалось выше, для подключения к Windows система Ansible использует SOAP-подобный протокол, реализованный поверх HTTP. По умолчанию Ansible пытается установить соединение по защищенному протоколу HTTP (HTTPS) с портом 5986, если в переменной `ansible_port` не указано другое значение.

PowerShell

PowerShell в Microsoft Windows – это мощный интерфейс командной строки и язык сценариев, реализованный на платформе .NET и поддерживающий полный спектр возможностей управления не только локальным окружением, но и удаленными хостами. Все модули Ansible для Windows написаны для PowerShell и на языке PowerShell .

В 2016 году компания Microsoft открыла исходный код PowerShell на условиях лицензии MIT. Исходный код и двоичные пакеты последних версий для macOS, Ubuntu и CentOS можно найти на GitHub (**https://oreil.ly/PbQOt**). На момент написания этих строк в начале 2022 года последней стабильной была версия PowerShell 7.1.3.

Ansible требует, чтобы на удаленных хостах была установлена версия PowerShell не ниже 3. Оболочка PowerShell 3 доступна в Microsoft Windows 7 SP1, Microsoft Windows Server 2008 SP1 и в более поздних версиях. Чтобы узнать номер версии PowerShell, установленной в системе, выполните следующую команду в консоли PowerShell:

```
$PSVersionTable
```

Вы должны увидеть вывод, как показано на рис. 12.1.

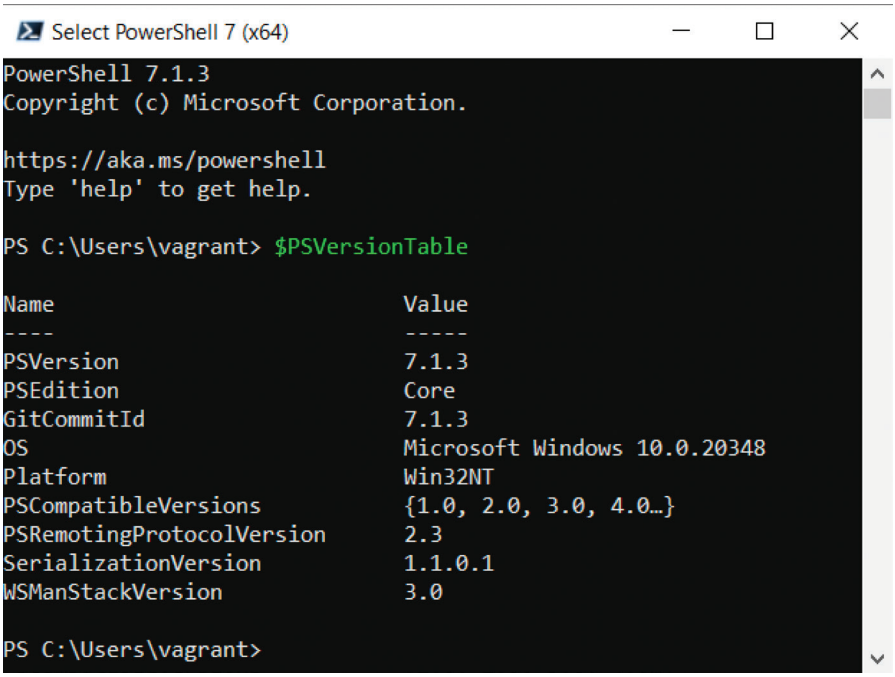


Рис. 12.1. Определение версии PowerShell

На управляющую машину, т. е. на машину, где работает Ansible, требование о наличии PowerShell не распространяется!

Однако в версии 3 имеются ошибки, поэтому, если по каким-то причинам вы не можете использовать более новую версию, вам придется установить последние исправления от Microsoft. Чтобы упростить процесс установки, обновления и настройки PowerShell и Windows, можно использовать сценарий, имеющийся в составе Ansible (**https://oreil.ly/shpIC**). Он прекрасно подходит для настройки окружения разработки, но для применения в промышленном окружении необходимо предпринять дополнительные меры предосторожности.

Установить и запустить его можно командами, представленными в примере 12.1. Сценарий ничего не нарушит, даже если запустить его несколько раз. Но имейте в виду, что для опробования примеров в этой главе запускать этот сценарий *не требуется*.

Пример 12.1. Установка в Windows поддержки Ansible

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
$url = "https://gist.github.com/bbaassssiiee/9b4b4156cba717548650b0e115344337"
$file = "$env:temp\ConfigureRemotingForAnsible.ps1"
(New-Object -TypeName System.Net.WebClient).DownloadFile($url, $file)
powershell.exe -ExecutionPolicy ByPass -File $file
```

Для проверки конфигурации соединений с хостами Windows выполним команду `win_ping`. Похожая на команду `ping` в Linux, она не использует протокол ICMP, а проверяет возможность установки соединения с Ansible:

```
$ ansible windows -i inventory -m win_ping
```

Если в ответ появится сообщение об ошибке, как показано в примере 12.2, необходимо или получить действительный публичный сертификат TLS/SSL, или добавить доверительную цепочку для существующего внутреннего удостоверяющего центра (Certificate Authority, CA).

Пример 12.2. Ошибка, вызванная недействительным сертификатом

```
$ ansible windows -i inventory -m win_ping
windows2022 | UNREACHABLE! => {
  "changed": false,
  "msg": "ssl: HTTPSConnectionPool(host='127.0.0.1', port=45986): Max
retries exceeded with url: /wsman (Caused by
SSLError(SSLCertVerificationError(1, '[SSL: CERTIFICATE_VERIFY_FAILED]
certificate verify failed: self signed certificate (_ssl.c:1131)')))",
  "unreachable": true
}
```

Вы можете запретить проверку сертификатов на свой страх и риск:

```
ansible_winrm_server_cert_validation: ignore
```

Если в ответ появится вывод, как показано в примере 12.3, значит, проверка подключения выполнилась успешно.

Пример 12.3. Результат успешной проверки подключения

```
$ ansible -m win_ping -i hosts windows
windows2022 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Дополнительные сведения о подключении к хостам с использованием WinRM можно найти в онлайн-документации (**https://oreil.ly/ghlAM**).

Модули поддержки Windows

Встроенной поддержка Windows в Ansible позволяет:

- собирать факты о хостах Windows;
- устанавливать и удалять MSI-дистрибутивы;
- включать и отключать функции Windows;
- запускать, останавливать и управлять службами Windows;
- создавать и управлять локальными пользователями и группами;
- управлять пакетами Windows с помощью диспетчера пакетов Chocolatey;
- устанавливать обновления Windows и управлять ими;
- загружать файлы с удаленных сайтов;
- отправлять и запускать любые сценарии PowerShell.

Имена модулей Ansible для Windows начинаются с префикса `win_`, за исключением модуля `setup`, который работает в обеих ОС – Linux и Windows. Вот простой пример создания каталога:

```
- name: Manage tools directory
  win_file:
    path: 'C:/Tools'
    state: directory
```

Краткий обзор всех модулей для Windows и примеры их применения можно найти в онлайн-документации (**https://oreil.ly/bgg0u**).

Наша машина для разработки на Java

Теперь, когда у нас есть хост с Windows, напишем сценарий Ansible, на примере которого покажем, как использовать некоторые модули для Windows. На машину будет установлено программное обеспечение для разработки на Java: не самая последняя версия, но в данном случае для вас важно понять основную идею. Chocolatey – диспетчер пакетов с от-

крытым исходным кодом для Windows. Его команда `choco` может устанавливать и обновлять множество пакетов, доступных в интернете (**https://chocolatey.org/**). Модуль Ansible `win_chocolatey` можно использовать так же, как модуль `package` в Linux, за исключением того, что он также может установить диспетчер пакетов Chocolatey на компьютер с Windows, если он отсутствует:

```
- name: Use Chocolatey
  win_chocolatey:
    name: "chocolatey"
    state: present
```

Широко распространена практика создания ролей для нескольких операционных систем. Вот как выглядит содержимое файла *tasks/main.yml* с такой ролью:

```
---
# файл с задачами для обслуживания нескольких платформ
- name: install software on Linux
  include_tasks: linux.yml
  when:
    - ansible_facts.os_family != 'Windows'
    - ansible_facts.os_family != 'Darwin'
  tags:
    - linux

- name: install software on MacOS
  include_tasks: macos.yml
  when:
    - ansible_facts.os_family == 'Darwin'
  tags:
    - mac

- name: install software on Windows
  include_tasks: windows.yml
  when: ansible_facts.os_family == 'Windows'
  tags:
    - windows
...
```

Создадим простой сценарий, представленный в примере 12.4, который установит необходимое программное обеспечение и выполнит некоторые настройки.

Пример 12.4. Сценарий для Windows

```
---
- name: Setup machine for Java development
  hosts: windows
  gather_facts: false
  vars:
```

```
  pre_tasks:
    - name: Verifying connectivity
      win_ping:
  roles:
    - role: win_config
      tags: config
    - role: win_choco
      tags: choco
    - role: win_vscode
      tags: vscode
    - role: java_developer
      tags: java
    - role: win_updates
      tags: updates
...
```

Сценарий в примере 12.4 не сильно отличается от того, что мы написали бы для Linux.

Добавление локального пользователя

В этой части главы мы посмотрим, как создавать учетные записи пользователей и групп в Windows. Кто-то может подумать, что это давно решенная проблема: достаточно воспользоваться Microsoft Active Directory. Однако хост с Windows может действовать где-то в облаке, а отказ от использования службы каталогов в некоторых случаях может дать дополнительные преимущества.

Сценарий в примере 12.5 создает группу `developers` и учетную запись пользователя, чтобы на конкретном примере продемонстрировать использование модулей. В промышленном окружении имена групп, пользователей могли бы определяться в виде словарей в переменных `group_vars`, а пароли в зашифрованных переменных, но для удобочитаемости мы поместили все это прямо в сценарий.

Пример 12.5. Управление локальными группами и пользователями в Windows

```
- name: Ensure group developers
  win_group:
    name: developers

- name: Ensure ansible user exists
  win_user:
    name: ansible
    password: '%4UJ[nLbQz*:BJ%9gV|x'
    groups: developers
    password_expired: true
    groups_action: add
```

Обратите внимание, что параметру `password_expired` присвоено значение `true`. Это означает, что при следующей попытке входа пользователь должен будет задать новый пароль.

По умолчанию для групп `win_user` выполняет операцию `replace`: пользователь исключается из любых других групп. Мы указали, что по умолчанию должна выполняться операция `add`, чтобы предотвратить исключение пользователей из групп. Поведение по умолчанию можно переопределить для каждого отдельного пользователя.

Функции Windows

В Windows есть функции, которые можно включать и отключать. Получите полный список таких функций, выполнив команду `Get-WindowsFeature` в PowerShell, и составьте список `windows_features_remove` с функциями для отключения:

```
- name: Manage Features
  win_feature:
    name: "{{ item }}"
    state: absent
  loop: "{{ windows_features_remove }}"

- name: Manage IIS Web-Server with sub features and management tools
  win_feature:
    name: Web-Server
    state: present
    include_sub_features: true
    include_management_tools: true
  register: win_iis_feature

- name: Reboot if installing Web-Server feature requires it
  win_reboot:
  when: win_iis_feature.reboot_required
```

После включения/отключения некоторых функций требуется перезагрузка Windows; о наличии такой необходимости говорит значение, возвращаемое модулем `win_feature`.

Установка программного обеспечения с помощью Chocolatey

Чтобы убедиться в возможности поддержки установленного программного обеспечения, создадим два списка. После этого мы сможем использовать этот файл *tasks/main.yml* в роли:

```
- name: Use Chocolatey
  win_chocolatey:
```

```
    name: "chocolatey"
    state: present

- name: Ensure absense of some packages
  win_chocolatey:
    name: "{{ uninstall_choco_packages }}"
    state: absent
    force: true

- name: Ensure other packages are present
  win_chocolatey:
    name: "{{ install_choco_packages }}"
    state: present
```

Эти задачи хорошо справляются с небольшими пакетами, но иногда интернет может работать не так, как хотелось бы. Чтобы сделать установку Visual Studio Code более надежной, мы добавили проверку `win_stat` и повторные попытки `retries`:

```
- name: Check for vscode
  win_stat:
    path: 'C:\Program Files\Microsoft VS Code\Code.exe'
  register: vscode

- name: Install VSCode
  when: not vscode.stat.exists|bool
  win_chocolatey:
    name: "{{ vscode_distribution }}"
    state: present
  register: download_vscode
  until: download_vscode is succeeded
  retries: 10
  delay: 2

- name: Install vscode extensions
  win_chocolatey:
    name: "{{ item }}"
    state: present
  with_items: "{{ vscode_extensions }}"
  retries: 10
  delay: 2
```

Настройки для поддержки Java

Теперь вам должно быть понятно, как устанавливать программное обеспечение с помощью Chocolatey, но в случае со старой доброй Java 8 нужно выполнить некоторые дополнительные настройки:

```
- name: Install Java8
  win_chocolatey:
    name: "{{ jdk_package }}"
    state: present

- name: Set Java_home
  win_environment:
    state: present
    name: JAVA_HOME
    value: "{{ win_java_home }}"
    level: machine

- name: Add Java to path
  win_path:
    elements:
      - "{{ win_java_path }}"
```

Как показано в этом примере, вы можете настроить переменные среды в Windows, а также переменную PATH.

Обновление Windows

Одна из важнейших повседневных задач администратора – установка обновлений безопасности. Это одна из задач, которые администраторы по-настоящему не любят в основном из-за рутины, даже притом, что она важна и необходима, а также потому, что может породить массу проблем, если что-то пойдет не так. Именно поэтому предпочтительнее запретить автоматическую установку обновлений в настройках операционной системы и проверять вновь появившиеся обновления перед их установкой в промышленном окружении.

Ansible поможет автоматизировать эту задачу с помощью простого сценария, представленного в примере 12.6. Сценарий не только устанавливает обновления безопасности, но также перезагружает машину после установки, если необходимо. В заключение он информирует всех пользователей о необходимости выйти перед остановкой системы.

Пример 12.6. Сценарий для установки обновлений безопасности

```
- name: Install critical and security updates
  win_updates:
    category_names:
      - CriticalUpdates
      - SecurityUpdates
    state: installed
  register: update_result

- name: Reboot if required
```

```
win_reboot:
when: update_result.reboot_required
```

Ansible делает управление хостами с Microsoft Windows таким же простым, как управление хостами Linux и Unix.

Заключение

Механизм Microsoft WinRM прекрасно работает, хотя и действует медленнее, чем протокол SSH. Модули Ansible для Windows позволяют выполнять достаточно широкий круг задач и своим удобством мало отличаются от других модулей. Сообщество пользователей Ansible, использующих эту систему для управления Windows, пока еще невелико. Тем не менее Ansible – уже самый простой инструмент для управления парком хостов с разными операционными системами.

Глава 13

Ansible и контейнеры

Проект Docker, появившийся в 2013 году, стремительно захватил мир ИТ. Я не могу вспомнить ни одной другой технологии, которая была бы так быстро подхвачена сообществом. В этой главе рассказывается, как с помощью Ansible создавать образы и развертывать образы контейнеров.

Что такое контейнер?

При виртуализации аппаратного обеспечения программное обеспечение, называемое гипервизором , воссоздает физическую машину целиком, включая виртуальные процессоры, память, а также устройства, такие как диски и сетевые интерфейсы. Виртуализация аппаратного обеспечения – очень гибкая технология, поскольку виртуализации подвергается вся машина целиком. В частности, в качестве гостевой можно установить любую операционную систему, даже в корне отличающуюся от системы-носителя (например, гостевую систему Windows Server 2016 в системе-носителе RedHat Enterprise Linux), и останавливать и запускать виртуальную машину точно так же, как физическую. Однако за эту гибкость приходится платить затратами производительности на виртуализацию аппаратного обеспечения.

Контейнеры иногда называют виртуализацией операционной системы, чтобы подчеркнуть отличие от технологий виртуализации аппаратного обеспечения. При виртуализации операционной системы (контейнеры) гостевые процессы просто изолируются от процессов системы-носителя. Они запускаются на том же ядре, что и система-носитель, но при этом система-носитель обеспечивает полную изоляцию гостевых процессов от ядра.

Контейнеризация – это одна из форм виртуализации . Когда виртуализация используется для запуска процессов в гостевой операционной системе, эти процессы невидимы операционной системе-носителю, выполняющейся на физической аппаратуре. В частности, процессы, запущенные в гостевой операционной системе, не имеют прямого доступа к физическим ресурсам, даже если наделены правами суперпользователя.

Если программное обеспечение поддержки контейнеров, такое как Docker, действует в ОС Linux, гостевые процессы также должны быть процессами Linux. При этом издержки оказываются гораздо ниже, чем при виртуализации аппаратного обеспечения, поскольку запускается только одна операционная система. В частности, процессы в контейнерах запускаются гораздо быстрее, чем на виртуальных машинах.

Docker, Inc. (компания-создатель технологии Docker – я буду использовать «Inc.», чтобы отличить компанию от названия продукта) создала не просто контейнеры, но настоящую платформу, в которой контейнеры играют роль строительных блоков. Контейнеры в Docker – это почти то же самое, что виртуальные машины для гипервизора, такого как VMWare или VirtualBox. Также Docker, Inc. разработала формат образов и Docker API.

Для иллюстрации сравним образ контейнера с образом виртуальной машины. *Образ контейнера* содержит файловую систему с установленной операционной системой, а также некоторые метаданные. Одно существенное отличие в том, что образы контейнеров – многоуровневые. Для создания нового образа Docker берется существующий образ и модифицируется добавлением, изменением или удалением файлов. Новый образ контейнера содержит ссылку на оригинальный образ, а также отличия в файловой системе между оригинальным и новым образами. Благодаря многоуровневой организации образы контейнеров гораздо меньше традиционных образов виртуальных машин, а значит, их легче передать через интернет. Проект Docker поддерживает *реестр* общедоступных образов (**https://hub.docker.com/**).

Также Docker поддерживает API удаленного управления, позволяющий осуществлять взаимодействия со сторонними инструментами. Этот API как раз используют модули `docker_*` в Ansible. С помощью этих модулей можно управлять контейнерами на платформе Docker и программным обеспечением в них.

Kubernetes

Обычно Ansible не используется для управления контейнерами, действующими под управлением Kubernetes, однако при необходимости такое возможно благодаря наличию модуля `k8s` (**https://oreil.ly/yRVOx**). Kubernetes Operator SDK предлагает три других способа управления ресурсами Kubernetes: Go Operators , Helm Charts и Ansible Operators . Наибольшей популярностью в сообществе пользуется Helm Charts. Я не буду вдаваться в подробное описание связки Kubernetes и Ansible. Но для тех, кому интересно, отмечу, что в настоящее время Джефф Герлинг (Jeff Geerling) пишет книгу «Ansible для Kubernetes». Джейсон Добис (Jason Dobies) и Джошуа Вуд (Joshua Wood) в своей книге «Kubernetes Operators» (**https://learning.oreilly.com/library/view/kubernetes-operators/9781492048039/**), вышедшей в издательстве O'Reilly, подробно описывают применение операторов.

Ищущим общедоступное облако для опробования контейнерных технологий Red Hat предлагает облачную платформу на основе OpenShift под названием OpenShift Online (**https://oreil.ly/t6XgM**), а Google – пробную версию своей платформы Google Kubernetes Engine. Обе платформы

имеют открытый исходный код, поэтому, если у вас есть свой парк серверов, вы сможете развернуть на них OpenShift или Kubernetes. Если вы решите развернуть ПО на другой платформе, прочитайте статью в блоге (**https://oreil.ly/b0aKF**) о настройке Vagrant. Также для настройки можно использовать Kubespray (**https://oreil.ly/M2jiC**).

Вы должны знать, что серьезные промышленные системы часто полагаются на использование Kubernetes в сочетании с «голым железом» или виртуальными машинами для организации хранилищ или запуска специализированного программного обеспечения; например, см. документацию по установке wire-server (**https://oreil.ly/rMZYp**). Ansible очень полезна для склеивания кусочков в подобных инфраструктурах.

Жизненный цикл приложения Docker

Вот как выглядит обычный жизненный цикл приложения Docker.

1. Извлечение базового образа контейнера из реестра.
2. Настройка образа контейнера на локальной машине.
3. Отправка образа контейнера с локальной машины в реестр.
4. Извлечение образов контейнеров на удаленный хост из реестра.
5. Запуск контейнеров на удаленном хосте путем передачи им информации о конфигурации.

Обычно образ контейнера создается на локальной машине или в системе непрерывной интеграции, поддерживающей их создание, например Jenkins или GitLab. После создания образ необходимо где-то сохранить, откуда его легко будет загрузить на удаленные хосты.

Реестры

Образы контейнеров обычно хранятся в хранилище, называемом *реестром*. Проект Docker поддерживает реестр *Docker Hub*, в котором могут храниться как публичные, так и частные образы. Существует инструмент командной строки со встроенной поддержкой размещения образов в реестре и загрузки из него. Red Hat поддерживает реестр Quay (**https://quay.io/**). Реестры можно размещать локально с помощью Sonatype Nexus (**https://oreil.ly/lvZ9G**). Некоторые поставщики облачных услуг тоже дают организациям-подписчикам возможность размещать свои частные реестры у них в облаке.

После размещения образа контейнера в реестре можно соединиться с удаленным хостом, загрузить образ контейнера и запустить его. Обратите внимание, что, если попытаться запустить контейнер, образа которого нет на хосте, Docker автоматически загрузит его из реестра.

Поэтому нет необходимости явно использовать команду загрузки образа из реестра.

Ansible и Docker

При использовании Ansible для создания образов Docker и запуска контейнеров на удаленных хостах жизненный цикл приложения будет выглядеть следующим образом.

1. Написание сценариев Ansible для создания образов Docker.
2. Выполнение сценариев для создания образов контейнеров на локальной машине.
3. Передача образов контейнеров с локальной машины в реестр.
4. Написание сценариев Ansible для извлечения образов контейнеров на удаленные хосты и их запуск путем передачи информации о конфигурации.
5. Выполнение сценариев Ansible для запуска контейнеров.

Подключение к демону Docker

Все модули Ansible Docker взаимодействуют с демоном Docker. Если вы работаете в Linux или в macOS и используете поддержку Docker для Mac, все модули должны просто работать без всяких дополнительных параметров.

Если вы работаете в macOS и используете Boot2Docker или Docker Machine, а также когда модуль и демон Docker выполняются на разных машинах, вам может понадобиться передать модулям дополнительную информацию, чтобы они могли связаться с демоном Docker. В табл. 13.1 перечислены параметры, которые можно передавать модулям через аргументы командной строки или через переменные окружения. Дополнительные подробности вы найдете в документации с описанием модуля `docker_container`.

Таблица 13.1. Параметры подключения к демону Docker

Аргумент модуля	Переменная окружения	Значение по умолчанию
`docker_host`	`DOCKER_HOST`	`unix://var/run/docker.sock`
`tls_hostname`	`DOCKER_TLS_HOSTNAME`	`localhost`
`api_version`	`DOCKER_API_VERSION`	`auto`
`cert_path`	`DOCKER_CERT_PATH`	(Нет)
`ssl_version`	`DOCKER_SSL_VERSION`	(Нет)
`tls`	`DOCKER_TLS`	`no`

tls_verify	DOCKER_TLS_VERIFY	no
timeout	DOCKER_TIMEOUT	60 (секунд)

Пример применения: Ghost

В этой главе мы оставим в стороне приложение Mezzanine и возьмем за основу другое приложение – Ghost. Ghost – это платформа блогинга с открытым исходным кодом, напоминающая WordPress. Проект Ghost имеет официальный контейнер Docker, который мы используем в качестве основы.

Вот о чем мы поговорим далее в этой главе:

- запуск контейнера Ghost на локальной машине;
- запуск контейнера Ghost поверх контейнера NGINX с настройкой SSL;
- добавление своего образа NGINX в реестр;
- развертывание контейнеров Ghost и NGINX на удаленной машине.

Запуск контейнера Docker на локальной машине

Модуль `docker_container` запускает и останавливает контейнеры Docker, реализуя некоторые возможности инструмента командной строки `docker`, такие как команды `run`, `kill` и `rm`.

Если предположить, что программное обеспечение Docker уже установлено на локальном компьютере, то следующая команда загрузит образ Ghost из реестра Docker и запустит его. Она отобразит порт 2368 в контейнере в порт 8000 локальной машины, благодаря чему вы сможете обратиться к Ghost по адресу *http://localhost:8000*.

```
$ ansible localhost -m docker_container -a "name=test-ghost image=ghost \
  ports=8000:2368»
```

В первый раз может потребоваться некоторое время на загрузку образа. В случае успеха команда `docker ps` покажет работающий контейнер:

```
$ docker ps --format "table {{.ID }} {{.Image}} {{.Ports}}"
  CONTAINER ID IMAGE PORTS
  ff728315015e ghost 0.0.0.0:8000->2368/tcp
```

Следующая команда остановит и удалит контейнер:

```
$ ansible localhost -m docker_container -a "name=test-ghost state=absent"
```

Модуль `docker_container` поддерживает несколько параметров: практически для всех параметров, поддерживаемых командой `docker`, модуль `docker_container` имеет свои эквивалентные параметры.

Создание образа из Dockerfile

Чтобы создать свой образ контейнера, нужно написать специальный текстовый файл, который называется *Dockerfile*, напоминающий сценарий на языке командной оболочки. Стандартный образ Ghost прекрасно работает сам по себе, но, чтобы обеспечить безопасность доступа, перед ним нужно запустить веб-сервер с настроенной поддержкой TLS.

Проект NGINX поддерживает свой стандартный образ NGINX, но нам нужно настроить его для работы с Ghost и включить в нем поддержку TLS, как мы делали это в главе 7, когда развертывали приложение Mezzanine. В примере 13.1 представлен файл Dockerfile, реализующий все необходимое.

Пример 13.1. *Dockerfile*

```
FROM nginx
RUN rm /etc/nginx/conf.d/default.conf
COPY ghost.conf /etc/nginx/conf.d/ghost.conf
```

В примере 13.2 приводится конфигурация веб-сервера NGINX, обслуживающего Ghost. Главное ее отличие от примера конфигурации для приложения Mezzanine заключается в том, что теперь NGINX взаимодействует с Ghost через TCP-сокет (порт 2368), тогда как для взаимодействий с Mezzanine использовался сокет домена Unix.

Другое отличие – путь к каталогу с файлами сертификатов TLS: */certs*.

Пример 15.2. *ghost.conf*

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    return 301 https://$host$request_uri;
}
server {
    listen 443 ssl;
    client_max_body_size 10M;
    keepalive_timeout    15;
    ssl_certificate      /certs/nginx.crt;
    ssl_certificate_key  /certs/nginx.key;
    ssl_session_cache    shared:SSL:10m;
    ssl_session_timeout  10m;
    ssl_protocols TLSv1.3;
    ssl_ciphers EECDH+AESGCM:EDH+AESGCM;
    ssl_prefer_server_ciphers on;
    location / {
```

```
        proxy_pass          http://ghost:2368;
        proxy_set_header    X-Real-IP $remote_addr;
        proxy_set_header    Host      $http_host;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Как можно заметить в этой конфигурации, веб-сервер NGINX обращается к серверу Ghost, используя имя хоста `ghost`. Развертывая эти контейнеры, вы должны гарантировать это соответствие; иначе контейнер NGINX не сможет обслуживать контейнер Ghost.

Если предположить, что Dockerfile и *nginx.conf* хранятся в каталоге *nginx*, следующая задача создаст образ *ansiblebook/nginx-ghost*. Здесь использован префикс *ansiblebook/*, потому что мы собираемся поместить образ в репозиторий Docker Hub с именем *ansiblebook/nginx-ghost*, но вы должны использовать префикс, соответствующий вашему имени пользователя на сайте Docker (**https://hub.docker.com/**):

```
- name: Create Nginx image
  docker_image:
    build:
      path: ./nginx
    source: build
    name: ansiblebook/nginx-ghost
    state: present
    force_source: "{{ force_source | default(false) }}"
    tag:  "{{ tag | default('latest') }}"
```

Убедиться в успешном выполнении задачи можно с помощью команды `docker images`:

```
$ docker images
REPOSITORY                  TAG       IMAGE ID       CREATED         SIZE
ansiblebook/nginx-ghost     latest    e8d39f3e9e57   6 minutes ago   133MB
ghost                       latest    e8bc5f42fe28   3 days ago      450MB
nginx                       latest    87a94228f133   3 weeks ago     133MB
```

Обратите внимание, что вызов модуля `docker_image` завершится ничем, если образ с таким именем уже существует, даже если содержимое Dockerfile изменилось. Если вы внесли изменения в Dockerfile и хотите пересобрать образ, добавьте параметр `force_source: true`:

```
$ ansible-playbook build.yml -e force_source=true
```

В общем случае предпочтительнее добавлять параметр `tag` с номером версии и увеличивать его для каждой новой сборки. В этом случае модуль `docker_image` будет создавать новые образы без явно заданного па-

раметра `force_source`. По умолчанию используется тег `latest`, но он совершенно не подходит для версионирования образов.

```
$ ansible-playbook build.yml -e tag=v2
```

Отправка образа в реестр Docker

Для отправки образа в Docker Hub мы используем отдельный сценарий, представленный в примере 13.3. Обратите внимание, что модуль `docker_login` должен вызываться для регистрации в реестре до попытки отправить туда образ. Оба модуля – `docker_login` и `docker_image` – по умолчанию используют в качестве реестра репозиторий Docker Hub.

Пример 13.3. *publish.yml*

```
---
- name: Publish image to docker hub
  hosts: localhost
  gather_facts: false

  vars_prompt:
    - name: username
      prompt: Enter Docker Registry username
    - name: password
      prompt: Enter Docker Registry password
      private: true

  tasks:
    - name: Authenticate with repository
      docker_login:
        username: "{{ username }}"
        password: "{{ password }}"
      tags:
        - login

    - name: Push image up
      docker_image:
        name: "ansiblebook/nginx-ghost"
        push: true
        source: local
        state: present
      tags:
        - push
```

Если вы собираетесь использовать другой реестр, определите параметр `registry_url` в `docker_login` и префикс имени образа с именем хоста и номером порта реестра (если реестр использует нестандартный порт

HTTP/HTTPS). В примере 13.4 показано, как следует изменить задачи при использовании реестра *http://reg.example.com*.

Пример 13.4. *publish.yml* для случая использования нестандартного реестра

```
tasks:
  - name: Authenticate with repository
    docker_login:
      registry_url: https://reg.example.com
      username: "{{ username }}"
      password: "{{ password }}"
    tags:
      - login

  - name: Push image up
    docker_image:
      name: reg.example.com/ansiblebook/nginx-ghost
      push: true
      source: local
      state: present
    tags:
      - push
```

Сценарий создания образа тоже необходимо изменить, чтобы отразить в нем новое имя образа: *reg.example.com/ansiblebook/nginx-ghost*.

Управление несколькими контейнерами на локальной машине

Часто бывает нужно запустить несколько контейнеров Docker и связать их вместе. В процессе разработки все такие контейнеры обычно запускаются на локальной машине. Но в промышленном окружении они нередко запускаются на разных машинах.

Для разработки, когда все контейнеры выполняются на одной машине, Docker предоставляет инструмент *Docker Compose*, упрощающий запуск и связывание контейнеров. Для управления контейнерами с помощью инструмента Docker Compose можно использовать модуль `docker_compose`.

В примере 13.5 представлен файл *docker-compose.yml*, который запускает NGINX и Ghost. В данном случае предполагается наличие каталога *./certs* с файлами сертификатов TLS.

Пример 13.5. *docker-compose.yml*

```
version: '2'
services:
  nginx:
```

```
    image: ansiblebook/nginx-ghost
    ports:
      - "8000:80"
      - "8443:443"
    volumes:
      - ${PWD}/certs:/certs
    links:
      - ghost
  ghost:
    image: ghost
```

В примере 13.6 приводится сценарий, который создает файл образа NGINX и самоподписанные сертификаты, а затем запускает службы, описанные в примере 13.5.

Пример 13.6. *ghost.yml*

```
---
- name: Run Ghost locally
  hosts: localhost
  gather_facts: false
  tasks:

    - name: Create Nginx image
      docker_image:
        build:
          path: ./nginx
        source: build
        name: bbaassssiiee/nginx-ghost
        state: present
        force_source: "{{ force_source | default(false) }}"
        tag: "{{ tag | default('v1') }}"

    - name: Create certs
      command: >
        openssl req -new -x509 -nodes
        -out certs/nginx.crt -keyout certs/nginx.key
        -subj '/CN=localhost' -days 365
      args:
        creates: certs/nginx.crt
    - name: Bring up services
      docker_compose:
        project_src: .
        state: present
...
```

Модуль `docker_compose` более интересен разработчикам приложений, потому что, когда дело доходит до развертывания в промышленном

окружении, требования времени выполнения часто диктуют необходимость использования Kubernetes.

Запрос информации о локальном образе

Модуль `docker_image_info` позволяет запросить метаданные, описывающие образ, хранящийся локально. В примере 13.7 показан сценарий, использующий этот модуль для получения информации из образа `ghost` об открытых портах и томах.

Пример 13.7. *image-info.yml*

```
---
- name: Get exposed ports and volumes
  hosts: localhost
  gather_facts: false
  vars:
    image: ghost
  tasks:

    - name: Get image info
      docker_image_info:
        name: ghost
      register: ghost

    - name: Extract ports
      set_fact:
        ports: "{{ ghost.images[0].Config.ExposedPorts.keys() }}"

    - name: We expect only one port to be exposed
      assert:
        that: "ports|length == 1"

    - name: Output exposed port
      debug:
        msg: "Exposed port: {{ ports[0] }}"

    - name: Extract volumes
      set_fact:
        volumes: "{{ ghost.images[0].Config.Volumes.keys() }}"

    - name: Output volumes
      debug:
        msg: "Volume: {{ item }}"
      with_items: "{{ volumes }}"
...
```

Если запустить его, он выведет следующее:

```
$ ansible-playbook image-info.yml
PLAY [Get exposed ports and volumes] ***************************************
TASK [Get image info] ******************************************************
ok: [localhost]
TASK [Extract ports] *******************************************************
ok: [localhost]
TASK [We expect only one port to be exposed] *******************************
ok: [localhost] ==> {
    "changed": false,
    "msg": "All assertions passed"
}
TASK [Output exposed port] *************************************************
ok: [localhost] ==> {
    "msg": "Exposed port: 2368/tcp"
}
TASK [Extract volumes] *****************************************************
ok: [localhost]
TASK [Output volumes] ******************************************************
ok: [localhost] => (item=/var/lib/ghost/content) => {
    "msg": "Volume: /var/lib/ghost/content"
}
```

Используйте модуль `docker_image_info` для вывода важной информации о ваших образах.

Развертывание приложения в контейнере Docker

По умолчанию в качестве базы данных Ghost использует SQLite. В промышленном окружении мы будем использовать базу данных MySQL.

Все приложение мы развернем на двух машинах. На одной (`ghost`) развернем контейнеры Ghost и NGINX. На другой (`mysql`) запустим сервер MySQL, который будет действовать как постоянное хранилище для данных Ghost.

В этом примере предполагается, что где-то, например в *group_vars/all*, определены следующие переменные, определяющие параметры настройки обеих машин:

- `database_name=ghost`;
- `database_user=ghost`;
- `database_password=mysupersecretpassword`.

MySQL

Для настройки машины с MySQL нужно установить пару пакетов (пример 13.8).

Пример 13.8. Сценарий комплектования машины с MySQL

```
- name: Provision database machine
  hosts: mysql
  become: true
  gather_facts: false
  tasks:

    - name: Install packages for mysql
      apt:
        update_cache: true
        cache_valid_time: 3600
        name:
          - mysql-server
          - python3-pip
        state: present

    - name: Install requirements
      pip:
        name: PyMySQL
        state: present
        executable: /usr/bin/pip3
```

Развертывание базы данных Ghost

Чтобы развернуть базу данных Ghost, нужно создать саму базу данных и пользователя базы данных для подключения с другого компьютера. Для этого мы должны настроить адрес привязки сервера MySQL, чтобы он прослушивал сеть, а затем перезапустить его с помощью обработчика (пример 13.9).

Пример 13.9. Развертывание базы данных

```
- name: Deploy database
  hosts: database
  become: true
  gather_facts: false

  handlers:
    - name: Restart Mysql
      systemd:
        name: mysql
        state: restarted
  tasks:

    - name: Listen
      lineinfile:
        path: /etc/mysql/mysql.conf.d/mysqld.cnf
        regexp: '^bind-address'
```

```
      line: 'bind-address           = 0.0.0.0'
      state: present
    notify: Restart Mysql

  - name: Create database
    mysql_db:
      name: "{{ database_name }}"
      state: present
      login_unix_socket: /var/run/mysqld/mysqld.sock

- name: Create database user
    mysql_user:
      name: "{{ database_user }}"
      password: "{{ database_password }}"
      priv: '{{ database_name }}.*:ALL'
      host: '%'
      state: present
      login_unix_socket: /var/run/mysqld/mysqld.sock
```

В этом примере мы настроили сервер MySQL на прослушивание адреса 0.0.0.0 и создали пользователя для подключения с любой машины (не самая безопасная настройка).

Веб-сервер

Развертывание веб-сервера – более сложная задача, потому что требуется развернуть два контейнера: Ghost и NGINX. Кроме того, их нужно связать между собой и вдобавок передать в контейнер Ghost конфигурационную информацию, необходимую для подключения к базе данных MySQL.

Чтобы связать контейнеры NGINX и Ghost, мы используем сети Docker. То есть мы создадим свою сеть Docker, подключим к ней контейнеры, и они смогут взаимодействовать друг с другом, используя имена контейнеров как имена хостов.

Сеть Docker создается просто:

```
- name: Create network
  docker_network:
    name: "{{ net_name }}"
```

Имя сети предпочтительнее хранить в переменной, потому что оно понадобится во всех запускаемых нами контейнерах. В примере 13.10 показан фрагмент сценария, отвечающий за запуск сети.

Пример 13.10. Развертывание Ghost

```
- name: Deploy Ghost
  hosts: ghost
```

```
  become: true
  gather_facts: false

  vars:
    url: "https://{{ inventory_hostname }}"
    database_host: "{{ groups['database'][0] }}"
    data_dir: /data/ghostdata
    certs_dir: /data/certs
    net_name: ghostnet

  tasks:
    - name: Create network
      docker_network:
        name: "{{ net_name }}"
```

Обратите внимание: здесь предполагается наличие группы с именем `database`, которая содержит единственный хост; сценарий использует эту информацию для заполнения переменной `database_host`.

Веб-сервер: Ghost

Нам нужно настроить возможность соединения Ghost с базой данных MySQL, а также предусмотреть запуск в режиме промышленной эксплуатации передачей флага `production` команде `npm start`. Мы также должны записать сгенерированные файлы хранилища в смонтированный том.

Вот часть сценария, которая создает каталог для хранения данных, генерирует конфигурационный файл Ghost из шаблона и запускает контейнер, подключенный к сети `ghostnet` (пример 13.11).

Пример 13.11. Контейнер Ghost

```
- name: Create ghostdata directory
  file:
    path: "{{ data_dir }}"
    state: directory
    mode: '0750'

- name: Start ghost container
  docker_container:
    name: ghost
    image: ghost
    container_default_behavior: compatibility
    network_mode: host
    networks:
      - name: "{{ net_name }}"
    volumes:
      - "{{ data_dir }}:/var/lib/ghost/content"
    env:
```

```
    database__client: mysql
    database__connection__host: "{{ database_host }}"
    database__connection__user: "{{ database_user }}"
    database__connection__password: "{{ database_password }}"
    database__connection__database: "{{ database_name }}"
    url: "https://{{ inventory_hostname }}"
    NODE_ENV: production
```

Обратите внимание, что нам не пришлось объявлять никакие сетевые порты, потому что с контейнером Ghost будет взаимодействовать только контейнер NGINX.

Веб-сервер: NGINX

Для контейнера NGINX была определена своя конфигурация, когда мы создавали образ *ansiblebook/nginx-ghost*: он настроен на подключение к `ghost:2368`.

Теперь нам нужно скопировать сертификаты TLS. Поступим так же, как в предыдущих примерах: сгенерируем самоподписанные сертификаты (пример 13.12).

Пример 13.12. Контейнер NGINX

```
- name: Create certs directory
  file:
    path: "{{ certs_dir }}"
    state: directory
    mode: '0750'

- name: Generate tls certs
  command: >
    openssl req -new -x509 -nodes
    -out "{{ certs_dir }}/nginx.crt"
    -keyout "{{ certs_dir }}/nginx.key"
    -subj "/CN={{ ansible_host }}" -days 90
  args:
    creates: certs/nginx.crt

- name: Start nginx container
  docker_container:
    name: nginx_ghost
    image: bbaassssiiee/nginx-ghost
    container_default_behavior: compatibility
    network_mode: "{{ net_name }}"
    networks:
      - name: "{{ net_name }}"
    pull: true
    ports:
```

```
      - "0.0.0.0:80:80"
      - "0.0.0.0:443:443"
    volumes:
      - "{{ certs_dir }}:/certs"
```

Используйте самоподписанные сертификаты только во время разработки во внутренней сети. Планируя развертывание в промышленном окружении, получите сертификаты, заверенные авторизованным центром.

Удаление контейнеров

Ansible предлагает простой способ остановки и удаления контейнеров, который может пригодиться в процессе разработки и тестирования сценариев развертывания. В примере 13.13 показан сценарий, который очищает хост `ghost`.

Пример 13.13. Удаление контейнера

```
---
- name: Remove all Ghost containers and networks
  hosts: ghost
  become: true
  gather_facts: false
  tasks:

    - name: Remove containers
      docker_container:
        name: "{{ item }}"
        state: absent
        container_default_behavior: compatibility
      loop:
        - nginx_ghost
        - ghost

    - name: Remove network
      docker_network:
        name: ghostnet
        state: absent
```

Модуль `docker_container` имеет логический параметр `cleanup`, который гарантирует удаление контейнера после каждого запуска.

Заключение

Технология Docker ясно продемонстрировала широту своих возможностей. В этой главе мы узнали, как управлять образами, контейнерами и сетями Docker с помощью модулей Ansible.

Глава 14

Обеспечение качества с помощью Molecule

Для разработки роли нужна тестовая инфраструктура. Использование одноразовых контейнеров Docker идеально подходит для тестирования с несколькими дистрибутивами или версиями Linux и избавляет от необходимости выполнять тестирование на машинах, используемых другими разработчиками.

Molecule – это фреймворк тестирования ролей Ansible для Python. Используя его, можно провести тестирование на нескольких экземплярах с разными операционными системами и дистрибутивами. Вы можете использовать пару фреймворков и столько сценариев тестирования, сколько потребуется. Molecule поддерживает различные платформы виртуализации посредством плагина-драйвера. Драйвер – это библиотека для Python, помогающая управлять тестовыми хостами (т. е. создавать и уничтожать их).

Molecule способствует последовательной и планомерной разработке простых и понятных ролей. Исходный код Molecule был открыт в 2015 году, опубликован на GitHub пользователем @retr0h и в настоящее время поддерживается сообществом в рамках проекта Ansible компании Red Hat.

Установка и настройка

Molecule зависит от версии Python 3.6 или выше и Ansible версии 2.8 или выше. В зависимости от операционной системы может потребоваться установить дополнительные пакеты. Ansible не является прямой зависимостью, а вызывается как инструмент командной строки.

Установку Python и необходимых зависимостей в Red Hat можно выполнить командой:

```
# yum install -y gcc python3-pip python3-devel openssl-devel python3-libselinux
```

а в Ubuntu:

```
# apt install -y python3-pip libssl-dev
```

После этого можно установить Molecule с помощью `pip`. Мы рекомендуем устанавливать этот фреймворк в виртуальной среде Python. Важно изолировать Molecule и его зависимости Python от системных пакетов Python. Это может сэкономить время и силы при решении проблем с упаковкой Python.

Настройка драйверов в Molecule

В состав дистрибутива Molecule входит только один драйвер: `delegated`. Если необходимо, чтобы Molecule управлял экземплярами в контейнерах, гипервизорах или в облаке, то следует установить соответствующие плагины драйверов и их зависимости. Некоторые плагины драйверов зависят от `pyyaml>=5.1,<6`.

Драйверы устанавливаются с помощью `pip`, как и другие зависимости Python. В настоящее время зависимости Ansible распространяются в виде коллекций (подробнее о коллекциях рассказывается в следующей главе). Чтобы установить нужную коллекцию, выполните следующую команду:

```
$ ansible-galaxy collection install <имя_коллекции>
```

Molecule можно адаптировать для использования в конкретном облачном окружении, что позволяет создать эфемерную инфраструктуру тестирования.

В табл. 14.1 перечислены драйверы для Molecule и их зависимости.

Таблица 14.1. Драйверы для Molecule

Плагин драйвера	Публичное облако	Частное облако	Контейнеры	Зависимости Python	Коллекция Ansible
`molecule-alicloud`	✓			`ansible_allcloud` `ansible_alicloud_module_utils`	
`molecule-azure`	✓				
`molecule-containers`			✓	`molecule-docker` `molecule-podman`	
`molecule-docker`			✓	`docker`	`community.docker`
`molecule-digitalocean`	✓				
`molecule-ec2`	✓			`boto3`	
`molecule-gce`	✓				`google.cloud` `community.crypto`
`molecule-hetznercloud`	✓				
`molecule-libvirt`					
`molecule-linode`					
`molecule-lxd`			✓		

Плагин драйвера	Публичное облако	Частное облако	Контейнеры	Зависимости Python	Коллекция Ansible
molecule-openstack		✓		openstacksdk	
molecule-podman			✓		containers.podman
molecule-vagrant				python-vagrant	
molecule-vmware		✓		pyvmomi	

Создание роли Ansible

Создать роль можно командой:

```
$ ansible-galaxy role init my_role
```

Она создаст следующие файлы в каталоге *my_role*:

```
my_role/
├── README.md
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

Чтобы инициализировать Molecule в существующей роли или добавить сценарий, выполните команду:

```
$ molecule init scenario -r <имя_роли> --driver-name docker s_name
```

`molecule init` расширяет команду `ansible-galaxy role init`, создавая дерево каталогов для роли с дополнительными файлами для тестирования с помощью Molecule. Следующая команда поможет вам запустить Molecule:

```
$ molecule init role my_new_role --driver-name docker
```

Она создаст следующие файлы в каталоге *my_new_role*:

```
├── README.md
├── defaults
│   └── main.yml
```

```
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── molecule
│   └── default
│       ├── converge.yml
│       ├── molecule.yml
│       └── verify.yml
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

Сценарии Molecule

В примере выше можно заметить подкаталог с именем *default*. Это первый сценарий Molecule, с помощью которого можно использовать команду `molecule test` для проверки синтаксиса, запуска инструментов статического анализа кода – линтеров (linter), запуска сценария Ansible с ролью, повторного запуска для проверки идемпотентности и выполнения дополнительных проверок. Все это происходит с использованием контейнера CentOS 8 в Docker.

Сценарии Molecule можно использовать, например, когда понадобится протестировать Ubuntu или Debian. Сценарии Molecule можно использовать независимо друг от друга со следующим флагом:

```
$ molecule test -s <имя_сценария>
```

Желаемое состояние

Бас часто добавляет сценарий Molecule для локального хоста, когда создает роль, устанавливающую программное обеспечение. Используя команды `molecule converge` (для установки) и `molecule cleanup` (для удаления), Бас может проверить нужные состояния. Содержимое каталога *tasks* в роли может содержать:

- *absent.yml*;
- *main.yml*;
- *present.yml*.

main.yml – это просто точка входа, откуда можно ссылаться на отсутствующие и присутствующие файлы, в зависимости от значения переменной `desired_state`:

```
---
- name: "Desired state is {{ desired_state }}"
  include_tasks: "{{ desired_state }}.yml"
...
```

Настройка сценариев в Molecule

Файл *molecule/s_name/molecule.yml* определяет настройки Molecule и драйвера, используемого в сценарии.

Рассмотрим три примера конфигураций, которые могут вам пригодиться. Минимальный пример (пример 14.1) использует локальную машину (localhost) для тестирования с драйвером `delegated`. Вам нужно лишь убедиться в возможности войти в систему по SSH. Драйвер `delegated` можно использовать в паре с существующим реестром.

Пример 14.1. Драйвер *delegated*

```
---
dependency:
  name: galaxy
  options:
    role-file: requirements.yml
    requirements-file: collections.yml
driver:
  name: delegated
lint: |
  set -e
  yamllint .
  ansible-lint
platforms:
  - name: localhost
provisioner:
  name: ansible
verifier:
  name: ansible
```

Обратите внимание, что Molecule может устанавливать роли и коллекции на этапе `dependency`, как показано в примере 14.1. Если сценарий выполняется локально, то можно настроить параметры так, чтобы игнорировать сертификаты, но не поступайте так при работе с удаленными машинами с применением соответствующих сертификатов.

Управление виртуальными машинами

Molecule отлично работает с контейнерами, но в некоторых случаях, например при работе с машинами Windows, мы предпочитаем использовать виртуальную машину. Специалисты по данным, работающие с Python, часто используют диспетчер пакетов Conda для Python и другие

библиотеки. Чтобы протестировать роль, устанавливающую Miniconda (**https://oreil.ly/YU8KJ**) в различных операционных системах, можно создать сценарий для Windows с отдельным файлом *molecule.yml*.

В примере 14.2 используется драйвер `vagrant`, чтобы запустить виртуальную машину Windows в VirtualBox.

Пример 14.2. Запуск Windows в VirtualBox с помощью Vagrant

```
---
driver:
  name: vagrant
  provider:
    name: virtualbox
lint: |
  set -e
  yamllint .
  ansible-lint
platforms:
  - name: WindowsServer2016
    box: jborean93/WindowsServer2016
    memory: 4069
    cpus: 2
    groups:
      - windows
provisioner:
  name: ansible
  inventory:
    host_vars:
      WindowsServer2016:
        ansible_user: vagrant
        ansible_password: vagrant
        ansible_port: 55986
        ansible_host: 127.0.0.1
        ansible_connection: winrm
        ansible_winrm_scheme: https
        ansible_winrm_server_cert_validation: ignore
verifier:
  name: ansible
```

Образ VirtualBox в этом примере был создан Джорданом Бореаном (Jordan Borean). Он описал процесс создания этого образа с помощью Packer в своем блоге (**https://oreil.ly/CXzzg**).

Управление контейнерами

Molecule может создавать сеть для контейнеров в Docker, что позволяет оценивать настройки кластера. Часто в качестве базы данных, кеша и брокера сообщений используется Redis – хранилище структур данных в

памяти с открытым исходным кодом. Redis поддерживает такие структуры данных, как строки, хеши, списки, множества, отсортированные множества с возможностью запроса диапазона, растровые изображения, гипержурналы, геопространственные индексы и потоки данных. Он отлично подходит для использования в масштабируемых приложениях и в качестве кеша для фактов Ansible. В примере 14.3 показано применение драйвера `docker` для имитации кластера Redis Sentinel, работающего в CentOS 7 (схема кластера показана на рис. 14.1).

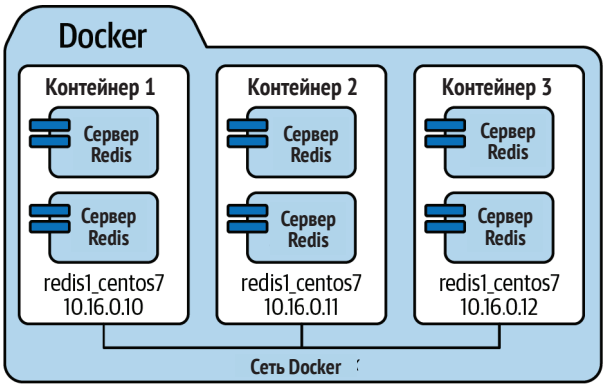


Рис. 14.1. Применение драйвера docker для имитации кластера Redis Sentinel в CentOS 7

В таком кластере выполняется несколько экземпляров Redis, наблюдающих друг за другом; если ведущий экземпляр выйдет из строя, его место займет один из ведомых.

Пример 14.3. Кластер Redis с Docker

```
---
dependency:
  name: galaxy
driver:
  name: docker
lint: |
  set -e
  yamllint .
  ansible-lint
platforms:
  - name: redis1_centos7
    image: milcom/centos7-systemd
    privileged: true
    groups:
      - redis_server
      - redis_sentinel
    docker_networks:
```

```
      - name: 'redis'
        ipam_config:
          - subnet: '10.16.0.0/24'
    networks:
      - name: "redis"
        ipv4_address: '10.16.0.10'
  - name: redis2_centos7
    image: milcom/centos7-systemd
    privileged: true
    groups:
      - redis_server
      - redis_sentinel
    docker_networks:
      - name: 'redis'
        ipam_config:
          - subnet: '10.16.0.0/24'
    networks:
      - name: "redis"
        ipv4_address: '10.16.0.11'
  - name: redis3_centos7
    image: milcom/centos7-systemd
    privileged: true
    groups:
      - redis_server
      - redis_sentinel
    docker_networks:
      - name: 'redis'
        ipam_config:
          - subnet: '10.16.0.0/24'
    networks:
      - name: "redis"
        ipv4_address: '10.16.0.12'
provisioner:
  name: ansible
verifier:
  name: ansible
```

Если выполнить команду `molecule converge` в каталоге роли, то можно воочию увидеть, как протекает создание кластера в Docker, а также установка и настройка программного обеспечения Redis.

Команды Molecule

Molecule – это команда с подкомандами, каждая из которых решает свою задачу в процессе контроля качества. Назначение каждой подкоманды приводится в табл. 14.2.

Таблица 14.2. Подкоманды Molecule

Команда	Назначение
`check`	Выполнить пробный прогон (уничтожение, установка зависимостей, создание, предварительная подготовка, настройка)
`cleanup`	Отменить любые изменения, внесенные во внешние системы на этапах тестирования
`converge`	Настроить экземпляры (установка зависимостей, создание, предварительная подготовка, настройка)
`create`	Запустить экземпляр
`dependency`	Установить зависимости, определяемые ролью
`destroy`	Уничтожить экземпляр
`drivers`	Вывести список драйверов
`idempotence`	Настроить экземпляр и исследовать вывод, чтобы оценить идемпотентность
`init`	Инициализировать новую роль или сценарий Molecule
`lint`	Выполнить статический анализ роли (зависимости, линтеры)
`list`	Вывести информацию о состоянии экземпляров
`login`	Выполнить вход в экземпляр
`matrix`	Вывести матрицу шагов, выполняемых для тестирования экземпляров
`prepare`	Выполнить предварительную подготовку экземпляров, чтобы привести их в определенное начальное состояние
`reset`	Очистить временные папки Molecule
`side-effect`	Выполнить побочные эффекты на экземплярах
`syntax`	Проверить синтаксис роли
`test`	Выполнить матрицу тестов
`verify`	Запустить автоматическое тестирование экземпляров

Обычно мы начинаем с того, что запускаем команду `molecule converge` несколько раз, чтобы привести роль Ansible в нужное состояние. Подкоманда `converge` запускает сценарий Ansible *converge.yml*, созданный командой `molecule init`. Если для роли имеются предварительные условия, например запуск первой другой роли, то имеет смысл создать сценарий Ansible *prepare.yml*, чтобы сэкономить время на этапе разработки. При использовании драйвера `delegated` создайте сценарий Ansible *cleanup.yml*. Вызвать эти дополнительные сценарии Ansible можно командами `molecule prepare` и `molecule cleanup` соответственно.

Статический анализ

Статический анализ (линтинг) осуществляется программами-линтерами. Они анализируют код сценария на наличие потенциальных ошибок, не запуская его. Файлы Ansible можно анализировать на нескольких уровнях: команда `ansible-playbook` имеет параметр `--syntax-check`,

также имеются другие программы, проверяющие форматирование YAML, применение передовых приемов и оформление кода. Molecule может запустить все эти линтеры одновременно. Если вы занимаетесь проверкой качества кода, то вам определенно пригодятся следующие настройки для `molecule lint`:

```
lint: |
  set -e
  yamllint .
  ansible-lint
  ansible-later
```

yamllint

`yamllint` проверяет файлы YAML (**https://oreil.ly/2rhid**) не только на правильность синтаксиса, но и на странности, такие как повторно используемые ключи и некоторые косметические проблемы, такие как длина строк, наличие конечных пробелов, отступы и т. д. `yamllint` помогает создавать файлы YAML оформленные единообразно, что очень полезно, когда вы передаете свой код другим. Обычно мы создаем для этого линтера конфигурационный файл с именем *.yamllint* (пример 14.4).

Пример 14.4. Конфигурационный файл *.yamllint*

```
---
extends: default
rules:
  braces:
    max-spaces-inside: 1
    level: error
  document-start: enable
  document-end: enable
  key-duplicates: enable
  line-length: disable
  new-line-at-end-of-file: enable
  new-lines:
    type: unix
  trailing-spaces: enable
  truthy: enable
...
```

Вы можете включить или отключить эти правила. Мы рекомендуем придерживаться хотя бы настроек `yamllint` по умолчанию.

ansible-lint

Линтер `ansible-lint` был создан Уиллом Темзом (Will Thames) как инструмент статического анализа для Ansible. Он проверяет сценарии

Ansible на возможность их улучшения. В своей работе он использует каталог с правилами (**https://oreil.ly/WtN09**), реализованными в виде сценариев на Python. При желании вы можете создать дополнительный каталог со своими правилами, если понадобится проверить какое-то конкретное поведение.

Запуск проверки сценария Ansible производится командой `ansible-lint` с именем файла сценария в аргументе. Например, чтобы запустить анализ примера 14.5, выполните команду:

```
$ ansible-lint lintme.yml
```

Пример 14.5. *lintme.yml*

```
---
- name: Run ansible-lint with the roles
  hosts: all
  gather_facts: true
  become: yes
  roles:
    - ssh
    - miniconda
    - redis
```

После запуска `ansible-lint` с примером 14.5 она выведет следующие сообщения:

```
WARNING  Listing 3 violation(s) that are fatal
yaml: truthy value should be one of [false, true] (yaml[truthy])
lintme.yml:6

yaml: missing document end "..." (yaml[document-end])
lintme.yml:14

yaml: too many blank lines (3> 0) (yaml[empty-lines])
lintme.yml:14

You can skip specific rules by adding them to your configuration file:
# .config/ansible-lint.yml
skip_list:
  - yaml  # Violations reported by yamllint.

Finished with 3 failure(s), 0 warning(s) on 22 files.
```

Обычно желательно сразу же исправлять все обнаруженные проблемы: это упростит поддержку вашего кода Ansible[1]. Линтер `ansible-lint` поддерживается сообществом Ansible на GitHub.

[1] Как вариант, можно сохранить `skip_list:` в файле с именем *.ansible-lint*.

ansible-later

`ansible-later` – еще один инструмент для проверки ролей и сценариев Ansible. Основой для него послужил другой проект (заброшенный) Уилла Темза (Will Thames) – `ansible-review`. Самая примечательная особенность этого инструмента – он помогает обеспечить соблюдение рекомендаций по оформлению кода (**https://oreil.ly/Yq7nq**). Следование этим рекомендациям поможет сделать роли Ansible более понятными для всех, кто будет сопровождать ваш код, и сократить время устранения неполадок. `ansible-later` может дополнять `yamllint` и `ansible-lint`, если настроить совместимость в файле *.later.yml* в каталоге верхнего уровня (пример 14.6).

Пример 14.6. Конфигурационный файл для *ansible-later* (*.later.yml*)

```
---
ansible:
  # Добавьте имена своих модулей Ansible, которые вы используете.
  custom_modules: []
  # Список логических литералов, совместимых с yamllint (ANSIBLE0014)
  literal-bools:
    - "true"
    - "false"
...
```

Верификаторы

Верификаторы – это инструменты, помогающие подтвердить успешное выполнение роли в сценарии Ansible. Мы знаем, что все модули Ansible тщательно протестированы, тем не менее результат выполнения роли не гарантируется. Хорошей практикой считается автоматизация тестирования для подтверждения результата. Для Molecule доступны три верификатора.

Ansible

`Верификатор по умолчанию.

Goss

Сторонний верификатор, основанный на спецификациях YAML.

TestInfra

Фреймворк тестирования для Python

Верификаторы Goss и TestInfra используют файлы из подкаталога `tests` сценария Molecule, *test_default.yaml* для Goss и *test_default.py* для TestInfra.

Ansible

Для проверки результатов шагов `converge` и `idempotence` можно написать сценарий Ansible с именем *verify.yml*, применяющий модули Ansible, такие как `wait_for`, `package_facts`, `service_facts`, `uri` и `assert`. Запустить проверку можно командой:

```
$ molecule verify
```

Goss

Легко и быстро выполнить проверку сервера можно с помощью Goss – программы на основе YAML (**https://oreil.ly/QTJ4H**), опубликованной Ахмедом Эльсаббахи (Ahmed Elsabbahy). Чтобы увидеть, что можно проверить с помощью Goss, рассмотрим файл *test_sshd.yml* в примере 14.7. Он проверяет, запущена ли служба SSH, запускается ли она после перезагрузки, прослушивает ли она TCP-порт 22, наличие ключей хоста и т. д.

Пример 14.7. Goss-файл для проверки SSH-сервера

```
---
file:
  /etc/ssh/ssh_host_ed25519_key.pub:
    exists: true
    mode: '0644'
    owner: root
    group: root
    filetype: file
    contains:
      - 'ssh-ed25519 '
port:
  tcp:22:
    listening: true
    ip:
      - 0.0.0.0
service:
  sshd:
    enabled: true
    running: true
user:
  sshd:
    exists: true
    uid: 74
    gid: 74
    groups:
      - sshd
```

```
    home: /var/empty/sshd
    shell: /sbin/nologin
group:
  sshd:
    exists: true
process:
  sshd:
    running: true
```

Если запустите Goss и передать этот файл для проверки настроек сервера, то вы получите примерно такой вывод:

```
$ /usr/local/bin/goss -g /tmp/molecule/goss/test_sshd.yml v -f tap
1..18
ok 1 - Group: sshd: exists: matches expectation: [true]
ok 2 - File: /etc/ssh/ssh_host_ed25519_key.pub: exists: matches expectation:
[true]
ok 3 - File: /etc/ssh/ssh_host_ed25519_key.pub: mode: matches expectation:
["0644"]
ok 4 - File: /etc/ssh/ssh_host_ed25519_key.pub: owner: matches expectation:
["root"]
ok 5 - File: /etc/ssh/ssh_host_ed25519_key.pub: group: matches expectation:
["root"]
ok 6 - File: /etc/ssh/ssh_host_ed25519_key.pub: filetype: matches expectation:
["file"]
ok 7 - File: /etc/ssh/ssh_host_ed25519_key.pub: contains: all expectations found:
[ssh-ed25519 ]
ok 8 - Process: sshd: running: matches expectation: [true]
ok 9 - User: sshd: exists: matches expectation: [true]
ok 10 - User: sshd: uid: matches expectation: [74]
ok 11 - User: sshd: gid: matches expectation: [74]
ok 12 - User: sshd: home: matches expectation: ["/var/empty/sshd"]
ok 13 - User: sshd: groups: matches expectation: [["sshd"]]
ok 14 - User: sshd: shell: matches expectation: ["/sbin/nologin"]
ok 15 - Port: tcp:22: listening: matches expectation: [true]
ok 16 - Port: tcp:22: ip: matches expectation: [["0.0.0.0"]]
ok 17 - Service: sshd: enabled: matches expectation: [true]
ok 18 - Service: sshd: running: matches expectation: [true]
```

Чтобы интегрировать Goss с Molecule, установите `molecule-goss` с помощью `pip` и создайте сценарий:

```
$ molecule init scenario -r ssh \
    --driver-name docker \
    --verifier-name goss goss
```

Создайте файлы YAML для Goss в подкаталоге *molecule/goss/tests/* вашей роли. Это один из самых быстрых и эффективных способов внедрить автоматизированное тестирование в операции.

TestInfra

При наличии дополнительных требований для нужд тестирования можно с успехом использовать фреймворки на Python. TestInfra позволяет писать модульные тесты на Python и проверять фактическое состояние серверов, сконфигурированных с помощью Ansible. TestInfra стремится стать Python-эквивалентом фреймворка ServerSpec на основе Ruby, широко используемого для тестирования систем, управляемых с помощью Puppet.

Чтобы использовать фреймворк TestInfra в качестве верификатора, сначала установите его:

```
$ pip install pytest-testinfra
```

и создайте сценарий:

```
$ molecule init scenario -r ssh \
    --driver-name docker \
    --verifier-name testinfra testinfra
```

Чтобы определить набор тестов TestInfra для сервера SSH, создайте файл с именем *molecule/testinfra/tests/test_default.py* и добавьте в него код из примера 14.8. После импорта библиотек он вызывает Molecule, чтобы получить список хостов из реестра в переменную `testinfra_hosts`.

Каждый хост в этом списке проверяется на наличие пакета `openssh-server`, службы `sshd`, файла с ключом хоста *ed25519*, а также соответствующего пользователя и группы.

Пример 14.8. Файл TestInfra для тестирования SSH-сервера

```
import os
import testinfra.utils.ansible_runner

testinfra_hosts = testinfra.utils.ansible_runner.AnsibleRunner(
    os.environ["MOLECULE_INVENTORY_FILE"]
).get_hosts("all")

def test_sshd_is_installed(host):
    sshd = host.package("openssh-server")
    assert sshd.is_installed

def test_sshd_running_and_enabled(host):
    sshd = host.service("sshd")
    assert sshd.is_running
    assert sshd.is_enabled

def test_sshd_config_file(host):
    sshd_config = host.file("/etc/ssh/ssh_host_ed25519_key.pub")
    assert sshd_config.contains("ssh-ed25519 ")
    assert sshd_config.user == "root"
```

```
    assert sshd_config.group == "root"
    assert sshd_config.mode == 0o644

def test_ssh_user(host):
    assert host.user("sshd").exists

def test_ssh_group(host):
    assert host.group("ssh").exists
```

Нетрудно представить, какие широкие возможности проверки серверов открывает поддержка Python. Для экономии времени и сил TestInfra предлагает также готовые тесты для типичных случаев.

Заключение

Если вы активно используете Ansible, то Molecule станет отличным дополнением к вашему набору инструментов. Этот фреймворк помогает разрабатывать согласованные, хорошо оформленные, легко читаемые и понятные роли.

Глава 15

Коллекции

Коллекции – это формат распространения контента Ansible. Типичная коллекция содержит набор связанных вариантов использования. Например, коллекция cisco.ios автоматизирует управление устройствами Cisco iOS. Коллекции контента Ansible (Ansible Content Collections), которые мы будем называть просто коллекциями до конца главы, – это новый стандарт автоматизации распространения, обслуживания и потребления. Коллекции можно рассматривать как формат пакетов для контента Ansible. Комбинируя несколько типов контента Ansible (сценарии, роли, модули и плагины), коллекции значительно повышают гибкость и масштабируемость.

Традиционно создателям модулей приходилось ждать, пока их модули будут включены в очередной выпуск Ansible, или же добавлять их в роли, что усложняло потребление и управление. Теперь, когда проект Ansible отделил выполняемые файлы Ansible от большей части контента, новые версии Ansible могут выходить чаще и независимо от выпусков коллекций.

Отправка модулей в Ansible Collections вместе с ролями и документацией устраняет барьер для входа, благодаря чему создатели могут выпускать свои коллекции по мере появления спроса на них и, соответственно, развертывать и автоматизировать новые функции для существующих или новых продуктов и услуг независимо от выхода новых версий Ansible.

Создать коллекцию и опубликовать ее в Ansible Galaxy или в частном экземпляре Automation Hub может любой желающий. Партнеры Red Hat могут публиковать сертифицированные коллекции в репозитории Red Hat Automation Hub, являющемся частью платформы Red Hat Ansible Automation Platform, с выходом новой версии которой коллекции контента Ansible становятся полностью поддерживаемыми.

Установка коллекций

Коллекции можно получить на веб-сайте Ansible Galaxy и с помощью команды `ansible-galaxy`. По умолчанию команда `ansible-galaxy collection`

`install` пытается установить коллекции с сайта *https://galaxy.ansible.com*, но роли и коллекции также можно хранить в частных репозиториях Git:

```
$ ansible-galaxy collection install мое_пространство_имен.моя_коллекция
```

Чистосердечное признание

До сих пор для простоты Бас использовал имена модулей, состоящие из одного слова и без учета пространств имен. Пространства имен помогают различать владельцев и их коллекции. В сценариях лучше использовать *полные имена коллекций*, потому что в таком случае имена модулей становятся достаточно конкретными, чтобы их можно было найти (попробуйте поискать в Google по имени «group» вместо «ansible.builtin.group»).

Вместо использования простого имени модуля, например:

```
- name: create group members
  group:
    name: members
```

мы используем полное имя в виде *пространство_имен.имя_коллекции.имя_модуля*:

```
- name: create group members
  ansible.builtin.group:
    name: members
```

Для `ansible.builtin` это может показаться излишним, но при использовании других коллекций очень важно избегать конфликтов имен.

Ключевое слово `collections` позволяет определить список коллекций, в которых роль или сценарий должны искать краткие имена модулей и действий. В таком случае можно использовать ключевое слово `collections`, а затем обращаться к модулям и плагинам действий по их кратким именам:

```
# myrole/meta/main.yml
collections:
  - my_namespace.first_collection:version
```

Коллекции можно устанавливать в имеющуюся установку Ansible и переопределять встроенные коллекции устанавливаемыми вами версиями.

Команде `ansible-galaxy` также можно передать файл *requirements.yml* со списком рекомендуемых коллекций и ролей, связанных с безопасностью:

```
$ ansible-galaxy install -r requirements.yml
```

По умолчанию коллекции устанавливаются «глобально» в подкаталог в вашем домашнем каталоге:

$HOME/.ansible/collections/ansible_collections

Если вы решите устанавливать коллекции в другой каталог, настройте его в параметре `collections_paths` в файле *ansible.cfg*. Каталог *collections* в папке с *playbook.yml* – одно из наиболее удобных мест в структуре проекта.

В примере 15.1 показано, как выглядит содержимое файла *requirements.yml*. В нем определено два списка: для ролей и коллекций.

Пример 15.1. *requirements.yml*

```
---
roles:
  - src: leonallen22.ansible_role_keybase
    name: keybase
  - src: https://github.com/dockpack/base_tailscale.git
    name: tailscale
collections:
  - check_point.gaia
  - check_point.mgmt
  - cyberark.conjur
  - cyberark.pas
  - fortinet.fortios
  - ibm.isam
  - junipernetworks.junos
  - paloaltonetworks.panos
...
```

Вывод списка коллекций

Первое, что нужно сделать после установки коллекций, – посмотреть, какие коллекции установлены отдельно, а какие вместе с Ansible:

```
$ ansible-galaxy collection list
```

Вы получите список, содержащий более сотни записей, – в Ansible «батарейки поставляются в комплекте». Чтобы получить список модулей, включенных в коллекцию, выполните:

```
$ ansible-doc -l пространство_имен.имя_коллекции
```

Коллекции Ansible расширяют ваши возможности. Чтобы не путаться в многообразии коллекций, установите только `ansible-core` и коллекции, которые действительно необходимы.

Использование коллекций в сценариях

Коллекции могут включать сценарии, роли, модули и плагины. Если вы используете модули из устанавливаемых вами коллекций, то имеет смысл использовать в сценариях полные имена модулей: например, вместо краткого имени модуля `file` лучше использовать полное имя `ansible.builtin.file`. Кроме того, используя сторонние коллекции, добав-

ляйте ключевое слово `collections` в начало сценария и объявляйте в нем применяемые коллекции (пример 15.2).

Пример 15.2. Объявление коллекций, используемых в сценарии

```
---
- name: Collections playbook
  hosts: all
  collections:
    - our_namespace.her_collection
  tasks:
    - name: Using her module from her collection
      her_module:
        option1: value

    - name: Using her role from her collection
      import_role:
        name: her_role

    - name: Using lookup and filter plug-ins from her collection
      debug:
        msg: '{{ lookup("her_lookup", "param1") | her_filter }}'

    - name: Create directory
      become: true
      become_user: root
      ansible.builtin.file:
        path: /etc/my_software
        state: directory
        mode: '0755'
...
```

Коллекции позволяют расширять «язык» Ansible «новыми словами», и мы можем запускать `ansible-core` только с коллекциями, которые действительно нужны.

Разработка коллекций

Коллекции имеют простую предсказуемую структуру. Утилита командной строки `ansible-galaxy` поддерживает управление коллекциями, предоставляя большую часть тех же возможностей, что всегда использовались для управления ролями. Например, `ansible-galaxy collection init` создаст заготовку новой пользовательской коллекции :

```
$ ansible-galaxy collection init a_namespace.the_bundle
```

Если попробовать создать коллекцию с именем `the_bundle` в пространстве имен `ansiblebook`, то команда создаст следующее дерево каталогов:

```
ansiblebook/
└── the_bundle
    ├── README.md
    ├── docs
    ├── galaxy.yml
    ├── plugins
    │   └── README.md
    └── roles
```

Метаданные коллекции, что хранятся в файле *galaxy.yml* (пример 15.3), включают ссылки на репозиторий, документацию и средство отслеживания проблем. Параметр `tags` используется для поиска в *https://galaxy.ansible.com*, а параметр `build_ignore` – для фильтрации файлов из артефакта.

Пример 15.3. *galaxy.yml*

```
namespace: community
name: postgresql
version: 2.1.3
readme: README.md
authors:
  - Ansible PostgreSQL community
description: null
license_file: COPYING
tags:
  - database
  - postgres
  - postgresql
repository: https://github.com/ansible-collections/community.postgresql
documentation: https://docs.ansible.com/ansible/latest/collections/community/
postgresql
homepage: https://github.com/ansible-collections/community.postgresql
issues: https://github.com/ansible-collections/community.postgresql/issues
build_ignore:
  - .gitignore
  - changelogs/.plugin-cache.yaml
  - '*.tar.gz'
```

Полную информацию о требованиях и процессе распространения вы найдете в руководстве разработчика по распространению коллекций (**https://oreil.ly/zo08v**).

Чтобы передать свою коллекцию в общее пользование, ее можно опубликовать на одном или нескольких серверах распространения, включая Ansible Galaxy, Red Hat Automation Hub (для сертифицированных партнеров Red Hat) и частный Automation Hub (см. главу 23).

Коллекции распространяются в виде архивов, а не в виде исходного кода, как роли в Ansible Galaxy (**https://galaxy.ansible.com/**). Для локального использования отлично подходит формат *tag.gz*. Архив с коллекцией можно создать такой командой:

```
$ ansible-galaxy collection build
```

Для уверенности протестируйте установку локально:

```
$ ansible-galaxy collection install \
    a_namespace-the_bundle-1.0.0.tar.gz \
    -p ./collections
```

После этого можно опубликовать коллекцию:

```
$ ansible-galaxy collection publish path/to/a_namespace-the_bundle-1.0.0.tar.gz
```

Заключение

Коллекции стали большим шагом вперед в развитии проекта Ansible. Представление проекта Ansible с «батарейками в комплекте» со временем оказалось малопригодным для сопровождения тысячами разработчиков. Мы считаем, что наличие надлежащих пространств имен и разделения обязанностей, участие поставщиков в экосистеме Red Hat и свобода инноваций вернут доверие пользователей к Ansible в деле автоматизации поддержки критически важных систем. Имея возможность уверенно управлять своими зависимостями – коллекциями, ролями и библиотеками Python, – вы сможете уверенно автоматизировать свои процессы администрирования.

Глава 16

Создание образов

Создание образов с помощью Packer

Packer – это инструмент, помогающий создавать образы машин для разных платформ из одного источника. С помощью Packer можно создавать как образы виртуальных машин, так и образы контейнеров.

Dockerfile позволяет упаковать приложение в единый образ, который можно развернуть в различных окружениях (но только на контейнерной платформе), поэтому проект Docker использует метафору транспортировочного контейнера. Его удаленный API упрощает автоматизацию программных систем, работающих поверх Docker, но важно помнить о проблемах безопасности такого API.

Стандартный файл Dockerfile отлично подходит для создания простых образов контейнеров. Однако при создании более сложных образов начинает остро ощущаться нехватка возможностей Ansible. К счастью, сценарии Ansible можно использовать как *средство подготовки* описаний образов для HashiCorp Packer (**https://oreil.ly/Fktch**) и избавиться от лишних сложностей.

Рабочие процессы, описанные в этой главе, могут пригодиться, если вы решите отложить выбор места и способа запуска ваших приложений, потому что на основе единственного источника можно создавать образы для разных облачных провайдеров и контейнерных платформ. Кроме того, вы сможете сократить расходы на облачные вычисления за счет комбинирования использования облачных служб и локальной разработки в Vagrant VirtualBox.

Vagrant VirtualBox VM

Для начала рассмотрим создание образа RHEL 8 виртуальной машины Vagrant/VirtualBox с помощью Packer.

Создайте образ, выполнив команду:

```
$ packer build rhel8.pkr.hcl
```

Этот файл определяет переменные для ISO-образа, используемого в Kickstart, свойства виртуальной машины, используемой для создания

образа, и шаги по его наполнению (пример 16.1). Установка вариантов Red Hat Linux основана на Kickstart: при запуске машины команда загрузки запрашивает конфигурацию Kickstart через HTTP и передает ее мастеру установки Red Hat под названием Anaconda.

Пример 16.1. *rhel8.pkr.hcl*

```
variable "iso_url1" {
  type    = string
  default = "file:///Users/Shared/rhel-8.4-x86_64-dvd.iso"
}
variable "iso_url2" {
  type    = string
  default = "https://developers.redhat.com/content-gateway/file/rhel-8.4-x86_64-dvd.iso"
}
variable "iso_checksum" {
  type    = string
  default = "sha256:48f955712454c32718dcde858dea5aca574376a1d7a4b0ed6908ac0b85597811"
}
source "virtualbox-iso" "rhel8" {
  boot_command           = [
   "<tab> text inst.ks=http://{{ .HTTPIP }}:{{ .HTTPPort }}/
 ks.cfg<enter><wait>"
  ]
  boot_wait              = "5s"
  cpus                   = 2
  disk_size              = 65536
  gfx_controller         = "vmsvga"
  gfx_efi_resolution     = "1920x1080"
  gfx_vram_size          = "128"
  guest_os_type          = "RedHat_64"
  guest_additions_mode   = "upload"
  hard_drive_interface   = "sata"
  headless               = true
  http_directory         = "kickstart"
  iso_checksum           = "${var.iso_checksum}"
  iso_urls               = ["${var.iso_url1}", "${var.iso_url2}"]
  memory                 = 4096
  nested_virt            = true
  shutdown_command       = "echo 'vagrant' | sudo -S /sbin/halt -h -p"
  ssh_password           = "vagrant"
  ssh_username           = "root"
  ssh_wait_timeout       = "10000s"
  rtc_time_base          = "UTC"
  virtualbox_version_file= ".vbox_version"
  vrdp_bind_address      = "0.0.0.0"
  vrdp_port_min          = "5900"
```

```
  vrdp_port_max          = "5900"
  vm_name                = "RedHat-EL8"
}
build {
  sources = ["source.virtualbox-iso.rhel8"]
  provisioner "shell" {
    execute_command = "echo 'vagrant' | {{ .Vars }} sudo -S -E bash '{{ .Path }}'"
    scripts         = ["scripts/vagrant.sh", "scripts/cleanup.sh"]
  }
  provisioner "ansible" {
    playbook_file = "./packer-playbook.yml"
  }
  post-processors {
    post-processor "vagrant" {
      keep_input_artifact  = true
      compression_level    = 9
      output               = "output-rhel8/rhel8.box"
      vagrantfile_template = "Vagrantfile.template"
    }
  }
}
```

Когда мастер Anaconda завершит работу, виртуальная машина перезагрузится и Packer приступит к ее наполнению, запуская сценарии, в том числе и *packer-playbook.yml* с помощью сценария наполнения `"ansible"`. Все эти действия будут выполнены на вашей машине.

Отдельные разработчики могут бесплатно зарегистрировать и управлять 16 системами RHEL 8 (**https://oreil.ly/Z8HUl**). Поскольку все действия выполняются на основе подписки, необходимо определить три переменные окружения с логином `RH_USER` и паролем `RH_PASS` для Red Hat и необязательным идентификатором пула `RH_POOL` (**https://oreil.ly/DuyQ8**). Все это можно сделать в командной оболочке перед запуском Packer. В примере 16.2 показан сценарий, который регистрирует виртуальную машину и устанавливает инструменты поддержки контейнеров.

Пример 16.2. *packer-playbook.yml*

```
---
- hosts: all:!localhost
  become: true
  gather_facts: false
  tasks:

    - name: Register RHEL 8
      redhat_subscription:
        state: present
        username: "{{ lookup('env','RH_USER') }}"
```

```
        password: "{{ lookup('env','RH_PASS') }}"
        pool_ids: "{{ lookup('env','RH_POOL') }}"
        syspurpose:
          role: "Red Hat Enterprise Server"
          usage: "Development/Test"
          service_level_agreement: "Self-Support"

    - name: Install packages
      yum:
        name: "{{ item }}"
        state: present
      loop:
        - podman
        - skopeo
...
```

После успешного завершения сборки можно добавить получившийся файл как шаблон для Vagrant/VirtualBox:

```
$ vagrant box add --force --name RedHat-EL8 output-rhel8/rhel8.box
```

В примерах кода для этой главы вы найдете Vagrantfile, с помощью которого можно запустить виртуальную машину с именем `rhel8`, основанную на этом шаблоне:

```
$ vagrant up rhel8
```

После запуска к ней можно подключиться с помощью Remote Desktop с учетными данными пользователя Vagrant:

```
rdp://localhost:5900
```

Запустите Visual Studio Code, чтобы увидеть, что было установлено.

Объединение Packer и Vagrant

Для создания образов с помощью Packer имеет смысл использовать Vagrant. В файле Vagrantfile можно определять прототипы новых функций, которые в конечном итоге будут добавлены в облачные образы. Сценарии Ansible работают на локальной виртуальной машине намного быстрее, чем полный сценарий Packer, что способствует повышению скорости разработки. Packer не останавливается на полпути и в случае сбоя уничтожит все созданные им ресурсы, тогда как, используя Vagrant, можно добавлять новые возможности поэтапно. Следующий файл Vagrantfile запускает виртуальную машину, используя определение образа с именем `"centos/7"`:

```
Vagrant.configure("2") do |config|
  config.vm.box = "centos/7"
  config.vm.box_check_update = true
```

```
  if Vagrant.has_plugin?("vagrant-vbguest")
    config.vbguest.auto_update = false
  end
  config.vm.graceful_halt_timeout=15
  config.ssh.insert_key = false
  config.ssh.forward_agent = true
  config.vm.provider "virtualbox" do |virtualbox|
    virtualbox.gui = false
    virtualbox.customize ["modifyvm", :id, "--memory", 2048]
    virtualbox.customize ["modifyvm", :id, "--vram", "64"]
  end
  config.vm.define :bastion do |host_config|
    host_config.vm.box = "centos/7"
    host_config.vm.hostname = "bastion"
    host_config.vm.network "private_network", ip: "192.168.56.20"
    host_config.vm.network "forwarded_port", id: 'ssh', guest: 22, host: 2220
    host_config.vm.synced_folder ".", "/vagrant", disabled: true
    host_config.vm.provider "virtualbox" do |vb|
      vb.name = "bastion"
      vb.customize ["modifyvm", :id, "--memory", 2048]
      vb.customize ["modifyvm", :id, "--vram", "64"]
    end
  end
  config.vm.provision :ansible do |ansible|
    ansible.compatibility_mode = "2.0"
    # Отключить ограничение по умолчанию для подключения ко всем серверам
    ansible.limit = "all"
    ansible.galaxy_role_file = "ansible/roles/requirements.yml"
    ansible.galaxy_roles_path = "ansible/roles"
    ansible.inventory_path = "ansible/inventories/vagrant.ini"
    ansible.playbook = "ansible/playbook.yml"
    ansible.verbose = ""
  end
end
```

Vagrant может автоматически настраивать многие аспекты Ansible, но вы также можете запускать отдельные секции в сценарии, используя теги, авторизоваться для проверки и т. д.

Облачные образы

Packer способен создавать образы виртуальных машин для всех основных облачных провайдеров (AWS EC2, Azure, Digital Ocean, GCP, Hetzner Cloud, Oracle) и гипервизоров (OpenStack, Hyper-V, Proxmox, VMWare, VirtualBox, QEMU). Он позволяет отложить принятие решения о развертывании приложений и предлагает универсальный интерфейс.

Следующие облачные провайдеры и технологии поддерживают работу с Ansible и Packer:

Alicloud ECS	Amazon EC2	Azure	CloudStack	Digital Ocean
Docker	Google Cloud Platform	Hetzner Cloud	HuaweiCloud	Hyper-V
Kamatera	Linode	LXC	LXD	OpenStack
Oracle	Parallels	ProfitBricks	Proxmox	QEMU
Scaleway	Vagrant	VirtualBox	VMware	Vultr

Google Cloud Platform

Начать работу с Google Cloud Platform (GCP) очень просто. Выполните вход (**https://oreil.ly/4hLD4**), создайте проект в Compute Engine и скопируйте идентификатор проекта (имя с номером в конце). Создайте переменную окружения и присвойте ей этот идентификатор проекта:

```
export GCP_PROJECT_ID=myproject-332421
```

Выберите зону на странице настроек (**https://oreil.ly/zTvzc**) проекта и создайте пару переменных окружения:

```
export CLOUDSDK_COMPUTE_REGION=europe-west4
export CLOUDSDK_COMPUTE_ZONE=europe-west4-b
```

Примеры в папке *ansiblebook/ch16/cloud* основаны на `ansible-roles` в файле *requirements.yml*. Чтобы установить эти роли, выполните команды:

```
cd ansible && ansible-galaxy install -f -p roles -r roles/requirements.yml
```

В примере 16.3 показан файл Packer, в котором определяются переменные для GCP, образ, используемый в качестве основы, имя результирующего образа, свойства виртуальной машины, в которой будет создаваться образ, и этапы наполнения образа дополнительным программным обеспечением. Тип машины, используемый для создания образа, не связан с типом машины, которая будет создана из этого образа. Для создания сложных образов мы используем экземпляры мощных машин – они стоят столько же, но выполняют работу быстрее.

Пример 16.3. *gcp.pkr.hcl*

```
variable "gcp_project_id" {
  type        = string
  default     = "${env("GCP_PROJECT_ID")}"
  description = "Create a project and use the project-id"
}
variable "gcp_region" {
  type        = string
  default     = "${env("CLOUDSDK_COMPUTE_REGION")}"
  description = "https://console.cloud.google.com/compute/settings"
```

```
}
variable "gcp_zone" {
  type        = string
  default     = "${env("CLOUDSDK_COMPUTE_ZONE")}"
  description = "https://console.cloud.google.com/compute/settings"
}
variable "gcp_centos_image" {
  type        = string
  default     = "centos-7-v20211105"
  description = ""
}
variable "image" {
  type        = string
  default     = "centos7"
  description = "Name of the image when created"
}
source "googlecompute" "gcp_image" {
  disk_size     = "30"
  image_family  = "centos-7"
  image_name    = "${var.image}"
  machine_type  = "e2-standard-2"
  project_id    = "${var.gcp_project_id}"
  region        = "${var.gcp_region}"
  source_image  = "${var.gcp_centos_image}"
  ssh_username  = "centos"
  state_timeout = "20m"
  zone          = "${var.gcp_zone}"
}
build {
  sources = ["googlecompute.gcp_image"]
  provisioner "shell" {
    execute_command = "{{ .Vars }} sudo -S -E bash '{{ .Path }}'"
    scripts         = ["scripts/ansible.sh"]
  }
  provisioner "ansible-local" {
    extra_arguments = ["--extra-vars \"image=${var.image}\""]
    playbook_dir    = "./ansible"
    playbook_file   = "ansible/packer.yml"
  }
  provisioner "shell" {
    execute_command = "{{ .Vars }} /usr/bin/sudo -S -E bash '{{ .Path }}'"
    script          = "scripts/cleanup.sh"
  }
}
```

Первый сценарий наполнения (provisioner) `"shell"` устанавливает систему Ansible на виртуальную машину. Затем она используется как сце-

нарий наполнения `"ansible-local"`. Фактически в виртуальную машину CGP выгружается весь каталог, в котором хранится файл Packer, поэтому будьте осторожны и не создавайте образы в том же каталоге.

Azure

Чтобы начать работу с Azure, выполните вход (**https://portal.azure.com/**) и найдите свой идентификатор подписки Subscription ID. Создайте переменную окружения с его значением:

```
export ARM_SUBSCRIPTION_ID=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

Прежде чем создавать образы, добавьте сначала группу ресурсов и учетную запись хранилища. Вам также придется выбрать место (**https://oreil.ly/UOXYU**) для их размещения.

В примере 16.4 показан соответствующий файл Packer для создания образа виртуальной машины. Он аналогичен файлу для GCP, но требует больше деталей и других переменных.

Пример 16.4. *azure.pkr.hcl*

```
variable "arm_subscription_id" {
  type        = string
  default     = "${env("ARM_SUBSCRIPTION_ID")}"
  description = "https://www.packer.io/docs/builders/azure/arm"
}
variable "arm_location" {
  type        = string
  default     = "westeurope"
  description = "https://azure.microsoft.com/en-us/global-infrastructure/geographies/"
}
variable "arm_resource_group" {
  type        = string
  default     = "${env("ARM_RESOURCE_GROUP")}"
  description = "make arm-resourcegroup in Makefile"
}
variable "arm_storage_account" {
  type        = string
  default     = "${env("ARM_STORAGE_ACCOUNT")}"
  description = "make arm-storageaccount in Makefile"
}
variable "image" {
  type        = string
  default     = "centos7"
  description = "Name of the image when created"
}
source "azure-arm" "arm_image" {
  azure_tags = {
```

```
    product = "${var.image}"
  }
  image_offer                       = "CentOS"
  image_publisher                   = "OpenLogic"
  image_sku                         = "7.7"
  location                          = "${var.arm_location}"
  managed_image_name                = "${var.image}"
  managed_image_resource_group_name = "${var.arm_resource_group}"
  os_disk_size_gb                   = "30"
  os_type                           = "Linux"
  subscription_id                   = "${var.arm_subscription_id}"
  vm_size                           = "Standard_D8_v3"
}
build {
  sources = ["source.azure-arm.arm_image"]
  provisioner "shell" {
    execute_command = "{{ .Vars }} sudo -S -E bash '{{ .Path }}'"
    scripts         = ["scripts/ansible.sh"]
  }
  provisioner "ansible-local" {
    extra_arguments = ["--extra-vars \"image=${var.image}\""]
    playbook_dir    = "./ansible"
    playbook_file   = "ansible/packer.yml"
  }
  provisioner "shell" {
    execute_command = "{{ .Vars }} /usr/bin/sudo -S -E bash '{{ .Path }}'"
    script          = "scripts/cleanup.sh"
  }
  provisioner "shell" {
    execute_command = "chmod +x {{ .Path }}; {{ .Vars }} sudo -E sh '{{ .Path }}'"
    inline = [
      "/usr/sbin/waagent -force -deprovision+user",
      "sync"
    ]
    inline_shebang  = "/bin/sh -x"
  }
}
```

В отличие от CGP здесь по завершении наполнения запускается `waagent`. Эта утилита удаляет учетные записи пользователей и ключи SSH из виртуальной машины, чтобы полученный образ можно было безопасно использовать в новом экземпляре виртуальной машины.

Amazon EC2

Чтобы начать работу с EC2 – предложением Amazon инфраструктуры как услуги (Infrastructure as a Service, IaaS), – выполните вход (**https://aws.**

amazon.com/console) и настройте управление идентификацией и доступом (Identity and Access Management). Мы предполагаем, что вы знаете, как использовать переменные окружения `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` и `AWS_REGION`. Более подробную информацию об облачной инфраструктуре Amazon вы найдете в следующей главе.

Файл Packer для Amazon EC2 (пример 16.5) похож на файлы для других поставщиков облачных услуг, но имеет важное отличие – базовый образ для конкретного региона должен быть указан в переменной `aws_centos_image`.

Пример 16.5. *aws.pkr.hcl*

```
variable "aws_region" {
  type        = string
  default     = "${env("AWS_REGION")}"
  description = "https://docs.aws.amazon.com/general/latest/gr/rande.html"
}

variable "aws_centos_image" {
  type        = string
  default     = "ami-0e8286b71b81c3cc1"
  description = "https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html"
}

variable "image" {
  type        = string
  default     = "centos7"
  description = "Name of the image when created"
}

locals { timestamp = regex_replace(timestamp(), "[- TZ:]", "") }

source "amazon-ebs" "aws_image" {
  ami_name      = "${var.image}-${local.timestamp}"
  instance_type = "t2.micro"
  region        = "${var.aws_region}"
  source_ami    = "${var.aws_centos_image}"
  ssh_username  = "centos"
  tags = {
    Name = "${var.image}"
  }
}

build {
  sources = ["source.amazon-ebs.aws_image"]
  provisioner "shell" {
    execute_command = "{{ .Vars }} sudo -S -E bash '{{ .Path }}'"
```

```
    scripts         = ["scripts/ansible.sh"]
  }

  provisioner "ansible-local" {
    extra_arguments = ["--extra-vars \"image=${var.image}\""]
    playbook_dir    = "./ansible"
    playbook_file   = "ansible/playbook.yml"
  }

  provisioner "shell" {
    execute_command = "{{ .Vars }} /usr/bin/sudo -S -E bash '{{ .Path }}'"
    script          = "scripts/cleanup.sh"
  }

}
```

Сценарий Ansible

Образы основаны на CentOS 7 – известном дистрибутиве, который можно использовать как хост-бастион или как VPN:

```
---
- hosts: all:127.0.0.1
  gather_facts: true
  become: true
  vars:
    net_allow:
      - '10.1.0.0/16'
      - '192.168.56.0/24'
  roles:
    - {role: common, tags: common}
    - {role: epel, tags: epel}
    - {role: ansible-auditd, tags: auditd}
    - {role: nettime, tags: nettime}
    - {role: rsyslog, tags: syslog}
    - {role: crontab, tags: crontab}
    - {role: keybase, tags: keybase}
    - {role: gpg_agent, tags: gpg}
    - {role: tailscale, tags: tailscale}
...
```

Виртуальные машины в облаке должны быть защищены, поэтому в сценарии запускается пара ролей для настройки безопасности, аудита и синхронизации времени. Затем настраиваются параметры SSH и устанавливается дополнительное программное обеспечение для шифрования и поддержки VPN.

Образ Docker: GCC 11

В завершение этой главы мы покажем пример применения Packer для создания сложного образа контейнера, включающего GCC. GCC – это компилятор, который используется для сборки ядра Linux и системного программного обеспечения. Практически все дистрибутивы Linux поставляются вместе с компилятором GCC, что позволяет вам компилировать исходный код на C/C++. GCC постоянно развивается, и более новые версии компилятора обычно создают более быстрые выполняемые файлы из того же исходного кода благодаря достижениям в технологии оптимизации. Проще говоря, если для вас важна высокая скорость выполнения программ, используйте самую свежую версию компилятора; и если необходимо, то скомпилируйте GCC 11 самостоятельно, потому что эта версия включена еще не во все дистрибутивы.

Чтобы скомпилировать GCC и использовать его для программирования на C++ в CentOS/RHEL 7, необходимо дополнительно установить некоторые пакеты, инструменты и библиотеки. Например, Boost – широко известный набор библиотек для программирования на C++; CMake – инструмент сборки. Набор инструментов разработчика Red Hat (Red Hat Developer Toolset, DTS) включает множество других инструментов, необходимых разработчикам.

Предположим, вы решили настроить версии и параметры в сценарии Ansible, которому требуются другие роли (Бас опубликовал их в Ansible Galaxy). Вы можете определить эти требования в файле *requirements.yml* в каталоге с именем *roles*:

```
---
- src: dockpack.base_gcc
  name: base_gcc
  version: '1.3.2'
- src: dockpack.compile_gcc
  name: compile_gcc
  version: 'v1.0.5'
- src: dockpack.base_cmake
  name: base_cmake
  version: '1.3.1'
- src: dockpack.base_boost
  name: base_boost
  version: '2.1.9'
- src: dockpack.base_python
  name: base_python
  version: 'v1.1.2'
```

Сценарий определяет переменные и задает порядок установки (пример 16.6). Чтобы скомпилировать исходный код GCC 11, нужен компилятор GCC. Возникает своеобразная проблема курицы и яйца. Мы уста-

новим Developer Toolset 10 из Software Collections (**https://oreil.ly/6EzPZ**) для CentOS 7, чтобы получить последнюю версию GCC, затем установим Python и CMake и после этого скомпилируем GCC. Скомпилировав GCC, мы сможем с его помощью скомпилировать Boost.

Пример 16.6. *докер-playbook.yml*

```
---
- hosts: all:!localhost
  gather_facts: true
  vars:
    # Устанавливать набор ПО Software Collections?
    collections_enabled: true
    # Версия Developer Toolset, которая используется для компиляции
    DTSVER: 10
    # Компилируемая версия компилятора C++
    GCCVER: '11.2.0'
    dependencies_url_signed: false
    # Компилируемая версия Boost
    boost_version: 1.66.0
    boost_cflags: '-fPIC -fno-rtti'
    boost_cxxflags: '-fPIC -fno-rtti'
    boost_properties: "link=static threading=multi runtime-link=shared"
  roles:
    - role: base_python
    - role: base_cmake
    - role: base_gcc
    - role: compile_gcc
    - role: base_boost
...
```

Поведение Packer определяется последовательностью объявлений и команд в файле *gcc.pkr.hcl* (пример 16.7). Они указывают, какие плагины использовать, как настроить каждый из этих плагинов и в каком порядке их запускать.

Пример 16.7. *gcc.pkr.hcl*

```
packer {
  required_plugins {
    docker = {
      version = ">= 0.0.7"
      source = "github.com/hashicorp/docker"
    }
  }
}
source "docker" "gcc" {
  changes     = ["CMD [\"/bin/bash\"]", "ENTRYPOINT [\"\"]"]
```

```
    commit = true
    image  = "centos:7"
    run_command = [
      "-d",
      "-i",
      "-t",
      "--network=host",
      "--entrypoint=/bin/sh",
      "--", "{{ .Image }}"
    ]
  }
  build {
    name    = "docker-gcc"
    sources = [
      "source.docker.gcc"
    ]
    provisioner "shell" {
      inline = ["yum -y install sudo"]
    }
    provisioner "ansible" {
      playbook_file = "./playbooks/docker-playbook.yml"
      galaxy_file   = "./roles/requirements.yml"
    }
    post-processors {
      post-processor "docker-tag" {
          repository =  "localhost/gcc11-centos7"
          tags = ["0.1"]
        }
    }
  }
```

Чтобы создать образ контейнера, запустите сборку Packer:

```
$ packer build gcc.pkr.hcl
```

Имейте в виду, что сборка займет несколько часов.

Заключение

Мы знаем, что создание комплексных образов Docker с помощью файлов Dockerfile может оказаться весьма сложной задачей. Packer и Ansible позволяют четко разделить задачи и по-иному определить действия, выполняемые нами с нашим программным обеспечением. Packer, Vagrant и Ansible – фантастическая комбинация инструментов для создания образов, используемых в облаке или локально. Если вы работаете в крупной организации, то с их помощью сможете создавать образы, служащие основой для других образов.

Глава 17

Облачная инфраструктура

В Ansible есть несколько функций, значительно упрощающих работу с общедоступными и частными облаками. Облако можно рассматривать как многоуровневую платформу, в которой пользователь может создавать ресурсы для запуска программных приложений[1]. Пользователи могут динамически создавать или удалять облачную инфраструктуру, включая вычислительные, сетевые ресурсы и ресурсы хранения, которая называется *инфраструктура как услуга* (Infrastructure as a Service, IaaS).

IaaS-облако – это услуга, позволяющая пользователю создавать новые виртуальные серверы. Все IaaS-облака обладают функцией самообслуживания, т. е. пользователь взаимодействует непосредственно с услугой, не подавая запросов в ИТ-отдел. Большинство IaaS-облаков предлагает пользователю три типа интерфейсов для взаимодействия с системой:

- веб-интерфейс;
- интерфейс командной строки;
- REST API.

В случае с EC2 веб-интерфейс называется «управляющей консолью AWS» (**https://oreil.ly/b443M**), а интерфейс командной строки (неоригинально) – интерфейсом командной строки AWS (**https://oreil.ly/tm9Rx**). Информацию о REST API можно найти на сайте Amazon (**http://amzn.to/1F7g6yA**).

Для создания серверов IaaS-облака обычно используют виртуальные машины, хотя вообще для создания такого облака можно использовать физические, выделенные серверы (т. е. пользователи будут работать непосредственно с аппаратным обеспечением вместо виртуальных машин) или контейнеры.

[1] Национальный институт стандартов и технологий (National Institute of Standards and Technology, NIST) дал отличное определение облачных вычислений в своем документе «The NIST Definition of Cloud Computing» (https://oreil.ly/Y1hnY).

Большинство IaaS-облаков позволяет вам делать большее, нежели запускать и останавливать серверы. В частности, большинство из них позволяет определять хранилища так, чтобы вы могли подключать и отключать диски от своих серверов. Хранилища этого типа обычно называются *блочными хранилищами*. Они также предоставляют возможность определить свою топологию сети, описывающую соединения между вашими серверами, а еще задать правила брандмауэра, ограничивающего доступ к ним.

Следующий уровень в облаке включает конкретные инновации, разработанные поставщиками облачных услуг, и среды выполнения приложений, такие как кластеры контейнеров, серверы приложений, бессерверные окружения, операционные системы и базы данных. Этот уровень называется *платформа как услуга* (Platform as a Service, PaaS). На этом уровне вы управляете своими приложениями и данными, а платформа – всем остальным. PaaS предоставляет свои возможности, являющиеся предметом конкуренции среди поставщиков облачных услуг, тем более что конкуренция за экономическую эффективность в IaaS – это гонка на выживание. Однако наибольший интерес представляет поддержка контейнерной платформы Kubernetes, имеющаяся в любом облаке.

Любое приложение, работающее в облаке, имеет много уровней, но если клиентам облачной услуги виден только один из них, то это – *программное обеспечение как услуга* (Software as a Service, SaaS). Клиенты просто используют программное обеспечение, ничего не зная о том, где физически находятся серверы.

Что подразумевается под подготовкой облачной услуги?

Мы постараемся быть педантичными в определении понятия «подготовка облачной услуги». Для начала приведем пример типичного взаимодействия пользователя с IaaS-облаком

Пользователь
: Мне необходимо пять новых серверов на Ubuntu 20.04, каждый из которых оснащен двумя CPU, 4 Гбайт оперативной памяти и 100 Гбайт дисковой памяти.

Услуга
: Запрос получен. Номер вашего обращения 432789.

Пользователь
: Каков статус обращения 432789?

Услуга
: Ваши серверы готовы к запуску, IP-адреса: *203.0.113.5*, *203.0.113.13*, *203.0.113.49*, *203.0.113.124*, *203.0.113.209*.

Пользователь
: Я закончил работу с серверами, полученными согласно обращению 432789.

Услуга
: Запрос получен, серверы будут удалены.

Подготовка облачной услуги – это процесс создания ресурсов, необходимых для настройки и запуска программного обеспечения.

Профессиональный способ создания ресурсов в облаке – использовать его API, называемый *инфраструктура как код*. Существует несколько универсальных облачных API и уникальные API конкретных поставщиков, а также, как это принято в мире программистов, имеются абстракции, позволяющие комбинировать некоторые из этих API. С их помощью можно создать *декларативную* модель желаемого состояния ресурсов и заставить инструмент сравнивать ее с текущим состоянием и выполнять соответствующие действия; или можно *императивно* запрограммировать действия, необходимые для достижения желаемого состояния. В любом случае выбранный метод должен описывать ресурсы и их свойства. При выборе императивного подхода необходимо знать детали создания программного стека: сеть, подсеть, группа безопасности, сетевой интерфейс, диск, образ виртуальной машины, виртуальная машина. При декларативном подходе достаточно знать только взаимозависимости. HashiCorp Terraform – это декларативный инструмент для подготовки облачной услуги, тогда как система Ansible имеет более императивный характер: она может определять состояние идемпотентным способом. Различия между этими двумя методами становятся особенно очевидными, когда возникает потребность изменить инфраструктуру, а также когда инфраструктура изменяет состояние с помощью других инструментов подготовки.

Насколько просто подготовить другую версию инфраструктуры? Модули Ansible не обязательно должны быть обратимыми, но, приложив некоторые усилия, мы можем сделать наши сценарии Ansible идемпотентными и *обратимыми*, использовав переменную с желаемым состоянием, позволяющей удалять ресурсы:

```
state: "{{ desired_state }}"
```

Но даже имея реализацию шаблона отмены/повтора, в системе Ansible нет состония, которое можно было бы использовать для планирования изменений, как это делает Terraform. Реестры Ansible могут иметь версии с идемпотентными сценариями подготовки желаемого состояния с аналогичным объемом кода из-за длины описаний свойств объекта. Но объем кода Ansible увеличивается, когда для внесения изменений необходимо запросить состояние инфраструктуры.

В состав Ansible входят модули поддержки для многих других облачных служб, включая Microsoft Azure, Alibaba, Cloudscale, Digital Ocean, Google Compute Engine, Hetzner, OracleCloud, IBM Cloud, Rackspace и

Vultr, а также частных облаков, созданных с использованием oVirt, OpenStack, CloudStack, Proxmox и VMWare vSphere.

После установки Ansible большинство возможностей становится доступно в виде связанных коллекций, причем не самых последних версий. При использовании облачной службы на постоянной основе имеет смысл установить для нее более свежую версию коллекции. Если не найдете своего поставщика облачных услуг в табл. 17.1, то загляните в документацию для коллекции community.general (**https://oreil.ly/HHKMk**) – она реализует большое количество функциональных возможностей. В общем случае если поставщик еще не опубликовал коллекцию для Ansible, то установите библиотеку Python для выбранного вами облака.

Таблица 17.1. Коллекции поддержки облачных служб и библиотеки Python

Облако	Коллекция	Библиотека для Python
Amazon Web Services (**https://oreil.ly/1T1Rp**)	amazon.aws	boto3
Alibaba Cloud Compute Services (**https://oreil.ly/9YoAD**)		footmark
Cloudscale.ch (**https://oreil.ly/k3iCE**)	cloudscale_ch.cloud	
CloudStack (**https://oreil.ly/AdPO8**)	ngine_io.cloudstack	cs
Digital Ocean (**https://oreil.ly/Nhbkq**)	community.digitalocean	
Google Cloud (**https://oreil.ly/TqTn9**)	google.cloud	google-auth requests
Hetzner Cloud (**https://oreil.ly/bh4Pw**)	hetzner.hcloud	hcloud-python
IBM Cloud (**https://oreil.ly/R11XU**)	ibm.cloudcollection	
Microsoft Azure (**https://oreil.ly/B4nmQ**)	azure.azcollection	ansible[azure]
Openstack (**https://oreil.ly/VGkRE**)	openstack.cloud	
Oracle Cloud Infrastructure (**https://oreil.ly/Si7nX**)	oracle.oci	oci
Ovirt (**https://www.ovirt.org/**)	ovirt.ovirt	
Packet.net (**https://oreil.ly/8PYcX**)		packet-python
Rackspace (**https://oreil.ly/ycnze**)	openstack.cloud	
Scaleway (**https://oreil.ly/Yf8Of**)	community.general	
Vultr (**https://www.vultr.com/**)	ngine_io.vultr	

В Ansible есть более сотни модулей, имеющих отношение к EC2 и другим возможностям, предлагаемым Amazon Web Services (AWS). Однако в книге не так много места, чтобы охватить все эти возможности, поэтому мы сосредоточимся лишь на самых основных.

Amazon EC2

В этой главе основное внимание уделяется Amazon Elastic Compute Cloud (EC2), потому что это самая популярная облачная служба. Однако многие понятия, описываемые здесь, применимы и к другим облакам, поддерживаемым в Ansible. В Ansible есть два механизма поддержки EC2:

- плагин динамического реестра (динамической инвентаризации) для автоматического заполнения реестра, избавляющий от необходимости вручную составлять список серверов;
- модули, выполняющие действия в EC2, такие как создание новых серверов.

В этой главе мы рассмотрим оба механизма: и плагин динамической инвентаризации EC2, и модули поддержки EC2.

Терминология

В EC2 используется множество разных понятий. Мы планируем пояснять их по мере их появления в тексте, однако три из них хотелось бы объяснить заранее: *экземпляр*, *образ машины Amazon (Amazon Machine Image)* и *теги*.

Экземпляр

В документации EC2 термин *экземпляр* используется для обозначения виртуальной машины, и мы будем придерживаться этой терминологии в данной главе. Имейте в виду, что с точки зрения Ansible экземпляр EC2 – это *хост*.

В документации EC2 (**http://amzn.to/1Fw5S8l**) термины *создание экземпляра* (creating instance), *запуск экземпляра* (launching instance) и *выполнение экземпляра* (running instance) взаимозаменяемы и описывают процесс запуска нового экземпляра. Однако термин *пуск экземпляра* (starting instance) означает нечто иное – пуск экземпляра, который ранее был приостановлен.

Образ машины Amazon

Образ машины Amazon (Amazon Machine Image, AMI) – это образ виртуальной машины с файловой системой, в которую установлена операционная система. Создавая экземпляр в EC2, вы выбираете операционную систему для своего экземпляра, указывая образ AMI, который EC2 будет использовать для создания экземпляра.

Каждый образ AMI имеет строковый идентификатор, называемый *идентификатором AMI* (AMI ID). Он начинается с префикса `ami-`, за кото-

рым следуют шестнадцатеричные символы, например `ami-1234567890abcdef0`. До января 2016 года идентификаторы, назначаемые вновь созданным образам AMI, включали восемь символов после дефиса (например, `ami-1a2b3c4d`). В период с января 2016 года по июнь 2018 года в Amazon прошел процесс изменения идентификаторов всех этих типов ресурсов, и теперь после дефиса в идентификаторах используется 17 символов. В зависимости от времени создания учетной записи у вас могут иметься ресурсы с короткими идентификаторами, но все новые ресурсы этих типов будут получать более длинные идентификаторы.

Теги

EC2 позволяет аннотировать экземпляры (и другие объекты, такие как образы AMI, тома и группы безопасности) с помощью настраиваемых метаданных, которые называются *тегами*. Теги – это просто пары строк ключ/значение. Например, мы могли бы аннотировать экземпляр со следующими тегами:

```
Name=Staging database
env=staging
type=database
```

Если вы когда-либо давали своему экземпляру EC2 имя в консоли управления AWS, то наверняка использовали теги, даже не подозревая об этом. EC2 реализует имена экземпляров как теги; ключ – `Name`, а значение – любое имя, которое вы дали экземпляру. Кроме этого, в теге `Name` нет ничего особенного, и вы также можете настроить консоль управления для отображения значений других тегов.

Теги не обязательно должны быть уникальными, поэтому можно создать сотню экземпляров с одним и тем же тегом. Поскольку модули поддержки Ansible EC2 часто используют теги для идентификации ресурсов и реализации идемпотентности, они будут упоминаться в этой главе несколько раз.

Старайтесь присваивать всем ресурсам в EC2 осмысленные теги, потому что они играют роль своеобразной документации.

Учетные данные пользователя

Выполняя запросы к Amazon EC2, необходимо указывать учетные данные. Перед использованием веб-консоли Amazon вы регистрируетесь и вводите свои логин и пароль для доступа. Однако все компоненты

Ansible, взаимодействующие с EC2, используют программный интерфейс EC2 API. Этот программный интерфейс не предусматривает использования имени пользователя и пароля. Вместо этого используются две строки – *идентификатор ключа доступа* (access key ID) и *секретный ключ доступа* (secret access key).

Обычно эти строки выглядят так:

- идентификатор ключа доступа: `AKIAIOSFODNN7EXAMPLE`;
- секретный ключ доступа: `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`.

Получить эти учетные данные можно в службе *идентификации и управления доступом* (Identity and Access Management, IAM). С ее помощью можно создать нескольких пользователей IAM с разными привилегиями. После создания пользователя для него можно сгенерировать идентификатор ключа и секретный ключ доступа.

Бас рекомендует хранить идентификатор ключа доступа и секретный ключ доступа в переменных окружения `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY` , потому что это позволяет использовать модули EC2 и плагины инвентаризации без сохранения учетных данных в файлах Ansible.

Бас помещает их в файл с именем *.envrc*, применяя шифрование с помощью `ansible-vault`. Этот файл загружается в момент запуска сеанса. Бас пользует Zsh, поэтому определяет переменные в файле *~/.zshrc*. Если вы пользуетесь Bash, то поместите их в файл *~/.bash_profile*. Если вы используете другую командную оболочку, отличную от Bash или Zsh, то, возможно, вы уже знаете, какой файл использовать для определения этих переменных окружения:

```
export ANSIBLE_VAULT_PASSWORD_FILE =~/.apw_exe
$(ansible-vault view ~/.ec2.rc)
```

`ANSIBLE_VAULT_PASSWORD_FILE` – это выполняемый файл, с помощью которого расшифровывается еще один файл с паролем. Бас использует GNU Privacy Guard (GPG), вариант PGP с открытым исходным кодом:

```
#!/bin/sh
exec gpg -q -d ${HOME}/vault_pw.gpg
```

GPG гарантирует хранение конфиденциальных данных в зашифрованном виде: другими словами, пароли к хранилищу нигде в системе не будут храниться в открытом виде. Агент GPG избавляет от необходимости постоянно вводить пароль.

При вызове модулей поддержки EC2 эти строки можно передать как аргументы. Для плагина динамической инвентаризации учетные дан-

ные можно определить в файле *aws_ec2.yml* (обсуждается в следующем разделе). Однако модули EC2 и плагин динамической инвентаризации позволяют передавать учетные данные в переменных окружения. Также можно использовать IAM-роли (**https://oreil.ly/2oll2**), если ваша управляющая машина сама является экземпляром Amazon EC2.

Переменные окружения

Передавать учетные данные EC2 в модули Ansible можно не только через аргументы, но и через переменные окружения. В примере 17.1 показано, как определить такие переменные.

Пример 17.1. Определение переменных окружения с учетными данными EC2

```
# Не забудьте заменить эти значения фактическими учетными данными!
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJatrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_DEFAULT_REGION=us-west-2
```

После настройки переменных окружения с учетными данными можно запускать модули EC2 Ansible на управляющей машине, а также использовать плагины инвентаризации.

Файлы конфигурации

В качестве небезопасной альтернативы переменным окружения можно использовать конфигурационный файл. Как обсуждается в следующем разделе, Ansible использует библиотеку Python Boto3 для поддержки соглашений Boto3 о хранении учетных данных в файле конфигурации Boto. Мы не будем рассматривать этот формат здесь, поэтому за дополнительной информацией обращайтесь к документации по Boto3 (**https://oreil.ly/FtqeK**).

Необходимое условие: библиотека Boto3 для Python

Для использования поддержки EC2 в Ansible необходимо установить на управляющей машине библиотеку Boto3 для Python как системный пакет. Для этого выполните следующую команду[1]:

```
# python3 -m venv --system-site-packages /usr/local
# source /usr/local/bin/activate
(local) # pip3 install boto3
```

Если у вас имеются работающие экземпляры EC2, попробуйте проверить правильность установки Boto3 и корректность учетных данных с помощью командной строки Python, как показано в примере 17.2.

[1] Для установки пакета может потребоваться использовать команду `sudo` или активировать виртуальное окружение, в зависимости от того, как была установлена система Ansible.

Пример 17.2. Тестирование Boto3 и учетных данных

```
$ python3
Python 3.6.8 (default, Sep  9 2021, 07:49:02)
[GCC 8.5.0 20210514 (Red Hat 8.5.0-3)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto3
>>> ec2 = boto3.client("ec2")
>>> regions = [region["RegionName"] for region in ec2.describe_regions()["Regions"]]
>>> for r in regions:
...     print(f"  - {r}")
...
  - eu-north-1
  - ap-south-1
  - eu-west-3
  - eu-west-2
  - eu-west-1
  - ap-northeast-3
  - ap-northeast-2
  - ap-northeast-1
  - sa-east-1
  - ca-central-1
  - ap-southeast-1
  - ap-southeast-2
  - eu-central-1
  - us-east-1
  - us-east-2
  - us-west-1
  - us-west-2
>>>
```

Исследуя модули, входящие в состав Ansible, можно наткнуться на устаревшие модули, требующие библиотеку Boto для Python 2, например модуль `ec2`, поддерживаемый командой Ansible Core Team (не Amazon):

```
fatal: [localhost]: FAILED! => changed=false
  msg: boto required for this module
```

В таких случаях следует проверить сценарий Ansible и убедиться, что в нем используются полные имена модулей с префиксом `amazon.aws`.

Динамическая инвентаризация

При работе с серверами в EC2 у вас едва ли появится желание поддерживать свою копию реестра Ansible, поскольку она будет устаревать по мере появления новых серверов и удаления старых. Гораздо проще отслеживать серверы EC2, используя плагин динамической инвентаризации, позволяющий получить информацию о хостах непосредственно из EC2.

Этот плагин является частью коллекции `amazon.aws` (версия 2.2.0 [**https://oreil.ly/OpS3x**]). Возможно, у вас уже установлена эта коллекция, если вы установили пакет Ansible. Чтобы проверить, какая версия установлена, выполните команду:

```
$ ansible-galaxy collection list|grep amazon.aws
```

А чтобы установить последнюю версию коллекции – команду:

```
$ ansible-galaxy collection install amazon.aws
```

Раньше у нас был файл *playbooks/inventory/hosts*, который играл роль реестра. Теперь мы будем использовать каталог *playbooks/inventory* и поместим в него файл *aws_ec2.yml*.

В примере 17.3 показано, как организовать динамическую инвентаризацию.

Пример 17.3. Динамическая инвентаризация EC2

```
---
# Минимальный пример использования переменных окружения
# Извлекает все хосты в eu-central-1
plugin: amazon.aws.aws_ec2
regions:
  - eu-north-1
  - ap-south-1
  - eu-west-1
  - ap-northeast-1
  - sa-east-1
  - ca-central-1
  - ap-southeast-1
  - eu-central-1
  - us-east-1
  - us-west-1
# Игнорировать ошибки 403
strict_permissions: false
...
```

Если вы определили переменные окружения, как описывалось в предыдущем разделе, у вас должно получиться проверить работоспособность сценария:

```
$ ansible-inventory --list|jq -r .aws_ec2
```

Сценарий должен вывести информацию о ваших экземплярах EC2, как показано ниже (конкретное содержимое списка у вас будет отличаться):

```
{
  "hosts": [
```

```
      "ec2-203-0-113-75.eu-central-1.compute.amazonaws.com"
    ]
}
```

Кеширование реестра

Когда Ansible использует плагин динамической инвентаризации EC2, он (плагин) должен послать запросы одной или нескольким конечным точкам EC2 для получения информации. Поскольку все это требует времени, при первом запуске плагин кеширует информацию локально, а при последующих вызовах использует кешированные данные, пока не истечет время действия кеша.

Такое поведение можно изменить, отредактировав параметры настройки в конфигурационном файле *ansible.cfg*. По умолчанию время действия кеша составляет 300 с (5 мин). Если кеш должен сохраняться в течение часа, то установите значение, равное 3600, как показано в примере 17.4.

Пример 17.4. *ansible.cfg*

```
[defaults]
fact_caching = jsonfile
fact_caching_connection = /tmp/ansible_fact_cache
fact_caching_timeout = 3600

[inventory]
cache = true
cache_plugin = jsonfile
cache_timeout = 3600
```

С такими настройками в течение следующего часа реестр будет извлекаться быстрее. Ansible кеширует реестр в кеше фактов. Чтобы убедиться, что кеш создан, выполните команду:

```
$ ls /tmp/ansible_fact_cache/
ansible_inventory_amazon.aws.aws_ec2_6b737s_3206c
```

При создании или удалении экземпляров плагин динамической инвентаризации EC2 не будет отражать эти изменения, кроме случаев, когда срок действия кеша истек или кеш был обновлен принудительно.

Другие параметры настройки

Файл *aws_ec2.yml* содержит несколько параметров, управляющих поведением плагина динамической инвентаризации. Поскольку сам файл

снабжен подробными комментариями (**https://oreil.ly/FGx2h**), мы не будем рассказывать об этих параметрах в деталях.

Определение динамических групп с помощью тегов

Напомним, что плагин динамической инвентаризации создает группы на основании таких данных, как тип экземпляра, группа безопасности, пара ключей и теги. Теги в EC2 являются наиболее удобным способом создания групп, поскольку их можно определить каким угодно способом.

При помощи плагина инвентаризации можно определить дополнительные метаданные для реестра, возвращаемого AWS. Например, можно использовать `keyed_groups` для создания групп согласно тегам экземпляров:

```
plugin: aws_ec2
keyed_groups:
  - prefix: tag
    key: tags
```

Ansible автоматически создаст группу с именем `tag_type_web`, содержащую все серверы с тегом `type=web`.

EC2 позволяет присваивать экземплярам по нескольку тегов. Например, при наличии отдельных окружений для тестирования и эксплуатации можно присвоить промышленным веб-серверам тег:

```
env=production
type=web
```

После этого на промышленные машины можно ссылаться с помощью `tag_env_production`, а на веб-серверы – с помощью `tag_type_web`. При необходимости сослаться на промышленные веб-серверы можно использовать перекрестный синтаксис Ansible:

```
hosts: tag_env_production:&tag_type_web
```

Присваивание тегов имеющимся ресурсам

В идеальном случае присваивание тегов экземплярам EC2 происходит в момент их создания. Однако если Ansible устанавливается для управления уже существующими экземплярами EC2, у вас наверняка будет иметься некоторое их количество, которым было бы желательно присвоить теги. В Ansible имеется модуль `ec2_tag`, позволяющий присвоить теги имеющимся экземплярам.

Например, чтобы присвоить экземплярам теги `env=prodution` и `type=web`, можно использовать простой сценарий, представленный в примере 17.5.

Пример 17.5. Присваивание тегов EC2 существующим экземплярам

```
---
- name: Add tags to existing instances
  hosts: localhost
  vars:
    web_production:
      - i-1234567890abcdef0
      - i-1234567890abcdef1
    web_staging:
      - i-abcdef01234567890
      - i-33333333333333333
  tasks:
    - name: Tag production webservers
      ec2_tag:
        resource: "{{ item }}"
        region: "{{ lookup('env','AWS_REGION') }}"
      args:
        tags: {type: web, env: production}
      loop: "{{ web_production }}"

    - name: Tag staging webservers
      ec2_tag:
        resource: "{{ item }}"
        region: "{{ lookup('env','AWS_REGION') }}"
      args:
        tags: {type: web, env: staging}
      loop: "{{ web_staging }}"
...
```

В этом примере используется синтаксис YAML встраивания словарей, когда теги (`{ type: web, env: production}`) помогают сделать сценарий более компактным, но точно так же можно использовать обычный синтаксис:

```
tags:
  type: web
  env: production
```

Создание более точных названий групп

Лорин не любит использовать такие имена групп, как `tag_type_web`. Он предпочитает более простые имена, например `web`.

Чтобы изменить имя, нужно в каталог *playbooks/inventory* добавить новый файл с информацией о группах. Это обычный файл реестра Ansible с именем *playbooks/inventory/hosts* (см. пример 17.6).

Пример 17.6. *playbooks/inventory/hosts*

```
[web:children]
tag_type_web
[tag_type_web]
```

После этого вы сможете обращаться к группе `web` в операциях Ansible.

Плагин инвентаризации `aws_ec2` поддерживает множество других возможностей для управления реестром. Для начала достаточно примера 17.3. Дополнительные сведения ищите в документации по плагину `aws_ec2` (**https://oreil.ly/nP8px**).

Виртуальные частные облака

Когда в 2006 году Amazon впервые запустила EC2, все экземпляры EC2 были подключены к одной плоской сети[1]. Каждый экземпляр EC2 получает приватный и публичный IP-адреса. В 2009 году Amazon представила новый механизм организации *виртуального частного облака* (Virtual Private Cloud, VPC). VPC позволяет пользователям управлять способом объединения экземпляров в сеть и определять, будет она публично доступной или изолированной. Термин *VPC* используется в Amazon для описания виртуальных сетей, которые пользователи могут создавать внутри EC2. VPC можно рассматривать как изолированную сеть. При создании VPC указывается диапазон IP-адресов. Это должно быть подмножество одного из диапазонов частных адресов (*10.0.0.0/8*, *172.16.0.0/12* или *192.168.0.0/16*).

Виртуальную сеть можно разделить на подсети с диапазонами IP-адресов, которые являются подмножествами диапазона IP-адресов всей сети VPC. В примере 17.14 показана VPC с диапазоном IP-адресов 10.0.0.0/16, и в ней определяются две подсети: *10.0.0.0/24* и *10.0.10/24*.

Запуская экземпляр, вы назначаете ему подсеть в VPC. Подсети можно настроить так, чтобы ваши экземпляры получали общедоступные или частные IP-адреса. EC2 также позволяет определять таблицы маршрутизации для трафика между подсетями и создавать интернет-шлюзы для маршрутизации трафика из подсетей в интернет.

Настройка сети – сложная тема, которая (далеко) выходит за рамки этой книги. Для получения дополнительной информации обращайтесь к документации Amazon EC2 по VPC (**http://amzn.to/1Fw89Af**).

[1] Внутренняя сеть Amazon делится на подсети, но пользователи не могут управлять распределением экземпляров по этим подсетям.

Конфигурирование ansible.cfg для использования с ES2

Когда Лорин использует Ansible для настройки экземпляров EC2, он добавляет следующие строки в файл *ansible.cfg*:

```
[defaults]
remote_user = ec2-user
host_key_checking = False
```

В зависимости от используемых образов часто требуется подключаться по SSH, взяв имя конкретного пользователя, в данном случае `ec2-user`, но это также может быть `ubuntu` или `centos`. Лорин также отключает проверку ключей хоста, так как заранее неизвестно, какие ключи понадобятся для новых экземпляров[1].

Запуск новых экземпляров

Модуль `amazon.aws.ec2_instance` позволяет запускать новые экземпляры в EC2. Это один из наиболее сложных модулей Ansible, поскольку поддерживает огромное количество аргументов.

В примере 17.7 показан простой сценарий для запуска экземпляра Ubuntu 20.04 в EC2.

Пример 17.7. Простой сценарий для создания экземпляра EC2

```
- name: Configure and start EC2 instance
  amazon.aws.ec2_instance:
    name: 'web1'
    image_id: 'ami-0e8286b71b81c3cc1'
    instance_type: 't2.micro'
    key_name: 'ec2key'
    region: "{{ lookup('env', 'AWS_REGION') }}"
    security_group: "{{ security_group }}"
    network:
      assign_public_ip: true
    tags:
      type: web
      env: production
    volumes:
      - device_name: /dev/sda1
        ebs:
          volume_size: 16
          delete_on_termination: true
```

[1] От Лорина: Получить ключи можно, послав EC2 запрос на вывод экземпляра в консоли. Но должен признаться, что я никогда не утруждал себя этим, поскольку никогда не сталкивался с необходимостью извлечения ключей хоста из вывода в консоли.

```
    wait: true
    register: ec2
```

Давайте рассмотрим значения параметров.

- Параметр `image_id` в примере 17.7 определяет идентификатор образа машины Amazon (AMI ID), который нужно указывать всегда. Как уже говорилось выше, образ – это, по сути, файловая система, содержащая установленную операционную систему. Использованный в примере идентификатор `ami-0e8286b71b81c3cc1` относится к образу с установленной 64-битной версией CentOS 7.
- Параметр `instance_type` описывает количество ядер CPU, объем памяти и хранилища, которыми будет располагать экземпляр. EC2 не позволяет устанавливать произвольные комбинации количества ядер, объема памяти и хранилища. Вместо этого Amazon определяет набор типов экземпляров[1]. В примере 17.7 используется тип экземпляра *t2.micro*. Это 64-битный экземпляр с одним процессором, 1 Гбайт оперативной памяти и хранилищем на основе EBS (подробнее об этом чуть ниже).
- Параметр `key_name` ссылается на пару ключей SSH. Amazon использует пары ключей SSH для предоставления доступа к их серверам. До запуска первого сервера вам необходимо либо создать новую пару SSH-ключей, либо выгрузить открытый ключ из пары, созданной заранее. В любом случае вы должны дать имя паре ключей SSH.
- Параметр `region` определяет географическое местоположение центра обработки данных, где будет запущен экземпляр. В этом примере мы используем значение переменной окружения `AWS_REGION`.
- Параметр `security_group` определяет список групп безопасности – правил брандмауэра, связанных с экземпляром. Эти группы безопасности определяют, какие типы входящих и исходящих соединений разрешены. Например, веб-серверу разрешено прослушивать TCP-порты 80 и 443, а системе Ansible разрешено подключаться по SSH через TCP-порт 22.

  В разделе `network` мы указали, что нам нужен общедоступный IP-адрес в интернете.
- Параметр `tags` связывает метаданные с экземпляром в форме тегов EC2 ключ/значение. В предыдущем примере были назначены следующие теги:

```
    tags:
      Name: ansiblebook
```

[1] Существует удобный (неофициальный) веб-сайт (https://oreil.ly/ztoCB), где можно найти единую таблицу со всеми доступными типами экземпляров EC2.

```
type: web
env: production
```

Вызов модуля `amazon.aws.ec2_instance` из командной строки – самый простой способ завершить экземпляр, если вы знаете его идентификатор:

```
$ ansible localhost -m amazon.aws.ec2_instance -a \
'instance_id=i-01176c6682556a360' \
-a state=absent'
```

Пары ключей EC2

В примере 17.7 мы предположили, что Amazon уже знает о паре ключей SSH с именем `mykey`. Давайте посмотрим, как можно использовать Ansible для создания новых пар ключей.

Создание нового ключа

При создании новой пары ключей на основе парольной фразы Amazon генерирует пару ключей типа ed25519 с защитой взлома методом простого перебора:

```
$ ssh-keygen -t ed25519 -a 100 -C '' -f ~/.ssh/ec2-user
```

Открытый ключ сохраняется в файле *~/.ssh/ec2-user.pub*. В этом файле будет создана всего одна строка, например:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIOvcnUtQI2wd4GwfOL4RckmwTinG1Zw7ia96EpVObs9x
```

Выгрузка открытого ключа

Создав пару ключей SSH, вы должны выгрузить открытый ключ в Amazon. Закрытый ключ не должен никому передаваться, также нежелательно передавать кому-либо открытый ключ. Это вопрос конфиденциальности и безопасности.

```
---
- name: Register SSH keypair
  hosts: localhost
  gather_facts: false
  tasks:
    - name: Upload public key
      amazon.aws.ec2_key:
        name: ec2key
        key_material: "{{ item }}"
        state: present
```

```
        force: true
      no_log: true
      with_file:
        - ~/.ssh/ec2key.pub
...
```

Группы безопасности

В примере 17.7 предполагается, что группа безопасности `my_security_group` уже существует. Проверить наличие групп перед их использованием можно с помощью модуля `amazon.aws.ec2_group`.

Группы безопасности похожи на правила брандмауэра: они определяют, кто и как может подключаться к машине. В примере 17.8 определяется группа безопасности, позволяющая любому хосту в интернете подключаться к портам 80 и 443. В этом примере также разрешается любому хосту подключаться к порту 22, но, возможно, вы решите ограничить список хостов, задав конкретные адреса. Разрешены также исходящие соединения с кем угодно в интернете по HTTP и HTTPS. Исходящие соединения нам необходимы для загрузки пакетов из интернета. Более безопасной альтернативой было бы разрешить доступ к репозиторию или фильтрующему прокси-серверу.

Пример 17.8. Группы безопасности

```
- name: Configure SSH security group
  amazon.aws.ec2_group:
    name: my_security_group
    description: SSH and Web Access
    rules:
      - proto: tcp
        from_port: 22
        to_port: 22
        cidr_ip: '0.0.0.0/0'
      - proto: tcp
        from_port: 80
        to_port: 80
        cidr_ip: 0.0.0.0/0
      - proto: tcp
        from_port: 443
        to_port: 443
        cidr_ip: 0.0.0.0/0
    rules_egress:
      - proto: tcp
        from_port: 443
        to_port: 443
        cidr_ip: 0.0.0.0/0
```

```
- proto: tcp
  from_port: 80
  to_port: 80
  cidr_ip: 0.0.0.0/0
```

Для тех, кто прежде не пользовался группами безопасности, поясним назначение параметров в словаре `rules` (см. табл. 17.2).

Таблица 17.2. Параметры правил групп безопасности

Парметр	Описание
`proto`	Протокол IP (`tcp`, `udp`, `icmp`) или `all`, чтобы разрешить все протоколы и порты
`cidr_ip`	Подсеть IP-адресов, разрешенных для подключения, в нотации CIDR
`from_port`	Первый порт в списке разрешенных
`to_port`	Последний порт в списке разрешенных

Разрешенные IP-адреса

Группы безопасности позволяют определять IP-адреса, которым разрешено подключаться к экземпляру. Подсеть определяется с помощью нотации бесклассовой адресации (Classless InterDomain Routing, CIDR). Пример подсети, описанной с помощью нотации CIDR: *203.0.113.0/24*[1]. Эта запись означает, что первые 24 бита IP-адреса должны соответствовать первым 24 битам адреса *203.0.113.0*. Иногда люди говорят «/24» для обозначения размера CIDR, заканчивающегося на */24*.

/24 – удобное значение, поскольку соответствует трем первым октетам адреса, а именно *203.0.113*[2]. Это значит, что любой IP-адрес, начинающийся с *203.0.113*, находится в этой подсети, т. е. любой IP-адрес из диапазона от *203.0.113.0* до *203.0.113.255*. Однако имейте в виду, что адреса с последним октетом 0 или 255 нельзя использовать для хостов.

Адрес *0.0.0.0/0* означает, что устанавливать соединения разрешено с любого IP-адреса.

Порты групп безопасности

Единственное, что нам кажется странным в группах безопасности EC2, – это нотация `from_port` и `to_port`. EC2 позволяет определять диапазон портов, к которым разрешен доступ. Например, вот как можно указать, что TCP-соединения разрешены с любым из портов с 5900 по 5999:

```
- proto: tcp
  from_port: 5900
```

[1] Так случилось, что этот пример соответствует особому диапазону IP-адресов TEST-NET-3, зарезервированному для примеров. Это *example.com* для IP-подсетей.

[2] Подсети /8, /16, /24 – очень хорошие примеры, поскольку расчеты в этом случае гораздо легче выполнять, чем, скажем, в случае /17 или /23.

```
    to_port: 5999
    cidr_ip: 0.0.0.0/0
```

Однако мы считаем такую нотацию запутывающей, поскольку сами никогда не указываем диапазоны портов[1]. Вместо этого мы обычно разрешаем порты с номерами, не идущими подряд, такими как 80 и 443. Вследствие этого почти в любой ситуации параметры `from_port` и `to_port` будут одинаковыми.

Модуль `amazon.aws.ec2_group` имеет много других параметров. За дополнительной информацией обращайтесь к документации.

Получение последней версии AMI

В примере 17.7 мы явно указали CentOS AMI:

```
image_id: ami-0e8286b71b81c3cc1
```

Но такой подход не годится, если вдруг появится желание запустить новейший образ Ubuntu 20.04. Связано это с тем, что Canonical (компания, управляющая проектом Ubuntu) часто выпускает небольшие обновления, и для каждого нового выпуска генерируется новый образ AMI. Если еще вчера идентификатор `ami-0d527b8c289b4af7f` соответствовал новейшему выпуску Ubuntu 20.04, то завтра это может быть уже не так.

В коллекции `amazon.aws` имеется интересный модуль `ec2_ami_info`, извлекающий список идентификаторов образов AMI, соответствующих критериям поиска, таким как имя образа или теги. Пример 17.9 демонстрирует, как использовать этот модуль для запуска последней 64-битной версии Ubuntu Focal 20.04 на EBS-экземпляре с дисками SSD. Вы можете использовать этот прием для создания экземпляра с последней версией AMI.

Пример 17.9. Получение идентификатора AMI новейшей версии Ubuntu

```
---
- name: Find latest Ubuntu image on Amazon EC2
  hosts: localhost
  gather_facts: false
  tasks:
    - name: Gather information on Ubuntu AMIs published by Canonical
      amazon.aws.ec2_ami_info:
        owners: 099720109477
        filters:
          name: "ubuntu/images/hvm-ssd/ubuntu-focal-20.04-*"
          architecture: "x86_64"
          root-device-type: "ebs"
```

[1] Внимательные читатели наверняка заметили, что порты 5900–5999 обычно используются протоколом VNC управления удаленным рабочим столом – одним из немногих, для которых указание диапазона портов имеет смысл.

```
          virtualization-type: "hvm"
          state: "available"
      register: ec2_ami_info

    - name: Sort the list of AMIs by date for the latest image
      set_fact:
        latest_ami: |
          {{ ec2_ami_info.images | sort(attribute='creation_date') | last }}
    - name: Display the latest AMI ID
      debug:
        var: latest_ami.image_id
...
```

В данном случае мы должны знать соглашение, используемое для именования образов Ubuntu. В случае с Ubuntu имя образа всегда заканчивается отметкой времени, например: *ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20211129*. В параметре `name` модуля `ec2_ami_info` допускается использовать шаблонный символ `*`. Задача регистрирует список образов AMI, благодаря чему можно узнать, какой образ самый свежий, отсортировав список по дате создания и взяв из него самый последний элемент.

В каждом дистрибутиве используется своя стратегия именования образов AMI, поэтому, чтобы развернуть образ AMI с дистрибутивом, отличным от Ubuntu, вам понадобится провести некоторое исследование и определить подходящую строку поиска.

Добавление нового экземпляра в группу

Иногда Лорин предпочитает написать единый сценарий для запуска экземпляра и затем выполнять на экземпляре другие сценарии.

К сожалению, до запуска сценария хост еще не существует. Запрет кеширования в сценарии динамической инвентаризации тут не поможет, потому что Ansible вызывает его в самом начале, до создания хоста.

Для добавления экземпляра в группу можно использовать задачу, вызывающую модуль `add_host`, как показано в примере 17.10.

Пример 17.10. Добавление экземпляра в группу

```
- name: Create an ubuntu instance on Amazon EC2
  hosts: localhost
  gather_facts: false
  tasks:
    - name: Configure and start EC2 instance
```

```
    amazon.aws.ec2_instance:
      name: 'web1'
      image_id: "{{ latest_ami.image_id }}"
      instance_type: "{{ instance_type }}"
      key_name: "{{ key_name }}"
      security_group: "{{ security_group }}"
      network:
        assign_public_ip: true
      tags: {type: web, env: production}
      volumes:
        - device_name: /dev/sda1
          ebs:
          volume_size: 16
          delete_on_termination: true
      wait: true
    register: ec2

  - name: Add the instances to the web and production groups
    add_host:
      hostname: "{{ item.public_dns_name }}"
      groupname:
        - web
        - production
    loop: "{{ ec2.instances }}"
  - name: Configure Web Server
    hosts: web:&production
    become: true
    gather_facts: true
    remote_user: ubuntu
    roles:
      - webserver
```

Модуль `amazon.aws.ec2_instance` возвращает словарь с большим количеством информации о запущенных экземплярах. Чтобы прочитать документацию, выполните следующую команду для установленной коллекции:

```
$ ansible-doc amazon.aws.ec2_instance
```

Ожидание запуска сервера

Облака IaaS, такие как EC2, требуют определенного времени для создания нового экземпляра. Это значит, что невозможно запустить сценарий на экземпляре EC2 сразу после отправки запроса на его создание. Необходимо подождать, пока этот экземпляр запустится. Также имейте

в виду, что экземпляр состоит из нескольких частей, создаваемых по очереди. Поэтому вам придется подождать, но как это организовать?

Модуль `ec2` поддерживает для этого параметр `wait`. Если в нем передать `yes`, то модуль `ec2` не вернет управления, пока экземпляр не перейдет в рабочее состояние.

Однако простой задержки в ожидании запуска экземпляра недостаточно, необходимо дождаться, пока экземпляр продвинется достаточно далеко в процессе загрузки и запустит сервер SSH.

Как раз для таких случаев написан модуль `wait_for`. Вот как можно использовать модули `ec2` и `wait_for`, чтобы запустить экземпляр и дождаться, когда он станет готов принимать соединения через SSH:

```
- name: Wait for EC2 instance to be ready
  wait_for:
    host: "{{ item.public_dns_name }}"
    port: 22
    search_regex: OpenSSH
    delay: 60
  loop: "{{ ec2.instances }}"
  register: wait
```

Вызов `wait_for` использует аргумент `search_regex` для поиска строки `OpenSSH` после подключения к хосту. Идея состоит в том, что в ответ на попытку установить соединение функционирующий сервер SSH вернет строку, похожую на ту, что показана в примере 17.11.

Пример 17.11. Ответ сервера SSH, работающего в Ubuntu

```
SSH-2.0-OpenSSH_8.2p1 Ubuntu-4ubuntu0.3
```

Можно было бы с помощью модуля `wait_for` просто проверить доступность порта 22. Однако иногда случается так, что в процессе загрузки сервер SSH успел открыть порт 22, но еще не готов обрабатывать запросы. Здесь также определена задержка в одну минуту, потому что для публикации имени сервера в DNS тоже требуется некоторое время. Ожидание первоначального ответа гарантирует, что модуль `wait_for` вернет управление, только когда сервер SSH будет полностью работоспособен.

Подведение итогов

В примере 17.12 приводится сценарий, создающий экземпляр EC2 и настраивающий его как веб-сервер. Сценарий является идемпотентным, т. е. его можно спокойно запускать несколько раз – он создаст новый экземпляр, только если тот еще не был создан.

Пример 17.12. *ec2-example.yml*: законченный сценарий для создания экземпляра EC2

```
---
- name: Provision Ubuntu Web Server on Amazon EC2
  hosts: localhost
  gather_facts: false
  vars:
    instance_type: t2.micro
    key_name: ec2key
    aws_region: "{{ lookup('env', 'AWS_REGION') }}"
    security_group: my_security_group
  tasks:
    - name: Upload public key ec2key.pub
      amazon.aws.ec2_key:
        name: "{{ key_name }}"
        key_material: "{{ item }}"
        state: present
        force: true
      no_log: true
      with_file:
        - ~/.ssh/ec2key.pub

    - name: Configure my_security_group
      amazon.aws.ec2_group:
        name: "{{ security_group }}"
        region: "{{ aws_region }}"
        description: SSH and Web Access
        rules:
          - proto: tcp
            from_port: 22
            to_port: 22
            cidr_ip: '0.0.0.0/0'
          - proto: tcp
            from_port: 80
            to_port: 80
            cidr_ip: 0.0.0.0/0
          - proto: tcp
            from_port: 443
            to_port: 443
            cidr_ip: 0.0.0.0/0
        rules_egress:
          - proto: tcp
            from_port: 443
            to_port: 443
            cidr_ip: 0.0.0.0/0
          - proto: tcp
```

```
        from_port: 80
        to_port: 80
        cidr_ip: 0.0.0.0/0

- name: Gather information on Ubuntu AMIs published by Canonical
  amazon.aws.ec2_ami_info:
    region: "{{ aws_region }}"
    owners: 099720109477
    filters:
      name: "ubuntu/images/hvm-ssd/ubuntu-focal-20.04-*"
      architecture: "x86_64"
      root-device-type: "ebs"
      virtualization-type: "hvm"
      state: "available"
  register: ec2_ami_info

- name: Sort the list of AMIs by date for the latest image
  set_fact:
    latest_ami: |
      {{ ec2_ami_info.images | sort(attribute='creation_date') | last }}

- name: Configure and start EC2 instance
  amazon.aws.ec2_instance:
    region: "{{ aws_region }}"
    name: 'web1'
    image_id: "{{ latest_ami.image_id }}"
    instance_type: "{{ instance_type }}"
    key_name: "{{ key_name }}"
    security_group: "{{ security_group }}"
    network:
      assign_public_ip: true
    tags:
      type: web
      env: production
    volumes:
      - device_name: /dev/sda1
        ebs:
          volume_size: 16
          delete_on_termination: true
    wait: true
  register: ec2

- name: Wait for EC2 instance to be ready
  wait_for:
    host: "{{ item.public_dns_name }}"
    port: 22
    search_regex: OpenSSH
```

```
        delay: 30
      loop: "{{ ec2.instances }}"
      register: wait
    - name: Add the instances to the web and production groups
      add_host:
        hostname: "{{ item.public_dns_name }}"
        groupname:
          - web
          - production
      loop: "{{ ec2.instances }}"

    - name: Configure Web Server
      hosts: web:&production
      become: true
      gather_facts: true
      remote_user: ubuntu
      roles:
        - ssh
        - webserver
...
```

Определения ролей для этого примера можно найти на GitHub (**https://oreil.ly/2hAPe**).

Создание виртуального частного облака

До сих пор мы запускали экземпляры в виртуальном частном облаке (VPC) по умолчанию. Однако Ansible позволяет также создавать новые облака VPC и запускать в них экземпляры.

В примере 17.13 показано, как создать VPC с интернет-шлюзом, двумя подсетями и таблицей маршрутизации, которая управляет прохождением исходящих соединений через интернет-шлюз.

Пример 17.13. *create-vpc.yml*: создание VPC

```
---
- name: Create a Virtual Private Cloud (VPC)
  hosts: localhost
  gather_facts: false
  vars:
    aws_region: "{{ lookup('env', 'AWS_REGION') }}"
  tasks:
    - name: Create a vpc
      amazon.aws.ec2_vpc_net:
        region: "{{ aws_region }}"
        name: "Book example"
        cidr_block: 10.0.0.0/16
```

```
        tags:
          env: production
      register: result

    - name: Set vpc_id as fact
      set_fact:
        vpc_id: "{{ result.vpc.id }}"

    - name: Add gateway
      amazon.aws.ec2_vpc_igw:
        region: "{{ aws_region }}"
        vpc_id: "{{ vpc_id }}"

    - name: Create web subnet
      amazon.aws.ec2_vpc_subnet:
        region: "{{ aws_region }}"
        vpc_id: "{{ vpc_id }}"
        cidr: 10.0.0.0/24
        tags:
          env: production
          tier: web

    - name: Create db subnet
      amazon.aws.ec2_vpc_subnet:
        region: "{{ aws_region }}"
        vpc_id: "{{ vpc_id }}"
        cidr: 10.0.1.0/24
        tags:
          env: production
          tier: db

    - name: Set routes
      amazon.aws.ec2_vpc_route_table:
        region: "{{ aws_region }}"
        vpc_id: "{{ vpc_id }}"
        tags:
          purpose: permit-outbound
        subnets:
          - 10.0.0.0/24
          - 10.0.1.0/24
        routes:
          - dest: 0.0.0.0/0
            gateway_id: igw
...
```

Все эти команды являются идемпотентными, но каждый модуль реализует механизм контроля идемпотентности по-своему (см. табл. 17.3).

Таблица 17.3. Логика контроля идемпотентности в некоторых модулях поддержки VPC

Модуль	Контроль идемпотентности
ec2_vpc_net	Параметры name и cidr
ec2_vpc_igw	Наличие интернет-шлюза
ec2_vpc_subnet	Параметры vpc_id и cidr
ec2_vpc_route_table	Параметры vpc_id и tags[1]

Если в ходе проверки идемпотентности будет обнаружено несколько экземпляров, Ansible завершит работу модуля с признаком ошибки.

Если не указать теги в ec2_vpc_route_table, то при каждом обращении к модулю будет создаваться новая таблица маршрутизации.

Необходимо отметить, что пример 17.12 довольно прост с точки зрения настройки сети, потому что мы определили всего две подсети: одна из них подключена к интернету, а другая – нет. Мы должны настроить группы безопасности для маршрутизации трафика из подсети web в базу данных и из интернета в подсеть web для организации доступа по SSH к внутренней подсети, в которой мы находимся, и задать правила для исходящего трафика, чтобы получить возможность устанавливать пакеты. В примере 17.14 показано определение таких групп безопасности.

Пример 17.14. Группы безопасности EC2

```
---
- name: Create EC2 Security Groups
  hosts: localhost
  vars:
    aws_region: "{{ lookup('env', 'AWS_REGION') }}"
    database_port: 5432
    cidrs:
      web: 10.0.0.0/24
      db: 10.0.1.0/24
      ssh: 203.0.113.0/24
  tasks:
    - name: DB security group
      amazon.aws.ec2_group:
        name: db
        region: "{{ aws_region }}"
        description: allow database access for web servers
        vpc_id: "{{ vpc_id }}"
        rules:
```

```
          - proto: tcp
            from_port: "{{ database_port }}"
            to_port: "{{ database_port }}"
            cidr_ip: "{{ cidrs.web }}"

    - name: Web security group
      amazon.aws.ec2_group:
        name: web
        region: "{{ aws_region }}"
        description: allow http and https access to web servers
        vpc_id: "{{ vpc_id }}"
        rules:
          - proto: tcp
            from_port: 80
            to_port: 80
            cidr_ip: 0.0.0.0/0
          - proto: tcp
            from_port: 443
            to_port: 443
            cidr_ip: 0.0.0.0/0

    - name: SSH security group
      amazon.aws.ec2_group:
        name: ssh
        region: "{{ aws_region }}"
        description: allow ssh access
        vpc_id: "{{ vpc_id }}"
        rules:
          - proto: tcp
            from_port: 22
            to_port: 22
            cidr_ip: "{{ cidrs.ssh }}"

    - name: Outbound security group
      amazon.aws.ec2_group:
        name: outbound
        description: allow outbound connections to the internet
        region: "{{ aws_region }}"
        vpc_id: "{{ vpc_id }}"
        rules_egress:
          - proto: all
            cidr_ip: 0.0.0.0/0
...
```

Обратите внимание, что `vpc_id` должен быть кешированным фактом или дополнительной переменной в командной строке.

Динамическая инвентаризация и VPC

При использовании VPC экземпляры часто помещаются в закрытую подсеть, не подключенную к интернету. В этом случае экземпляры не имеют публичных IP-адресов.

В такой ситуации может потребоваться запустить Ansible в экземпляре внутри VPC. Сценарий динамической инвентаризации достаточно эффективно отличает внутренние IP-адреса экземпляров VPC, не имеющих публичных IP-адресов.

Заключение

Ansible поддерживает не только EC2, но и другие службы AWS. Использование Ansible с EC2 – достаточно обширная тема, чтобы ей можно было посвятить целую книгу. На самом деле Ян Курниаван (Yan Kurniawan) написал такую книгу: «Ansible for AWS» (Packt, 2016). После изучения этой главы у вас должно быть достаточно знаний, чтобы без труда освоить другие модули.

Глава 18

Плагины обратного вызова

Система Ansible поддерживает так называемые *плагины обратного вызова* (callback plugins), которые могут выполнять некоторые действия в ответ на такие события, как запуск операции или завершение задачи на хосте. Плагины обратного вызова можно использовать, например, для отправки сообщений Slack или для вывода записей в удаленный журнал. Даже информация, которую вы видите в окне терминала во время выполнения сценария Ansible, фактически выводится плагином обратного вызова.

Ansible поддерживает три вида плагинов обратного вызова:

- *плагины стандартного вывода*;
- *плагины уведомлений*;
- *плагины агрегирования*.

Плагины стандартного вывода управляют форматом отображения информации на экране. Однако Ansible не различает плагины уведомлений и агрегирования, которые выполняют самые разные действия, не связанные с выводом.

Плагины стандартного вывода

В каждый конкретный момент времени активным может быть только один плагин стандартного вывода. Назначается плагин стандартного вывода установкой параметра `stdout_callback` в секции `defaults` в файле *ansible.cfg*. Например, вот как можно выбрать плагин `yaml`, преобразующий вывод в более удобочитаемый формат:

```
[defaults]
stdout_callback = yaml
```

С помощью команды `ansible-doc -t callback -l` можно получить список плагинов, доступных в установленной версии Ansible. В табл. 18.1 перечислены некоторые плагины стандартного вывода, которыми предпочитает пользоваться Бас.

Таблица 18.1. Плагины стандартного вывода

Имя	Описание	Зависимости Python
`ara`	ARA Records Ansible	*ara (сервер)*
`debug`	Выводит содержимое stderr и stdout в удобочитаемом виде	
`default`	Отображает вывод по умолчанию	
`dense`	Затирает старый вывод вместо прокрутки	
`json`	Выводит информацию в формате JSON	
`minimal`	Выводит результаты выполнения задач с минимальным форматированием	
`null`	Не отображает выводимую информацию	
`oneline`	Действует подобно плагину minimal, но выводит информацию в одну строку	

Плагин `actionable` был удален в версии Ansible 2.11. Вместо него можно использовать плагин `default` с параметрами `display_skipped_hosts = false` и `display_ok_hosts = false`.

ARA

ARA Records Ansible (ARA, еще один рекурсивный акроним) – не просто плагин обратного вызова. Он дает возможность сохранять все детали выполнения команд `ansible` и `ansible-playbook` (рис. 18.1). Если все разработчики в команде используют ARA, то любой сможет увидеть вывод, сгенерированный этим плагином!

В простейшем случае записи сохраняются в файл SQLite, но при желании можно сохранять данные в любой другой базе данных, а также просматривать их в браузере, настроив веб-сайт на Django, который обращается к ARA API, или в клиенте командной строки (**https://oreil.ly/RCqcF**). Установить модули ARA для версии Python, которую использует система Ansible, можно так:

```
$ pip3 install --user "ara[server]"
$ export ANSIBLE_CALLBACK_PLUGINS="$(python3 -m ara.setup.callback_plugins)"
# ... запустите свои сценарии ...
$ ara-manage runserver
```

Подробности ищите в документации ARA (**https://oreil.ly/1q40c**).

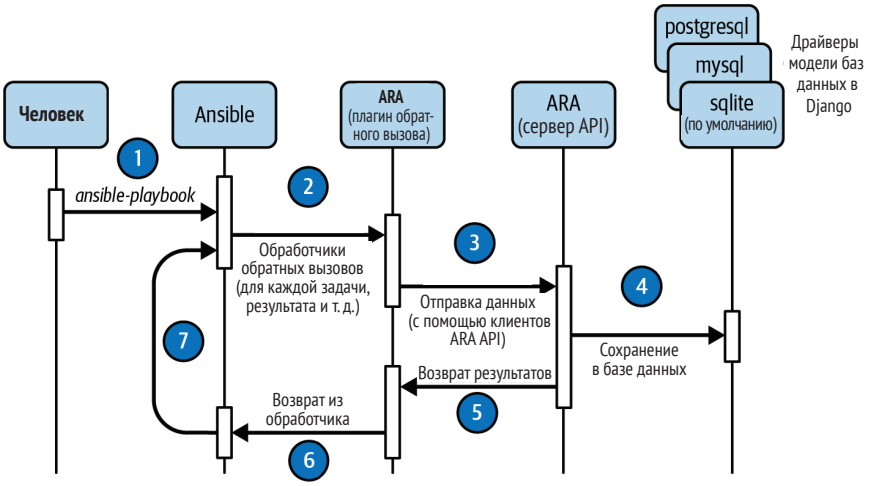


Рис. 18.1. Запись данных из Ansible в базу данных с помощью ARA

debug

Плагин `debug` упрощает чтение потоков `stdout` и `stderr` задачи и может пригодиться для отладки. При использовании плагина `default` чтение вывода может оказаться сложной задачей:

```
TASK [Clone repository] ************************************************************
fatal: [one]: FAILED! => {"changed": false, "cmd": "/usr/bin/git clone --origin
origin '' /tmp/mezzanine_example", "msg": "Cloning into '/tmp/mezzanine_example'...
\n/private/tmp/mezzanine_example/.git: Permission denied", "rc": 1, "stderr":
"Cloning into '/tmp/mezzanine_example'...\n/private/tmp/mezzanine_example/.git:
Permission denied\n", "stderr_lines": ["Cloning into '/tmp/mezzanine_example'...",
"/private/tmp/mezzanine_example/.git: Permission denied"], "stdout": "",
"stdout_lines": []}
```

Но благодаря дополнительному форматированию, осуществляемому плагином `debug`, читать вывод намного проще:

```
TASK [Clone repository] ************************************************************
fatal: [one]: FAILED! => {
    "changed": false,
    "cmd": "/usr/bin/git clone --origin origin '' /tmp/mezzanine_example",
    "rc": 1
}
STDERR:
Cloning into '/tmp/mezzanine_example'...
/private/tmp/mezzanine_example/.git: Permission denied
MSG:
Cloning into '/tmp/mezzanine_example'...
/private/tmp/mezzanine_example/.git: Permission denied
```

default

Если не настроить `stdout_callback`, то для отображения информации используется плагин `default`, который форматирует вывод так:

```
TASK [Clone repository] ************************************************************
changed: [one]
```

dense

Плагин `dense` (появился в версии Ansible 2.3) всегда отображает только две строки вывода. Он затирает предыдущие строки, не выполняя скроллинга:

```
PLAY 1: LOCAL
task 1: one
```

json

Плагин `json` выводит информацию в машиночитаемом формате JSON. Он может пригодиться в случаях, когда требуется организовать обработку вывода Ansible с использованием программ. Обратите внимание, что этот плагин не генерирует вывода, пока сценарий не завершится целиком. Вывод в формате JSON обычно получается слишком объемным, чтобы показать его здесь.

minimal

Плагин применяет минимум обработки к результатам, возвращаемым с событием Ansible. Например, если плагин `default` форматирует вывод задачи так:

```
TASK [Clone repository] ************************************************************
changed: [one]
```

то плагин `minimal` выведет:

```
one | CHANGED => {
    "after": "2c19a94be566058e4430c46b75e3ce9d17c25f56",
    "before": null,
    "changed": true
}
```

null

Плагин `null` полностью отключает вывод.

oneline

Плагин `oneline` напоминает плагин `minimal`, но выводит информацию в одну строку (здесь пример вывода показан в нескольких строках, потому что на книжной странице он не умещается в одну строку):

```
one | CHANGED => {"after": "2c19a94be566058e4430c46b75e3ce9d17c25f56","before": ...
```

Плагины уведомлений и агрегирования

Другие плагины выполняют разнообразные действия, такие как запись времени выполнения или отправка уведомлений Slack. Эти плагины перечислены в табл. 18.2.

Таблица 18.2. Другие плагины

Имя	Описание	Зависимости Python
`foreman`	Посылает уведомление в Foreman	*requests*
`jabber`	Посылает уведомление в Jabber	*xmpppy*
`junit`	Записывает данные в XML-файл в формате Junit	*junit_xml*
`log_plays`	Записывает в журнал результаты выполнения сценария для каждого хоста	
`logentries`	Посылает уведомление в Logentries	*certifi flatdict*
`logstash`	Посылает результаты в Logstash	*logstash*
`mail`	Посылает электронное письмо, если выполнение задачи завершилось с ошибкой	
`nrdp`	Посылает результаты задачи на сервер Nagios	
`say`	Озвучивает уведомление с помощью ПО голосового синтезатора	
`profile_roles`	Создает отчет о времени выполнения для каждой роли	
`profile_tasks`	Создает отчет о времени выполнения для каждой задачи	
`slack`	Посылает уведомление в Slack	*prettytable*
`splunk`	Посылает результаты задачи в Splunk	
`timer`	Создает отчет об общем времени выполнения	

В отличие от плагинов стандартного вывода другие плагины могут действовать одновременно. Активировать плагины из этой категории можно с помощью параметра `callback_whitelist` в файле *ansible.cfg*, перечислив их через запятую, например:

```
[defaults]
callback_whitelist = mail, slack
```

Параметр `callback_whitelist` вскоре будет переименован в `callback_enabled`.

Многие из этих плагинов имеют дополнительные параметры настройки, определяемые через переменные окружения или в файле *ansible.cfg*. Бас предпочитает настраивать эти параметры в *ansible.cfg*, чтобы не захламлять окружение дополнительными переменными. Кроме того, *ansible.cfg* можно сохранить в системе управления версиями, чтобы этими настройками могли воспользоваться другие разработчики или пользователи.

Выяснить, какие параметры поддерживает конкретный плагин, можно с помощью команды:

```
$ ansible-doc -t callback <имя_плагина>
```

Зависимости Python

Многие плагины требуют, чтобы на управляющей машине Ansible были установлены дополнительные библиотеки для Python. В табл. 18.2 перечислены плагины и их зависимости. Установите их чтобы получить возможность использовать эти плагины; например, вот как можно установить библиотеку `prettytable` для поддержки Slack:

```
$ pip3 install prettytable
```

foreman

Плагин `foreman` посылает уведомления в Foreman (http://theforeman.org/). В табл. 18.3 перечислены параметры, используемые для настройки плагина, которые должны находиться в секции `[callback_foreman]` в файле *ansible.cfg*.

Таблица 18.3. Параметры настройки для плагина `foreman`

Параметр	Описание	Значение по умолчанию
`url`	URL сервера Foreman	*http://localhost:3000*
`client_cert`	Сертификат X509 для аутентификации на сервере Foreman, если используется протокол HTTPS	*/etc/foreman/client_cert.pem*
`client_key`	Соответствующий закрытый ключ	*/etc/foreman/client_key.pem*
`verify_certs`	Необходимость проверки сертификата Foreman. Значение 1 требует проверять сертификаты SSL с использованием установленных центров сертификации. Значение 0 запрещает проверку	1

jabber

Плагин `jabber` посылает уведомления в Jabber (**http://jabber.org/**). Обратите внимание, что настройки для этого плагина не имеют значений по умолчанию. Они перечислены в табл. 18.4 и определяются исключительно через переменные окружения.

Таблица 18.4. Переменные окружения плагина `jabber`

Переменная окружения	Описание
JABBER_SERV	Имя хоста сервера Jabber
JABBER_USER	Имя пользователя Jabber для аутентификации
JABBER_PASS	Пароль пользователя Jabber для аутентификации
JABBER_TO	Пользователь Jabber, которому посылается уведомление

junit

Плагин `junit` записывает результаты выполнения сценария в XML-файл в формате JUnit. Настраивается с помощью переменных окружения, перечисленных в табл. 18.5. Создание XML-отчетов производится в соответствии с соглашениями, перечисленными в табл. 18.6 .

Таблица 18.5. Переменные окружения плагина `junit`

Переменная окружения	Описание	Значение по умолчанию
JUNIT_OUTPUT_DIR	Каталог для файлов отчетов	*~/.ansible.log*
JUNIT_TASK_CLASS	Настройки вывода: по одному классу в файле YAML	false
JUNIT_FAIL_ON_CHANGE	Каждую задачу, вернувшую статус `"changed"`, рассматривает как неудачный тест Junit	false
JUNIT_FAIL_ON_IGNORE	Каждую задачу, вернувшую статус `"changed"`, рассматривает как неудачный тест JUnit, даже если установлен параметр `ignore_on_error`	false
JUNIT_HIDE_TASK_ARGUMENTS	Скрывать аргументы задачи	false
JUNIT_INCLUDE_SETUP_TASKS_IN_REPORT	Определяет необходимость включения в отчет задач, осуществляющих подготовку окружения тестирования	true

Таблица 18.6. Отчет JUnit

Вывод задачи Ansible	Отчет Junit
ok	pass
Ошибка с текстом EXPECTED FAILURE в имени задачи	pass
Ошибка как результат исключения	error
Ошибка по другой причине	failure
skipped	skipped

log_plays

Плагин `log_plays` записывает результаты в файлы журналов в каталоге `log_folder` по одному файлу на хост.

logentries

Плагин `logentries` генерирует объекты JSON и посылает их в Logentries (**http://logentries.com/**). Параметры настройки плагина должны определяться в секции `[callback_logentries]` в файле *ansible.cfg* и перечислены в табл. 18.7.

Таблица 18.7. Параметры настройки плагина `logentries`

Параметр	Описание	Значение по умолчанию
`token`	Токен сервера Logentries	*(Нет)*
`api`	Имя хоста конечной точки Logentries	`data.logentries.com`
`port`	Порт Logentries	`80`
`tls_port`	Порт TLS Logentries	`443`
`use_tls`	Использовать TLS для взаимодействий с Logentries	`false`
`flatten`	Реструктурировать результаты	`false`

logstash

Плагин `logstash` передает факты и события задач в Logstash. Параметры настройки плагина должны определяться в секции `[callback_logstash]` в файле *ansible.cfg* и перечислены в табл. 18.8.

Таблица 18.9. Параметры настройки плагина `logstash`

Параметр	Описание	Значение по умолчанию
`format_version`	Формат журналирования	`v1`
`server`	Имя хоста сервера Logstash	`localhost`
`port`	Порт сервера Logstash	`5000`
`pre_command`	Команда, которая должна выполняться перед запуском. Ее результат помещается в поле `ansible_pre_command_output`	`null`
`type`	Тип сообщения	`ansible`

mail

Плагин `mail` посылает электронное письмо, когда задача завершается с ошибкой. Параметры настройки плагина должны определяться в секции `[callback_mail]` в файле *ansible.cfg* и перечислены в табл. 18.9.

Таблица 18.9. Параметры настройки плагина `mail`

Параметр	Описание	Значение по умолчанию
`bcc`	Скрытые получатели копии письма	`null`
`cc`	Получатели копии письма	`null`
`mta`	Агент транспорта электронной почты	`localhost`
`mtaport`	Порт агента транспорта электронной почты	`25`
`sender`	Отправитель	`null`
`to`	Получатель	`root`

profile_roles

Этот плагин генерирует отчет о времени выполнения ролей Ansible.

profile_tasks

Плагин `profile_tasks` генерирует отчет о времени выполнения отдельных задач и общего времени выполнения сценария, например:

```
Wednesday 11 August 2021  23:00:43 +0200 (0:00:00.910)       0:01:26.498 ******
===============================================================================
Install apt packages ---------------------------------------------- 83.50s
Gathering Facts --------------------------------------------------- 1.46s
Check out the repository on the host ------------------------------ 0.91s
Create project path ----------------------------------------------- 0.40s
Create a logs directory -------------------------------------------- 0.21s
```

Плагин также выводит информацию о времени в процессе выполнения задач, в том числе:

- дату и время запуска задачи;
- время выполнения предыдущей задачи, в скобках;
- накопленное время выполнения для данного сценария.

Вот пример вывода такой информации:

```
TASK [Create project path] ****************************************************
Wednesday 11 August 2021  23:00:42 +0200 (0:01:23.500)            0:01:24.975
changed: [web] ==> {"changed": true, "gid": 1000, "group": "vagrant", "mode":
"0755", "owner": "vagrant", "path": "/home/vagrant/mezzanine/mezzanine_example",
"size": 4096, "state": "directory", "uid": 1000}
```

В табл. 18.10 перечислены переменные окружения, используемые для настройки плагина.

Таблица 18.10. Переменные окружения плагина profile_tasks

Переменная окружения	Описание	Значение по умолчанию
PROFILE_TASKS_SORT_ORDER	Сортировка вывода (ascending, none)	none
PROFILE_TASKS_TASK_OUTPUT_LIMIT	Максимальное количество задач в отчете или all	20

say

Плагин say использует программу say или espeak чтобы сгенерировать голосовое уведомление. Этот плагин не имеет параметров настройки. Модуль say, в свою очередь, имеет параметр voice.

Обратите внимание, что в версии 2.8 плагин osx_say был переименован в say.

slack

Плагин slack посылает уведомления в Slack (**http://slack.com/**). Параметры настройки плагина должны определяться в секции [callback_slack] в файле *ansible.cfg* и перечислены в табл. 18.11.

Таблица 18.11. Параметры настройки плагина slack

Параметр	Описание	Значение по умолчанию
webhook_url	Адрес URL точки входа в Slack	(Нет)
channel	Комната Slack для отправки сообщения	#ansible
username	Имя пользователя, отправившего сообщение	ansible
validate_certs	Проверять сертификат SSL сервера Slack	true

splunk

Этот плагин отправляет результаты выполнения задачи в формате JSON в HTTP-коллектор Splunk. Параметры настройки плагина должны определяться в секции [callback_splunk] в файле *ansible.cfg* и перечислены в табл. 18.12.

Таблица 18.12. Параметры настройки плагина splunk

Параметр	Описание	Значение по умолчанию
authtoken	Токен аутентификации, используемый для подключения к HTTP-коллектору Splunk	null
include_milliseconds	Определяет необходимость добавления миллисекунд в поле времени	false
url	Адрес URL HTTP-коллектора Splunk	ansible
validate_certs	Проверять сертификат SSL сервера Splunk	true

timer

Плагин `timer` выводит общее время выполнения сценария, например:

```
Playbook run took 0 days, 0 hours, 2 minutes, 16 seconds
```

Для этой цели обычно лучше использовать плагин `profile_tasks`, который дополнительно выводит время выполнения каждой задачи.

Заключение

Плагины обратного вызова в Ansible поддерживают множество способов отправки отчетов в информационные каналы, используемые в организации, что придает Ansible дополнительную ценность, так как все эти возможности позволяют использовать систему для создания комплексных решений в разных областях в сочетании с другими инструментами.

Глава 19

Собственные модули

Иногда требуется решить задачу, слишком сложную для модуля `command` или `shell`, и при этом не существует готовых модулей для ее выполнения. В таком случае можно написать модуль самостоятельно.

Модули можно считать «глаголами» в «языке» Ansible – без них YAML ничего не смог бы сделать. Для машин Linux/BSD/Unix модули Ansible программируются на языке Python, а для машин Windows – на PowerShell, но, в принципе, они могут программироваться на любом языке. На рис. 19.1 показаны основные компоненты Ansible: проекты со сценариями, реестр и модули.

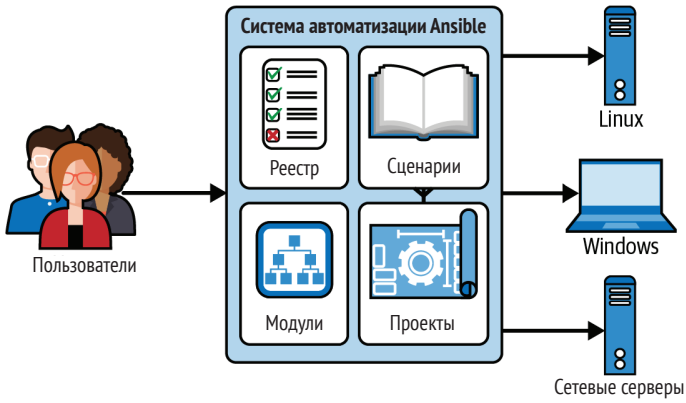


Рис. 19.1. Модули

Пример: проверка доступности удаленного сервера

Допустим, нужно проверить доступность конкретного порта удаленного сервера. Если соединение с этим портом установить невозможно, нужно, чтобы Ansible считала это ошибкой и прекращала операцию.

Свой модуль, которым мы будем заниматься в этой главе, является упрощенной версией модуля `wait_for`.

Использование модуля script вместо написания своего модуля

Помните, как в примере 7.13 мы использовали модуль `script` для запуска своих сценариев на удаленных хостах? Иногда действительно проще использовать модуль `script`, чем писать свой, полноценный модуль Ansible.

Лорин хранит такие сценарии в папке *scripts* рядом со сценариями Ansible. Например, можно создать сценарий *playbooks/scripts/can_reach.sh*, который принимает имя хоста, порт и количество попыток соединения:

```
$ ./can_reach.sh www.example.com 80 1
```

Можно создать сценарий командной оболочки, вызывающий `netcat`, как показано в примере 19.1.

Пример 19.1. *can_reach.sh*

```
#!/bin/bash -eu
host="$1"
port="$2"
timeout="$3"
nc -z -w "$timeout" "$host" "$port"
```

А затем вызвать его, как показано ниже:

```
- name: Run my custom script
  script: scripts/can_reach.sh www.google.com 80 1
```

Имейте в виду, что сценарий будет запускаться на удаленных хостах так же, как модули Ansible. Вследствие этого любые программы, необходимые сценарию (такие как `nc` в примере 19.1), должны быть установлены на удаленных хостах заранее. Например, файл Vagrantfile для этой главы устанавливает все необходимое, выполняя команду `vagrant up`, что дает возможность экспериментировать со сценарием *playbook.yml*.

Сценарий можно написать на языке Perl, если Perl установлен на удаленных хостах. В первой строке сценарий должен взывать интерпретатор Perl, как показано в примере 19.2[1].

Пример 19.2. *can_reach.pl*

```
#!/usr/bin/perl
use strict;
use English qw( -no_match_vars );    # PBP 79
use Carp;                            # PBP 283
use warnings;                        # PBP 431
```

[1] Обратите внимание, что этот сценарий компилируется в `perlcritic --brutal`.

```
use Socket;
our $VERSION = 1;
my $host = $ARGV[0], my $port = $ARGV[1];

# создать сокет, подключиться к порту
socket SOCKET, PF_INET, SOCK_STREAM, ( getprotobyname 'tcp' )[2]
    or croak "Can't create a socket $OS_ERROR\n";
connect SOCKET, pack_sockaddr_in( $port, inet_aton($host) )
    or croak "Can't connect to port $port! \n";

# вывести отчет
print "Connected to $host:$port\n" or croak "IO Error $OS_ERROR";

# закрыть сокет
close SOCKET or croak "close: $OS_ERROR";
__END__
```

С модулем `script` можно использовать сценарии, написанные на любом языке.

can_reach как модуль

Теперь реализуем `can_reach` в виде полноценного модуля Ansible на Python, который можно вызвать так:

```
- name: Check if host can reach the database
  can_reach:
    host: example.com
    port: 5432
    timeout: 1
```

Так можно проверить доступность порта 5432 на хосте *example.com*. Если соединение установить невозможно, через секунду будет зафиксирована ошибка превышения тайм-аута.

Мы будем пользоваться этим примером на протяжении всей главы.

Когда следует разрабатывать модули?

Прежде чем приступать к разработке модуля, желательно ответить на несколько простых вопросов. Модуль действительно предлагает какие-то новые возможности? Существуют ли похожие модули? Может быть, лучше использовать или разработать плагин? Можно ли ту же задачу решить с помощью простой роли? Может быть, лучше создать коллекцию вместо единственного модуля? Намного проще использовать существующий код или имеющиеся возможности Ansible, чем программировать на Python. Если вы разрабатываете Python API для своего продукта, то тогда имеет смысл разработать коллекцию для него. Модули могут входить в состав коллекций, как обсуждалось в главе 15.

Где хранить свои модули

Ansible ищет модули в каталоге *library*, находящемся рядом со сценарием Ansible. В нашем примере сценарии Ansible хранятся в каталоге *playbooks*, поэтому свой модуль мы сохраним в файле *playbooks/library/can_reach*. `ansible-playbook` просматривает каталог *library* по умолчанию, но если вам нужно использовать модуль в специализированных командах Ansible, то добавьте в *ansible.cfg* следующую строку:

```
library = library
```

Модули также можно сохранять в подкаталоге *library* в ролях или в коллекциях Ansible. В имени файла модуля можно использовать расширение *.py* или другое, соответствующее выбранному языку сценариев.

Как Ansible вызывает модули

Прежде чем реализовать модуль, давайте посмотрим, как Ansible вызывает их. Для этого Ansible:

1) генерирует автономный сценарий на Python с аргументами (только модули на Python);
2) копирует модуль на хост;
3) создает файл аргументов на хосте (только для модулей не на языке Python);
4) вызывает модуль на хосте, передавая ему файл с аргументами;
5) анализирует стандартный вывод модуля.

Разберем каждый шаг более детально.

Генерация автономного сценария на Python с аргументами (только модули на Python)

Если модуль написан на Python и использует вспомогательный код, предоставляемый системой Ansible (описан ниже), то Ansible сгенерирует автономный сценарий на Python со встроенным вспомогательным кодом и аргументами модуля.

Копирование модуля на хост

Сгенерированный сценарий на Python (для модулей на Python) или локальный файл *playbooks/library/can_reach* (для модулей не на языке Python) копируется во временный каталог на удаленном хосте. Если соединение с удаленным хостом устанавливается от имени пользователя `vagrant`, то Ansible сохранит файл в каталоге, путь к которому выглядит примерно так:

/home/vagrant/.ansible/tmp/ansible-tmp-1412459504.14-47728545618200/can_reach.

Создание файла с аргументами на хосте (для модулей не на языке Python)

Если модуль написан не на языке Python, Ansible создаст на удаленном хосте файл, путь к которому выглядит примерно так:

/home/vagrant/.ansible/tmp/ansible-tmp-1412459504.14-47728545618200/arguments.

Если вызвать модуль, как показано ниже:

```
- name: Check if host can reach the database server
  can_reach:
    host: db.example.com
    port: 5432
    timeout: 1
```

то файл аргументов будет содержать следующую информацию:

```
host=db.example.com port=5432 timeout=1
```

Можно потребовать от Ansible сгенерировать файл аргументов в формате JSON, добавив следующую строку в *playbooks/library/can_reach*:

```
# WANT_JSON
```

В этом случае содержимое файла с аргументами будет выглядеть так:

```
{"host": "www.example.com", "port": "80", "timeout": "1"}
```

Вызов модуля

Ansible вызовет модуль и передаст ему файл с аргументами. Если модуль написан на Python, то Ansible выполнит эквивалент следующей команды (заменив `/path/to/` действительным путем к каталогу):

```
/path/to/can_reach
```

Если модуль написан на другом языке, то Ansible определит интерпретатор по первой строке в модуле и выполнит эквивалент следующей команды:

```
/path/to/interpreter /path/to/can_reach /path/to/arguments
```

Если предположить, что модуль `can_reach` реализован как сценарий Bash и начинается со строки `#!/bin/bash`, то Ansible выполнит такую команду:

```
/bin/bash /path/to/can_reach /path/to/arguments
```

Но это только приближенный эквивалент. *На самом деле* Ansible произведет ряд сложных манипуляций – заключит модуль в команду оболочки, задаст региональные настройки и предусмотрит удаление модуля после выполнения:

```
/bin/sh -c 'LANG=en_US.UTF-8 LC_CTYPE=en_US.UTF-8 /bin/bash /path/to/can_reach \
/path/to/arguments; rm -rf /path/to/ >/dev/null 2>&1'
```

Точную команду, которую выполняет Ansible, можно увидеть, передав параметр `-vvv` утилите `ansible-playbook`.

В Drbian может потребоваться задать следующие региональные настройки:

```
localedef -i en_US -f UTF-8 en_US.UTF-8
```

Порядок запуска модулей на Python удаленно во многом зависит от особенностей командной оболочки. Обратите внимание, что Ansible не использует ограниченные командные оболочки.

Ожидаемый вывод

Ansible ожидает, что модуль выведет результат в формате JSON. Например:

```
{"changed": false, "failed": true, "msg": "could not reach the host"}
```

Как вы увидите ниже, если модуль написан на Python, то Ansible предоставляет вспомогательные методы, облегчающие вывод информации в JSON.

Ожидаемые выходные переменные

Модуль может выводить любые переменные, однако Ansible определяет специальные правила для возвращаемых переменных:

changed

Все модули Ansible должны возвращать переменную `changed`. По этой логической переменной Ansible определяет факт изменения состояния хоста модулем. Если в задаче имеется выражение `notify` для уведомления обработчика, то уведомление будет отправлено, только если `changed` имеет значение `true`.

failed

Если модуль потерпел неудачу, он должен вернуть `"failed": true`. Ansible расценит попытку выполнения такой задачи неудачной и прервет

выполнение последующих задач на хосте, кроме случая, когда задача содержит выражение `ignore_errors` или `failed_when`.

Если модуль выполнился успешно, он должен вернуть `"failed": false` или вообще опустить эту переменную.

msg

Переменную `msg` можно использовать для вывода сообщения с причиной неудачи выполнения модуля.

Если задача потерпела неудачу и модуль вернул переменную `msg`, то Ansible выведет значение этой переменной, хотя и в несколько ином виде. Например, если модуль вернул:

```
{"failed": true, "msg": "could not reach www.example.com:81"}
```

то Ansible выведет:

```
failed: [fedora] ==> {"failed": true}
msg: could not reach www.example.com:81
```

Если на одном хосте модуль потерпит неудачу, то Ansible продолжит работу с другими хостами, где модуль выполнится успешно.

Реализация модулей на Python

Для модулей на Python Ansible предоставляет класс `AnsibleModule`, упрощающий анализ входной информации, вывод результатов в формате JSON и вызов сторонних программ.

Фактически, обрабатывая модули на Python, Ansible внедряет аргументы непосредственно в сгенерированный код, избавляя от необходимости анализировать файл с аргументами. Подробнее об этом мы поговорим далее в этой главе.

Давайте создадим модуль на Python и сохраним его в файле *can_reach*. Сначала рассмотрим полную реализацию, а потом обсудим ее (см. пример 19.3).

Пример 19.3. *can_reach*

```
#!/usr/bin/env python3
""" модуль can_reach для ansible """
from ansible.module_utils.basic import AnsibleModule                    ❶

def can_reach(module, host, port, timeout):
    """ метод can_reach устанавливает tcp-соединение с помощью nc """
    nc_path = module.get_bin_path('nc', required=True)                  ❷
    args = [nc_path, "-z", "-w", str(timeout), host, str(port)]
    # (return_code, stdout, stderr) = module.run_command(args)
    return module.run_command(args,check_rc=True)                       ❸
```

```
def main():
    """ модуль для ansible, использующий netcat
        для проверки возможности соединения """
    module = AnsibleModule(                                          ❹
        argument_spec=dict(                                          ❺
            host=dict(required=True),
            port=dict(required=True, type='int'),                    ❻
            timeout=dict(required=False, type='int', default=3)      ❼
        ),
        supports_check_mode=True                                     ❽
    )

    # В режиме проверки никаких действий не выполняется
    # Так как этот модуль не изменяет состояния хоста, он просто
    # возвращает changed=False
    if module.check_mode:                                            ❾
        module.exit_json(changed=False)                              ❿
    host = module.params['host']                                     ⓫
    port = module.params['port']
    timeout = module.params['timeout']

    if can_reach(module, host, port, timeout)[0] == 0:
        msg = "Could reach %s:%s" % (host, port)
        module.exit_json(changed=False, msg=msg)                     ⓬
    else:
        msg = "Could not reach %s:%s" % (host, port)
        module.fail_json(msg=msg)                                    ⓭

if __name__ == "__main__":
    main()
```

❶ Импорт вспомогательного класса `AnsibleModule`.
❷ Получение пути к внешней программе.
❸ Вызов внешней программы.
❹ Создание экземпляра класса `AnsibleModule`.
❺ Определение допустимого набора аргументов.
❻ Обязательный аргумент.
❼ Необязательный аргумент со значением по умолчанию.
❽ Признак, что модуль поддерживает режим проверки.
❾ Проверка запуска модуля в режиме проверки.
❿ Успешное завершение, передает возвращаемое значение.
⓫ Извлечение аргументов.
⓬ Выход с признаком успеха, возвращает сообщение об успехе.
⓭ Выход с признаком ошибки, возвращает сообщение с описанием ошибки.

Анализ аргументов

Гораздо проще понять, как AnsibleModule выполняет анализ аргументов, на примере. Напомню, что наш модуль вызывается, как показано ниже:

```
- name: Check if host can reach the database server
  can_reach:
    host: db.example.com
    port: 5432
    timeout: 1
```

Предположим, параметры host и port являются обязательными, а timeout – нет, со значением по умолчанию 3 с.

Создадим экземпляр AnsibleModule, передав словарь argument_spec, ключи которого соответствуют именам параметров, а значения являются словарями с информацией о параметрах:

```
module = AnsibleModule(
    argument_spec=dict(
        ...
```

В примере 19.2 мы объявили аргумент host обязательным. Ansible выдаст ошибку, если забыть передать его в вызов задачи:

```
host=dict(required=True),
```

Параметр timeout является необязательным. Ansible считает, что в аргументах передаются строки, кроме случаев, когда заявлено иное. Переменная timeout – целое число. Ее тип определяется как int, чтобы Ansible могла автоматически преобразовать значение в число Python. Если параметр timeout не задан, модуль установит его равным 3:

```
timeout=dict(required=False, type='int', default=3)
```

Конструктор AnsibleModule принимает также другие аргументы, кроме argument_spec. В предыдущем примере мы добавили аргумент:

```
supports_check_mode = True
```

Он сообщает, что модуль поддерживает режим проверки. Мы рассмотрим его далее в этой главе.

Доступ к параметрам

После объявления объекта AnsibleModule появляется возможность доступа к значениям аргументов через словарь params:

```
module = AnsibleModule(...)
host = module.params["host"]
port = module.params["port"]
timeout = module.params["timeout"]
```

Импортирование вспомогательного класса AnsibleModule

Ansible развертывает модули на хостах, передавая их в файлах ZIP, включающих также вспомогательные файлы. Как следствие, есть возможность явно импортировать классы, например:

```
from ansible.module_utils.basic import AnsibleModule
```

Свойства аргументов

Каждый аргумент модуля Ansible имеет несколько свойств, перечисленных в табл. 19.1.

Таблица 19.1. Свойства аргументов

Свойство	Описание
`required`	Если `True`, то аргумент считается обязательным
`default`	Значение по умолчанию для необязательного аргумента
`choices`	Список допустимых значений для аргумента
`deprecated_aliases`	Кортеж или список словарей с именами, версиями, датами, именами коллекций
`aliases`	Другие имена, которые можно использовать как псевдонимы этого аргумента
`type`	Тип аргумента.
`elements`	Если тип определен как список, то его элементы определяют типы элементов списка
`fallback`	Кортеж с функцией и списком параметров для передачи ей
`no_log`	Значение `True` запрещает журналирование поведения модуля
`options`	Реализует возможность создания составных аргументов в виде словарей[2]
`mutually_exclusive`	Список взаимоисключающих аргументов
`required_together`	Список аргументов, которые должны передаваться вместе
`required_one_of`	Список аргументов, из которых хотя бы один должен передаваться модулю
`required_if`	Последовательность последовательностей
`required_by`	Словарь, отображающий имена параметров в последовательность имен параметров

required

Свойство `required` – единственное, которое всегда должно определяться. Если его значение равно `True`, Ansible сообщит об ошибке при попытке вызвать модуль без этого аргумента.

В примере модуля can_reach аргументы host и port являются обязательными, а timeout – нет.

default

Для аргументов с required=False необходимо определить в этом свойстве значение по умолчанию. В нашем примере:

```
timeout=dict(required=False, type='int', default=3)
```

Если пользователь попытается вызвать модуль так:

```
can_reach: host=www.example.com port=443
```

то аргумент module.params["timeout"] автоматически получит значение 3.

choices

Свойство choices позволяет ограничить значения аргумента предопределенным списком, как в случае с аргументом distros в следующем примере:

```
distro=dict(required=True, choices=['ubuntu', 'centos', 'fedora'])
```

Если пользователь попробует передать в аргументе значение, отсутствующее в списке, например:

```
distro=debian
```

то Ansible выведет сообщение об ошибке.

aliases

Свойство aliases позволяет использовать другие имена для обращения к аргументу. Например, рассмотрим аргумент package в модуле apt:

```
module = AnsibleModule(
    argument_spec=dict(
        ...
        package = dict(default=None, aliases=['pkg', 'name'], type='list'),
    )
)
```

Поскольку pkg и name являются псевдонимами аргумента package, следующие вызовы модуля эквиваленты:

```
- apt:
    package: vim

- apt:
    name: vim

- apt:
    pkg: vim
```

type

Свойство `type` дает возможность объявить тип аргумента. По умолчанию Ansible считает, что аргументы являются строками.

Однако вы можете явно объявить тип аргумента, и Ansible преобразует аргумент в желаемый формат. Поддерживаются следующие типы:

- `str`
- `list`;
- `dict`;
- `bool`;
- `int`;
- `float`;
- `path`;
- `raw`;
- `jsonarg`;
- `json`;
- `bytes`;
- `bits`.

В нашем примере мы объявили аргумент `port` с типом `int`:

```
port=dict(required=True, type='int'),
```

При обращении к нему через словарь `params`:

```
port = module.params['port']
```

мы получим переменную `port` с целым числом. Если бы мы не объявили тип аргумента как `int` в момент объявления свойства `port`, то ссылка `module.params['port']` вернула бы строку, а не целое число.

Элементы в списках разделяются запятой. Например, если представить, что у нас есть модуль `foo` с аргументом `colors`, принимающим список:

```
colors=dict(required=True, type='list')
```

то мы должны будем передавать в нем список, как показано ниже:

```
foo: colors=red,green,blue
```

Передавать словари можно в виде списков пар `ключ=значение`, разделенных запятыми, либо в формате JSON.

Например, пусть имеется модуль `bar` с аргументом `tags` типа `dict`:

```
tags=dict(required=False, type='dict', default={})
```

В этом случае аргумент `tags` можно передать так:

```
- bar: tags=env=staging,function=web
```

или так:

```
- bar: tags={"env": "staging", "function": "web"}
```

Для обозначения списков и словарей, которые передаются модулям в качестве аргументов, в официальной документации Ansible используется термин *составные аргументы* (complex args). Порядок передачи сценариям аргументов этих типов описывается в разделе «Короткое отступление: составные аргументы задач» главы 7.

AnsibleModule: параметры метода инициализатора

Метод-инициализатор класса `AnsibleModule` принимает несколько параметров (табл. 19.2). Единственным обязательным параметром является `argument_spec`.

Таблица 19.2. Аргументы инициализатора AnsibleModule

Параметр	По умолчанию	Описание
`argument_spec`	*(Нет)*	Словарь с информацией об аргументах
`bypass_checks`	`False`	Если `True`, то не проверяет никаких ограничений для параметров
`no_log`	`False`	Если `True`, то не журналирует поведения этого модуля
`check_invalid_arguments`	`True`	Если `True`, то возвращает ошибку при попытке вызвать модуль с неопознанным аргументом
`mutually_exclusive`	*(Нет)*	Список взаимоисключающих аргументов
`required_together`	*(Нет)*	Список аргументов, которые должны передаваться вместе
`required_one_of`	*(Нет)*	Список аргументов, из которых хотя бы один должен передаваться модулю
`add_file_common_args`	`False`	Поддержка аргументов модуля `file`
`supports_check_mode`	`False`	Если `True`, то модуль поддерживает режим проверки

argument_spec

Словарь, содержащий описания всех допустимых аргументов модуля, как рассказывалось в предыдущем разделе.

no_log

Когда модуль выполняется на хосте, он выводит информацию о работе в журнал syslog, находящийся в Ubuntu в каталоге */var/log/syslog*.

Вывод выглядит следующим образом:

```
Aug 29 18:55:05 ubuntu-focal python3[5688]: ansible-lineinfile Invoked with
dest=/etc/ssh/sshd_config.d/10-crypto.conf regexp=^HostKeyAlgorithms line=
state=present path=/etc/ssh/sshd_config.d/10-crypto.conf backrefs=False
create=False backup=False firstmatch=False unsafe_writes=False
search_string=None insertafter=None insertbefore=None validate=None
mode=None owner=None group=None seuser=None serole=None selevel=None
setype=None attributes=None
Aug 29 18:55:05 ubuntu-focal python3[5711]: ansible-stat Invoked with
path=/etc/ssh/ssh_host_ed25519_key follow=False get_md5=False
get_checksum=True get_mime=True get_attributes=True checksum_algorithm=sha1
Aug 29 18:55:06 ubuntu-focal python3[5736]: ansible-file Invoked with
path=/etc/ssh/ssh_host_ed25519_key mode=384 recurse=False force=False
follow=True modification_time_format=%Y%m%d%H%M.%S
access_time_format=%Y%m%d%H%M.%S unsafe_writes=False state=None
_original_basename=None _diff_peek=None src=None modification_time=None
access_time=None owner=None group=None seuser=None serole=None selevel=None
setype=None attributes=None
Aug 29 18:55:06 ubuntu-focal python3[5759]: ansible-lineinfile Invoked with
dest=/etc/ssh/sshd_config regexp=^HostKey /etc/ssh/ssh_host_ed25519_key
line=HostKey /etc/ssh/ssh_host_ed25519_key insertbefore=^# HostKey
/etc/ssh/ssh_host_rsa_key mode=384 state=present path=/etc/ssh/sshd_config
backrefs=False create=False backup=False firstmatch=False
unsafe_writes=False search_string=None insertafter=None validate=None
owner=None group=None seuser=None serole=None selevel=None setype=None
attributes=None
```

Если модуль принимает конфиденциальную информацию в аргументах, то предпочтительнее отключить журналирование. Для отключения записи в *syslog* передайте в инициализатор `AnsibleModule` параметр `no_log=True`.

check_invalid_arguments

По умолчанию Ansible проверяет допустимость всех аргументов, передаваемых модулю. Эту проверку можно отключить, передав в инициализатор `AnsibleModule` параметр `check_invalid_arguments=False`.

mutually_exclusive

Параметр `mutually_exclusive` содержит список аргументов, которые нельзя одновременно передавать модулю. Например, модуль `lineinfile` позволяет добавить строку в файл. Ему можно передать аргумент `insertbefore` со строкой для вставки перед указанной или аргумент `insertafter` со строкой для вставки после указанной. Но нельзя передать сразу оба аргумента.

Поэтому модуль определяет эти два аргумента как взаимоисключающие:

```
mutually_exclusive=[['insertbefore', 'insertafter']]
```

required_one_of

Параметр `required_one_of` определяет список аргументов, из которых хотя бы один должен быть передан модулю. Например, модуль `pip`, используемый для установки пакетов Python, может принять либо аргумент `name` с именем пакета, либо аргумент `requirements` с именем файла, содержащим список пакетов. Необходимость передачи хотя бы одного из аргументов определена в модуле так:

```
required_one_of=[['name', 'requirements']]
```

add_file_common_args

Многие модули создают или модифицируют файлы. Пользователю часто требуется установить некоторые атрибуты конечного файла, такие как владелец, группа и разрешения.

Установку этих атрибутов можно произвести с помощью модуля `file`:

```
- name: Download a file
  get_url:
    url: http://www.example.com/myfile.dat
    dest: /tmp/myfile.dat

- name: Set the permissions
  file:
    path: /tmp/myfile.dat
    owner: vagrant
    mode: '0600'
```

Ansible позволяет указать, что модуль принимает все те же аргументы, что и модуль `file`. Благодаря этому можно потребовать установить атрибуты файла, просто передав соответствующие аргументы модулю, который создает или изменяет файлы. Например:

```
- name: Download a file
  get_url:
    url: http://www.example.com/myfile.dat
    dest: /tmp/myfile.dat
    owner: vagrant
    mode: '0600'
```

Чтобы объявить поддержку модулем этих аргументов, необходимо передать параметр:

```
add_file_common_args=True
```

Класс `AnsibleModule` предоставляет вспомогательные методы для обработки перечисленных параметров.

Метод `load_file_common_arguments` принимает словарь с параметрами и возвращает словарь параметров со всеми аргументами, соответствующими установленным атрибутам файла.

Метод `set_fs_attributes_if_different` принимает словарь с параметрами и логический флаг как признак изменения состояния хоста. Метод устанавливает атрибуты файла и возвращает `true`, если состояние хоста изменилось (либо входной аргумент-флаг имел значение `true`, либо выполнено изменение файла как побочный эффект).

Если вы используете общие аргументы для установки атрибутов файлов, не определяйте их явно. Для доступа к этим аргументам и установки атрибутов файла используйте вспомогательные методы:

```
module = AnsibleModule(
    argument_spec=dict(
        dest=dict(required=True),
        ...
    ),
    add_file_common_args=True
)

# "changed" получит значение True, если модуль изменил состояние хоста
changed = do_module_stuff(param)

file_args = module.load_file_common_arguments(module.params)

changed = module.set_fs_attributes_if_different(file_args, changed)
module.exit_json(changed=changed, ...)
```

Ansible предполагает, что модуль имеет аргумент `path` или `dest`, содержащий путь к файлу. К сожалению, не все модули определяют эти параметры, поэтому в случае сомнений проверьте:

```
$ ansible-doc module
```

bypass_checks

Прежде чем запустить модуль, Ansible проверит, все ли аргументы удовлетворяют ограничениям, и, если какое-то ограничение нарушено, сообщит об ошибке. Проверка считается пройденной, если:

- нет взаимоисключающих аргументов;
- переданы все аргументы, отмеченные как `required`;

- аргументы со свойством choices имеют допустимые значения;
- аргументы с заданным типом type имеют соответствующие значения;
- аргументы со свойством required_together используются совместно;
- передан хотя бы один аргумент из списка required_one_of.

Все эти проверки можно запретить, установив bypass_checks=True.

Возврат признака успешного завершения или неудачи

Чтобы сообщить об успешном завершении, используйте метод exit_json. Вы всегда должны возвращать флаг changed, и хорошей практикой считается возвращать msg с осмысленным сообщением:

```
module = AnsibleModule(...)
...
module.exit_json(changed=False, msg="meaningful message goes here")
```

Для вывода сообщения о неудаче используйте метод fail_json. Всегда возвращайте сообщение msg, объясняющее причины неудачи:

```
module = AnsibleModule(...)
...
module.fail_json(msg="Out of disk space")
```

Вызов внешних команд

Класс AnsibleModule предоставляет метод run_command для вызова внешних программ, который использует модуль Python subprocess. Он принимает аргументы, перечисленные в табл. 19.3.

Таблица 19.3. Аргументы *run_command*

Аргумент	Тип	По умолчанию	Описание
args (по умолчанию)	Строка или список строк	(Нет)	Команда для выполнения (см. следующий раздел)
check_rc	Логический	False	Если True, производит вызов fail_json, когда команда возвращает ненулевое значение
close_fds	Логический	True	Передает как аргумент close_fds в вызов subprocess.Popen
executable	Строка (путь к программе)	(Нет)	Передает как аргумент executable в вызов subprocess.Popen
data	Строка	(Нет)	Посылается в стандартный ввод дочернего процесса

Аргумент	Тип	По умолчанию	Описание
`binary_data`	Логический	`False`	Если `False` и присутствует `data`, то Ansible передаст символ перевода строки в стандартный ввод после `data`
`path_prefix`	Строка (список путей)	(Нет)	Список путей, разделенных двоеточиями, для добавления перед содержимым переменной окружения `PATH`
`cwd`	Строка (путь к каталогу)	(Нет)	Если определен, то Ansible перейдет в этот каталог перед запуском
`use_unsafe_shell`	Логический	`False`	См. следующий раздел

Если `args` передается как список (см. пример 19.4), то Ansible вызовет `subprocess.Popen` с параметром `shell=False`.

Пример 19.4. Передача *args* со списком

```
module = AnsibleModule(...)
...
module.run_command(['/usr/local/bin/myprog', '-i', 'myarg'])
```

Если в `args` передать строку, как показано в примере 19.5, то поведение будет зависеть от значения `use_unsafe_shell`. Если `use_unsafe_shell=false`, то Ansible разобьет `args` на список и вызовет `subprocess.Popen` с параметром `shell=False`. Если `use_unsafe_shell=true`, то Ansible передаст `args` в `subprocess.Popen` в виде строки с `shell=True`[1].

Пример 19.5. Передача args со строкой

```
module = AnsibleModule(...)
...
module.run_command('/usr/local/bin/myprog -i myarg')
```

Режим проверки (пробный прогон)

Ansible поддерживает специальный *режим проверки*, который включается при передаче команде `ansible-playbook` параметра `-C` или `--check`. По своей сути он похож на режим *пробного прогона*, который поддерживают многие другие инструменты.

При выполнении в режиме проверки сценарий не производит на хосте никаких изменений, а просто сообщает, какие задачи могут изменить состояние хоста, возвращая признак успешного выполнения без внесения изменений или сообщение об ошибке.

[1] За дополнительной информацией о классе `subprocess.Popen` в стандартной библиотеке Python обращайтесь к документации (https://oreil.ly/trNKm).

Модуль должен явно поддерживать режим проверки. Если вы собираетесь написать свой модуль, то рекомендую добавить в него поддержку режима проверки, чтобы он был добропорядочным гражданином Ansible.

Чтобы сообщить Ansible, что модуль поддерживает режим проверки, передайте методу-инициализатору класса `AnsibleModule` параметр `supports_check_mode` со значением `True`, как показано в примере 19.6.

Пример 19.6. Уведомление Ansible о поддержке режима проверки

```
module = AnsibleModule(
    argument_spec=dict(...),
    supports_check_mode=True)
```

Модуль должен определить режим проверкой значения атрибута `check_mode` объекта `AnsibleModule`, как показано в примере 19.7, и вызвать метод `exit_json` или `fail_json`, как обычно.

Пример 19.7. Проверка режима

```
module = AnsibleModule(...)
...if module.check_mode:
   # проверить, мог бы модуль внести изменения
   would_change = would_executing_this_module_change_something()
   module.exit_json(changed=would_change)
```

Как автор модуля, вы должны также гарантировать, что в режиме проверки ваш модуль не изменит состояния хоста.

Документирование модуля

В соответствии со стандартами проекта Ansible модули обязательно должны документироваться, чтобы HTML-документация по модулю генерировалась корректно и программа `ansible-doc` могла отобразить ее. Ansible использует особый синтаксис YAML для документирования модулей.

Ближе к началу модуля определите строковую переменную `DOCUMENTATION` с описанием и строковую переменную `EXAMPLES` с примерами использования. Если модуль возвращает информацию в формате JSON, то отразите это в переменной `RETURN`.

В примере 19.8 приводится раздел с документацией для модуля `can_reach`.

Пример 19.8. Пример модуля с документацией

```
DOCUMENTATION = r'''
---
```

```
module: can_reach
short_description: Проверяет доступность сервера
description: Проверяет возможность подключения к удаленному серверу
version_added: "1.8"
options:
  host:
    description:
      - Имя хоста или IP-адрес
    required: true
  port:
    description:
      - Номер порта TCP
    required: true
  timeout:
    description:
      - Длительность попытки (в секундах) установить соединение,
        прежде чем она будет объявлена неудачной
    required: false
    default: 3
requirements: [nmap]
author: Lorin Hochstein, Bas Meijer
notes:
  - Это просто пример, демонстрирующий, как писать модули.
  - Возможно, вы предпочтете использовать встроенный модуль M(wait_for).
'''
EXAMPLES = r'''
# Проверить, запущен ли сервер ssh с тайм-аутом по умолчанию
- can_reach: host=localhost port=22 timeout=1
# Проверить, запущен ли сервер postgres с тайм-аутом по умолчанию
- can_reach: host=example.com port=5432
'''
```

В документации допускается использовать рудиментарную разметку. В табл. 19.4 описывается синтаксис разметки, поддерживаемой инструментом вывода документации, а также советы по ее использованию.

Таблица 19.4. Разметка в документации

Тип	Пример синтаксиса	Когда использовать
URL	`U(http://www.example.com)`	Для отображения адресов URL
Модуль	`M(apt)`	Имена модулей
Курсив	`I(port)`	Имена параметров
Моноширинный	`C(/bin/bash)`	Имена файлов и параметров

Существующие модули Ansible являются превосходным источником примеров для документирования.

Отладка модуля

В репозитории Ansible на GitHub имеется пара сценариев, позволяющих запускать модули непосредственно на локальной машине, без использования команды `ansible` или `ansible-playbook`.

Клонируйте репозиторий Ansible:

```
$ git clone https://github.com/ansible/ansible.git
```

Перейдите в корневой каталог клонированного репозитория:

```
$ cd ansible
```

Создайте виртуальное окружение:

```
$ python3 -m venv venv
```

Активируйте виртуальное окружение:

```
$ source venv/bin/activate
```

Установите необходимые зависимости:

```
$ python3 -m pip install --upgrade pip
$ pip install -r requirements.txt
```

Запустите сценарий настройки окружения разработки:

```
$ source hacking/env-setup
```

Вызовите модуль:

```
$ ansible/hacking/test-module -m /path/to/can_reach -a "host=example.com port=81"
```

Поскольку на *example.com* нет службы, обслуживающей порт 81, модуль завершится с ошибкой и вернет сообщение:

```
* including generated source, if any, saving to:
/Users/bas/.ansible_module_generated
* ansiballz module detected; extracted module source to:
/Users/bas/debug_dir
********************************
RAW OUTPUT

{"cmd": "/usr/bin/nc -z -v -w 3 example.com 81", "rc": 1, "stdout": "",
"stderr": "nc: connectx to example.com port 81 (tcp) failed: Operation timed
out\n", "failed": true, "msg": "nc: connectx to example.com port 81 (tcp)
failed: Operation timed out", "invocation": {"module_args": {"host":
"example.com", "port": 81, "timeout": 3}}}

********************************
PARSED OUTPUT
{
```

```
        "cmd": "/usr/bin/nc -z -v -w 3 example.com 81",
        "failed": true,
        "invocation": {
                "module_args": {
                        "host": "example.com",
                        "port": 81,
                        "timeout": 3
                }
        },
        "msg": "nc: connectx to example.com port 81 (tcp) failed: Operation
timed out",
        "rc": 1,
        "stderr": "nc: connectx to example.com port 81 (tcp) failed: Operation
timed out\n",
        "stdout": ""
}
```

Как следует из полученного сообщения, при запуске `test-module` Ansible сгенерирует сценарий на Python и скопирует его в *~/.ansible_module_generated*. Это автономный сценарий на Python, который можно использовать непосредственно.

Начиная с версии Ansible 2.1.0, этот сценарий на Python включает содержимое ZIP-файла в формате base64 с исходным кодом вашего модуля, а также код для распаковки этого ZIP-файла и выполнения кода внутри него.

Этот файл не принимает никаких аргументов – все необходимые аргументы Ansible встраивает непосредственно в файл, в переменную `ANSIBALLZ_PARAMS`:

```
ANSIBALLZ_PARAMS = '{"ANSIBLE_MODULE_ARGS": {"_ansible_selinux_special_fs":
["fuse", "nfs", "vboxsf", "ramfs", "9p", "vfat"], "_ansible_tmpdir":
"/Users/bas/.ansible/tmp/ansible-local-12753r6nenhh",
"_ansible_keep_remote_files": false, "_ansible_version": "2.12.0.dev0",
"host": "example.com", "port": "81"}}'
```

Изучение особенностей отладки модулей в Ansible поможет вам лучше понять, как работает система, даже если вы не собираетесь писать свои модули.

Создание модуля на Bash

Если вы собираетесь создавать свои модули для Ansible, я советую писать их на Python, потому что, как мы видели выше в этой главе, для таких модулей Ansible предоставляет вспомогательные классы. Для управления системами Windows используются модули, написанные на PowerShell. Однако при желании модули можно писать на других язы-

ках. Это может потребоваться, например, если модуль зависит от сторонней библиотеки, не реализованной на Python. Или, может быть, модуль настолько прост, что его проще написать на Bash.

В этом разделе мы рассмотрим пример реализации модуля в виде сценария на Bash. Он будет очень похож на реализацию в примере 19.1. Главное отличие – анализ входных аргументов и генерация вывода, который ожидает получить Ansible.

Модули на Bash и сокращенный синтаксис ввода

В модулях на Bash можно использовать сокращенный синтаксис ввода. Но я не рекомендую использовать этот прием, потому что он предполагает использование встроенной команды source, что несет потенциальную угрозу безопасности. Однако если вы настроены решительно, то прочитайте статью «Shell scripts as Ansible modules» («Сценарии на языке командной оболочки в качестве модулей Ansible») по адресу **https://oreil.ly/A11X6**, написанную Яном-Питом Менсом (Jan-Piet Mens):

```
source ${1}  # Очень, *очень*, опасно!
```

Мы используем формат JSON для передачи входных аргументов и инструмент `jq` (**http://stedolan.github.io/jq/**) для парсинга JSON в командной строке. Это значит, что для запуска модуля на хосте придется установить `jq`. В примере 19.9 приводится полная реализация модуля на Bash.

Пример 19.9. Модуль *can_reach* на Bash

```
#!/bin/bash -e
# WANT_JSON

# Чтение переменных из файла с помощью jq
host=$(jq -r .host <"$1")
port=$(jq -r .port <"$1")
timeout=$(jq -r .timeout <"$1")

# По умолчанию timeout=3
if [[ $timeout = null ]]; then
    timeout=3
fi

# Проверить достижимость хоста
if nc -z -w "$timeout" "$host" "$port"; then
    echo '{"changed": false}'
else
    echo "{\"failed\": true, \"msg\": \"could not reach $host:$port\"}"
fi
```

Мы добавили в комментарий `WANT_JSON`, чтобы Ansible знала, что входные данные должны передаваться в формате JSON. Майкл Де-Хаан (Michael DeHaan) называет такой код JSON «рудиментарным»; в 2013 году он написал: «Ansible имеет рудиментарную поддержку JSON, распознавая только список пар ключ/значение, поэтому технически нельзя передавать данные в полноценном формате JSON».

Альтернативное местоположение интерпретатора Bash

Обратите внимание: модуль предполагает, что интерпретатор Bash находится в */bin/bash*. Однако не во всех системах выполняемый файл интерпретатора находится именно там. Мы можем предложить Ansible проверить наличие интерпретатора Bash в других каталогах, определив переменную `ansible_bash_interpreter` на хостах, где он может устанавливаться в другие каталоги.

Например, допустим, что у нас имеется хост *fileserver.example.com* с ОС FreeBSD, где интерпретатор Bash доступен как */usr/local/bin/bash*. Создав файл *host_vars/fileserver.example.com* со следующим содержимым:

```
ansible_bash_interpreter: /usr/local/bin/bash
```

можно создать переменную хоста.

В таком случае, когда Ansible будет запускать модуль на хосте *fileserver.example.com*, она использует */usr/local/bin/bash* вместо */bin/bash*.

Выбор интерпретатора системой Ansible определяется поиском символов `#!` и просмотром базового имени первого элемента. В нашем примере Ansible найдет строку:

```
#!/bin/bash
```

извлечет из */bin/bash* базовое имя, т. е. *bash*. Затем использует переменную `ansible_bash_interpreter`, если она задана пользователем.

Учитывая, как Ansible определяет местоположение интерпретатора, если в строке `#!` указать вызов команды */usr/bin/env*, например `#!/usr/bin/env bash`, то Ansible ошибочно определит интерпретатор как `env`, потому что вызовет `basename` для анализа пути */usr/bin/env*, чтобы определить интерпретатор.

Поэтому, чтобы не попасть впросак, не вызывайте `env` в строке `#!`, точно указывайте путь к интерпретатору и переопределяйте его с помощью переменной `ansible_bash_interpreter` (или ее аналога), если это необходимо.

Заключение

В этой главе мы узнали, как писать модули на Python и на других языках, а также как можно избежать необходимости писать свои модули, использовав модуль `script`. Если вы все-таки решите взяться за написание модуля, я рекомендую прочитать руководство разработчика модулей (**https://oreil.ly/YCSdz**). Лучший способ овладеть искусством создания модулей – углубиться в чтение исходного кода модулей, входящих в состав Ansible, на GitHub (**https://oreil.ly/G4CUl**).

Глава 20

Ускорение работы Ansible

Начав использовать Ansible на регулярной основе, у вас быстро появится желание ускорить работу сценариев. В этой главе мы обсудим стратегии сокращения времени, которое требуется Ansible для выполнения сценариев.

Мультиплексирование SSH и ControlPersist

Дочитав книгу до этой главы, вы уже знаете, что в качестве основного транспортного механизма Ansible использует протокол SSH. В частности, по умолчанию Ansible использует именно SSH.

Поскольку протокол SSH работает поверх протокола TCP, вам потребуется установить новое TCP-соединение с удаленной машиной. Клиент и сервер должны выполнить начальную процедуру установки соединения, прежде чем начать выполнять какие-то фактические действия. Эта процедура занимает некоторое время. Для каждого отдельного хоста и для каждой задачи это время невелико, но если хостов и/или задач много, то суммарное время может оказаться внушительным.

Во время выполнения сценариев Ansible устанавливает достаточно много SSH-соединений, например для копирования файлов или выполнения команд. Каждый раз Ansible устанавливает новое SSH-соединение с хостом.

OpenSSH – наиболее распространенная реализация SSH и SSH-клиент по умолчанию, который установлен на вашей локальной машине, если вы работаете в Linux или macOS. OpenSSH поддерживает вид оптимизации с названием *мультиплексирование каналов SSH*, который также называют *ControlPersist*. Когда используется мультиплексирование, несколько SSH-сеансов с одним и тем же хостом использует одно и то же TCP-соединение, т. е. TCP-соединение устанавливается лишь однажды.

Когда активируется мультиплексирование:

- при первом подключении к хосту OpenSSH устанавливает основное соединение;

- OpenSSH создает сокет домена Unix (известный как управляющий сокет), связанный с удаленным хостом;
- при следующем подключении к хосту вместо нового TCP-подключения OpenSSH использует контрольный сокет.

Основное соединение остается открытым в течение заданного пользователем интервала времени, а затем закрывается SSH-клиентом. По умолчанию Ansible устанавливает интервал, равный 60 с.

Включение мультиплексирования SSH вручную

Ansible включает мультиплексирование SSH автоматически. Но, чтобы вы понимали, что за этим стоит, включим его вручную и соединимся с удаленной машиной посредством SSH.

В примере 20.1 показаны настройки мультиплексирования из файла *~/.ssh/config*.

Пример 20.1. Включение мультиплексирования в ssh/config

```
ControlMaster auto
ControlPath ~/.ssh/sockets/%r@%h:%p
ControlPersist 10m
```

Строка `ControlMaster auto` включает мультиплексирование SSH и сообщает клиенту SSH о необходимости создать основное соединение и управляющий сокет, если они еще не существуют.

Строка `ControlPersist 10m` требует от SSH разорвать основное соединение, если в течение 10 минут не производилось попыток создать SSH-подключение.

Строка `ControlPath ~/.ssh/sockets/%r@%h:%p` сообщает клиенту SSH, где расположить файл сокета домена Unix в файловой системе.

- `%l` – имя локального хоста, включая доменное имя;
- `%h` – имя целевого хоста;
- `%p` – номер порта;
- `%r` – имя пользователя на удаленном хосте;
- `%C` – соответствует хешу `%l%h%p%r`.

Если соединение осуществляется от имени пользователя `vagrant`:

```
$ ssh -i ~/.vagrant.d/insecure_private_key vagrant@192.168.56.10.nip.io
```

в этом случае SSH создаст файл управляющего сокета *~/.ssh/sockets/vagrant@192.168.56.10.nip.io:22* при первом подключении к серверу. В аргументах для `ControlPath` можно использовать символ тильды (~) для ссылки на домашний каталог пользователя. Мы советуем передавать команде `ControlPath` передавать по меньшей мере `%h`, `%p` и `%r` (или `%C`) и помещать файл сокета в каталог, доступный для записи другим пользо-

вателям. Это гарантирует уникальную идентификацию совместно используемых соединений.

Проверить состояние основного соединения можно с помощью параметра `-O check`:

```
$ ssh -O check vagrant@192.168.56.10.nip.io
```

Если основное соединение активно, эта команда вернет:

```
Master running (pid=5099)
```

Вот так выглядит основной управляющий процесс в выводе команды `ps 5099`:

```
PID   TT  STAT      TIME COMMAND
5099  ??  Ss     0:00.00 ssh: /Users/bas/.ssh/sockets/vagrant@192.168.56.10.
nip.io:22 [mux]
```

Разорвать основное соединение можно с помощью параметра `-O exit`:

```
$ ssh -O exit vagrant@192.168.56.10.nip.io
```

Больше деталей об этих настройках можно найти на странице *ssh_config* руководства *man*:

```
$ man 5 ssh_config
```

Мы протестировали скорость создания SSH-соединений. Следующая команда вернет время, которое требуется для создания SSH-подключения и выполнения программы */usr/bin/true*, которая всегда завершается с кодом 0:

```
$ time ssh -i ~/.vagrant.d/insecure_private_key \
    vagrant@192.168.56.10.nip.io \
    /usr/bin/true
```

Когда мы первый раз запустили ее, результат выглядел так[1]:

```
real 0m0.319s
user 0m0.018s
sys 0m0.011s
```

Наибольший интерес представляет общее время: `0m0.319s total`. Этот результат говорит о том, что на выполнение всей команды потребовалось 0,319 с. (Общее время иногда также называют *астрономическим временем*, поскольку оно показывает, сколько прошло времени, как если бы его измеряли по настенным часам.)

Во второй раз результат выглядел так:

```
real 0m0.010s
user 0m0.004s
sys 0m0.006s
```

[1] Формат результата может отличаться в зависимости от командной оболочки и ОС. Мы использовали Bash в macOS.

Общее время сократилось до 0,010 с, т. е. экономия составляет примерно 0,3 с для каждого SSH-соединения, начиная со второго. Напомним, что для выполнения задачи Ansible открывает, по крайней мере, два SSH-сеанса: один – для копирования файла модуля на хост, второй – для запуска модуля на хосте[1]. Это означает, что мультиплексирование может сэкономить порядка одной или двух секунд на каждой задаче в сценарии.

Параметры мультиплексирования SSH в Ansible

В табл. 20.1 перечислены параметры мультиплексирования SSH, используемые в Ansible.

Таблица 20.1. Параметры мультиплексирования SSH в Ansible

Параметр	Значение
`ControlMaster`	`auto`
`ControlPath`	`~/.ssh/sockets/%r@%h:%p`
`ControlPersist`	`60s`

Нам никогда не приходилось изменять значение по умолчанию `ControlMaster`. `ControlPersist=10m` уменьшает расходы на создание сокетов, но если ваш ноутбук уйдет в спящий режим в то время, когда активно мультиплексирование, то потребуется дополнительное время на восстановление после выхода из спящего режима.

На практике нам *приходилось* изменять только значение `ControlPath`, потому что операционная система устанавливает максимальную длину пути к файлу сокета домена Unix. Если строка в `ControlPath` окажется слишком длинной, мультиплексирование не будет работать. К сожалению, система Ansible не сообщает, если строка в `ControlPath` превысит это ограничение, она просто не будет использовать мультиплексирование SSH.

Управляющую машину можно протестировать вручную, устанавливая SSH-соединение с помощью того же значения `ControlPath`, что использует Ansible:

```
$ CP=~/.ansible/cp/ansible-ssh-%h-%p-%r
$ ssh -o ControlMaster=auto -o ControlPersist=60s \
    -o ControlPath=$CP \
    ubuntu@ec2-203-0-113-12.compute-1.amazonaws.com \
    /bin/true
```

[1] Один из этих шагов можно оптимизировать, использовав конвейерный режим, описанный далее в этой главе.

Если строка `ControlPath` окажется слишком длинной, вы увидите сообщение об ошибке, как показано в примере 20.2.

Пример 20.2. Слишком длинная строка *ControlPath*

```
"/Users/lorin/.ansible/cp/ansible-ssh-ec2-203-0-113-12.compute-1.amazonaws.
com-22-ubuntu.KIwEKEsRzCKFABch"
too long for Unix domain socket
```

Это обычное дело при подключении к экземплярам Amazon EC2, которым назначаются длинные имена хостов.

Решить проблему можно настройкой использования более коротких строк в `ControlPath`. Официальная документация (**https://oreil.ly/V6qpw**) рекомендует так определять этот параметр в файле *ansible.cfg*:

```
[ssh_connection]
control_path = %(directory)s/%%h-%%r
```

Ansible заменит `%(directory)s` на `$HOME/.ansible/cp` (двойной знак процента (`%%`) необходим для экранирования, потому что знак процента в файлах *.ini* является специальным символом).

При изменении конфигурации SSH-соединения, например параметра `ssh_args`, когда мультиплексирование уже включено, такое изменение не вступит в силу, пока управляющий сокет остается открытым с прошлого подключения.

Еще о настройке SSH

При подготовке множества серверов или для наблюдения за их безопасностью часто желательно оптимизировать настройки SSH на клиентах и серверах. Протокол SSH поддерживает несколько алгоритмов установки соединений, аутентификации клиентов и серверов и настройки параметров сеансов. Для установки соединений требуется время, и разные алгоритмы работают с разной скоростью и уровнем безопасности. Если вы используете Ansible для управления серверами ежедневно, то, вероятно, имеет смысл рассмотреть настройки SSH более подробно.

Рекомендации по выбору алгоритмов

Основные дистрибутивы Linux распространяются с «совместимой» конфигурацией серверов SSH. Она позволяет любому подключиться и авторизоваться на сервере, используя любое программное обеспечение клиента, если он знает учетные данные пользователя. Стоит вдумчиво поразмышлять – действительно ли это то, что вам нужно!

Бас исследовал скорость установки SSH-соединений Ansible, изменяя порядок параметров и их значения в `ssh_args`, и пришел к выводу, что большинство из них уже оптимизированы. Однако Бас нашел два значения в `ssh_args`, экономящие несколько микросекунд в комбинации с параметрами мультиплексирования, обсуждавшимися выше:

```
ssh_args = -4 -o PreferredAuthentications=publickey
```

Значение `-4` выбирает семейство протоколов inet (ipv4), а `PreferredAuthentications` перенаправляет аутентификацию пользователя на сокет `ssh-agent`.

В `sshd_config` Бас сначала выбирает самый быстрый алгоритм и добавляет несколько безопасных альтернатив для совместимости, но в обратном порядке для скорости.

Чтобы еще немного увеличить скорость, Бас изменил типы пар ключей на современный стандарт. Криптографическая эллиптическая кривая Curve25519 быстрее и безопаснее, чем RSA (**https://oreil.ly/7KzzL**), поэтому он использует ее с `PublicKeyAuthentication` и для ключей хоста.

Генерируя пару ключей на своей машине, Бас использует параметр `-a 100` для защиты от атаки методом перебора:

```
$ ssh-keygen -t ed25519 -a 100 -C bas
```

Следующая задача гарантирует, что со своим ключом Бас будет иметь исключительный доступ к учетной записи пользователя `deploy`:

```
- name: Change ssh key to ed25519
  authorized_key:
    user: deploy
    key: "{{ lookup('file', '~/.ssh/id_ed25519.pub') }}"
    exclusive: true
```

Следующие задачи обеспечат создание и настройку ключей хоста:

```
- name: Check the ed25519 host key
  stat:
    path: /etc/ssh/ssh_host_ed25519_key
  register: ed25519

- name: Generate ed25519 host key
  command: ssh-keygen -t ed25519 -f /etc/ssh/ssh_host_ed25519_key -N ""
  when:
    - not ed25519.stat.exists|bool
  notify: Restart sshd
  changed_when: true

- name: Set permissions
  file:
    path: /etc/ssh/ssh_host_ed25519_key
```

```
    mode: '0600'

- name: Configure ed25519 host key
  lineinfile:
    dest: /etc/ssh/sshd_config
    regexp: '^HostKey /etc/ssh/ssh_host_ed25519_key'
    line: 'HostKey /etc/ssh/ssh_host_ed25519_key'
    insertbefore: '^# HostKey /etc/ssh/ssh_host_rsa_key'
    mode: '0600'
    state: present
  notify: Restart sshd
```

Бас также проверяет соответствие конфигурации его сервера SSH с конфигурацией клиента, поэтому первый этап согласования при установке соединения происходит с использованием совместимых алгоритмов для обеих сторон. Оптимизация конфигурационных параметров на клиенте не так сильно повышает производительность, как на стороне сервера, потому что эти файлы читаются перед установкой каждого соединения SSH.

Конвейерный режим

Вспомним, как Ansible выполняет задачу:

- генерирует сценарий на Python, основанный на вызываемом модуле,
- копирует его на хост,
- запускает его там.

Ansible поддерживает прием оптимизации – *конвейерный режим*, – объединяя открытие сеанса SSH с запуском сценария на Python. Конвейерный режим, если поддерживается плагином `connection`, уменьшает количество сетевых операций, выполняя множество модулей Ansible без фактического копирования файлов. Ansible выполняет сценарии на Python, объединяя их в сеансе SSH. Экономия достигается за счет того, что в этом случае требуется открыть только один сеанс SSH вместо двух.

Включение конвейерного режима

По умолчанию конвейерный режим не используется, потому что требует настройки удаленных хостов, но нам нравится использовать его, поскольку он ускоряет процесс. Чтобы включить этот режим, внесите изменения в файл *ansible.cfg*, как показано в примере 20.3.

Пример 20.3. *ansible.cfg*, включение конвейерного режима

```
[connection]
pipelining = True
```

Настройка хостов для поддержки конвейерного режима

Для поддержки конвейерного режима необходимо убедиться, что на хостах в файле */etc/sudoers* выключен параметр `requiretty`. Иначе при выполнении сценария вы будете получать ошибки, как показано в примере 20.4.

Пример 20.4. Ошибка при включенном параметре *requiretty*

```
failed: [centos] ==> {"failed": true, "parsed": false}
invalid output was: sudo: sorry, you must have a tty to run sudo
```

Если утилита `sudo` на хостах настроена на чтение файлов из каталога */etc/sudoers.d*, тогда самое простое решение – добавить файл конфигурации *sudoers*, выключающий ограничение `requiretty` для пользователя, с именем которого вы устанавливаете SSH-соединения.

Если каталог */etc/sudoers.d* существует, хосты должны поддерживать добавление файлов конфигурации *sudoers*. Проверить наличие каталога можно с помощью утилиты `ansible`:

```
$ ansible vagrant -a «file /etc/sudoers.d»
```

Если каталог имеется, вы увидите примерно такие строки:

```
centos | CHANGED | rc=0 >>
/etc/sudoers.d: directory
ubuntu | CHANGED | rc=0 >>
/etc/sudoers.d: directory
fedora | CHANGED | rc=0 >>
/etc/sudoers.d: directory
debian | CHANGED | rc=0 >>
/etc/sudoers.d: directory
```

Если каталог отсутствует, то вы увидите:

```
vagrant3 | FAILED | rc=1 >>
/etc/sudoers.d: ERROR: cannot open `/etc/sudoers.d' (No such file or
directory)
vagrant2 | FAILED | rc=1 >>
/etc/sudoers.d: ERROR: cannot open `/etc/sudoers.d' (No such file or
directory)
vagrant1 | FAILED | rc=1 >>
/etc/sudoers.d: ERROR: cannot open `/etc/sudoers.d" (No such file or
directory)
```

Если каталог имеется, создайте файл шаблона, как показано в примере 20.5.

Пример 20.5. *templates/disable-requiretty.j2*

```
Defaults:{{ ansible_user }} !requiretty
```

Затем запустите сценарий, приведенный в примере 20.6, заменив `vagrant` именами ваших хостов. Не забудьте выключить конвейерный режим, прежде чем сделать это, иначе сценарий завершится с ошибкой.

Пример 20.6. *disable-requiretty.yml*

```
---
- name: Do not require tty for ssh-ing user
  hosts: vagrant
  become: true

  tasks:
    - name: Set a sudoers file to disable tty
      template:
        src: disable-requiretty.j2
        dest: /etc/sudoers.d/disable-requiretty
        owner: root
        group: root
        mode: '0440'
        validate: 'bash -c "cat /etc/sudoers /etc/sudoers.d/* %s | visudo -cf-"'
...
```

Проверка достоверности файлов

Модули `copy` и `template` поддерживают выражение `validate`. Оно позволяет указать программу для проверки файла, сгенерированного системой Ansible. Используйте `%s` вместо имени файла. Например:

```
validate: 'bash -c "cat /etc/sudoers /etc/sudoers.d/* %s|visudo -cf-"'
```

При наличии выражения `validate` Ansible скопирует файл сначала во временный каталог, а потом запустит указанную программу проверки. Если программа завершится успешно (0), то Ansible скопирует файл из временного каталога в постоянное местоположение. Если программа вернет результат, отличный от нуля, то Ansible выведет сообщение об ошибке:

```
SSH | 367
failed: [myhost] ==> {"checksum": "ac32f572f0a670c3579ac2864cc3069ee8a19588",
"failed": true}
msg: failed to validate: rc:1 error:
FATAL: all hosts have already failed -- aborting
```

Поскольку ошибки в файлах *sudoers* и в файлах, созданных в */etc/sudoers.d*, могут нарушить доступ к привилегиям пользователя root, их всегда полезно проверить с помощью программы `visudo`. Для понимания проблем, которые несут файлы *sudoers*, мы рекомендуем прочитать статью участника проекта Ansible Жан-Пита Мэна (Jan-Piet Men) «Don't try this at the office: /etc/sudoers» (**https://oreil.ly/B9H0n**).

Mitogen для Ansible

Mitogen – это сторонняя библиотека для Python, используемая для разработки распределенных самокопирующихся программ. Mitogen для Ansible – это совершенно новый механизм передачи и выполнения модулей в UNIX для Ansible. Он требует минимум настроек и замещает медленную и расточительную реализацию Ansible на основе командной оболочки эквивалентами на Python, использующими высокоэффективные вызовы удаленных процедур, туннелируемые через SSH.

Обратите внимание, что на момент написания этих строк библиотека Mitogen поддерживала только Ansible 2.9; более поздние версии не поддерживались. На целевых хостах никаких изменений вносить не требуется, но на управляющей машине Ansible нужно установить библиотеку Mitogen:

```
$ pip3 install --user mitogen
```

Вот как выглядит настройка использования Mitogen в виде плагина стратегии в файле *ansible.cfg*:

```
[defaults]
strategy_plugins = /path/to/strategy
strategy = mitogen_linear
Fact Caching
```

Кеширование фактов

Факты о серверах содержат все переменные, какие только могут пригодиться в сценарии Ansible. Сбор фактов производится в самом начале выполнения сценария и требует времени, поэтому это еще один кандидат на оптимизацию. Один из вариантов – создать локальный кеш с фактами; другой – вообще отключить сбор фактов.

Если операция не использует факты Ansible, то их сбор можно отключить с помощью выражения `gather_facts`. Например:

```
- name: An example play that doesn't need facts
  hosts: myhosts
  gather_facts: false
  tasks:
    # здесь находятся задачи:
```

Также можно отключить сбор фактов по умолчанию, добавив в файл *ansible.cfg*:

```
[defaults]
gathering = explicit
```

Если есть операции, использующие факты, их сбор можно организовать так, что Ansible будет делать это для каждого хоста только однажды,

даже если этот же или другой сценарий будет запускаться неоднократно для того же самого хоста.

Если кеширование фактов включено, Ansible сохранит факты в кеше, полученные после первого подключения к хостам. В последующих попытках выполнить сценарий Ansible будет извлекать факты из кеша, не обращаясь к удаленным хостам. Такое положение вещей сохраняется до истечения времени хранения кеша.

В примере 20.7 приводятся строки, которые необходимо добавить в файл *ansible.cfg* для включения кеширования фактов. Значение `fact_caching_timeout` выражается в секундах, в примере используется тайм-аут, равный 24 ч (86 400 с).

Как это всегда бывает с решениями, использующими кеширование, существует опасность, что кешированные данные окажутся неактуальными. Некоторые факты, такие как архитектура CPU (факт `ansible_architecture`), редко изменяются. Другие, такие как дата и время, сообщаемые машиной (факт `ansible_date_time`), гарантированно изменяются очень часто.

Если вы решили включить кеширование фактов, то обязательно проверьте, как часто изменяются факты, используемые вашим сценарием, и задайте соответствующее значение тайм-аута кеширования. Чтобы очистить кеш перед запуском сценария, передайте утилите `ansible-playbook` параметр `--flush-cache`.

Пример 20.7. *ansible.cfg*. Включение кеширования фактов

```
[defaults]
gathering = smart
# кеш остается действительным 24 часа, измените, если необходимо
fact_caching_timeout = 86400
# Обязательно укажите реализацию кеширования фактов
fact_caching = ...
```

Значение `smart` в параметре `gathering` сообщает, что необходимо использовать *интеллектуальный сбор фактов* (smart gathering). То есть Ansible будет собирать факты, только если они отсутствуют в кеше или срок хранения кеша истек. Механизм кеширования базируется на плагинах, и список этих плагинов можно получить командой:

```
$ ansible-doc -t cache -l
```

Необходимо явно указать реализацию кеширования `fact_caching` в *ansible.cfg*, иначе кеширование не будет использоваться. На момент написания книги имелись три реализации:

- в файлах JSON, YAML, Pickle;
- в оперативной памяти (недолговечное хранилище);
- в базах данных NoSQL: Redis, Memcached, MongoDB.

На практике чаще всего используется кеширование в Redis.

Если вы собираетесь использовать кеширование фактов, убедитесь, что в сценариях отсутствует выражение `gather_facts: true` или `gather_facts: false`. Когда включен режим интеллектуального сбора фактов, факты будут собираться, только если они отсутствуют в кеше.

Кеширование фактов в файлах JSON

Реализация кеширования фактов в файлах JSON записывает собранные факты в файлы на управляющей машине. Если файлы присутствуют в вашей системе, Ansible будет использовать их вместо соединений с хостами.

Чтобы задействовать реализацию кеширования фактов в файлах JSON, добавьте в файл *ansible.cfg* настройки, как показано в примере 20.8.

Пример 20.8. *ansible.cfg*, включение кеширования фактов в файлах JSON

```
[defaults]
gathering = smart
# кеш остается действительным 24 часа, измените, если необходимо
fact_caching_timeout = 86400
# кешировать в файлах JSON
fact_caching = jsonfile
fact_caching_connection = /tmp/ansible_fact_cache
```

Параметр `fact_caching_connection` определяет каталог, куда Ansible будет сохранять файлы JSON с фактами. Если каталог отсутствует, Ansible создаст его.

Для определения тайм-аута кеширования Ansible использует время модификации файла. Вариант кеширования с использованием файлов JSON – самый простой, но имеет ограниченное применение в многопользовательских окружениях или когда имеется несколько управляющих машин из-за сложностей с назначением прав доступа к этим файлам или выбором их местоположения.

Кеширование фактов в Redis

Redis – популярное хранилище данных типа ключ/значение, часто используемое в качестве кеша. Для кеширования фактов в Redis необходимо:

1) установить Redis на управляющей машине,
2) убедиться, что служба Redis запущена на управляющей машине,
3) установить пакет Redis для Python,
4) включить кеширование в Redis в файле *ansible.cfg*.

В примере 20.9 показано, какие настройки следует добавить в *ansible.cfg*, чтобы организовать кеширование в Redis.

Пример 20.9. *ansible.cfg*, кеширование фактов в Redis

```
[defaults]
gathering = smart
# кеш остается действительным 24 часа, измените, если необходимо
fact_caching_timeout = 86400

fact_caching = redis
```

Для работы с хранилищем Redis требуется установить пакет Redis для Python на управляющей машине, например с помощью `pip`[1]:

```
$ pip install redis
```

Вы также должны установить программное обеспечение Redis и запустить его на управляющей машине. В macOS Redis можно установить с помощью диспетчера пакетов Homebrew. В Linux это можно сделать с помощью системного диспетчера пакетов.

Кеширование фактов в Memcached

Memcached – еще одно популярное хранилище данных типа ключ/значение, которое также часто используется в качестве кеша. Для кеширования фактов в Memcached необходимо:

1) установить Memcached на управляющей машине,
2) убедиться, что служба Memcached запущена на управляющей машине,
3) установить пакет Memcached для Python,
4) включить кеширование в Memcached в файле *ansible.cfg*.

В примере 20.10 показано, какие настройки следует добавить в *ansible.cfg*, чтобы организовать кеширование в Memcached.

Пример 20.10. *ansible.cfg*, кеширование фактов в Memcached

```
[defaults]
gathering = smart
# кеш остается действительным 24 часа, измените, если необходимо
```

[1] Может потребоваться выполнить команду `sudo` или активировать virtualenv, в зависимости от способа установки Ansible на управляющей машине.

```
fact_caching_timeout = 86400
fact_caching = memcached
```

Для работы с хранилищем Memcached требуется установить пакет Memcached для Python на управляющей машине, например с помощью `pip`. Может потребоваться выполнить команду `sudo` или активировать virtualenv, в зависимости от способа установки Ansible на управляющей машине.

```
$ pip install python-memcached
```

Вы также должны установить программное обеспечение Memcached и запустить его на управляющей машине. В macOS Memcached можно установить с помощью диспетчера пакетов Homebrew. В Linux это можно сделать с помощью системного диспетчера пакетов.

Более полную информацию о кешировании фактов можно найти в официальной документации.

Параллелизм

Для каждой задачи Ansible устанавливает соединения сразу с несколькими хостами и запускает на них одну и ту же задачу параллельно. Однако Ansible необязательно будет устанавливать соединения сразу *со всеми* хостами – уровень параллелизма контролируется параметром со значением по умолчанию, равным 5. Изменить его можно одним из двух способов .

Можно настроить переменную окружения `ANSIBLE_FORKS`, как это показано в примере 20.11.

Пример 20.11. Настройка *ANSIBLE_FORKS*

```
$ export ANSIBLE_FORKS=8
$ ansible-playbook playbook.yml
```

Можно также изменить настройки в конфигурационном файле Ansible (*ansible.cfg*), определив параметр `forks` в секции `default`, как показано в примере 20.12. Бас считает, что существует прямая связь между количеством ядер процессора на управляющей машине Ansible и оптимальным значением `forks`: если задать слишком большое число, то затраты на переключение контекста превзойдут выгоды от параллельного выполнения задач. Я устанавливаю значение 8 на моей 8-ядерной машине. Немаловажную роль играет также объем доступной оперативной памяти на управляющей машине. Чем больше экземпляров задачи выполняется параллельно, тем больше памяти требуется управляющему процессу. В промышленных окружениях обычно используются значения от 25 до 50, которые, конечно же, зависят еще и от общего числа хостов.

Пример 20.12. *ansible.cfg*. Настройка параллелизма

```
[defaults]
forks = 8
```

Асинхронное выполнение задач с помощью async

В Ansible появилось новое выражение `async`, позволяющее выполнять асинхронные действия и обходить проблемы с тайм-аутами SSH. Если время выполнения задачи превышает тайм-аут SSH, то Ansible закроет соединение с хостом и сообщит об ошибке. Если добавить в определение такой задачи выражение `async`, это устранит риск истечения тайм-аута SSH.

Однако механизм поддержки асинхронных действий можно также использовать для других целей, например чтобы запустить вторую задачу до завершения первой. Это может пригодиться, например, если обе задачи выполняются очень долго и не зависят друг от друга (т. е. нет нужды ждать, пока завершится первая, чтобы запустить вторую).

В примере 20.13 показан список задач, в котором имеется задача с выражением `async`, выполняющая клонирование большого репозитория Git. Так как задача отмечена как асинхронная, Ansible не будет ждать завершения клонирования репозитория и продолжит установку системных пакетов.

Пример 20.13. Использование async для параллельного выполнения задач

```
- name: Install git
  become: true
  apt:
    name: git
    update_cache: true

- name: Clone Linus's git repo
  git:
    repo: git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
    dest: /home/vagrant/linux
  async: 3600                                    ❶
  poll: 0                                        ❷
  register: linux_clone                          ❸

- name: Install several packages
  apt:
    name:
      - apt-transport-https
      - ca-certificates
      - linux-image-extra-virtual
      - software-properties-common
```

```
      - python-pip
    become: true

  - name: Wait for linux clone to complete
    async_status:                                    ❹
      jid: "{{ linux_clone.ansible_job_id }}"        ❺
    register: result
    until: result.finished                           ❻
    retries: 3600
```

❶ Задача объявляется асинхронной и что она должна выполняться не дольше 3600 с. Если время выполнения задачи превысит это значение, Ansible автоматически завершит процесс, связанный с задачей.

❷ Значение 0 в аргументе `poll` сообщает системе Ansible, что она может сразу перейти к следующей задаче после запуска этой. Если указать ненулевое значение, то Ansible не сможет перейти к следующей задаче. Вместо этого она периодически будет опрашивать состояние асинхронной задачи, ожидая ее завершения, приостанавливаясь между проверками на интервал времени, указанный в параметре `poll` (в секундах).

❸ Когда имеется асинхронная задача, необходимо добавить выражение `register`, чтобы захватить результат ее выполнения. Объект `result` содержит значение `ansible_job_id`, которое можно использовать позднее для проверки состояния задания.

❹ Для опроса состояния асинхронного задания используется модуль `async_status`.

❺ Для идентификации асинхронного задания необходимо указать значение `jid`.

❻ Модуль `async_status` выполняет опрос только один раз. Чтобы продолжить опрос до завершения задания, нужно указать выражение `until` и определить значение `retries` максимального числа попыток.

Заключение

Теперь вы знаете, как настроить мультиплексирование SSH, конвейерный режим, кеширование фактов, а также параллельное и асинхронное выполнения задач, чтобы ускорить выполнение сценария. Далее мы обсудим сетевые возможности Ansible и безопасность.

Глава 21

Сети и безопасность

Управление сетевыми устройствами

Управление сетевыми устройствами и их настройка всегда вызывает у нас ностальгические чувства. Вход с консоли через `telnet`, ввод нескольких команд, сохранение конфигурации – и работа сделана. Долгое время мы использовали две основные стратегии управления сетевыми устройствами:

- приобретение дорогостоящего патентованного программного обеспечения для настройки этих устройств;
- разработку минималистского набора инструментов для управления конфигурационными файлами: копирования файлов в локальную систему, внесения некоторых изменений путем редактирования и копирования их обратно в устройство.

Однако в последние несколько лет ситуация стала заметно меняться. Первое, что мы заметили, – производители сетевых устройств стали создавать или открывать свои API. Во-вторых, так называемое движение DevOps не остановилось и продолжило спуск по стеку к ядру: аппаратные серверы, балансировщики нагрузки, устройства защиты сетей, сетевые устройства и даже роутеры. Начиная с версии Ansible 2.5, компания Red Hat стала координировать применение Ansible для автоматизации управления сетевыми устройствами. Между версиями 2.5 и 2.9 основное внимание уделялось развитию сетевых модулей. Но затем из-за сложности поддержки от этой идеи отказались в пользу *коллекций*. По итогам обсуждения, как отмечается в блоге JP Mens (**https://oreil.ly/DizNw**), было решено (**https://oreil.ly/MW1Ie**) основной команде Ansible сосредоточиться на развитии ядра `ansible-core`, создание сертифицированного контента делегировать партнерам Red Hat, а все остальное – сообществу. Производители сетевого оборудования поддержали эту тенденцию, так как она дает им возможность выпускать такой контент независимо.

Список поддерживаемых производителей сетевого оборудования

Первый вопрос, который вы, скорее всего, зададите: «Поддерживается ли выбранный мной производитель сетевого оборудования или операционной системы?» Список коллекций, предлагаемых производителями, длинный и очень динамичный, чтобы приводить его в книге, но желающие смогут найти его по адресу **https://oreil.ly/CEHsD**. Пространство имен Community содержит большое количество инструментов, разработанных независимо от производителей. Кроме того, `ansible.netcommon` предлагает абстракции, которые можно использовать с разными поставщиками, что также означает их согласованность и продуманность (и это здорово). Вот неполный список производителей, предлагающих свои коллекции:

- Arista (**https://oreil.ly/AsBf2**);
- Checkpoint (**https://oreil.ly/sLvpl**);
- Cisco ACI (**https://oreil.ly/TNOAT**);
- Cisco Meraki (**https://oreil.ly/gExAe**);
- Cyberark (**https://oreil.ly/vMQse**);
- F5 Networks (**https://oreil.ly/GcDFd**);
- Fortinet (**https://oreil.ly/R1sDM**);
- IBM (**https://oreil.ly/fiiWQ**);
- Infoblox (**https://oreil.ly/yCcpH**);
- Juniper (**https://oreil.ly/Js4de**);
- Vyos (**https://oreil.ly/MnTbI**).

Некоторые из этих производителей предлагают виртуальные устройства для использования с Vagrant. Файл Vagrantfile для этой главы в репозитории книги включает устройства junos, nxosv и vyos.

Старайтесь явно определять имена используемых модулей автоматизации управления сетевыми устройствами из установленных коллекций. Указывайте полные имена этих модулей, включающие имена коллекций, чтобы не возникло путаницы с модулями, входящими в состав ядра Ansible. При исследовании файлов задач или сценариев ищите имена модулей с точками, такие как `cisco.iosxr.iosxr_l2_interfaces`.

Ansible Connection для автоматизации управления сетевыми устройствами

Ansible позволяет управлять самыми разными сетевыми устройствами, но есть некоторые отличия в управлении машинами с Windows, ma-

cOS или Linux. Системы Linux повсеместно управляются через SSH, а компьютерами Windows можно управлять через соединение WinRM. Из других типов соединений, которые мы использовали до сих пор, можно назвать `local`, `docker` и `raw`. Использование REST с модулем `uri` не поддерживается параметром `ansible_connection`, потому что это «соединение» не позволяет использовать другие модули.

Поскольку сетевые устройства не поддерживают возможность выполнения сценариев на Python, для управления ими нужна другая парадигма. Инструменты автоматизации работают на управляющей машине и общаются с API сетевых устройств. В заголовке сценария автоматизации управления сетевыми устройствами обычно можно встретить такую строку:

```
hosts: localhost
```

Значение в `ansible_connection` на узле, управляющем устройством, зависит от платформы и назначения используемых модулей. Транспортным протоколом может служить SSH или HTTP/HTTPS. Соединения HTTPS обычно используются для доступа к REST API, тогда как SSH позволяет взаимодействовать с интерфейсом командной строки, как это делают модули `command` и `shell` в «обычном» Ansible. XML через SSH применяется для конфигурации сети (`netconf`). Чтобы использовать этот тип соединений, нужно установить библиотеку `ncclient` для Python.

Привилегированный режим

Некоторые сетевые устройства поддерживают разделение между обычным пользовательским и *привилегированным режимами*, последний из которых предусматривается для выполнения критически важных задач и доступен через параметр `ansible_become: true`. Обратите внимание, что здесь вместо утилиты `sudo`, известной нам в Linux, используется метод, который называется `enable`. Мы предпочитаем задавать параметр `become` в начале задачи, прямо под ее именем, чтобы потом было проще анализировать ее поведение.

Настраивать соединения Ansible для разных типов устройств можно с помощью нескольких параметров. Естественный выбор для регистрации этих параметров – блок `vars` в реестре. Помимо протокола соединения, системе Ansible нужно сообщить операционную систему сетевого устройства, как показано в файле реестра в примере 21.1.

Пример 21.1. *playbooks/inventory/hosts*

```
[arista:vars]
# https://galaxy.ansible.com/arista/eos
ansible_connection=ansible.netcommon.httpapi
ansible_network_os=arista.eos.eos
```

```
ansible_become_method=enable

[cisco:vars]
# https://galaxy.ansible.com/cisco/ios
ansible_connection=ansible.netcommon.network_cli
ansible_network_os=cisco.ios.ios
ansible_become_method=enable

[junos:vars]
# https://galaxy.ansible.com/junipernetworks/junos
ansible_connection=ansible.netcommon.netconf
ansible_network_os=junipernetworks.junos.junos
ansible_become_method=enable
```

Реестр сетевых устройств

Мы предпочитаем определять файлы реестров и динамические реестры для облачных окружений и Vagrant в простом формате INI, однако формат YAML лучше подходит для определения реестров крупных и иерархических сетевых топологий (пример 21.2). Лучшей практикой в моделировании является определение ответов на основные вопросы: что это такое? где это? кому принадлежит? и когда пройдут этапы разработки, тестирования, пилотного проекта, обкатки и передачи в производство?

Пример 21.2. Реестр в формате YAML

```
backbone:
  hosts:
    rt_dc1_noc_p:
      ansible_host: 10.31.1.1
  vars:
    ansible_connection: ansible.netcommon.network_cli
    ansible_network_os: cisco.ios.ios
    ansible_become_method: enable

perimeter:
  hosts:
    proxy_dc1_soc_p:
      ansible_host: 10.31.2.1
    vars:
      ansible_become_method: sudo

network:
  children:
    backbone:
    perimeter:
```

С помощью следующей команды можно представить реестр в виде графика, чтобы оценить его:

```
ansible-inventory -i inventory/hosts.yml --graph
```

Примеры использования автоматизации управления сетевыми устройствами

Распространенное мнение, что для проектирования долговечной инфраструктуры корпоративной сети достаточно создать тщательно проработанную схему, было опровергнуто в последние десятилетия общей энтропией: вы легко сможете назвать несколько противников стабильности, поразмышляв о развитии ИТ, разрушительной конкуренции, глобальных кризисах и нестабильности рынка. Организации должны быстро адаптироваться к меняющимся условиям, а это подразумевает изменения – постоянные изменения – и гибкость.

Идея о том, что многофункциональные команды могут работать автономно и достигать бизнес-целей, используя собственные облачные технологии, приобретаемые независимо, беспокоит сетевые центры и центры управления безопасностью (мягко говоря). Ansible способна анализировать состояние всех устройств и хостов и собирать факты, необходимые для управления конфигурацией и предоставления информации о текущей ситуации. Она может настраивать устройства, автоматизировать обновления и проверять, как работают все устройства. В целом механизм автоматизации управления сетевыми устройствами в Ansible – это большой шаг вперед по сравнению с настройкой устройств вручную.

Безопасность

Каждая организация предъявляет свои уникальные требования к безопасности. Существует несколько базовых уровней безопасности, таких как CIS (**https://oreil.ly/4oGAp**), DISA-STIG (**https://oreil.ly/UQ3f0**), PCI (**https://oreil.ly/eM8aP**), HIPAA (**https://oreil.ly/CVYED**), NIST (**https://oreil.ly/mq03N**) и FedRAMP (**https://www.fedramp.gov/**), применяемых в различных отраслях в США, включая платежные карты, здравоохранение, федеральное правительство и оборонные производства. В Европе существуют свои национальные институты, такие как BSI Germany (**https://oreil.ly/jyRtY**), BSI UK (**https://oreil.ly/RNXOj**) и NCSC (**https://oreil.ly/pBdtl**), публикующие рекомендации по защите компьютеров и их сетевых соединений. Если ваше правительство не определило стандарт безопасности, то вы можете ознакомиться с примерами, представленными фондами программного обеспечения, такими как Mozilla (**https://oreil.ly/vzWsX**).

Еще до того, как Red Hat купила Ansible, Inc., существовала возможность обеспечить соблюдение определенных базовых стандартов

безопасности. В 2015 году Ansible, Inc. поручила координацию проекта `ansible-lockdown` (**https://oreil.ly/0lzC8**) с открытым исходным кодом компании MindPointGroup[1], специализирующейся на решениях безопасности. С тех пор много воды утекло. Этот контент частично переместился из PDF-документов и электронных таблиц в сценарии Ansible. И теперь автоматизация безопасности стала одной из областей, в которой Ansible набирает силу.

Применение Ansible для настройки безопасности таких систем, как сетевые устройства, кластеры и хосты, кажется отличной идеей. Разделение задач – один из принципов теории управления, поэтому на практике вам потребуется инструмент сканирования для оценки уровня защиты на основе выбранного профиля безопасности.

Центр интернет-безопасности (Center for Internet Security) поддерживает эталонные тесты проверки защищенности для широкого спектра операционных систем и промежуточного программного обеспечения, которые детально исследуют конфигурации. Существуют сканеры безопасности, распространяемые на коммерческой основе. OpenSCAP (**https://oreil.ly/l4EiB**) бесплатно публикует руководство по безопасности (**https://oreil.ly/3oMj6**), изучив которое вы сможете выбрать профиль, подходящий для вашей отрасли, чтобы тщательно просканировать системы RHEL и проверить их соответствие требованиям. Знаете ли вы, что можно даже сгенерировать сценарий Ansible, поддерживаемый Red Hat, для устранения отклонений? (Это действительно круто!) На GitHub можно найти и другие проекты по усилению защиты от независимых разработчиков, например DevSec Project (**https://dev-sec.io/project**) из Германии.

Соблюдение требований соответствия

Однако, даже имея эти инструменты, остается нерешенным вопрос, заданный Томпсоном (Thompson): кому доверять[2]? Углубившись в детали, можно обнаружить еще больше вопросов. следует ли доверять руководству по настройке безопасности с помощью Ansible больше, чем результатам сканирования производителя? Можно ли утверждать, что соответствие требованиям равно безопасности? Ограничивают ли национальные стандарты круг криптографических методов для применения в вашей стране (**https://oreil.ly/68zJp**)? Как влияют на ваши решения в области безопасности методы слежения, обнаружения вторжений, выявления вредоносных программ, законодательство об интеллектуальной собственности, гражданских правах, трудовых отношениях, а также профсоюзы и политика? Мешают ли проблемы с кибербезопасностью

[1] Бас в течение некоторого времени занимался реализацией стандартов CIS (https://oreil.ly/mAxJw) и DISA-STIG (https://oreil.ly/EgDNP).

[2] *Кен Томпсон (Ken Thompson).* Reflections on Trusting Trust (https://oreil.ly/f52cw). *Communications of the ACM 27*. № 8 (август 1984 г.).

вашей организации достичь своих целей? Насколько хорошо обеспечивается тайна личной переписки?

На использование интернета и криптографии в современных ИТ-архитектурах влияет несколько факторов. В прокси-серверах широко используется проверка SSL, чтобы избежать заражения ПК вредоносными программами. Такая проверка позволяет администраторам наблюдать за трафиком, исходящим из веб-браузеров в компании на веб-сайты, и ограничивать его. Во избежание юридических последствий эти прокси-серверы поддерживают списки доверенных и ненадежных категорий сайтов. Прокси-серверы могут ограничивать использование интернета сотрудниками с добрыми намерениями, но проблемы с безопасностью могут исходить и от программного обеспечения. Имейте в виду, что проверка на стороне прокси-сервера может помочь предотвратить проникновение вирусов и программ-вымогателей в корпоративную сеть, но она может также препятствовать процессу разработки и внедрения инноваций.

Также рекомендуется создать прокси-сервер для библиотек программного обеспечения, чтобы упростить цепочку поставок для программистов. В главе 23 мы рассмотрим пример создания такого прокси с помощью Sonatype Nexus. Веб-трафик как бизнес-пользователей, так и ИТ-персонала должен регулироваться политикой, исключающей использование закрытых каналов.

Защищено, но не безопасно

Пример для этой главы создает виртуальную машину Vagrant с именем *ansiblebook/Bastion* (**https://oreil.ly/ajtGQ**), защищенную в соответствии с профилем защиты операционной системы (Operating System Protection Profile, OSPP) для RHEL 8.

> *Этот конфигурационный профиль соответствует стандарту CNSSI-1253, который требует, чтобы системы национальной безопасности США придерживались определенных конфигурационных параметров. Соответственно, этот профиль подходит для использования в системах национальной безопасности США.*

Значит ли, что эта защищенная машина находится в безопасности? Конечно!

Роль `ansible_role_ssh` в примере может применять (настраиваемую) общесистемную криптополитику. Роль `ansible_role_ansible` устанавливает Python, необходимые зависимости, Ansible, коллекции и роли в эту защищенную операционную систему. Она определяет ограничивающие параметры монтирования томов, SELinux и `fapolicyd`.

Мы опубликовали эти две роли отдельно, чтобы вы могли использовать их в других сценариях:

- `ansible_role_ssh` (**https://oreil.ly/H3Ha6**);
- `ansible_role_ansible` (**https://oreil.ly/c9XmX**).

В конфигурации запуска (пример 21.3) расширение `org_fedora_oscap` использует профиль `ospp`, основанный на криптополитике FIPS. Криптополитика FIPS:OSPP еще больше ограничивает набор алгоритмов, чем FIPS. В настоящее время FIPS исключает некоторые криптографические алгоритмы, а правительственным учреждениям США предписано использовать только определенный набор алгоритмов, проверенных национальным институтом стандартов NIST.

Пример 21.3. *packer-playbook.yml*

```
---
- name: Provisioner
  hosts: all
  become: true
  gather_facts: true
  vars:
    crypto_policy: FIPS:OSPP
    intended_user: vagrant
    home_dir: "/home/{{ intended_user }}"
  pre_tasks:
    - name: Generate 4096 bits RSA key pair for SSH
      user:
        name: "{{ intended_user }}"
        generate_ssh_key: true
        ssh_key_bits: 4096

    - name: Fetch ssh keys
      fetch:
        flat: true
        src: "{{ home_dir }}/.ssh/{{ item }}"
        dest: files/
        mode: '0600'
      loop:
        - id_rsa
        - id_rsa.pub

    - name: Install authorized_keys from generated file
      authorized_key:
        user: "{{ intended_user }}"
        state: present
        key: "{{ lookup('file','files/id_rsa.pub') }}"
        exclusive: false

    - name: Fix auditd max_log_file_action
```

```
      lineinfile:
        path: /etc/audit/auditd.conf
        regexp: '^max_log_file_action'
        line: max_log_file_action = rotate
        state: present
  roles:
    - ansible_book_ssh
    - ansible_book_ansible
```

Машина *ansiblebook/Bastion* подготавливается с помощью Packer, и для нее создается пара ключей с размером больше, чем принято в Vagrant по умолчанию. Вы сможете запустить ее с Vagrant после загрузки этого 4096-битного ключа RSA; сохраните его в файле с именем, как указано в Vagrantfile:

```
config.ssh.private_key_path = "./playbooks/files/id_rsa"
```

Сценарий Ansible, показанный в примере 21.4, проверит защищенность машины и создаст отчет в папке *Downloads*.

Пример 21.4. *vagrant-playbook.yml*

```
---
- name: Security Audit
  hosts: bastion
  become: true
  gather_facts: true
  tasks:
    - name: 'Run the audit and create a report.'
      command:
        oscap xccdf eval \
          --report /tmp/report.html
          --profile ospp
          /usr/share/xml/scap/ssg/content/ssg-rhel8-ds.xml
      no_log: true
      ignore_errors: true

    - name: 'Fetch the report.'
      fetch:
        flat: true
        src: /tmp/report.html
        dest: "~/Downloads/ospp.html"
...
```

Вы увидите, что машина успешно преодолевает 198 из 200 тестов безопасности, что довольно хорошо! Она защищена.

Однако если запустить `ssh-audit` (**https://oreil.ly/gepyo**) в этой «защищенной» системе, то вы увидите множество недостатков:

```
# key exchange algorithms
(kex) ecdh-sha2-nistp256      -- [fail] using weak elliptic curves
(kex) ecdh-sha2-nistp384      -- [fail] using weak elliptic curves
(kex) ecdh-sha2-nistp521      -- [fail] using weak elliptic curves
# host-key algorithms
(key) ecdsa-sha2-nistp256     -- [fail] using weak elliptic curves
                              `- [warn] using weak random number generator could
                              reveal the key
# encryption algorithms (ciphers)
(enc) aes256-cbc              -- [fail] removed (in server) since OpenSSH 6.7,
unsafe algorithm
                              `- [warn] using weak cipher mode
(enc) aes128-cbc              -- [fail] removed (in server) since OpenSSH 6.7,
unsafe algorithm
                              `- [warn] using weak cipher mode
# message authentication code algorithms
(mac) hmac-sha2-256           -- [warn] using encrypt-and-MAC mode
(mac) hmac-sha2-512           -- [warn] using encrypt-and-MAC mode

# algorithm recommendations (for OpenSSH 8.0)
(rec) -aes128-cbc             -- enc algorithm to remove
(rec) -aes256-cbc             -- enc algorithm to remove
(rec) -ecdh-sha2-nistp256     -- kex algorithm to remove
(rec) -ecdh-sha2-nistp384     -- kex algorithm to remove
(rec) -ecdh-sha2-nistp521     -- kex algorithm to remove
(rec) -ecdsa-sha2-nistp256    -- key algorithm to remove
(rec) -hmac-sha2-256          -- mac algorithm to remove
(rec) -hmac-sha2-512          -- mac algorithm to remove
```

Подобные недостатки можно найти в настройке OpenSSH 8 по умолчанию и в рекомендациях государств, одобряющих наблюдение за безопасностью систем. Вы можете использовать роль SSH со значением по умолчанию `crypto_policy: STRICT`, чтобы использовать кривые ed25519. Этот алгоритм быстрее и безопаснее, как доказали исследования технического университета Эйндховена (**https://oreil.ly/Tz9u0**). Использование кривых ed25519 предлагается также для обновленной версии FIPS, но документ FIPS 186-5 по-прежнему имеет статус «предварительный». Криптополитика `STRICT` обеспечивает прохождение тестов `ssh-audit`. Обратите внимание, что при этом ваша система может соответствовать только базовому уровню безопасности со слабой криптографией.

Появление квантовых компьютеров может иметь серьезные последствия для организаций, обрабатывающих конфиденциальную информацию: данные, зашифрованные с помощью популярных криптографических алгоритмов, возможно, уже были перехвачены и ожидают расшифровки с помощью будущего квантового компьютера. В версии OpenSSH 9 (**https://oreil.ly/lZgN1**) произошли существенные изменения; те-

перь она по умолчанию использует алгоритм NTRU и обмен ключами X25519 ECDH, чтобы предотвратить эти последствия.

Теневые ИТ-ресурсы

Ваше устройство *защищено* или *безопасно*? Эффективны ли ваши меры безопасности? Действуют ли ограничения политики неукоснительно, или их можно обойти? А что можно сказать обо всех других устройствах в вашей компании? Защищает ли ваша служба ИТ сетевую инфраструктуру, серверы, доступ к данным и ваши рабочие столы настолько строго, что вы вынуждены отправлять файлы по электронной почте на свой личный адрес, чтобы выполнить полезную работу? Приходится ли вам обращаться к другим альтернативам? Корпоративное управление может затормозить инновационные инициативы внедрением чрезмерных процессов утверждения и аудитов рисков и соответствия, не говоря уже о технических мерах безопасности, таких как защита конечных точек, проверка SSL и изолированные окружения[1]. Сотрудники либо тратят оплачиваемое время, чтобы преодолеть все эти ограничения, либо создают *теневые ИТ-ресурсы*.

К теневым ИТ-ресурсам (Shadow IT) относятся любые вычислительные ресурсы, которые не закупаются и не предоставляются в рамках корпоративного управления. Сюда входят личные ноутбуки, старые ПК, спрятанные под столами, персональные облачные подписки, персональные серверы и т. д. Чтобы обойти конкурентов, некоторые корпорации даже создают совершенно новые компании с целью избежать бюрократических проволочек, накопившихся за десятилетия. Если центральное ИТ-подразделение поставляет системы, не соответствующие ожиданиям разработчиков, то разработчики будут создавать свои системы.

Солнечные ИТ-ресурсы

Наибольшую эффективность в создании современного программного обеспечения показывают автономные команды, поддерживаемые платформами, которые не снижают их продуктивность. Эти команды обладают, если можно так выразиться, *интеллектуальной автономией*; т. е. у них есть доступ к любой информации, API, AI, SaaS, IaaS, PaaS, исходному коду, библиотекам или инструментам, необходимым для выполнения работы. Они могут организовать свою работу и общаться, сохраняя строгую конфиденциальность. Со стратегической точки зрения автономия – это явное конкурентное преимущество. Она может пошатнуть ваше положение в центральном ИТ-отделе, но все не так страшно!

[1] Келли Шортридж (Kelly Shortridge) опубликовала в своем блоге красноречивую статью (https://oreil.ly/NCrp9) о таком обструкционизме безопасности.

Под термином *солнечные ИТ-ресурсы* (Sunshine IT) подразумевается общая платформа, основанная на API с выходом в интернет, инфраструктуре самообслуживания и безопасной совместной работе, предоставляющая командам возможности проявить себя. Наряду с бизнес-приложениями стек технологий облегчает командам работу благодаря таким элементам, как:

- программно определяемая инфраструктура: ориентированная на приложения/облако;
- услуги платформы: CI/CD как услуга, контейнерные платформы;
- платформа интеграции: диспетчеры API/потоковой передачи событий / обмена сообщениями;
- технологический мониторинг.

Таким образом, внедрение солнечных ИТ-ресурсов вместо автономии способствует сотрудничеству между командами в организации, а некоторые базовые элементы могут усилить автономные команды.

Нулевое доверие

Идея *нулевого доверия* (zero trust) – модный термин, придуманный экспертом по безопасности Джоном Киндервагом (John Kindervag), который утверждает, что традиционная модель безопасности основывается на устаревшем предположении: все внутри сети организации, с ее бастионами и брандмауэрами, защищающими периметр, должно пользоваться безоговорочным доверием. Однако такое доверие означает отсутствие детальных элементов управления безопасностью, в результате, оказавшись в сети, пользователи, в том числе и злоумышленники, могут свободно перемещаться в горизонтальном направлении, получать доступ к конфиденциальным данным или извлекать их. Эта модель потеряла свою актуальность в эпоху облачных и контейнерных технологий. Продавцы предложат вам управление идентификацией, явную проверку, автоматизацию, минимальные привилегии и другие модные словечки, чтобы постараться продать побольше и подороже. Просто укажите им на эту цитату из Киндервага[1]:

> «Отличительной чертой нулевого доверия является простота. Никакой пользователь, пакет, сетевой интерфейс и устройство не должны пользоваться безграничным доверием. Если это правило соблюдается, то защита ресурсов становится простой. Чтобы уменьшить сложность среды кибербезопасности, организации могут отдавать приоритет технологиям и инструментам безопас-

[1] *Джон Киндерваг* (John Kindervag). «The Hallmark of Zero Trust Is Simplicity». *Wall Street Journal*, 15 апреля 2021 (https://oreil.ly/41KGi).

ности, поддерживающим простоту, автоматизируя повторяющиеся и выполняемые вручную задачи, интегрируя несколько инструментов и систем безопасности и управляя ими, а также автоматически устраняя известные уязвимости».

К настоящему времени появилось новое поколение программного обеспечения для сетевой безопасности, которым можно управлять с помощью простых приложений. Эти программные продукты позволяют администраторам создавать группы доверенных пользователей, системы которых могут подключаться через ненадежные сети. Они предлагают полный контроль за пользователями и кросс-платформенное шифрование.

Заключение

Узнать больше об автоматизации сетевых устройств с помощью Ansible можно в статьях «Network Getting Started» (**https://oreil.ly/JLMz6**) и «Network Advanced Topics» (**https://oreil.ly/1NvKm**). Желающим поэкспериментировать мы советуем установить роли и коллекции из примера 15.1. Также посетите сайт советов Mozilla Foundation (**https://oreil.ly/ViJ3a**).

Автоматизация безопасности – это сфера использования Ansible, о которой можно написать целую книгу, и нам очень повезло, что команда Ansible опубликовала руководство «Security Automation» (**https://oreil.ly/JF7g6**). В следующей главе мы продолжим тему автоматизации и рассмотрим применение Ansible для организации конвейера CI/CD.

Глава 22

CI/CD и Ansible

Роли – это основные компоненты, используемые для создания инфраструктуры как кода (Infrastructure as Code, IaC) с помощью Ansible. Отношение к системному администрированию как к разработке программного обеспечения и применение методов разработки к IaC – одна из основ методологии гибкой разработки. Тестируя изменения в программных окружениях и автоматизируя контроль за изменениями, можно уменьшить количество ошибок, повысить продуктивность и сократить периоды простоя. Оценивая качество кода и автоматически выполняя тесты в изолированных окружениях, можно устранять ошибки на самых ранних этапах, до того, как они просочатся в промышленное окружение.

В этой главе описывается, как настроить основу среды непрерывной интеграции и непрерывной доставки (Continuous Integration/Continuous Delivery, CI/CD) программного обеспечения, состоящую из прокси-сервера центрального репозитория для двоичных файлов и библиотек, системы управления исходным кодом, инструмента контроля качества кода и сервера непрерывной интеграции. В примере, который мы рассмотрим далее, представлены четыре виртуальные машины с Sonatype Nexus3, Gitea, SonarQube и Jenkins. Jenkins может использовать специальные команды и сценарии Ansible через плагин Ansible. Плагин Ansible Tower для Jenkins позволяет получить доступ к платформе автоматизации Ansible (известной как Tower) для выполнения различных операций, таких как запуск шаблонов заданий.

Непрерывная интеграция

В 2006 году Мартин Фаулер (Martin Fowler) опубликовал важную статью о непрерывной интеграции (**https://oreil.ly/A03QV**), описывающую успешную практику разработки программного обеспечения следующим образом:

> «Практика разработки программного обеспечения, когда члены команды достаточно часто интегрируют результаты своего труда (например, каждый интегрирует свой код по меньшей мере еже-

> дневно), приводит к выполнению большого количества интеграций в течение дня. Каждая интеграция проверяется автоматизированной сборкой (и тестированием), чтобы как можно раньше обнаружить ошибки интеграции. Многие команды считают, что такой подход значительно сокращает проблемы интеграции и позволяет команде быстрее разрабатывать программное обеспечение».

Эти методы часто незаменимы, когда требуется доставлять программное обеспечение надежным и воспроизводимым способом. Как выразился Фаулер, «каждый должен иметь возможность подключить девственно чистую машину, проверить исходный код из репозитория, выполнить единственную команду и получить работающую систему на своей машине».

В настоящее время появились еще более серьезные проблемы: большинство современных систем настолько сложны, что для работы им требуется несколько машин, а их инфраструктура, управление конфигурацией, системные операции, обеспечение безопасности и соответствия стандартам часто находятся *в коде*.

Разработчики хранят весь этот код в системе управления версиями и выполняют различные задачи на серверах интеграции, благодаря чему имеют возможность протестировать этот код и безопасно сохранить в репозитории, чтобы развернуть его, когда все будет готово к выпуску. Проще говоря, все это желательно автоматизировать.

Элементы системы непрерывной интеграции

Хранение всего необходимого в системе управления версиями (Version Control System, VCS) является важным условием для непрерывной интеграции (CI). Существует два типа VCS: для текстовых данных, таких как исходный код любого типа, и хранилища артефактов для двоичных данных, таких как пакеты программного обеспечения любого типа.

Репозиторий артефактов

Самыми популярными хранилищами артефактов являются, пожалуй, JFrog Artifactory и Sonatype Nexus. Пример кода, прилагаемый к этой книге, развертывает Nexus как прокси для библиотек Python. Nexus – это программа на Java, и для ее развертывания достаточно очень простого сценария:

```
#!/usr/bin/env ansible-playbook
---
- name: Artefact Repository
  hosts: nexus
  become: true
  roles:
```

```
    - role: java
      tags: java
    - role: nexus
      tags: nexus
```

У нас есть реестр с группой под названием `nexus` и соответствующим сервером в ней. Для этого проекта вы можете создать реестр с четырьмя серверами по своему выбору; он многоразовый. Роли устанавливаются из Ansible Galaxy с помощью файла *roles/requirements.yml*:

```
---
roles:
  - src: ansible-thoteam.nexus3-oss
    name: nexus
  - src: geerlingguy.java
    name: java
```

Далее создадим *group_vars/nexus*. Для этого примера мы определили простые конфигурационные параметры, как показано ниже:

```
nexus_config_pypi: true
nexus_config_docker: true
nexus_admin_password: 'changeme'
nexus_anonymous_access: true
nexus_public_hostname: "{{ ansible_fqdn }}"
nexus_public_scheme: http
httpd_setup_enable: false
```

Nexus имеет множество конфигурационных параметров и поддерживает сценарии.

Gitea

Для управления версиями исходного кода в настоящее время наиболее широко используется Git. В числе известных репозиториев, использующих эту систему, можно назвать GitHub (**https://github.com/**), Atlassian BitBucket (**https://bitbucket.org/**) и GitLab ([**https://gitlab.com/**] с открытым исходным кодом). В корпоративной среде BitBucket обычно используется в комбинации с другими инструментами Atlassian, такими как Confluence и Jira. GitHub и GitLab предлагают корпоративные решения и конкурируют по набору возможностей. Если вы решите «развернуть свой репозиторий Git», то обратите внимание на облегченную версию – Gitea, решение с открытым исходным кодом, имеющее пользовательский интерфейс, подобный Github, и хорошо структурированный, развитый API.

Давайте создадим группу с именем `git` в нашем реестре и сценарий Ansible для развертывания Gitea с системой управления базами данных MySQL на одном хосте:

```
---
- name: Git Server
  hosts: git
  become: true
  collections:
    - community.mysql
  roles:
    - role: mysql
      tags: mysql
    - role: gitea
      tags: gitea
```

Коллекция и роли устанавливаются из Ansible Galaxy с помощью следующих строк в *roles/requirements.yml*:

```
collections:
  - community.mysql
roles:
  - src: do1jlr.gitea
    name: gitea

  - src: do1jlr.mysql
    name: mysql
```

В *group_vars/git* определена конфигурация для базы данных и Gitea:

```
# https://github.com/roles-ansible/ansible_role_gitea
gitea_db_host: '127.0.0.1:3306'
gitea_db_name: 'gitea'
gitea_db_type: 'mysql'
gitea_db_password: "YourOwnPasswordIsBetter"
gitea_require_signin: false
gitea_fqdn: "{{ ansible_fqdn }}"
gitea_http_listen: '0.0.0.0'
gitea_http_port: '3000'

# https://github.com/roles-ansible/ansible_role_mysql
mysql_bind_address: '127.0.0.1'
mysql_root_password: ''  # небезопасно
mysql_user_home: /home/vagrant
mysql_user_name: vagrant
mysql_user_password: vagrant
mysql_databases:
  - name: 'gitea'
mysql_users:
  - name: "{{ gitea_db_name }}"
    password: "{{ gitea_db_password }}"
    priv: "{{ gitea_db_name }}.*:ALL"
    state: present
```

Эта конфигурация – лишь упрощенный вариант установки Gitea; ее можно расширить и дополнить более сложными настройками.

Качество кода

Разработчикам нужны инструменты проверки качества программного обеспечения. Дополнительные инструменты нужны также для оценки технического долга и выявления проблем безопасности. Для этой цели можно использовать SonarSource SonarQube – программное обеспечение с открытым исходным кодом. Вот сценарий, устанавливающий SonarQube:

```
- name: Code Quality
  hosts: sonar
  become: true
  collections:
    - community.postgres
  roles:
    - role: utils
    - role: java
    - role: postgres
      tags: postgres
    - role: sonarqube
```

Коллекция и роли устанавливаются из Ansible Galaxy с помощью следующих строк в *roles/requirements.yml*:

```
---
collections:
  - community.postgresql
roles:
  - src: dockpack.base_utils
    name: utils
  - src: geerlingguy.java
    name: java
  - src: lrk.sonarqube
    name: sonarqube
  - src: robertdebock.postgres
    name: postgres
```

В *group_vars/sonar* определена конфигурация для базы данных и инструмента SonarQube, известного как Sonar, а также необходимые пакеты. Возможности Sonar можно расширить с помощью плагинов. Существует плагин для запуска `ansible-lint`, который может очень пригодиться в проектах, использующих Ansible и исходный код на других языках. SonarQube – это программа на Java, но она поддерживает множество языков программирования. Она прекрасно интегрируется с базой дан-

ных Postgres; однако для создания пользователей необходимо установить несколько дополнительных пакетов, чтобы создать базу данных для библиотек Python. Ниже показан минимум, что вам понадобится:

```
base_utils:
  - gcc
  - make
  - python36-devel
  - unzip
java_packages:
  - java-11-openjdk-devel
```

Сервер CI

В зависимости от применяемых методов управления исходным кодом может понадобиться, чтобы ваш собственный сервер сборки выполнял некоторые задачи автоматически. В GitHub для этой цели есть Actions, а в GitLab – Runners, автоматически запускающие задачи в контейнерах. Оба варианта доступны как в облаке, так и локально, с различными коммерческими тарифами. Как вариант, можно запустить свой сервер CI, например, с использованием TeamCity, Atlassian Bamboo или Jenkins.

Jenkins

Jenkins – это де-факто стандартный сервер CI, программа на Java, которая легко настраивается под разные нужды с помощью плагинов. Среди них есть несколько плагинов для работы с системами Git, включая Gitea, GitHub и BitBucket. Также доступны плагины для Ansible и Ansible Tower.

Однако для системных администраторов настройка Jenkins долгое время оставалась трудоемкой задачей, выполняемой вручную, включающей установку зависимостей, запуск и настройку сервера Jenkins, определение конвейеров и настройку заданий. Излишне говорить, что все это должно быть максимально автоматизировано.

Мы создали группу `jenkins` в нашем реестре и сценарии Ansible для развертывания Jenkins, использовав роли, написанные Джеффом Герлингом (автором книги «Ansible for DevOps» и владельцем учетной записи @geerlingguy на Ansible Galaxy и GitHub):

```
- name: CI Server
  hosts: jenkins
  become: true
  roles:
    - role: epel
      tags: epel
    - role: utils
      tags: utils
```

```
    - role: java
    - role: docker
      tags: docker
    - role: jenkins
      tags: jenkins
    - role: configuration
      tags: qa
```

Большинство ролей устанавливается из Ansible Galaxy с помощью следующих строк в *roles/requirements.yml*:

```
---
roles:
  - src: dockpack.base_utils
    name: utils
  - src: geerlingguy.repo-epel
    name: epel
  - src: geerlingguy.docker
    name: docker
  - src: geerlingguy.java
    name: java
  - src: geerlingguy.jenkins
    name: jenkins
...
```

В *group_vars/jenkins* определена базовая конфигурация для плагинов и нескольких инструментов:

```
jenkins_plugins:
  - ansible
  - ansible-tower
  - ansicolor
  - configuration-as-code
  - docker
  - docker-build-step
  - docker-workflow
  - git
  - gitea
  - job-dsl
  - pipeline-build-step
  - pipeline-rest-api
  - pipeline-stage-view
  - sonar
  - timestamps
  - ws-cleanup
base_utils:
  - unzip
  - git
```

```
docker_users:
  - jenkins
  - vagrant
```

Этот код устанавливает Docker и позволяет Jenkins его использовать.

Jenkins и Ansible

Установка плагинов для Ansible и Ansible Tower добавляет только архивы Java с расширением *.jpi*, поэтому Python и Ansible вам придется устанавливать самостоятельно. Вариантов установки много, но для этого примера просто создадим роль для Jenkins и протестируем с ее помощью некоторые роли.

Конфигурация Jenkins как код

Если вы убежденный сторонник идей управления конфигурациями, то вы наверняка предпочтете автоматизировать настройку Jenkins. Для этого предусмотрен API, который используется в роли `geerlingguy.jenkins`, имеющий такие методы, как `get_url` и `uri`. Внутренне Jenkins настраивается преимущественно с использованием XML-файлов, однако мы можем использовать ряд модулей Ansible, перечисленных в табл. 22.1.

Таблица 22.1. Модули Ansible для настройки Jenkins

Модуль	Назначение
jenkins_job	Управление заданиями Jenkins
jenkins_job_facts	Получение информации о заданиях Jenkins
jenkins_job_info	Получение информации о заданиях Jenkins
jenkins_plugin	Добавление и удаление плагинов Jenkins
jenkins_script	Выполнение сценария Groovy на экземпляре Jenkins

Groovy – это язык сценариев JVM, который используется внутри Jenkins.

Jenkins также можно использовать из командной строки при условии, что вы загрузили jar-файл из API:

```
- name: Get Jenkins CLI for automation
  get_url:
    url: "http://127.0.0.1:8080/jnlpJars/jenkins-cli.jar"
    dest: /var/lib/jenkins/jenkins-cli.jar
    mode: '0755'
    timeout: 300
  retries: 3
  delay: 10
```

Для сложной системы автоматизации, такой как Jenkins, следует использовать Ansible как можно меньше, чтобы она управлялась сама. Плагин `configuration-as-code` (`casc`) использует файл YAML для настройки различных элементов Jenkins. Jenkins может сам установить некоторые инструменты, используя этот конфигурационный файл на YAML, который мы устанавливаем с помощью модуля `template`:

```
tool:
  ansibleInstallation:
    installations:
    - home: "/usr/local/bin"
      name: "ansible"
  git:
    installations:
    - home: "git"
      name: "Default"
  jdk:
    installations:
    - properties:
      - installSource:
          installers:
          - jdkInstaller:
              acceptLicense: true
              id: "jdk-8u221-oth-JPR"
  maven:
    installations:
    - name: "Maven3"
      properties:
      - installSource:
          installers:
          - maven:
              id: "3.8.4"
  mavenGlobalConfig:
    globalSettingsProvider: "standard"
    settingsProvider: "standard"
  sonarRunnerInstallation:
    installations:
    - name: "SonarScanner"
      properties:
      - installSource:
          installers:
          - sonarRunnerInstaller:
              id: "4.6.2.2472"
```

Он поддерживает не все инструменты, поэтому мы установили Git с помощью роли `utils`.

Главное преимущество этого метода – Jenkins будет устанавливать инструменты по запросу на тех агентах сборки, которые в них нуждаются. (*Агенты сборки* – это дополнительные серверы, добавляемые с увеличением нагрузки.) Ниже показано, как настроить Jenkins с помощью файлов YAML. Обратите внимание, что Jenkins необходимо перезапустить с дополнительным параметром Java, который сообщит, где найти эти файлы:

```
- name: Ensure casc_configs directory exists
  file:
    path: "{{ casc_configs }}"
    state: directory
    owner: jenkins
    group: root
    mode: '0750'

- name: Create Jenkins jobs configuration
  template:
    src: jenkins.yaml.j2
    dest: "{{ casc_configs }}/jenkins.yaml"
    owner: jenkins
    group: root
    mode: '0440'

- name: Enable configuration as code
  lineinfile:
    dest: /etc/sysconfig/jenkins
    regexp: '^JENKINS_JAVA_OPTIONS='
    line:>-
      JENKINS_JAVA_OPTIONS="-Djava.awt.headless=true
      -Djenkins.install.runSetupWizard=false
      -Dcasc.jenkins.config={{ casc_configs }}"
    state: present
    mode: '0600'
  notify: Restart Jenkins

- name: Flush handlers
  meta: flush_handlers

- name: Wait for Jenkins
  wait_for:
    port: 8080
    state: started
    delay: 10
    timeout: 600
```

Сохраните файл YAML в каталоге */var/lib/jenkins/casc_configs* и настройте параметр Java `Dcasc.jenkins.config=/var/lib/jenkins/casc_configs`. Он сообщит Jenkins, где искать конфигурацию.

Конфигурации заданий Jenkins как код

При необходимости с помощью плагина `job-dsl` (**https://oreil.ly/AXKGW**) можно реализовать дополнительный уровень автоматизации. Вот что об этом говорится в документации плагина Jenkins (**https://oreil.ly/QuJRE**):

> «Jenkins – замечательная система управления сборкой, и многим нравится настраивать ее задания с помощью ее пользовательского интерфейса. К сожалению, с ростом числа заданий поддерживать их становится утомительно, и парадигма использования пользовательского интерфейса оказывается в проигрыше. Кроме того, в этой ситуации используется типичный шаблон – копирование заданий для создания новых. Но эти "потомки" имеют привычку отклоняться от своего первоначального "шаблона", что затрудняет поддержание согласованности между заданиями.
>
> Плагин Job DSL пытается решить эту проблему, позволяя определять задания программно в удобочитаемом файле. К счастью, чтобы написать такой файл, не обязательно быть экспертом в Jenkins, потому что конфигурацию из веб-интерфейса легко преобразовать в код».

Проще говоря, вы можете создавать задания Jenkins, основанные на начальных заданиях. Чтобы настроить Jenkins для этого, добавьте дополнительный блок в шаблон `casc` на YAML:

```
jobs:
  - file: /home/jenkins/jobs.groovy
```

Теперь нужно описать задания в файле Groovy. Как знатоки Ansible, мы используем шаблон Jinja2 – `jobs.groovy.j2`:

```
{% for repo in git_repositories %}
pipelineJob('{{ repo }}') {
  triggers {
    scm ''
  }
  definition {
    cpsScm {
      scm {
        git {
          remote {
            url('https://{{ git_host }}/{{ git_path }}/{{ repo }}.git')
          }
```

```
        }
      }
      scriptPath('Jenkinsfile')
    }
  }
}
{% endfor %}
```

Для этого шаблона необходимо определить следующие переменные:

```
git_host: github.com
git_path: ansiblebook
git_repositories:
  - ansible_role_ssh
  - ansible_role_ansible
  - ansible_role_web
```

Теперь файл *jobs.groovy* установлен. Вы можете использовать модуль `command` для активации заданий с помощью `jenkins-cli.jar`, инструмента командной строки Java для Jenkins:

```
- name: Create Job DSL plugin seed job
  template:
    src: jobs.groovy.j2
    dest: /home/jenkins/jobs.groovy
    owner: jenkins
    mode: '0750'

- name: Activate jobs configuration with Jenkins CLI
  command: |
    java -jar jenkins-cli.jar \
    -s http://127.0.0.1:8080/ \
    -auth admin:{{ jenkins_admin_password }} \
    reload-jcasc-configuration
  changed_when: true
  args:
    chdir: /var/lib/jenkins
```

Запуск CI для ролей Ansible

Molecule (рассматривается в главе 14) – отличный фреймворк для оценки качества ролей Ansible. Чтобы автоматизировать задание Jenkins, добавьте сценарий на Groovy в корневой каталог каждого репозитория с исходным кодом, для проверки которого предполагается использовать Jenkins. Этот сценарий должен называться `Jenkinsfile`. В нашем примере `Jenkinsfile` определяет этапы Jenkins для каждого этапа Molecule:

```
pipeline {
  agent any
  options {
    disableConcurrentBuilds()
    ansiColor('vga')
  }
  triggers {
    pollSCM 'H/15 * * * *'
    cron 'H H * * *'
  }
  stages {
    stage ("Build Environment") {
      steps {
        sh '''
          source /usr/local/bin/activate
          python -V
          ansible --version
          molecule --version
        '''
      }
    }
    stage ("Syntax") {
      steps {
        sh '(source /usr/local/bin/activate && molecule syntax)'
      }
    }
    stage ("Linting") {
      steps {
        sh '(source /usr/local/bin/activate && molecule lint)'
      }
    }
    stage ("Playbook") {
      steps {
        sh '(source /usr/local/bin/activate && molecule converge)'
      }
    }
    stage ("Verification") {
      steps {
        sh '(source /usr/local/bin/activate && molecule verify)'
      }
    }
    stage ("Idempotency") {
      steps {
        sh '(source /usr/local/bin/activate && molecule idempotence)'
      }
    }
  }
}
```

Определение этих этапов позволяет следить за их выполнением в интерфейсе Jenkins (рис. 22.1).

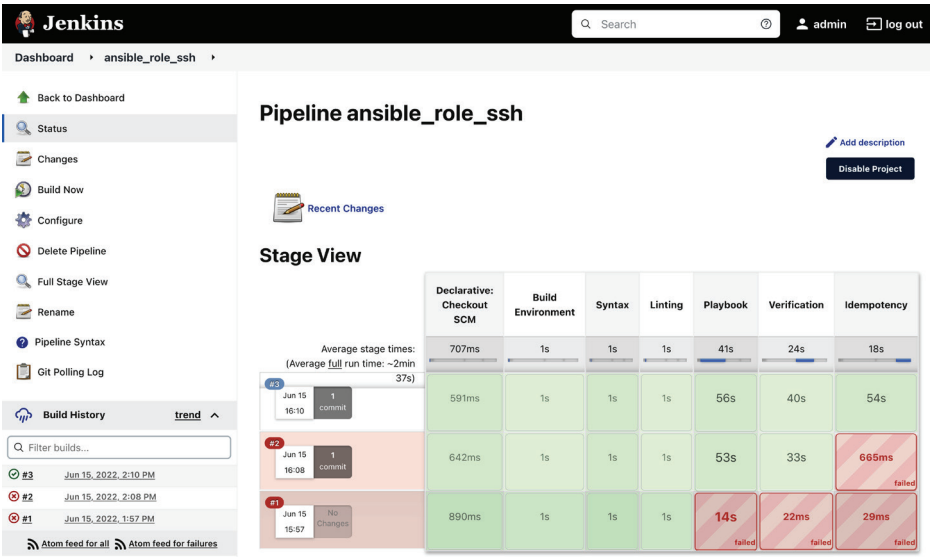


Рис. 22.1. Конвейер Jenkins для роли Ansible

Файлы Jenkinsfile поддерживают массу возможностей. Выше приведен лишь простой пример конвейера заданий, точно соответствующих этапам Molecule, но в нем не реализованы другие задачи. Дополнительную информацию о конвейерах вы найдете в документации Jenkins (**https://oreil.ly/YOt04**).

Обкатка

Большинство организаций, разрабатывающих программное обеспечение, имеют план обкатки. Под обкаткой подразумевается запуск отдельных окружений для разных целей в жизненном цикле программного обеспечения. Вы разрабатываете программное обеспечение на виртуальном рабочем столе, а программное обеспечение создается в среде разработки, тестируется в тестовой среде, затем развертывается для «приемки» и в конечном итоге в промышленном окружении. Сделать все это можно разными способами, но в целом желательно, чтобы проблемы обнаруживались как можно раньше. Хорошей практикой считается использование разделения сети и элементов управления безопасностью, таких как брандмауэры, доступом и резервированием. На рис. 22.2 показаны такие промежуточные окружения.

Базовая настройка обычно быстро усложняется, но Jenkins и особенно агенты Jenkins, ограниченные такими окружениями, могут помочь автоматизировать процесс обкатки достаточно безопасным способом.

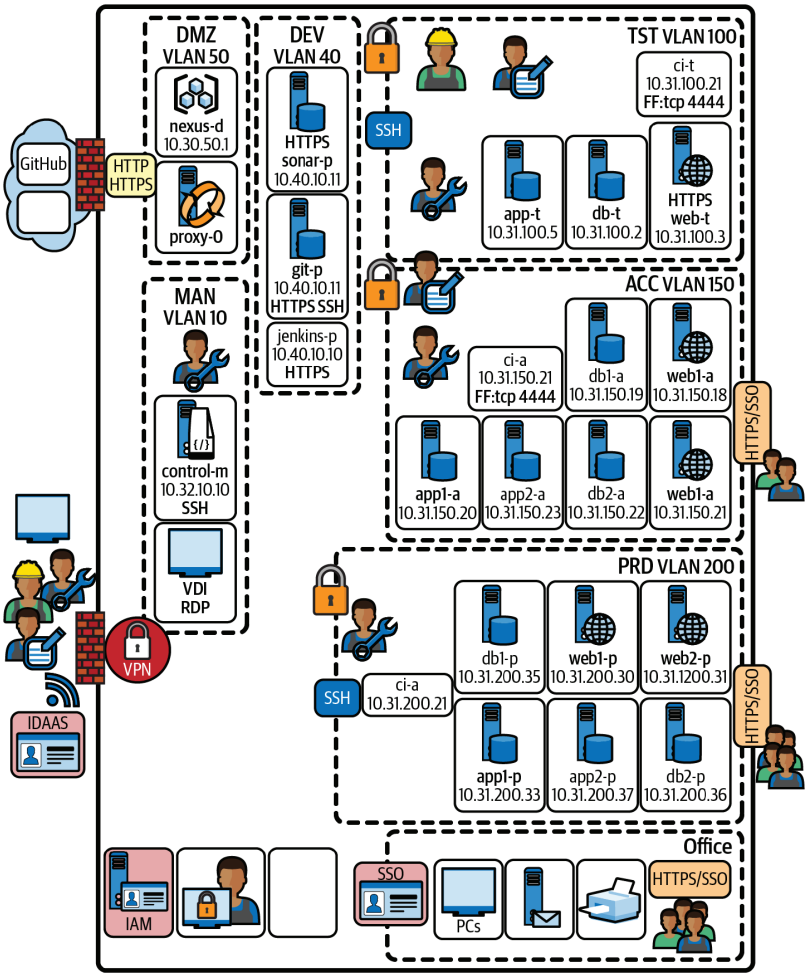


Рис. 22.2. Различные окружения обкатки

Плагин Ansible

Плагин Ansible для Jenkins создает пользовательский интерфейс для этапа сборки в задании Jenkins. Если решите использовать конвейерное задание с файлом Jenkinsfile, то сможете использовать фрагмент, подобный представленному ниже, для запуска сценария как этапа вашего конвейера:

```
ansiblePlaybook become: true, colorized: true, credentialsId: 'Machines',
disableHostKeyChecking: true, installation: 'ansible', inventory:
'inventory/hosts', limit: 'webservers', playbook: 'playbooks/playbook.yml',
tags: 'ssh', vaultCredentialsId: 'ANSIBLE_VAULT_PASSWORD'
```

Используйте Snippet Generator для параметризации этапа сборки (рис. 22.3).

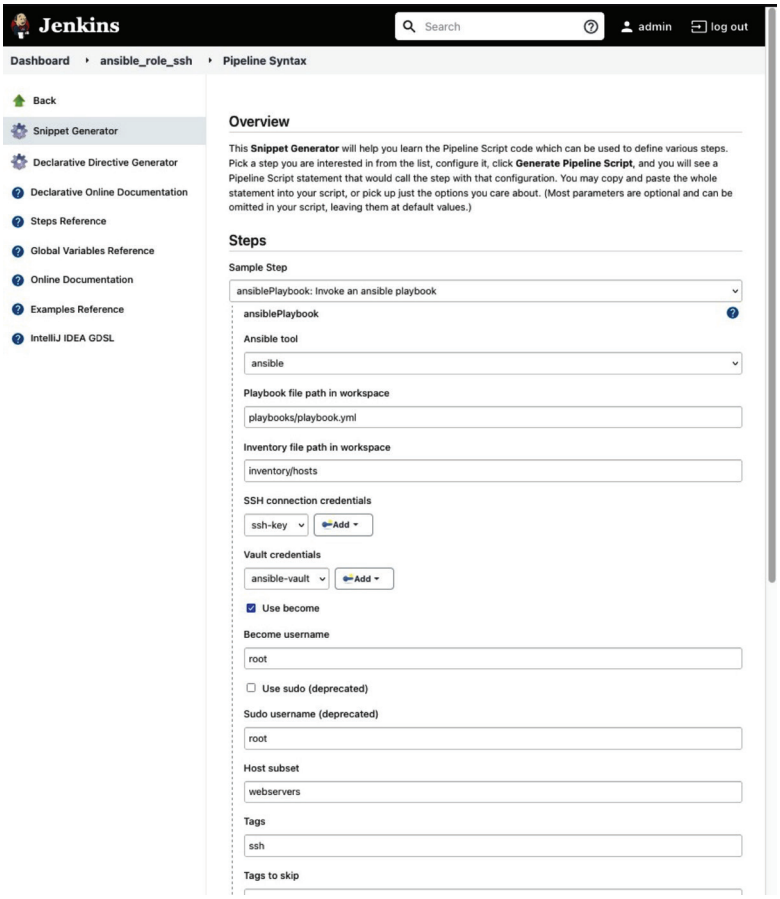


Рис. 22.3. Jenkins Snippet Generator для применения сценария Ansible на этапе сборки

Преимуществами использования Jenkins для запуска сценариев являются централизация и журналирование. Это естественное решение для команд разработчиков, которые уже знают и используют Jenkins. Ansible должен присутствовать на сервере Jenkins или на агентах Jenkins, которые будут выполнять задания.

Плагин Ansible Tower

Если вы автоматизируете производственную среду вашего предприятия с помощью Ansible Automation Controller (см. главу 23), то вам наверняка понадобится плагин Ansible Tower. Ansible Automation Controller обеспечивает лучшее масштабирование как по количеству команд, которые могут его использовать, так и по управлению доступом на основе ролей. Ansible Automation Controller также имеет больше функций безопасности, чем Jenkins.

Чтобы отделить задачи внутреннего контроля, в организациях часто создаются окружения обкатки и ограничивается доступ к промышленным окружениям. Разработчикам могут быть даны права на запуск заданий или рабочих процессов с четко определенной комбинацией сценариев, компьютеров, учетных данных и других предварительно настроенных параметров. Использование Jenkins для запуска шаблона задания может стать отличным шагом на пути к непрерывной доставке! С помощью Jenkins Snippet Generator можно организовать доступ к Ansible Automation Controller для запуска сценария Ansible с заданными параметрами (шаблон задания; рис. 22.4). В Ansible Automation Controller можно безопасно хранить учетные данные и делегировать их использование заданию Jenkins. Это означает, что разработчикам не придется входить в реестр для развертывания своего приложения. Им может быть даже запрещено это из соображений соответствия требованиям или рисков.

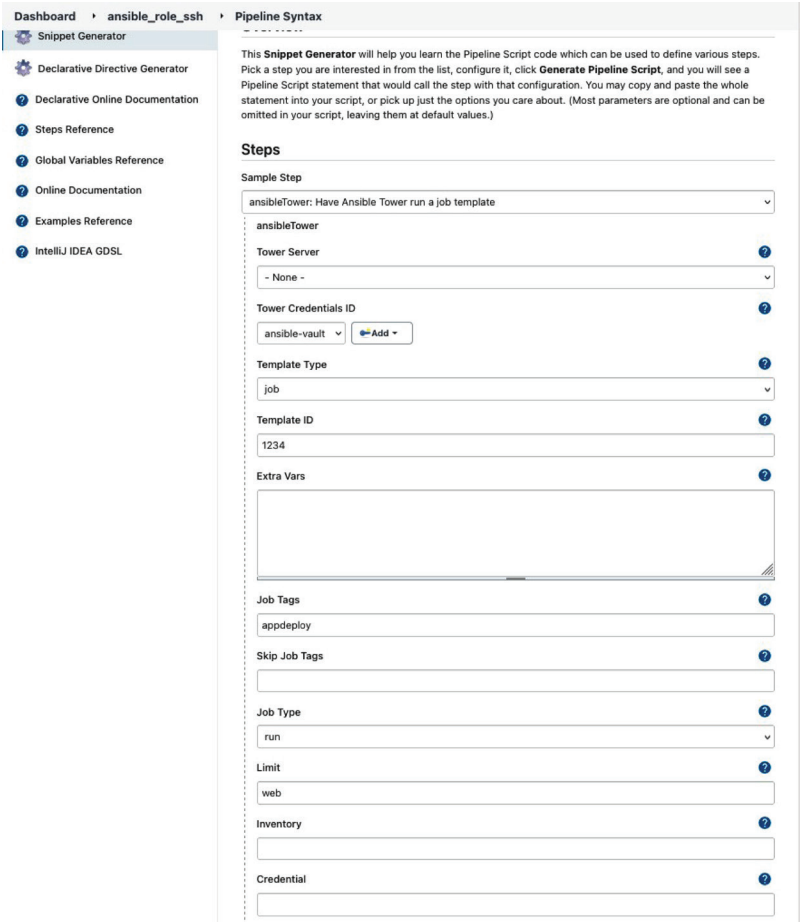


Рис. 22.4. Jenkins Snippet Generator для шага построения шаблона задания Ansible Tower

Этот плагин можно использовать после создания и тестирования программного обеспечения в окружении обкатки для развертывания приложения в промышленном окружении. Этот последний шаг сборки можно создать в Jenkinsfile с помощью Snippet Generator, использовав следующий код:

```
ansibleTower jobTags: 'appdeploy', jobTemplate: '1234', jobType: 'run', limit:
'web', throwExceptionWhenFail: false, towerCredentialsId:
'ANSIBLE_VAULT_PASSWORD', towerLogLevel: 'false', towerServer: 'tower'
```

Заключение

Ansible – отличный инструмент для непрерывной доставки сложных программных систем. Он может не только управлять средой разработки, но и глубоко интегрироваться в процессы обкатки программного обеспечения, автоматизируя все рутинные задачи, снижающие продуктивность при выполнении вручную.

Глава 23

Ansible Automation Platform

Ansible Automation Platform – это коммерческий программный продукт, предлагаемый Red Hat. Ansible Automation Platform 2 – это платформа автоматизации нового поколения для предприятий. Она включает переработанный *Automation Controller 4*, ранее известный как Tower/AWX, и *Automation Hub* – локальный репозиторий для контента Ansible, замещающий локальный репозиторий Ansible Galaxy. Вы можете настроить Automation Hub в соответствии с политиками управления в вашей организации или просто синхронизировать его с репозиторием сообщества. В примере 23.1 показан файл, который можно передать администратору Automation Hub (см. рис. 23.1). Он определяет коллекции, которые Automation Hub будет обслуживать в локальной сети. Для загрузки коллекций Automation Hub должен иметь подключение к интернету.

Пример 23.1. requirements.yml для получения контента сообщества из Automation Hub

```
---
collections:
  # Установка коллекций из Ansible Galaxy.
  - name: ansible.windows
    source: https://galaxy.ansible.com
  - name: ansible.utils
    source: https://galaxy.ansible.com
  - name: awx.awx
    source: https://galaxy.ansible.com
  - name: community.crypto
    source: https://galaxy.ansible.com
  - name: community.docker
    source: https://galaxy.ansible.com
  - name: community.general
    source: https://galaxy.ansible.com
  - name: community.kubernetes
    source: https://galaxy.ansible.com
...
```

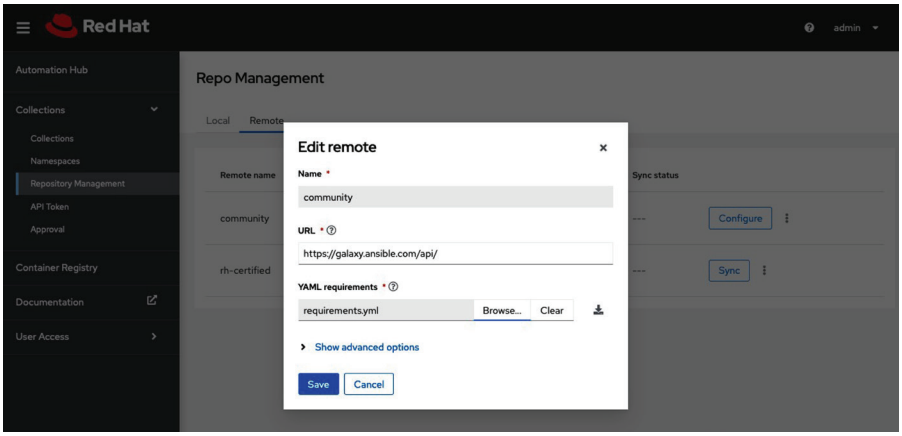


Рис. 23.1. Выгрузка файла требований

При желании в файле *ansible.cfg* можно настроить несколько серверов для команды `ansible-galaxy`, если вы используете Private Automation Hub в Ansible Automation Platform 2 (пример 23.2).

Пример 23-2. *ansible.cfg*

```
[galaxy]
server_list = automation_hub, release_galaxy, my_org_hub, my_test_hub

[galaxy_server.automation_hub]
url=https://cloud.redhat.com/api/automation-hub/
auth_url=https://sso.redhat.com/auth/realms/redhat-external/protocol/openid-connect/
token
token=my_ah_token

[galaxy_server.release_galaxy]
url=https://galaxy.ansible.com/
token=my_token

[galaxy_server.my_org_hub]
url=https://automation.my_org/
username=my_user
password=my_pass

[galaxy_server.my_test_hub]
url=https://automation-test.my_org/
username=test_user
password=test_pass
```

Окружения обкатки, такие как `my_test_hub`, можно использовать для тестирования локальных коллекций, которые в конечном итоге будут опубликованы в `my_org_hub`.

В архитектуре Ansible Automation Platform 2 используются разработки в области контейнерных технологий. Она более масштабируемая и безопасная, чем предыдущее поколение. Самое большое отличие заключается в отделении плоскости управления от окружений выполнения, как показано на рис. 23.2.

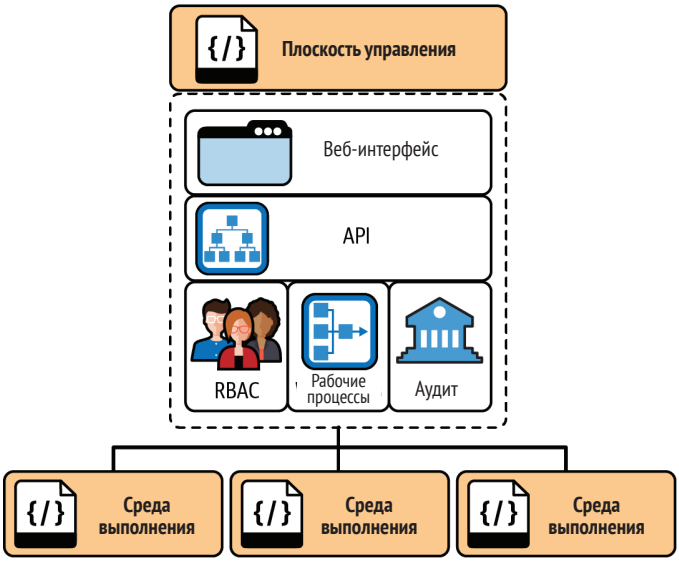


Рис. 23.2. Архитектура Ansible Automation Platform 2

Для управления зависимостями в Ansible Tower использовались виртуальные окружения Python, но этот метод создавал проблемы для оперативных команд Tower. Поэтому в Ansible Automation Platform 2 была добавлена автоматизация создания окружений выполнения; другими словами, автоматизация применяется к образам контейнеров, которые включают Ansible, контент Ansible и любые другие зависимости, как показано на рис. 23.3.

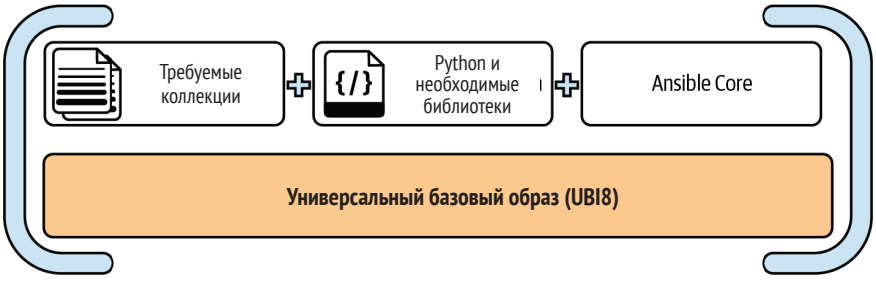


Рис. 23.3. Среда выполнения Ansible

Окружения выполнения Ansible основаны на `ansible-builder` ([**https://oreil.ly/NlgNY**] обсуждается далее в этой главе).

Ansible Automation Platform можно установить в RedHat OpenShift или на хостах Red Hat Enterprise Linux 8 (rhel/8). Пример кода для этой главы создает кластер разработки в VirtualBox с помощью Vagrant. Для создания машины rhel/8 в VirtualBox представлена также конфигурация Packer (Packer обсуждается в главе 16).

Automation Controller обеспечивает более точное управление политиками доступа на основе пользователей и ролей в сочетании с пользовательским веб-интерфейсом (рис. 23.4) RESTful API.

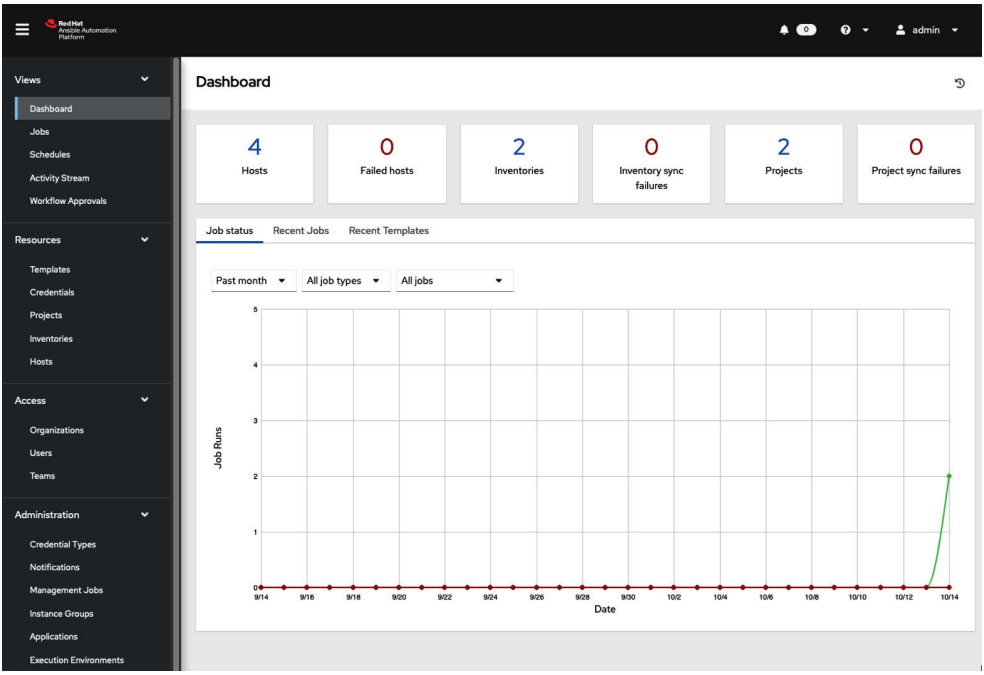


Рис. 23.4. Панель управления Ansible Automation Controller

Модели подписки

Red Hat предлагает поддержку (**https://oreil.ly/qsiFg**) в виде трех типов ежегодных подписок, каждый с разными соглашениями об уровне обслуживания (Service-Level Agreement, SLA):

- самостоятельная поддержка (без официальной поддержки и каких-либо обязательств);
- стандартная (поддержка с уровнем: 8×5);
- премиум (поддержка с уровнем: 24×7).

Все подписки включают рассылку регулярных обновлений и новых версий Ansible Automation Platform.

Как разработчик, вы можете получить бесплатный доступ ко многим технологическим ресурсам, предлагаемым компанией Red Hat. Для этого достаточно зарегистрироваться (**https://oreil.ly/Q7UDb**) и получить подписку Red Hat Developer for Individuals.

Пробная версия Ansible Automation Platform

Red Hat предоставляет бесплатную 60-дневную пробную лицензию (**https://oreil.ly/wSoD5**) с набором возможностей из модели подписки самостоятельной поддержки, до 100 управляемых хостов.

После регистрации в качестве разработчика и подачи заявки на пробную версию вы сможете экспортировать манифест лицензии (**https://oreil.ly/7j8MF**) для активации своего экземпляра, как показано на рис. 23.5.

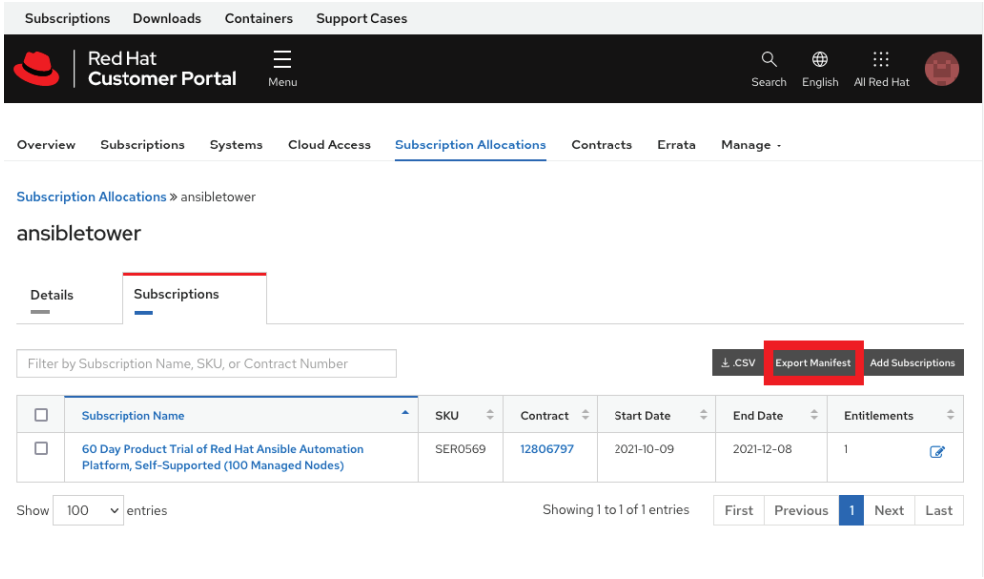


Рис. 23.5. Управление подписками

После приобретения Ansible, Inc. в 2015 году Red Hat выразила намерение продолжить разработку открытой версии Ansible Tower под названием AWX . Она устанавливается в Kubernetes с помощью AWX Operator. Подробные инструкции вы найдете в документации (**https://oreil.ly/NjaVt**). Исходный код AWX доступен на GitHub (**https://oreil.ly/heqzB**).

Для быстрой оценки этой версии можно воспользоваться Vagrant и сценарием из GitHub (**https://oreil.ly/FRY0I**):

```
$ git clone https://github.com/ansiblebook/ansiblebook.git
$ cd ansiblebook/ch23 && vagrant up
```

Если машина Vagrant недоступна по адресу *https://server03/*, то вам может потребоваться выполнить следующую команду внутри машины Vagrant, чтобы создать сетевой интерфейс, связанный с IP-адресом 192.168.56.13:

```
$ sudo systemctl restart network.service
```

Какие задачи решает Ansible Automation Platform

Ansible Automation Platform – не просто веб-интерфейс к Ansible. Она добавляет в Ansible некоторые дополнительные возможности, такие как управление доступом, проектами, реестрами и запуск заданий. Рассмотрим их поближе в этом разделе.

Управление доступом

В крупных организациях Ansible Automation Platform помогает управлять автоматизацией посредством делегирования. Вы можете создать организацию для каждого отдела, а локальный системный администратор – группы с ролями и добавить в них сотрудников, наделив их правами для управления хостами и устройствами, насколько это необходимо для выполнения служебных обязанностей.

Ansible Automation Platform создавалась с учетом разделения обязанностей – весьма мощной идеи при правильном применении. Представьте, что разработчики сценария – это другие люди, не являющиеся владельцами инфраструктуры. Попробуйте создать репозиторий для ваших сценариев и еще один для реестра, чтобы команда со своими машинами могла создать еще один *реестр* для повторного использования ваших сценариев. Ansible Automation Platform поддерживает концепцию *организаций* с *командами*, каждая из которых имеет разные уровни разрешений.

Ansible Automation Platform действует как защита для хостов. При ее использовании ни одна группа и ни один работник не должны иметь прямого доступа к управляемым хостам. Это снижает сложность и увеличивает безопасность. На рис. 23.6 показан веб-интерфейс Ansible Automation Platform для управления пользователями. С Ansible Automation Platform также можно использовать другие системы аутентификации, такие как Azure AD, GitHub, Google OAuth2, LDAP, RADIUS, SAML или TACACS+. Соединение Ansible Automation Platform с существующими системами аутентификации, такими как каталоги LDAP, может снизить снизить затраты на администрирование пользователей.

Проекты

Проектом в терминологии Ansible Automation Platform называется пакет логически связанных сценариев и ролей.

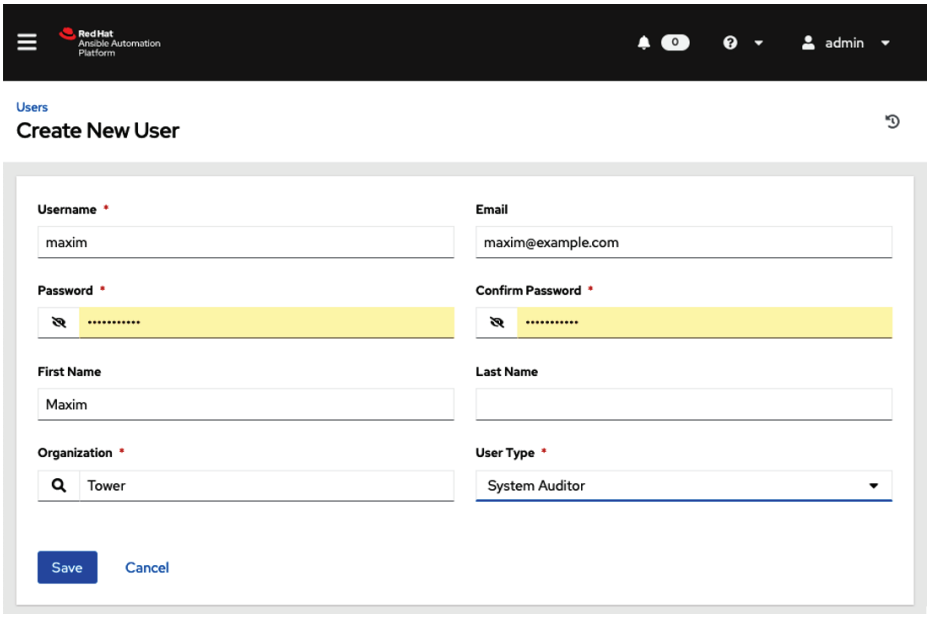


Рис. 23.6. Веб-интерфейс для управления пользователями

В классических проектах Ansible вместе со сценариями и ролями часто можно видеть статические реестры вместе со сценариями и ролями. Ansible Automation Platform осуществляет инвентаризацию отдельно. Все, что имеет отношение к инвентаризации и связанным с ней переменным, таким как переменные групп или хостов, будет недоступно.

Цель (например, `hosts: <target>`) в этих сценариях особенно важна. Старайтесь использовать общие имена. Это позволит вам выполнять сценарии с разными реестрами, о чем подробнее рассказывается далее в этой главе.

Следуя общепринятым рекомендациям, мы храним свои проекты со сценариями в системе управления версиями. Механизм управления проектами в Ansible Automation Platform поддерживает такие системы, как Git, Mercurial и Subversion, и может быть настроен на загрузку проектов из них.

В крайнем случае, если нет возможности использовать систему управления версиями, можно определить статический путь в каталоге */var/lib/awx/projects*, где проект будет храниться локально, на сервере Ansible

Automation Platform. Также есть возможность загружать архивы из удаленного хранилища.

Так как проекты имеют свойство развиваться с течением времени, исходный код сценариев на сервере Ansible Automation Controller должен синхронизироваться с содержимым системы управления версиями. Для этого в Ansible Automation Platform имеется множество решений.

Например, гарантировать использование последних версий проектов в Ansible Automation Platform можно, установив флажок «Update on Launch» (обновление на запуске) в параметрах проекта, как показано на рис. 23.7. Также можно настроить задания обновления проектов по расписанию. Наконец, проекты можно обновлять вручную, если вы хотите сами управлять обновлением.

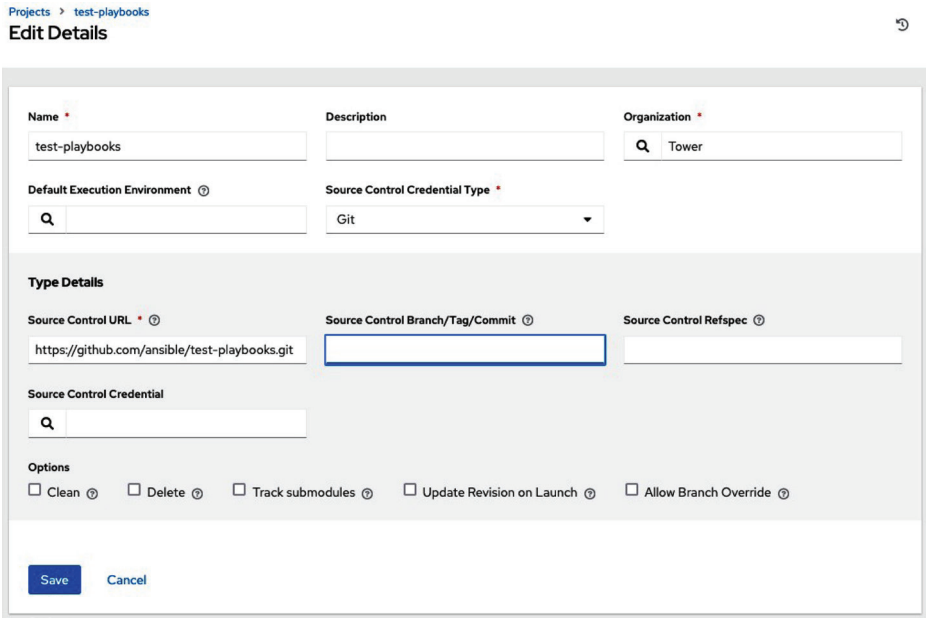


Рис. 23.7. Параметры настройки обновления проекта из системы управления версиями в Ansible Automation Controller

Управление инвентаризацией

Ansible Automation Platform позволяет управлять реестрами как самостоятельными ресурсами, включая управление доступом к этим реестрам. Типичный шаблон – определить разные реестры с хостами для эксплуатации, обкатки и тестирования и своими учетными данными и переменными.

В каждом из реестров можно определить свои переменные по умолчанию и вручную добавлять группы и хосты. Кроме того, как показано на рис. 23.8, Ansible Automation Platform позволяет запрашивать спи-

сок хостов динамически из некоторого ресурса (например, из Microsoft Azure Resource Manager) и помещать их в группу.

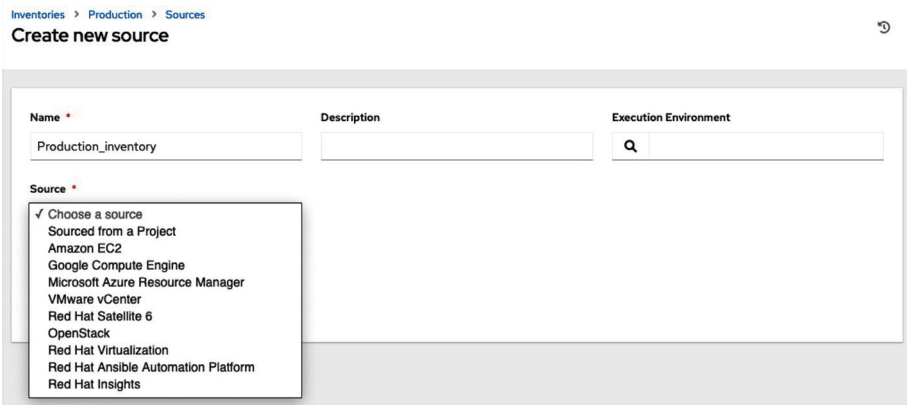


Рис. 23.8. Выбор источника информации о хостах в Ansible Automation Controller

С помощью специальной формы можно добавлять переменные групп и хостов и переопределять значения по умолчанию.

Также есть возможность временно отключать хосты, щелкая по кнопкам, как показано на рис. 23.9, и тем самым исключать их из обработки.

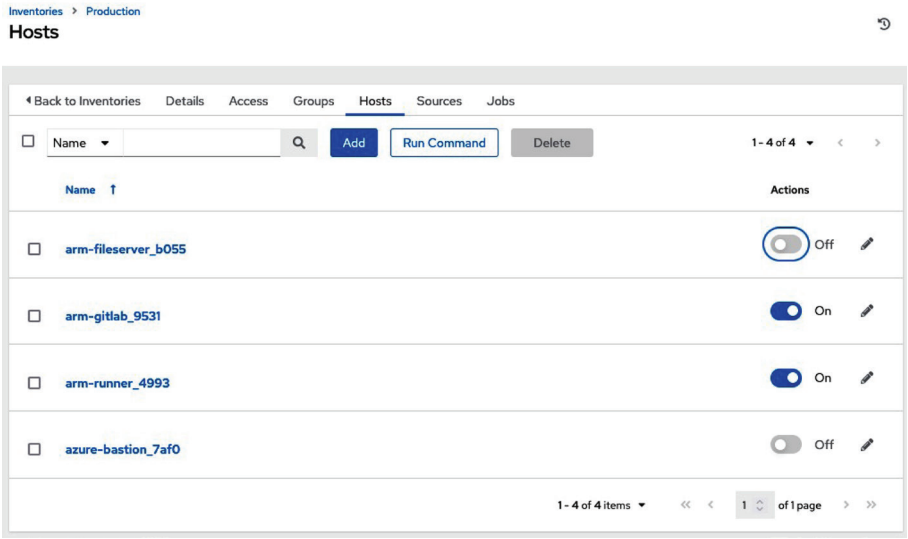


Рис. 23.9. Исключение хостов из обработки в Ansible Automation Platform

Запуск заданий из шаблонов

Шаблоны заданий, как показано на рис. 23.10, связывают проекты с реестрами. Они определяют, как пользователи могут запускать сценарии из проекта на определенных хостах из выбранного реестра.

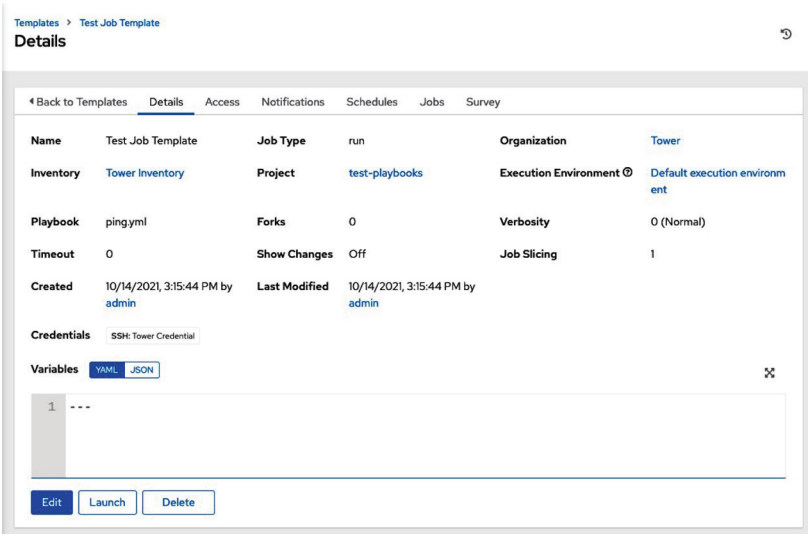


Рис. 23.10. Шаблоны заданий в Ansible Automation Platform

На уровне сценария можно применять такие уточнения, как дополнительные параметры и теги. Также есть возможность указать *режим* запуска сценария. Например, одним пользователям можно позволить запускать сценарии только в *режиме проверки*, а другим – только на определенном подмножестве хостов, зато в *полноценном режиме*.

На уровне целей есть возможность выбирать определенные хосты и группы.

Для выполняемого шаблона создается новая *запись задания*, как показано на рис. 23.11.

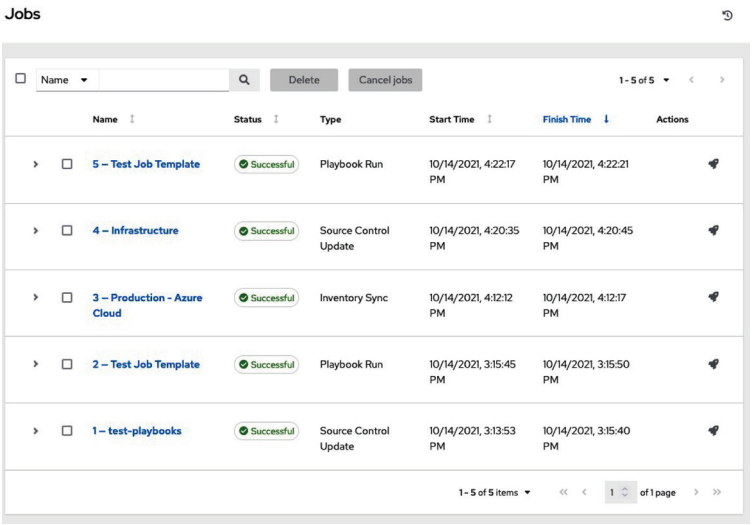


Рис. 23.11. Записи заданий в Ansible Automation Platform

В детальном обзоре каждой записи, как показано на рис. 23.12, приводится информация не только об успехе или неудаче его выполнения, но также о дате и времени запуска задания, о моменте его завершения, кто его запустил и с какими параметрами. Есть возможность даже выполнять фильтрацию по операциям, чтобы увидеть все задачи и их результаты. Вся эта информация сохраняется в базе данных, что дает возможность исследовать ее в любой момент.

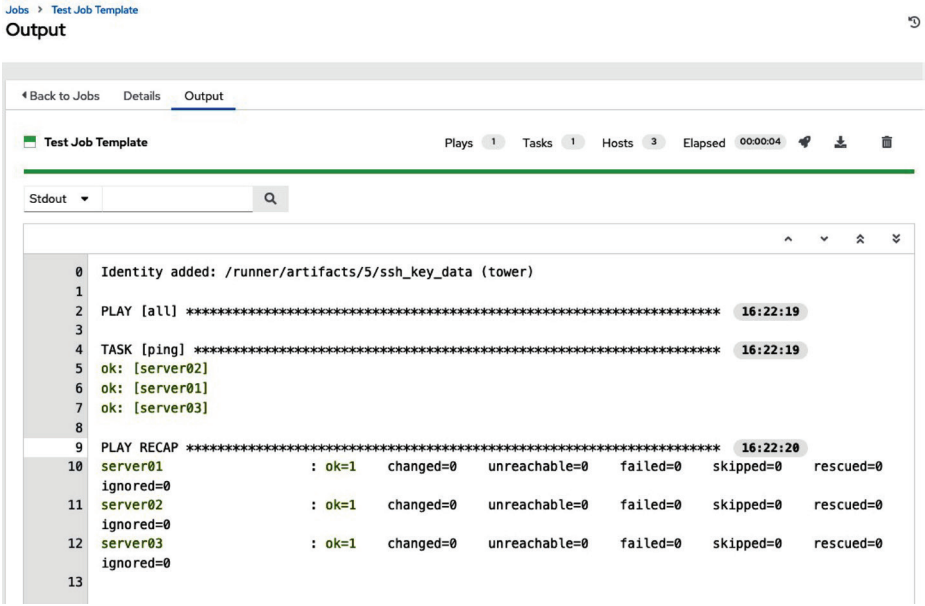


Рис. 23.12. Подробный обзор результатов задания в Ansible Automation Platform

RESTful API

Сервер Ansible Automation Controller поддерживает REST API (Representational State Transfer – программный интерфейс передачи представления о состоянии), позволяющий интегрировать его с имеющимися конвейерами сборки и установки или системами непрерывного развертывания.

API можно исследовать с помощью браузера, открывая в нем страницы с адресами вида *http://<tower_server>/api/v2/* (рис. 23.13):

```
$ firefox https://server03/api/v2/
```

На момент написания этих строк последней версией API была версия `v2`.

Теоретически API можно использовать для нужд интеграции, но вообще для доступа к Ansible Automation Controller существует коллекция Ansible: `awx.awx`.

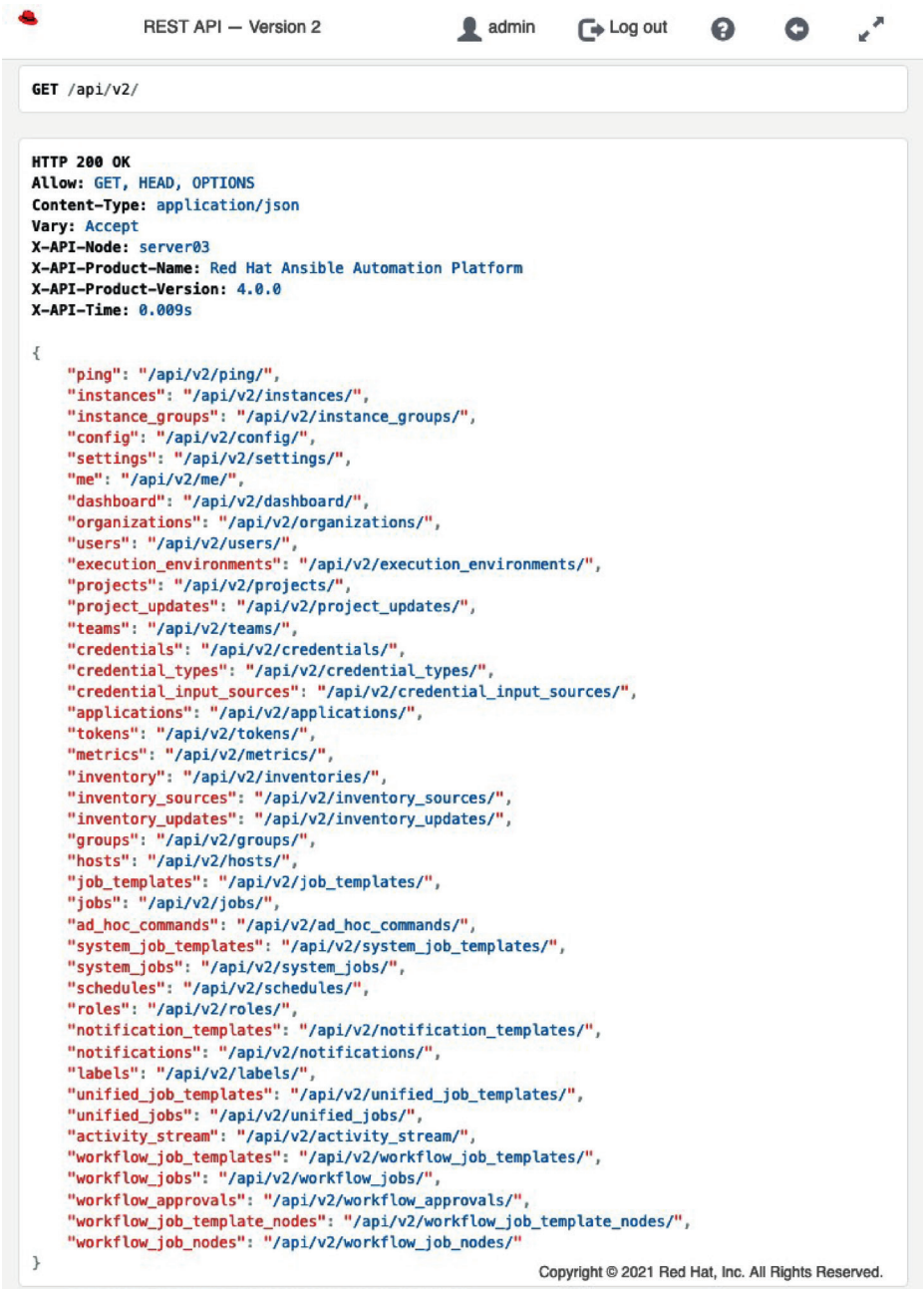


Рис. 23.13. Ansible Automation Platform API версии 2

AWX.AWX

Итак, как создать нового пользователя в Ansible Automation Controller или запустить задание, используя только API? Конечно, можно восполь-

зоваться самым популярным инструментом командной строки cURL, но Ansible предлагает еще более удобный способ: сценарии!

В отличие от приложения Ansible Automation Platform, Ansible Tower CLI – это программное обеспечение с открытым исходным кодом, опубликованное на GitHub (**https://oreil.ly/ryjSo**) под лицензией Apache 2.0.

Установка

Чтобы установить `awx.awx`, используйте Ansible Galaxy:

```
$ ansible-galaxy collection install awx.awx
```

Поскольку Ansible Automation Platform использует предварительно настроенный самоподписанный сертификат SSL/TLS, отключите проверку в шаблоне для файла *tower_cli.cfg*:

```
[general]
host = https://{{ awx_host }}
verify_ssl = false
oauth_token = {{ awx_token }}
```

Прежде чем обратиться к API, нужно настроить учетные данные в дополнительной переменной `admin_password`, как показано в примере 23.3.

Пример 23.3. *awx-config.yml*

```
---
- name: Configure awx
  hosts: automationcontroller
  become: false
  gather_facts: false

  vars:
    awx_host: "{{ groups.automationcontroller[0] }}"
    awx_user: admin
    cfg: "-k --conf.host https://{{ awx_host }} --conf.user {{ awx_user }}"

  tasks:

    - name: Login to Tower
      delegate_to: localhost
      no_log: true
      changed_when: false
      command: "awx {{ cfg }} --conf.password {{ admin_password }} -k login"
```

```
      register: awx_login

    - name: Set awx_token
      delegate_to: localhost
      set_fact:
        awx_token: "{{ awx_login.stdout | from_json | json_query('token') }}"

    - name: Create ~/.tower_cli.cfg
      delegate_to: localhost
      template:
        src: tower_cli.cfg
        dest: "~/.tower_cli.cfg"
        mode: '0600'
...
```

В результате будет создан файл *~/.tower_cli.cfg* с токеном. Теперь можно создать сценарий для автоматизации Automation Controller – автоматизации нового уровня!

Создание организации

Модель данных, показанная на рис. 23.13, требует наличия некоторых объектов, прежде чем можно будет создать другие, поэтому первое, что нужно создать, – добавить организацию:

```
---
- name: Configure Organization
  hosts: localhost
  gather_facts: false
  collections:
    - awx.awx

  tasks:

    - name: Create organization
      tower_organization:
        name: "Tower"
        description: "Tower organization"
        state: present

    - name: Create a team
      tower_team:
        name: "Tower Team"
        description: "Tower team"
        organization: "Tower"
        state: present
```

Все ссылки указывают либо на организации, либо на реестры.

Создание реестра

Для нашего примера мы с помощью коллекции `awx.awx` создали простой реестр Ansible Automation Platform . Обычно модулю `tower_project` передается ссылка на Git-репозиторий, а `tower_inventory_source` привязывается к `tower_inventory`:

```
---
- name: Configure Tower Inventory
  hosts: localhost
  gather_facts: false
  collections:
    - awx.awx

  tasks:

    - name: Create inventory
      tower_inventory:
        name: "Tower Inventory"
        description: "Tower infra"
        organization: "Tower"
        state: present

    - name: Populate inventory
      tower_host:
        name: "{{ item }}"
        inventory: "Tower Inventory"
        state: present
      with_items:
        - 'server01'
        - 'server02'
        - 'server03'

    - name: Create groups
      tower_group:
        name: "{{ item.group }}"
        inventory: "Tower Inventory"
        state: present
        hosts:
          - "{{ item.host }}"
      with_items:
        - group: automationcontroller
          host: 'server03'
        - group: automationhub
          host: 'server02'
        - group: database
          host: 'server01'
```

Если вы создаете и уничтожаете виртуальные машины с помощью Ansible, то тем самым вы управляете реестром.

Запуск сценария с помощью шаблона задания

Если вы привыкли запускать сценарии, используя только Ansible Core в командной строке, то, вероятно, часто используете привилегии администратора. В Ansible Automation Platform есть способ смоделировать это в виде защищенного окружения.

Сценарии хранятся в системе управления исходным кодом, такой как Git. *Проект* соответствует такому репозиторию Git. Импортировать проект можно с помощью модуля `tower_project`:

```
- name: Create project
  tower_project:
    name: "test-playbooks"
    organization: "Tower"
    scm_type: git
    scm_url: https://github.com/ansible/test-playbooks.git
```

Перед запуском сценария Ansible в командной строке вы, вероятно, настраиваете ключи SSH или другой способ входа в целевые системы, перечисленные в реестре. При таком способе запуска сценарий привязывается к вашей учетной записи пользователя на управляющем хосте Ansible. При использовании Ansible Automation Platform *учетные данные машины* сохраняются в (зашифрованной) базе данных платформы.

SSH-ключи являются конфиденциальными данными, но есть способ добавить зашифрованные закрытые ключи в Ansible Automation Controller, чтобы он запрашивал парольную фразу при запуске шаблона задания, в котором эти ключи используются:

```
- name: Create machine credential
  tower_credential:
    name: 'Tower Credential'
    credential_type: Machine
    ssh_key_unlock: ASK
    organization: "Tower"
    inputs:
      ssh_key_data: "{{ lookup('file', 'files/tower_ed25519') }}"
```

Теперь, создав проект и реестр и организовав доступ к машинам с их учетными данными, можно создать *шаблон задания* для запуска сценария из проекта на машинах, перечисленных в реестре:

```
- name: Create job template
  tower_job_template:
    name: "Test Job Template"
    project: "test-playbooks"
```

```
    inventory: "Tower Inventory"
    credential: 'Tower Credential'
    playbook: ping.yml
```

Почти наверняка у вас появится желание автоматизировать выполнение задания из шаблона. Коллекция `awx.awx` упрощает эту задачу – вам достаточно знать имя шаблона задания для запуска:

```
- name: Launch the Job Template
      tower_job_launch:
        job_template: "Test Job Template"
```

Шаблоны заданий чрезвычайно полезны для выполнения стандартных процедур. Примеры, представленные выше, вы с легкостью сможете опробовать в своей системе разработки. Если ваш шаблон задания предназначен для использования несколькими командами, то организуйте запрос на ввод реестра и учетных данных перед запуском шаблона задания. Так вы сможете делегировать все виды стандартных задач командам в их инфраструктурных окружениях.

Запуск Ansible в контейнерах

Контейнеры упрощают работу с Ansible в двух областях. Одна из них – тестирование ролей Ansible с помощью фреймворка Molecule (**https://oreil.ly/cQr6T**), обсуждавшегося в главе 14.

Второй аргумент в пользу использования контейнеров – сложности с внешними зависимостями, разные для разных проектов или команд. Когда требуется импортировать библиотеки Python и внешние компоненты Ansible, такие как роли, модули, плагины и коллекции, создание и использование образов контейнеров может помочь обеспечить их актуальность для долгосрочного использования. Пакеты Linux, Python, Ansible, роли и коллекции Ansible постоянно обновляются. Часто бывает сложно создать одну и ту же среду выполнения для Ansible на нескольких машинах или в разные моменты времени. Среды выполнения (**https://oreil.ly/hpefh**) – это согласованный, воспроизводимый и переносимый метод, позволяющий организовать выполнение заданий Ansible Automation на вашем ноутбуке в точно такой же среде, как на платформе AWX/Ansible Automation Platform.

Создание сред выполнения

Создание сред выполнения Ansible – сложная тема, но этот прием может вам понадобиться при работе с Ansible Automation Platform 2. Среды выполнения возникли в результате работы над библиотекой Python `ansible-runner` (**https://oreil.ly/bkOei**). Они создаются с помощью Podman на RHEL 8 и инструмента Python под названием `ansible-builder` (**https://oreil.**

ly/1vpq5). (Podman – это среда выполнения контейнера для разработчиков на RHEL 8).

Давайте посмотрим, как создать среду выполнения. Сначала создадим виртуальное окружение для работы с `ansible-builder` и `ansible-runner`:

```
$ python3 -m venv .venv
```

Активируем виртуальное окружение и обновим инструменты:

```
$ source .venv/bin/activate
$ python3 -m pip install --upgrade pip
$ pip3 install wheel
```

Затем установим `ansible-builder` и `ansible-runner`:

```
$ pip3 install ansible-builder
$ pip3 install ansible-runner
```

Ansible Builder требует определить файл с именем *execute-environment.yml*:

```
---
version: 1

ansible_config: 'ansible.cfg'

dependencies:
  galaxy: requirements.yml
  python: requirements.txt
  system: bindep.txt

additional_build_steps:
  prepend: |
    RUN pip3 install --upgrade pip setuptools
  append:
    - RUN yum clean all
```

Библиотеки Python должны быть перечислены в файле *requirements.txt*, а зависимости Ansible – в файле *requirements.yml*. Для двоичных зависимостей, таких как пакеты `git` и `unzip`, используется файл нового типа *bindep.txt*:

```
git [platform:rpm]
unzip [platform:rpm]
```

Определив среду выполнения, создадим ее:

```
$ ansible-builder \
    --build-arg ANSIBLE_RUNNER_IMAGE=quay.io/ansible/ansible-runner:stable-2.11-latest
\
    -t ansible-controller -c context --container-runtime podman
```

Чтобы использовать среду выполнения, создадим сценарий-обертку вокруг этой команды:

```
$ podman run --rm --network=host -ti \
    -v${HOME}/.ssh:/root/.ssh \
    -v ${PWD}/playbooks:/runner \
    -e RUNNER_PLAYBOOK=playbook.yml \
    ansible-controller
```

Заключение

Ansible Automation Platform 2 – это продукт автоматизации ИТ для предприятий. Контроллер автоматизации Automation Controller (ранее известный как Ansible Tower) предлагает возможности управления доступом на основе ролей, разделения обязанностей и делегирования полномочий. Проекты Ansible извлекаются из системы управления версиями, есть возможность безопасно управлять учетными данными, распределять ресурсы и учитывать каждое системное изменение. Это позволяет организациям с сотнями команд управлять десятками тысяч машин. Недаром стоимость лицензии рассчитывается по количеству хостов.

Automation Hub предлагает коллекции Ansible, созданные партнерами Red Hat, которые позволяют администраторам курировать контент сообщества и ограничивать или подменять доступ к Ansible Galaxy.

Среды выполнения Ansible в Ansible Automation Platform 2 изолируют программные зависимости в контейнерах, что обеспечивает бóльшую гибкость, чем виртуальные окружения, используемые в Ansible Tower. Вы можете с легкостью хранить технический долг Ansible (требуемые конкретные версии, конфликтующие библиотеки и т. д.) в нескольких контейнерах. Среды выполнения могут создаваться командами, а не администратором, что позволяет сократить время их подготовки.

Глава 24

Практические рекомендации

В этой главе мы предлагаем вашему вниманию набор обобщенных практических рекомендаций, но имейте в виду, что практические рекомендации редко бывают универсальными для разных контекстов. То, что хорошо для Spotify или Netflix, может не подходить для других компаний. Наша главная цель – заставить вас подумать о возможности использования подходящих вам рекомендаций и обсудить те, что могут вызвать у вас беспокойство. Практические рекомендации основаны на принципах проектирования и опыте использования Ansible в различных условиях. На уровне управления необходимо учитывать особенности труда специалистов-практиков и оценки команд DevOps.

Простота, модульность и сочетаемость

Майкл ДеХаан (Michael DeHaan) создавал Ansible для автоматизации рутинных задач самым простым из мыслимых способов, потому что хотел тратить свое время на что-то более интересное. Теперь неопытные пользователи могут заглянуть на сайт Ansible Galaxy (**https://galaxy.ansible.com/**), отыскать нужные им роли и коллекции и в течение нескольких часов организовать решение некоторых задач с помощью Ansible.

С тех пор как ДеХаан и Грег ДеКенигсберг (Greg DeKoenigsberg) основали сообщество Ansible, они обдумывали и формулировали практические рекомендации (**https://oreil.ly/ubBBZ**), однако в документации к версии 2.10 название «практические рекомендации» (**https://oreil.ly/Yp36l**) было заменено на «советы и рекомендации» (**https://oreil.ly/0pOeP**). Они отмечают, что проекты с открытым исходным кодом с большей вероятностью привлекут и удержат участников, если обладают двумя особыми свойствами: высокой модульностью и высокой ценностью возможностей. *Высокая модульность*, или *слабая связанность*, позволяет свободно расширять возможности Ansible. *Высокая ценность возможностей*, также известная как *сочетаемость*, позволяет выбирать, например, между Galaxy и Terraform, для подготовки инфраструктуры и Ansible к управлению системами. Сочетаемость также является одной из основ Дао HashiCorp (**https://oreil.ly/Kohiw**).

Организуйте контент

- Используйте GitHub, чтобы сохранить свой контент Ansible и сделать его доступным для других.
- Сохраняйте в репозитории все роли, коллекции, проекты и реестры.
- Следите за изменениями и одобрениями с помощью рабочего процесса, такого как GitHub Flow (**https://oreil.ly/kgyjK**).
- Управляйте своими зависимостями: дистрибутивами, пакетами, библиотеками, инструментами.
- Волшебство возможно, только если разместить файлы в нужных местах.
- Используйте правильные инструменты для работы: сначала попробуйте найти готовый модуль.
- Не решайте сложности с помощью Ansible; попробуйте написать модуль на Python.

Отделяйте реестры от проектов

- Делайте проекты многоразовыми для обслуживания нескольких пользователей.
- Разрешайте владельцам инфраструктуры определять доступ к хостам в реестре.
- Используйте реестры с группами, соответствующими функциям (или ролям).
- Комбинируйте проекты и реестры с помощью отдельных репозиториев Git.
- Создавайте окружения обкатки для исчерпывающего тестирования перед запуском.
- Используйте альтернативную структуру каталога (**https://oreil.ly/HH0VX**) для подготовки к переходу на AWX/Ansible Automation Platform.

Отделяйте роли и коллекции

- Помните, что роли – это способ автоматической загрузки переменных, файлов, задач, обработчиков и шаблонов на основе известной файловой структуры. Соглашение о конфигурации – мощный шаблон.
- Определяйте по одному действию для каждой роли.

- Коллекции состоят из ролей, модулей, плагинов и т. д. Тестируйте их как компоненты.
- Группируйте контент по ролям, чтобы упростить совместное использование с другими пользователями.
- Используйте манифест *roles/requirements.yml* для выражения зависимостей.
- Отделяйте роли проекта, общие роли и роли Galaxy. Настройте `roles_path` для поиска этих ролей.
- Используйте каталоги верхнего уровня: файлы, шаблоны для локальной реализации шаблонов ролей.
- Значения по умолчанию легко могут переопределяться пользователем с помощью `group_vars`.
- Переменные не предназначены для изменения пользователем.

Сценарии

- Старайтесь сделать сценарии максимально удобочитаемыми для неспециалистов (заметка на будущее).
- Представляйте желаемое состояние или простое изменение состояния декларативным способом.
- Определяйте безопасные настройки по умолчанию для новичков. Сделайте решение задач простым для всей команды.
- Если есть простое решение, используйте его.
- Делайте сценарии выполняемыми (со строкой `#!`), а файлы vars – нет.

Оформляйте код

- Форматируйте сценарии в стиле, характерном для YAML .
- Редакторы выбирают подсветку синтаксиса, основываясь на расширении файлов.
- Всегда давайте осмысленные имена своим сценариям, операциям и задачам, чтобы было понятно, что и как выполнялось при разборе журналов.
- Начинайте комментарии со знака решетки (`#`). Не злоупотребляйте комментариями и пустыми строками.
- Чтобы найти проблемы в контенте до отправки в репозиторий, используйте `ansible-lint` (**https://oreil.ly/HHMti**), `ansible-later` (**https://oreil.ly/zmXVV**), `yamllint` (**https://oreil.ly/4SW35**), `SonarQube` (**https://oreil.ly/07p8h**), `Pylint` (**https://oreil.ly/B6TRI**), `ShellCheck` (**https://oreil.ly/vX2mS**), `Perl::Critic` (**https://oreil.ly/hBnfg**) и любые другие статические анализаторы кода (линтеры), подходящие для вашего проекта.

Снабжайте тегами и тестируйте все, что только возможно

- Теги помогают организовать выполнение сценариев. Они позволяют запускать или пропускать части сценариев.
- Теги могут помочь в тестировании. Добавляйте задачи модульного тестирования с помощью тега `unitTest` (**https://oreil.ly/kBZYZ**).
- Используйте Molecule (**https://oreil.ly/iTjBY**) для тестирования ролей; проверяйте результаты.

Описывайте желаемое состояние

- Идемпотентность: одна и та же операция должна давать один и тот же результат снова и снова .
- Гарантируйте, что ничего не изменится, если ничто не должно не измениться.
- Никакой неопределенности: опишите желаемое состояние и используйте переменные для переключения состояния.
- Попробуйте поддерживать режим проверки.
- Тестируйте состояния, используя драйвер `delegated`: `molecule converge` и `molecule cleanup`.

Доставляйте непрерывно

- Старайтесь планировать подготовку и развертывание как можно раньше и как можно чаще.
- Используйте одни и те же сценарии в разных окружениях с разными учетными данными.
- Развертывайте изменения во всех окружениях поэтапно и максимально наглядно с помощью Tower или Jenkins с ARA.
- Используйте ключевое слово `serial` для организации непрерывного обновления.

Обеспечивайте безопасность

- Упростите управление зашифрованными переменными (**https://oreil.ly/15D7z**).
- Не входите в систему с привилегиями root. Не используйте учетные записи служб в интерактивном режиме.
- Создавайте пользователей и группы с минимальными привилегиями.
- Не храните учетные данные в реестре.

- Шифруйте учетные данные и токены с помощью `ansible-vault`.
- Используйте идентификаторы шифрования для разных уровней доступа.
- Документируйте все для облегчения аудита.
- Защищайте SSH (**https://oreil.ly/gTwbw**) и свои системы от атак.
- Запускайте `ssh-audit` (**https://oreil.ly/BIJwU**) для проверки криптозащиты (**https://oreil.ly/twN0f**) SSH.
- Подумайте о возможности использования ключей SSH, подписанных центром сертификации (**https://oreil.ly/J1GUT**).

Контролируйте развертывание

- Создавайте и храните пакеты программного обеспечения в репозитории, таком как Nexus (**https://oreil.ly/XOHiB**) или Artifactory (**https://oreil.ly/kI9AZ**).
- Выпуск программного обеспечения – это целостный этап, а не передача байтов.
- Управляйте конфигурацией приложений с помощью централизованной системы или рабочего процесса Git.
- Создавайте дымовые тесты, чтобы подтвердить запуск и проверить правильный порядок запуска.

Оценивайте эффективность

Если вы руководитель команды, скрам-мастер, владелец продукта или иной участник программного проекта, то вам понадобятся критерии оценки. CALMS – это платформа, оценивающая способность внедрять процессы DevOps, а также способ измерения успешности внедрения. Джез Хамбл (Jez Humble), соавтор книги «The DevOps Handbook» (IT Revolution Press)[1], придумал аббревиатуру CALMS, которая означает «Culture, Automation, Lean, Measurement and Sharing» (культура, автоматизация, бережливое производство, измерение и совместное использование).

Набор ключевых показателей эффективности внедрения передового опыта в области разработки программного обеспечения включает:

сотрудничество:

делятся ли члены команды техническими знаниями между собой и насколько активно команда сотрудничает с другими командами для интеграции приложений и окружений?

[1] *Джин Ким, Патрик Дебуа, Джон Уиллис и Джез Хамбл*. «Руководство по DevOps». Манн, Иванов и Фербер (МИФ), 2018. ISBN: 978-5-00100-750-0. – *Прим. перев.*

автоматизацию:

автоматизирует ли команда процесс развертывания приложений и окружений?

культуру:

стремится ли команда к совершенствованию и использованию передовых методов и общих принципов при создании и настройке приложений и окружений?

измерение:

подтверждает ли команда функциональные и нефункциональные требования (автоматически) перед развертыванием приложений в промышленном окружении?

совместное использование:

предоставляет ли и получает ли команда обратную связь, необходимую для контроля над решениями, которыми она управляет?

Контрольные показатели

Надлежащего применения практических рекомендаций должно быть достаточно для решения всех следующих проблем.

- Можно ли точно воспроизвести любое из окружений, включая версию операционной системы, конфигурацию сети, стек ПО, развернутое в ней приложение и его конфигурацию?
- Легко ли вносить дополнительные изменения в любой из отдельных элементов и развернуть это изменение в любом или во всех окружениях?
- Легко ли увидеть каждое изменение в конкретном окружении и проследить его, чтобы точно определить, что это за изменение, кто его внес и когда?
- Выполняются ли все действующие нормативные требования?
- Сможет ли каждый член команды с легкостью получить необходимую информацию и внести изменения? Или наша стратегия мешает эффективной доставке, увеличивая время цикла и ухудшая обратную связь?
- Возникает ли ощущение энтузиазма у нового члена команды при приеме на работу?

Заключительные слова

Написав все эти страницы, мы вряд ли можем утверждать, что вы сможете изучить Ansible за два часа и на третий развернуть NGINX и Postgres,

но после прочтения книги «Запускаем Ansible» вы можете попробовать научить своих коллег тому, чему научились сами, или даже поделиться демонстрационным проектом с группой единомышленников. Сообщество Ansible является глобальным! Если вы решите присоединиться к нему, то просто посетите страницу сообщества Ansible (**https://oreil.ly/7KNaF**). Участники проекта Ansible часто посещают каналы IRC, GitHub, Discord и Reddit для поддержки дискуссий и оказания помощи.

Если рядом с вами нет местной группы единомышленников, то создайте ее. Если она неактивна, то постарайтесь возродить ее. Именно так Бас начал свою деятельность в группе Ansible Benelux Meetup в 2014 году. Встречи единомышленников – это отличный способ узнать что-то новое и познакомиться с людьми, имеющими общие интересы. Бас с теплотой вспоминает дискуссии, демонстрации и семинары, которые проводились в разных местах Амстердама. Спасибо всем, кто участвовал во встречах!

Дорогие читатели, мы надеемся, что вы получили то, что искали, и достаточно узнали об Ansible, чтобы приступить к решению стоящих перед вами задач.

Удачи!

Библиография

Barrett, Daniel, Richard Silverman and Robert Byrnes. *SSH The Secure Shell: The Definitive Guide*. Sebastopol, CA: O'Reilly Media, 2005.

Bauer, Kirk. *Automating UNIX and Linux Administration*. New York: Apress, 2003.

Clark, Mike. *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications*. Raleigh, NC: Pragmatic Bookshelf, 2004.

Conway, Damien. *Perl Best Practices*. Sebastopol, CA: O'Reilly Media, 2005.

Dobies, Jason, and Joshua Wood. *Kubernetes Operators*. Sebastopol, CA: O'Reilly Media, 2020.

Duvall, Paul, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Upper Saddle River, NJ: Pearson Education, 2007[1].

Forsgren, Nicole, Jez Humble, and Gene Kim. *Accelerate: Building and Scaling High Performing Technology Organizations*. Portland, OR: IT Revolution, 2018[2].

Geewax, JJ. *Google Cloud Platform in Action*. Shelter Island, NY: Manning Publications, 2018.

Gift, Noah, and Jeremy Jones. *Python for Unix and Linux System Administration*. Sebastopol, CA: O'Reilly Media, 2008[3].

Hashimoto, Mitchell. *Vagrant: Up and Running*. Sebastopol, CA: O'Reilly Media, 2013.

Holzner, Steve. *Ant: The Definitive Guide*. Sebastopol, CA: O'Reilly Media, 2005.

Humble, Jeff, and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Upper Saddle River, NJ: Pearson Education, 2011[4].

Hunt, Andrew, and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Boston, MA: Addison-Wesley, 2000[5].

---

[1] *Поль М. Дюваль, Стивен Матиас и Эндрю Гловер*. Непрерывная интеграция. Улучшение качества программного обеспечения и снижение риска. Вильямс, 2016. ISBN: 978-5-8459-1408-8. – *Прим. перев.*

[2] *Форсгрен Николь, Хамбл Джез, Ким Джин*. Ускоряйся! Наука DevOps: Как создавать и масштабировать высокопроизводительные цифровые организации. Альпина PRO, 2022. ISBN: 978-5-6042881-1-5. – *Прим. перев.*

[3] *Ноа Гифт, Джереми М. Джонс*. Python в системном администрировании UNIX и Linux. Символ-Плюс, 2009. ISBN: 978-5-93286-149-3. – *Прим. перев.*

[4] *Джез Хамбл, Дейвид Фарли*. Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ (Signature Series). Вильямс, 2021. ISBN: 978-5-8459-1739-3, 978-0-321-60191-9. – *Прим. перев.*

[5] *Хант Э., Томас Д., Алексашин А.* Программист-прагматик. Путь от подмастерья к мастеру. Лори,

Jaynes, Matt. *Taste Test: Puppet, Chef, Salt, Ansible*. Self-published, 2014.

Kernighan, Brian, and Rob Pike. *The UNIX Programming Environment*. Hoboken, NJ: Prentice Hall, 1984[1].

Kim, Gene, Jez Humble, Patrick DeBois, and John Willis. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. Portland, OR: IT Revolution, 2016[2].

Kleppmann, Martin. Designing *Data-Intensive Applications*. Sebastopol, CA: O'Reilly Media, 2015[3].

Kurniawan, Yan. *Ansible for AWS*. Leanpub, 2016.

Limoncelli, Thomas A., Christina J. Hogan, and Strata R. Chalup. *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems*. Boston, MA: Addison-Wesley Professional, 2014.

Luksa, Marko. *Kubernetes in Action*. Shelter Island, NY: Manning Publications, 2018[4].

Mell, Peter, and Timothy Grance. *The NIST Definition of Cloud Computing*. NIST Special Publication 800-145, 2011.

Morris, Kief. *Infrastructure as Code: Dynamic Systems for the Cloud Age*. Sebastopol, CA: O'Reilly Media, 2021.

OpenSSH/Cookbook/Multiplexing, Wikibooks (**http://bit.ly/1bpeV0y**), October 28, 2014.

Oram, Andrew, and Steve Talbott. *Managing Projects with Make*. Sebastopol, CA: O'Reilly Media, 1986.

Reitz, Kenneth, and Tanya Schlusser. *The Hitchhiker's Guide to Python: Best Practices for Development*. Sebastopol, CA: O'Reilly Media, 2016[5].

Ryan, Mike, and Federico Lucifredi. *AWS System Administration*. Sebastopol, CA: O'Reilly Media, 2018.

Shafer, Andrew Clay. *Agile Infrastructure in Web Operations: Keeping the Data on Time*. Sebastopol, CA: O'Reilly Media, 2010.

Turnbull, James, and Jeffrey McCune. *Pro Puppet: Maximize and Customize Puppet's Capabilities for Your Environment*. New York: Apress, 2011.

2016. ISBN 0-201-61622-x. – *Прим. перев.*

[1] *Пайк Роб, Керниган Брайан У.* Unix. Программное окружение. Символ-Плюс, 2003. ISBN: 5-93286-029-4, 0-13-937681-X. – *Прим. перев.*

[2] *Джин Ким, Патрик Дебуа, Джон Уиллис и Джез Хамбл*. Руководство по DevOps. Манн, Иванов и Фербер (МИФ), 2018. ISBN: 978-5-00100-750-0. – *Прим. перев.*

[3] *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. Питер, 2023. ISBN: 978-5-4461-0512-0. – *Прим. перев.*

[4] *Лукша М.* Kubernetes в действии. ДМК-Пресс, 2019. ISBN: 978-5-97060-657-5. – *Прим. перев.*

[5] *Шлюссер Таня, Рейтц Кеннет*. Автостопом по Python. Питер, 2017. ISBN: 978-5-496-03023-6. – *Прим. перев.*

Об авторах

Бас Мейер (Bas Meijer) – программист-фрилансер и консультант по DevOps. По окончании университета в Амстердаме был одним из первопроходцев в веб-разработке в начале 1990-х. Затем работал в сфере торговли, облачной безопасности, авиапромышленности и в правительственных организациях. Получил звание Ansible Ambassador в 2014 году и HashiCorp Ambassador в 2020–2021.

Лорин Хохштейн (Lorin Hochstein) – ведущий инженер-программист в подразделении Chaos Team в компании Netflix, где занимается обеспечением высокой доступности служб Netflix. Соавтор книги «OpenStack Operations Guide» (O'Reilly), а также множества академических публикаций.

Рене Мозер (René Moser) живет в Швейцарии со своей женой и тремя детьми. Любит простые программы, которые легко масштабируются. Имеет диплом о высшем образовании в сфере информационных технологий. Участвует в жизни сообщества программного обеспечения с открытым кодом уже более 15 лет. В последнее время участвует в разработке ASF CloudStack, а также занимается интеграцией CloudStack в Ansible и написал уже более 30 модулей для поддержки CloudStack. Стал членом основной команды разработчиков Ansible в апреле 2016 года и в настоящее время работает старшим системным инженером в SWISS TXT.

Об изображении на обложке

На обложке «Запускаем Ansible» изображена корова голштино-фризской породы (Bos primigenius), которую в Северной Америке часто называют голштинской, а в Европе – фризской. Она была выведена в Европе, в Нидерландах, с целью получить коров, питающихся исключительно травой – самый богатый ресурс в этом районе, – в результате чего получилась черно-белая молочная порода. Голштино-фризская порода была завезена в США где-то между 1621 и 1664 годом, но она не вызывала интереса у американских селекционеров до 1830-х годов.

Животные этой породы отличаются крупными размерами, четкими черными и белыми пятнами и высокой продуктивностью молока. Черно-белая окраска является результатом искусственного отбора селекционерами. Телята рождаются крупными, весом 40–45 кг; зрелые голштинцы могут достигать в весе 580 кг и в холке до 1,5 м. Половая зрелость у этой породы наступает в возрасте 13–15 месяцев; срок беременности длится 9,5 месяца.

Коровы этой породы дают в среднем 7600 л молока в год; продуктивность племенных животных может достигать 8100 л в год, а в течение жизни могут производить до 26 000 л.

В сентябре 2000 года голштинцы оказались в центре жарких дискуссий, когда компания Hanoverhill Starbuck клонировала одно животное из замороженных клеток соединительной ткани, взятых у него за месяц до смерти. Клонированный экземпляр появился через 21 год и 5 месяцев после рождения оригинала.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой вымирания; все они очень важны для биосферы.

Изображение для обложки взято из второго тома энциклопедии Лидеккера (Lydekker) «Royal Natural History».

Предметный указатель

Символы

A

B

C

D

E

F

G

H

I

J

K

L

M

N

T

U

V

W

Y

А

Б

В

Г

Д

Ж

З

И

К

М

Н

О

П

P

C

Т

У

Ф

Х

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.galaktika-dmk.com.

Бас Мейер, Лорин Хохштейн и Рене Мозер

Запускаем Ansible

Простой способ автоматизации управления конфигурациями и развертыванием приложений

Третье издание

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Киселев А. Н.*
Корректор *Абросимова Л. А.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Печать цифровая. Усл. печ. л. 39,16.
Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

O'REILLY

Среди множества имеющихся инструментов управления конфигурациями Ansible выделяется своими преимуществами, такими как небольшой объем, отсутствие необходимости устанавливать что-либо на управляемые хосты и простота в изучении и освоении.
В этом обновленном третьем издании вы узнаете, как быстро приступить к использованию этого инструмента – для развертывания кода в промышленном окружении или автоматизации задач системного администрирования.
Авторы покажут вам, как писать сценарии Ansible и управлять удаленными серверами и помогут овладеть всей широтой возможностей этого замечательного инструмента. Вы увидите, что Ansible обладает всем, что только может вам понадобиться, и без лишних сложностей.

«Бас Мейер подхватил работу Лорина Хохштейна и Рене́ Мозера и обновил ее содержимое до этого третьего издания, дополнив эту и без того замечательную книгу. Она понравится всем без исключения пользователям Ansible – и начинающим, и умудренным опытом».

Ян-Пит Менс, консультант

Прочитав книгу, вы научитесь:

- автоматизировать управление конфигурациями и развертыванием;
- настраивать компьютеры с Linux и Windows и сетевые устройства;
- применять передовые практики Ansible;
- использовать новый формат коллекций;
- писать свои модули и плагины;
- генерировать контент Ansible для ПО с открытым программным кодом;
- создавать образы контейнеров и облачных экземпляров и настраивать облачную инфраструктуру;
- автоматизировать CI/CD окружений разработки;
- использовать Ansible Automation Platform для поддержки DevOps.

Бас Мейер – программист-фрилансер и консультант по DevOps. Один из первопроходцев в веб-разработке в начале 1990-х.

Лорин Хохштейн – ведущий инженер-программист в подразделении Chaos Team в компании Netflix и соавтор книги «OpenStack Operations Guide».

Рене́ Мозер – системный инженер из Швейцарии и разработчик ASF CloudStack. Занимается интеграцией CloudStack в Ansible.

ISBN 978-6-01763-867-2