O'REILLY®

«Я запоем прочитал рукопись первого издания книги за несколько часов. Лорин создал нечто невероятное, продемонстрировав Ansible со всех сторон. Я очень обрадовался, узнав, что он решил объединить свои усилия с Рене для работы над вторым изданием. Авторы проделали выдающуюся работу, показав как эффективно использовать необычайно полезный инструмент. По моему мнению, никто бы не смог справиться с этой задачей на столь же глубоком уровне».

– Жан-Пит Менс (Jan-Piet Mens), консультант

Среди множества систем управления конфигурациями Ansible обладает неоспоримыми преимуществами. Он минималистичен, не требует установки программного обеспечения на узлах, а также легок в освоении. Второе издание книги научит вас выстраивать продуктивную работу в кратчайшие сроки, будь вы разработчик, разворачивающий код в производственной среде, или системный администратор в поисках более эффективного решения для автоматизации.

Авторы книги расскажут вам, как написать сценарий (скрипт управления конфигурациями Ansible), установить контроль над удаленными серверами, а также задействовать мощный функционал встроенных декларативных модулей.

Вы поймете, что Ansible обладает всеми функциональными возможностями, которые вам необходимы, и той простотой, о которой вы мечтаете.

- узнайте, чем Ansible отличается от других систем управления конфигурациями;
- используйте формат файлов YAML для написания собственных сценариев;
- изучите пример полного сценария для развертывания нетривиального приложения;
- администрируйте машины Windows и автоматизируйте конфигурацию сетевых устройств;
- производите развертывание приложений на Amazon EC2 и других облачных платформах;
- используйте Ansible для создания образов Docker и развертывания контейнеров Docker.

Мы рекомендуем изучать книгу последовательно от начала и до конца, поскольку последующие главы основаны на содержании предыдущих. Книга написана в стиле учебного пособия, что дает возможность выполнять все операции на вашем компьютере во время ее чтения. Большинство примеров основано на веб-приложениях.

Лорин Хохштейн (Lorin Hochstein) *является старшим инженером по программному обеспечению (Senior Software Engineer) команды Chaos в компании Netflix. Он также работал старшим инженером по программному обеспечению в компании SendGrid Labs, был ведущим архитектором облачных сервисов (Lead Architect for Cloud Services) в компании Nimbis Services и занимал должность ученого в области компьютерных наук в Институте информатики Университета Южной Калифорнии (University of Southern California's Information Sciences Institute).*

Рене́ Мозер (René Moser) *занимает позицию системного инженера в компании Swiss, является разработчиком ASF CloudStack, автором интеграции CloudStack в Ansible и ключевым членом сообщества Ansible с 2016 года.*

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
orders@alians-kniga.ru
Оптовая продажа:
"Альянс-книга"
тел.(499)782-38-89
books@alians-kniga.ru

ISBN 978-5-97060-513-4

9 785970 605134 >

www.дмк.рф

Запускаем Ansible

O'REILLY®

Запускаем Ansible

Лорин Хохштейн
Рене́ Мозер

Лорин Хохштейн и Рене́ Мозер

Запускаем Ansible

Lorin Hochstein & René Moser

Ansible
Up & Running

Automating Configuration Management and Deployment the Easy Way

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Лорин Хохштейн и Рене́ Мозер

Запускаем Ansible

Простой способ автоматизации управления конфигурациями и развертыванием приложений

ДМК Пресс
издательство
Москва, 2018

УДК 004.4'234Ansible
ББК 32.972.1
М15

Хохштейн Л., Мозер Р.
М15 Запускаем Ansible / пер. с анг. Е. В. Филонова, А. Н. Киселева. – М.: ДМК Пресс, 2018. – 382 с.: ил.

ISBN 978-5-97060-513-4

Книга рассказывает о системе управления конфигурациями Ansible с множеством примеров продуктивной работы. Она минималистична, не требует установки программного обеспечения на узлах, и легка в освоении. Вы узнаете, как написать скрипт управления конфигурациями, установить контроль над удаленными серверами, а также задействовать мощный функционал встроенных модулей. Рассмотрено чем Ansible отличается от других систем управления конфигурациями, приведены примеры развертывания на различных облачных платформах.

Издание будет полезно разработчикам и системным администраторам, принимающим решения о выборе способов автоматизации.

УДК 004.4'234Ansible
ББК 32.972.1

ISBN 978-1-491-97980-8 (анг.)
ISBN 978-5-97060-513-4 (рус.)

Содержание

Я буквально проглотил рукопись первого издания «Установка и работа с Ansible» за несколько часов: Лорин прекрасно справился с задачей описания всех аспектов Ansible, и я был рад услышать, что он решил объединиться с Рене для подготовки второго издания. Эти два автора проделали громадную работу, чтобы показать нам, как пользоваться невероятно удобной утилитой, и я не могу вспомнить ни одного момента, которого бы они не охватили в полной мере.

– Ян-Пит Менс (Jan-Piet Mens), консультант

Впечатляющая глубина освещения Ansible. Эта книга прекрасно подойдет не только начинающим, но и опытным специалистам, желающим понять все тонкости использования продвинутых возможностей. Фантастический источник информации для стремящихся повысить свой уровень владения Ansible.

– Мэтт Джейнс (Matt Jaynes),
ведущий инженер, High Velocity Ops

Самое замечательное в Ansible – возможность начать с простого прототипа и быстро продвигаться к намеченной цели. Однако со временем начинает ощущаться нехватка знаний, которые порой трудно получить.

«Установка и работа с Ansible» – очень ценный источник, восполняющий эту нехватку и разъясняющий особенности Ansible с самых основ до сложностей работы с YAML и Jinja2. А благодаря наличию массы практичных примеров она позволяет получить представление, как другие автоматизируют свои окружения.

В течение последних нескольких лет, проводя теоретические и практические занятия, я всегда рекомендовал эту книгу своим коллегам и клиентам.

– Даг Виерс (Dag Wieers),
консультант и инженер-системотехник
в области систем на основе Linux,
долгое время участвовавший в разработке Ansible

Эта книга помогает быстро приступить к использованию системы управления конфигурациями Ansible и описывает ее во всех подробностях. В ней приводится большое количество подсказок и практических советов и охватывается широкий круг вариантов использования, включая AWS, Windows и Docker.

– Инго Йохим (Ingo Jochim),
руководитель отдела облачных реализаций, itelligence GMS/CIS

Лорин и Рене проделали большую работу, написав эту книгу. Авторы берут читателя за руку и ведут его через наиболее важные этапы создания и управления проектов Ansible. Эта книга намного больше, чем справочник, – она охватывает ряд важнейших концептуальных тем, отсутствующих в официальной документации. Это превосходный источник знаний для начинающих и практических идей для более опытных пользователей Ansible.

– Доминик Бартон (Dominique Barton),
инженер DevOps в confirm IT solutions

Предисловие

Разработка системы Ansible началась в феврале 2012-го с создания простого побочного проекта, и ее стремительное развитие стало приятным сюрпризом. Сейчас над продуктом работает порядка тысячи человек (а идеи принадлежат даже большему числу людей), и он широко используется практически во всех странах мира. И наверняка вам удастся обнаружить, по крайней мере, нескольких человек, использующих его, в сообществе знакомых вам ИТ-специалистов.

Привлекательность Ansible объясняется ее простотой. И правда, Ansible не несет в себе новых, но объединяет все лучшее из уже существующих идей, разработанных другими экспертами, делая их чуть более доступными.

Создавая Ansible, я старался найти для нее место где-то между решениями автоматизации ИТ-задач (естественная реакция на огромные коммерческие пакеты программного обеспечения) и простыми сценариями, минимально необходимыми для выполнения своей работы. Кроме того, мне хотелось заменить систему управления конфигурациями, развертыванием и организацией проектов и нашу библиотеку произвольных, но важных сценариев командной оболочки единой системой. Вот в чем состояла идея.

Могли ли мы убрать важные архитектурные компоненты из стека автоматизации? Устранив демоны управления и переложив работу на OpenSSH, система могла бы начать управление компьютерами незамедлительно без установки агентов на контролируемых машинах. Кроме того, система стала бы более надежной и безопасной.

Я заметил, что в предыдущих попытках создания систем автоматизации простые вещи заметно усложнялись, а написание сценариев автоматизации часто и надолго уводило меня в сторону от того, чему бы я хотел посвятить больше времени. В то же время мне не хотелось получить систему, на изучение которой не нужны месяцы.

Честно говоря, мне больше нравится писать программы, чем заниматься управлением системами автоматизации. Мне хотелось бы тратить на автоматизацию как можно меньше времени, чтобы высвободить его на решение более интересных задач. Ansible – это не та система, с которой приходится работать сутки напролет. Используя ее, вы сможете зайти, что-то поправить, выйти и продолжить заниматься своими делами. Я надеюсь, что эта черта Ansible понравится вам.

Хотя я потратил много времени, стараясь сделать документацию для Ansible исчерпывающей, всегда полезно взглянуть на одни и те же вещи под разными углами. Полезно увидеть практическое применение справочной документации. В книге «Установка и работа с Ansible» Лорин представляет Ansible, используя идиоматический подход, в точности как следовало бы изучать эту

систему. Лорин работал с Ansible практически с самого начала, и я очень благодарен ему за его вклад.

Я также безмерно благодарен каждому, кто принимал участие в проекте до настоящего времени, и каждому, кто подключится к нему будущем. Наслаждайтесь книгой и получайте удовольствие от управления вашим компьютерным флотом! И не забудьте установить cowsay!

– *Майкл ДеХаан* (*Michael DeHaan*)
Создатель Ansible (программной части),
бывший технический директор компании Ansible, Inc.
апрель 2015

Предисловие ко второму изданию

За время, прошедшее с момента выхода первого издания (еще в 2014 году), в мире Ansible произошли большие изменения. Проект Ansible достиг следующей старшей версии 2.0. Также большие изменения произошли за рамками проекта: Ansible, Inc. – компания, стоящая за проектом Ansible, – была приобретена компанией Red Hat. Это никак не повлияло на разработку проекта Ansible: он так же активно развивается и привлекает новых пользователей.

Мы внесли множество изменений в это издание. Наиболее заметным стало появление пяти новых глав. Теперь книга охватывает плагины обратного вызова, хосты под управлением Windows, сетевое оборудование и Ansible Tower. Мы добавили в главу «Сложные сценарии» так много нового, что пришлось разбить ее на две части и добавить главу «Настройка хостов, запуск и обработчики». Мы также переписали главу «Docker», включив в нее описание новых модулей Docker.

Мы обновили все примеры кода для совместимости с Ansible 2.3. Например, устаревшую инструкцию `sudo` мы повсюду заменили более новой `become`. Мы также удалили ссылки на устаревшие модули, такие как `docker`, `ec2_vpc` и `ec2_ami_search`, и заменили их более новыми. Глава «Vagrant» теперь охватывает локальные сценарии вызова Ansible, глава «Amazon EC2» – Packer Ansible, механизм удаленного вызова, глава «Ускорение работы Ansible» – асинхронные задания, а глава «Отладка сценариев Ansible» – новые средства отладки, появившиеся в версии 2.1.

Также было внесено множество мелких изменений. Например, мы отказались от использования контрольных сумм MD5 в OpenSSH и перешли на хэши SHA256, внеся соответствующие изменения в примеры. Наконец, мы исправили ошибки, обнаруженные нашими читателями.

Примечание к стилю изложения

Первое издание книги было написано одним автором, и в нем часто использовалось местоимение «я» первого лица. Это издание написано уже двумя авторами, поэтому употребление местоимения в первом лице кое-где может показаться странным. Тем не менее мы решили не исправлять его, потому что в большинстве случаев оно используется для выражения мнения одного из авторов.

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 Ровану (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 отзыв о книге, включая объяснения, почему в качестве названия было выбрано *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.

Предисловие к первому изданию

Почему я написал эту книгу

Когда я писал свое первое веб-приложение, используя Django, популярный фреймворк на Python, я запомнил чувство удовлетворения, когда приложение наконец-то заработало на моем компьютере. Я запустил команду `django manage.py runserver`, указал в браузере *http://localhost:8000* и увидел свое веб-приложение во всей его красе.

Потом я подумал про все эти… *моменты*, которые необходимо учесть, чтобы просто запустить это чертово приложение на Linux-сервере. Кроме Django и моего приложения, мне потребовалось установить Apache и модуль `mod_python`, чтобы Apache мог работать с приложениями Django. Затем мне пришлось установить правильные значения в конфигурационном файле Apache, заставлявшие мое приложение работать и правильно обслуживать статичные компоненты.

Это было несложно – немного усилий, и готово. Мне не хотелось завязнуть в работе с файлами конфигурации, я лишь хотел, чтобы мое приложение работало. И оно работало, и все было прекрасно… пока через несколько месяцев мне не понадобилось запустить его снова на другом сервере и проделать всю ту же работу с самого начала.

В конце концов, я осознал, что все, что я делал, я делал неправильно. Правильный способ решать такого рода задачи имеет название, и это название – *управление конфигурациями*. Самое замечательное в управлении конфигурациями – полученные знания всегда сохраняют свою актуальность. Больше нет необходимости рыться в поисках нужной страницы в документации или копаться в старых записях.

Недавно коллега заинтересовался применением Ansible для внедрения нового проекта и спросил, как можно использовать идею Ansible на практике, кроме того что указано в официальной документации. Я не знал, что посоветовать почитать, и решил написать книгу, которая восполнит этот пробел, – и вот вы видите эту книгу перед собой. Увы, для него эта книга вышла слишком поздно, но я надеюсь, она окажется полезной для вас.

Кому адресована эта книга

Эта книга для всех, кто работает с Linux- или Unix-подобными серверами. Если вы когда-либо использовали термины *системное администрирование*, *раз-*

вертывание, *управление конфигурациями* или (вздох) *DevOps*, вы обязательно найдете для себя что-то полезное.

Хотя я изучал Linux-серверы, моя квалификация связана с разработкой программного обеспечения. А это значит, что все примеры в книге более тяготеют к внедрению программного обеспечения, хотя мы с Эндрю Клей Шафером (Andrew Clay Shafer, [webops]) пришли к тому, что внедрение и конфигурация не имеют четкой границы.

Структура книги

Я не большой фанат общепринятых принципов структурирования книг: глава 1 охватывает то-то и то-то, глава 2 охватывает это и то и тому подобное. Я подозреваю, что никто не читает этих строк (я лично – никогда), гораздо проще заглянуть в оглавление.

Книга построена так, что каждая последующая глава опирается на предыдущую. Таким образом, я предполагаю, что вы будете читать книгу от начала и до конца. Книга написана в основном в стиле учебного пособия и дает возможность выполнять примеры на вашем компьютере в процессе чтения. Большинство примеров основано на веб-приложениях.

Обозначения и соглашения, принятые в этой книге

В книге действуют следующие типографские соглашения:

Курсив
: Указывает на новые термины, названия файлов и их расширения.

`Моноширинный шрифт`
: Используется для 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`
: Служит для выделения команд или другого текста, который должен быть набран самим пользователем.

`Моноширинный курсив`
: Указывает на текст, который нужно заменить данными пользователя, или значениями, определяемыми контекстом.


Так обозначаются примечания общего характера.


Так обозначаются советы и рекомендации.


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: «*Хохштейн Л., Мозер Р.* Запускаем Ansible. М.: O'Reilly; ДМК Пресс, 2018. Copyright © 2017 O'Reilly Media, Inc., 978-1-491-97980-8 (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.

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.

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.

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[1]. Вместо этого мы медленно превращаемся в системных инженеров.

Сейчас мы устанавливаем 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 кэширования данных в оперативной памяти, очереди задач, очереди сообщений, базы данных SQL, системы хранения данных, NoSQL-хранилища и балансировщики нагрузки.

Мы также должны убедиться в наличии достаточного количества ресурсов, и в случае падения системы (а она будет падать) 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[2].

Мы можем связать эти службы вручную: «прикрутить» нужные серверы, зайдя на каждый из них, установив пакеты приложений, отредактировав файлы конфигурации, и т. д. Но это серьезный труд. Такой процесс требует много времени, способствует появлению множества ошибок, да 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 управления. Кем

---

[1] Да, мы согласны, никто и никогда на самом деле так не поставлял программное обеспечение.

[2] Рекомендую превосходные книги «The Practice of Cloud System Administration» и «Designing Data-Intensive Applications» по созданию и поддержке этих типов распределенных систем.

бы вы ни были, разработчиком или системным администратором, ищущим лучшего 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 2.3.0.0, которая на момент написания книги являлась самой свежей. Поскольку поддержка предыдущих версий является важной целью проекта Ansible, эти примеры должны поддерживаться и последующими версиями в неизменном виде.

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 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

Систему Ansible часто описывают как средство управления конфигурациями, и обычно она упоминается в том же контексте, что и *Chef*, *Puppet* и *Salt*. Когда мы говорим об управлении конфигурациями, то часто подразумеваем некое описательное состояние серверов, а затем фиксацию их реального состояния с использованием специальных средств: необходимые пакеты приложений установлены, файлы конфигурации содержат ожидаемые значения и имеют требуемые разрешения в файловой системе, необходимые службы работают 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) и запуск служб. *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 об *инициализации* (provisioning) новых серверов. В контексте облачных услуг, таких как Amazon EC2, под инициализацией подразумевается развертывание нового экземпляра виртуальной машины. Ansible охватывает и эту область, предоставляя несколько модулей поддержки облаков, включая EC2, Azure, Digital Ocean, Google Compute Engine, Linode и Rackspace, а также любые облака, поддерживающие OpenStack API.

Несколько сбивает с толку использование термина *инициатор* в документации к утилите *Vagrant*, которую мы обсудим далее в этой главе, в отношении системы управления конфигурациями. Так, Vagrant называет Ansible своего рода инициатором там, где, как мне кажется, инициатором 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 Стейси, применяет 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 (Ansible называет их *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 файлы конфигурации для Nginx;
- скопировать сертификат безопасности;
- запустить Nginx.

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

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

Ansible устанавливает параллельные SSH-соединения с хостами *web1*, *web2* и *web3*. Выполняет первую задачу из списка на всех хостах одновременно.

В этом примере первая задача – установка apt-пакета Nginx (поскольку Ubuntu использует диспетчер пакетов `apt`). То есть данная задача в сценарии выглядит примерно так:

```
- name: install nginx
  apt: 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.

Далее Ansible приступит к следующей задаче в списке и повторит описанные эти же четыре шага. Важно отметить, что:

- каждая задача выполняется на всех хостах одновременно;
- Ansible ожидает, пока задача будет завершена на всех хостах, прежде чем приступить к выполнению следующей;
- задачи выполняются в установленном вами порядке.

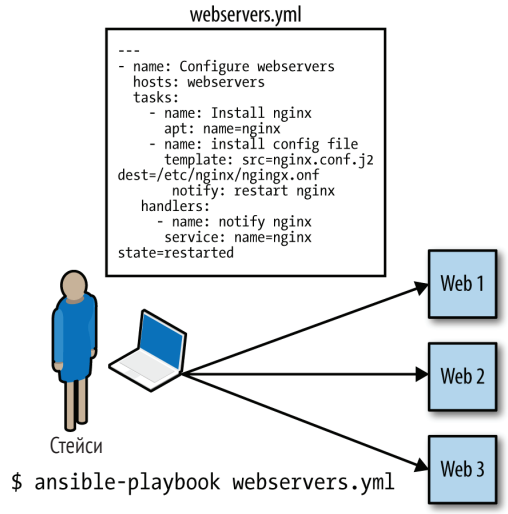

Рис. 1.1 ❖ Ansible выполняет сценарий настройки трех веб-серверов

Какие преимущества дает Ansible?

Существует несколько открытых систем управления конфигурациями. Ниже перечисляются некоторые особенности, привлекшие мое внимание к Ansible.

Простота синтаксиса

Напомню, что задачи управления конфигурациями в Ansible определяются в виде *сценариев* (playbooks). 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.

Мне нравится думать про сценарии Ansible как про выполняемую документацию. 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.

Отсутствие необходимости установки на удаленных хостах

Для управления серверами с помощью Ansible на них должна быть установлена поддержка SSH и Python версии 2.5 или выше либо Python 2.4 с библиотекой *simplejson*. Нет никакой необходимости устанавливать на хостах любое другое программное обеспечение.

На управляющей машине (той, что вы используете для управления удаленными машинами) должен быть установлен Python версии 2.6 или выше.

Некоторые модули могут потребовать установки Python версии 2.5 или выше, другие могут иметь иные требования. Обязательно проверяйте документацию по каждому модулю, чтобы понять, имеет ли он специфические требования.

Основан на 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:

1. Вы: вносите изменения в сценарий управления конфигурациями.
2. Вы: передаете изменения центральный службе.
3. Агент на сервере: периодически включается по таймеру.
4. Агент на сервере: подключается к центральной службе.
5. Агент на сервере: читает новые сценарии управления конфигурациями.
6. 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.

Принудительная настройка дает важное преимущество – вы контролируете время обновления серверов. Вам не приходится ждать. Сторонники добровольной настройки утверждают, что их подход лучше масштабируется на большое число серверов и удобнее, когда новые серверы могут появиться в любой момент. Однако отложим эту дискуссию на потом, а пока отмечу, что Ansible с успехом использовался для управления тысячами узлов и показал отличные результаты в сетях с динамически добавляемыми и удаляемыми серверами.

Если вам действительно нравится модель, основанная на приемах добровольной настройки, для вас Ansible официально поддерживает особый режим, называемый *ansible-pull*. Я не раскрываю особенностей этого режима в рамках данной книги. Но вы можете узнать больше об этом из официальной документации: http://docs.ansible.com/ansible/playbooks_intro.html#ansible-pull.

Управление небольшим числом серверов

Да, Ansible можно использовать для управления сотнями и даже тысячами узлов. Но управлять единственным узлом с помощью Ansible также очень легко – вам нужно лишь написать один сценарий. Ansible подтверждает принцип Алана Кея: «Простое должно оставаться простым, а сложное – возможным».

Встроенные модули

Ansible можно использовать для выполнения произвольных команд оболочки на удаленных серверах, но его действительно мощной стороной является набор модулей. Модули необходимы для выполнения таких задач, как установка пакетов приложений, перезапуск службы или копирование файлов конфигурации.

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`:

```
user: name=deploy group=web
```

Модули также являются *идемпотентными*[1]. Если пользователь `deploy` не существует, Ansible создаст его. Если он существует, Ansible просто перейдет к следующему шагу. То есть сценарии Ansible можно запускать на сервере много раз. Это большое усовершенствование, по сравнению с подходом на основе сценариев командной оболочки, потому что повторный запуск таких сценариев может привести к незапланированным и хорошо, если безобидным последствиям.

---

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

Как обстоит дело с конвергенцией?

В книгах по управлению конфигурациями часто упоминается идея *конвергенции* (или сходимости), которая нередко ассоциируется с именем Марка Бургесса (Mark Burgess) и его системой управления конфигурациями *CFEngine*. Если система управления конфигурациями конвергентна, 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.

Если вам интересно, что думает автор Ansible об идее конвергенции, прочтите публикацию Майкла ДеХаана «Идемпотентность, конвергенция и другие причудливые слова, которые мы используем слишком часто» («Idempotence, convergence, and other silly fancy words we use too often») на странице группы Ansible Project: https://bit.ly/1InGh1A.

Очень тонкий слой абстракции

Некоторые системы управления конфигурациями предоставляют уровень абстракции настолько мощный, что позволяют использовать одни и те же сценарии для управления серверами с разными операционными системами. Например, вместо конкретных диспетчеров пакетов, таких как yum или apt, можно использовать абстракцию «пакет», поддерживаемую системой управления конфигурациями.

Ansible работает не так – для установки пакетов в системы, основанные на диспетчере apt, вы должны использовать диспетчер apt, а в системы, основанные на диспетчере yum, – диспетчер yum.

На практике это упрощает использование Ansible, хотя на первый взгляд может показаться недостатком. Ansible не требует изучения новых наборов абстракций, нивелирующих разницу между операционными системами. Это сокращает объем документации для изучения перед началом написания сценариев.

При желании вы можете писать собственные сценарии Ansible для выполнения определенных действий, в зависимости от операционной системы на удаленном сервере. Но я стараюсь избегать этого, концентрируя свое внимание на написании сценариев для конкретных операционных систем, таких как Ubuntu.

Модуль является основной единицей повторного использования в сообществе Ansible. Поскольку область применения модуля ограничена и зависит от определенной операционной системы, это позволяет писать качественные и надежно работающие модули. Проект Ansible всегда открыт для новых модулей, предлагаемых сообществом. Я это знаю, поскольку сам предложил несколько.

Сценарии Ansible не предназначены для использования в разных контекстах. В главе 7 мы обсудим *роли* как средство организации сценариев для повторного использования. Также мы обсудим Ansible Galaxy – онлайн-репозиторий ролей.

Однако на практике каждая организация сервера настраивается с некоторыми отличиями, поэтому лучше постараться написать сценарии для своей компании, чем пытаться использовать универсальные. Единственный повод для изучения чужих сценариев – это, например, взглянуть, как и что было сделано.

Связь между Ansible и Ansible, Inc.

Название *Ansible* относится как к программному обеспечению, так и к компании, ведущей проект. 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-го Red Hat купила Ansible Inc.

Не слишком ли проста система Ansible?

В период работы над книгой мой редактор сказал мне, что «некоторые специалисты, использующие систему управления конфигурациями XYZ, называют Ansible «циклом for по сценариям». 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 широкую функциональность, 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.

А как насчет масштабируемости SSH? В главе 12 будет показано, что Ansible применяет SSH-мультиплексирование для оптимизации производительности. Некоторые специалисты используют Ansible для управления тысячами узлов[1].

[1] Например, ознакомьтесь с материалом «Использование Ansible для управления масштабируемым публичным облаком» («Using Ansible at Scale to Manage a Public Cloud») от Jesse Keating, бывшего сотрудника Rackspace.

Я не настолько хорошо знаком с остальными системами, чтобы рассматривать их различия в деталях. Если вам необходим детальный сравнительный анализ систем управления конфигурациями, прочитайте книгу «Taste Test: Puppet, Chef, Salt, Ansible» Мэтта Джейнса (Matt Jaynes). Так случилось, что Мэтт предпочел Ansible.

Что я должен знать?

Для эффективной работы с Ansible необходимо знать основы администрирования операционной системы Linux. Ansible позволяет автоматизировать процессы, но не выполняет волшебным образом тех из них, с которыми вы не справляетесь.

Предполагаю, что 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 среды;
- писать сценарии (на любом языке).

Если все это вам известно, 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.

Ansible использует формат файлов YAML и язык шаблонов Jinja2. Следовательно, вам необходимо изучить их, но обе технологии просты в освоении.

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 200, и они достаточно хорошо представлены в официальной документации.

Книга охватывает только основные возможности механизма шаблонов Jinja2, поскольку их вполне достаточно для работы с Ansible. Для более глубокого изучения Jinja2 я рекомендую обратиться к официальной документации по Jinja2 на странице http://jinja.pocoo.org/docs/dev/.

Книга не дает детального описания функций Ansible, используемых в основном для работы в ранних версиях Linux. Сюда относятся клиент SSH *Paramiko* и *ускоренный режим*.

Наконец, я не рассматриваю некоторых функций Ansible I, чтобы сохранить размер книги в разумных пределах. К ним относятся: режим обновления конфигурации по инициативе клиентов, журналирование, соединение с хостами по протоколам, отличным от SSH, и запрос у пользователя паролей и другой информации.

Установка Ansible

На сегодняшний день все основные дистрибутивы Linux включают пакет Ansible. Поэтому, если вы работаете в Linux, вы сможете установить его, используя «родной» диспетчер пакетов. Но имейте в виду, что это может быть не самая последняя версия Ansible. Если вы работаете в Mac OS X, я рекомендую использовать замечательный диспетчер пакетов Homebrew.

Если такого пакета в вашей версии ОС нет, вы можете установить Ansible с помощью *pip*, диспетчера пакетов Python, выполнив следующую команду:

```
$ sudo pip install ansible
```

При желании Ansible можно установить в локальное *виртуальное окружение* Python (virtualenv). Если вы незнакомы с виртуальными окружениями, можете использовать более новый инструмент под названием *pipsi*. Он автоматически создаст новое виртуальное окружение и установит в него Ansible:

```
$ wget https://raw.githubusercontent.com/mitsuhiko/pipsi/master/get-pipsi.py
$ python get-pipsi.py
$ pipsi install ansible
```

Если вы решите воспользоваться pipsi, добавьте путь *~/.local/bin* в переменную окружения PATH. Некоторые плагины и модули Ansible могут потребовать установки дополнительных библиотек Python. Если вы произвели установку с помощью pipsi и хотели бы установить *docker-py* (необходимый для модулей из библиотеки Ansible Docker) и *boto* (необходимый для модулей из библиотеки Ansible EC2), выполните следующие команды:

```
$ cd ~/.local/venvs/ansible
$ source bin/activate
$ pip install docker-py boto
```

Если вам интересно испытать в работе новейшую версию Ansible, загрузите ее из GitHub:

```
$ git clone https://github.com/ansible/ansible.git –recursive
```

При работе с этой версией вам каждый раз будет нужно выполнять следующие команды, чтобы установить переменные окружения, включая переменную PATH, чтобы оболочка смогла находить программы *ansible* и *ansible-playbooks*.

```
$ cd ./ansible
$ source ./hacking/env-setup
```

Дополнительную информацию об установке можно найти на следующих ресурсах:

- официальная документация по установке Ansible (http://docs.ansible.com/ansible/intro_installation.html);
- pip (http://pip.readthedocs.io/en/stable/);
- virtualenv (http://docs.python-guide.org/en/latest/dev/virtualenvs/);
- pipsi (https://github.com/mitsuhiko/pipsi).

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 службах облачных услуг, таких как Amazon EC2, Google Compute Engine, Microsoft Azure[1], Digital Ocean, Linode.., в общем, вы поняли.

Использование Vagrant для подготовки сервера

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

В Vagrant имеется встроенная возможность подготовки виртуальных машин с Ansible. Подробнее об этом будет рассказано в главе 3. А пока будем считать виртуальную машину под управлением Vagrant обычным сервером Linux.

Vagrant требует установки VirtualBox. Скачайте VirtualBox (https://www.virtualbox.org/), а затем Vagrant (https://www.vagrantup.com/).

Рекомендую создать отдельный каталог для сценариев Ansible и прочих файлов. В следующем примере я создал такой каталог с именем *playbooks*.

Выполните следующие команды, чтобы создать файл конфигурации Vagrant (Vagrantfile) для 64-битного образа виртуальной машины[2] с Ubuntu 14.04 (Trusty Tahr) и загрузить ее.

```
$ mkdir playbooks
$ cd playbooks
$ vagrant init ubuntu/trusty64
$ 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:

---

[1] Да, Azure поддерживает серверы Linux.

[2] Виртуальная машина в терминологии Vagrant называется *machine*, а ее образ – *box*.

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/trusty64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/trusty64' is up to date...
==> default: Setting the name of the VM: playbooks_default_1474348723697_56934
==> default: Clearing any previously set forwarded ports...
==> 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: 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: Warning: Remote connection disconnect. Retrying...
    default: Warning: Remote connection disconnect. Retrying...
    default:
    default: Vagrant insecure key detected. Vagrant will automatically replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
    default: The guest additions on this VM do not match the installed version
    default: of VirtualBox! In most cases this is fine, but in rare cases it can
    default: prevent things such as shared folders from working properly. If you
    default: see shared folder errors, please make sure the guest additions
    default: within the virtual machine match the version of VirtualBox you have
    default: installed on your host and reload your VM.
    default:
    default: Guest Additions Version: 4.3.36
    default: VirtualBox Version: 5.0
==> default: Mounting shared folders...
    default: /vagrant => /Users/lorin/dev/ansiblebook/ch01/playbooks
```

Теперь можно попробовать зайти по SSH на вашу новую виртуальную машину Ubuntu 14.04, выполнив следующую команду:

```
$ vagrant ssh
```

Если все прошло благополучно, вы увидите экран с приветствием:

```
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-96-generic x86_64)

 * Documentation: https://help.ubuntu.com/

  System information as of Fri Sep 23 05:13:05 UTC 2016

  System load:  0.76             Processes:           80
```

```
  Usage of /:   3.5% of 39.34GB  Users logged in:     0
  Memory usage: 25%              IP address for eth0: 10.0.2.15
  Swap usage:   0%

  Graph this data and manage this system at:
    https://landscape.canonical.com/

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

New release '16.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.
```

Введите **exit**, чтобы завершить сеанс SSH.

Этот подход позволяет взаимодействовать с командной оболочкой. Однако Ansible требует подключения к виртуальной машине посредством SSH-клиента, а не команды `vagrant ssh`.

Попросите Vagrant вывести на экран детали SSH-подключения:

```
$ vagrant ssh-config
```

Я у себя получил такой результат:

```
Host default
  HostName 127.0.0.1
  User vagrant
  Port 2222
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /Users/lorin/dev/ansiblebook/ch01/playbooks/.vagrant/
  machines/default/virtualbox/private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

Вот самые важные строки:

```
HostName 127.0.0.1
User vagrant
Port 2222
IdentityFile /Users/lorin/dev/ansiblebook/ch01/playbooks/.vagrant/
machines/default/virtualbox/private_key
```

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

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

Убедитесь, что сможете начать новый SSH-сеанс из командной строки, используя эту информацию. В моем случае команда выглядит так:

```
$ ssh vagrant@127.0.0.1 -p 2222 -i /Users/lorin/dev/ansiblebook/ch01/
playbooks/.vagrant/machines/default/virtualbox/private_key
```

Вы должны увидеть экран входа 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.

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 *playbooks* файл с именем *hosts*. Он будет служить реестром. Если в качестве тестового сервера вы используете виртуальную машину Vagrant, файл *hosts* должен выглядеть, как в примере 1.1. Я разбил содержимое файла на несколько строк, чтобы уместить его по ширине страницы. В действительности информация в файле представлена одной строкой без обратных косых.

Пример 1.1 ❖ Файл playbooks/hosts

```
testserver ansible_host=127.0.0.1 ansible_port=2222 \
  ansible_user=vagrant \
  ansible_private_key_file=.vagrant/machines/default/virtualbox/private_key
```

Здесь можно видеть один из недостатков использования Vagrant: мы вынуждены явно передать дополнительные аргументы, чтобы сообщить Ansible параметры подключения. В большинстве случаев в этих дополнительных данных нет необходимости.

Далее в этой главе вы увидите, как использовать файл *ansible.cfg*, чтобы избежать нагромождения информации в файле реестра. В последующих главах вы увидите, как с той же целью можно использовать переменные Ansible.

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

```
testserver ansible_host=ec2-203-0-113-120.compute-1.amazonaws.com \
  ansible_user=ubuntu ansible_private_key_file=/path/to/keyfile.pem
```

Ansible поддерживает программу ssh-agent, поэтому нет необходимости явно указывать файлы SSH-ключей в реестре. Если прежде вам не доводилось пользоваться этой программой, более детальную информацию о ней вы найдете в разделе «Агент SSH» в приложении A.

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

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

```
$ ansible testserver -i hosts -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.
RSA key fingerprint is e8:0d:7d:ef:57:07:81:98:40:31:19:53:a8:d0:76:21.
Are you sure you want to continue connecting (yes/no)?
```

Просто введите **yes**.

В случае успеха появится следующий результат:

```
testserver | success >> {
  "changed": false,
  "ping": "pong"
}
```

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

```
$ ansible testserver -i hosts -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.

Упрощение задачи с помощью файла ansible.cfg

Нам пришлось ввести много текста в файл реестра, чтобы сообщить системе Ansible информацию о тестовом сервере. 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* – для определения некоторых настроек 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:

1. Файл, указанный в переменной окружения `ANSIBLE_CONFIG`.
2. *./ansible.cfg* (*ansible.cfg* в текущем каталоге).

3. *~/.ansible.cfg* (*.ansible.cfg* в вашем домашнем каталоге).
4. */etc/ansible/ansible.cfg*.

Я обычно храню *ansible.cfg* в текущем каталоге, вместе со сценариями. Это позволяет хранить его в том же репозитории, где хранятся мои сценарии.

Пример 1.2 показывает, как в файле *ansible.cfg* определяются местоположение файла реестра (`inventory`), имя пользователя SSH (`remote_user`) и приватный ключ SSH (`private_key_file`). Эти настройки предполагают использование Vagrant. При использовании отдельного сервера необходимо установить только значения `remote_user` и `private_key_file`.

В нашем примере конфигурации проверка SSH-ключей хоста отключена. Это удобно при работе с Vagrant. В противном случае необходимо вносить изменения в файл *~/.ssh/known_hosts* каждый раз, когда удаляется имеющийся или создается новый Vagrant-сервер. Однако отключение проверки ключей для серверов в сети несет определенные риски. Если вы незнакомы с аутентификацией при помощи ключей хоста, то можете прочитать об этом в приложении А.

Пример 1.2 ❖ ansible.cfg

```
[defaults]
inventory = hosts
remote_user = vagrant
private_key_file = .vagrant/machines/default/virtualbox/private_key
host_key_checking = False
```

Ansible и система управления версиями

Ansible по умолчанию хранит реестр в файле */etc/ansible/hosts*. Однако лично я предпочитаю хранить его вместе с моими сценариями в системе управления версиями.

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

С настройками по умолчанию отпадает необходимость указывать имя пользователя или файл с ключами SSH в файле *hosts*. Запись упрощается до:

```
testserver ansible_ host=127.0.0.1 ansible_ port=2222
```

Мы также можем запустить Ansible без аргумента `-i hostname`:

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

Мне 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 | success | rc=0 >>
 17:14:07 up 1:16,   1 user,  load average: 0.16, 0.05, 0.04
```

Модуль `command` настолько часто используется, что сделан модулем по умолчанию, то есть его имя можно опустить в команде:

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

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

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

Вывод, возвращаемый машиной Vagrant, выглядит следующим образом:

```
testserver | success | rc=0 >>
[    5.170544] type=1400 audit(1409500641.335:9): apparmor="STATUS" operation=
"profile_replace" profile="unconfined" name="/usr/lib/NetworkManager/nm-dhcp-c
lient.act on" pid=888 comm="apparmor_parser"
[   5.170547] type=1400 audit(1409500641.335:10): apparmor="STATUS" operation=
"profile_replace" profile="unconfined" name="/usr/lib/connman/scripts/dhclientscript"
pid=888 comm="apparmor_parser"
[   5.222366] vboxvideo: Unknown symbol drm_open (err 0)
[   5.222370] vboxvideo: Unknown symbol drm_poll (err 0)
[   5.222372] vboxvideo: Unknown symbol drm_pci_init (err 0)
[   5.222375] vboxvideo: Unknown symbol drm_ioctl (err 0)
[   5.222376] vboxvideo: Unknown symbol drm_vblank_init (err 0)
[   5.222378] vboxvideo: Unknown symbol drm_mmap (err 0)
[   5.222380] vboxvideo: Unknown symbol drm_pci_exit (err 0)
[   5.222381] vboxvideo: Unknown symbol drm_release (err 0)
```

Чтобы выполнить команду с привилегиями root, нужно передать параметр `-b`. В этом случае Ansible выполнит команду от лица пользователя root. Например, для доступа к */var/log/syslog* требуются привилегии root:

```
$ ansible testserver -b -a "tail /var/log/syslog"
```

Результат будет выглядеть примерно так:

```
testserver | success | rc=0 >>
Aug 31 15:57:49 vagrant-ubuntu-trusty-64 ntpdate[1465]: /
adjust time server 91.189
94.4 offset -0.003191 sec
Aug 31 16:17:01 vagrant-ubuntu-trusty-64 CRON[1480]: (root) CMD ( cd /
&& run-p
rts --report /etc/cron.hourly)
Aug 31 17:04:18 vagrant-ubuntu-trusty-64 ansible-ping: Invoked with data=None
```

```
Aug 31 17:12:33 vagrant-ubuntu-trusty-64 ansible-ping: Invoked with data=None
Aug 31 17:14:07 vagrant-ubuntu-trusty-64 ansible-command: Invoked with executable
None shell=False args=uptime removes=None creates=None chdir=None
Aug 31 17:16:01 vagrant-ubuntu-trusty-64 ansible-command: Invoked with executable
None shell=False args=tail /var/log/messages removes=None creates=None chdir=None
Aug 31 17:17:01 vagrant-ubuntu-trusty-64 CRON[2091]: (root) CMD ( cd /
&& run-pa
rts --report /etc/cron.hourly)
Aug 31 17:17:09 vagrant-ubuntu-trusty-64 ansible-command: Invoked with /
executable=
N one shell=False args=tail /var/log/dmesg removes=None creates=None chdir=None
Aug 31 17:19:01 vagrant-ubuntu-trusty-64 ansible-command: Invoked with /
executable=
None shell=False args=tail /var/log/messages removes=None creates=None chdir=None
Aug 31 17:22:32 vagrant-ubuntu-trusty-64 ansible-command: Invoked with /
executable=
one shell=False args=tail /var/log/syslog removes=None creates=None chdir=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 apt -a name=nginx
```

Если установить Nginx не удалось, 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`.

Что дальше

Вспомним, о чем рассказывалось в этой главе. Здесь мы рассмотрели основные понятия системы Ansible, включая взаимодействия с удаленными серверами, и отличия от других систем управления конфигурациями. Мы также увидели, как пользоваться утилитой командной строки `ansible` для выполнения простых задач на единственном хосте.

Однако использование `ansible` для выполнения команд на одном хосте не особенно интересно. В следующей главе мы рассмотрим действительно полезные сценарии.

Глава 2

Сценарии: начало

Работая с Ansible, большую часть времени вы будете уделять написанию сценариев. *Сценарием* в Ansible называется файл, описывающий порядок управления конфигурациями. Рассмотрим, например, установку веб-сервера Nginx и его настройку для поддержки защищённых соединений.

К концу этой главы у вас должны появиться следующие файлы:

- *playbooks/ansible.cfg*;
- *playbooks/hosts*;
- *playbooks/Vagrantfile*;
- *playbooks/web-notls.yml*;
- *playbooks/web-tls.yml*;
- *playbooks/files/nginx.key*;
- *playbooks/files/nginx.crt*;
- *playbooks/files/nginx.conf*;
- *playbooks/templates/index.html.j2*;
- *playbooks/templates/nginx.conf.j2*.

Подготовка

Прежде чем запустить сценарий на машине Vagrant, необходимо открыть порты 80 и 443. Как показано на рис. 2.1, мы настроим Vagrant так, чтобы запросы к портам 8080 и 8443 на локальной машине перенаправлялись портам 80 и 443 на машине Vagrant. Это позволит получить доступ к веб-серверу, запущенному на Vagrant, по адресам http://localhost:8080 и https://localhost:8443.

Измените содержимое *Vagrantfile*, как показано ниже:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.network "forwarded_port", guest: 443, host: 8443
end
```

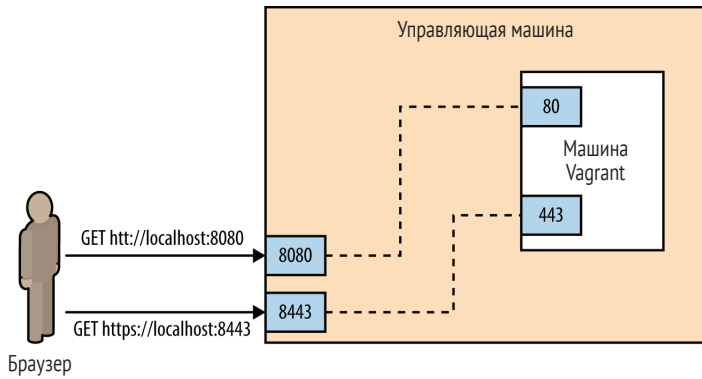

Рис. 2.1 ❖ Открытие портов на машине Vagrant

Эти настройки отобразят порты 8080 и 8443 локальной машины в порты 80 и 443 машины Vagrant. После сохранения изменений дайте команду применить их:

```
$ vagrant reload
```

В результате на экране должны появиться следующие строки:

```
==> default: Forwarding ports...
    default: 80 => 8080 (adapter 1)
    default: 443 => 8443 (adapter 1)
    default: 22 => 2222 (adapter 1)
```

Очень простой сценарий

В нашем первом примере сценария мы настроим хост для запуска веб-сервера Nginx. В этом примере мы не будем настраивать поддержку TLS-шифрования веб-сервером. Это сделает установку проще. Однако правильный веб-сайт должен поддерживать TLS-шифрование, и мы увидим, как это сделать, далее в этой главе.

TLS и SSL

Возможно, вам более знакома аббревиатура *SSL*, чем *TLS*. SSL – это более старый протокол, используемый для обеспечения безопасности взаимодействий браузеров и веб-серверов. Но теперь он постепенно вытесняется более новым протоколом TLS. Несмотря на то что многие продолжают использовать аббревиатуру *SSL*, подразумевая более новый протокол, в этой книге я буду использовать точное название: *TLS*.

Сначала посмотрим, что получится, если запустить сценарий из примера 2.1, а затем детально изучим его содержимое.

Пример 2.1 ❖ web-notls.yml

```
- name: Configure webserver with nginx
  hosts: webservers
  become: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/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=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
        mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

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

Внимательный читатель заметит, что в примере 2.1 в одном случае используется `True` (для запуска `sudo`) и `yes` в другом случае (для обновления кэша 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 `sudo: 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, y, Y
```

Ложь в YAML

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

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
```

Я склонен следовать примерам из официальной документации Ansible, где обычно для передачи в аргументах модулей используются `yes` и `no` (что соответствует документации по модулям) и `True` и `False` во всех других случаях.

Файл конфигурации Nginx

Данному сценарию необходимы два дополнительных файла. Сначала создадим файл конфигурации Nginx.

Nginx поставляется с файлом конфигурации, готовым к использованию только для обслуживания статичных файлов. Но чаще его необходимо дорабатывать под свои нужды. Поэтому мы изменим файл конфигурации по умолчанию в рамках данного примера. Как станет понятно позже, мы должны добавить в файл конфигурации поддержку TLS. В примере 2.2 приводится стандартный файл конфигурации Nginx. Сохраните его с именем *playbooks/files/nginx.conf*[1].

В соответствии с соглашениями, принятыми в Ansible, файлы должны сохраняться в подкаталоге *files*, а шаблоны Jinja2 – в подкаталоге *templates*. Я буду придерживаться этого соглашения на протяжении всей книги.

Пример 2.2 ❖ files/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;
    }
}
```

Создание начальной страницы

Добавим свою начальную страницу. Используем шаблоны Ansible, чтобы сгенерировать файл. Сохраните файл из примера 2.3 в *playbooks/templates/index.html.j2*.

Пример 2.3 ❖ 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>{{ ansible_managed }}</p>
  </body>
</html>
```

[1] Обратите внимание, что если сохранить его с именем *nginx.conf*, он заменит файл *sites-enabled/default*, а не основной файл конфигурации */etc/nginx.conf*.

В этом шаблоне используется специальная переменная Ansible `ansible_managed`. Обрабатывая шаблон, Ansible заменит ее информацией о времени создания файла шаблона. На рис. 2.2 показан скриншот веб-браузера с созданной HTML-страницей.

Рис. 2.2 ❖ Вид получившейся начальной страницы

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 позднее. Откройте файл *playbooks/hosts* в редакторе и добавьте строку `[webservers]` над строкой `testserver`, как показано в примере 2.4. Это означает, что `testserver` включен в группу `webservers`.

Пример 2.4 ❖ playbooks/hosts

```
[webservers]
testserver ansible_host=127.0.0.1 ansible_port=2222
```

Теперь можно попробовать выполнить команду `ping` для группы `webservers` с помощью утилиты `ansible`:

```
$ ansible webservers -m ping
```

Результат должен выглядеть так:

```
testserver | success >> {
    "changed": false,
    "ping": "pong"
}
```

Запуск сценария

Сценарии выполняются командой `ansible-playbook`, например:

```
$ ansible-playbook web-notls.yml
```

В примере 2.5 показано, как должен выглядеть результат.

Пример 2.5 ❖ Результат запуска сценария командой ansible-playbook

```
PLAY [Configure webserver with nginx] ********************************

GATHERING FACTS ************************************************************
ok: [testserver]

TASK: [install nginx] ******************************************************
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
```

Программа 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 коров, можете отключить вызов *cowsay*, установив переменную окружения ANSIBLE_NOCOWS:

```
$ export ANSIBLE_NOCOWS=1
```

Отключить *cowsay* можно также, добавив в файл *ansible.cfg* строки:

```
[defaults]
nocows = 1
```

Если вы не получили никаких ошибок[1], у вас должно получиться открыть в браузере страницу *http://localhost:8080*. В результате вы должны увидеть начальную страницу, как показано на рис. 2.2.

[1] Если вы столкнулись с ошибкой, обратитесь к главе 14, где описывается, как ее устранить.

Если файл сценария отмечен как выполняемый и начинается с такой строки[1]:

```
#!/usr/bin/env ansible-playbook
```

в подобном случае вы сможете запустить его непосредственно:

```
$ ./web-notls.yml
```

Сценарии пишутся на YAML

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

Начало файла

Файлы YAML начинаются с трех дефисов, обозначающих начало документа:

```
---
```

Однако Ansible не посчитает ошибкой, если вы забудете указать три дефиса в начале сценария.

Комментарии

Комментарии начинаются со знака «решетка» и продолжаются до конца строки, как в сценариях на языке командной оболочки, Python и Ruby:

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

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. Обычно это строки с фигурными скобками {{ и }}, которые используются для подстановки значений переменных. Но об этом чуть позже.

Булевы выражения

В YAML есть собственный логический тип. Он предлагает широкий выбор строк, которые могут интерпретироваться как «истина» и «ложь». Этот вопрос мы рассмотрели в заметке «Почему в одном случае используется “True”,

[1] Известной также как *shebang*.

а в другом “Yes”?» выше. Я лично всегда использую константы `True` и `False` в своих сценариях.

Например, вот булево выражение на YAML:

```
True
```

Аналог в JSON выглядит так:

```
true
```

Списки

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

Списки оформляются с помощью дефиса:

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

Аналог в JSON:

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

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. Он выглядит так:

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

Словари

Словари в YAML подобны объектам в JSON, словарям в Python или хэш-массивам в Ruby. Технически в YAML они называются *отображениями*, но я называю их *словарями*, чтобы избежать противоречий с официальной документацией Ansible.

Они выглядят так:

```
address: 742 Evergreen Terrace
city: Springfield
state: North Takoma
```

Аналог в JSON:

```
{
  "address": "742 Evergreen Terrace",
  "city": "Springfield",
  "state": "North Takoma"
}
```

YAML также поддерживает формат встроенных словарей:

```
{address: 742 Evergreen Terrace, city: Springfield, state: North Takoma}
```

Объединение строк

Во время написания сценариев часто возникают ситуации, когда необходимо передать модулю много аргументов. В эстетических целях их можно поместить в несколько строк в файле. Однако при этом необходимо, чтобы Ansible воспринимал их как единую строку.

В YAML для этого можно воспользоваться знаком «больше» (>). Парсер YAML в этом случае заменит разрывы строк пробелами. Например:

```
address: >
    Department of Computer Science,
    A.V. Williams Building,
    University of Maryland
city: College Park
state: Maryland
```

Аналог в JSON:

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

Структура сценария

Рассмотрим наш сценарий с точки зрения YAML. В примере 2.6 он приводится снова:

Пример 2.6 ❖ web-notls.yml

```
- name: Configure webserver with nginx
  hosts: webservers
  become: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/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=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
```

```
        mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

В примере 2.7 приводится аналог этого файла в формате JSON.

Пример 2.7 ❖ Аналог web-notls.yml в формате JSON

```
[
  {
    "name": "Configure webserver with nginx",
    "hosts": "webservers",
    "become": true,
    "tasks": [
      {
        "name": "Install nginx",
        "apt": "name=nginx update_cache=yes"
      },
      {
        "name": "copy nginx config file",
        "template": "src=files/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=templates/index.html.j2 dest=/usr/share/nginx/html/
index.html mode=0644"
      },
      {
        "name": "restart nginx",
        "service": "name=nginx state=restarted"
      }
    ]
  }
]
```

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.

Операции

В любом формате – YAML или JSON – сценарий является списком словарей, или списком *операций*.

Вот как выглядит операция из нашего примера[1]:

```
- name: Configure webserver with nginx
  hosts: webservers
  become: True
  tasks:
    - name: install nginx
      apt: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/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=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
        mode=0644

    - name: restart nginx
      service: name=nginx state=restarted
```

Каждая операция должна содержать:

- список настраиваемых *хостов*;
- список *задач*, выполняемых на этих хостах.

Воспринимайте операцию как нечто, связывающее хосты и задачи.

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.

`become`
: Если имеет значение «истина», Ansible выполнит каждую задачу, предварительно приобретя привилегии пользователя root (по умолчанию). Это может пригодиться для управления серверами Ubuntu, поскольку по умолчанию эта система не позволяет устанавливать SSH-соединение с привилегиями root.

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

---

[1] На самом деле это список, содержащий одну операцию.

Задачи

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

```
- name: install nginx
  apt: name=nginx update_cache=yes
```

Поскольку параметр `name` не является обязательным, задачу можно записать так:

```
- apt: name=nginx update_cache=yes
```

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 задачи. В этом случае необходимо сослаться на задачу по имени.

Каждая задача должна содержать ключ с названием модуля и его аргументами. В данном примере модуль называется `apt` и принимает аргументы `name=nginx update_cache=yes`.

Эти аргументы сообщают модулю `apt` установить пакет *nginx* и обновить кэш пакетов (аналог команды `apt-get update`) перед установкой.

Важно понять, что с точки зрения парсера YAML, используемого Ansible, аргументы воспринимаются как строки, а не словари. То есть, чтобы разбить аргументы на несколько строк, необходимо использовать правило объединения строк YAML:

```
- name: install nginx
  apt: >
      name=nginx
      update_cache=yes
```

Ansible поддерживает также синтаксис, позволяющий определять аргументы модулей как словари YAML. Это может пригодиться при работе с модулями, имеющими составные аргументы. Мы рассмотрим этот вопрос в заметке «Короткое отступление: составные аргументы задач» в главе 6.

Ansible поддерживает также старый синтаксис, использующий ключ `action` и записывающий имя модуля в значение. Например, предыдущий пример можно записать так:

```
- name: install nginx
  action: apt name=nginx update_cache=yes
```

Модули

Модули – это сценарии[1], которые поставляются с Ansible и производят определенное действие на хосте. Правда, надо признать, что это довольно общее описание, но среди модулей Ansible встречается множество вариантов. В этой главе используются следующие модули:

`apt`
: Устанавливает или удаляет пакеты с использованием диспетчера пакетов apt.

`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.

Чтение документации по модулям Ansible

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

```
$ ansible-doc service
```

Для пользователей Mac OS X существует прекрасное средство просмотра документации Dash (https://kapeli.com/dash), обладающее поддержкой Ansible. Dash индексирует всю документацию по модулям Ansible. Это коммерческая программа (на момент написания книги ее стоимость составляла $24.99), но, по моему мнению, она бесценна.

Как рассказывалось в первой главе, Ansible выполняет задачу на хосте, генерируя сценарий, исходя из имени модуля и его аргументов, а затем копирует его на хост и запускает.

В состав Ansible входит более 200 модулей, и их число растет с каждой новой версией. Также можно найти модули, написанные сторонними разработчиками, или написать свои собственные.

---

[1] Модули, поставляемые с Ansible, написаны на Python. Но, в принципе, они могут быть написаны на любом языке.

Резюме

Итак: сценарий содержит одну или несколько операций. Операции связываются с неупорядоченным множеством хостов и упорядоченным списком задач. Каждая задача соответствует только одному модулю.

Диаграмма на рис. 2.3 изображает взаимосвязи между сценариями, операциями, хостами, задачами и модулями.

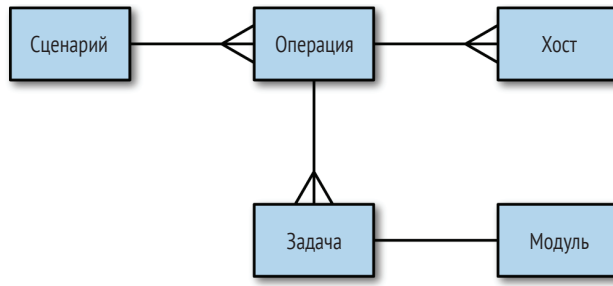

Рис. 2.3 ❖ Диаграмма взаимосвязей

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 2.5 и обратите внимание, что состояние некоторых задач указано как `changed` (изменено), а других – `ok`. Например, задача `install nginx` имеет статус `changed`. На моем терминале он выделен желтым.

```
TASK: [install nginx] *****************************************************
changed: [testserver]
```

С другой стороны, задача `enable configuration` имеет статус `ok`, на моем терминале выделенный 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, Ansible не предпринимает никаких действий и сообщает, что статус `ok`.

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

Как показано в примере выше, задача `install nginx` 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 я копировал. Причина в том, что файл *nginx.conf*, который я использовал в своем сценарии, идентичен файлу *nginx.conf*, который устанавливается из пакета *nginx* в Ubuntu.

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 TLS веб-сервером. Для этого нам понадобятся следующие новые элементы:

- переменные;
- обработчики.

В примере 2.8 приводится наш сценарий с включенной настройкой поддержки TLS.

Пример 2.8 ❖ web-tls.yml

```
- name: Configure webserver with nginx and tls
  hosts: webservers
  become: True
  vars:
    key_file: /etc/nginx/ssl/nginx.key
    cert_file: /etc/nginx/ssl/nginx.crt
    conf_file: /etc/nginx/sites-available/default
    server_name: localhost
  tasks:
    - name: Install nginx
      apt: name=nginx update_cache=yes cache_valid_time=3600

    - name: create directories for ssl certificates
      file: path=/etc/nginx/ssl state=directory

    - name: copy TLS key
      copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
      notify: restart nginx

    - name: copy TLS certificate
      copy: src=files/nginx.crt dest={{ cert_file }}
      notify: restart nginx

    - name: copy nginx config file
      template: src=templates/nginx.conf.j2 dest={{ conf_file }}
      notify: restart nginx

    - name: enable configuration
```

```
    file: dest=/etc/nginx/sites-enabled/default src={{ conf_file }} state=link
    notify: restart nginx

  - name: copy index.html
    template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html
    mode=0644

handlers:
  - name: restart nginx
    service: name=nginx state=restarted
```

Создание сертификата TLS

Мы должны вручную создать сертификат TLS. Для промышленной эксплуатации сертификат TLS необходимо приобрести в центре сертификации или использовать бесплатную службу, такую как Let's Encrypt, которая поддерживается в Ansible посредством модуля `letsencrypt`. Мы используем «самоподписанный» (self-signed) сертификат, поскольку его можно создать бесплатно.

Создайте подкаталог *files* в каталоге *playbooks*, а затем сертификат TLS и ключ:

```
$ mkdir files
$ openssl req -x509 -nodes -days 3650 -newkey rsa:2048 \
    -subj /CN=localhost \
    -keyout files/nginx.key -out files/nginx.crt
```

Эта пара команд создаст файлы *nginx.key* и *nginx.crt* в каталоге *files*. Срок действия сертификата ограничен 10 годами (3650 дней) со дня его создания.

Переменные

Теперь операция в нашем сценарии включает раздел `vars`:

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

Этот раздел определяет 4 переменные и их значения.

В нашем примере каждое значение – это строка (например, `/etc/nginx/ssl/nginx.key`), 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 {{ и }}. Ansible заменит скобки значением переменной.

Предположим, что в сценарии имеется следующая задача:

```
- name: copy TLS key
  copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
```

При выполнении задачи Ansible заменит {{ key_file }} на /etc/nginx/ssl/nginx.key.

Когда использовать кавычки

Если ссылка на переменную следует сразу после имени модуля, 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 синтаксический анализатор YAML. Чтобы избежать этого, необходимо заключить в кавычки все выражение аргумента.
К сожалению, простое заключение аргумента в кавычки целиком также не решит проблему.

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

Это удовлетворит синтаксический анализатор YAML, но результат будет отличаться от ожидаемого:

```
TASK: [show a debug message] ***************************************************
ok: [localhost] => {
    "msg": "The"
}
```

Аргумент msg модуля debug требует заключения строки в кавычки для сохранения пробелов. В данном конкретном случае необходимо заключить в кавычки не только аргумент целиком, но и сам аргумент msg. Ansible распознает одинарные и двойные кавычки, т. е. можно поступить так:

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

Это даст ожидаемый результат:

```
TASK: [show a debug message] ***************************************************
ok: [localhost] => {
    "msg": "The debug module will print a message: neat, eh?"
}
```

Ansible сгенерирует вполне информативные сообщения об ошибках, если вы забудете расставить кавычки и у вас получится недопустимый код YAML.

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, в котором 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 2.9.

Пример 2.9 ❖ Шаблон электронного письма

```
Dear {{ name }},

You have {{ num_comments }} new comments on your blog: {{ blog_name }}.
```

В случае с Ansible это не HTML-страницы или электронные письма, а файлы конфигурации. Если можно избежать редактирования файлов конфигурации вручную, 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 Ansible использует механизм Jinja2. Если вы когда-либо пользовались библиотеками шаблонов, такими как Mustache, ERB или Django, тогда 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 2.10.

Пример 2.10 ❖ templates/nginx.conf.j2

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

    listen 443 ssl;

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

    server_name {{ server_name }};
    ssl_certificate {{ cert_file }};
    ssl_certificate_key {{ 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 три переменные:

- `server_name` – название хоста веб-сервера (например, `www.example.com`);
- `cert_file` – путь к файлу сертификата TLS;
- `key_file` – путь к файлу приватного ключа TLS.

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.

Ранние версии Ansible использовали знак доллара ($) вместо фигурных скобок для обозначения переменных в сценариях. Прежде, чтобы разыменовать переменную *foo*, на нее нужно было сослаться как `$foo`, в то время как сейчас используется форма `{{ foo }}`. Знак доллара прекратили использовать. И если вы встретите его в сценарии, найденном в Интернете, знайте, что перед вами код, созданный в ранней версии Ansible.

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 (http://jinja.pocoo.org/docs/dev/templates/). Впрочем, вам едва ли потребуются все продвинутые возможности. Но вы почти наверняка будете пользоваться фильтрами; мы рассмотрим их в последующей главе.

Обработчики

А теперь вернемся к нашему сценарию *web-tls.yml*. Мы не обсудили еще два элемента. Один из них – раздел обработчиков `handlers`:

```
handlers:
- name: restart nginx
  service: name=nginx state=restarted
```

И второй – ключ `notify` в некоторых задачах:

```
- name: copy TLS key
  copy: src=files/nginx.key dest={{ key_file }} owner=root mode=0600
  notify: restart nginx
```

Обработчики – это одна из условных форм, 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[1], если изменится любой из компонентов:

---

[1] Вместо перезапуска службы можно перезагрузить файл конфигурации командой `state=reloaded`.

- 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 выполняются только после завершения всех задач и только один раз, даже если было получено несколько. 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. Лично я использую их исключительно для перезапуска служб. Надо сказать, что это не дает особой выгоды, потому что перезапуск службы всегда можно организовать в конце сценария и обойтись без использования уведомлений.

Другое неудобство обработчиков состоит в том, что они могут создавать сложности при отладке сценария, например:

1. Я запускаю сценарий.
2. Одна из задач с уведомлением изменяет состояние.
3. В следующей задаче возникает ошибка, прерывающая работу Ansible.
4. Я исправляю ошибку в сценарии.
5. Запускаю Ansible снова.
6. Ни одна из задач не сообщает об изменении состояния во второй раз, Ansible не запускает обработчика.

Дополнительную информацию об обработчиках и их применении вы найдете в разделе «Улучшенные обработчики» в главе 9.

Запуск сценария

Запуск сценария выполняется командой `ansible-playbook`.

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

Вывод должен выглядеть примерно так:

```
PLAY [Configure webserver with nginx and tls] *********************************

GATHERING FACTS ***************************************************************
ok: [testserver]

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

TASK: [create directories for tls certificates] *******************************
changed: [testserver]

TASK: [copy TLS key] **********************************************************
```

```
changed: [testserver]

TASK: [copy TLS certificate] ***************************************************
changed: [testserver]

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

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

NOTIFIED: [restart nginx] ******************************************************
changed: [testserver]

PLAY RECAP *********************************************************************
testserver                 : ok=8    changed=6    unreachable=0    failed=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» (см. рис. 2.4).

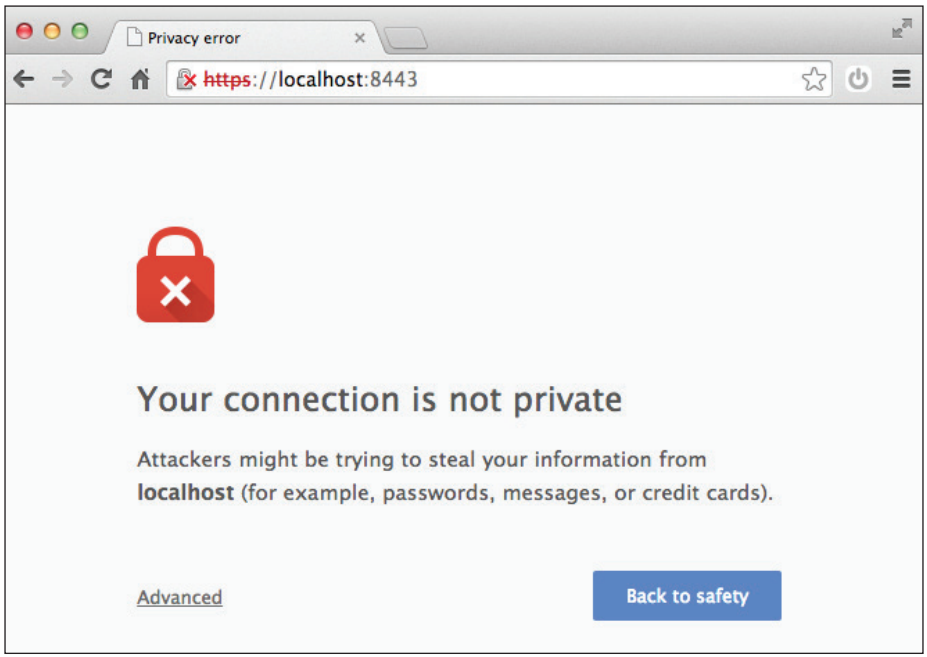

Рис. 2.4 ❖ Некоторые браузеры, такие как Chrome, не доверяют «самоподписанным» сертификатам TLS

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

В этой главе мы изучили многое из того, *что* делает Ansible с хостами. Обработчики – лишь одна из форм контроля, поддерживаемых в Ansible. В последующих главах 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 выполняются сценарии.

Глава 3

Реестр: описание серверов

До настоящего момента мы рассматривали работу лишь с одним сервером (или *хостом*, в терминологии Ansible). В действительности вам предстоит управлять многими хостами. Группа хостов, данными о которых располагает Ansible, называется *реестром*. В этой главе вы узнаете, как составить реестр, описывающий группу хостов.

Файл реестра

Самый простой способ описать имеющиеся хосты – перечислить их в текстовых файлах, называемых *файлами реестра*. Простейший файл реестра содержит самый обычный список имен хостов, как показано в примере 3.1.

Пример 3.1 ❖ Простейший файл реестра

```
ontario.example.com
newhampshire.example.com
maryland.example.com
virginia.example.com
newyork.example.com
quebec.example.com
rhodeisland.example.com
```

По умолчанию Ansible использует локальный SSH-клиент. То есть система поймет любые псевдонимы, которые вы определите в файле конфигурации SSH. Однако это не относится к случаю, когда Ansible настроена на использование плагина Paramiko, а не плагина SSH по умолчанию.

По умолчанию 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.

Даже притом, что Ansible автоматически добавляет `localhost` в реестр, в вашем файле реестра должен иметься хотя бы один другой хост. Иначе выполнение команды `ansible-playbook` завершится с ошибкой:

```
ERROR: provided hosts list is empty
```

Если у вас нет других хостов для включения в файл реестра, просто добавьте в него явную запись с именем `localhost`, например:

```
localhost ansible_connection=local
```

Вводная часть: несколько машин Vagrant

Для обсуждения реестра нам необходимо иметь несколько хостов. Давайте сконфигурируем в Vagrant три хоста и назовем их `vagrant1`, `vagrant2` и `vagrant3`. Прежде чем вносить изменения в существующий файл *Vagrantfile*, не забудьте удалить существующую виртуальную машину, выполнив команду

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

Если запустить эту команду без флага `--force`, Vagrant предложит подтвердить удаление виртуальной машины.

После этого измените файл *Vagrantfile*, как показано в примере 3.2.

Пример 3.2 ❖ Vagrantfile с тремя серверами

```
VAGRANTFILE_API_VERSION = "2"

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

  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/trusty64"
    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/trusty64"
    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 = "ubuntu/trusty64"
    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 разные SSH-ключи для каждого хоста. В примере 3.2 содержится 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 только один SSH-ключ в файле *ansible.cfg*. Нам также необходимо изменить значение `host_key_checking` в файле *ansible.cfg*. Измененный файл должен выглядеть, как показано в примере 3.3.

Пример 3.3 ❖ ansible.cfg

```
[defaults]
hostfile = inventory
remote_user = vagrant
private_key_file = ~/.vagrant.d/insecure_private_key
host_key_checking = False
```

Предполагается, что каждый из этих серверов потенциально может быть веб-сервером, поэтому в примере 3.2 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...
...
    vagrant3: 80 => 8082 (adapter 1)
    vagrant3: 443 => 8445 (adapter 1)
    vagrant3: 22 => 2201 (adapter 1)
==> vagrant3: Booting VM...
==> vagrant3: Waiting for machine to boot. This may take a few minutes...
    vagrant3: SSH address: 127.0.0.1:2201
    vagrant3: SSH username: vagrant
    vagrant3: SSH auth method: private key
    vagrant3: Warning: Connection timeout. Retrying...
==> vagrant3: Machine booted and ready!
==> vagrant3: Checking for guest additions in VM...
==> vagrant3: Mounting shared folders...
    vagrant3: /vagrant => /Users/lorinhochstein/dev/oreilly-ansible/playbooks
```

Теперь создадим файл реестра, включающий все три машины.

Сначала посмотрим, какие порты локальной машины отображены в порт SSH (22) каждой виртуальной машины. Напомню, 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/lorinhochstein/.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/lorinhochstein/.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/lorinhochstein/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

Как видите, для `vagrant1` используется порт 2222, для `vagrant2` – порт 2200 и для `vagrant3` – порт 2201.

Измените файл *hosts*, как показано ниже:

```
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

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

```
$ ansible vagrant2 -a "ip addr show dev eth0"
```

На моей машине я получил такой результат:

```
vagrant2 | success | rc=0 >>
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
    link/ether 08:00:27:fe:1e:4d brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global eth0
    valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fefe:1e4d/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 имя хоста (127.0.0.1) и порт (2222, 2200 или 2201), к которому будет подключаться SSH-клиент системы Ansible.

В Ansible эти переменные называются *поведенческими параметрами*. Некоторые из них можно использовать для изменения значений по умолчанию (табл. 3.1).

Таблица 3.1. 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
`ansible_user`	`root`	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_private_key_file`	(нет)	Приватный 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, Ansible использует локальный SSH-клиент. 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 удаленная оболочка – это оболочка Bourne Shell, доступная как */bin/sh*, и создает соответствующие параметры командной строки, которые используются с оболочкой Bourne Shell.

В этой переменной можно также передать значение `csh`, `fish` или `powershell` (при работе с Windows). Однако я никогда не сталкивался с необходимостью менять тип оболочки.

ansible_python_interpreter

Поскольку модули, входящие в состав Ansible, реализованы на Python 2, чтобы использовать их, Ansible должна знать местоположение интерпретатора Python на удаленной машине. Вам может потребоваться изменить эту переменную, если на удаленной машине путь к выполняемому файлу интерпретатора Python отличается от */usr/bin/python*. Например, для хостов с Arch Linux может понадобиться присвоить этой переменной значение */usr/bin/python2*, потому что путь */usr/bin/python* в Arch Linux соответствует интерпретатору Python 3, а модули Ansible пока не совместимы с Python 3.

ansible_*_interpreter

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

Переопределение поведенческих параметров по умолчанию

Вы можете переопределить некоторые поведенческие параметры по умолчанию в секции `[defaults]` файла *ansible.cfg* (табл. 3.2). Напомню, что мы уже использовали эту возможность для изменения пользователя SSH по умолчанию.

Таблица 3.2. Значения по умолчанию, которые могут быть заменены в ansible.cfg

Поведенческий параметр	Параметр в файле ansible.cfg
`ansible_port`	`remote_port`
`ansible_user`	`remote_user`
`ansible_private_key_file`	`private_key_file`
`ansible_shell_type`	`executable` (см. ниже)

Параметр `executable` в файле *ansible.cfg* – не совсем то же самое, что поведенческий параметр `ansible_shell_type`. Параметр `executable` определяет полный путь к используемой 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 в себя 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"
```

Я у себя получил такой результат:

```
vagrant3 | success | rc=0 >>
Sun Sep 7 02:56:46 UTC 2014

vagrant2 | success | rc=0 >>
Sun Sep 7 03:03:46 UTC 2014

vagrant1 | success | rc=0 >>
Sun Sep 7 02:56:47 UTC 2014
```

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:

```
ontario.example.com
newhampshire.example.com
maryland.example.com
virginia.example.com
newyork.example.com
quebec.example.com
rhodeisland.example.com

[vagrant]
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

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

```
maryland.example.com
newhampshire.example.com
newyork.example.com
ontario.example.com
quebec.example.com
rhodeisland.example.com
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
virginia.example.com

[vagrant]
vagrant1
vagrant2
vagrant3
```

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.

Необходимо развернуть данное приложение в разных окружениях: промышленной (для реального использования), тестовой (для тестирования на хостах, к которым члены нашей команды имеют доступ) и 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.

Представим также, что в окружении для тестирования мы решили использовать меньше хостов, чем в промышленном окружении. Это позволит сократить издержки, поскольку нагрузка на тестовое окружение будет существенно ниже. Допустим, для тестового окружения мы решили использовать 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 3.4 представлен вариант возможного файла реестра, в котором наши серверы сгруппированы по признаку принадлежности к окружению

(промышленному, тестовому, Vagrant) и по функциональности (веб-сервер, диспетчер очереди задач и т. д.).

Пример 3.4 ❖ Файл реестра для развертывания приложения Django

```
[production]
delaware.example.com
georgia.example.com
maryland.example.com
newhampshire.example.com
newjersey.example.com
newyork.example.com
northcarolina.example.com
pennsylvania.example.com
rhodeisland.example.com
virginia.example.com

[staging]
ontario.example.com
quebec.example.com

[vagrant]
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201

[lb]
delaware.example.com

[web]
georgia.example.com
newhampshire.example.com
newjersey.example.com
ontario.example.com
vagrant1

[task]
newyork.example.com
northcarolina.example.com
maryland.example.com
ontario.example.com
vagrant2

[rabbitmq]
pennsylvania.example.com
quebec.example.com
vagrant3

[db]
rhodeisland.example.com
virginia.example.com
quebec.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 еще 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_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

Имена `vagrant1`, `vagrant2`, `vagrant3` – это *псевдонимы*. Они – не настоящие имена серверов, но их удобно использовать для обозначения этих хостов.

Ansible поддерживает синтаксис `<hostname>:<port>` описания хостов. То есть строку с описанием `vagrant1` можно заменить объявлением `127.0.0.1:2222`. Однако нам не удастся задействовать гипотетический реестр, представленный в примере 3.5.

Пример 3.5 ❖ Этот реестр не работает

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

Причина в том, что с IP-адресом *127.0.0.1* можно определить только один хост, поэтому группа `vagrant` содержала бы в этом случае лишь один хост вместо трех.

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 3.4 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*. 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., 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
```

Ansible 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:

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

Переменные хостов и групп: внутренняя сторона реестра

Вспомните, как мы определили поведенческие параметры для хостов Vagrant:

```
vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_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 установить ее значение для каждого сервера:

```
newhampshire.example.com color=red
maryland.example.com color=green
ontario.example.com color=blue
quebec.example.com color=purple
```

Эту переменную затем можно использовать в сценарии, как любую другую.

Лично я редко закрепляю переменные за отдельными хостами, но я часто связываю 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 системы в рабочее состояние нам необходимо настроить:

- в веб-сервере: имя хоста, порт, имя пользователя и пароль основного сервера Postgres, а также имя базы данных;
- в диспетчере очереди: имя хоста, порт, имя пользователя и пароль основного сервера Postgres, а также имя базы данных;
- в веб-сервере: имя хоста и порт сервера RabbitMQ;
- в диспетчере очереди: имя хоста и порт сервера RabbitMQ;
- в основном сервере Postgres: имя хоста, порт, имя пользователя и пароль копии сервера 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 3.6 показан один из вариантов объявления этой информации в виде переменных групп в файле реестра.

Пример 3.6 ❖ Определение переменных групп в реестре

```
[all:vars]
ntp_server=ntp.ubuntu.com

[production:vars]
db_primary_host=rhodeisland.example.com
db_primary_port=5432
db_replica_host=virginia.example.com
db_name=widget_production
db_user=widgetuser
db_password=pFmMxcyD;Fc6)6
rabbitmq_host=pennsylvania.example.com
rabbitmq_port=5672

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

[vagrant:vars]
db_primary_host=vagrant3
db_primary_port=5432
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 [<group name>:vars]. Также отметьте, что мы воспользовались группой all, которую Ansible создает автоматически для определения переменных 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 предлагает 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. В нашем случае это один и тот же каталог.

Например, если бы я хранил сценарии в каталоге */home/lorin/playbooks/*, а файл реестра – в каталоге */home/lorin/playbooks/hosts*, я должен был бы сохранить переменные для хоста *quebec.example.com* в файле */home/lorin/playbooks/host_vars/quebec.example.com*, а переменные для группы хостов в промышленном окружении – в файле */home/lorin/playbooks/group_vars/production*.

В примере 3.7 показано, как выглядел бы файл */home/lorin/playbooks/group_vars/production*.

Пример 3.7 ❖ group_vars/production

```
db_primary_host: rhodeisland.example.com
db_primary_port=5432
db_replica_host: virginia.example.com
db_name: widget_production
db_user: widgetuser
db_password: pFmMxcyD;Fc6)6
rabbitmq_host:pennsylvania.example.com
rabbitmq_port=5672
```

Обратите внимание, что для представления этих значений также можно использовать словари YAML, как показано в примере 3.8.

Пример 3.8 ❖ group_vars/production, со словарями

```
db:
    user: widgetuser
    password: pFmMxcyD;Fc6)6
```

```
    name: widget_production
    primary:
        host: rhodeisland.example.com
        port: 5432
    replica:
        host: virginia.example.com
        port: 5432

rabbitmq:
    host: pennsylvania.example.com
    port: 5672
```

При использовании словарей YAML меняется способ доступа к переменным, сравните:

```
{{ db_primary_host }}
```

и

```
{{ 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, можно переменные, описывающие базу данных, поместить в один файл, а переменные, описывающие RabbitMQ, – в другой, как показано в примерах 3.9 и 3.10.

Пример 3.9 ❖ group_vars/production/db

```
db:
    user: widgetuser
    password: pFmMxcyD;Fc6)6
    name: widget_production
    primary:
        host: rhodeisland.example.com
        port: 5432
    replica:
        host: virginia.example.com
        port: 5432
```

Пример 3.10 ❖ group_vars/production/rabbitmq

```
rabbitmq:
    host: pennsylvania.example.com
    port: 6379
```

В общем и целом я считаю, что лучше не усложнять и не разбивать переменные на слишком большое количество файлов.

Динамический реестр

До настоящего момента мы описывали наши хосты в файле реестра. Однако вам может понадобиться хранить всю информацию о хостах во внешней си-

стеме. 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 вашими собственными серверами, используя автоматизированную систему инициализации, такую как Cobbler или Ubuntu MAAS, она уже отслеживает ваши серверы. Или, может быть, вся ваша информация хранится в одной из тех причудливых баз данных управления конфигурациями (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, Ansible будет интерпретировать его как сценарий динамического реестра и запускать его вместо чтения.

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

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

Интерфейс сценария динамического реестра

Сценарий динамического реестра должен поддерживать два параметра командной строки:

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

Вывод информации о хосте

Чтобы получить данные о конкретном хосте, Ansible вызывает сценарий динамического реестра командой

```
$ ./dynamic.py --host=vagrant2
```

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

```
{ "ansible_ssh_host": "127.0.0.1", "ansible_ssh_port": 2200,
  "ansible_ssh_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. Например, если предположить, что сценарий динамического реестра называется *dynamic.py*, тогда для получения списка членов всех групп Ansible вызовет его командой

```
$ ./dynamic.py --list
```

Результат должен выглядеть так:

```
{"production": ["delaware.example.com", "georgia.example.com",
                "maryland.example.com", "newhampshire.example.com",
                "newjersey.example.com", "newyork.example.com",
                "northcarolina.example.com", "pennsylvania.example.com",
                "rhodeisland.example.com", "virginia.example.com"],
 "staging": ["ontario.example.com", "quebec.example.com"],
 "vagrant": ["vagrant1", "vagrant2", "vagrant3"],
 "lb": ["delaware.example.com"],
 "web": ["georgia.example.com", "newhampshire.example.com",
         "newjersey.example.com", "ontario.example.com", "vagrant1"]
 "task": ["newyork.example.com", "northcarolina.example.com",
          "ontario.example.com", "vagrant2"],
 "rabbitmq": ["pennsylvania.example.com", "quebec.example.com", "vagrant3"],
 "db": ["rhodeisland.example.com", "virginia.example.com", "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_ssh_host": "127.0.0.1", "ansible_ssh_port": 2222,
                   "ansible_ssh_user": "vagrant"},
    "vagrant2": { "ansible_ssh_host": "127.0.0.1", "ansible_ssh_port": 2200,
                  "ansible_ssh_user": "vagrant"},
    ...
}
```

Написание сценария динамического реестра

Одной из удобных функций Vagrant является возможность получить список запущенных виртуальных машин командой `vagrant status`. Допустим, у нас имеется файл *Vagrantfule*, как показано в примере 3.2. Если запустить команду `vagrant status`, результат будет выглядеть, как в примере 3.11:

Пример 3.11 ❖ Выведение статуса Vagrant

```
$ 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[1]. Наш сценарий будет получать необходимую информацию, выполняя команду `vagrant status`. Ее вывод, который приводится в примере 3.11, 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:

```
1410577818,vagrant1,provider-name,virtualbox
1410577818,vagrant1,state,running
1410577818,vagrant1,state-human-short,running
1410577818,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`.
1410577818,vagrant2,provider-name,virtualbox
1410577818,vagrant2,state,running
1410577818,vagrant2,state-human-short,running
1410577818,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`.
1410577818,vagrant3,provider-name,virtualbox
1410577818,vagrant3,state,running
1410577818,vagrant3,state-human-short,running
1410577818,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`.
```

[1] Да, в Ansible уже имеется сценарий динамического реестра. Однако вам будет полезно проделать это упражнение.

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/lorinhochstein/.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. Ниже приводится интерактивный сеанс Python, объясняющий, как использовать Paramiko:

```
>>> import subprocess
>>> import paramiko
>>> cmd = "vagrant ssh-config vagrant2"
>>> p = subprocess.Popen(cmd.split(), stdout=subprocess.PIPE)
>>> config = paramiko.SSHConfig()
>>> config.parse(p.stdout)
>>> config.lookup("vagrant2")
{'identityfile': ['/Users/lorinhochstein/.vagrant.d/insecure_private_key'],
 'loglevel': 'FATAL', 'hostname': '127.0.0.1', 'passwordauthentication': 'no',
 'identitiesonly': 'yes', 'userknownhostsfile': '/dev/null', 'user': 'vagrant',
 'stricthostkeychecking': 'no', 'port': '2200'}
```

Для использования сценария необходимо установить библиотеку Paramiko для Python. Это можно сделать с помощью диспетчера пакетов `pip`:

```
$ sudo pip install paramiko
```

В примере 3.12 приводится полный сценарий *vagrant.py*.

Пример 3.12 ❖ vagrant.py

```
#!/usr/bin/env python
# Основан на реализации Марка Мандела (Mark Mandel)
# https://github.com/ansible/ansible/blob/devel/plugins/inventory/vagrant.py
# Лицензия: GNU General Public License, Version 3 <http://www.gnu.org/licenses/>
import argparse
import json
import paramiko
import subprocess
```

```
import sys

def parse_args():
    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():
    cmd = "vagrant status --machine-readable"
    status = subprocess.check_output(cmd.split()).rstrip()
    hosts = []
    for line in status.split('\n'):
        (_, host, key, value) = line.split(',')
        if key == 'state' and value == 'running':
            hosts.append(host)
        return hosts

def get_host_details(host):
    cmd = "vagrant ssh-config {}".format(host)
    p = subprocess.Popen(cmd.split(), stdout=subprocess.PIPE)
    config = paramiko.SSHConfig()
    config.parse(p.stdout)
    c = config.lookup(host)
    return {'ansible_ssh_host': c['hostname'],
            'ansible_ssh_port': c['port'],
            'ansible_ssh_user': c['user'],
            'ansible_ssh_private_key_file': c['identityfile'][0]}

def main():
    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()
```

Предопределенные сценарии реестра

В состав Ansible входит несколько сценариев динамического реестра, и вы можете использовать их. Мне никогда не удавалось понять, куда эти файлы устанавливает мой диспетчер пакетов, поэтому я всегда загружаю нужные мне непосредственно из GitHub. Вы можете загрузить их со страницы Ansible GitHub (https://github.com/ansible/ansible) непосредственно в каталог *plugins/inventory*.

Многие из этих сценариев идут в сопровождении файла конфигурации. В главе 14 мы подробно рассмотрим сценарий для Amazon EC2.

Деление реестра на несколько файлов

Если вам необходим обычный файл реестра и сценарий динамического реестра (или их комбинация), 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.

Например, вот как могла бы выглядеть структура такого каталога: *inventory/hosts* и *inventory/vagrant.py*.

Для подобной организации файл *ansible.cfg* должен содержать строки:

```
[defaults]
hostfile = inventory
```

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.

add_host

Модуль `add_host` добавляет хост в реестр. Этот модуль может пригодиться, если вы используете Ansible 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`, если потребуется запустить и настроить новый экземпляр виртуальной машины в ходе выполнения сценария.

Если новый хост появится во время выполнения сценария, 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:

```
add_host name=hostname groups=web,staging myvar=myval
```

Определение списка групп и дополнительных переменных можно опустить.

Ниже команда `add_host` представлена в действии. Она добавляет новую машину Vagrant и настраивает ее:

```
- name: Provision a vagrant machine
  hosts: localhost
  vars:
    box: trusty64
  tasks:
    - name: create a Vagrantfile
      command: vagrant init {{ box }} creates=Vagrantfile

    - name: Bring up a vagrant machine
      command: vagrant up

    - name: add the vagrant machine to the inventory
      add_host: >
          name=vagrant
          ansible_host=127.0.0.1
          ansible_port=2222
          ansible_user=vagrant
          ansible_private_key_file=/Users/lorin/.vagrant.d/
          insecure_private_key

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

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

Подготавливая свои сценарии, я предпочитаю разбивать их на две части. Первая выполняется на локальном хосте и создает хосты, а вторая настраивает их.

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

```
- name: create a Vagrantfile
  command: vagrant init {{ box }} creates=Vagrantfile
```

Он сообщает системе Ansible, что если файл *Vagrantfile* имеется, хост уже находится в правильном состоянии и нет необходимости выполнять команду снова. Это способ достижения идемпотентности в сценарии, который запускает командный модуль, благодаря которому команда (потенциально неидемпотентная) выполняется только один раз.

group_by

Посредством модуля `group_by` Ansible позволяет создавать новые группы во время исполнения сценария, основываясь на значении переменной, которая была установлена для каждого хоста и в терминологии Ansible называет *фактом*[1].

[1] Факты рассматриваются в главе 4.

Если включен сбор фактов, Ansible ассоциирует набор переменных с хостом. Например, для 32-разрядных x86 машин переменная `ansible_machine` будет иметь значение `i386`, а для 64-разрядных x86 машин – значение `x86_64`. Если Ansible управляет хостами с разной аппаратной архитектурой, 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, CentOS).

```
- name: create groups based on Linux distribution
  group_by: key={{ ansible_distribution }}
```

В примере 3.13 мы создаем отдельные группы для хостов с Ubuntu и CentOS, используя модуль `group_by`, а затем устанавливаем пакеты – в Ubuntu с помощью модуля `apt` и в CentOS с помощью модуля `yum`.

Пример 3.13 ❖ Создание специальных групп для разных дистрибутивов Linux

```
- name: group hosts by distribution
  hosts: myhosts
  gather_facts: True
  tasks:
    - name: create groups based on distro
      group_by: key={{ ansible_distribution }}

- name: do something to Ubuntu hosts
  hosts: Ubuntu
  tasks:
    - name: install htop
      apt: name=htop
    # ...

- name: do something else to CentOS hosts
  hosts: CentOS
  tasks:
    - name: install htop
      yum: name=htop
    # ...
```

Даже притом, что `group_by` – один из способов реализации условного поведения Ansible, я никогда не видел, чтобы он широко использовался. В главе 6 вы увидите пример использования параметра задачи `when` для осуществления разных действий на основе значений переменных.

На этом мы заканчиваем обсуждение реестра Ansible. В следующей главе мы поближе познакомимся с переменными. Более подробную информацию о функции *ControlPersist*, также известной как *мультиплексирование SSH*, вы найдете в главе 11.

Глава 4

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 2.8, где определили несколько переменных конфигурации:

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

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 4.1.

Пример 4.1 ❖ nginx.yml

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

В главе 6 мы увидим пример, как использовать секцию `vars_files`, чтобы переместить переменные с конфиденциальной информацией в отдельный файл.

Как уже обсуждалось в главе 3, Ansible позволяет определить переменные, связанные с хостами или группами, в файле реестра или в отдельных файлах, существующие наряду с файлом реестра.

Вывод значений переменных

Для отладки часто удобно иметь возможность вывести значения переменных. В главе 2 мы видели, как использовать модуль `debug` для вывода произвольного сообщения. Его также можно использовать для вывода значений переменных:

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

В этой главе нам несколько раз потребуется такая форма использования модуля `debug`.

Регистрация переменных

Часто требуется установить значение переменной в зависимости от результата задачи. Для этого создадим *зарегистрированную переменную* при запуске модуля с помощью ключевого слова `register`. Пример 4.2 демонстрирует, как сохранить ввод команды `whoami` в переменной `login`.

Пример 4.2 ❖ 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 4.3.

Пример 4.3 ❖ whoami.yml

```
- name: show return value of command module
  hosts: server1
  tasks:
    - name: capture output of id command
      command: id -un
      register: login
    - debug: var=login
```

Вот что выведет модуль debug:

```
TASK: [debug var=login] ***************************************************
ok: [server1] => {
    "login": {
        "changed": true, ❶
        "cmd": [ ❷
            "id",
            "-un"
        ],
        "delta": "0:00:00.002180",
        "end": "2015-01-11 15:57:19.193699",
        "invocation": {
            "module_args": "id -un",
            "module_name": "command"
        },
        "rc": 0, ❸
        "start": "2015-01-11 15:57:19.191519",
        "stderr": "", ❹
        "stdout": "vagrant", ❺
        "stdout_lines": [ ❻
            "vagrant"
        ],
        "warnings": [ ]
    }
}
```

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 запущенную команду в виде списка строк.
❸ Ключ rc содержит код возврата. Если он не равен нулю, 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 4.4.

Пример 4.4 ❖ 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 полезно как-то обработать вывод задачи, потерпевшей ошибку. Однако если задача потерпела ошибку, 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 4.5.

Пример 4.5 ❖ 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, как возвращаемое значение модуля `command`, но другие модули возвращают отличающиеся ключи. В примере 4.6 показано, что возвращается модуль `apt` после установки пакета, который не был установлен ранее.

Пример 4.6 ❖ Результат работы модуля apt при установке нового пакета

```
ok: [server1] => {
    "result": {
        "changed": true,
        "invocation": {
            "module_args": "name=nginx",
            "module_name": "apt"
        },
        "stderr": "",
        "stdout": "Reading package lists...\nBuilding dependency tree...",
        "stdout_lines": [
            "Reading package lists...",
            "Building dependency tree...",
            "Reading state information...",
            "Preparing to unpack .../nginx-common_1.4.6-1ubuntu3.1_all.deb ...",
            ...
            "Setting up nginx-core (1.4.6-1ubuntu3.1) ...",
            "Setting up nginx (1.4.6-1ubuntu3.1) ...",
            "Processing triggers for libc-bin (2.19-0ubuntu6.3) ..."
        ]
    }
}
```

Организация доступа к ключам словаря в переменной

Если переменная содержит словарь, получить доступ к его ключам можно при помощи точки (`.`) или индекса (`[ ]`). В примере 4.4 был представлен способ ссылки на переменную с использованием точки:

```
{{ login.stdout }}
```

Однако точно так же можно было бы использовать индекс:

```
{{ login['stdout'] }}
```

Это правило применимо к любому уровню вложенности, то есть все следующие выражения эквивалентны:

```
ansible_eth1['ipv4']['address']
ansible_eth1['ipv4'].address
ansible_eth1.ipv4['address']
ansible_eth1.ipv4.address
```

Обычно я предпочитаю пользоваться точкой, кроме случаев, когда ключ содержит символы, которые нельзя использовать в качестве имени переменной, такие как точка, пробел или дефис.
Для разыменования переменных Ansible использует Jinja2. За дополнительной информацией обращайтесь к документации Jinja2 на странице: *http://jinja.pocoo.org/docs/dev/templates/#variables*.

В примере 4.7 показано, что возвращает модуль `apt`, когда пакет уже был установлен на хосте.

Пример 4.7 ❖ Результат работы модуля apt, когда пакет уже установлен

```
ok: [server1] => {
    "result": {
        "changed": false,
        "invocation": {
            "module_args": "name=nginx",
            "module_name": "apt"
        }
    }
}
```

Обратите внимание, что ключи `stdout`, `stderr` и `stdout_lines` присутствуют в возвращаемом значении, только если прежде пакет не был установлен.

Если вы собираетесь использовать зарегистрированные переменные в своих сценариях, 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.

Факты

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

```
GATHERING FACTS ***************************************************
ok: [servername]
```

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. Эта информация

сохраняется в переменных, называемых *фактами*. Это самые обычные переменные, как любые другие.

Вот простой сценарий, который выводит названия операционной системы для каждого сервера:

```
- name: print out operating system
  hosts: all
  gather_facts: True
  tasks:
- debug: var=ansible_distribution
```

Так выглядит вывод для серверов с Ubuntu и CentOS.

```
PLAY [print out operating system] **********************************************

GATHERING FACTS ****************************************************************
ok: [server1]
ok: [server2]

TASK: [debug var=ansible_distribution] *****************************************
ok: [server1] => {
    "ansible_distribution": "Ubuntu"
}
ok: [server2] => {
    "ansible_distribution": "CentOS"
}

PLAY RECAP *********************************************************************
server1                    : ok=2    changed=0    unreachable=0    failed=0
server2                    : ok=2    changed=0    unreachable=0    failed=0
```

Список некоторых доступных фактов можно найти в официальной документации Ansible (http://bit.ly/1G9pVfx). Я поддерживаю более полный список фактов в GitHub (http://bit.ly/1G9pX7a).

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 server1 -m setup
```

Ansible выведет все факты, как показано в примере 4.8.

Пример 4.8 ❖ Результат запуска модуля setup

```
server1 | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.2.15",
            "192.168.4.10"
        ],
```

```
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fefe:1e4d",
            "fe80::a00:27ff:fe67:bbf3"
        ],
    (множество других фактов)
```

Обратите внимание, что возвращаемое значение является словарем с ключом `ansible_facts`, значением которого является словарь, содержащий имена 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[1]. Например, команда

```
$ ansible web -m setup -a 'filter=ansible_eth*'
```

выведет:

```
web | success >> {
    "ansible_facts": {
        "ansible_eth0": {
            "active": true,
            "device": "eth0",
            "ipv4": {
                "address": "10.0.2.15",
                "netmask": "255.255.255.0",
                "network": "10.0.2.0"
            },
            "ipv6": [
                {
                    "address": "fe80::a00:27ff:fefe:1e4d",
                    "prefix": "64",
                    "scope": "link"
                }
            ],
            "macaddress": "08:00:27:fe:1e:4d",
            "module": "e1000",
            "mtu": 1500,
            "promisc": false,
            "type": "ether"
        },
        "ansible_eth1": {
            "active": true,
            "device": "eth1",
            "ipv4": {
                "address": "192.168.33.10",
```

[1] Шаблонные символы, например, поддерживают командные оболочки, позволяя определять шаблоны файлов (например, `*.txt`).

```
                "netmask": "255.255.255.0",
                "network": "192.168.33.0"
            },
            "ipv6": [
                {
                    "address": "fe80::a00:27ff:fe23:ae8e",
                    "prefix": "64",
                    "scope": "link"
                }
            ],
            "macaddress": "08:00:27:23:ae:8e",
            "module": "e1000",
            "mtu": 1500,
            "promisc": false,
            "type": "ether"
        }
    },
    "changed": false
}
```

Любой модуль может возвращать факты

Если внимательно рассмотреть пример 4.8, 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.

Для модулей, возвращающих факты, нет необходимости регистрировать переменные, поскольку Ansible создает их автоматически. Например, следующие задачи используют модуль `ec2_facts` для извлечения фактов Amazon EC2[1] о сервере и вывода идентификатора экземпляра.

```
- name: get ec2 facts
  ec2_facts:

- debug: var=ansible_ec2_instance_id
```

Результат будет выглядеть так:

```
TASK: [debug var=ansible_ec2_instance_id] *********************************
ok: [myserver] => {
    "ansible_ec2_instance_id": "i-a3a2f866"
}
```

Обратите внимание, что нет необходимости использовать ключевое слово `register` при вызове модуля `ec2_facts`, потому что он возвращает факты. В состав Ansible входит несколько модулей, возвращающих факты. Один из них, модуль `docker`, мы рассмотрим в главе 15.

---

[1] Amazon EC2 будет рассматриваться в главе 14.

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, не принимающими аргументов и возвращающими результат в формате JSON.

Эти факты доступны в виде ключей особой переменной `ansible_local`.

В примере 4.9 приводится файл факта в формате *.ini*.

Пример 4.9 ❖ /etc/ansible/facts.d/example.fact

```
[book]
title=Ansible: Up and Running
author=Lorin Hochstein
publisher=O'Reilly Media
```

Если скопировать этот файл в */etc/ansible/facts.d/example.fact* на удаленном хосте, мы получим доступ 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: [server1] => {
    "ansible_local": {
        "example": {
            "book": {
                "author": "Lorin Hochstein",
                "publisher": "O'Reilly Media",
                "title": "Ansible: Up and Running"
            }
        }
    }
}

TASK: [print book title] ******************************************************
ok: [server1] => {
    "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. Я часто использую set_fact непосредственно после register, чтобы упростить обращения к переменным. Пример 4.10 демонстрирует, как использовать set_fact, чтобы к переменной можно было обращаться по имени snap вместо snap_result.stdout.

Пример 4.10 ❖ Использование set_fact для упрощения ссылок на переменные

```
- name: get snapshot id
  shell: >
    aws ec2 describe-snapshots --filters
    Name=tag:Name,Values=my-snapshot
    | jq --raw-output ".Snapshots[].SnapshotId"
  register: snap_result

- set_fact: snap={{ snap_result.stdout }}

- name: delete old snapshot
  command: aws ec2 delete-snapshot --snapshot-id "{{ snap }}"
```

Встроенные переменные

Ansible определяет несколько переменных, всегда доступных в сценариях. Они перечислены в табл. 4.1.

Таблица 4.1. Встроенные переменные

Параметр	Описание
hostvars	Словарь, ключи которого – имена хостов Ansible, а значения – словари, отображающие имена переменных в их значения
inventory_hostname	Полное квалифицированное доменное имя текущего хоста, как оно задано в Ansible (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 тестовом режиме (см. раздел «Тестовый режим» в главе 16)
ansible_play_batch	Список имен хостов из реестра, активных в текущем пакете (см. раздел «Пакетная обработка хостов» в главе 9)
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 файл конфигурации, 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 кэширование фактов[1].

Продолжим наш пример. Если сервер базы данных имеет имя *db.example.com*, тогда мы можем добавить в шаблоне 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*.

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:

```
server1 ansible_ssh_host=192.168.4.10
```

Тогда переменная inventory_hostname получит значение server1.

Вот как с помощью hostvars и inventory_hostname можно вывести все переменные, связанные с текущим хостом:

```
- debug: var=hostvars[inventory_hostname]
```

---

[1] Информация о кэшировании данных приводится в главе 11.

groups

Переменная groups может пригодиться для доступа к переменным, определенным для группы хостов. Допустим, мы настраиваем хост балансировщика нагрузки, и требуется добавить в файл конфигурации IP-адреса всех серверов в группе web. Тогда мы можем добавить в наш конфигурационный файл следующий фрагмент:

```
backend web-backend
{% for host in groups.web %}
  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
```

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 4.11 показано, как установить переменную token со значением 12345.

Пример 4.11 ❖ Установка переменной в командной строке

```
$ ansible-playbook example.yml -e token=12345
```

Используйте метод ansible-playbook -e var=value, когда сценарий Ansible предполагается применять подобно сценарию командной оболочки, принимающему аргумент командной строки. Параметр -e позволяет передавать переменные как аргументы.

В примере 4.12 демонстрируется очень простой сценарий, который выводит сообщение, определяемое переменной.

Пример 4.12 ❖ greet.yml

```
- name: pass a message on the command line
  hosts: localhost
  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
```

он выведет:

```
PLAY [pass a message on the command line] ***********************************

TASK: [output a message] ******************************************************
ok: [localhost] => {
    "msg": "hiya"
}

PLAY RECAP ********************************************************************
localhost              : ok=1     changed=0     unreachable=0     failed=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 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 переменными, для чего в параметре `-e` следует передать имя файла `@filename.yml`. Например, допустим, что у нас имеется файл, как показано в примере 4.13.

Пример 4.13 ❖ greetvars.yml

```
greeting: hiya
```

Этот файл можно передать сценарию, как показано ниже:

```
$ ansible-playbook greet.yml -e @greetvars.yml
```

Приоритет

Мы рассмотрели несколько различных 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, имейте в виду правила приоритета Ansible. Когда одна переменная определяется множеством способов, правила приоритета определяют, какое из значений она получит в конце концов.

Основные правила приоритета выглядят так:

1. (Высший) `ansible-playbook -e var=value`.
2. Переменные задач.
3. Блочные переменные.
4. Переменные ролей и включений.
5. Модуль `set_fact`.
6. Зарегистрированные переменные.
7. Секция `vars_files`.
8. `vars_prompt`.
9. Переменные сценария.

10. Факты хостов.
11. Секция `host_vars` в сценарии.
12. Секция `group_vars` в сценарии.
13. Секция `host_vars` в реестре.
14. Секция `group_vars` в реестре.
15. Переменные реестра.
16. В файле *defaults/main.yml* роли[1].

В этой главе мы рассмотрели разные способы определения переменных и доступа к фактам и переменным. В следующей главе мы сконцентрируемся на практических примерах развертывания приложений.

[1] Роли обсуждаются в главе 7.

Глава 5

Введение в Mezzanine: тестовое приложение

В главе 2 мы рассмотрели основные правила написания сценариев. Но в реальной жизни все более запутано, чем во вводных главах книг по программированию. По этой причине мы рассмотрим законченный пример развертывания нетривиального приложения.

В качестве примера используем систему управления контентом (CMS) Mezzanine (http://mezzanine.jupo.org/), сходную по духу с WordPress. Mezzanine устанавливается поверх Django, свободно распространяемого фреймворка веб-приложений.

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 5.1 показано, что необходимо для запуска приложения на ноутбуке[1].

Пример 5.1 ❖ Запуск Mezzanine в окружении разработки

```
$ virtualenv venv
$ source venv/bin/activate
$ pip install mezzanine
$ mezzanine-project myproject
$ cd myproject
```

[1] В данном случае будет произведена установка пакетов Python в виртуальное окружение. Подробнее о виртуальных окружениях мы поговорим в разделе «Установка Mezzanine и других пакетов в виртуальное окружение» в главе 6.

```
$ sed -i.bak 's/ALLOWED_HOSTS = \[\]/ALLOWED_HOSTS = ["127.0.0.1"]/' myproject\
/settings.py
$ python manage.py createdb
$ python manage.py runserver
```

Вам будет предложено ответить на несколько вопросов. Я ответил «да» на каждый вопрос, требующий ответа «да» или «нет», и принял ответы по умолчанию там, где они были предложены. Вот так выглядели мои действия:

```
Operations to perform:
  Apply all migrations: admin, auth, blog, conf, contenttypes, core,
  django_comments, forms, galleries, generic, pages, redirects, sessions, sites,
  twitter
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying sites.0001_initial... OK
  Applying blog.0001_initial... OK
  Applying blog.0002_auto_20150527_1555... OK
  Applying conf.0001_initial... OK
  Applying core.0001_initial... OK
  Applying core.0002_auto_20150414_2140... OK
  Applying django_comments.0001_initial... OK
  Applying django_comments.0002_update_user_email_field_length... OK
  Applying django_comments.0003_add_submit_date_index... OK
  Applying pages.0001_initial... OK
  Applying forms.0001_initial... OK
  Applying forms.0002_auto_20141227_0224... OK
  Applying forms.0003_emailfield... OK
  Applying forms.0004_auto_20150517_0510... OK
  Applying forms.0005_auto_20151026_1600... OK
  Applying galleries.0001_initial... OK
  Applying galleries.0002_auto_20141227_0224... OK
  Applying generic.0001_initial... OK
  Applying generic.0002_auto_20141227_0224... OK
  Applying pages.0002_auto_20141227_0224... OK
  Applying pages.0003_auto_20150527_1555... OK
  Applying redirects.0001_initial... OK
```

```
  Applying sessions.0001_initial... OK
  Applying sites.0002_alter_domain_unique... OK
  Applying twitter.0001_initial... OK

A site record is required.
Please enter the domain and optional port in the format 'domain:port'.
For example 'localhost:8000' or 'www.example.com'.
Hit enter to use the default (127.0.0.1:8000):

Creating default site record: 127.0.0.1:8000 ...


Creating default account ...

Username (leave blank to use 'lorin'):
Email address: lorin@ansiblebook.com
Password:
Password (again):
Superuser created successfully.
Installed 2 object(s) from 1 fixture(s)

Would you like to install some initial demo pages?
Eg: About us, Contact form, Gallery. (yes/no): yes
```

В конечном итоге вы должны увидеть следующий результат на терминале:

```
           .....
         _d^^^^^^^^^b_
      .d''           ``b.
    .p'                `q.
   .d'                   `b.
  .d'                     `b.  * Mezzanine 4.2.2
  ::                       ::  * Django 1.10.2
 ::    M E Z Z A N I N E    :: * Python 3.5.2
  ::                       ::  * SQLite 3.14.1
  `p.                     .q'  * Darwin 16.0.0
    `p.                 .q'
     `b.               .d'
       `q..          ..p'
          ^q........p^
              ''''

Performing system checks...

System check identified no issues (0 silenced).
October 04, 2016 - 04:57:44
Django version 1.10.2, using settings 'myproject.settings'
Starting development server at http://127.0.0.1:8000/
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. 5.1.

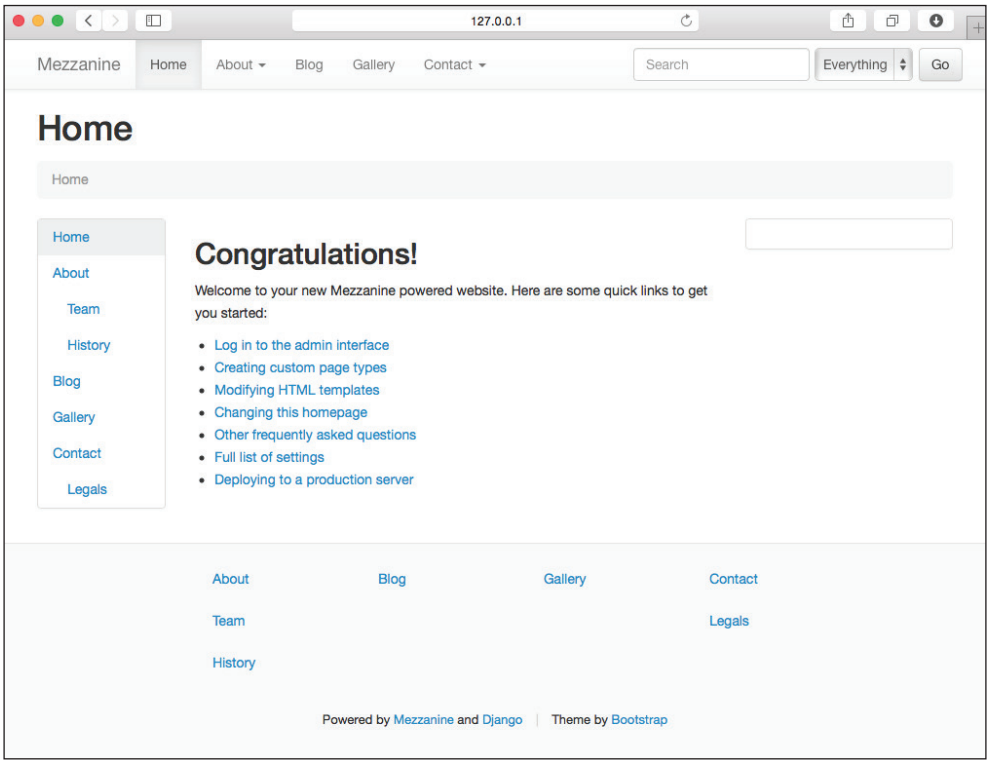

Рис. 5.1 ❖ Главная страница Mezzanine сразу после установки

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, 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. 5.2):

- 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`).

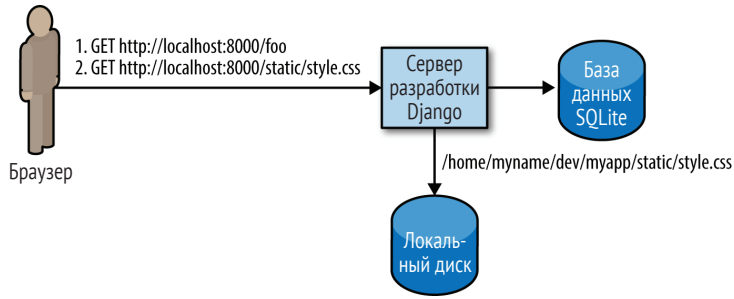

Рис. 5.2 ❖ Приложение Django в режиме разработки

Теперь посмотрим, что происходит при развертывании в промышленном окружении.

База данных PostgreSQL

SQLite – встраиваемая база данных. В промышленном окружении мы должны запустить промышленную базу данных, которая обеспечит лучшую поддержку многочисленных одновременных запросов и позволит 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 трудозатраты. 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://docs.djan-

goproject.com/en/1.7/intro/tutorial01/#the-development-server) в документации к Django 1.10:

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 любой HTTP-сервер, поддерживающий WSGI, подойдет для запуска Django-приложений, таких как Mezzanine. Мы будем использовать Gunicorn – один из популярных HTTP-серверов с поддержкой WSGI, который использует сценарий развертывания Mezzanine.

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, поскольку 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].

Для обработки статических объектов и поддержки шифрования TLS мы будем использовать Nginx, как показано на рис. 5.3.

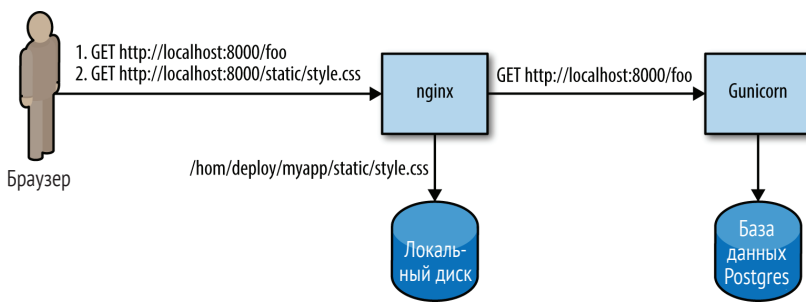

Рис. 5.3 ❖ Nginx как реверсивный прокси

Мы должны настроить Nginx как *реверсивный прокси* для Gunicorn. Если поступит запрос на статический объект, например файл *.css*, 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.

---

[1] Протокол WSGI задокументирован в Python Enhancement Proposal (PEP) 3333 (https://www.python.org/dev/peps/pep-3333/).

[2] Поддержка шифрования TLS была добавлена в Gunicorn 0.17. До этого для поддержки шифрования приходилось использовать отдельное приложение, такое как Nginx.

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 мы запустили процесс.

В просторечии такие процессы называют *демонами*, или *службами*. Мы должны запустить Gunicorn как демон, и еще нам нужна возможность с легкостью останавливать и перезапускать его. Существует много диспетчеров задач, способных выполнить эту работу. Мы будем использовать Supervisor, поскольку именно его используют сценарии развертывания Mezzanine.

Теперь вы должны понимать, что требуется для развертывания веб-приложения в промышленном окружении. В главе 6 мы перейдем к реализации этой задачи с помощью Ansible.

Глава 6

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[1], в конце главы в примере 6.28 вы увидите сценарий полностью. Он также доступен в репозитории GitHub по адресу: http://bit.ly/19P0OAj. Прежде чем запустить его, прочитайте файл *README*: http://bit.ly/1Onko4u.

Я старался оставаться как можно ближе к оригинальным сценариям Fabric[2], которые написал Стефан МакДональд (Stephen McDonald), автор Mezzanine.

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 в сценарии. Это простой способ выяснить, какие действия производятся сценарием. Вот как можно ее использовать:

```
$ ansible-playbook --list-tasks mezzanine.yml
```

Пример 6.1 демонстрирует вывод этой команды для сценария *mezzanine.yml*, приведенного в примере 6.28.

Пример 6.1 ❖ Список задач в сценарии Mezzanine

```
playbook: mezzanine.yml

  play #1 (web): Deploy mezzanine     TAGS: []
    tasks:
```

---

[1] Моя жена Стейси особенно преуспела в этом.

[2] Сценарии Fabric, поставляемые с Mezzanine, можно найти по адресу: http://bit.ly/19P0T73.

```
install apt packages     TAGS: []
create project path      TAGS: []
create a logs directory  TAGS: []
check out the repository on the host     TAGS: []
install Python requirements globally via pip     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: []
install requirements.txt  TAGS: []
install required python packages  TAGS: []
apply migrations to create the database, collect static content   TAGS: []
set the site id   TAGS: []
set the admin password    TAGS: []
```

Организация устанавливаемых файлов

Как уже говорилось, Mezzanine развертывается поверх Django. В терминологии Django веб-приложение называется *проектом*. Нам нужно дать имя проекту, я выбрал *mezzanine_example*.

Наш сценарий производит установку на машину Vagrant и помещает файлы в домашний каталог пользователя Vagrant.

В примере 6.2 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 для установки дополнительных пакетов на языке Python;
- */home/vagrant/logs* – каталог для хранения журналов, создаваемых приложением Mezzanine.

Пример 6.2 ❖ Структура каталогов в /home/vagrant

```
.
├── logs
├── mezzanine
│   └── mezzanine_example
└── .virtualenvs
    └── mezzanine_example
```

Переменные и скрытые переменные

Как показано в примере 6.3, сценарий определяет довольно много переменных.

Пример 6.3 ❖ Определение 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.xip.io
  domains:
    - 192.168.33.10.xip.io
    - www.192.168.33.10.xip.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/python"
  database_name: "{{ proj_name }}"
  database_user: "{{ proj_name }}"
  database_host: localhost
  database_port: 5432
  gunicorn_procname: gunicorn_mezzanine
  num_workers: "multiprocessing.cpu_count() * 2 + 1"
vars_files:
  - secrets.yml
```

В большинстве случаев я старался использовать те же имена переменных, какие использует Fabric-сценарий установки Mezzanine. Я также добавил несколько переменных, чтобы сделать процесс более прозрачным. Например, сценарии Fabric используют переменную `proj_name` для хранения имени базы данных и имени пользователя базы данных. Я предпочитаю задавать вспомогательные переменные, такие как `database_name` и `data_base_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 моем 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:
# http://mezzanine.jupo.org/docs/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. Также отметьте, 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 ваш 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 зашифрованной версии файла *secrets.yml* при помощи утилиты `vault` из Ansible.

Использование цикла (with_items) для установки большого количества пакетов

Нам потребуется установить два типа пакетов, чтобы развернуть Mezzanine. Во-первых, мы должны установить системные пакеты. Поскольку мы собираемся развертывать приложение в Ubuntu, будем использовать для этого диспетчер пакетов `apt`. Во-вторых, нам нужно установить пакеты Python, и для этих целей мы воспользуемся диспетчером `pip`.

Устанавливать системные пакеты обычно проще, чем пакеты Python, потому что они созданы для непосредственного использования с операционной системой. Однако в репозиториях системных пакетов зачастую отсутствуют новейшие версии библиотек для Python, которые нам необходимы. Поэтому

для их установки воспользуемся диспетчером пакетов для Python. Это компромисс между стабильностью и использованием новейших и самых лучших версий.

В примере 6.4 показана задача, которую мы используем для установки системных пакетов.

Пример 6.4 ❖ Установка системных пакетов

```
- name: install apt packages
  apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
  become: True
  with_items:
    - git
    - libjpeg-dev
    - libpq-dev
    - memcached
    - nginx
    - postgresql
    - python-dev
    - python-pip
    - python-psycopg2
    - python-setuptools
    - python-virtualenv
    - supervisor
```

Для установки большого количества пакетов мы воспользовались поддержкой цикла в Ansible, выражением `with_items`. Мы могли бы устанавливать пакеты по одному, как показано ниже:

```
- name: install git
  apt: pkg=git

- name: install libjpeg-dev
  apt: pkg=libjpeg-dev
...
```

Однако гораздо проще сгруппировать все пакеты в список. Вызывая модуль `apt`, мы передаем ему `{{ item }}`. Эта переменная цикла будет последовательно принимать значения элементов списка в выражении `with_items`.

Ansible всегда использует имя `item` для обозначения переменной цикла. В главе 8 вы увидите, как можно использовать другие имена.

Кроме того, модуль `apt` оптимизирует одновременную установку нескольких пакетов с применением выражения `with_items`. Ansible передает модулю `apt` полный список пакетов, и модуль вызывает программу `apt` только один раз, передавая ей сразу весь список пакетов для установки. Некоторые модули, подобные `apt`, предусматривают обработку списков таким способом. Если в модуле отсутствует собственная поддержка списков, Ansible просто будет вызывать модуль много раз – для каждого элемента списка в отдельности.

Можно сказать, что модуль apt достаточно умен, чтобы обработать большое количество пакетов одновременно, поэтому результат выглядит так:

```
TASK: [install apt packages] **************************************************
ok: [web] => (item=[u'git', u'libjpeg-dev', u'libpq-dev', u'memcached',
u'nginx', u'postgresql', u'python-dev', u'python-pip', u'python-psycopg2',
u'python-setuptools', u'python-virtualenv', u'supervisor'])
```

С другой стороны, модуль pip не поддерживает списков пакетов, поэтому Ansible вынуждена вызывать его снова для каждого элемента списка, соответственно, результат выглядит так:

```
TASK [install required python packages] ***************************************
ok: [web] => (item=gunicorn)
ok: [web] => (item=setproctitle)
ok: [web] => (item=psycopg2)
ok: [web] => (item=django-compressor)
ok: [web] => (item=python-memcached)
```

Добавление выражения become в задачу

В примерах сценариев в главе 2 нам требовалось, чтобы сценарий целиком выполнялся с привилегиями пользователя root, поэтому мы добавляли 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 6.4.

Обновление кэша диспетчера пакетов apt

Все примеры команд в этом разделе выполняются на удаленном хосте (Ubuntu), а не на управляющей машине.

Ubuntu поддерживает кэш с именами 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 кэша информацию об известной версии этой программы:

```
$ apt-cache policy libssl-dev
```

Результат показан в примере 6.5.

Пример 6.5 ❖ Вывод apt cache

```
libssl-dev:
  Installed: (none)
  Candidate: 1.0.1f-1ubuntu2.21
```

```
  Version table:
    1.0.1f-1ubuntu2.21 0
      500 http://archive.ubuntu.com/ubuntu/ trusty-updates/main amd64 Packages
      500 http://security.ubuntu.com/ubuntu/ trusty-security/main amd64 Packages
    1.0.1f-1ubuntu2 0
      500 http://archive.ubuntu.com/ubuntu/ trusty/main amd64 Packages
```

Как видите, этот пакет не был установлен. Согласно информации из кэша, на локальной машине новейшая версия – 1.0.1f-1ubuntu2.21. 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 кэш `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*:

```
$ apt-get install libssl-dev
```

Если версия 1.0.1f-1ubuntu2.21 больше не доступна в архиве пакетов, мы увидим следующее сообщение:

```
Err http://archive.ubuntu.com/ubuntu/ trusty-updates/main libssl-dev amd64
1.0.1f-1ubuntu2.21
  404  Not Found [IP: 91.189.88.153 80]
Err http://security.ubuntu.com/ubuntu/ trusty-security/main libssl-dev amd64
1.0.1f-1ubuntu2.21
  404  Not Found [IP: 91.189.88.149 80]
Err http://security.ubuntu.com/ubuntu/ trusty-security/main libssl-doc all
1.0.1f-1ubuntu2.21
  404  Not Found [IP: 91.189.88.149 80]
E: Failed to fetch
http://security.ubuntu.com/ubuntu/pool/main/o/openssl/libssl-dev_1.0.1f-1ubuntu2.
21_amd64.deb
404 Not Found [IP: 91.189.88.149 80]
Updating the Apt Cache | 99
E: Failed to fetch
http://security.ubuntu.com/ubuntu/pool/main/o/openssl/libssl-doc_1.0.1f-1ubuntu2.
21_all.deb
404 Not Found [IP: 91.189.88.149 80]

E: Unable to fetch some archives, maybe run apt-get update or try with
--fix-missing?
```

Чтобы привести локальный кэш пакетов `apt` в актуальное состояние, можно выполнить команду `apt-get update`. Вызывая модуль `apt` в Ansible, ему необходимо передать аргумент `update_cache=yes`, чтобы обеспечить поддержание локального кэша `apt` в актуальном состоянии, как это показано в примере 6.4.

Обновление кэша занимает некоторое время, а мы можем запускать сценарий много раз подряд для отладки, поэтому, чтобы избежать ненужных

затрат времени на обновление кэша, можно передать модулю аргумент `cache_valid_time`. Он разрешает обновление кэша, только если тот старше установленного порогового значения. В примере 6.4 используется аргумент `cache_valid_time=3600`, который разрешает обновление кэша, только если он старше 3600 секунд (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, если вы знаете язык 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, 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://github.com/lorin/mezzanine_example) с проектом Django, содержащий все необходимые файлы. Этот проект будет развертывать наш сценарий.

С помощью программы `mezzanine-project`, которая поставляется вместе с Mezzanine, я создал файлы проекта, как показано ниже:

```
$ mezzanine-project mezzanine_example
$ chmod +x mezzanine_example/manage.py
```

Учтите, что у меня в 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 6.6 показано, как использовать модуль `git` для извлечения проекта из удаленного репозитория Git.

Пример 6.6 ❖ Извлечение проекта из репозитория Git

```
- name: check out the repository on the host
  git: repo={{ repo_url }} dest={{ proj_path }} accept_hostkey=yes
```

Я открыл общий доступ к репозиторию, чтобы читатели смогли обращаться к нему, но в реальной жизни вам придется обращаться к закрытым репозиториям Git по SSH. Поэтому я настроил переменную `repo_url` для использования схемы, которая клонирует репозиторий по SSH:

```
repo_url: git@github.com:lorin/mezzanine_example.git
```

Если вы пробуете выполнять примеры на своем компьютере, для запуска сценария вы должны:

1) иметь учетную запись на GitHub;
2) иметь публичный ключ SSH, связанный с вашей учетной записью на GitHub;
3) запустить SSH-агента на управляющей машине с включенным агентом перенаправления;
4) добавить свой ключ SSH в SSH-агента.

Запустив SSH-агента, добавьте в него свой ключ:

```
$ ssh-add
```

В случае успеха следующая команда выведет публичный ключ SSH, только что добавленный вами:

```
$ ssh-add -l
```

Вывод должен выглядеть примерно так:

```
2048 SHA256:o7H/I9rRZupXHJ7JnDi10RhSzeAKYiRVrlH9L/JFtfA /Users/lorin/.ssh/id_rsa
```

Чтобы включить агента перенаправления, добавьте следующие строки в файл *ansible.cfg*:

```
[ssh_connection]
ssh_args = -o ControlMaster=auto -o ControlPersist=60s -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, выполнив команду

```
$ ansible web -a "ssh -T git@github.com"
```

В случае успеха ее вывод должен выглядеть примерно так:

```
web | FAILED | rc=1 >>
Hi lorin! You've successfully authenticated, but GitHub does not provide shell
access.
```

Пусть вас не смущает слово `FAILED`[1] в выводе, если появилось сообщение от сервера GitHub, значит, все в порядке.

Кроме URL-репозитория в параметре `repo` и пути к репозиторию в параметре `dest`, мы должны также передать дополнительный параметр `accept_hostkey`, связанный с *проверкой ключей хоста*. Агента перенаправления SSH и проверку ключей хоста мы подробно рассмотрим в приложении А.

---

[1] Переводится как «неудача, ошибка». – *Прим. перев.*

Установка Mezzanine и других пакетов в virtualenv

Как уже упоминалось выше в этой главе, мы установим некоторые пакеты как пакеты Python, чтобы получить более свежие версии, чем доступны диспетчеру `apt`.

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.

Модуль Ansible `pip` позволяет создавать такие изолированные окружения virtualenv и устанавливать пакеты в них.

В примере 6.7 демонстрируется использование модуля `pip` для установки пакетов Python в системный каталог. Обратите внимание, что для этого необходим параметр `become: True`.

Пример 6.7 ❖ Установка пакетов Python в системный каталог

```
- name: install Python requirements globally via pip
  pip: name={{ item }} state=latest
  with_items:
    - pip
    - virtualenv
    - virtualenvwrapper
  become: True
```

В примере 6.8 приводятся две задачи, которые устанавливают пакеты Python в изолированное окружение. Обе они используют модуль pip, хотя и немного по-разному.

Пример 6.8 ❖ Установка пакетов Python в изолированное окружение

```
- name: install requirements.txt
  pip: requirements={{ proj_path }}/{{ reqs_path }} virtualenv={{ venv_path }}

- name: install required python packages
  pip: name={{ item }} virtualenv={{ venv_path }}
  with_items:
    - gunicorn
    - setproctitle
    - psycopg2
    - django-compressor
    - python-memcached
```

Общим для проектов Python является перечисление пакетов зависимостей в файле *requirements.txt*. И действительно, репозиторий в нашем примере с системой Mezzanine содержит файл *requirements.txt*. Он приводится в примере 6.9.

Пример 6.9 ❖ requirements.txt

```
Mezzanine==4.2.2
```

В файл *requirements.txt* не включены некоторые другие пакеты Python, которые требуется установить. Поэтому для их установки мы создали отдельную задачу.

Обратите внимание, что в файле *requirements.txt* указана конкретная версия пакета Python Mezzanine (4.2.2), в то время как для остальных пакетов версии не указаны. В этом случае будет установлена новейшая доступная версия. Если бы нам не требовалось зафиксировать версию Mezzanine, мы могли бы добавить имя пакета в общий список, как показано ниже:

```
- name: install python packages
  pip: name={{ item }} virtualenv={{ venv_path }}
  with_items:
    - mezzanine
    - gunicorn
    - setproctitle
    - south
    - psycopg2
    - django-compressor
    - python-memcached
```

И наоборот, если бы понадобилось зафиксировать версии всех пакетов, у нас на выбор было бы несколько вариантов. Можно было бы создать файл *requirements.txt* с информацией о пакетах и их зависимостях. Содержимое такого файла приводится в примере 6.10.

Пример 6.10 ❖ Пример файла requirements.txt

```
beautifulsoup4==4.5.3
bleach==1.5.0
chardet==2.3.0
Django==1.10.4
django-appconf==1.0.2
django-compressor==2.1
django-contrib-comments==1.7.3
filebrowser-safe==0.4.6
future==0.16.0
grappelli-safe==0.4.5
gunicorn==19.6.0
html5lib==0.9999999
Mezzanine==4.2.2
oauthlib==2.0.1
olefile==0.43
Pillow==4.0.0
psycopg2==2.6.2
python-memcached==1.58
pytz==2016.10
```

```
rcssmin==1.0.6
requests==2.12.4
requests-oauthlib==0.7.0
rjsmin==1.0.12
setproctitle==1.1.10
six==1.10.0
tzlocal==1.3
```

Если бы у нас уже имелось готовое изолированное окружение virtualenv с установленными в него пакетами, мы могли бы воспользоваться командой `pip freeze`, чтобы вывести список установленных пакетов. Например, если окружение virtualenv находится в *~/mezzanine_example*, активировать его и получить список установленных пакетов можно было бы так:

```
$ source ~/mezzanine_example/bin/activate
$ pip freeze > requirements.txt
```

В примере 6.11 показано, как можно установить пакеты с использованием файла *requirements.txt*.

Пример 6.11 ❖ Установка из requirements.txt

```
- name: copy requirements.txt file
  copy: src=files/requirements.txt dest=~/requirements.txt
- name: install packages
  pip: requirements=~/requirements.txt virtualenv={{ venv_path }}
```

Также можно было бы указать в списке не только имена пакетов, но и их версии, как показано в примере 6.12. Мы передаем список словарей и разыменовываем элементы, обращаясь к ним как `item.name` и `item.version`.

Пример 6.12 ❖ Определение имен пакетов и их версий

```
- name: python packages
  pip: name={{ item.name }} version={{ item.version }} virtualenv={{ venv_path }}
  with_items:
    - {name: mezzanine, version: 4.2.2 }
    - {name: gunicorn, version: 19.6.0 }
    - {name: setproctitle, version: 1.1.10 }
    - {name: psycopg2, version: 2.6.2 }
    - {name: django-compressor, version: 2.1 }
    - {name: python-memcached, version: 1.58 }
```

Короткое отступление: составные аргументы задач

До настоящего момента каждый раз, вызывая модуль, мы передавали ему аргумент в виде строки. В примере 6.12 мы передали модулю `pip` строку в аргументе:

```
- name: install package with pip
  pip: name={{ item.name }} version={{ item.version }} virtualenv={{ venv_path }}
```

Если вам не нравятся длинные строки, строку аргумента можно разбить на несколько строк с помощью функции свертки строк в YAML, о чем уже упоминалось в разделе «Объединение строк» в главе 2:

```
- name: install package with pip
  pip: >
    name={{ item.name }}
    version={{ item.version }}
    virtualenv={{ venv_path }}
```

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 6.12 мог бы выглядеть так:

```
- name: install package with pip
  pip:
    name: "{{ item.name }}"
    version: "{{ item.version }}"
    virtualenv: "{{ venv_path }}"
```

Подход на основе словарей также очень удобно использовать для вызова модулей, использующих составные аргументы. *Составной аргумент* – это аргумент, включающий список или словарь. Хорошим примером модуля с составными аргументами может служить модуль `ec2`. В примере 6.13 показано, как можно обратиться к модулю, принимающему список в аргументе `group` и словарь в аргументе `instance_tags`. Более подробно данный модуль рассматривается в главе 14.

Пример 6.13 ❖ Вызов модуля с составными аргументами

```
- name: create an ec2 instance
  ec2:
    image: ami-8caa1ce4
    instance_type: m3.medium
    key_name: mykey
    group:
      - web
      - ssh
  instance_tags:
    type: web
    env: production
```

Эти приемы передачи аргументов можно смешивать и передавать одни аргументы в виде строк, а другие в виде словарей, с помощью выражения `args`. Например, предыдущий пример можно переписать так:

```
- name: create an ec2 instance
  ec2: image=ami-8caa1ce4 instance_type=m3.medium key_name=mykey
  args:
    group:
```

```
      - web
      - ssh
    instance_tags:
      type: web
      env: production
```

При использовании выражения `local_action` (мы рассмотрим его в главе 9) синтаксис составных аргументов несколько изменяется. Необходимо добавить `module: <имя_модуля>`, как показано ниже:

```
- name: create an ec2 instance
  local_action:
    module: ec2
    image: ami-8caa1ce4
    instance_type: m3.medium
    key_name: mykey
    group:
      - web
      - ssh
    instance_tags:
      type: web
      env: production
```

При использовании `local_action` также допускается смешивать простые и составные аргументы:

```
- name: create an ec2 instance
  local_action: ec2 image=ami-8caa1ce4 instance_type=m3.medium key_name=mykey
  args:
    image: ami-8caa1ce4
    instance_type: m3.medium
    key_name: mykey
    group:
      - web
      - ssh
    instance_tags:
      type: web
      env: production
```

Ansible позволяет определять разрешения для файлов, которые используются несколькими модулями, включая `file`, `copy` и `template`. Если вы решите указать восьмеричное значение в составном аргументе, оно должно начинаться с 0 или заключаться в кавычки как строка.

Например, обратите внимание, что аргумент `mode` начинается с 0:

```
- name: copy index.html
  copy:
  src: files/index.html
  dest: /usr/share/nginx/html/index.html
  mode: "0644"
```

Если значение аргумента mode не будет начинаться с 0 или не будет заключено в кавычки, Ansible воспримет это значение как десятичное число и установит разрешения для файла не те, что вы ожидаете. За подробностями обращайтесь по адресу: http://bit.ly/1GASfbl.

Если вы хотите разбить аргумент на несколько строк и не передаете составных аргументов, можете самостоятельно выбрать, в какой форме это сделать. Это дело вкуса. Я обычно предпочитаю словари, но в книге использую оба варианта.

Настройка базы данных

Когда среда Django действует в режиме для разработки, в качестве базы данных она использует SQLite. В этом случае создается файл базы данных, если такового не существует.

Чтобы задействовать систему управления базами данных, такую как Postgres, сначала нужно создать базу данных внутри Postgres, а затем учетную запись пользователя, владеющего базой данных. 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 6.14 показано, как пользоваться этими модулями в сценариях.

Пример 6.14 ❖ Создание базы данных и пользователя

```
- name: create project locale
  locale_gen: name={{ locale }}
  become: True

- name: create a DB user
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"
    become: True
    become_user: postgres

  - name: create the database
    postgresql_db:
      name: "{{ database_name }}"
      owner: "{{ database_user }}"
      encoding: UTF8
      lc_ctype: "{{ locale }}"
      lc_collate: "{{ locale }}"
      template: template0
    become: True
    become_user: postgres
```

Обратите внимание на выражения become: True и become_user: postgres в двух последних задачах. Когда выполняется установка Postgres в Ubuntu, в ее про-

цессе 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 необходимо выполнить команду `become` для пользователя Postgres в сценарии, 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, изменяющиеся от установки к установке (*local_settings.py*).

Мы определили настройки, неизменные для всех установок, в файле *settings.py*, в репозитории проекта. Вы найдете этот файл по адресу: http://bit.ly/2jaw4zf.

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 6.15 приводится шаблон Jinja2, который мы используем.

Пример 6.15 ❖ local_settings.py.j2

```
from __future__ import unicode_literals

SECRET_KEY = "{{ secret_key }}"
NEVERCACHE_KEY = "{{ nevercache_key }}"
ALLOWED_HOSTS = [{% for domain in domains %}"{{ domain }}",{% endfor %}]

DATABASES = {
    "default": {
        # Может завершаться "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 }}",
        # Не используется с sqlite3.
        "PASSWORD": "{{ db_pass }}",
```

[1] За более подробной информацией о шаблонах баз данных обращайтесь к документации Postgres: http://bit.ly/1F5AYpN.

```
        # 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": "",
    }
}

SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTOCOL", "https")

CACHE_MIDDLEWARE_SECONDS = 60

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`, `never-cache_key`, `proj_name` и `db_pass`. Единственная неочевидная вещь – это строка, приведенная в примере 6.16:

Пример 6.16 ❖ Использование цикла for в шаблоне Jinja2

```
ALLOWED_HOSTS = [{% for domain in domains %}"{{ domain }}",{% endfor %}]
```

Если вернуться к определению переменной, можно увидеть, что переменная `domains` определена так:

```
domains:
  - 192.168.33.10.xip.io
  - www.192.168.33.10.xip.io
```

Система Mezzanine будет отвечать только на запросы к серверам, перечисленным в списке в переменной `domains`, в нашем случае http://192.168.33.10.xip.io и http://www.192.168.33.10.xip.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.xip.io", "www.192.168.33.10.xip.io"]
```

Для этого можно использовать цикл `for`, как показано в примере 6.16. Но обратите внимание, что результат получается не совсем тот, которого мы добиваемся, – получающаяся строка содержит завершающую запятую:

```
ALLOWED_HOSTS = ["192.168.33.10.xip.io", "www.192.168.33.10.xip.io",]
```

Однако Python вполне устраивает наличие завершающей запятой в списке, и мы можем оставить все, как есть.

Что такое xip.io?

Вероятно, вы заметили, что используемые доменные имена выглядят немного странно: *192.168.33.10.xip.io* и *www.192.168.33.10.xip.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, мы 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*). 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 *xip.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.xip.io* – это DNS-запись, разрешающаяся в адрес *AAA.BBB.CCC.DDD*. Например, *192.168.33.10.xip.io* разрешается в *192.168.33.10*. Кроме того, *www.192.168.33.10.xip.io* также разрешается в 192.168.33.10.

Мне кажется, *xip.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.

Рассмотрим синтаксис цикла `for` в Jinja2. Чтобы было удобнее, разобьем его на несколько строк:

```
ALLOWED_HOSTS = [
    {% for domain in domains %}
        "{{ domain }}",
    {% endfor %}
]
```

Сгенерированный файл конфигурации, все еще корректный с точки зрения Python, будет выглядеть, как показано ниже:

```
ALLOWED_HOSTS = [
                 "192.168.33.10.xip.io",
                 "www.192.168.33.10.xip.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, то есть нет необходимости явно передавать переменные в шаблон.

Выполнение команд django-manage

Приложения Django используют особый сценарий *manage.py* (http://bit.ly/2iica5a) 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] ***********************************************
failed: [web] => {"cmd": "python manage.py createdb --noinput --nodata", "failed"
: true, "path": "/home/vagrant/mezzanine_example/bin:/usr/local/sbin:/usr/local/b
in:/usr/sbin: /usr/bin:/sbin:/bin:/usr/games:/usr/local/games", "state": "absent"
, "syspath": ["", "/usr/lib/python2.7", "/usr/lib/python2.7/plat-x86_64-linux-gnu
", "/usr/lib/python2.7/lib-tk", "/usr/lib/python2.7/lib-old", "/usr/lib/python2.7
/lib-dynload", "/usr/local/lib/python2.7/dist-packages", "/usr/lib/python2.7/dist
-packages"]}
msg:
:stderr: CommandError: Database already created, you probably want the syncdb or
migrate command
```

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 }}"
  with_items:
    - 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 (http://bit.ly/2hYWztG), содержащий доменное имя сайта (в нашем случае *192.168.33.10.xip.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 мне требуется выполнить свой код на Python, я пишу свой модуль для Ansible. Однако, насколько я знаю, Ansible не позволяет запускать модули в контексте virtualenv. Поэтому данный вариант исключается.

Вместо этого я воспользовался модулем `script`. Он копируется поверх нестандартного сценария и выполняет его. Я написал два сценария: один – для создания записи Site, другой – для создания учетной записи пользователя с правами администратора.

Вы можете передавать аргументы модулю `script` через командную строку и анализировать их. Но я решил передать аргументы через переменные окружения. Мне не хотелось передавать пароли через командую строку (их можно увидеть в списке процессов, который выводит команда `ps`), а кроме того, переменные среды легче проанализировать в сценариях, чем аргументы командной строки.

Ansible позволяет устанавливать переменные среды посредством выражения `environment`, которое принимает словарь с именами и значениями переменных окружения. 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 6.17 показан пример запуска двух сценариев.

Пример 6.17 ❖ 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: "{{ 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 }}"
```

Сами сценарии приводятся в примерах 6.18 и 6.19. Я поместил их в каталог *scripts*.

Пример 6.18 ❖ scripts/setsite.py

```
#!/usr/bin/env python
# Сценарий настраивает домен сайта
# Предполагается наличие трех переменных окружения
#
# WEBSITE_DOMAIN: домен сайта (например, www.example.com)
# PROJECT_DIR: корневой каталог проекта
# PROJECT_APP: имя проекта приложения
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 6.19 ❖ scripts/setadmin.py

```
#!/usr/bin/env python
# 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
114 | Chapter 6: Deploying Mezzanine with Ansible
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()
```

Настройка конфигурационных файлов служб

Далее настроим конфигурационный файл для Gunicorn (сервера приложений), Nginx (веб-сервера) и Supervisor (диспетчер процессов), как показано в примере 6.20. Шаблон файла конфигурации для Gunicorn приводится в примере 6.22, а шаблон файла конфигурации для Supervisor – в примере 6.23

Пример 6.20 ❖ Настройка файлов конфигурации

```
- name: set the gunicorn config file
  template:
    src: templates/gunicorn.conf.py.j2
    dest: "{{ proj_path }}/gunicorn.conf.py"

- name: set the supervisor config file
  template:
    src: templates/supervisor.conf.j2
    dest: /etc/supervisor/conf.d/mezzanine.conf
  become: True
  notify: restart supervisor

- name: set the nginx config file
  template:
    src: templates/nginx.conf.j2
    dest: /etc/nginx/sites-available/mezzanine.conf
  notify: restart nginx
  become: True
```

Во всех трех случаях файлы конфигурации генерируются из шаблонов. Процессы Supervisor и Nginx запускаются с привилегиями пользователя root (хотя они тут же и понижают свои привилегии), поэтому нужно выполнить команду `sudo`, чтобы получить право на доступ к файлам конфигурации.

Если файл конфигурации Supervisor изменится, Ansible запустит обработчик `restart supervisor`. Если изменится файл конфигурации Nginx, Ansible запустит обработчик `restart nginx`, как показано в примере 621.

Пример 6.21 ❖ Обработчики

```
handlers:
  - name: restart supervisor
    supervisorctl: name=gunicorn_mezzanine state=restarted
    sudo: True

  - name: restart nginx
    service: name=nginx state=restarted
    sudo: True
```

Пример 6.22 ❖ templates/gunicorn.conf.py.j2

```
from __future__ import unicode_literals
import multiprocessing

bind = "127.0.0.1:{{ gunicorn_port }}"
workers = multiprocessing.cpu_count() * 2 + 1
loglevel = "error"
proc_name = "{{ proj_name }}"
```

Пример 6.23 ❖ 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 6.24 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`, выполняется включение поддержки TLS. В шаблоне там и сям можно увидеть операторы `if`, например:

```
{% if tls_enabled %}
...
{% endif %}
```

В нем также используется 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.xip.io
  - www.192.168.33.10.xip.io
```

После применения шаблона получаем следующее:

```
server_name 192.168.33.10.xip.io, www.192.168.33.10.xip.io;
```

Пример 6.24 ❖ 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(", ") }};
    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_cache shared:SSL:10m;
    ssl_session_timeout 10m;
    # элемент ssl_ciphers слишком длинный, чтобы показать его целиком в книге
    # См. https://github.com/ansiblebook/ansiblebook
    #     ch06/playbooks/templates/nginx.conf.j2
    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;
    }
}
```

Активация конфигурации Nginx

По соглашениям, принятым для файлов конфигурации Nginx, они должны помещаться в каталог */etc/nginx/sites-available* и активироваться символической ссылкой в каталог */etc/nginx/sites-enabled*.

Сценарии Fabric Mezzanine просто копируют файл конфигурации непосредственно в *sites-enabled*. Но я собираюсь отклониться от способа, принятого в Mezzanine, и использовать модуль `file` для создания символической ссылки. Также нужно удалить файл конфигурации, который создает пакет Nginx в */etc/nginx/sites-enabled/default*.

Как показано в примере 6.25, я использовал 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; и для настройки свойств, таких как разрешения и принадлежность.

Пример 6.25 ❖ Активация конфигурации Nginx

```
- name: enable the nginx config file
  file:
    src: /etc/nginx/sites-available/mezzanine.conf
    dest: /etc/nginx/sites-enabled/mezzanine.conf
    state: link
  become: True

- name: remove the default nginx config file
  file: path=/etc/nginx/sites-enabled/default state=absent
  notify: restart nginx
  become: True
```

Установка сертификатов TLS

В нашем сценарии определяется переменная `tls_enabled`. Если она получает значение `true`, сценарий установит сертификаты TLS. В нашем примере мы ис-

пользуем самоподписанный сертификат, поэтому сценарий создаст сертификат, если он не существует.

В условиях промышленной эксплуатации необходимо скопировать существующий сертификат TLS, который вы получили от центра сертификации.

В примере 6.26 представлены две задачи, которые вовлечены в процесс настройки сертификатов TLS. Модуль `file` используется, чтобы при необходимости создать каталог для сертификатов TLS.

Пример 6.26 ❖ Установка сертификатов TLS

```
- name: ensure config path exists
  file: path={{ conf_path }} state=directory
  sudo: True
  when: tls_enabled

- name: create tls certificates
  command: >
    openssl req -new -x509 -nodes -out {{ proj_name }}.crt
    -keyout {{ proj_name }}.key -subj '/CN={{ domains[0] }}' -days 3650
    chdir={{ conf_path }}
    creates={{ conf_path }}/{{ proj_name }}.crt
  sudo: True
  when: tls_enabled
  notify: restart nginx
```

Обратите внимание, что обе задачи содержат выражение:

```
when: tls_enabled
```

Если значение `tls_enabled` равно `false`, 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 возможность свертки строк в YAML (подробности см. в разделе «Объединение строк» в главе 2), чтобы разбить команду на несколько строк.

Две строки в конце команды содержат дополнительные параметры, передаваемые модулю. Они не передаются в командную строку.

```
chdir={{ conf_path }}
creates={{ conf_path }}/{{ proj_name }}.crt
```

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`, Mezzanine извлечет твиты из настроенных учетных записей и поместит их на домашнюю страницу. Сцена-

рии 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 мой взгляд, более изящно. В примере 6.27 представлена задача, которая устанавливает задание cron.

Пример 6.27 ❖ 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, командой crontab -l можно убедиться, что требуемое задание присутствует в общем списке. Вот как все это выглядит у меня:

```
#Ansible: poll twitter
*/5 * * * * /home/vagrant/.virtualenvs/mezzanine_example/bin/python \
/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. Следующая задача:

```
- name: remove cron job
  cron: name="poll twitter" state=absent
```

вызовет модуль cron, который отыщет строку комментария с указанным именем и удалит задание.

Сценарий целиком

В примере 6.28 представлен полный сценарий во всем своем великолепии.

Пример 6.28 ❖ 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.xip.io
  domains:
    - 192.168.33.10.xip.io
    - www.192.168.33.10.xip.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/python"
  database_name: "{{ proj_name }}"
  database_user: "{{ proj_name }}"
  database_host: localhost
  database_port: 5432
  gunicorn_procname: gunicorn_mezzanine
  num_workers: "multiprocessing.cpu_count() * 2 + 1"
vars_files:
  - secrets.yml
tasks:
  - name: install apt packages
    apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
    become: True
    with_items:
      - git
      - libjpeg-dev
      - libpq-dev
      - memcached
      - nginx
      - postgresql
      - python-dev
      - python-pip
      - python-psycopg2
      - python-setuptools
      - python-virtualenv
      - supervisor
  - name: create project path
    file: path={{ proj_path }} state=directory
  - name: create a logs directory
    file:
      path: "{{ ansible_env.HOME }}/logs"
      state: directory
  - name: check out the repository on the host
    git: repo={{ repo_url }} dest={{ proj_path }} accept_hostkey=yes
  - name: install Python requirements globally via pip
    pip: name={{ item }} state=latest
    with_items:
      - pip
      - virtualenv
      - virtualenvwrapper
    become: True
```

```
- name: create project locale
  locale_gen: name={{ locale }}
  become: True
- name: create a DB user
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"
  become: True
  become_user: postgres
- name: create the database
  postgresql_db:
    name: "{{ database_name }}"
    owner: "{{ database_user }}"
    encoding: UTF8
    lc_ctype: "{{ locale }}"
    lc_collate: "{{ locale }}"
    template: template0
  become: True
  become_user: postgres
- name: ensure config path exists
  file: path={{ conf_path }} state=directory
  become: True
- name: create tls certificates
  command: >
    openssl req -new -x509 -nodes -out {{ proj_name }}.crt
    -keyout {{ proj_name }}.key -subj '/CN={{ domains[0] }}' -days 3650
    chdir={{ conf_path }}
    creates={{ conf_path }}/{{ proj_name }}.crt
  become: True
  when: tls_enabled
  notify: restart nginx
- name: remove the default nginx config file
  file: path=/etc/nginx/sites-enabled/default state=absent
  notify: restart nginx
  become: True
- name: set the nginx config file
  template:
    src=templates/nginx.conf.j2
    dest=/etc/nginx/sites-available/mezzanine.conf
  notify: restart nginx
  become: True
- name: enable the nginx config file
  file:
    src: /etc/nginx/sites-available/mezzanine.conf
    dest: /etc/nginx/sites-enabled/mezzanine.conf
    state: link
  become: True
  notify: restart nginx
- name: set the supervisor config file
  template:
    src=templates/supervisor.conf.j2
```

```
      dest=/etc/supervisor/conf.d/mezzanine.conf
    become: True
    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
  - name: generate the settings file
    template:
      src=templates/local_settings.py.j2
      dest={{ settings_path }}/local_settings.py
  - name: install requirements.txt
    pip: requirements={{ proj_path }}/{{ reqs_path }} virtualenv={{ venv_path }}
  - name: install required python packages
    pip: name={{ item }} virtualenv={{ venv_path }}
    with_items:
      - gunicorn
      - setproctitle
      - psycopg2
      - django-compressor
      - python-memcached
  - 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 }}"
      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
```

```
    supervisorctl: "name={{ gunicorn_procname }} state=restarted"
    become: True
  - name: restart nginx
    service: name=nginx state=restarted
    become: True
```

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 6.29, настраивает машину Vagrant с этим IP-адресом.

Пример 6.29 ❖ Vagrantfile

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.network "private_network", ip: "192.168.33.10"
end
```

Развертывание Mezzanine на машине Vagrant выполняется командой

```
$ ansible-playbook mezzanine.yml
```

После этого можно увидеть готовый сайт Mezzanine по любому из перечисленных ниже адресов:

- *http://192.168.33.10.xip.io*;
- *https://192.168.33.10.xip.io*;
- *http://www.192.168.33.10.xip.io*;
- *https://www.192.168.33.10.xip.io*.

Устранение проблем

При попытке выполнить сценарий на своей 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 своего файла *~/.ssh/known_hosts*. Подробности смотрите во врезке «Неправильный ключ хоста может повлечь проблемы даже при отключенной проверке ключа» в приложении А.

Недоступен хост с адресом 192.168.33.10.xip.io

Некоторые маршрутизаторы WiFi имеют встроенный сервер DNS, который не распознает имя хоста *192.168.33.10.xip.io*. Проверить это можно следующей командой:

```
dig +short 192.168.33.10.xip.io
```

Она должна вывести:

```
192.168.33.10
```

Если выводится пустая строка, значит, ваш сервер DNS не распознает имена хостов *xip.io*. В этом случае добавьте в свой файл */etc/hosts* следующую строку:

```
192.168.33.10 192.168.33.10.xip.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.xip.io
  - www.192.168.33.10.xip.io
```

Установка Mezzanine на нескольких машинах

Мы развернули Mezzanine на одной-единственной машине. Однако нередко база данных устанавливается на отдельном хосте. В главе 7 мы рассмотрим сценарий, который устанавливает базу данных и веб-службы на разные хосты.

Теперь вы знаете, как осуществляется развертывание обычного приложения с поддержкой Mezzanine. В следующей главе мы рассмотрим другие возможности Ansible, не использованные в нашем примере.

Глава 7

Роли: масштабирование сценария

Одной из особенностей Ansible, вызывающих у меня восхищение, является вертикальное масштабирование – вверх и вниз. Здесь я имею в виду не количество хостов, а сложность автоматизируемых задач.

Масштабирование вниз обусловлено простотой разработки отдельных задач. Масштабирование вверх упрощается благодаря механизмам деления сложных задач на небольшие части.

Роли в Ansible – это основной механизм деления сценария на отдельные файлы. Они упрощают написание 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`.

Базовая структура роли

Роль в Ansible имеет имя, например `database`. Файлы, связанные с ролью `database`, хранятся в каталоге *roles/database*, содержащем следующие файлы и каталоги:

- *roles/database/tasks/main.yml* – задачи;
- *roles/database/files/* – файлы, выгружаемые на хосты;
- *roles/database/templates/* – файлы шаблонов Jinja2;
- *roles/database/handlers/main.yml* – обработчики;
- *roles/database/vars/main.yml* – переменные, которые не должны переопределяться;
- *roles/database/defaults/main.yml* – переменные, которые могут переопределяться;
- *roles/database/meta/main.yml* – информация о зависимостях данной роли.

Все файлы являются необязательными. Если роль не имеет обработчиков, тогда нет необходимости создавать пустой файл *handlers/main.yml*.

Где Ansible будет искать мои роли?

Ansible обращается к ролям, хранящимся в подкаталоге *roles*, в каталоге со сценарием. Системные роли можно помещать в */etc/ansible/roles*. Местоположение системных ролей можно изменить, переопределив параметр `roles_path` в секции `defaults` файла *ansible.cfg*, как показано в примере 7.1.

Пример 7.1 ❖ ansible.cfg: изменение пути к каталогу с системными ролями

```
[defaults]
roles_path = ~/ansible_roles
```

То же самое можно сделать, изменив переменную окружения `ANSIBLE_ROLES_PATH`, как показано в приложении B.

Примеры ролей: database и mezzanine

Возьмем за основу наш сценарий развертывания Mezzanine и изменим его, реализовав роли. Можно было бы создать единственную роль с именем `mezzanine`, но я дополнительно выделю развертывание базы данных Postgres в отдельную роль с именем `database`. Это упростит развертывание базы данных на хосте, отличном от хоста для приложения Mezzanine.

Использование ролей в сценариях

Прежде чем погрузиться в детали определения ролей, посмотрим, как назначать роли к хостам в сценариях. В примере 7.2 представлен наш сценарий для развертывания Mezzanine на единственном хосте после добавления ролей `database` и `mezzanine`.

Пример 7.2 ❖ 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
      live_hostname: 192.168.33.10.xip.io
      domains:
        - 192.168.33.10.xip.io
        - www.192.168.33.10.xip.io
```

При использовании ролей в сценарии должна иметься секция `roles` со списком ролей. В нашем примере список содержит две роли – `database` 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 были определены для роли (в *vars/main.yml* или *defaults/main.yml*), их значения будут переопределены переданными здесь.

Если ролям не передаются никакие переменные, можно просто определить имена ролей:

```
roles:
  - database
  - mezzanine
```

После определения ролей database и mezzanine написание сценария для развертывания веб-приложения и базы данных на нескольких хостах становится намного проще. В примере 7.3 приводится сценарий развертывания базы данных на хосте db и веб-службы на хосте *web*. Обратите внимание, что этот сценарий содержит две отдельные операции.

Пример 7.3 ❖ mezzanine-across-hosts.yml

```
- name: deploy postgres on vagrant
  hosts: db
  vars_files:
    - secrets.yml
  roles:
    - role: database
      database_name: "{{ mezzanine_proj_name }}"
      database_user: "{{ mezzanine_proj_name }}"

- name: deploy mezzanine on vagrant
  hosts: web
  vars_files:
    - secrets.yml
  roles:
    - role: mezzanine
      database_host: "{{ hostvars.db.ansible_eth1.ipv4.address }}"
      live_hostname: 192.168.33.10.xip.io
      domains:
        - 192.168.33.10.xip.io
        - www.192.168.33.10.xip.io
```

Предварительные и заключительные задачи

Иногда требуется запускать некоторые задачи до или после запуска ролей. Допустим, необходимо обновить кэш диспетчера apt перед развертыванием Mezzanine, а после развертывания отправить уведомление в канал Slack.

Ansible позволяет определить списки задач для выполнения до и после вызова роли. Эти задачи необходимо определить в секциях pre_tasks и post_tasks соответственно. В примере 7.4 представлен один из вариантов.

Пример 7.4 ❖ Списки задач для выполнения до и после вызова роли

```
- name: deploy mezzanine on vagrant
  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_eth1.ipv4.address }}"
      live_hostname: 192.168.33.10.xip.io
      domains:
        - 192.168.33.10.xip.io
        - www.192.168.33.10.xip.io
  post_tasks:
    - name: notify Slack that the servers have been updated
      local_action: >
        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

Задачей нашей роли `database` являются установка Postgres и создание необходимых базы данных и пользователя.

Все аспекты роли `database` определяются в следующих файлах:

- *roles/database/tasks/main.yml*;
- *roles/database/defaults/main.yml*;
- *roles/database/handlers/main.yml*;
- *roles/database/files/pg_hba.conf*;
- *roles/database/files/postgresql.conf*.

Эта роль включает два особых файла конфигурации Postgres.

postgresql.conf

Изменяет заданный по умолчанию параметр `listen_addresses`, чтобы Postgres принимал соединения на любом сетевом интерфейсе. По умолчанию Postgres принимает соединения только от `localhost`, что нам не подходит для случая, когда 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 *ch08*, на странице https://github.com/ansiblebook/ansiblebook.

В примере 7.5 показаны задачи, вовлеченные в процесс развертывания Postgres.

Пример 7.5 ❖ roles/database/tasks/main.yml

```
- name: install apt packages
  apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
  become: True
  with_items:
    - libpq-dev
    - postgresql
    - python-psycopg2

- name: copy configuration file
  copy: >
    src=postgresql.conf dest=/etc/postgresql/9.3/main/postgresql.conf
    owner=postgres group=postgres mode=0644
  become: True
  notify: restart postgres

- name: copy client authentication configuration file
  copy: >
    src=pg_hba.conf dest=/etc/postgresql/9.3/main/pg_hba.conf
    owner=postgres group=postgres mode=0640
  become: True
  notify: restart postgres

- name: create project locale
  locale_gen: name={{ locale }}
  become: True

- name: create a user
  postgresql_user:
    name: "{{ database_user }}"
    password: "{{ db_pass }}"
  become: True
  become_user: postgres

- name: create the database
  postgresql_db:
    name: "{{ database_name }}"
    owner: "{{ database_user }}"
    encoding: UTF8
    lc_ctype: "{{ locale }}"
    lc_collate: "{{ locale }}"
    template: template0
  become: True
  become_user: postgres
```

В примере 7.6 представлен файл обработчиков.

Пример 7.6 ❖ roles/database/handlers/main.yml

```
- name: restart postgres
  service: name=postgresql state=restarted
  become: True
```

Единственной переменной по умолчанию, которую мы определим, является порт базы данных, как показано в примере 7.7.

Пример 7.7 ❖ roles/database/defaults/main.yml

```
database_port: 5432
```

Обратите внимание, что в списке задач упоминается несколько переменных, которые не определены в роли:

- `database_name`;
- `database_user`;
- `db_pass`;
- `locale`.

Переменные `database_name` и `database_user` передаются в вызов роли в примерах 7.2 и 7.3. 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 быть использована разными ролями или сценариями, поэтому я определю 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. Такие переменные можно было переопределить, только передав их в вызов роли в виде аргументов.

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, то есть 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

Задачей роли `mezzanine` является установка Mezzanine. Сюда входят установка Nginx в качестве обратного прокси и Supervisor в качестве монитора процессов.

Ниже перечислены файлы, реализующие роль:

- *roles/mezzanine/defaults/main.yml*;
- *roles/mezzanine/handlers/main.yml*;
- *roles/mezzanine/tasks/django.yml*;

- *roles/mezzanine/tasks/main.yml*;
- *roles/mezzanine/tasks/nginx.yml*;
- *roles/mezzanine/templates/gunicorn.conf.py.j2*;
- *roles/mezzanine/templates/local_settings.py.filters.j2*;
- *roles/mezzanine/templates/local_settings.py.j2*;
- *roles/mezzanine/templates/nginx.conf.j2*;
- *roles/mezzanine/templates/supervisor.conf.j2*;
- *roles/mezzanine/vars/main.yml*.

В примере 7.8 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. Такое поведение может приводить к нежелательным последствиям, если случайно использовать одно и то же имя переменной в двух разных ролях.

Пример 7.8 ❖ roles/mezzanine/vars/main.yml

```
# файл vars для mezzanine
mezzanine_user: "{{ ansible_user }}"
mezzanine_venv_home: "{{ ansible_env.HOME }}"
mezzanine_venv_path: "{{ mezzanine_venv_home }}/{{ mezzanine_proj_name }}"
mezzanine_repo_url: git@github.com:lorin/mezzanine-example.git
mezzanine_proj_dirname: project
mezzanine_proj_path: "{{ mezzanine_venv_path }}/{{ mezzanine_proj_dirname }}"
mezzanine_reqs_path: requirements.txt
mezzanine_conf_path: /etc/nginx/conf
mezzanine_python: "{{ mezzanine_venv_path }}/bin/python"
mezzanine_manage: "{{ mezzanine_python }} {{ mezzanine_proj_path }}/manage.py"
mezzanine_gunicorn_port: 8000
```

В примере 7.9 показаны переменные по умолчанию для роли `mezzanine`. В данном случае определена лишь одна переменная. Объявляя переменные по умолчанию, я обычно не использую префикс с именем роли, потому что могу целенаправленно изменить их где-то еще.

Пример 7.9 ❖ roles/mezzanine/defaults/main.yml

```
tls_enabled: True
```

Поскольку список задач получился достаточно большим, я решил разбить его на несколько файлов. В примере 7.10 показан файл с задачей верхнего уровня для роли `mezzanine`. Она устанавливает apt-пакеты, а затем использует инструкции `include`, чтобы запустить задачи в двух других файлах, находящихся в том же каталоге (см. примеры 7.11 и 7.12).

Пример 7.10 ❖ roles/mezzanine/tasks/main.yml

```
- name: install apt packages
  apt: pkg={{ item }} update_cache=yes cache_valid_time=3600
```

```
  become: True
  with_items:
    - git
    - libjpeg-dev
    - libpq-dev
    - memcached
    - nginx
    - python-dev
    - python-pip
    - python-psycopg2
    - python-setuptools
    - python-virtualenv
    - supervisor

- include: django.yml

- include: nginx.yml
```

Пример 7.11 ❖ roles/mezzanine/tasks/django.yml

```
- name: create a logs directory
  file: path="{{ ansible_env.HOME }}/logs" state=directory

- name: check out the repository on the host
  git:
    repo: "{{ mezzanine_repo_url }}"
    dest: "{{ mezzanine_proj_path }}"
    accept_hostkey: yes

- name: install Python requirements globally via pip
  pip: name={{ item }} state=latest
  with_items:
    - pip
    - virtualenv
    - virtualenvwrapper

- name: install required python packages
  pip: name={{ item }} virtualenv={{ mezzanine_venv_path }}
  with_items:
    - gunicorn
    - setproctitle
    - psycopg2
    - django-compressor
    - python-memcached

- name: install requirements.txt
  pip: >
    requirements={{ mezzanine_proj_path }}/{{ mezzanine_reqs_path }}
    virtualenv={{ mezzanine_venv_path }}

- name: generate the settings file
  template: src=local_settings.py.j2 dest={{ mezzanine_proj_path }}/local_settings.py

- name: apply migrations to create the 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: scripts/setsite.py
  environment:
    PATH: "{{ mezzanine_venv_path }}/bin"
    PROJECT_DIR: "{{ mezzanine_proj_path }}"
    PROJECT_APP: "{{ mezzanine_proj_app }}"
    WEBSITE_DOMAIN: "{{ live_hostname }}"

- name: set the admin password
  script: scripts/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=gunicorn.conf.py.j2 dest={{ mezzanine_proj_path }}/gunicorn.conf.py

- name: set the supervisor config file
  template: src=supervisor.conf.j2 dest=/etc/supervisor/conf.d/mezzanine.conf
  become: True
  notify: restart supervisor

- name: ensure config path exists
  file: path={{ mezzanine_conf_path }} state=directory
  become: True
  when: tls_enabled

- name: install poll twitter cron job
  cron: >
    name="poll twitter" minute="*/5" user={{ mezzanine_user }}
    job="{{ mezzanine_manage }} poll_twitter"
```

Пример 7.12 ❖ roles/mezzanine/tasks/nginx.yml

```
- name: set the nginx config file
  template: src=nginx.conf.j2 dest=/etc/nginx/sites-available/mezzanine.conf
  notify: restart nginx
  become: True

- name: enable the nginx config file
  file:
    src: /etc/nginx/sites-available/mezzanine.conf
    dest: /etc/nginx/sites-enabled/mezzanine.conf
    state: link
  notify: restart nginx
```

```
  become: True

- name: remove the default nginx config file
  file: path=/etc/nginx/sites-enabled/default state=absent
  notify: restart nginx
  become: True

- name: create tls certificates
  command: >
    openssl req -new -x509 -nodes -out {{ mezzanine_proj_name }}.crt
    -keyout {{ mezzanine_proj_name }}.key -subj '/CN={{ domains[0] }}' -days 3650
    chdir={{ mezzanine_conf_path }}
    creates={{ mezzanine_conf_path }}/{{ mezzanine_proj_name }}.crt
  become: True
  when: tls_enabled
  notify: restart nginx
```

Есть существенная разница между задачами, объявленными в роли, и задачами, объявленными в сценарии как обычно. Она касается использования модулей `copy` и `template`.

Когда модуль `copy` вызывается в задаче для роли, Ansible сначала проверит наличие копируемых файлов в каталоге *rolename/files/*. Аналогично, когда модуль `template` вызывается в задаче для роли, Ansible сначала проверит наличие шаблонов в каталоге *rolename/templates*.

Это значит, что задача, которая раньше была определена в сценарии так:

```
- name: set the nginx config file
  template: src=templates/nginx.conf.j2 \
  dest=/etc/nginx/sites-available/mezzanine.conf
```

теперь, когда она вызывается в роли, должна выглядеть так (обратите внимание на изменившийся параметр `src`):

```
- name: set the nginx config file
  template: src=nginx.conf.j2 dest=/etc/nginx/sites-available/mezzanine.conf
  notify: restart nginx
```

В примере 7.13 приводится файл обработчиков.

Пример 7.13 ❖ roles/mezzanine/handlers/main.yml

```
- name: restart supervisor
  supervisorctl: name=gunicorn_mezzanine state=restarted
  become: True

- name: restart nginx
  service: name=nginx state=restarted
  become: True
```

Я не буду приводить здесь файлы шаблонов, поскольку они остались теми же, что и в прошлой главе, хотя имена некоторых переменных изменились. За дополнительной информацией обращайтесь к примерам кода, прилагаемым к книге.

Создание файлов и директорий 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 (подробнее об этом чуть позже). Но с его помощью также можно сгенерировать начальный набор файлов и каталогов для роли:

```
$ ansible-galaxy init -p playbooks/roles web
```

Параметр `-p` сообщает местоположение каталога ролей. Если его опустить, `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`.

---

[1] NTP (Network Time Protocol) – протокол сетевого времени, используется для синхронизации времени.

Ansible поддерживает возможность определения *зависимостей между ролями* для подобных случаев. Определяя роль, вы можете указать, что она зависит от одной или нескольких других ролей, а Ansible позаботится о том, чтобы зависимые роли выполнялись первыми.

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 7.14.

Пример 7.14 ❖ 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 7.15.

Пример 7.15 ❖ roles/django/meta/main.yml

```
dependencies:
  - { role: web }
  - { role: memcached }
```

За более подробной информацией о зависимостях между ролями в Ansible обращайтесь к официальной документации: http://bit.ly/1F6tH9a.

Ansible Galaxy

Если вам понадобится установить на ваши хосты программное обеспечение с открытым исходным кодом, 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.

Веб-интерфейс

Вы можете исследовать доступные роли 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 ресурса Ansible Galaxy.

Установка роли

Допустим, вы захотели установить роль `ntp`, написанную пользователем GitHub с именем *bennojoy*. Эта роль настраивает хост для синхронизации часов с сервером NTP.

Установите роль командой `install`.

```
$ ansible-galaxy install -p ./roles bennojoy.ntp
```

Программа `ansible-galaxy` по умолчанию устанавливает роли в системный каталог (см. врезку «Где Ansible будет искать мои роли?» в начале главы), который в предыдущем примере мы заменили своим каталогом, передав параметр `-p`.

Результат должен выглядеть так:

```
downloading role 'ntp', owned by bennojoy
no version specified, installing master
- downloading role from https://github.com/bennojoy/ntp/archive/master.tar.gz
- extracting bennojoy.ntp to ./roles/bennojoy.ntp
write_galaxy_install_info!
bennojoy.ntp was installed successfully
```

Инструмент `ansible-galaxy` установит файлы роли в *roles/bennojoy.ntp*.

Ansible поместит некоторые метаданные об установленной роли в файл *./roles/bennojoy.ntp/meta/.galaxy_install_info*. На моей машине этот файл содержит:

```
{install_date: 'Sat Oct 4 20:12:58 2014', version: master}
```


Роль *bennojoy.ntp* не имеет конкретного номера версии, поэтому версия определена просто как `master` (основная). Некоторые роли имеют определенную версию, например 1.2.

Вывод списка установленных ролей

Получить список установленных ролей можно следующей командой:

```
$ ansible-galaxy list
```

Результат должен выглядеть так:

```
bennojoy.ntp, master
```

Удаление роли

Удалить роль можно командой `remove`:

```
$ ansible-galaxy remove bennojoy.ntp
```

Добавление собственной роли

Чтобы узнать, как поделиться своей ролью с другими членами сообщества, обращайтесь к разделу «How To Share Roles You've Written» по адресу: https://galaxy.ansible.com/intro. Поскольку роли располагаются в репозитории GitHub, вам потребуется создать свою учетную запись.

Теперь вы знаете, как использовать роли, создавать собственные роли и загружать роли, написанные другими. Роли – мощный инструмент организации сценариев. Я пользуюсь ими все время и настоятельно рекомендую вам.

Глава 8

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.

Команды changed_when и failed_when

В главе 6 мы предпочли отказаться от команды `manage.py createdb`, представленной в примере 8.1, потому что она не является идемпотентной.

Пример 8.1 ❖ Вызов команды createdb из manage.py

```
- name: initialize the database
  django_manage:
    command: createdb --noinput --nodata
    app_path: "{{ proj_path }}"
    virtualenv: "{{ venv_path }}"
```

Мы обошли эту проблему запуском нескольких идемпотентных команд `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`, влияющие на то, как Ansible обнаруживает изменение состояния или ошибку.

Сначала нужно разобраться, что выводит команда в первый раз, а что во второй.

Как мы уже делали это в главе 4, 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 8.2.

Пример 8.2 ❖ 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 8.3 показан вывод сценария после попытки запустить его второй раз.

Пример 8.3 ❖ Вывод сценария в случае, если база данных уже создана

```
TASK: [debug var=result] ******************************************************
ok: [default] => {
    "result": {
        "cmd": "python manage.py createdb --noinput --nodata",
        "failed": false,
        "failed_when_result": false,
        "invocation": {
            "module_args": '',
            "module_name": "django_manage"
        },
        "msg": "\n:stderr: CommandError: Database already created, you probably
want the syncdb or migrate command\n",
        "path":
"/home/vagrant/mezzanine_example/bin:/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games",
        "state": "absent",
        "syspath": [
            ``,
            "/usr/lib/python2.7",
            "/usr/lib/python2.7/plat-x86_64-linux-gnu",
            "/usr/lib/python2.7/lib-tk",
            "/usr/lib/python2.7/lib-old",
            "/usr/lib/python2.7/lib-dynload",
            "/usr/local/lib/python2.7/dist-packages",
            "/usr/lib/python2.7/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 default --become --become-user postgres -m postgresql_db -a \
"name=mezzanine_example state=absent"
```

Если теперь запустить сценарий, он выведет строки, показанные в примере 8.4.

Пример 8.4 ❖ Вывод сценария при первом запуске

```
ASK: [debug var=result] ***************************************************
ok: [default] => {
    "result": {
        "app_path": "/home/vagrant/mezzanine_example/project",
        "changed": false,
        "cmd": "python manage.py createdb --noinput --nodata",
        "failed": false,
        "failed_when_result": false,
        "invocation": {
            "module_args": '',
            "module_name": "django_manage"
        },
        "out": "Creating tables ...\nCreating table auth_permission\nCreating
table auth_group_permissions\nCreating table auth_group\nCreating table
auth_user_groups\nCreating table auth_user_user_permissions\nCreating table
auth_user\nCreating table django_content_type\nCreating table
django_redirect\nCreating table django_session\nCreating table
django_site\nCreating table conf_setting\nCreating table
core_sitepermission_sites\nCreating table core_sitepermission\nCreating table
generic_threadedcomment\nCreating table generic_keyword\nCreating table
generic_assignedkeyword\nCreating table generic_rating\nCreating table
blog_blogpost_related_posts\nCreating table blog_blogpost_categories\nCreating
table blog_blogpost\nCreating table blog_blogcategory\nCreating table
forms_form\nCreating table forms_field\nCreating table forms_formentry\nCreating
table forms_fieldentry\nCreating table pages_page\nCreating table
pages_richtextpage\nCreating table pages_link\nCreating table
galleries_gallery\nCreating table galleries_galleryimage\nCreating table
twitter_query\nCreating table twitter_tweet\nCreating table
south_migrationhistory\nCreating table django_admin_log\nCreating table
django_comments\nCreating table django_comment_flags\n\nCreating default site
record: vagrant-ubuntu-trusty-64 ... \n\nInstalled 2 object(s) from 1
fixture(s)\nInstalling custom SQL ...\nInstalling indexes ...\nInstalled 0
object(s) from 0 fixture(s)\n\nFaking initial migrations ...\n\n",
        "pythonpath": null,
        "settings": null,
        "virtualenv": "/home/vagrant/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 8.5.

Пример 8.5 ❖ 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 8.3. Вместо нее объявлена переменная `msg`. Это означает, что, запустив сценарий во второй раз, он выведет следующую (не особенно информативную) ошибку:

```
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 8.6.

Пример 8.6 ❖ 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, а также для 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 (*http://bit.ly/1FvOGzI*). Кроме того, Ansible добавляет свои фильтры, расширяя возможности фильтров Jinja2 (http://bit.ly/1FvOIrj).

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, тогда 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 8.17 показано, как этого добиться, передав фильтр `failed` в аргументе выражению `failed_when`.

Пример 8.7 ❖ 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. 8.1 перечислены фильтры, которые можно использовать для проверки статуса зарегистрированных переменных.

Таблица 8.1. 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. 8.2 перечислены фильтры для работы с переменными, содержащими пути к файлам в файловой системе управляющей машины.

Рассмотрим следующий фрагмент сценария:

```
vars:
  homepage: /usr/share/nginx/html/index.html
tasks:
```

```
  - name: copy home page
    copy: src=files/index.html dest={{ homepage }}
```

Таблица 8.2. Фильтры путей к файлам

Имя	Имя пути
`basename`	Базовое имя файла
`dirname`	Путь к файлу (каталог)
`expanduser`	Путь к файлу со знаком ~, заменяющим домашний каталог
`realpath`	Канонический путь к файлу, разрешает символические ссылки

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 *index.html* файла, выделив его из полного пути, что позволит записать сценарий неповторения имени файла[1]:

```
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 для Mezzanine мы создали файл *local_settings.py* из шаблона, содержащего строку, показанную в примере 8.8.

Пример 8.8 ❖ 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:

```
ALLOWED_HOSTS = [{{ domains|join(", ") }}]
```

Однако в получившемся результате имена хостов не будут заключены в кавычки, как показано в примере 8.9.

Пример 8.9 ❖ Имена хостов лишились кавычек

```
ALLOWED_HOSTS = [www.example.com, example.com]
```

Если бы у нас имелся фильтр (см. пример 8.10), заключающий строки в кавычки, тогда шаблон сгенерировал бы строку, как показано в примере 8.8.

---

[1] Спасибо Джону Джарвису (John Jarvis) за эту подсказку.

Пример 8.10 ❖ Использование фильтра для заключения строк в кавычки

```
ALLOWED_HOSTS = [{{ domains|surround_by_quote|join(", ") }}]
```

К сожалению, готового фильтра `surround_by_quote` не существует. Но мы можем написать его сами. На самом деле Хэнфи Сан (Hanfei Sun) на Stack Overflow уже раскрыл этот вопрос (https://stackoverflow.com/questions/15514365/).

Ansible ищет нестандартные фильтры в каталоге *filter_plugins*, находящемся в одном каталоге со сценариями.

В примере 8.11 показано, как выглядит реализация фильтра.

Пример 8.11 ❖ filter_plugins/surround_by_quotes.py

```
# Взято по адресу: http://stackoverflow.com/a/15515929/742

def surround_by_quote(a_list):
    return ['"%s"' % an_element for an_element in a_list]

class FilterModule(object):
    def filters(self):
        return {'surround_by_quote': surround_by_quote}
```

Функция `surround_by_quote` реализует фильтр Jinja2. Класс `FilterModule` определяет метод `filters`, который выводит словарь с именем функции фильтра и самой функцией. Класс `FilterModule` обеспечивает доступность фильтра для Ansible.

Кроме того, в каталог *~/.ansible/plugins/filter* или */usr/share/ansible/plugins/filter* можно установить свои плагины фильтров. Или указать другой каталог в переменной окружения `ANSIBLE_FILTER_PLUGINS`, где хранятся ваши плагины.

Подстановки

В идеальном мире вся информация о вашей 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 *etcd*[2]. Ansible поддерживает функции *подстановки*,

---

[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.*).

[2] *etcd* – распределенное хранилище типа «ключ/значение», поддерживаемое проектом CoreOS (https://coreos.com/etcd/).

позволяющие читать настройки из разных источников, а затем использовать их в сценариях и шаблонах.

Перечень этих функций приводится в табл. 8.3.

Таблица 8.3. Функции поиска

Имя	Описание
`file`	Содержимое файла
`password`	Случайно сгенерированный пароль
`pipe`	Вывод команды, выполненной локально
`env`	Переменная окружения
`template`	Шаблон Jinja2
`csvfile`	Запись в файле *.csv*
`dnstxt`	Запись в DNS типа TXT
`redis_kv`	Значение ключа в Redis
`etcd`	Значение ключа в etcd

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', '/путь/к/файлу/file.txt')
```

В сценариях подстановка должна заключаться в фигурные скобки {{ }}, их также можно использовать в шаблонах.

В этом разделе будет представлен только краткий обзор доступных подстановок. Более подробную информацию можно найти в документации Ansible (http://docs.ansible.com/ansible/playbooks_lookups.html).

Все плагины подстановок в Ansible выполняются на управляющей машине, а не на удаленном хосте.

file

Допустим, на управляющей машине имеется текстовый файл, содержащий публичный SSH-ключ, который необходимо скопировать на удаленный сервер. В примере 8.12 показано, как использовать подстановку `file` для чтения содержимого файла и его передачи модулю в параметре.

Пример 8.12 ❖ Использование подстановки file

```
- name: Add my public key as an EC2 key
  ec2_key: name=mykey key_material="{{ lookup('file', \
  '/Users/lorin/.ssh/id_rsa.pub') }}"
```

Подстановки также можно использовать в шаблонах. Если потребуется использовать тот же прием для создания файла *authorized_keys* с содержимым файла публичного ключа, можно создать шаблон Jinja2, выполняющий под-

становку, как показано в примере 8.13, и затем вызвать модуль `template`, как показано в примере 8.14.

Пример 8.13 ❖ authorized_keys.j2

```
{{ lookup('file', '/Users/lorin/.ssh/id_rsa.pub') }}
```

Пример 8.14 ❖ Задача, генерирующая файл authorized_keys

```
- name: copy authorized_host file
  template: src=authorized_keys.j2 dest=/home/deploy/.ssh/authorized_keys
```

pipe

Подстановка `pipe` запускает внешнюю программу на управляющей машине и принимает ее вывод.

Например, если сценарий использует систему контроля версий `git` и требуется получить значение SHA-1 последней команды `git commit`[1], для этого можно использовать подстановку `pipe`:

```
- name: get SHA of most recent commit
  debug: msg="{{ lookup('pipe', 'git rev-parse HEAD') }}"
```

Результат будет выглядеть примерно так:

```
TASK: [get the sha of the current commit] *********************************
ok: [myserver] => {
    "msg": "e7748af0f040d58d61de1917980a210df419eae9"
}
```

env

Подстановка `env` извлекает значение переменной окружения на управляющей машине. Например:

```
- name: get the current shell
  debug: msg="{{ lookup('env', 'SHELL') }}"
```

Поскольку я использую командную оболочку Zsh, у меня результат выглядит так:

```
TASK: [get the current shell] *************************************************
ok: [myserver] => {
    "msg": "/bin/zsh"
}
```

password

Подстановка `password` возвращает случайно сгенерированный пароль, а также записывает его в файл, указанный в аргументе. Например, если потребуется

[1] Если это покажется вам странным, не беспокойтесь – это всего лишь пример выполнения команды.

создать пользователя базы данных Postgres с именем deploy и случайным паролем, а затем записать пароль в файл *deploypassword.txt* на управляющей машине, это можно сделать так:

```
- name: create deploy postgres user
  postgresql_user:
    name: deploy
    password: "{{ lookup('password', 'deploy-password.txt') }}"
```

template

Подстановка template позволяет получить результат применения шаблона Jinja2. Например, для шаблона, представленного в примере 8.15:

Пример 8.15 ❖ message.j2

```
This host runs {{ ansible_distribution }}
```

следующая задача:

```
- name: output message from template
  debug: msg="{{ lookup('template', 'message.j2') }}"
```

вернет такой результат:

```
TASK: [output message from template] ***************************************
ok: [myserver] => {
    "msg": "This host runs Ubuntu\n"
}
```

csvfile

Подстановка csvfile читает запись из файла *.csv*. Допустим, у нас имеется файл *.csv*, который выглядит, как показано в примере 8.16.

Пример 8.16 ❖ users.csv

```
username,email
lorin,lorin@ansiblebook.com
john,john@example.com
sue,sue@example.org
```

и нам нужно получить электронный адрес Сью, 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*, разделитель и какие столбцы необходимо вывести. В данном примере мы используем файл *users.csv* и указываем, что поля разделены запятыми. Мы также сообщаем плагину, что ищем строку, в первом столбце которой хранится имя `sue`, и нам нужно значение второго столбца (столбец 1, индексация начинается с 0). В ответ плагин возвращает значение *sue@example.org*.

Если искомое имя пользователя хранится в переменной `username`, 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')
```

dnstxt

Модуль `dnstxt` требует установки пакета Python *dnspython* на управляющей машине.

Многие читатели наверняка знают, что такое система доменных имен (Domain Name System, DNS). DNS – это служба, преобразующая имена хостов, такие как *ansiblebook.com*, в IP-адреса, например *64.99.80.30*.

DNS ассоциирует с именем хоста одну или несколько записей. Наиболее используемыми типами записей DNS являются записи *A* и *CNAME*, которые связывают имя хоста с IP-адресом (запись *A*) или с псевдонимом (запись *CNAME*).

Протокол DNS содержит еще один тип записей – *TXT*. Запись *TXT* – это всего лишь произвольная строка, которую можно связать с именем хоста. Если вы привязали запись *TXT* к имени хоста, любой сможет получить этот текст с помощью клиента DNS.

Например, я владею доменом *ansiblebook.com* и хочу создать запись *TXT*, связанную с любыми именами хостов, входящих в домен[1]. Я привязал запись *TXT* к имени хоста *ansiblebook.com*, она содержит номер ISBN этой книги. Получить запись *TXT* можно с помощью инструмента командной строки `dig`, как показано в примере 8.17.

Пример 8.17 ❖ Извлечение записи TXT с помощью инструмента dig

```
$ dig +short ansiblebook.com TXT
"isbn=978-1491979808"
```

Подстановка `dnstxt` запрашивает у сервера DNS запись *TXT*, ассоциированную с хостом. Если создать такую задачу в сценарии:

[1] Провайдеры услуг DNS обычно предоставляют интерфейс для выполнения задач, связанных с DNS, таких как создание записей *TXT*.

```
- name: look up TXT record
  debug: msg="{{ lookup('dnstxt', 'ansiblebook.com') }}"
```

она вернет:

```
TASK: [look up TXT record] ****************************************************
ok: [myserver] => {
    "msg": "isbn=978-1491979808"
}
```

Если с хостом связано несколько записей *TXT*, тогда модуль вернет их «склеенными» вместе. Порядок «склеивания» каждый раз может быть разным. Например, если бы для *ansiblebook.com* была определена вторая запись *TXT* с текстом:

```
author=lorin
```

тогда подстановка `dnstxt` вывела бы случайным образом один из вариантов:

- `isbn=978-1491979808author=lorin`;
- `author=lorinisbn=978-1491979808`.

redis_kv

Модуль `redis_kv` требует установки пакета Python *redis* на управляющей машине.

Redis – популярное хранилище типа «ключ/значение», часто используемое как кэш, а также для хранения данных в службах очередей заданий, таких как Sidekiq. С помощью подстановки `redis_kv` можно извлекать значения ключей. Ключ должен иметь вид строки, поскольку модуль выполняет эквивалент команды `GET`.

Допустим, у нас имеется сервер Redis, запущенный на управляющей машине, и мы определили ключ `weather` со значением `sunny`:

```
$ redis-cli SET weather sunny
```

Если определить в сценарии задачу извлечения этого ключа из хранилища Redis:

```
- name: look up value in Redis
  debug: msg="{{ lookup('redis_kv', 'redis://localhost:6379,weather') }}"
```

она вернет следующее:

```
TASK: [look up value in Redis] ************************************************
ok: [myserver] => {
    "msg": "sunny"
}
```

Если адрес URL не задан, модуль по умолчанию использует адрес *redis://localhost:6379*. То есть предыдущую задачу можно переписать так (обратите внимание на запятую перед ключом):

```
lookup('redis_kv', ',weather')
```

etcd

Etcd – распределенное хранилище типа «ключ/значение», обычно используется для хранения данных конфигураций и реализации поиска служб. Для получения значения ключа из этого хранилища можно использовать подстановку `etcd`.

Допустим, у нас имеется сервер `etcd`, запущенный на управляющей машине, и мы определили ключ `weather` со значением `cloudy`:

```
$ curl -L http://127.0.0.1:4001/v2/keys/weather -XPUT -d value=cloudy
```

Если определить в сценарии задачу извлечения этого ключа из хранилища `etcd`:

```
- name: look up value in etcd
  debug: msg="{{ lookup('etcd', 'weather') }}"
```

она вернет следующее:

```
TASK: [look up value in etcd] *************************************************
ok: [localhost] => {
    "msg": "cloudy"
}
```

По умолчанию подстановка `etcd` обращается к серверу etcd по адресу *http://127.0.0.1:4001*. Но его можно изменить, установив переменную окружения `ANSIBLE_ETCD_URL` перед запуском команды `ansible-playbook`.

Написание собственного плагина

Если ни один из имеющихся плагинов вас не устраивает, всегда можно написать собственный плагин. Разработка собственных плагинов для подстановок не является темой данной книги, но если вас действительно заинтересовал данный вопрос, я предлагаю изучить исходный код плагинов для подстановок, которые поставляются с Ansible (https://github.com/ansible/ansible/tree/devel/lib/ansible/plugins/lookup).

Написав свой плагин, поместите его в один из следующих каталогов:

- *lookup_plugins* в каталоге со сценарием;
- *~/.ansible/plugins/lookup*;
- */usr/share/ansible_plugins/lookup_plugins*;
- указанный в переменной окружения `ANSIBLE_LOOKUP_PLUGINS`.

Сложные циклы

До сих пор, когда мы писали задачи, выполняющие обход списка объектов, мы использовали выражение `with_items`, в котором определяли список объектов. Это самый распространенный способ выполнения циклов, но Ansible поддерживает также другие механизмы реализации итераций. Их список приводится в табл. 8.4.

Таблица 8.4. 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 в шаблоне

В официальной документации (http://bit.ly/1F6kfCP) эта тема рассматривается достаточно подробно, поэтому я приведу лишь несколько примеров, чтобы дать вам представление, как работают эти конструкции.

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 отправить Slack-сообщение для каждого из них:

```
Leslie Lamport
Silvio Micali
Shafi Goldwasser
Judea Pearl
```

В примере 8.18 показано, как использовать with_lines для чтения файла и выполнения итераций по файлу, строка за строкой.

Пример 8.18 ❖ Цикл с помощью with_lines

```
- name: Send out a slack message
  slack:
    domain: example.slack.com
    token: "{{ slack_token }}"
    msg: "{{ item }} was in the list"
  with_lines:
    - cat files/turing.txt
```

with_fileglob

Конструкция with_fileglob используется, когда нужно выполнить итерации по набору файлов на контрольной машине.

В примере 8.19 показано, как обойти файлы с расширением *.pub* в каталоге */var/keys*, а также в подкаталоге *keys*, находящемся в одном каталоге со сценарием.

Пример 8.19 ❖ Использование with_fileglob для добавления ключей

```
- name: add public keys to account
  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 `eth0`, тогда в Ansible будет существовать факт с именем `ansible_eth0` и с ключом `ipv4`, который содержит примерно такой словарь:

```
{
  "address": "10.0.2.15",
  "netmask": "255.255.255.0",
  "network": "10.0.2.0"
}
```

Можно обойти элементы этого словаря и вывести их по одному:

```
- name: iterate over ansible_eth0
  debug: msg={{ item.key }}={{ item.value }}
  with_dict: "{{ ansible_eth0.ipv4 }}"
```

Результат будет выглядеть так:

```
TASK: [iterate over ansible_eth0] *********************************************
ok: [myserver] => (item={'key': u'netmask', 'value': u'255.255.255.0'}) => {
    "item": {
        "key": "netmask",
        "value": "255.255.255.0"
    },
    "msg": "netmask=255.255.255.0"
}
ok: [myserver] => (item={'key': u'network', 'value': u'10.0.2.0'}) => {
    "item": {
        "key": "network",
        "value": "10.0.2.0"
    },
    "msg": "network=10.0.2.0"
}
```

```
ok: [myserver] => (item={'key': u'address', 'value': u'10.0.2.15'}) => {
    "item": {
        "key": "address",
        "value": "10.0.2.15"
    },
    "msg": "address=10.0.2.15"
}
```

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 8.12 можно переписать с использованием формы `with_file`, как показано в примере 8.20.

Пример 8.20 ❖ Использование подстановки file в качестве конструкции цикла

```
- name: Add my public key as an EC2 key
  ec2_key: name=mykey key_material="{{ item }}"
  with_file: /Users/lorin/.ssh/id_rsa.pub
```

Обычно плагины подстановок используются в роли циклических конструкций, только если требуется получить список. Именно поэтому я отделил плагины из табл. 8.3 (возвращающие строки) от плагинов в табл. 8.4 (возвращающие списки).

Управление циклами

Начиная с версии 2.1 Ansible предоставляет пользователям еще более богатые возможности выполнения циклических операций.

Выбор имени переменной цикла

Выражение `loop_var` позволяет дать переменной цикла другое имя, отличное от имени `item`, используемого по умолчанию, как показано в примере 8.21.

Пример 8.21 ❖ Использование переменной цикла user

```
- user:
  name: "{{ user.name }}"
  with_items:
    - { name: gil }
    - { name: sarina }
    - { name: leanne }
  loop_control:
    loop_var: user
```

В примере 8.21 выражение `loop_var` дает лишь косметическое удобство, но вообще с ее помощью можно определять гораздо более сложные циклы.

В примере 8.22 реализован цикл по нескольким задачам. Для этого в нем используется инструкция `include` с выражением `with_items`.

Однако файл *vhosts.yml* может включать задачи, также 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, мы можем указать другое имя в выражении `loop_var` для внешнего цикла.

Пример 8.22 ❖ 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 ❶
```

❶ Изменение имени переменной внешнего цикла для предотвращения конфликтов.

В подключаемой задаче (объявленной в файле *vhosts.yml*), которая представлена в примере 8.23, мы теперь без опаски можем использовать имя `item` по умолчанию.

Пример 8.23 ❖ Подключаемый файл может содержать циклы

```
- name: create nginx directories
  file:
    path: /var/www/html/{{ vhost.domain }}/{{ item }} ❶
  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
```

❶ 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
  template:
    src: "{{ item.domain }}.conf.j2"
    dest: "/etc/nginx/conf.d/{{ item.domain }}.conf
```

```
  with_items:
    - { domain: www1.example.com, ssl_enabled: yes }
    - { domain: www2.example.com }
    - { domain: www3.example.com,
        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] **********************************************
ok: [localhost] => (item={u'domain': u'www1.example.com', u'ssl_enabled': True})
ok: [localhost] => (item={u'domain': u'www2.example.com'})
ok: [localhost] => (item={u'domain': u'www3.example.com', u'aliases':
[u'edge2.www.example.com', u'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
  template:
    src: "{{ item.domain }}.conf.j2"
    dest: "/etc/nginx/conf.d/{{ item.domain }}.conf"
  with_items:
    - { domain: www1.example.com, ssl_enabled: yes }
    - { domain: www2.example.com }
    - { domain: www3.example.com,
        aliases: [ edge2.www.example.com, eu.www.example.com ] }
  loop_control:
    label: "for domain {{ item.domain }}" ❶
```

❶ Добавление метки

В результате вывод получится более удобочитаемым:

```
TASK [create nginx vhost configs] **********************************************
ok: [localhost] => (item=for domain www1.example.com)
ok: [localhost] => (item=for domain www2.example.com)
ok: [localhost] => (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`.

Подключение

Функция `include` позволяет подключать задачи или даже целые сценарии, в зависимости от того, где используется эта функция. Она часто применяется в ролях для определения или группировки задач и их аргументов в отдельных подключаемых файлах.

Рассмотрим пример. В примере 8.24 определены две задачи, использующие идентичные аргументы в выражениях tag, when и become.

Пример 8.24 ❖ Идентичные аргументы

```
- name: install nginx
  package:
    name: nginx
  tags: nginx ❶
  become: yes ❷
  when: ansible_os_family == 'RedHat' ❸

- name: ensure nginx is running
  service:
    name: nginx
    state: started
    enabled: yes
  tags: nginx ❶
  become: yes ❷
  when: ansible_os_family == 'RedHat' ❸
```

❶ Идентичные теги
❷ Идентичные привилегии
❸ Идентичные условия

Если выделить эти две задачи в отдельный файл, как показано в примере 8.25, и подключать его, как показано в примере 8.26, можно упростить сценарий, определив аргументы только в операции подключения.

Пример 8.25 ❖ Выделение задач в отдельный файл

```
- name: install nginx
  package:
    name: nginx

- name: ensure nginx is running
  service:
    name: nginx
    state: started
    enabled: yes
```

Пример 8.26 ❖ Подключение задач и применение общих аргументов

```
- include: nginx_include.yml
  tags: nginx
  become: yes
  when: ansible_os_family == 'RedHat'
```

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.

```
- include: Redhat.yml
  when: ansible_os_family == 'Redhat'

- include: Debian.yml
  when: ansible_os_family == 'Debian'
```

Начиная с версии 2.0 Ansible позволяет динамически подключать файлы, используя подстановку переменных:

```
- include: "{{ ansible_os_family }}.yml"
  static: no
```

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 4) не заполняются, когда выполняется команда `--list-tasks`.

Подключение ролей

Выражение `include_role` – это особый вид операции подключения. В отличие от выражения `role`, которое будет использовать все компоненты роли, выражение `include_role` позволяет явно определить, какие компоненты подключаемой роли должны использоваться.

По аналогии с выражением `include`, подключение ролей имеет два режима: статический и динамический, и Ansible автоматически угадывает, какой режим использовать. Однако вы всегда можете добавить выражение `static: yes` или `static: no`, – чтобы явно определить режим.

```
- name: install nginx
  yum:
    pkg: nginx

- name: install php
  include_role:
    name: php ❶

- name: configure nginx
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
```

❶ Подключает и выполняет *main.yml* из роли `php`.

Выражение `include_role` также открывает доступ к обработчикам.

Выражение `include_role` также может помочь избежать конфликтов компонентов ролей, зависящих друг от друга. Представьте, что в зависимой роли, которая выполняется перед главной ролью, имеется задача `file`, изменяющая 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`.

На момент написания этих строк выражение `include_role` все еще было отмечено как *экспериментальное*, это означает, что оно не обеспечивает обратной совместимости.

Блоки

Подобно выражению `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: для обработки ошибок.

Обработка ошибок с помощью блоков

Обработка ошибок всегда была непростой задачей. Система Ansible изначально предусматривает возможность появления ошибок на хостах. Если возника-

ет 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 вам возможность выполнить какие-то действия, когда операция объявляется потерпевшей неудачу.

А благодаря выражению `block`, как показано в примере 8.27, Ansible поднимает обработку ошибок еще на уровень выше и позволяет автоматизировать повторное выполнение или откат задач, потерпевших ошибку.

Пример 8.27 ❖ app-upgrade.yml

```
---
- block: ❶
  - debug: msg="You will see a failed tasks right after this"
  - command: /bin/false
  - debug: "You won't see this message"
  rescue: ❷
  - debug: "You only see this message in case of an failure in the block"
  always: ❸
  - debug: "This will be always executed"
```

❶ Начало выражения `block`.
❷ Определяет задачи, выполняемые, если в выражении `block` произойдет ошибка.
❸ Задачи, которые выполняются всегда.

Если у вас есть опыт программирования, реализация обработки ошибок в Ansible может напомнить вам парадигму *try-catch-finally*, и она работает похожим образом.

Для демонстрации возьмем самую обычную повседневную задачу: обновление приложения. Приложение распределяется в кластере виртуальных машин (ВМ) и развертывается в облаке IaaS (Apache CloudStack). Кроме того, облако поддерживает возможность создания 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 8.28).

Пример 8.28 ❖ 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, работающий под управлением балансировщика нагрузки, даже если попытка обновления потерпит неудачу.

Задачи в выражении `always` будут выполняться всегда, даже если будут обнаружены ошибки при выполнении задач в выражении `rescue`! Тщательно отбирайте задачи, помещаемые в `always`.

Если под управление балансировщиком нагрузки должна возвращаться только обновленная ВМ, сценарий нужно изменить, как показано в примере 8.29.

Пример 8.29 ❖ 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 сценария. Они будут выполнены, только если выражение `rescue` не будет выполнено. То есть под управление балансировщика нагрузки будут возвращаться только обновленные ВМ.

Окончательная версия сценария представлена в примере 8.30.

Пример 8.30 ❖ Сценарий обновления приложения с обработкой ошибок

```
---
- hosts: app-servers
```

```
serial: 1
tasks:
- name: Take app server out of the load balancer
  local_action:
    module: cs_loadbalancer_rule_member
    name: balance_http
    vm: "{{ inventory_hostname_short }}"
    state: absent
- name: Create a VM snapshot before an upgrade
  local_action:
    module: cs_vmsnapshot
    name: Snapshot before upgrade
    vm: "{{ inventory_hostname_short }}"
    snapshot_memory: yes

- 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
    local_action:
      module: cs_vmsnapshot
      name: Snapshot before upgrade
      vm: "{{ inventory_hostname_short }}"
      state: revert

- name: Re-add app server to the loadbalancer
  local_action:
    module: cs_loadbalancer_rule_member
    name: balance_http
    vm: "{{ inventory_hostname_short }}"
    state: present

- name: Remove a VM snapshot after successful upgrade or successful rollback
  local_action:
    module: cs_vmsnapshot
    name: Snapshot before upgrade
    vm: "{{ inventory_hostname_short }}"
    state: absent
```

Шифрование конфиденциальных данных при помощи Vault

Сценарий Mezzanine требует доступа 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 он 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.

Вот как можно зашифровать имеющийся файл:

```
$ ansible-vault encrypt secrets.yml
```

А так можно создать новый зашифрованный файл *secrets.yml*:

```
$ ansible-vault create secrets.yml
```

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, по умолчанию используется `vim`.

В примере 8.31 показан вариант содержимого файла, зашифрованного с помощью `ansible-vault`.

Пример 8.31 ❖ Содержимое файла, зашифрованного с помощью ansible-vault

```
$ANSIBLE_VAULT;1.1;AES256
34306434353230663665633539363736353836333936383931316434343030316366653331363262
66306333663831353862663330303936343036646136623 50a623837663462393031626233376232
31613735376632333231626661663766262393337383565323931623038633930333036666383530
...
62346633343464313330383832646531623338633438336465323166626335623639383363643438
64636665366538343038383031656461613665662656306639643833316565343 6
```

К файлу, зашифрованному с помощью `ansible-vault`, можно обращаться в секции `vars_files`, как к обычному файлу, – вам не придется ничего менять в примере 6.28, 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 mezzanine.yml --ask-vault-pass
```

Также можно сохранить пароль в текстовом файле и сообщить `ansible-playbook`, где он находится, добавив параметр `--vault-password-file`:

```
$ ansible-playbook mezzanine --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. 8.5 перечислены доступные команды `ansible-vault`.

Таблица 8.5. 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*

Глава 9

Управление хостами, задачами и обработчиками

Иногда поведение по умолчанию системы Ansible не в полной мере соответствует нашим желаниям. В этой главе мы познакомимся с инструментами Ansible, которые позволяют настроить выбор обслуживаемых хостов, запуск задач и использование обработчиков.

Шаблоны для выбора хостов

До сих пор параметр `hosts` 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 базы данных в тестовом окружении (группа `staging`) можно выбрать так:

```
hosts: staging:&database
```

В табл. 9.1 перечислены шаблоны, поддерживаемые в Ansible. Обратите внимание, что регулярные выражения всегда начинаются со знака тильды (~).

Таблица 9.1. 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`

Ansible поддерживает также комбинации шаблонов. Например:

```
hosts: dev:staging:&database:!queue
```

Ограничение обслуживаемых хостов

Для ограничения хостов, на которых будет выполняться сценарий, используются флаги `-l hosts` или `--limit hosts`, как показано в примере 9.1.

Пример 9.1 ❖ Ограничение обслуживаемых хостов

```
$ ansible-playbook -l hosts playbook.yml
$ ansible-playbook --limit hosts 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 `local_action`.

Представьте, что сервер, на который нужно установить Mezzanine, только что перезагрузился. Если запустить сценарий слишком рано, мы получим ошибку, поскольку сервер еще не закончил процедуру загрузки. Можно приостановить сценарий, обратившись к модулю `wait_for`, и перед повторным запуском сценария дождаться момента, когда сервер SSH будет готов принимать соединения. В данном случае мы запускаем модуль на нашем ноутбуке, а не на удаленном хосте.

Первая задача приостанавливает сценарий:

```
- name: wait for ssh server to be running
  local_action: wait_for port=22 host="{{ inventory_hostname }}"
    search_regex=OpenSSH
```

Обратите внимание, что в задаче мы ссылаемся на переменную `inventory_hostname`, вместо которой будет подставлено имя удаленного хоста, а не `localhost`. Это происходит потому, что эти переменные все еще представляют удаленные хосты, хотя задача выполняется локально.

Если операция охватывает несколько хостов и вы используете `local_action`, задача выполнится несколько раз – по одному для каждого хоста. Такое поведение можно запретить, использовав `run_once`, как показано в разделе «Последовательное выполнение задачи на хостах по одному» ниже.

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 уведомления 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 9.2 показано, как можно было бы использовать выражение `delegate_to` в этом случае.

Пример 9.2 ❖ Использование delegate_to для настройки Nagios

```
- name: enable alerts for web servers
  hosts: web
  tasks:
    - name: enable alerts
      nagios: action=enable_alerts service=web host={{ inventory_hostname }}
      delegate_to: nagios.example.com
```

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://github.com/ansible/ansible-examples).

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 при-

ложений выводится из-под управления балансировщиком нагрузки, обновляется и возвращается обратно. При этом не хотелось бы приостанавливать все серверы сразу, потому что в этом случае служба станет недоступной.

Ограничить число хостов, на которых Ansible запускает сценарий, можно выражением `serial`. В примере 9.3 продемонстрированы последовательный вывод хостов из-под управления балансировщиком нагрузки Amazon EC2, обновление системных пакетов и возвращение хостов обратно. Подробнее о Amazon EC2 рассказывается в главе 14.

Пример 9.3 ❖ 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
      local_action: ec2_elb
      args:
        instance_id: "{{ ansible_ec2_instance_id }}"
        state: absent

    - name: upgrade packages
      apt: update_cache=yes upgrade=yes

    - name: put the host back in the elastic load balancer
      local_action: ec2_elb
      args:
        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, 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` можно также передать проценты вместо целого числа. В этом случае Ansible сама определит, сколько хостов из числа участвующих в операции соответствуют этому значению, как показано в примере 9.4.

Пример 9.4 ❖ 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, и 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 список с размерами пакетов. Элементами этого могут быть целые числа или проценты, как показано в примере 9.5.

Пример 9.5 ❖ Использование списка с размерами пакетов

```
- name: configure CDN servers
  hosts: cdn
  serial:
    - 1
    - 30%
  tasks:
    # далее следуют задачи
```

Ansible будет ограничивать количество хостов в каждом пакете, следуя по списку, пока не будет достигнут последний его элемент или не останется хос-

тов 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 (то есть 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 `local_action`, если сценарий вовлекает несколько хостов и необходимо выполнить локальную задачу только один раз:

```
- name: run the task locally, only once
  local_action: command /opt/my-custom-command
  run_once: true
```

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 9.7, для демонстрации применения разных стратегий. Мы используем минимальный файл `hosts`, представленный в примере 9.6, содержащий три хоста, для каждого из которых определена переменная `sleep_seconds` со своим значением секунд.

Пример 9.6 ❖ Файл hosts с тремя хостами и с разными значениями переменной `sleep_seconds`

```
one sleep_seconds=1
two sleep_seconds=6
three sleep_seconds=10
```

linear

Сценарий в примере 9.7 выполняет операцию с тремя задачами локально, как того требует выражение `connection: local`. Каждая задача приостанавливается на время, указанное в переменной `sleep_seconds`.

Пример 9.7 ❖ Сценарий для проверки стратегии linear

```
---
- hosts: all
  connection: local
  tasks:
  - name: first task
    shell: sleep "{{ sleep_seconds }}"

  - name: second task
    shell: sleep "{{ sleep_seconds }}"

  - name: third task
    shell: sleep "{{ sleep_seconds }}"
```

Если запустить этот сценарий со стратегией по умолчанию `linear`, он выведет результаты, показанные в примере 9.8.

Пример 9.8 ❖ Результаты выполнения сценария со стратегией linear

```
$ ansible-playbook strategy.yml -i hosts

PLAY [all] *********************************************************************

TASK [setup] *******************************************************************
ok: [two]
ok: [three]
ok: [one]

TASK [first task] **************************************************************
changed: [one]
changed: [two]
changed: [three]

TASK [second task] *************************************************************
changed: [one]
changed: [two]
changed: [three]

TASK [third task] **************************************************************
changed: [one]
changed: [two]
changed: [three]

PLAY RECAP *********************************************************************
one                        : ok=4   changed=3   unreachable=0   failed=0
three                      : ok=4   changed=3   unreachable=0   failed=0
two                        : ok=4   changed=3   unreachable=0   failed=0
```

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 9.9, его вывод изменится (см. пример 9.10).

Пример 9.9 ❖ Выбор стратегии free в сценарии

```
---
- hosts: all
  connection: local
  strategy: free ❶
  tasks:
  - name: first task
    shell: sleep "{{ sleep_seconds }}"

  - name: second task
    shell: sleep "{{ sleep_seconds }}"

  - name: third task
    shell: sleep "{{ sleep_seconds }}"
```

❶ Установлена стратегия free.

Как показывает вывод в примере 9.10, хост one завершил операцию еще до того, как два других хоста успели выполнить первую задачу.

Пример 9.10 ❖ Результаты выполнения сценария со стратегией free

```
$ ansible-playbook strategy.yml -i hosts

PLAY [all] *********************************************************************

TASK [setup] *******************************************************************
ok: [one]
ok: [two]
ok: [three]

TASK [first task] **************************************************************
changed: [one]

TASK [second task] *************************************************************
changed: [one]
```

```
TASK [third task] ************************************************************
changed: [one]

TASK [first task] ************************************************************
changed: [two]
changed: [three]

TASK [second task] ***********************************************************
changed: [two]

TASK [third task] ************************************************************
changed: [two]

TASK [second task] ***********************************************************
changed: [three]

TASK [third task] ************************************************************
changed: [three]

PLAY RECAP *******************************************************************
one                        : ok=4   changed=3   unreachable=0   failed=0
three                      : ok=4   changed=3   unreachable=0   failed=0
two                        : ok=4   changed=3   unreachable=0   failed=0
```

В обоих случаях операция выполняется за то же время. Однако при определенных условиях операция может выполняться быстрее, когда используется стратегия `free`.

Подобно многим базовым компонентам в Ansible, управление стратегиями реализовано в виде плагина нового типа.

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, вы 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:

```
---
- hosts: localhost
  pre_tasks:
  - command: echo Pre Tasks
    notify: print message
```

```
  tasks:
  - command: echo Tasks
    notify: print message

  post_tasks:
  - command: echo Post Tasks
    notify: print message

  handlers:
  - name: print message
    command: echo handler executed
```

Если запустить этот сценарий, он выведет следующее:

```
$ ansible-playbook pre_post_tasks_handlers.yml

PLAY [localhost] ***************************************************************

TASK [setup] *******************************************************************
ok: [localhost]

TASK [command] *****************************************************************
changed: [localhost]

RUNNING HANDLER [print message] ************************************************
changed: [localhost]

TASK [command] *****************************************************************
changed: [localhost]

RUNNING HANDLER [print message] ************************************************
changed: [localhost]

TASK [command] *****************************************************************
changed: [localhost]

RUNNING HANDLER [print message] ************************************************
changed: [localhost]

PLAY RECAP *********************************************************************
localhost                  : ok=7    changed=6    unreachable=0    failed=0
```

Принудительный запуск обработчиков

Возможно, вам показалось странным, что выше я написал: *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 9.12 приводится часть роли `nginx`, где используется модуль `meta` с выражением `flush_handlers` в середине.

Сделано это по двум причинам:

1) чтобы очистить некоторые старые данные в разделе *vhost* конфигурации Nginx, что можно сделать только в отсутствие любых процессов, использующих его (например, после перезапуска службы);

2) чтобы выполнить некоторые *тесты* и убедиться, что обращение к некоторому URL возвращает OK. Но такая проверка не имеет большого смысла до перезапуска служб.

В примере 9.11 показана конфигурация роли `nginx`: имя хоста и порт для проверки, список в разделе `vhosts` с именем и шаблоном и некоторые устаревшие виртуальные хосты, которые требуется удалить:

Пример 9.11 ❖ Конфигурация для роли `nginx`

```
nginx_healthcheck_host: health.example.com
nginx_healthcheck_port: 8080

vhosts:
  - name: www.example.com
    template: default.conf.j2

absent_vhosts:
  - obsolete.example.com
  - www2.example.com
```

В файл задач для роли *roles/nginx/tasks/main.yml* (см. пример 9.12) мы добавили задачи `meta` с соответствующим аргументом `flush_handlers`, между другими задачами, но именно там, где нам хотелось бы: перед задачами проверки и очистки.

Пример 9.12 ❖ Очистка и проверка после перезапуска службы

```
---
- name: install nginx
  yum:
    pkg: nginx
  notify: restart nginx

- name: configure nginx vhosts
  template:
    src: conf.d/{{ item.template | default(item.name) }}.conf.j2
    dest: /etc/nginx/conf.d/{{ item.name }}.conf
  with_items: "{{ vhosts }}"
  when: item.name not in vhosts_absent
  notify: restart nginx

- name: removed unused nginx vhosts
  file:
    path: /etc/nginx/conf.d/{{ item }}.conf
    state: absent
  with_items: "{{ vhosts_absent }}"
  notify: restart nginx

- name: validate nginx config ❶
  command: nginx -t
  changed_when: false
  check_mode: false
```

```
- name: flush the handlers
  meta: flush_handlers ❷

- name: remove unused vhost directory
  file:
    path: /srv/www/{{ item }} state=absent
  when: item not in vhosts
  with_items: "{{ vhosts_absent }}"

- name: check healthcheck ❸
  local_action:
    module: uri
    url: http://{{ nginx_healthcheck_host }}:{{ nginx_healthcheck_port }}/healthcheck
    return_content: true
  retries: 10
  delay: 5
  register: webpage

- fail:
  msg: "fail if healthcheck is not ok"
when: not webpage|skipped and webpage|success and "ok" not in webpage.content
```

❶ Проверка конфигурации непосредственно перед принудительным запуском обработчиков.
❷ Принудительный запуск обработчиков между задачами.
❸ Выполнение проверочных тестов. Обратите внимание, что это может быть динамическая страница, проверяющая доступность базы данных.

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:

```
---
- hosts: mailservers
  tasks:
    - copy:
      src: main.conf
      dest: /etc/postfix/main.cnf
    notify: postfix config changed ❶

handlers:
  - name: restart postfix
    service: name=postfix state=restarted
    listen: postfix config changed ❶
```

❶ Регистрация *события*, появления которого должны дождаться обработчики.

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 дополнительных обработчиках то же событие.

Область видимости обработчиков ограничивается уровнем операции. Нельзя известить обработчики в другой операции с использованием или без использования выражения `listen`.

Выполнение обработчиков по событиям: случай SSL

Истинная ценность задержки обработчиков проявляется при определении ролей или зависимостей между ролями. Один из очевидных случаев, с которыми я сталкивался, – 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 9.13) 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 9.14).

Пример 9.13 ❖ 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
  with_items: "{{ ssl_certs }}"

- name: copy SSL keys
  copy:
    src: "{{ item }}"
    dest: "{{ ssl_keys_path }}/"
    owner: root
    group: root
    mode: 0644
  with_items: "{{ ssl_keys }}"
  no_log: true
```

Пример 9.14 ❖ 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 9.15) 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 9.15 ❖ Настройки по умолчанию для роли SSL

```
---
ssl_certs: []
ssl_keys: []
```

С этого момента у нас появляется возможность использовать роль SSL в других ролях как зависимость, как показано в примере 9.16, 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 9.16 ❖ Роль nginx зависит от SSL

```
---
dependencies:
  - role: ssl
```

Логически зависимости имеют однонаправленный характер: роль `nginx` зависит от роли `ssl`, как показано на рис. 9.1.

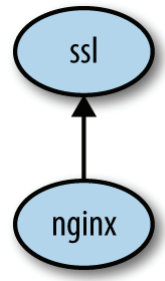

Рис. 9.1 ❖ Однонаправленная зависимость

Конечно, роль `nginx` могла бы обрабатывать все аспекты, касающиеся вебсервера Nginx. Эта роль имеет задачу в файле *roles/nginx/tasks/main.yml* (пример 9.17) для развертывания шаблона с конфигурацией *nginx* и перезапускает службу *nginx*, посылая уведомление обработчику по его имени.

Пример 9.17 ❖ Задачи в роли `nginx`

```
---
- name: configure nginx
```

```
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
  notify: restart nginx ❶
```

❶ Известить обработчик, перезапускающий службу *nginx*.

Соответствующий обработчик для роли `nginx` определен в файле *roles/nginx/handlers/main.yml*, как показано в примере 9.18.

Пример 9.18 ❖ Обработчики для роли `nginx`

```
---
- name: restart nginx  ❶
  service:
    name: nginx
    state: restarted
```

❶ Обработчик `restart nginx` перезапускает службу Nginx.

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 9.19.

Пример 9.19 ❖ 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 ❶

- 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 ❶
```

❶ Известить обработчик в роли `nginx`.

Отлично, сработало! Но подождите, мы только что добавили новую зависимость в нашу роль SSL: зависимость от роли `nginx`, как показано на рис. 9.2.

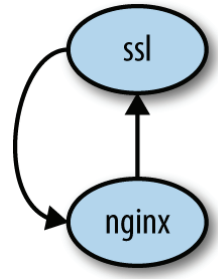

Рис. 9.2 ❖ Роль nginx зависит от роли SSL, а роль SSL зависит от роли nginx

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 было 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 9.20.

Пример 9.20 ❖ 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 ❶

- 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 ❶
```

❶ Отправка события `ssl_certs_changed`.

Как отмечалось, Ansible продолжит жаловаться на попытку уведомить неизвестный обработчик, и чтобы избавиться от назойливых жалоб, достаточно лишь добавить пустой обработчик в роль SSL, как показано в примере 9.21.

Пример 9.21 ❖ Добавление пустого обработчика в роль SSL

```
---
- name: SSL certs changed
  debug:
    msg: SSL changed event triggered
  listen: ssl_certs_changed ❶
```

❶ Пустой обработчик события `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 требуемый обработчик, мы просто добавим в него выражение `listen`, как показано в примере 9.22.

Пример 9.22 ❖ Добавление выражения `listen` в существующий обработчик в роли `nginx`

```
---
- name: restart nginx
  service:
    name: nginx
    state: restarted
  listen: ssl_certs_changed ❶
```

❶ Добавление выражения `listen` в существующий обработчик.

Если теперь опять взглянуть на граф зависимостей, можно заметить, что он изменился, как показано на рис. 9.3. Мы восстановили однонаправленный характер зависимости и получили возможность использовать роль `ssl` в других ролях.

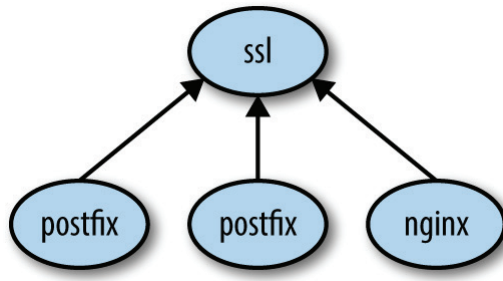

Рис. 9.3 ❖ Использование роли ssl в других ролях

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.

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 9.23.

Пример 9.23 ❖ Ожидание запуска SSH-сервера

```
- name: Deploy mezzanine
  hosts: web
  gather_facts: False
  # разделы vars и vars_files здесь не показаны
  tasks:
    - name: wait for ssh server to be running
      local_action: wait_for port=22 host="{{ inventory_hostname }}"
        search_regex=OpenSSH

    - name: gather facts
      setup:
  # Далее следуют остальные задачи
```

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:
```

```
  - 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 как факт. Каждый сетевой интерфейс представлен связанным с ним фактом. Например, данные о сетевом интерфейсе `eth0` хранятся в факте `ansible_eth0`. Это показано в примере 9.24.

Пример 9.24 ❖ Факт `ansible_eth0`

```
"ansible_eth0": {
    "active": true,
    "device": "eth0",
    "ipv4": {
        "address": "10.0.2.15",
        "netmask": "255.255.255.0",
        "network": "10.0.2.0"
    },
    "ipv6": [
        {
            "address": "fe80::a00:27ff:fefe:1e4d",
            "prefix": "64",
            "scope": "link"
        }
    ],
    "macaddress": "08:00:27:fe:1e:4d",
    "module": "e1000",
    "mtu": 1500,
    "promisc": false,
    "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_eth1.ipv4.address }}.xip.io"
domains:
  - "{{ ansible_eth1.ipv4.address }}.xip.io"
  - "www.{{ ansible_eth1.ipv4.address }}.xip.io"
```

Глава 10

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 два вида плагинов обратного вызова:

- *плагины стандартного вывода* (stdout plugins), влияющие на информацию, что выводится в окно терминала;
- *другие плагины*, выполняющие любые другие действия, не изменяющие вывода на экран.

Технически плагины обратного вызова делятся на три вида, а не на два:
- стандартного вывода;
- уведомлений;
- агрегаты.

Однако, поскольку реализация Ansible не различает плагины уведомлений и агрегаты, мы будем рассматривать их как одну разновидность, под названием *другие плагины*.

Плагины стандартного вывода

Плагин стандартного вывода управляет форматом отображения информации на экране. В каждый конкретный момент времени активным может быть только один плагин стандартного вывода.

Назначается плагин стандартного вывода установкой параметра `stdout_callback` в разделе `defaults` в файле *ansible.cfg*. Например, вот как можно выбрать плагин `actionable`:

```
[defaults]
stdout_callback = actionable
```

В табл. 10.1 перечислены плагины стандартного вывода, поддерживаемые в Ansible.

Таблица 10.1. Плагины стандартного вывода

Имя	Описание
`actionable`	Выводит только сообщения об изменениях и ошибках
`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
`oneline`	Действует подобно плагину minimal, но выводит информацию в одну строку
`selective`	Отображает вывод только отмеченных задач
`skippy`	Подавляет вывод для пропущенных хостов

actionable

Плагин `actionable` отображает вывод задачи, только если она изменила состояние хоста или потерпела неудачу. Это способствует уменьшению объема вывода.

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 [check out the repository on the host] ************************************
fatal: [web]: FAILED! => {"changed": false, "cmd": "/usr/bin/git clone --origin o
rigin '' /home/vagrant/mezzanine/mezzanine_example", "failed": true, "msg": "Clon
ing into '/home/vagrant/mezzanine/mezzanine_example'...\nPermission denied (publi
ckey).\r\nfatal: Could not read from remote repository.\n\nPlease make sure you h
ave the correct access rights\nand the repository exists.", "rc": 128, "stderr":
"Cloning into '/home/vagrant/mezzanine/mezzanine_example'...\nPermission 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"
: ["Cloning into '/home/vagrant/mezzanine/mezzanine_example'...", "Permission den
ied (publickey).", "fatal: Could not read from remote repository.", "", "Please m
ake sure you have the correct access rights", "and the repository exists."], "std
out": "", "stdout_lines": []}
```

Но благодаря дополнительному форматированию, осуществляемому плагином `debug`, читать вывод намного проще:

```
TASK [check out the repository on the host] ************************************
fatal: [web]: FAILED! => {
```

```
    "changed": false,
    "cmd": "/usr/bin/git clone --origin origin '' /home/vagrant/mezzanine/mezzani
    ne_example",
    "failed": true,
    "rc": 128
}

STDERR:

Cloning into '/home/vagrant/mezzanine/mezzanine_example'...
Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

MSG:

Cloning into '/home/vagrant/mezzanine/mezzanine_example'...
Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

dense

Плагин `dense` (появился в версии Ansible 2.3) всегда отображает только две строки вывода. Он затирает предыдущие строки, не выполняя скроллинга:

```
PLAY 1: CONFIGURE WEBSERVER WITH NGINX
task 6: testserver
```

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, поэтому приведем лишь фрагмент:

```
{
  "plays": [
    "play": {
      "id": "a45e60df-95f9-5a33-6619-000000000002"
      "name": "Configure webserver with nginx",
    },
    "tasks": [
      {
        "task": {
        "name": "install nginx",
        "id": "a45e60df-95f9-5a33-6619-000000000004"
      }
```

```
      "hosts": {
        "testserver": {
          "changed": false,
          "invocation": {
            "module_args": { ... }
          }
        }
      }
    ]
  ]
}
```

minimal

Плагин применяет минимум обработки к результатам, возвращаемым с событием Ansible. Например, если плагин `default` форматирует вывод задачи так:

```
TASK [create a logs directory] ************************************************
ok: [web]
то плагин minimal выведет:
web | SUCCESS => {
    "changed": false,
    "gid": 1000,
    "group": "vagrant",
    "mode": "0775",
    "owner": "vagrant",
    "path": "/home/vagrant/logs",
    "size": 4096,
    "state": "directory",
    "uid": 1000
}
```

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):

```
web | SUCCESS => {"changed": false, "gid": 1000, "group": "vagrant", "mode": "0775", "owner":
"vagrant", "path": "/home/vagrant/logs", "size": 4096, "state": "directory", "uid": 1000}
```

selective

Плагин `selective` отображает вывод задач, завершившихся благополучно, только если они отмечены тегом `print_action`. Сообщения об ошибках выводятся всегда.

skippy

Плагин `skippy` не отображает ничего для пропускаемых хостов. Плагин `default` выводит `skipping: [hostname]`, если хост пропускается для данной задачи, – плагин `skippy` подавляет этот вывод.

Другие плагины

Другие плагины выполняют разнообразные действия, такие как запись времени выполнения или отправка уведомлений Slack. Эти плагины перечислены в табл. 10.2.

В отличие от плагинов стандартного вывода, другие плагины могут действовать одновременно. Активировать любые плагины из этой категории можно с помощью параметра `callback_whitelist` в файле *ansible.cfg*, перечислив их через запятую, например:

```
[defaults]
callback_whitelist = mail, slack
```

Многие из этих плагинов имеют дополнительные параметры настройки, определяемые через переменные окружения.

Таблица 10.2. Другие плагины

Имя	Описание
`foreman`	Посылает уведомление в Foreman
`hipchat`	Посылает уведомление в HipChat
`jabber`	Посылает уведомление в Jabber
`junit`	Записывает данные в XML-файл в формате Junit
`log_plays`	Записывает в журнал результаты выполнения сценария для каждого хоста
`logentries`	Посылает уведомление в Logentries
`logstash`	Посылает результаты в Logstash
`mail`	Посылает электронное письмо, если выполнение задачи завершилось с ошибкой
`osx_say`	Голосовые уведомления в macOS
`profile_tasks`	Создает отчет о времени выполнения для каждой задачи
`slack`	Посылает уведомление в Slack
`timer`	Создает отчет об общем времени выполнения

foreman

Плагин `foreman` посылает уведомления в Foreman. В табл. 10.3 перечислены переменные окружения, используемые для настройки плагина.

Таблица 10.3. Переменные окружения плагина foreman

Переменная	Описание	По умолчанию
`FOREMAN_URL`	Адрес URL-сервера Foreman	`http://localhost:3000`
`FOREMAN_SSL_CERT`	Сертификат X509 для аутентификации на сервере Foreman, если используется протокол HTTPS	`/etc/foreman/client_cert.pem`
`FOREMAN_SSL_KEY`	Соответствующий приватный ключ	`/etc/foreman/client_key.pem`
`FOREMAN_SSL_VERIFY`	Необходимость проверки сертификата Foreman. Значение 1 требует проверять сертификаты SSL с использованием установленных центров сертификации. Значение 0 запрещает проверку	1

hipchat

Плагин `hipchat` посылает уведомления в HipChat. В табл. 10.4 перечислены переменные окружения, используемые для настройки плагина.

Таблица 10.4. Переменные окружения плагина hipchat

Переменная	Описание	По умолчанию
`HIPCHAT_TOKEN`	Адрес URL сервера Foreman	(Нет)
`HIPCHAT_ROOM`	Комната HipChat для отправки сообщения	`ansible`
`HIPCHAT_NAME`	Имя в HipChat для подписи сообщения	`ansible`
`HIPCHAT_NOTIFY`	Добавлять флаг уведомления к важным сообщениям	`true`

Для использования плагина `hipchat` требуется установить Python-библиотеку `prettytable`:

```
pip install prettytable
```

jabber

Плагин `jabber` посылает уведомления в Jabber. Обратите внимание, что настройки для этого плагина не имеют значений по умолчанию. Они перечислены в табл. 10.5.

Таблица 10.5. Переменные окружения плагина jabber

Переменная	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

Для использования плагина `jabber` требуется установить Python-библиотеку `xmpp`:

```
pip install git+https://github.com/ArchipelProject/xmpppy
```

junit

Плагин `junit` записывает результаты выполнения сценария в XML-файл в формате JUnit. Настраивается с помощью переменных окружения, перечисленных в табл. 10.6. Создание XML-отчетов производится в соответствии с соглашениями, перечисленными в табл. 10.7.

Таблица 10.6. Переменные окружения плагина junit

Переменная	Описание	По умолчанию
`JUNIT_OUTPUT_DIR`	Каталог для файлов отчетов	`~/.ansible.log`
`JUNIT_TASK_CLASS`	Настройки вывода: по одному классу в файле YAML	`false`

Таблица 10.7. 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

Для использования плагина junit требуется установить Python-библиотеку junit_xml:

```
pip install junit_xml
```

log_plays

Плагин log_plays записывает результаты в файлы журналов в */var/log/ansible/hosts*, по одному файлу на хост. Путь к каталогу не настраивается.

Вместо плагина log_plays можно использовать параметр настройки log_path в *ansible.cfg*. Например:

```
[defaults]
log_path = /var/log/ansible.log
```

В результате будет создаваться единый файл журнала для всех хостов, в отличие от плагина, который создает отдельные файлы для разных хостов.

logentries

Плагин logentries посылает результаты в Logentries. В табл. 10.8 перечислены переменные окружения, используемые для настройки плагина.

Таблица 10.8. Переменные окружения плагина logentries

Переменная	Описание	По умолчанию
LOGENTRIES_ANSIBLE_TOKEN	Токен сервера Logentries	(Нет)
LOGENTRIES_API	Имя хоста конечной точки Logentries	data.logentries.com
LOGENTRIES_PORT	Порт Logentries	80
LOGENTRIES_TLS_PORT	Порт TLS Logentries	443
LOGENTRIES_USE_TLS	Использовать TLS для взаимодействий с Logentries	false
LOGENTRIES_FLATTEN	Реструктурировать результаты	false

Для использования плагина logentries требуется установить Python-библиотеки certifi и flctdict:

```
pip install certifi flctdict
```

logstash

Плагин logstash записывает результаты в Logstash. В табл. 10.9 перечислены переменные окружения, используемые для настройки плагина.

Таблица 10.9. Переменные окружения плагина logstash

Переменная	Описание	По умолчанию
LOGSTASH_SERVER	Имя хоста сервера Logstash	localhost
LOGSTASH_PORT	Порт сервера Logstash	5000
LOGSTASH_TYPE	Тип сообщения	ansible

Для использования плагина logstash требуется установить Python-библиотеку python-logstash:

```
pip install python-logstash
```

mail

Плагин mail посылает электронное письмо, когда задача завершается с ошибкой. В табл. 10.10 перечислены переменные окружения, используемые для настройки плагина.

Таблица 10.10. Переменные окружения плагина mail

Переменная	Описание	По умолчанию
SMTPHOST	Имя хоста сервера SMTP	localhost

osx_say

Плагин osx_say использует программу say для вывода голосовых оповещений в macOS. Не имеет параметров настройки.

profile_tasks

Плагин profile_tasks генерирует отчет о времени выполнения отдельных задач и общего времени выполнения сценария, например:

```
Saturday 22 April 2017  20:05:51 -0700 (0:00:01.465)       0:01:02.732 ********
===============================================================================
install nginx ----------------------------------------------------------- 57.82s
Gathering Facts --------------------------------------------------------- 1.90s
restart nginx ----------------------------------------------------------- 1.47s
copy nginx config file -------------------------------------------------- 0.69s
copy index.html --------------------------------------------------------- 0.44s
enable configuration ---------------------------------------------------- 0.35s
```

Плагин также выводит информацию о времени во время 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 [install nginx] ***********************************************************
Saturday 22 April 2017  20:09:31 -0700 (0:00:01.983)       0:00:02.030 ******
ok: [testserver]
```

В табл. 10.11 перечислены переменные окружения, используемые для настройки плагина.

Таблица 10.11. Переменные окружения плагина profile-tasks

Переменная	Описание	По умолчанию
PROFILE_TASKS_SORT_ORDER	Сортировка вывода (ascending, none)	none
PROFILE_TASKS_TASK_OUTPUT_LIMIT	Максимальное количество задач в отчете или all	20

slack

Плагин slack посылает уведомления в Slack. В табл. 10.12 перечислены переменные окружения, используемые для настройки плагина.

Таблица 10.12. Переменные окружения плагина slack

Переменная	Описание	По умолчанию
SLACK_WEBHOOK_URL	Адрес URL точки входа в Slack	(Нет)
SLACK_CHANNEL	Комната Slack для отправки сообщения	#ansible
SLACK_USERNAME	Имя пользователя, отправившего сообщение	ansible
SLACK_INVOCATION	Показать детали вызова команды	20

Для использования плагина slack требуется установить Python-библиотеку prettytable:

```
pip install prettytable
```

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.

Глава 11

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, хоть и небольшое.

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 Mac OS X. 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, то есть 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 секундам.

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 11.1 показаны настройки мультиплексирования из файла *~/.ssh/config* для *myserver.example.com*.

Пример 11.1 ❖ Включение мультиплексирования в ssh/config

```
Host myserver.example.com
  ControlMaster auto
  ControlPath /tmp/%r@%h:%p
  ControlPersist 10m
```

Строка `ControlMaster auto` включает мультиплексирование SSH и сообщает SSH о необходимости создать основное соединение и управляющий сокет, если они еще не существуют.

Строка `ControlPath /tmp/%r@%h:%p` сообщает SSH, где расположить файл сокета домена Unix в файловой системе. `%h` – имя целевого хоста, `%r` – имя пользователя для удаленного доступа, и `%p` – порт. Если соединение осуществляется от имени пользователя `ubuntu`:

```
$ ssh ubuntu@myserver.example.com
```

В этом случае SSH создаст файл управляющего сокета */tmp/ubuntu@myserver.example.com:22* при первом подключении к серверу.

Строка `ControlPersist 10m` требует от SSH разорвать основное соединение, если в течение 10 минут не производилось попыток создать SSH-подключение.

Проверить состояние основного соединения можно с помощью параметра `-O check`:

```
$ ssh -O check ubuntu@myserver.example.com
```

Если основное соединение активно, эта команда вернет следующее:

```
Master running (pid=4388)
```

Вот так выглядит основной управляющий процесс в выводе команды `ps 4388`:

```
 PID  TT  STAT      TIME COMMAND
4388  ??  Ss    0:00.00 ssh: /tmp/ubuntu@myserver.example.com:22 [mux]
```

Разорвать основное соединение можно с помощью параметра `-O exit`:

```
$ ssh -O exit ubuntu@myserver.example.com
```

Больше деталей об этих настройках можно найти на странице *ssh_config* руководства *man*.

Я протестировал скорость создания SSH-соединения:

```
$ time ssh ubuntu@myserver.example.com /bin/true
```

Эта команда вернет время, которое требуется для SSH-подключения и выполнения программы `/bin/true`, которая завершается с кодом 0.

Когда я первый раз запустил ее, результат по времени выглядел так[1]:

```
0.01s user 0.01s system 2% cpu 0.913 total
```

Наибольший интерес для нас представляет общее время: `0.913 total`. Это говорит о том, что на выполнение всей команды потребовалось 0.913 секунды. 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:

```
0.00s user 0.00s system 8% cpu 0.063 total
```

Общее время сократилось до 0.063 секунды, то есть экономия составляет примерно 0.85 секунды 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[2]. Это означает, что мультиплексирование может сэкономить порядка одной или двух секунд на каждой задаче в сценарии.

Параметры мультиплексирования SSH в Ansible

В табл. 11.1 перечислены параметры мультиплексирования SSH, используемые в Ansible.

Таблица 11.1. Параметры мультиплексирования SSH в Ansible

Параметр	Значение
`ControlMaster`	`auto`
`ControlPath`	`$HOME/.ansible/cp/ansible-ssh-%h-%p-%r`
`ControlPersist`	`60s`

На практике мне 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 со-

---

[1] Формат результата может отличаться в зависимости от командной оболочки и ОС. Я использую Zsh в Mac OS X.

[2] Один из этих шагов можно оптимизировать, использовав конвейерный режим, описанный далее в этой главе.

общает, если строка в ControlPath превысит это ограничение, она просто не будет использовать мультиплексирования 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
```

Если строка ControlPath окажется слишком длинной, вы увидите сообщение об ошибке, как показано в примере 11.2.

Пример 11.2 ❖ Слишком длинная строка ControlPath

```
ControlPath
"/Users/lorinhochstein/.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 (http://bit.ly/2kKpsJI) 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.

Конвейерный режим

Вспомним, как Ansible выполняет задачу:

1. Генерирует сценарий на Python, основанный на вызываемом модуле.
2. Копирует его на хост.
3. И запускает его там.

Ansible поддерживает прием оптимизации – *конвейерный режим*, – объединяя открытие сеанса SSH с запуском сценария на Python. 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 мне нравится использовать его, поскольку он ускоряет процесс. Чтобы включить этот режим, внесите изменения в файл *ansible.cfg*, как показано в примере 11.3.

Пример 11.3 ❖ *ansible.cfg*, включение конвейерного режима

```
[defaults]
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 11.4.

Пример 11.4 ❖ Ошибка при включенном параметре `requiretty`

```
failed: [vagrant1] => {"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:

```
vagrant1 | success | rc=0 >>
/etc/sudoers.d: directory

vagrant3 | success | rc=0 >>
/etc/sudoers.d: directory

vagrant2 | success | rc=0 >>
/etc/sudoers.d: directory
```

Если каталог отсутствует, 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 шаблон файла, как показано в примере 11.5.

Пример 11.5 ❖ templates/disable-requiretty.j2

```
Defaults:{{ ansible_user }} !requiretty
```

Затем запустите сценарий, приведенный в примере 11.6, заменив `myhosts` именами ваших хостов. Не забудьте выключить конвейерный режим, прежде чем сделать это, иначе сценарий завершится с ошибкой.

Пример 11.6 ❖ disable-requiretty.yml

```
- name: do not require tty for ssh-ing user
  hosts: myhosts
  sudo: True
  tasks:
    - name: Set a sudoers file to disable tty
    template: >
      src=templates/disable-requiretty.j2
      dest=/etc/sudoers.d/disable-requiretty
      owner=root group=root mode=0440
      validate="visudo -cf %s"
```

Обратите внимание на использование `validate="visudo -cf %s"`. В разделе «Проверка достоверности файлов», в приложении А, вы узнаете, почему желательно использовать проверку при изменении файлов *sudoers*.

Кэширование фактов

Если в вашем сценарии не используются факты, 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 ваши операции используют факты, их сбор можно организовать так, что Ansible будет делать это для каждого хоста только однажды, даже если вы запустите этот же или другой сценарий для того же самого хоста.

Если кэширование фактов включено, Ansible сохранит факты в кэше, полученные после первого подключения к хостам. В последующих попытках выполнить сценарий Ansible будет извлекать факты из кэша, не обращаясь к удаленным хостам. Такое положение вещей сохраняется до истечения времени хранения кэша.

В примере 11.7 приводятся строки, которые необходимо добавить в файл *ansible.cfg* для включения кэширования фактов. Значение `fact_caching_timeout` выражается в секундах, в примере используется тайм-аут, равный 24 часам (86 400 секундам).

Как это всегда бывает с решениями, использующими кэширование, существует опасность, что кэшированные данные станут 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 кэширование фактов, убедитесь, что знаете, как часто изменяются факты, используемые вашим сценарием, и задайте соответствующее значение тайм-аута кэширования. Чтобы очистить кэш до запуска сценария, передайте параметр `--flush-cache` утилите `ansible-playbook`.

Пример 11.7 ❖ *ansible.cfg*. Включение кэширования фактов

```
[defaults]
gathering = smart
# кэш остается действительным 24 часа, измените, если необходимо
fact_caching_timeout = 86400
# Обязательно укажите реализацию кэширования 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 кэше или срок хранения кэша истек.

Если вы собираетесь использовать кэширование 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 кэше.

Необходимо явно указать тип `fact_caching` в *ansible.cfg*, иначе кэширование не будет использоваться. На момент написания книги имелись три реализации кэширования данных:

- в файлах JSON;
- Redis;
- Memcached.

Кэширование фактов в файлах JSON

Реализация кэширования 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 кэширования фактов в файлах JSON, добавьте в файл *ansible.cfg* настройки, как показано в примере 11.8.

Пример 11.8 ❖ *ansible.cfg*, включение кэширования фактов в файлах JSON

```
[defaults]
gathering = smart

# кэш остается действительным 24 часа, измените, если необходимо
fact_caching_timeout = 86400

# кэшировать 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 кэширования Ansible использует время модификации файла.

Кэширование фактов в Redis

Redis – популярное хранилище данных типа «ключ/значение», часто используемое в качестве кэша. Для кэширования 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 кэширование в Redis в файле *ansible.cfg*.

В примере 11.9 показано, какие настройки следует добавить в *ansible.cfg*, чтобы организовать кэширование в Redis.

Пример 11.9 ❖ *ansible.cfg*, кэширование фактов в Redis

```
[defaults]
gathering = smart

# кэш 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
```

[1] Может потребоваться выполнить команду `sudo` или активировать virtualenv, в зависимости от способа установки Ansible на управляющей машине.

Вы также должны установить программное обеспечение Redis и запустить его на управляющей машине. В OS X Redis можно установить с помощью диспетчера пакетов Homebrew. В Linux это можно сделать с помощью системного диспетчера пакетов.

Кэширование фактов в Memcached

Memcached – еще одно популярное хранилище данных типа «ключ/значение», которое также часто используется в качестве кэша. Для кэширования 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 кэширование в Memcached в файле *ansible.cfg*.

В примере 11.10 показано, какие настройки следует добавить в *ansible.cfg*, чтобы организовать кэширование в Memcached.

Пример 11.10 ❖ *ansible.cfg*, кэширование фактов в Memcached

```
[defaults]
gathering = smart

# кэш остается действительным 24 часа, измените, если необходимо
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 OS X Memcached можно установить с помощью диспетчера пакетов Homebrew. В Linux это можно сделать с помощью системного диспетчера пакетов.

Более полную информацию о кэшировании фактов можно найти в официальной документации (http://bit.ly/1F6BHap).

Параллелизм

Для каждой задачи Ansible устанавливает соединения параллельно 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 по умолчанию, равным 5. Изменить его можно одним из двух способов.

Можно настроить переменную среды ANSIBLE_FORKS, как это показано в примере 11.11.

Пример 11.11 ❖ Настройка ANSIBLE_FORKS

```
$ export ANSIBLE_FORKS=20
$ ansible-playbook playbook.yml
```

Можно также изменить настройки в файле конфигурации Ansible (*ansible.cfg*), определив параметр forks в секции default, как показано в примере 11.12.

Пример 11.12 ❖ *ansible.cfg*. Настройка параллелизма

```
[defaults]
forks = 20
```

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, 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 окончания выполнения первой. Это может пригодиться, например, если обе задачи выполняются очень долго и не зависят друг от друга (то есть нет нужды ждать, пока завершится первая, чтобы запустить вторую).

В примере 11.13 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 11.13 ❖ Использование async для параллельного выполнения задач

```
- name: install git
  apt: name=git update_cache=yes
  become: yes
- 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: "{{ item }}"
  with_items:
    - apt-transport-https
```

```
    - ca-certificates
    - linux-image-extra-virtual
    - software-properties-common
    - python-pip
  become: yes
- 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 Определяем эту задачу как асинхронную, и что она должна выполняться не дольше 3600 секунд. 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 бы мы указали ненулевое значение, Ansible не смогла бы перейти к следующей задаче. Вместо этого она периодически опрашивала бы 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 мы используем 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.

Теперь вы знаете, как настроить мультиплексирование SSH, конвейерный режим, кэширование фактов, а также параллельное и асинхронное выполнения задач, чтобы ускорить выполнение сценария. Далее мы обсудим написание собственных модулей.

Глава 12

Собственные модули

Иногда желательно выполнить задачу, слишком сложную для модулей `command` или `shell`. И не существует готовых модулей для ее выполнения. В этом случае можно написать модуль самостоятельно.

В прошлом я писал свои модули для получения публичного IP-адреса, когда управляющая машина находилась за шлюзом, выполняющим преобразование сетевых адресов (Network Address Translation, NAT), и требовалось создавать базы данных в окружении OpenStack. Я думал о написании своего модуля для создания самоподписанного сертификата, хотя так и не занялся этим.

Свои модули могут также пригодиться для взаимодействий со сторонними службами REST API. Например, GitHub предлагает то, что они называют Releases, позволяющее сохранять в репозитории двоичные ресурсы. Если для развертывания проекта требуется загрузить двоичный ресурс, хранящийся в частном репозитории GitHub, это станет отличным поводом написать свой модуль.

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 данной главе, является упрощенной версией модуля `wait_for`.

Использование модуля script вместо написания своего модуля

Помните, как в примере 6.17 мы использовали модуль `script` для запуска своих сценариев на удаленных хостах? Иногда действительно проще использовать модуль `script`, чем писать свой, полноценный модуль Ansible.

Я храню 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, как в примере 12.1.

Пример 12.1 ❖ *can_reach.sh*

```
#!/bin/bash
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.example.com 80 1
```

Имейте в виду, что сценарий будет запускаться на удаленных хостах так же, как модули Ansible. Вследствие этого любые программы, необходимые сценарию, должны быть установлены на удаленных хостах заранее. Например, можно написать сценарий на Ruby, если Ruby установлен на удаленных хостах, и в первой строке указать интерпретатор Ruby:

```
#!/usr/bin/ruby
```

can_reach как модуль

Теперь реализуем `can_reach` в виде полноценного модуля Ansible, который можно вызвать так:

```
- name: check if host can reach the database server
  can_reach: host=db.example.com port=5432 timeout=1
```

Так можно проверить доступность порта 5432 на хосте *db.example.com*. Если соединение установить невозможно, через секунду будет зафиксирована ошибка превышения тайм-аута.

Мы будем пользоваться этим примером на протяжении всей главы.

Где хранить свои модули

Поиск модулей производится в каталоге *library*, находящемся рядом со сценарием Ansible. В нашем примере сценарии хранятся в каталоге *playbooks*, поэтому свой модуль мы сохраним в файле *playbooks/library/can_reach*.

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), 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 `ubuntu`, Ansible сохранит файл: */home/ubuntu/.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: */home/ubuntu/.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
```

файл аргументов в этом случае 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 файл аргументов 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, Ansible выполнит эквивалент следующей команды (заменив */path/to/* действительным путем к каталогу):

```
/path/to/can_reach
```

Если модуль написан на другом языке, 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
```

тогда Ansible выполнит такую команду:

```
/bin/bash /path/to/can_reach /path/to/arguments
```

Но это только приближенный эквивалент. На самом деле Ansible выполнит такую команду:

```
/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`.

Ожидаемый вывод

Ansible ожидает, что модуль выведет результат в формате JSON. Например:

```
{'changed': false, 'failed': true, 'msg': 'could not reach the host'}
```

До версии 1.8 Ansible поддерживала вывод информации в формате условных обозначений, также известный как *baby JSON*, который выглядел как `key=value`. Поддержка этого формата была прекращена в версии 1.8. Как вы увидите ниже, если модуль написан на Python, Ansible предоставляет вспомогательные методы, облегчающие вывод информации в JSON.

Ожидаемые выходные переменные

Модуль может выводить любые переменные, однако Ansible определяет специальные правила для переменных возврата:

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, 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`, 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"}
```

Ansible выведет:

```
failed: [vagrant1] => {"failed": true}
msg: could not reach www.example.com:81
```

Реализация модулей на Python

Для модулей на Python Ansible предоставляет класс `AnsibleModule`, упрощающий следующие действия:

- анализ входной информации;
- вывод результатов в формате JSON;
- вызов сторонних программ.

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 12.2).

Пример 12.2 ❖ *can_reach*

```
#!/usr/bin/python
from ansible.module_utils.basic import AnsibleModule ❶

def can_reach(module, host, port, timeout):
    nc_path = module.get_bin_path('nc', required=True) ❷
    args = [nc_path, "-z", "-w", str(timeout),
            host, str(port)]
    (rc, stdout, stderr) = module.run_command(args) ❸
    return rc == 0

def main():
    module = AnsibleModule( ❹
        argument_spec=dict( ❺
            host=dict(required=True), ❻
            port=dict(required=True, type='int'),
            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):
    module.exit_json(changed=False)
else:
    msg = "Could not reach %s:%s" % (host, port) ⓬
    module.fail_json(msg=msg)

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 Определяет, что модуль поддерживает режим проверки.
❾ Определение запуска модуля в режиме проверки.
❿ Успешное завершение, передает возвращаемое значение.
⓫ Извлекает аргумент.
⓬ Завершается с ошибкой, 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 секунды.

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 нашем примере 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

Начиная с версии Ansible 2.1.0 модули на хосты стали передаваться в файле ZIP, включающем также вспомогательные файлы для импорта. Как следствие теперь можно явно импортировать классы, например:

```
from ansible.module_utils.basic import AnsibleModule
```

До версии Ansible 2.1.0 инструкция `import` в модуле Ansible в действительности была псевдоинструкцией импорта. В предыдущих версиях Ansible копировала на удаленный хост единственный файл с кодом на Python. Ansible имитировала поведение традиционной инструкции `import` включением импортируемого кода непосредственно в генерируемый файл на Python (примерно так, как это делает инструкция `#include` в C или C+\+). Поскольку она вела себя иначе, чем традиционная инструкция `import`, при попытке явно импортировать класс отладка модулей Ansible превращалась порой в очень сложную задачу. Вы должны были использовать инструкцию импорта с шаблонным символом и вставлять ее в конец файла, непосредственно перед началом главного блока:

```
...
from ansible.module_utils.basic import *
if __name__ == "__main__":
    main()
```

Свойства аргументов

Каждый аргумент модуля Ansible имеет несколько свойств, перечисленных в табл. 12.1.

Таблица 12.1. Свойства аргументов

Свойство	Описание
`required`	Если `True`, аргумент считается обязательным
`default`	Значение по умолчанию для необязательного аргумента
`choices`	Список допустимых значений для аргумента
`aliases`	Другие имена, которые можно использовать как псевдонимы этого аргумента
`type`	Тип аргумента. Допустимые значения: `'str'`, `'list'`, `'dict'`, `'bool'`, `'int'`, `'float'`

required

Свойство `required` – единственное, которое всегда нужно определять. 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
```

аргумент `module.params["timeout"]` автоматически получит значение `3`.

choices

Свойство `choices` позволяет ограничить значения аргумента предопределенным списком, как аргумент `distros` в следующем примере:

```
distro=dict(required=True, choices=['ubuntu', 'centos', 'fedora'])
```

Если пользователь попробует передать в аргументе значение, отсутствующее в списке, например:

```
distro=suse
```

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.

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, ссылка module.params['port'] вернула бы строку, а не целое число.

Списки разделяются запятой. Например, если представить, что у нас есть модуль foo с аргументом colors, принимающим список:

```
colors=dict(required=True, type='list')
```

мы должны будем передавать в нем список, как показано ниже:

```
foo: colors=red,green,blue
```

Для передачи словарей можно использовать нотацию пар key=value, разделенных запятыми, либо формат 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 передачи этих типов аргументов описывается в разделе «Короткое отступление: составные аргументы задач».

AnsibleModule: параметры метода инициализатора

Метод-инициализатор класса `AnsibleModule` принимает несколько параметров. Единственным обязательным параметром является `argument_spec`.

Таблица 12.2. Аргументы инициализатора AnsibleModule

Параметр	По умолчанию	Описание
`argument_spec`	(Нет)	Словарь с информацией об аргументах
`bypass_checks`	`False`	Если `True`, не проверяет никаких ограничений для параметров
`no_log`	`False`	Если `True`, не журналирует поведения этого модуля
`check_invalid_arguments`	`True`	Если `True`, 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`, 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:

```
Sep 28 02:31:47 vagrant-ubuntu-trusty-64 ansible-ping: Invoked with data=None
Sep 28 02:32:18 vagrant-ubuntu-trusty-64 ansible-apt: Invoked with dpkg_options=
force-confdef,force-confold upgrade=None force=False name=nginx package=['nginx'
] purge=False state=installed update_cache=True default_release=None install_rec
ommends=True deb=None cache_valid_time=None Sep 28 02:33:01 vagrant-ubuntu-trust
y-64 ansible-file: Invoked with src=None
```

```
original_basename=None directory_mode=None force=False remote_src=None selevel=N
one seuser=None recurse=False serole=None content=None delimiter=None state=dire
ctory diff_peek=None mode=None regexp=None owner=None group=None path=/etc/nginx
/ssl backup=None validate=None setype=None
Sep 28 02:33:01 vagrant-ubuntu-trusty-64 ansible-copy: Invoked with src=/home/va
grant/.ansible/tmp/ansible-tmp-1411871581.19-43362494744716/source directory_mod
e=None force=True remote_src=None dest=/etc/nginx/ssl/nginx.key selevel=None seu
ser=None serole=None group=None content=NOT_LOGGING_PARAMETER setype=None origin
al_basename=nginx.key delimiter=None mode=0600 owner=root regexp=None validate=N
one backup=False
Sep 28 02:33:01 vagrant-ubuntu-trusty-64 ansible-copy: Invoked with src=/home/va
grant/.ansible/tmp/ansible-tmp-1411871581.31-95111161791436/source directory_mod
e=None force=True remote_src=None dest=/etc/nginx/ssl/nginx.crt selevel=None seu
ser=None serole=None group=None content=NOT_LOGGING_PARAMETER setype=None origin
al_basename=nginx.crt delimiter=None mode=None owner=None regexp=None validate=N
one backup=False
```

Если модуль принимает конфиденциальную информацию в аргументах, предпочтительнее отключить журналирование. Для отключения записи в syslog передайте параметр `no_log=True` в инициализатор `AnsibleModule`.

check_invalid_arguments

По умолчанию Ansible проверяет допустимость всех аргументов, передаваемых пользователем. Эту проверку можно отключить, передав параметр `check_invalid_arguments=False` в инициализатор `AnsibleModule`.

mutually_exclusive

Параметр `mutually_exclusive` содержит список аргументов, которые нельзя одновременно передавать в вызов модуля. Например, модуль `lineinfile` позволяет добавить строку в файл. Ему можно передать аргумент `insertbefore` со строкой для вставки перед указанной или аргумент `insertbefore` 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=ubuntu 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=ubuntu 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.

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 equired_one_of.

Все эти проверки можно отменить, 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 следующие аргументы.

Таблица 12.3. Аргументы run_command

Аргумент	Тип	Значение 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

Окончание табл. 12.3

Аргумент	Тип	Значение по умолчанию	Описание
`executable`	Строка (путь к программе)	(Нет)	Передает как аргумент `executable` в вызов `subprocess.Popen`
`data`	Строка	(Нет)	Посылается в стандартный ввод дочернего процесса
`binary_data`	Логический	False	Если `False` и присутствует `data`, тогда 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 директории)	(Нет)	Если определена, Ansible перейдет в этот каталог перед запуском
`use_unsafe_shell`	Логический	False	См. следующий раздел

Если `args` передается как список (см. пример 12.3), тогда Ansible вызовет `subprocess.Popen` с параметром `shell=False`.

Пример 12.3 ❖ Передача `args` со списком

```
module = AnsibleModule(...)
...
module.run_command(['/usr/local/bin/myprog', '-i', 'myarg'])
```

Если в `args` передать строку, как показано в примере 12.4, поведение в этом случае будет зависеть от значения `use_unsafe_shell`. Если `use_unsafe_shell=False`, Ansible разобьет `args` на список и вызовет `subprocess.Popen` с параметром `shell=False`. Если `use_unsafe_shell=True`, Ansible передаст `args` в `subprocess.Popen` в виде строки с `shell=True`[1].

Пример 12.4 ❖ 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 электронной документации: http://bit.ly/1F72tiU.

Модуль должен явно поддерживать режим проверки. Если вы собираетесь написать свой модуль, 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 12.5.

Пример 12.5 ❖ Уведомление Ansible о поддержке режима проверки

```
module = AnsibleModule(
    argument_spec=dict(...),
    supports_check_mode=True)
```

Модуль должен определить режим проверкой значения атрибута `check_mode`[1] объекта `AnsibleModule`, как показано в примере 12.6, и вызвать метод `exit_json` или `fail_json`, как обычно.

Пример 12.6 ❖ 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.

В примере 12.7 приводится раздел с документацией для модуля `can_reach`.

Пример 12.7 ❖ Пример модуля с документацией

```
DOCUMENTATION = '''
---
module: can_reach
short_description: Проверяет доступность сервера
description:
  - Проверяет возможность подключения к удаленному серверу
version_added: "1.8"
```

[1] Уф! Слишком много проверок.

```
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
  flavor:
    description:
      - Это искусственный параметр, чтобы показать, какой выбор был сделан.
    required: false
    choices: ["chocolate", "vanilla", "strawberry"]
    aliases: ["flavor"]
    default: chocolate
  requirements: [netcat]
  author: Lorin Hochstein
  notes:
    - Это просто пример, демонстрирующий, как писать модули.
    - Возможно, вы предпочтете использовать встроенный модуль M(wait_for).
'''

EXAMPLES = '''
# Проверка доступности хоста через ssh с тайм-аутом по умолчанию
- can_reach: host=myhost.example.com port=22

# Проверка доступности сервера postgres с нестандартным тайм-аутом
- can_reach: host=db.example.com port=5432 timeout=1
'''
```

В документации допускается использовать рудиментарную разметку. В табл. 12.4 описывается синтаксис разметки, поддерживаемой инструментом вывода документации, а также советы по ее использованию.

Таблица 12.4. 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 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 --recursive
```

Настройте переменные окружения, чтобы было можно вызвать модуль:

```
$ source ansible/hacking/env-setup
```

Вызовите модуль:

```
$ ansible/hacking/test-module -m /path/to/can_reach -a "host=example.com port=81"
```

Занимаясь отладкой, вы можете столкнуться с такими ошибками:

```
ImportError: No module named yaml
ImportError: No module named jinja2.exceptions
```

В этом случае установите эти недостающие зависимости:

```
pip install pyYAML jinja2
```

Поскольку на `example.com` нет службы, обслуживающей порт 81, модуль завершится с ошибкой и вернет сообщение:

```
* including generated source, if any, saving to:
/Users/lorin/.ansible_module_generated
* ansiballz module detected; extracted module source to:
/Users/lorin/debug_dir
********************************
RAW OUTPUT
{"msg": "Could not reach example.com:81", "failed": true, "invocation":
{"module_args": {"host": "example.com", "port": 81, "timeout": 3}}}

********************************
PARSED OUTPUT
{
    "failed": true,
    "invocation": {
        "module_args": {
            "host": "example.com",
            "port": 81,
            "timeout": 3
        }
    },
    "msg": "Could not reach example.com:81"
}
```

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 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": {"host": "example.com", \
    "_ansible_selinux_special_fs": ["fuse", "nfs", "vboxsf", "ramfs"], \
    "port": "81"}}'
```

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. Однако при желании модули можно писать на других языках. 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. Или вам просто нравится писать сценарии на Ruby.

В этом разделе мы рассмотрим пример создания модуля в виде сценария на Bash. Он будет очень похож на реализацию в примере 12.1. Главным отличием являются анализ входных аргументов и генерация вывода, который ожидает получить Ansible.

Я использую формат JSON для передачи входных аргументов и инструмент `jq` (https://stedolan.github.io/jq/) для парсинга JSON в командной строке. Это значит, что для запуска модуля на хосте придется установить `jq`. В примере 12.8 приводится полный листинг модуля на Bash.

Пример 12.8. ❖ Модуль *can_reach* на Bash

```
#!/bin/bash
# WANT_JSON

# Чтение переменных из файла
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
nc -z -w $timeout $host $port

# Вернуть результат
```

```
if [ $? -eq 0 ]; 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.

Модули 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, прочитайте статью «Shell scripts as Ansible modules» («Сценарии на языке оболочки в качестве модулей Ansible») по адресу: *http://bit.ly/1F789tb*, написанную Яном-Питом Менсом (Jan-Piet Mens).

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.

Тогда, когда Ansible будет запускать модуль на хосте *fileserver.example.com*, она использует */usr/local/bin/bash* вместо */bin/bash*.

Выбор интерпретатора Ansible определяет поиском символов `#!` и просмотром базового имени первого элемента. В нашем примере Ansible найдет строку:

```
#!/bin/bash
```

извлечет из */bin/bash* базовое имя, то есть *bash*. Затем использует переменную `ansible_bash_interpreter`, если она задана пользователем.

Учитывая, как Ansible определяет интерпретатор, если вы в строке «she-bang» укажете вызов команды */usr/bin/env*, например:

```
#!/usr/bin/env bash
```

Ansible ошибочно определит интерпретатор как `env`, потому что будет анализировать путь */usr/bin/env*.

Поэтому, чтобы не попасть впросак, не вызывайте `env` в строке «she-bang», точно указывайте путь к интерпретатору и переопределяйте его с помощью переменной `ansible_bash_interpreter` (или ее аналогом), если это необходимо.

Примеры модулей

Лучшим способом научиться писать модули для Anisble является изучение исходного кода модулей, поставляемых с Ansible. Вы найдете их в GitHub (https://github.com/ansible/ansible/tree/devel/lib/ansible/modules).

В этой главе мы рассмотрели, как писать модули на Python и на других языках, а также как можно избежать написания собственных модулей с использованием модуля `script`. Если вы все-таки беретесь за написание модуля, я рекомендую постараться включить его в основной проект Ansible.

Глава 13

Vagrant

Vagrant является отличной средой для тестирования сценариев Ansible. Именно поэтому я использую ее на протяжении всей книги, а также часто тестирую в ней свои собственные сценарии. Vagrant подходит не только для тестирования сценариев управления конфигурациями. Изначально это программное обеспечение разрабатывалось для многократного создания окружений разработки. Если вам доводилось присоединяться к новой команде разработчиков и тратить несколько дней на выяснение, какое ПО следует установить на свой ноутбук, чтобы запустить внутреннюю версию разрабатываемого продукта, значит, вы испытали ту боль, которую Vagrant может облегчить. Сценарии Ansible – прекрасный способ определения конфигурации машины Vagrant, помогающий новичкам в вашей команде взяться за дело незамедлительно.

В Vagrant встроена определенная поддержка Ansible, преимуществами которой мы еще не пользовались. В этой главе мы рассмотрим эти дополнительные возможности, помогающие настраивать машины Vagrant.

Полное описание Vagrant не является целью данной книги. За дополнительной информацией обращайтесь к книге «Vagrant: Up and Running» Митчела Хашимото (Mitchell Hashimoto), создателя Vagrant.

Полезные параметры настройки Vagrant

Vagrant предлагает большое количество параметров настройки виртуальных машин, но две из них, на мой взгляд, являются особенно полезными при использовании для тестирования – установка IP-адреса и настройка перенаправления агента.

Перенаправление портов и приватные IP-адреса

При создании нового файла *Vagrantfile* командой `vagrant init` сетевые настройки по умолчанию позволяют получить доступ к виртуальной машине Vagrant только через порт SSH, перенаправленный с локального хоста. Первой машине Vagrant назначается порт 2222, каждой последующей – порт с номером на единицу больше. Как следствие единственный способ получить доступ к машине

Vagrant с сетевыми настройками по умолчанию – установка SSH-соединения с портом 2222 на локальный хост `localhost`. Vagrant перенаправляет это соединение на порт 22 в машине Vagrant.

Сетевые настройки по умолчанию плохо подходят для тестирования веб-приложений, поскольку они ожидают соединений на другом порте, к которому у нас нет доступа.

Есть два способа решить эту проблему. Первый – дать Vagrant команду настроить перенаправление еще одного порта. Например, если веб-приложение принимает соединения на порте 80 внутри машины Vagrant, можно перенаправить порт 8000 на локальной машине в порт 80 на машине Vagrant. В примере 13.1 показано, как настроить перенаправление порта в файле *Vagrantfile*.

Пример 13.1 ❖ Перенаправление локального порта 8000 в порт 80 на машине Vagrant

```
# Vagrantfile
VAGRANTFILE_API_VERSION = "2"

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

  config.vm.network :forwarded_port, host: 8000, guest: 80
end
```

Перенаправление порта решает проблему, но мне кажется, полезнее будет присвоить машине Vagrant собственный IP-адрес. В этом случае взаимодействие будет больше походить на связь с настоящим удаленным сервером – можно установить соединение напрямую с портом 80 по IP-адресу машины вместо порта 8000 локальной машины.

Самый простой способ – присвоить машине приватный IP-адрес. В примере 13.2 показано, как присвоить IP-адрес *192.168.33.10* машине, отредактировав файл *Vagrantfile*.

Пример 13.2 ❖ Присваивание приватного IP-адреса машине Vagrant

```
# Vagrantfile
VAGRANTFILE_API_VERSION = "2"

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

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

Если теперь на машине Vagrant запустить веб-сервер, обслуживающий порт 80, к нему можно получить доступ по ссылке: *http://192.168.33.10*.

Данная конфигурация использует *частную сеть* Vagrant. Это значит, что виртуальная машина будет доступна только с машины, где действует Vagrant. Подключиться к этому IP-адресу с другой физической машины невозможно, даже находящейся в той же сети, что и машина, где выполняется Vagrant. Но разные машины Vagrant могут соединяться друг с другом.

За более полной информацией о разных параметрах настройки сети обращайтесь к документации по Vagrant.

Перенаправление агента

Если вы извлекаете файлы из репозитория Git по SSH и вам нужно использовать перенаправление агента, настройте машину Vagrant так, чтобы Vagrant включала перенаправление агента при соединении с ним через SSH. В примере 13.3 показано, как это сделать. Дополнительную информацию о перенаправлении вы найдете в приложении А.

Пример 13.3 ❖ Включение перенаправления агента

```
# Vagrantfile
VAGRANTFILE_API_VERSION = "2"

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

  config.ssh.forward_agent = true
end
```

Сценарий наполнения Ansible

В Vagrant существует понятие *сценариев наполнения* (provisioners). Сценарии наполнения являются внешним инструментом, который используется системой Vagrant для настройки виртуальной машины в момент запуска. В дополнение к Ansible Vagrant может также взаимодействовать со сценариями оболочки, Chef, Puppet, CFengine и даже Docker.

В примере 13.4 показан файл *Vagrantfile*, использующий Ansible для наполнения виртуальной машины, в данном случае с помощью сценария *playbook.yml*.

Пример 13.4 ❖ *Vagrantfile*

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"

  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "playbook.yml"
  end
end
```

Когда выполняется сценарий наполнения

Когда в первый раз запускается команда `vagrant up`, Vagrant выполнит сценарий наполнения и сделает запись, что он запускался. После остановки и повторного запуска виртуальной машины Vagrant вспомнит, что сценарий наполнения уже выполнялся, и не будет повторно запускать его.

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

```
$ vagrant provision
```

Можно 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
```

Реестр, генерируемый системой Vagrant

Во время работы Vagrant создает файл реестра Ansible с именем *.vagrant/provisioners/ansible/inventory/vagrant_ansible_inventory*. В примере 13.5 показано содержимое этого в нашем случае.

Пример 13.5 ❖ `vagrant_ansible_inventory`

```
# Generated by Vagrant

default ansible_ssh_host=127.0.0.1 ansible_ssh_port=2202
```

Обратите внимание, что в качестве имени хоста используется имя `default`. Поэтому, когда будете писать сценарии Ansible для использования в качестве сценариев наполнения Vagrant, используйте объявления `hosts: default` или `hosts: all`.

Еще интереснее случай, когда имеется окружение с большим количеством машин Vagrant, в котором *Vagrantfile* определяет несколько виртуальных машин. Взгляните на пример 13.6.

Пример 13.6 ❖ *Vagrantfile* (несколько машин)

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/trusty64"
    vagrant1.vm.provision "ansible" do |ansible|
      ansible.playbook = "playbook.yml"
    end
  end
  config.vm.define "vagrant2" do |vagrant2|
    vagrant2.vm.box = "ubuntu/trusty64"
    vagrant2.vm.provision "ansible" do |ansible|
      ansible.playbook = "playbook.yml"
    end
  end
  config.vm.define "vagrant3" do |vagrant3|
```

```
    vagrant3.vm.box = "ubuntu/trusty64"
    vagrant3.vm.provision "ansible" do |ansible|
      ansible.playbook = "playbook.yml"
    end
  end
end
```

Сформированный файл реестра будет выглядеть, как показано в примере 13.7. Обратите внимание, что псевдонимы (`vagrant1`, `vagrant2`, `vagrant3`) соответствуют именам машин в файле *Vagrantfile*.

Пример 13.7 ❖ `vagrant_ansible_inventory` (несколько машин)

```
# Generated by Vagrant

vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201
```

Наполнение нескольких машин одновременно

В примере 13.6 показано, что Vagrant вызывает `ansible-playbook` для каждой виртуальной машины с параметром `--limit`, чтобы сценарий наполнения запускался каждый раз только для одной машины.

Однако при таком подходе не используется возможность Ansible выполнять задачи на хостах параллельно. Эту проблему можно решить, настроив в *Vagrantfile* запуск сценария наполнения после запуска последней виртуальной машины и потребовав от Vagrant не передавать Ansible параметр `--limit`, как показано в примере 13.8.

Пример 13.8 ❖ *Vagrantfile* (несколько машин с параллельным выполнением сценария наполнения)

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Для всех машин используется один и тот же ключ
  config.ssh.insert_key = false

  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/trusty64"
  end
  config.vm.define "vagrant2" do |vagrant2|
    vagrant2.vm.box = "ubuntu/trusty64"
  end
  config.vm.define "vagrant3" do |vagrant3|
    vagrant3.vm.box = "ubuntu/trusty64"
    vagrant3.vm.provision "ansible" do |ansible|
      ansible.limit = 'all'
      ansible.playbook = "playbook.yml"
    end
  end
end
```

Теперь при первом запуске команды `vagrant up` сценарий наполнения будет запущен только после запуска всех трех виртуальных машин.

С точки зрения Vagrant только последняя виртуальная машина, `vagrant3`, выполняет сценарий наполнения. Поэтому выполнение `vagrant provision vagrant1` или `vagrant provision vagrant2` не даст никакого результата.

Как это уже обсуждалось в разделе «Вводная часть: несколько машин Vagrant» в главе 3, Vagrant 1.7+ по умолчанию использует разные ключи SSH для разных хостов. Если требуется организовать параллельное выполнение сценария наполнения, необходимо настроить виртуальные машины так, чтобы все они использовали один и тот же ключ SSH. Именно поэтому пример 13.8 содержит строку

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

Определение групп

Иногда полезно объединить виртуальные машины Vagrant в группы, особенно при использовании сценариев, ссылающихся на существующие группы. В примере 13.9 показано, как поместить `vagrant1` в группу `web`, `vagrant2` в группу `task` и `vagrant3` в группу `redis`.

Пример 13.9 ❖ *Vagrantfile* (несколько машин с группами)

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Для всех машин используется один и тот же ключ
  config.ssh.insert_key = false

  config.vm.define "vagrant1" do |vagrant1|
    vagrant1.vm.box = "ubuntu/trusty64"
  end
  config.vm.define "vagrant2" do |vagrant2|
    vagrant2.vm.box = "ubuntu/trusty64"
  end
  config.vm.define "vagrant3" do |vagrant3|
    vagrant3.vm.box = "ubuntu/trusty64"
    vagrant3.vm.provision "ansible" do |ansible|
      ansible.limit = 'all'
      ansible.playbook = "playbook.yml"
      ansible.groups = {
        "web" => ["vagrant1"],
        "task" => ["vagrant2"],
        "redis" => ["vagrant3"]
      }
    end
  end
end
```

В примере 13.10 представлен окончательный вариант файла реестра, сгенерированного системой Vagrant.

Пример 13.10 ❖ `vagrant_ansible_inventory` (несколько машин, с группами)

```
# Generated by Vagrant

vagrant1 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
vagrant2 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2200
vagrant3 ansible_ssh_host=127.0.0.1 ansible_ssh_port=2201

[web]
vagrant1

[task]
vagrant2

[redis]
vagrant3
```

Локальные сценарии наполнения

Начиная с версии 1.8 Vagrant позволяет также запускать Ansible из гостевой системы. Этот режим может пригодиться, когда нежелательно устанавливать Ansible на хост-машину. Если Ansible отсутствует на гостевой машине, Vagrant попытается установить ее с помощью `pip`, однако это поведение можно изменить.

Vagrant ищет сценарии Ansible в каталоге */vagrant* гостевой системы. По умолчанию Vagrant монтирует каталог хоста, содержащий *Vagrantfile*, в каталог */vagrant*, то есть фактически Vagrant просматривает тот же каталог, как при использовании обычного сценария наполнения Ansible.

Чтобы использовать локальный сценарий наполнения Ansible, определите режим наполнения `ansible_local`, как показано в примере 13.11.

Пример 13.11 ❖ *Vagrantfile* (локальный сценарий наполнения Ansible)

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.provision "ansible_local" do |ansible|
    ansible.playbook = "playbook.yml"
  end
end
```

Эта глава оказалась коротким, но, я надеюсь, полезным обзором эффективного использования Vagrant и Ansible. Сценарии наполнения Ansible в Vagrant поддерживают многие другие параметры, которые не были упомянуты в данной главе. Дополнительную информацию можно найти в официальной документации к Vagrant (http://bit.ly/1F7ekxp).

Глава 14

Amazon EC2

Ansible поддерживает возможность работы с облачными услугами типа «инфраструктура как услуга» (Infrastructure-as-a-Service, IaaS). Основное внимание в этой главе мы будем уделять Amazon EC2, поскольку это самое популярное IaaS-облако, к тому же его я знаю лучше всего. Однако большинство идей можно применить к другим облакам, которые поддерживает Ansible.

Ansible включает два механизма поддержки EC2:

- плагин для автоматического заполнения реестра Ansible вместо определения серверов вручную;
- модули для выполнения действий с EC2, такие как создание новых серверов.

В этой главе мы рассмотрим оба механизма: и плагин динамической инвентаризации EC2, и модули поддержки EC2.

На момент написания этих строк в Ansible имелась почти сотня модулей поддержки EC2, а также других инструментов, предлагаемых Amazon Web Services (AWS). Мы не можем в полной мере охватить их все в этой книге, поэтому сосредоточимся только на самых основных.

Что такое облако IaaS?

Вероятно, вы столько раз сталкивались с упоминанием *облака* в технической прессе, что уже устали от этого словечка[1]. Я постараюсь быть педантичным в своем определении облака типа «инфраструктура как услуга» (IaaS).

Для начала приведу пример типичного взаимодействия пользователя с IaaS-облаком:

Пользователь

Мне необходимо пять новых серверов на Ubuntu 16.04, каждый из которых оснащен двумя CPU, 4 Гбайт оперативной памяти и 100 Гбайт дисковой памяти.

---

[1] Национальный институт стандартов и технологий США (NIST) дал хорошее определение облачным вычислениям в работе Питера Мелла (Peter Mell) и Тимоти Гранца (Timothy Grance) «The NIST Definition of Cloud Computing». NIST Special Publication 800–145, 2011.

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.

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://console.aws.amazon.com), а интерфейс командной строки (неоригинально) – интерфейсом командной строки AWS (http://aws.amazon.com/cli/). Информацию о REST API можно найти на сайте Amazon по ссылке 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. Облака SoftLayer и Rackspace, например, предлагают физические серверы; облака Amazon Elastic Compute Cloud, Google Compute Engine и Joyent предлагают контейнеры.

Большинство IaaS-облаков позволяет вам делать большее, нежели запускать и останавливать серверы. В частности, как правило, они позволяют определять хранилища так, чтобы вы могли подключать и отключать диски от своих серверов. Этот тип хранилища обычно называется *блочное хранилище*. Они также предоставляют возможность определить свою топологию сети, описывающую соединения между вашими серверами, а еще задать правила брандмауэра, ограничивающего доступ к ним.

Amazon EC2 является самым популярным поставщиком публичных облаков типа IaaS, но наряду с ним существует ряд других облаков того же типа. Кроме EC2, система Ansible включает также поддержку Microsoft Azure, Digital Ocean, Google Compute Engine, Linode и Rackspace, а еще облаков, построенных с использованием oVirt, OpenStack, CloudStack и VMWare vSphere.

Терминология

EC2 использует много разных понятий. Я планирую пояснять их по мере их появления в тексте, однако два из них мне хотелось бы объяснить заранее.

Экземпляр

В документации EC2 используется понятие *экземпляра* (instance) для обозначения виртуальной машины. В данной главе я использую именно этот термин. Имейте в виду, что понятию экземпляра в EC2 соответствует понятие хоста в Ansible.

В документации EC2 (http://amzn.to/1Fw5S8l) используются взаимозаменяемые понятия *создания экземпляров* и *запуска экземпляров* для описания процесса создания новых экземпляров. Но понятие *перезапуска экземпляров* означает нечто совершенно другое – перезапуск экземпляра, который ранее был приостановлен.

Образ машины Amazon

Образ машины Amazon (Amazon Machine Image, AMI) – это образ виртуальной машины с файловой системой и установленной операционной системой. При создании экземпляра EC2 предлагается выбрать операционную систему, которая будет запущена в образе, указав на тот или иной образ AMI.

Каждому образу AMI присвоена своя строка идентификации, называемая *AMI ID*. Она начинается с приставки `ami-`, за которой следуют восемь шестнадцатеричных символов. Например, `ami-12345abc`.

Теги

EC2 позволяет отмечать экземпляры[1] комментариями с метаданными. Они называются *тегами*. Тег – это пара строк типа ключ/значение. Например, можно снабдить свой экземпляр такими тегами:

```
Name=Staging database
env=staging
type=database
```

Если вам приходилось задавать имя экземпляру EC2 в консоли управления AWS, то вы, сами того не осознавая, уже использовали теги. Имена экземпляров в EC2 реализованы как тег с ключом `Name`, значением которого является имя, присвоенное экземпляру. В остальном тег `Name` ничем не отличается от других тегов, и вы можете настроить консоль управления так, чтобы она выводила значения всех остальных тегов в дополнение к тегу `Name`.

Теги не обязательно должны быть уникальными. Можно создать 100 экземпляров с одним и тем же тегом. Модули Ansible для поддержки EC2 широко ис-

[1] Теги можно добавлять не только к экземплярам, но и к другим объектам, таким как образы AMI, тома и группы безопасности.

пользуют теги для идентификации ресурсов и обеспечения идемпотентности, в чем вы не раз убедитесь в этой главе.

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.

При вызове модулей поддержки EC2 эти строки можно передать как аргументы. Для плагина динамической инвентаризации учетные данные можно определить в файле *ec2.ini* (обсуждается в следующем разделе). Однако модули EC2 и плагин динамической инвентаризации позволяют передавать учетные данные в переменных окружения. Также можно использовать IAM-роли, если ваша управляющая машина сама является экземпляром Amazon EC2. Этот случай рассмотрен в приложении B.

Переменные окружения

Учетные данные EC2 модулям Ansible можно передавать не только через аргументы, но и через переменные окружения. В примере 14.1 показано, как определить такие переменные.

Пример 14.1 ❖ 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=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Не все модули EC2 в Ansible поддерживают переменную окружения `AWS_REGION`, поэтому я рекомендую всегда передавать регион EC2 явно, как аргумент. Во всех примерах в этой главе регион передается как аргумент.

Я рекомендую использовать переменные окружения, так как они позволяют применять модули EC2 и плагины инвентаризации без сохранения учетных данных в файлах Ansible. Я помещаю их в файл с именем, начинающимся с точки, который выполняется при запуске сеанса. Я пользуюсь Zsh, поэтому использую для этого файл *~/.zshrc*. Если вы пользуетесь Bash, можете поместить их в файл *~/.profile*[1]. Если вы используете другую командную оболочку, отличную от Bash или Zsh, возможно, вы уже знаете, какой файл изменить для определения этих переменных окружения.

После настройки переменных окружения с учетными данными можно запускать модули EC2 Ansible на управляющей машине, а также использовать плагины инвентаризации.

Файлы конфигурации

В качестве альтернативы переменным окружения можно использовать конфигурационный файл. Как обсуждается в следующем разделе, Ansible использует библиотеку Python Boto для поддержки соглашений Boto о хранении учетных данных в файле конфигурации Boto. Я не буду рассматривать этот формат здесь и за дополнительной информацией отсылаю вас к документации по Boto (http://bit.ly/1Fw66MM).

Необходимое условие: библиотека Python Boto

Для использования поддержки EC2 в Ansible необходимо установить на управляющей машине библиотеку Python Boto как системный пакет Python. Для этого выполните следующую команду[2]:

```
$ pip install boto
```

Если у вас имеются запущенные экземпляры EC2, попробуйте проверить правильность установки Boto и корректность учетных данных с помощью командной строки Python, как показано в примере 14.2.

Пример 14.2 ❖ Тестирование Boto и учетных данных

```
$ python
Python 2.7.12 (default, Nov 6 2016, 20:41:56)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto.ec2
>>> conn = boto.ec2.connect_to_region("us-east-1")
>>> statuses = conn.get_all_instance_status()
>>> statuses
[]
```

---

[1] Или, может быть, в *~/.bashrc*? Я никогда не понимал разницы между этими файлами в Bash.

[2] Для установки пакета может потребоваться использовать команду `sudo` или активировать виртуальное окружение virtualenv, в зависимости от того, как была установлена система Ansible.

Динамическая инвентаризация

Я уверен, что при работе с северами 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 преимущества динамической инвентаризации, которая позволяет получить информацию о хостах непосредственно из EC2. В составе Ansible имеется сценарий динамической инвентаризации EC2, но я рекомендую загрузить его последнюю версию из репозитория Ansible[1]. Вам потребуются два файла:

- *ec2.py* – актуальный сценарий инвентаризации (http://bit.ly/2lAsfV8);
- *ec2.ini* – файл конфигурации для сценария инвентаризации (http://bit.ly/2l168KP).

Ранее у нас имелся файл *playbooks/hosts*, который мы использовали в качестве реестра. Сейчас мы будем использовать каталог *playbooks/inventory*. Поместим файлы *ec2.py* и *ec2.ini* в этот каталог и установим для *ec2.py* разрешение на выполнение. В примере 14.3 показано, как это сделать.

Пример 14.3 ❖ Установка сценария динамической инвентаризации EC2

```
$ cd playbooks/inventory
$ wget https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory\
/ec2.py
$ wget https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory\
/ec2.ini
$ chmod +x ec2.py
```

Если вы используете Ansible в дистрибутиве Linux (например, Arch Linux), где по умолчанию применяется Python версии 3.x, тогда сценарий *ec2.py* не будет работать без изменений, потому что он написан для Python версии 2.x.
Убедитесь, что в вашей системе установлена версия Python 2.x, и замените первую строку в *ec2.py*:

```
#!/usr/bin/env python
```

на:

```
#!/usr/bin/env python2
```

Если вы определили переменные окружения, как описывалось в предыдущем разделе, у вас должно получиться проверить работоспособность сценария:

```
$ ./ec2.py --list
```

Сценарий должен вывести информацию о ваших экземплярах EC2, как показано ниже:

```
{
  "_meta": {
```

[1] По правде сказать, я не имею ни малейшего представления, куда устанавливают этот файл диспетчеры пакетов.

```
    "hostvars": {
      "ec2-203-0-113-75.compute-1.amazonaws.com": {
        "ec2_id": "i-1234567890abcdef0",
        "ec2_instance_type": "c3.large",
        ...
      }
    }
  },
  "ec2": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
  ],
  "us-east-1": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
  ],
  "us-east-1a": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
  ],
  "i-12345678": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
  ],
  "key_mysshkeyname": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
  ],
  "security_group_ssh": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
  ],
  "tag_Name_my_cool_server": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
  ],
  "type_c3_large": [
    "ec2-203-0-113-75.compute-1.amazonaws.com",
    ...
  ]
}
```

Если вы не включили явно в своей учетной записи AWS поддержку RDS и ElastiCache, сценарий *ec2.py* завершится с ошибкой. Чтобы включить RDS и ElastiCache, зарегистрируйтесь в службе реляционной базы данных (Relational Database Service, RDS) и ElastiCache с помощью консоли AWS и дождитесь, когда Amazon активирует эти службы для вас.

Если вы не пользуетесь этими службами, отредактируйте свой файл *ec2.ini*, чтобы запретить сценарию инвентаризации подключаться к ним:

```
[ec2]
...
```

```
rds = False
elasticache = False
```

Эти строки присутствуют в файле, но закомментированы по умолчанию, поэтому вам останется только раскомментировать их!

Кэширование реестра

Когда Ansible использует сценарий динамической инвентаризации EC2, он (сценарий) должен посылать запросы одной или нескольким конечным точкам EC2 для получения информации. Поскольку все это требует времени, при первом запуске скрипт кэширует информацию, создавая следующие файлы:

- *$HOME/.ansible/tmp/ansible-ec2.cache*;
- *$HOME/.ansible/tmp/ansible-ec2.index*.

В последующем сценарий динамической инвентаризации будет обращаться к кэшу, пока срок его годности не истечет.

Такое поведение можно изменить, изменив параметр настройки `cache_max_age` в файле конфигурации *ec2.ini*. По умолчанию время действия кэша составляет 300 секунд (5 минут). Если вы не хотите сохранять кэш, можно установить значение, равное 0:

```
[ec2]
...
cache_max_age = 0
```

Также можно заставить сценарий инвентаризации обновить кэш, запустив его с параметром `--refresh-cache`:

```
$ ./ec2.py --refresh-cache
```

При создании или удалении экземпляров сценарий динамической инвентаризации EC2 не будет отражать эти изменения, кроме случаев, когда срок годности кэша истек или был обновлен принудительно.

Другие параметры настройки

Файл *ec2.ini* содержит несколько параметров настройки, управляющих поведением сценария динамической инвентаризации. Поскольку сам файл снабжен подробными комментариями, я не буду рассказывать об этих параметрах в деталях.

Автоматические группы

Сценарий динамической инвентаризации EC2 автоматически создает следующие группы:

Таблица 14.1. Группы, сгенерированные EC2

Тип	Пример	Название группы в Ansible
Экземпляр	i-1234567890abcdef0	i-1234567890abcdef0
Образ AMI	ami-79df8219	ami-79df8219
Тип экземпляра	c1.medium	type_c1_medium

Окончание табл. 14.1

Тип	Пример	Название группы в Ansible
Группа безопасности	ssh	security_group_ssh
Пара ключей SSH	foo	key_foo
Регион	us-east-1	us-east-1
Тег	env=staging	tag_env_staging
Зона доступности	us-east-1b	us-east-1b
VPC	vpc-14dd1b70	vpc_id_vpc-14dd1b70
Все экземпляры	нет	ec2

В именах группы допускается использовать только буквенно-цифровые символы, дефис и нижнее подчеркивание. Все другие символы сценарий динамической инвентаризации преобразует в нижнее подчеркивание.

Например, представьте, что у вас имеется экземпляр с тегом:

```
Name=My cool server!
```

Ansible сгенерирует имя группы `tag_Name_my_cool_server`.

Определение динамических групп с помощью тегов

Напоминаю, что сценарий динамической инвентаризации создает группы на основании таких данных, как тип экземпляра, группа безопасности, пара ключей и теги. Теги EC2 являются наиболее удобным способом создания групп, поскольку их можно определить каким угодно способом.

Например, всем веб-серверам можно присвоить тег:

```
type=web
```

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 промышленной 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 14.4.

Пример 14.4 ❖ 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=us-west-1
      args:
        tags: { type: web, env: production }
      with_items: "{{ web_production }}"

    - name: Tag staging webservers
      ec2_tag: resource={{ item }} region=us-west-1
      args:
        tags: { type: web, env: staging }
      with_items: "{{ web_staging }}"
```

В этом примере используется синтаксис YAML встраиваемых 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

Лично мне не нравятся имена групп, такие как `tag_type_web`. Я бы предпочел более простое имя `web`.

Для этого в каталог *playbooks/inventory* необходимо добавить новый файл с информацией о группах. Это обычный файл реестра Ansible с именем *playbooks/inventory/hosts* (см. пример 14.5).

Пример 14.5 ❖ *playbooks/inventory/hosts*

```
[web:children]
tag_type_web

[tag_type_web]
```

После этого вы можете обращаться к группе web в операциях Ansible.

Если не определить пустую группу tag_type_web в статическом файле реестра, и сценарий динамической инвентаризации не определяет ее, Ansible выдаст ошибку:

```
ERROR! Attempted to read "/Users/lorin/dev/ansiblebook
/ch12/playbooks/inventory/hosts" as YAML:
'AnsibleUnicode' object has no attribute 'keys'
Attempted to read "/Users/lorin/dev/ansiblebook
/ch12/playbooks/inventory/hosts" as ini file:
/Users/lorin/dev/ansiblebook/ch12
/playbooks/inventory/hosts:4:
Section [web:children] includes undefined group:
tag_type_web
```

EC2 Virtual Private Cloud (VPC) и EC2 Classic

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 приватного 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. Термин *EC2-VPC* в Amazon обозначает экземпляры, запущенные внутри VPC, а термин *EC2-Classic* – экземпляры, запущенные вне VPC.

Amazon активно стимулирует пользователей к использованию EC2-VPC. Например, некоторые типы экземпляров, такие как *t2.micro*, доступны только в EC2-VPC. В зависимости от того, где вы создали учетную запись AWS и в каком регионе EC2 в прошлом вы запускали свои экземпляры, EC2-Classic может быть не доступен. В табл. 14.2 перечислены учетные записи, которым доступен EC2-Classic[2].

Таблица 14.2. Есть ли у меня доступ к EC2-Classic?

Моя учетная запись создана	Доступность EC2-Classic
До 18 марта 2013 года	Да, но только в регионе, где вы работали раньше
Между 18 марта и 4 декабря 2013 года	Возможно, но только в регионе, где вы работали раньше
После 4 декабря 2013 года	Нет

[1] Внутренняя сеть Amazon делится на подсети, но пользователи не могут управлять распределением экземпляров по этим подсетям.

[2] Дополнительная информация о VPC доступна на сайте Amazon (http://amzn.to/1Fw6v1D). О доступности EC2-Classic в вашем регионе можно узнать по адресу: http://amzn.to/1Fw6w5M.

Основная разница в наличии или отсутствии поддержки EC2-Classic состоит в том, что происходит при создании нового экземпляра EC2 и отсутствии возможности явно привязать VPC ID к этому экземпляру. Если в учетной записи активирована поддержка EC2-Classic, новый экземпляр не связывается с VPC. Если поддержка EC2-Classic в учетной записи выключена, новый экземпляр будет привязан к VPC по умолчанию.

Вот одна из причин, по которой стоит заботиться об этой разнице: в EC2-Classic все экземпляры могут устанавливать исходящие соединения к любым хостам в Интернете. В EC2-VPC исходящие соединения по умолчанию запрещены. Если экземпляру в VPC потребуется установить исходящее соединение, он должен быть включен в группу безопасности, которая позволяет это.

В этой главе я буду предполагать, что у нас имеется только поддержка EC2-VPC, и я включу все экземпляры в группу безопасности, которая позволяет осуществлять исходящие соединения.

Конфигурирование ansible.cfg для использования с ec2

Используя Ansible для настройки экземпляров EC2, я добавляю следующие строки в файл *ansible.cfg*:

```
[defaults]
remote_user = ubuntu
host_key_checking = False
```

Я всегда использую образы Ubuntu и подключаюсь к ним по SSH с использованием имени пользователя `ubuntu`. Я также отключаю проверку ключей хоста, поскольку заранее не знаю, какие ключи получат новые экземпляры[1].

Запуск новых экземпляров

Модуль `ec2` позволяет запускать новые экземпляры в EC2. Это один из наиболее сложных модулей Ansible, поскольку поддерживает огромное количество аргументов.

В примере 14.6 показан простой сценарий для запуска экземпляра EC2 Ubuntu 16.04.

Пример 14.6 ❖ Простой сценарий для создания экземпляра EC2

```
- name: Create an ubuntu instance on Amazon EC2
  hosts: localhost
  tasks:
  - name: start the instance
    ec2:
```

[1] 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 анализа ключей хоста.

```
image: ami-79df8219
region: us-west-1
instance_type: m3.medium
key_name: mykey
group: [web, ssh, outbound]
instance_tags: { Name: ansiblebook, type: web, env: production }
```

Давайте рассмотрим значения параметров.

Параметр `image` определяет идентификатор образа машины Amazon (AMI), который всегда нужно указывать. Как уже говорилось выше, образ – это, по сути, файловая система, содержащая установленную операционную систему. Использованный в примере идентификатор `ami-79df8219` относится к образу с установленной 64-битной версией Ubuntu 16.04.

Параметр `region` определяет географическое местоположение, где будет запущен экземпляр[1].

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[2]. В примере 14.6 используется тип экземпляра *m3.medium*. Это 64-битный экземпляр с одним ядром, 3.75 Гбайт оперативной памяти и 4 Гбайт дискового пространства на SSD.

Не все образы совместимы со всеми типами экземпляров. Я на самом деле не тестировал, работает ли `ami-8caa1ce4` с *m3.medium*. Поэтому будьте внимательны!

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 вами заранее. Вне зависимости от того, что вы сделаете, – создадите новую пару или выгрузите существующую, – вам необходимо дать имя паре ключей SSH.

Параметр `group` определяет список групп безопасности, связанных с экземпляром. Эти группы определяют, какие типы входящих и исходящих соединений разрешены.

Параметр `instance_tags` связывает метаданные с экземпляром в форме тегов EC2 ключ/значение. В предыдущем примере были назначены следующие теги:

```
Name=ansiblebook
type=web
env=production
```

---

[1] Список поддерживаемых регионов вы найдете на сайте Amazon (http://amzn.to/1Fw6OcE).

[2] Существует удобный (неофициальный) веб-сайт (http://www.ec2instances.info/), где можно найти единую таблицу со всеми доступными типами экземпляров EC2.

Вызов модуля ec2 из командной строки – самый простой способ завершить экземпляр, если вы знаете его идентификатор:

```
$ ansible localhost -m ec2 -a \
  'instance_id=i-01176c6682556a360 \
  state=absent'
```

Пары ключей EC2

В примере 14.6 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 Amazon генерирует закрытый и соответствующий ему открытый ключ. Затем отправляет вам приватный ключ. Amazon не хранит копию приватного ключа, то есть вам обязательно нужно сохранить его. Вот как можно создать новый ключ с помощью Ansible:

Пример 14.7 ❖ Создание новой пары ключей SSH

```
- name: create a new keypair
  hosts: localhost
  tasks:
  - name: create mykey
    ec2_key: name=mykey region=us-west-1
    register: keypair

  - name: write the key to a file
    copy:
      dest: files/mykey.pem
      content: "{{ keypair.key.private_key }}"
      mode: 0600
    when: keypair.changed
```

В примере 14.7 для создания новой пары ключей использовался модуль ec2_key. Затем мы вызвали модуль copy с параметром content для сохранения закрытого ключа SSH в файл.

Если модуль создал новую пару ключей, зарегистрированная переменная keypair будет содержать значения, подобные следующим:

```
"keypair": {
  "changed": true,
  "key": {
    "fingerprint": "c5:33:74:84:63:2b:01:29:6f:14:a6:1c:7b:27:65:69:61:f0:e8:b9",
    "name": "mykey",
    "private_key": "-----BEGIN RSA PRIVATE KEY-----\nMIIEowIBAAKCAQEAjAJpvhY3QGKh
...
0PkCRPl8ZHKtShKESIsG3WC\n-----END RSA PRIVATE KEY-----"
  }
}
```

Если пара ключей уже существует, тогда зарегистрированная переменная keypair будет содержать такие значения:

```
"keypair": {
  "changed": false,
  "key": {
    "fingerprint": "c5:33:74:84:63:2b:01:29:6f:14:a6:1c:7b:27:65:69:61:f0:e8:b9",
    "name": "mykey"
  }
}
```

Поскольку значение private_key отсутствует, если ключ уже существует, необходимо добавить выражение when в вызов copy, чтобы запись в файл с закрытым ключом производилась, только если это необходимо.

Добавим строку:

```
when: keypair.changed
```

чтобы просто записать файл на диск, если произошло изменение состояния при выполнении ec2_key (например, был создан новый ключ). Можно поступить иначе – проверить наличие значения private_key:

```
- name: write the key to a file
  copy:
    dest: files/mykey.pem
    content: "{{ keypair.key.private_key }}"
    mode: 0600
  when: keypair.key.private_key is defined
```

Мы используем проверку defined[1], поддерживаемую механизмом Jinja2 для проверки наличия private_key.

Выгрузка существующего ключа

Если у вас уже есть публичный ключ SSH, его можно выгрузить в Amazon и привязать к паре ключей:

```
- name: create a keypair based on my ssh key
  hosts: localhost
  tasks:
  - name: upload public key
    ec2_key: name=mykey key_material="{{ item }}"
    with_file: ~/.ssh/id_rsa.pub
```

Группы безопасности

В примере 14.6 подразумевается, что группы безопасности web, ssh и outbound уже существуют. Проверить наличие групп перед их использованием можно с помощью модуля ec2_group.

---

[1] За дополнительной информацией о проверках, поддерживаемых Jinja2, обращайтесь к документации по адресу: http://bit.ly/1Fw77nO.

Группы безопасности похожи на правила брандмауэра: они определяют правила, кто и как может устанавливать подключения.

В примере 14.8 определяется группа web, позволяющая любому хосту в Интернете подключаться к портам 80 и 443. Группа ssh разрешает любому осуществлять подключение к порту 22. Группа outbound разрешает устанавливать исходящие соединения с кем угодно в Интернете. Исходящие соединения нам необходимы для загрузки пакетов из Интернета.

Пример 14.8 ❖ Группы безопасности

```
- name: web security group
  ec2_group:
    name: web
    description: allow http and https access
    region: "{{ region }}"
    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
  ec2_group:
    name: ssh
    description: allow ssh access
    region: "{{ region }}"
    rules:
      - proto: tcp
        from_port: 22
        to_port: 22
        cidr_ip: 0.0.0.0/0

- name: outbound group
  ec2_group:
    name: outbound
    description: allow outbound connections to the internet
    region: "{{ region }}"
    rules_egress:
      - proto: all
        cidr_ip: 0.0.0.0/0
```

При использовании EC2-Classic нет необходимости задавать группу outbound, поскольку EC2-Classic не запрещает исходящих соединений для экземпляров.

Для тех, кто прежде не пользовался группами безопасности, поясним назначение параметров в словаре rules (см. табл. 14.3).

Таблица 14.3. Параметры правил групп безопасности

Параметр	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, разрешенные для соединения с экземпляром. 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, то есть любой IP-адрес из диапазона от *203.0.113.0* до *203.0.113.255*.

Адрес *0.0.0.0/0* означает, что устанавливать соединения разрешено любому IP-адресу.

Порты групп безопасности

Единственное, что мне 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
  to_port: 5999
  cidr_ip: 0.0.0.0/0
```

Однако я считаю такую нотацию запутывающей, поскольку сам никогда не указываю диапазона портов[3]. Вместо этого я обычно разрешаю 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.

---

[1] Так случилось, что этот пример соответствует особому диапазону IP-адресов TEST-NET-3, зарезервированному для примеров. Это *example.com* из 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.

[3] Проницательные 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.

Модуль ec2_group имеет много других параметров, в том числе для определения правил входящих соединений с использованием идентификаторов групп, а также правил исходящих соединений. За дополнительной информацией обращайтесь к документации.

Получение новейшего AMI

В примере 14.6 машина AMI была выбрана явно:

```
image: ami-79df8219
```

Но такой подход не годится, если вдруг появится желание запустить новейший образ Ubuntu 16.04. Связано это с тем, что Canonical[1] часто выпускает небольшие обновления для Ubuntu, и каждый раз при их выпуске генерируется новая машина AMI. Если еще вчера идентификатор ami-79df8219 соответствовал новейшему релизу Ubuntu 16.04, то завтра это может быть уже не так.

В Ansible имеется интересный модуль ec2_ami_find, извлекающий список идентификаторов AMI, соответствующих критериям поиска, таким как имя образа или теги. Пример 14.9 демонстрирует, как использовать этот модуль для запуска последней 64-битной версии Ubuntu Xenial Xerus 16.04.

Пример 14.9 ❖ Извлечение идентификатора AMI новейшей версии Ubuntu

```
- name: Create an ubuntu instance on Amazon EC2
  hosts: localhost
  tasks:
  - name: Get the ubuntu xenial ebs ssd AMI
    ec2_ami_find:
      name: "ubuntu/images/ebs-ssd/ubuntu-xenial-16.04-amd64-server-*"
      region: "{{ region }}"
      sort: name
      sort_order: descending
      sort_end: 1
        no_result_action: fail
    register: ubuntu_image

- name: start the instance
  ec2:
    region: "{{ region }}"
    image: "{{ ubuntu_image.results[0].ami_id }}"
    instance_type: m3.medium
    key_name: mykey
    group: [web, ssh, outbound]
    instance_tags: { type: web, env: production }
```

В данном случае мы должны знать соглашение, используемое для именования образов Ubuntu. В случае с Ubuntu имя образа всегда заканчивается от-

[1] Canonical – это компания, которая управляет проектом Ubuntu.

меткой времени, например: *ubuntu/images/ebs-ssd/ubuntu-xenial-16.04-amd64-server-20170202*.

В параметре `name` модуля `ec2_ami_find` допускается использовать шаблонный символ `*`, благодаря чему можно получить самый свежий образ, отсортировав список имен в порядке убывания и ограничив область поиска одним именем (то есть взять первый элемент из этого отсортированного списка).

По умолчанию модуль `ec2_ami_find` возвращает признак успеха, даже если не находит ни одного образа AMI, соответствующего заданным критериям. Поскольку это почти всегда нежелательно, я рекомендую добавлять выражение `no_result_action: fail`, чтобы заставить модуль вернуть признак ошибки в отсутствие результатов.

Каждый дистрибутив следует своей стратегии именования AMI, поэтому, если вы решите использовать другой дистрибутив, отличный от Ubuntu, вы должны самостоятельно выяснить правила именования и определить соответствующую строку поиска.

Добавление нового экземпляра в группу

Иногда я предпочитаю писать единый сценарий для запуска экземпляра и затем выполнять на экземпляре другой сценарий.

К сожалению, до запуска сценария хост еще не существует. Запрет кэширования в сценарии динамической инвентаризации тут не поможет, потому что Ansible вызывает его только в самом начале выполнения сценария, что предшествует созданию хоста.

Для добавления экземпляра в группу можно использовать задачу, вызывающую модуль `add_host`, как показано в примере 14.10.

Пример 14.10 ❖ Добавление экземпляра в группу

```
- name: Create an ubuntu instance on Amazon EC2
  hosts: localhost
  tasks:
  - name: start the instance
    ec2:
      image: ami-8caa1ce4
      instance_type: m3.medium
      key_name: mykey
      group: [web, ssh, outbound]
      instance_tags: { type: web, env: production }
    register: ec2

  - name: add the instance to web and production groups
    add_host: hostname={{ item.public_dns_name }} groups=web,production
    with_items: "{{ ec2.instances }}"

- name: do something to production webservers
  hosts: web:&production
  tasks:
  - ...
```

Значение, возвращаемое модулем ec2

Модуль ec2 выводит словарь с тремя полями, перечисленными в табл. 14.4.

Таблица 14.4. Возвратные значения модуля ec2

Поле	Описание
instance_ids	Список идентификаторов экземпляров
instances	Список словарей экземпляра
tagged_instances	Список словарей экземпляра

При передаче пользователем параметра exact_count модуль ec2 может не создавать новых экземпляров, как это описано ниже, в разделе «Создание экземпляров идемпотентным способом». В этом случае поля instance_ids и instances будут заполнены, только если модуль создает новые экземпляры. Однако поле tagged_instances будет содержать словари всех экземпляров, удовлетворяющих тегу, были ли они только что созданы или существовали ранее.
Словарь экземпляра содержит поля, перечисленные в табл. 14.5.

Таблица 14.5. Содержимое словарей экземпляров

Поле	Описание
id	Идентификатор экземпляра
ami_launch_index	Индекс экземпляра между 0 и *N*–1, если запущено *N* экземпляров
private_ip	Внутренний IP-адрес (недоступный за пределами EC2)
private_dns_name	Внутреннее имя DNS (недоступное за пределами EC2)
public_ip	Публичный IP-адрес
public_dns_name	Публичное имя DNS
state_code	Код причины изменения состояния
architecture	Аппаратная архитектура
image_id	AMI
key_name	Имя пары ключей
placement	Место, где был запущен экземпляр
kernel	Образ ядра Amazon (Amazon Kernel Image, AKI)
ramdisk	Образ RAM-диска Amazon (Amazon Kamdisk Image, ARI)
launch_time	Время запуска экземпляра
instance_type	Тип экземпляра
root_device_type	Тип корневого устройства (ephemeral, EBS)
root_device_name	Имя корневого устройства
state	Состояние экземпляра
hypervisor	Тип гипервизора

За дополнительной информацией о значениях полей обращайтесь к документации с описанием класса boto.ec2.instance.Instance (http://bit.ly/1Fw7HSO) или к документации с описанием результатов команды run-instances (http://amzn.to/1Fw7Jd9).

Ожидание запуска сервера

Облака IaaS, такие как EC2, также 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 запустится экземпляр EC2.

Модуль `ec2` поддерживает для этого параметр `wait`. Если в нем передать `yes`, модуль `ec2` не вернет управления, пока экземпляр не перейдет в рабочее состояние:

```
- name: start the instance
  ec2:
    image: ami-8caa1ce4
    instance_type: m3.medium
    key_name: mykey
    group: [web, ssh, outbound]
    instance_tags: { type: web, env: production }
    wait: yes
  register: ec2
```

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: start the instance
  ec2:
    image: ami-8caa1ce4
    instance_type: m3.medium
    key_name: mykey
    group: [web, ssh, outbound]
    instance_tags: { type: web, env: production }
    wait: yes
  register: ec2

- name: wait for ssh server to be running
  wait_for: host={{ item.public_dns_name }} port=22 search_regex=OpenSSH
  with_items: "{{ ec2.instances }}"
```

Вызов `wait_for` использует аргумент `search_regex` для просмотра 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 14.11.

Пример 14.11 ❖ Ответ сервера SSH, работающего в Ubuntu

```
SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.4
```

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. Ожидание первоначального ответа гарантирует, что модуль `wait_for` вернет управление, только когда сервер SSH будет полностью работоспособен.

Создание экземпляров идемпотентным способом

Сценарий, вызывающий модуль `ec2`, обычно не является идемпотентным. Если бы потребовалось выполнить пример 14.6 несколько раз, EC2 создал бы несколько экземпляров.

Придать идемпотентность сценариям, использующим модуль `ec2`, можно с помощью параметров `count_tag` и `exact_count`.

Допустим, требуется написать сценарий, запускающий три экземпляра, и этот сценарий должен быть идемпотентным. То есть, если три экземпляра уже работают, сценарий не должен ничего делать. В примере 14.12 показано, как это можно реализовать.

Пример 14.12 ❖ Создание экземпляров идемпотентным способом

```
- name: start the instance
  ec2:
    image: ami-8caa1ce4
    instance_type: m3.medium
    key_name: mykey
    group: [web, ssh, outbound]
    instance_tags: { type: web, env: production }
    exact_count: 3
    count_tag: { type: web }
```

Параметр `exact_count: 3` сообщает системе Ansible, что она должна проверить наличие именно трех экземпляров, удовлетворяющих тегу, указанному в `count_tag`. В нашем примере я указал только один тег в `count_tag`, но вообще можно указать несколько тегов.

При выполнении этого сценария в первый раз Ansible проверит, сколько экземпляров с тегом `type=web` существует на данный момент. Так как таких экземпляров нет, Ansible создаст три новых и присвоит им теги `type=web` и `env=production`.

При повторном запуске Ansible проверит, сколько экземпляров с тегом `type=web` существует на данный момент. Обнаружив три экземпляра, она не будет создавать новых.

Подведение итогов

В примере 14.13 приводится сценарий, создающий три экземпляра EC2 и настраивающий их как веб-серверы. Сценарий является идемпотентным, то есть его можно спокойно запускать несколько раз, он будет создавать новые экземпляры, только если они еще не были созданы.

Обратите внимание, что здесь вместо значения instances используется значение tagged_instances, возвращаемое модулем ec2. Причина была описана выше, во врезке «Значение, возвращаемое модулем ec2».

Пример 14.13 ❖ *ec2-example.yml*: законченный сценарий EC2

```
---
- name: launch webservers
  hosts: localhost
  vars:
    region: us-west-1
    instance_type: t2.micro
    count: 1
  tasks:
  - name: ec2 keypair
    ec2_key: "name=mykey key_material={{ item }} region={{ region }}"
    with_file: ~/.ssh/id_rsa.pub
  - name: web security group
    ec2_group:
      name: web
      description: allow http and https access
      region: "{{ region }}"
      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
    ec2_group:
      name: ssh
      description: allow ssh access
      region: "{{ region }}"
      rules:
        - proto: tcp
          from_port: 22
          to_port: 22
          cidr_ip: 0.0.0.0/0
  - name: outbound security group
    ec2_group:
      name: outbound
      description: allow outbound connections to the internet
      region: "{{ region }}"
      rules_egress:
        - proto: all
          cidr_ip: 0.0.0.0/0
  - name: Get the ubuntu xenial ebs ssd AMI
    ec2_ami_find:
      name: "ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"
```

```
      region: "{{ region }}"
      sort: name
      sort_order: descending
      sort_end: 1
      no_result_action: fail
    register: ubuntu_image
  - set_fact: "ami={{ ubuntu_image.results[0].ami_id }}"
  - name: start the instances
    ec2:
      region: "{{ region }}"
      image: "{{ ami }}"
      instance_type: "{{ instance_type }}"
      key_name: mykey
      group: [web, ssh, outbound]
      instance_tags: { Name: ansiblebook, type: web, env: production }
      exact_count: "{{ count }}"
      count_tag: { type: web }
      wait: yes
    register: ec2
  - name: add the instance to web and production groups
    add_host: hostname={{ item.public_dns_name }} groups=web,production
    with_items: "{{ ec2.tagged_instances }}"
    when: item.public_dns_name is defined
  - name: wait for ssh server to be running
    wait_for: host={{ item.public_dns_name }} port=22 search_regex=OpenSSH
    with_items: "{{ ec2.tagged_instances }}"
    when: item.public_dns_name is defined

- name: configure webservers
  hosts: web:&production
  become: True
  gather_facts: False
  pre_tasks:
    - name: install python
      raw: apt-get install -y python-minimal
  roles:
    - web
```

Создание виртуального приватного облака

До сих пор мы запускали экземпляры в виртуальном приватном облаке (VPC) по умолчанию. Однако Ansible позволяет также создавать новые облака VPC и запускать в них экземпляры.

Что такое VPC?

Виртуальное приватное облако (VPC) можно рассматривать как изолированную сеть. Создавая такое облако, вы должны определить диапазон IP-адресов. Это должно быть подмножество одного из приватных диапазонов (*10.0.0.0/8*, *172.16.0.0/12* или *192.168.0.0/16*).

Виртуальное облако делится на подсети – поддиапазоны IP-адресов из общего диапазона всего облака. В примере 14.14 облако располагает диапазоном *10.0.0.0/16*, и мы выделили в нем две подсети: *10.0.0.0/24* и *10.0.10/24*.

Запуская экземпляр, вы должны включить его в одну из подсетей в облаке. Подсети можно настроить так, что экземпляры будут получать публичные или приватные IP-адреса. EC2 также позволяет определять таблицы маршрутизации для передачи трафика между подсетями и создавать интернет-шлюзы для передачи трафика из подсетей в Интернет.

Настройка сети – сложная тема, выходящая далеко за рамки этой книги. Дополнительную информацию вы сможете найти в документации Amazon с описанием приемов создания EC2 в VPC (http://amzn.to/1Fw89Af).

В примере 14.14 показано, как создать VPC с интернет-шлюзом, двумя подсетями и таблицей маршрутизации, которая определяет порядок направления исходящих соединений через интернет-шлюз.

Пример 14.14 ❖ *create-vpc.yml*: создание VPC

```
- name: create a vpc
  ec2_vpc_net:
    region: "{{ region }}"
    name: "Book example"
    cidr_block: 10.0.0.0/16
    tags:
      env: production
  register: result
- set_fact: "vpc_id={{ result.vpc.id }}"
- name: add gateway
  ec2_vpc_igw:
    region: "{{ region }}"
    vpc_id: "{{ vpc_id }}"
- name: create web subnet
  ec2_vpc_subnet:
      region: "{{ region }}"
      vpc_id: "{{ vpc_id }}"
      cidr: 10.0.0.0/24
      tags:
        env: production
        tier: web
- name: create db subnet
  ec2_vpc_subnet:
    region: "{{ region }}"
    vpc_id: "{{ vpc_id }}"
    cidr: 10.0.1.0/24
    tags:
      env: production
      tier: db
- name: set routes
  ec2_vpc_route_table:
```

```
    region: "{{ 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. 14.6).

Таблица 14.6. 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 модуль с признаком ошибки.

Если не указать теги в `ec2_vpc_route_table`, при каждом обращении к модулю будет создаваться новая таблица маршрутизации.

Необходимо отметить, что пример 14.14 довольно прост с точки зрения настройки сети, потому что мы определили всего две подсети и обе они подключены к Интернету. В реальной жизни чаще встречаются ситуации, когда выход в Интернет имеет только одна из подсетей, а также заданы правила маршрутизации трафика между ними.

В примере 14.15 показан законченный сценарий создания VPC и запуска на нем экземпляров.

Пример 14.15 ❖ *ec2-vpc-example.yml*: законченный сценарий EC2 для задания VPC

```
---
- name: launch webservers into a specific vpc
  hosts: localhost
  vars:
    region: us-west-1
    instance_type: t2.micro
    count: 1
    cidrs:
      web: 10.0.0.0/24
```

[1] Если в параметре `lookup` передано значение `id`, для контроля идемпотентности вместо `tags` будет использоваться параметр `route_table_id`.

```
    db: 10.0.1.0/24
tasks:
- name: create a vpc
  ec2_vpc_net:
    region: "{{ region }}"
    name: book
    cidr_block: 10.0.0.0/16
    tags: {env: production }
  register: result
- set_fact: "vpc_id={{ result.vpc.id }}"
- name: add gateway
  ec2_vpc_igw:
    region: "{{ region }}"
    vpc_id: "{{ vpc_id }}"
- name: create web subnet
  ec2_vpc_subnet:
    region: "{{ region }}"
    vpc_id: "{{ vpc_id }}"
    cidr: "{{ cidrs.web }}"
    tags: { env: production, tier: web}
  register: web_subnet
- set_fact: "web_subnet_id={{ web_subnet.subnet.id }}"
- name: create db subnet
  ec2_vpc_subnet:
    region: "{{ region }}"
    vpc_id: "{{ vpc_id }}"
    cidr: "{{ cidrs.db }}"
    tags: { env: production, tier: db}
- name: add routing table
  ec2_vpc_route_table:
    region: "{{ region }}"
    vpc_id: "{{ vpc_id }}"
    tags:
      purpose: permit-outbound
    subnets:
      - "{{ cidrs.web }}"
      - "{{ cidrs.db }}"
    routes:
      - dest: 0.0.0.0/0
        gateway_id: igw
- name: set ec2 keypair
  ec2_key: "name=mykey key_material={{ item }}"
  with_file: ~/.ssh/id_rsa.pub
- name: web security group
  ec2_group:
    name: web
    region: "{{ region }}"
    description: allow http and https access
    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
  ec2_group:
    name: ssh
    region: "{{ region }}"
    description: allow ssh access
    vpc_id: "{{ vpc_id }}"
    rules:
      - proto: tcp
        from_port: 22
        to_port: 22
        cidr_ip: 0.0.0.0/0
- name: outbound security group
  ec2_group:
    name: outbound
    description: allow outbound connections to the internet
    region: "{{ region }}"
    vpc_id: "{{ vpc_id }}"
    rules_egress:
      - proto: all
        cidr_ip: 0.0.0.0/0
- name: Get the ubuntu xenial ebs ssd AMI
  ec2_ami_find:
    name: "ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"
    region: "{{ region }}"
    sort: name
    sort_order: descending
    sort_end: 1
    no_result_action: fail
  register: ubuntu_image
- set_fact: "ami={{ ubuntu_image.results[0].ami_id }}"
- name: start the instances
  ec2:
    image: "{{ ami }}"
    region: "{{ region }}"
    instance_type: "{{ instance_type }}"
    assign_public_ip: True
    key_name: mykey
    group: [web, ssh, outbound]
    instance_tags: { Name: book, type: web, env: production }
    exact_count: "{{ count }}"
    count_tag: { type: web }
    vpc_subnet_id: "{{ web_subnet_id }}"
    wait: yes
  register: ec2
```

```
  - name: add the instance to web and production groups
    add_host: hostname={{ item.public_dns_name }} groups=web,production
    with_items: "{{ ec2.tagged_instances }}"
    when: item.public_dns_name is defined
  - name: wait for ssh server to be running
    wait_for: host={{ item.public_dns_name }} port=22 search_regex=OpenSSH
    with_items: "{{ ec2.tagged_instances }}"
    when: item.public_dns_name is defined

- name: configure webservers
  hosts: web:&production
  become: True
  gather_facts: False
  pre_tasks:
    - name: install python
      raw: apt-get install -y python-minimal
  roles:
    - web
```

Динамическая инвентаризация и VPC

При использовании VPC экземпляры часто помещаются в приватную подсеть, не соединенную с Интернетом. 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.

В приложении B подробно рассказывается, как можно использовать роли IAM для запуска Ansible внутри VPC без необходимости копирования учетных данных EC2 в экземпляр.

Создание AMI

Существуют два подхода к созданию образов Amazon (AMI) с помощью Ansible: используя модуль `ec2_ami` или инструмент Packer, который поддерживает Ansible.

Использование модуля ec2_ami

Модуль `ec2_ami` создает снимок запущенного экземпляра AMI. В примере 14.16 показано, как действует этот модуль.

Пример 14.16 ❖ Создание AMI с помощью модуля `ec2_ami`

```
- name: create an AMI
  hosts: localhost
  vars:
    instance_id: i-e5bfc266641f1b918
  tasks:
```

```
- name: create the AMI
  ec2_ami:
    name: web-nginx
    description: Ubuntu 16.04 with nginx installed
    instance_id: "{{ instance_id }}"
    wait: yes
  register: ami

- name: output AMI details
  debug: var=ami
```

Использование Packer

Модуль `ec2_ami` прекрасно справляется со своей задачей, но требует писать дополнительный код для создания и удаления экземпляров. Существует инструмент с открытым кодом, Packer (https://www.packer.io), который автоматически создает и удаляет экземпляры. Этот инструмент написал Митчел Хашимото (Mitchell Hashimoto), который также является создателем Vagrant.

Packer может создавать разные типы образов и поддерживает разные инструменты управления конфигурациями. В этом разделе мы сфокусируемся на использовании Packer для создания AMI с помощью Ansible, но его также можно использовать для создания образов других облаков IaaS, например Google Compute Engine, DigitalOcean или OpenStack. Его даже можно использовать для создания машин Vagrant и контейнеров Docker. Он также поддерживает другие инструменты управления конфигурациями, такие как Chef, Puppet и Salt.

Для использования Packer необходимо создать файл конфигурации в формате JSON и затем использовать утилиту `packer` для создания образа.

Packer поддерживает два *механизма наполнения*, которые можно использовать из сценариев Ansible для создания AMI: более новый механизм Ansible Remote (с именем `ansible`) и более старый Ansible Local (с именем `ansible-local`). Чтобы понять разницу между ними, сначала нужно разобраться в том, как работает Packer.

В процессе создания образа AMI Packer выполняет следующие действия:

1. Запускает новый экземпляр EC2, опираясь на образ AMI, указанный в файле конфигурации.
2. Создает временную пару ключей и группу безопасности.
3. Выполняет вход в новый экземпляр через SSH и выполняет все сценарии наполнения, указанные в файле конфигурации.
4. Останавливает экземпляр.
5. Создает новый образ AMI.
6. Удаляет экземпляр, группу безопасности и пару ключей.
7. Выводит идентификатор AMI в терминал.

Механизм Ansible Remote

Когда используется механизм наполнения Ansible Remote, Packer выполняет сценарии Ansible на локальной машине. Когда используется механизм наполнения Ansible Local, Packer копирует сценарии Ansible в экземпляр и запускает

их там. Я предпочитаю использовать механизм Ansible Remote, потому что он проще в конфигурировании, как вы убедитесь чуть ниже.

Сначала рассмотрим механизм Ansible Remote. В примере 14.17 представлен сценарий *web-ami.yml*, выполняющий подготовку экземпляра для создания образа. Это простой сценарий, который применяет роль `web` к машине с именем `default`. Packer автоматически создает псевдоним `default`. При желании псевдоним можно изменить, определив параметр `host_alias` в разделе Ansible внутри файла конфигурации Packer.

Пример 14.17 ❖ *web-ami.yml*

```
- name: configure a webserver as an ami
  hosts: default
  become: True
  roles:
    - web
```

В примере 14.18 показан вариант файла конфигурации Packer, который использует механизм Ansible Remote для создания AMI с использованием нашего сценария.

Пример 14.18 ❖ *web.json* с использованием механизма Ansible Remote

```
{
  "builders": [
    {
      "type": "amazon-ebs",
      "region": "us-west-1",
      "source_ami": "ami-79df8219",
      "instance_type": "t2.micro",
      "ssh_username": "ubuntu",
      "ami_name": "web-nginx-{{timestamp}}",
      "tags": {
        "Name": "web-nginx"
      }
    }
  ],
  "provisioners": [
    {
      "type": "shell",
      "inline": [
        "sleep 30",
        "sudo apt-get update",
        "sudo apt-get install -y python-minimal"
      ]
    },
    {
      "type": "ansible",
      "playbook_file": "web-ami.yml"
    }
  ]
}
```

Используйте команду `packer build` для создания AMI:

```
$ packer build web.json
```

Результат будет выглядеть примерно так:

```
==> amazon-ebs: Prevalidating AMI Name...
    amazon-ebs: Found Image ID: ami-79df8219
==> amazon-ebs: Creating temporary keypair:
    packer_58a0d118-b798-62ca-50d3-18d0e270e423
==> amazon-ebs: Creating temporary security group for this instance...
==> amazon-ebs: Authorizing access to port 22 the temporary security group...
==> amazon-ebs: Launching a source AWS instance...
    amazon-ebs: Instance ID: i-0f4b09dc0cd806248
==> amazon-ebs: Waiting for instance (i-0f4b09dc0cd806248) to become ready...
==> amazon-ebs: Adding tags to source instance
==> amazon-ebs: Waiting for SSH to become available...
==> amazon-ebs: Connected to SSH!
==> amazon-ebs: Provisioning with shell script: /var/folders/g_/523vq6g1037d1
0231mmbx1780000gp/T/packer-shell574734910
...

==> amazon-ebs: Stopping the source instance...
==> amazon-ebs: Waiting for the instance to stop...
==> amazon-ebs: Creating the AMI: web-nginx-1486934296
    amazon-ebs: AMI: ami-42ffa322
==> amazon-ebs: Waiting for AMI to become ready...
==> amazon-ebs: Adding tags to AMI (ami-42ffa322)...
==> amazon-ebs: Tagging snapshot: snap-01b570285183a1d35
==> amazon-ebs: Creating AMI tags
==> amazon-ebs: Creating snapshot tags
==> amazon-ebs: Terminating the source AWS instance...
==> amazon-ebs: Cleaning up any extra volumes...
==> amazon-ebs: No volumes to clean up, skipping
==> amazon-ebs: Deleting temporary security group...
==> amazon-ebs: Deleting temporary keypair...

Build 'amazon-ebs' finished.

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:

us-west-1: ami-42ffa322
```

В примере 14.18 имеются две секции: `builders` и `provisioners`. Секция `builders` определяет тип создаваемого образа. В данном случае создается EBS-образ (Elastic Block Store–backed), поэтому мы используем построитель `amazon-ebs`.

Чтобы создать образ AMI, Packer должен запустить новый экземпляр, поэтому мы должны указать всю необходимую информацию, обычно используемую при создании экземпляра: регион EC2, AMI и тип экземпляра. Packer не требует настраивать группу безопасности, потому что, как отмечалось выше, он автоматически создает временную группу безопасности, а затем удаляет ее по завершении. Подобно Ansible, Packer должен иметь возможность установить

SSH-соединение с созданным экземпляром. Поэтому мы должны указать имя пользователя в файле конфигурации Packer.

Также необходимо указать имя экземпляра и все теги, которые требуется присвоить. Поскольку имена AMI должны быть уникальными, мы использовали функцию `{{timestamp}}` для добавления метки времени. Метка времени определяет дату и время в виде количества секунд, прошедших с 1 января 1970 года, UTC. За дополнительной информацией о функциях Packer обращайтесь к документации Packer (http://bit.ly/1Fw9hEc).

Поскольку для создания образа требуется взаимодействовать с EC2, Packer должен иметь доступ к учетным данным EC2. Так же как Ansible, Packer может читать их из переменных окружения, поэтому нет необходимости явно указывать их в файле конфигурации, хотя, если хочется, это можно сделать.

Секция `provisioners` определяет инструменты, используемые для конфигурации экземпляра до его сохранения в виде образа. Packer поддерживает сценарий на языке командной оболочки, который позволяет запускать произвольные команды на экземпляре. Он используется в примере 14.18 для установки Python 2. Чтобы избежать состояния гонки, пытаясь установить пакеты до того, как операционная система полностью загрузится, сценарий командной оболочки в нашем примере настроен так, чтобы выждать 30 секунд перед установкой Ansible.

Механизм Ansible Local

Использование механизма наполнения Ansible Local похоже на использование механизма Ansible Remote, за исключением небольших, но важных отличий.

По умолчанию механизм Ansible Local копирует на удаленный хост только файл самого сценария Ansible: любые другие файлы, от которых зависит сценарий, не копируются автоматически. Для решения этой проблемы необходимо средство доступа к нескольким файлам. Packer позволяет задать в параметре `playbook_dir` каталог, который будет рекурсивно копироваться в целевой каталог на экземпляре. Вот пример фрагмента файла конфигурации Packer, где определяется каталог для копирования:

```
{
  "type": "ansible-local",
  "playbook_file": "web-ami-local.yml",
  "playbook_dir": "../playbooks"
}
```

Если требуется скопировать только файлы, определяющие роли, с помощью параметра `role_paths` можно явно задать список каталогов ролей:

```
{
  "type": "ansible-local",
  "playbook_file": "web-ami-local.yml",
  "role_paths": [
    "../playbooks/roles/web"
  ]
}
```

Другое важное отличие: в выражении `hosts` вместо `default` следует использовать `localhost`.

Возможности Packer намного шире, чем мы рассмотрели здесь, включая множество параметров настройки механизмов наполнения обоих типов. Дополнительную информацию вы найдете в документации: https://www.packer.io/docs/.

Другие модули

Ansible обладает более обширной поддержкой EC2, чем мы описали, а также может взаимодействовать с другими службами AWS. Например, Ansible можно использовать для запуска стеков CloudFormation с помощью модуля `cloudformation`, сохранения файлов в S3 с помощью модуля `s3`, изменения записи DNS с помощью модуля `route53`, создания группы автоматического масштабирования с помощью модуля `ec2_asg`, создания конфигурации автоматического масштабирования с помощью модуля `ec2_lc` и многого другого.

Использование Ansible с EC2 – обширная тема, о которой можно написать отдельную книгу. На самом деле Ян Курниаван (Yan Kurniawan) работает над книгой об Ansible и AWS.

Теперь вы обладаете достаточным объемом знаний, чтобы справиться с этими дополнительными модулями без проблем.

Глава 15

Docker

Проект Docker стремительно захватил мир IT. 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 контейнеры Docker.

Что такое контейнер?

Контейнер – 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.

Контейнеры иногда называют *виртуализацией операционной системы*, чтобы отделить их от технологий *виртуализации аппаратного обеспечения*. При виртуализации аппаратного обеспечения программное обеспечение, называемое *гипервизором*, воссоздает физическую машину целиком, включая виртуальные CPU, 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 2012 в системе-носителе RedHat Enterprise Linux), и останавливать и запускать виртуальную машину точно так же, как физическую. Эта гибкость несет с собой потери в производительности, необходимые для 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. 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 – это больше, чем контейнеры. Это ПО можно считать платформой, в которой контейнеры играют роль строительных блоков. Контейнеры в Docker – это почти то же самое, что виртуальные машины в облаках IaaS. Две другие важные составляющие Docker – формат образов и Docker API.

Образы Docker можно считать аналогами образов виртуальных машин. Образ Docker содержит файловую систему с установленной операционной системой, а также некоторые метаданные. Одно существенное отличие в том, что образы Docker – многоуровневые. Для создания нового образа Docker берется существующий образ и модифицируется добавлением, изменением или удалением файлов. Представление нового образа Docker содержит информацию об оригинальном образе Docker, а также отличия в файловой системе между оригинальным и новым образами Docker. Например: официальный образ Nginx для Docker (http://bit.ly/2ktXbqS) реализован наложением дополнительных уровней на официальный образ Debian Jessie. Благодаря такой многоуровневой организации образы Docker гораздо меньше традиционных образов виртуальных машин, а значит, их легче передать через Интернет. Проект Docker поддерживает реестр публично доступных образов (https://registry.hub.docker.com/).

Также Docker поддерживает API удаленного управления, позволяющий осуществлять взаимодействия со сторонними инструментами. Этот API как раз использует модуль Ansible `docker`.

Объединение Docker и Ansible

Контейнеры Docker позволяют легко упаковать приложение в образ, который легко развертывается в различных системах. Именно поэтому в отношении проекта Docker используется метафора корабельных контейнеров. API удаленного управления в Docker упрощает автоматизацию программных систем, запускаемых поверх Docker.

Есть две области, в которых Ansible упрощает работу с Docker. Одна связана с управлением контейнерами Docker. Для установки «Docker'изированного» программного обеспечения обычно приходится создавать несколько контейнеров Docker, содержащих разные службы. Эти службы должны иметь возможность взаимодействовать между собой, поэтому нужно правильно объединить соответствующие контейнеры и убедиться, что они запускаются в правильном порядке. Изначально проект Docker не включал развитых инструментов управления, по этой причине появились сторонние инструменты. Ansible была создана для управления, поэтому ее использование для развертывания приложений в Docker выглядит естественным решением.

Другая область – это создание образов Docker. Официальный способ создания своих образов Docker заключается в создании особых текстовых файлов, называемых *Dockerfiles*, которые похожи на сценарии командной оболочки. Dockerfiles прекрасно подходят для создания простых образов. Но когда требу-

ется реализовать что-то более сложное, вся мощь Ansible теряется. К счастью, Ansible можно использовать для создания сценариев.

Совсем недавно появился новый проект *Ansible Container* – официальное решение использования сценариев Ansible для создания образов контейнеров Docker. На момент написания этих строк самой свежей была версия Ansible Container 0.2. В конце января 2017-го руководители проекта объявили в списке рассылки Ansible Container, что следующая версия проекта, *Ansible Container Mk. II*, будет включать существенные отличия. Поскольку Ansible Container все еще находится на начальной стадии развития, мы решили не углубляться в описание его особенностей в этой книге. Но будет очень хорошо, если вы самостоятельно познакомитесь с этим проектом.

Жизненный цикл приложения Docker

Вот как выглядит обычный жизненный цикл приложения Docker:

1. Создание образов Docker на локальной машине.
2. Передача образов Docker с локальной машины в реестр.
3. Извлечение образов Docker на удаленный хост из реестра.
4. Запуск контейнеров Docker на удаленном хосте путем передачи им информации о конфигурации.

Обычно образ Docker создается на локальной машине или в системе непрерывной интеграции, поддерживающей их создание, например Jenkins или CircleCI. После создания образ необходимо где-то сохранить, откуда его легко будет загрузить на удаленные хосты.

Образы Docker 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.

После размещения образа Docker 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.

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 Docker на локальной машине.
3. Передача образов Docker с локальной машины в реестр.
4. Написание сценариев Ansible для извлечения образов Docker на удаленные хосты, запуск контейнеров Docker на удаленных хостах путем передачи информации о конфигурации.
5. Выполнение сценариев Ansible для запуска контейнеров.

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.

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. 15.1 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 15.1. 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*)

Запуск контейнера 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, 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
CONTAINER ID   IMAGE          COMMAND                  CREATED
48e69da90023   ghost          "/entrypoint.sh np..."   37 seconds ago
               STATUS         PORTS                    NAMES
               Up 36 seconds  0.0.0.0:8000->2368/tcp   test-ghost
```

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`, имеются эквивалентные параметры для модуля `docker_container`.

Создание образа из Dockerfile

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 6, когда развертывали приложение Mezzanine. В примере 15.1 представлен файл *Dockerfile*, реализующий все необходимое.

Пример 15.1 ❖ *Dockerfile*

```
FROM nginx
RUN rm /etc/nginx/conf.d/default.conf
COPY ghost.conf /etc/nginx/conf.d/ghost.conf
```

В примере 15.2 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*

```
upstream ghost {
    server ghost:2368;
}
```

```
server {

    listen 80;

    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;
    # # параметр ssl_ciphers слишком длинный,
    # # чтобы приводить его на страницах книги
    ssl_prefer_server_ciphers on;

    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://ghost;
    }
}
```

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 *lorin/nginx-ghost*. Здесь использован префикс *ansiblebook/*, потому что собираемся поместить образ в репозиторий Docker Hub с именем *ansiblebook/nginx-ghost*:

```
- name: create Nginx image
  docker_image:
    name: ansiblebook/nginx-ghost
    path: nginx
```

Убедиться в успешном выполнении задачи можно с помощью команды `docker images`:

```
$ docker images
REPOSITORY               TAG      IMAGE ID        CREATED
ansiblebook/nginx-ghost  latest   23fd848947a7    37 seconds ago
ghost                    latest   066a22d980f4    3 days ago
nginx                    latest   cc1b61406712    11 days ago
                         SIZE
                         182 MB
                         326 MB
                         182 MB
```

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: yes`.

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 явного использования параметра `forced`.

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 нужд разработки, когда все контейнеры выполняются на одной машине, проект Docker предоставляет инструмент *Docker Compose*, упрощающий запуск и связывание контейнеров. Для управления контейнерами с помощью инструмента Docker Compose можно использовать модуль `docker_service`.

В примере 15.3 представлен файл *docker-compose.yml*, который запускает Nginx и Ghost. В данном случае предполагается наличие каталога *./certs* с файлами сертификатов TLS.

Пример 15.3 ❖ *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 15.4 приводится сценарий, который создает файл образа Nginx, затем создает самоподписанные сертификаты и запускает службы, описанные в примере 15.3.

Пример 15.4 ❖ *ghost.yml*

```
---
- name: Run Ghost locally
  hosts: localhost
```

```
  gather_facts: False
  tasks:
    - name: create Nginx image
      docker_image:
        name: ansiblebook/nginx-ghost
        path: nginx
    - name: create certs
      command: >
        openssl req -new -x509 -nodes
        -out certs/nginx.crt -keyout certs/nginx.key
        -subj '/CN=localhost' -days 3650
        creates=certs/nginx.crt
    - name: bring up services
      docker_service:
        project_src: .
        state: present
```

Отправка образа в реестр Docker

Для отправки образа в репозиторий Docker Hub мы используем отдельный сценарий, представленный в примере 15.5. 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 15.5 ❖ *publish.yml*

```
- name: publish images to docker hub
  hosts: localhost
  gather_facts: False
  vars_prompt:
    - name: username
      prompt: Enter Docker Registry username
    - name: email
      prompt: Enter Docker Registry email
    - name: password
      prompt: Enter Docker Registry password
      private: yes
  tasks:
    - name: authenticate with repository
      docker_login:
        username: "{{ username }}"
        email: "{{ email }}"
        password: "{{ password }}"
    - name: push image up
      docker_image:
        name: ansiblebook/nginx-ghost
        push: yes
```

Если вы собираетесь использовать другой реестр, определите параметр `registry_url` в `docker_login` и префикс имени образа с именем хоста и номером пор-

та реестра (если реестр использует нестандартный порт HTTP/HTTPS). В примере 15.6 показано, как следует изменить задачи при использовании реестра http://reg.example.com. Сценарий создания образа также необходимо изменить, чтобы отразить в нем новое имя образа: *reg.example.com/ansiblebook/nginx-ghost*.

Пример 15.6 ❖ *publish.yml* для случая использования нестандартного реестра

```
tasks:
  - name: authenticate with repository
    docker_login:
      username: "{{ username }}"
      email: "{{ email }}"
      password: "{{ password }}"
      registry_url: http://reg.example.com
  - name: push image up
    docker_image:
      name: reg.example.com/ansiblebook/nginx-ghost
      push: yes
```

Проверить сохранение образа в реестре Docker можно с использованием локального реестра. Сценарий в примере 15.7 запускает реестр в контейнере Docker, отмечает образ *ansiblebook/nginx-ghost* как *localhost:5000/ansiblebook/nginx-ghost* и помещает его в реестр. Обратите внимание, что по умолчанию локальные реестры не требуют аутентификации, поэтому в данном сценарии отсутствует задача, вызывающая `docker_login`.

Пример 15.7 ❖ *publish.yml* для случая использования локального реестра

```
- name: publish images to local docker registry
  hosts: localhost
  gather_facts: False
  vars:
    repo_port: 5000
    repo: "localhost:{{repo_port}}"
    image: ansiblebook/nginx-ghost
  tasks:
    - name: start a registry locally
      docker_container:
        name: registry
        image: registry:2
        ports: "{{ repo_port }}:5000"
    - debug:
        msg: name={{ image }} repo={{ repo }}/{{ image }}
    - name: tag the nginx-ghost image to the repository
      docker_image:
        name: "{{ image }}"
        repository: "{{ repo }}/{{ image }}"
        push: yes
```

Проверку можно выполнить, загрузив файл манифеста:

```
$ curl http://localhost:5000/v2/ansiblebook/nginx-ghost/manifests/latest
```

```
{
    "schemaVersion": 1,
    "name": "ansiblebook/nginx-ghost",
    "tag": "latest",
    ...
}
```

Запрос информации о локальном образе

Модуль `docker_image_facts` позволяет запросить метаданные, описывающие образ, который хранится локально. Пример 15.8 демонстрирует сценарий, использующий этот модуль для получения информации из образа `ghost` об открытых портах и томах.

Пример 15.8 ❖ *image-facts.yml*

```
---
- name: get exposed ports and volumes
  hosts: localhost
  gather_facts: False
  vars:
    image: ghost
  tasks:
    - name: get image info
      docker_image_facts: 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-facts.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) => {
    "item": "/var/lib/ghost",
    "msg": "Volume: /var/lib/ghost"
}

PLAY RECAP **********************************************************************
localhost      : ok=6    changed=0    unreachable=0     failed=0
```

Развертывание приложения в контейнере Docker

По умолчанию в качестве базы данных Ghost использует SQLite. В промышленном окружении мы будем использовать базу данных Postgres по причинам, обсуждавшимся в главе 5.

Все приложение мы развернем на двух машинах. На одной (`ghost`) развернем контейнеры Ghost и Nginx. На другой (`postgres`) – контейнер Postgres, который будет действовать как постоянное хранилище для данных Ghost.

В этом примере предполагается, что где-то, например в *group_vars/all*, определены следующие переменные с параметрами настройки обеих машин:

- `database_name`;
- `database_user`;
- `database_password`.

Postgres

Для настройки контейнера Postgres мы должны передать имя пользователя базы данных, пароль и имя базы данных в переменных окружения. Нам также нужно смонтировать каталог на машине-носителе как том для хранения данных, чтобы хранимые данные не исчезли после остановки и удаления контейнера.

В примере 15.9 приводится сценарий, развертывающий контейнер Postgres. В нем определены только две задачи: одна создает каталог для хранения данных, а другая запускает контейнер Postgres. Обратите внимание: этот сценарий

предполагает, что на хосте `postgres` уже установлено программное обеспечение Docker Engine.

Пример 15.9 ❖ *postgres.yml*

```
- name: deploy postgres
  hosts: postgres
  become: True
  gather_facts: False
  vars:
    data_dir: /data/pgdata
  tasks:
    - name: create data dir with correct ownership
      file:
        path: "{{ data_dir }}"
        state: directory
    - name: start postgres container
      docker_container:
        name: postgres_ghost
        image: postgres:9.6
        ports:
          - "0.0.0.0:5432:5432"
        volumes:
          - "{{ data_dir }}:/var/lib/postgresql/data"
        env:
          POSTGRES_USER: "{{ database_user }}"
          POSTGRES_PASSWORD: "{{ database_password }}"
          POSTGRES_DB: "{{ database_name }}"
```

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 Postgres.

Чтобы связать контейнеры Nginx и Ghost, мы используем сети Docker. Сети замещают устаревший механизм ссылок `links`, использовавшийся ранее для связывания контейнеров. 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=ghostnet
```

Имя сети предпочтительнее хранить в переменной окружения, потому что оно понадобится во всех запускаемых нами контейнерах. Вот как выглядит фрагмент сценария, отвечающий за запуск сети:

```
- name: deploy ghost
  hosts: ghost
  become: True
  gather_facts: False
  vars:
    url: "https://{{ ansible_host }}"
    database_host: "{{ groups['postgres'][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 существование группы с именем `postgres`, 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 Postgres, а также предусмотреть запуск в режиме промышленной эксплуатации передачей флага `--production` команде `npm start`.

Мы также должны гарантировать запись файлов 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`:

```
- name: create ghostdata directory
  file:
    path: "{{ data_dir }}"
    state: directory
- name: generate the config file
  template: src=templates/config.js.j2 dest={{ data_dir }}/config.js
- name: start ghost container
  docker_container:
    name: ghost
    image: ghost
    command: npm start --production
    volumes:
      - "{{ data_dir }}:/var/lib/ghost"
    networks:
      - name: "{{ net_name }}"
```

Обратите внимание, что нам не пришлось объявлять никаких сетевых портов, 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:

```
- name: create certs directory
  file:
    path: "{{ certs_dir }}"
    state: directory
- 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 3650
    creates=certs/nginx.crt
- name: start nginx container
  docker_container:
    name: nginx_ghost
    image: ansiblebook/nginx-ghost
    pull: yes
    networks:
      - name: "{{ net_name }}"
    ports:
      - "0.0.0.0:80:80"
      - "0.0.0.0:443:443"
    volumes:
      - "{{ certs_dir }}:/certs"
```

Удаление контейнеров

Ansible предлагает простой способ остановки и удаления контейнеров, который может пригодиться в процессе разработки и тестирования сценариев развертывания. Вот сценарий, который очищает хост `ghost`.

```
- name: remove all ghost containers and networks
  hosts: ghost
  become: True
  gather_facts: False
  tasks:
    - name: remove containers
      docker_container:
        name: "{{ item }}"
        state: absent
      with_items:
        - nginx_ghost
        - ghost
    - name: remove network
      docker_network:
        name: ghostnet
        state: absent
```

Прямое подключение к контейнерам

Ansible обладает возможностью напрямую взаимодействовать с запущенными контейнерами. Плагин инвентаризации контейнеров Docker в Ansible автоматически генерирует реестр действующих и доступных хостов, а плагин соединений действует подобно команде `docker exec`, позволяя запускать процессы в контексте выполняющихся контейнеров.

Плагин инвентаризации контейнеров Docker доступен в репозитории *ansible/ansible* на GitHub как *contrib/inventory/docker.py*. По умолчанию этот плагин обращается к демону Docker на локальной машине. С его помощью можно также взаимодействовать с демонами Docker на удаленных машинах посредством Docker REST API или серверов SSH, выполняющихся в контейнерах. Но оба варианта требуют дополнительной настройки. Для удаленного доступа к Docker REST API-хост, на котором запущен Docker, должен открыть порт TCP. Чтобы подключиться к контейнеру через SSH, в контейнере должен быть настроен запуск сервера SSH. Мы не будем рассматривать эти случаи здесь, но вы сможете найти пример конфигурационного файла в репозитории: *contrib/inventory/docker.yml*.

Допустим, что на локальной машине запущены следующие контейнеры:

```
CONTAINER ID   IMAGE                     NAMES
63b6767de77f   ansiblebook/nginx-ghost   ch14_nginx_1
057d72a95016   ghost                     ch14_ghost_1
```

В этом случае сценарий инвентаризации *docker.py* создаст следующий список хостов:

- `ch14_nginx_1`;
- `ch14_ghost_1`.

Он также создаст группы, соответствующие коротким идентификаторам, длинным идентификаторам, образам Docker, а также группу со всеми действующими контейнерами. В данном случае будут созданы следующие группы:

- `63b6767de77fe` (`ch14_nginx_1`);
- `63b6767de77fe01aa6d840dd897329766bbd3dc60409001cc36e900f8d501d6d` (`ch14_nginx_1`);
- `057d72a950163` (`ch14_ghost_1`);
- `057d72a950163769c2bcc1ecc81ba377d03c39b1d19f8f4a9f0c748230b42c5c` (`ch14_ghost_1`)$
- `image_ansiblebook/nginx-ghost` (`ch14_nginx_1`);
- `image_ghost` (`ch14_ghost_1`);
- `running` (`ch14_nginx_1`, `ch14_ghost_1`).

Вот как можно использовать сценарий динамической инвентаризации Docker с плагином подключения к Docker (включается передачей флага `-c` команде `docker`), чтобы получить список всех процессов, выполняющихся в каждом контейнере:

```
$ ansible -c docker running -m raw -a 'ps aux'
ch14_ghost_1 | SUCCESS | rc=0 >>
USER       PID %CPU %MEM    VSZ    RSS TTY     STAT START   TIME COMMAND
```

```
user         1  0.0  2.2 1077892  45040 ?      Ssl  05:19   0:00 npm
user        34  0.0  0.0    4340    804 ?      S    05:19   0:00 sh -c node ind
user        35  0.0  5.9 1255292 121728 ?      Sl   05:19   0:02 node index
root       108  0.0  0.0    4336    724 ?      Ss   06:20   0:00 /bin/sh -c ps
root       114  0.0  0.1   17500   2076 ?      R    06:20   0:00 ps aux

ch14_nginx_1 | SUCCESS | rc=0 >>
USER       PID %CPU %MEM    VSZ    RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.2   46320   5668 ?      Ss   05:19   0:00 nginx: master
nginx        6  0.0  0.1   46736   3020 ?      S    05:19   0:00 nginx: worker
root        71  0.0  0.0    4336    752 ?      Ss   06:20   0:00 /bin/sh -c ps
root        77  0.0  0.0   17500   2028 ?      R    06:20   0:00 ps aux
```

Контейнеры Ansible

Одновременно с версией Ansible 2.1 проект Ansible выпустил новый инструмент с названием Ansible Container, упрощающий работу с образами и контейнерами Docker. Далее мы рассмотрим версию Ansible Container 0.9, вышедшую одновременно с Ansible 2.3.

Ansible Container обладает довольно богатыми возможностями. В частности, его можно использовать для:

- создания новых образов (взамен файлов Dockerfiles);
- публикации образов Docker в реестрах (взамен `docker push`);
- запуска контейнеров Docker в режиме разработки (взамен Docker Compose);
- развертывания в промышленном окружении (взамен Docker Swarm).

На момент написания этих строк Ansible Container поддерживал развертывание в Kubernetes и OpenShift, хотя этот список, скорее всего, будет расти. Если вы не пользуетесь ни одним из этих окружений, не волнуйтесь: вы сможете писать сценарии, использующие модуль `docker_container` (как описывается далее в этой главе) для остановки и запуска своих контейнеров в любом окружении, имеющемся у вас.

Контейнер Conductor

Ansible Container позволяет настраивать образы Docker, используя роли Ansible вместо файлов Dockerfiles. Когда для настройки хостов используется система Ansible, на них должен быть установлен интерпретатор Python. Но в общем случае, когда используются контейнеры Docker, такое требование оказывается нежелательным, потому что пользователи обычно стремятся создавать контейнеры как можно меньшего размера и не горят желанием устанавливать Python в контейнер, если он фактически не нужен для работы.

Ansible Container избавляет от необходимости устанавливать Python в контейнеры благодаря использованию особого контейнера с названием *Conductor* и возможности Docker монтировать тома из одного контейнера в другой.

В момент запуска Ansible Container создает локальный каталог *ansible-deployment*, копирует в него все файлы, необходимые контейнеру Conductor, и монтирует этот каталог в контейнер Conductor.

Ansible Container монтирует каталоги со средой выполнения Python и всеми необходимыми библиотеками из контейнера Conductor в настраиваемые контейнеры. Для этого каталог */usr* из контейнера Conductor монтируется в каталог */_usr* внутри настраиваемого контейнера и затем Ansible настраивается на использование интерпретатора Python из каталога */_usr*. Чтобы это решение работало как надо, дистрибутив Linux в контейнере Conductor должен совпадать с дистрибутивом Linux образа Docker, служащего основой для других контейнеров, подлежащих настройке.

Если базовый образ для контейнера Conductor является официальным образом одного из поддерживаемых дистрибутивов Linux, Ansible Container автоматически добавит в контейнер необходимые пакеты. Версия 0.9.0 поддерживала такие дистрибутивы, как Fedora, CentOS, Debian, Ubuntu и Alpine. Вы можете взять за основу неподдерживаемый образ, но в этом случае вам придется самим обеспечить установку всех необходимых пакетов.

Более полный перечень пакетов, которые должны устанавливаться в контейнер Conductor, вы найдете в файле *container/docker/templates/conductor-dockerfile.j2*, в репозитории Ansible Container на GitHub (https://github.com/ansible/ansible-container).

Если вы не хотите, чтобы Ansible Container монтировал среду выполнения из контейнера Conductor в настраиваемые контейнеры, передайте флаг `--use-local-python` команде `ansible-container`. В этом случае Ansible Container будет использовать интерпретатор Python, установленный в образ настраиваемого контейнера.

Создание образов Docker

Давайте задействуем Ansible Container для создания простого образа Nginx на основе примера 15.1.

Создание начальных файлов

В первую очередь нужно выполнить команду инициализации:

```
$ ansible-container init
```

Эта команда создаст набор файлов в текущем каталоге:

```
.
├── ansible-requirements.txt
├── ansible.cfg
├── container.yml
├── meta.yml
└── requirements.yml
```

Создание ролей

Далее мы должны создать роль для настройки контейнера. Назовем ее `ghost-nginx`, потому что она отвечает за настройку образа для обслуживания Ghost.

Это очень простая роль; ей нужны лишь конфигурационный файл *ghost.conf* из примера 15.2 и файл с задачами, реализующими пример 15.1. Вот как выглядит структура каталогов для роли:

```
.
└── roles
    └── ghost-nginx
        ├── files
        │   └── ghost.conf
        └── tasks
            └── main.yml
```

А вот содержимое файла *tasks/main.yml*:

```
---
- name: remove default config
  file:
    path: /etc/nginx/conf.d/default.conf
    state: absent
- name: add ghost config
  copy:
    src: ghost.conf
    dest: /etc/nginx/conf.d/ghost.conf
```

Настройка container.yml

Далее добавим сценарий *container.yml*, использующий роль, описанную выше, для создания контейнера, как показано в примере 15.10. Он напоминает сценарий *docker-compose.yml* и добавляет дополнительные поля, характерные для Ansible, и использует поддержку фильтров и подстановки переменных в стиле Jinja2.

Пример. 15.10 ❖ *container.yml*

```
version: "2" ❶
settings:
  conductor_base: debian:jessie ❷
services: ❸
  ac-nginx: ❹
    from: nginx ❺
    command: [nginx, -g, daemon off;] ❻
    roles:
      - ghost-nginx ❼
registries: {} ❽
```

❶ Это выражение сообщает инструменту Ansible Container, что поддерживается версия 2 схем Docker Compose. По умолчанию используется версия 1, но вы почти всегда должны использовать версию 2.

❷ В качестве основы для контейнера Conductor используется образ `debian:jessie`, потому что с его помощью предполагается настраивать официальный образ Nginx, который также основан на образе `debian:jessie`.
❸ Поле `services` – это словарь, ключами которого являются имена создаваемых нами контейнеров. В данном примере создается только один контейнер.
❹ Контейнеру присваивается имя `ac-nginx`, от *Ansible Conductor Nginx*.
❺ Задается базовый образ `nginx`.
❻ Мы должны указать команду, которую требуется вызвать сразу после запуска контейнера.
❼ Роли, которые должны использоваться для настройки этого образа. В данном случае используется только одна роль: `ghost-nginx`.
❽ Поле `registries` определяет внешние реестры для сохранения контейнеров. Мы еще не настроили ни одного реестра, поэтому поле оставлено пустым.

Ansible Container не загружает базовых образов на локальную машину. Это нужно сделать вручную, до сборки контейнеров. Например, прежде чем запустить пример 15.10, необходимый для сборки `ac-nginx`, базовый образ `nginx` можно загрузить так:

```
$ docker pull nginx
```

Сборка контейнеров

Наконец, можно выполнить сборку:

```
$ ansible-container build
```

Вот как должен выглядеть вывод этой команды:

```
Building Docker Engine context...
Starting Docker build of Ansible Container Conductor image (please be patient)...
Parsing conductor CLI args.
Docker™ daemon integration engine loaded. Build starting. project=ans-con
Building service...    project=ans-con service=ac-nginx

PLAY [ac-nginx] ****************************************************************

TASK [Gathering Facts] *********************************************************
ok: [ac-nginx]

TASK [ghost-nginx : remove default config] **************************************
changed: [ac-nginx]

TASK [ghost-nginx : add ghost config] ******************************************
changed: [ac-nginx]

PLAY RECAP *********************************************************************
ac-nginx :             ok=3    changed=2    unreachable=0    failed=0

Applied role to service role=ghost-nginx service=ac-nginx
Committed layer as image image=sha256:5eb75981fc5117b3fca3207b194f3fa6c9ccb85
7718f91d674ec53d86323ffe3 service=ac-nginx
Build complete. service=ac-nginx
All images successfully built.
Conductor terminated. Cleaning up. command_rc=0 conductor_id=8c68ca4720beae5d9c
7ca10ed70a3c08b207cd3f68868b3670dcc853abf9b62b save_container=False
```

Для именования образов Ansible Container использует соглашение {имя_проекта}-{имя_службы}; имя проекта определяется по имени каталога, в котором выполняется команда `ansible-container init`. В данном случае каталог называется *ans-con*, поэтому созданный образ получил имя *ans-con-ac-nginx*.

Кроме того, Ansible всегда создает образ контейнера Conductor, следуя шаблону {имя_проекта}-conductor.

Если нежелательно, чтобы в качестве имени проекта Ansible Container использовал имя каталога, можно передать параметр `--project-name` с требуемым именем.

Если теперь выполнить команду

```
$ docker images
```

она выведет следующие созданные образы контейнеров:

```
REPOSITORY         TAG             IMAGE ID      CREATED         SIZE
ans-con-ac-nginx   20170424035545 5eb75981fc51  2 minutes ago   182 MB
ans-con-ac-nginx   latest          5eb75981fc51  2 minutes ago   182 MB
ans-con-conductor  latest          742cf2e046a3  2 minutes ago   622 MB
```

Устранение неполадок во время сборки

Если команда сборки завершилась с ошибкой, выяснить ее причины можно, заглянув в журналы, генерируемые контейнером Conductor. Сделать это можно двумя способами.

Первый: использовать флаг `--debug` в вызове команды `ansible-container`.

Если по каким-то причинам повторный запуск команды с флагом `--debug` нежелателен, можно заглянуть в журнал, который генерирует Docker. Для этого нужно знать идентификатор контейнера Conductor. Поскольку этот контейнер больше не выполняется, используйте команду `docker ps -a`, чтобы вывести идентификаторы завершившихся контейнеров:

```
$ docker ps -a
CONTAINER ID  IMAGE         COMMAND                CREATED         STATUS
78e78b9a1863  0c238eaf1819  "/bin/sh -c 'cd /_..."  21 minutes ago  Exited (1)
```

Имея идентификатор, можно получить записи из журнала, как показано ниже:

```
$ docker logs 78e78b9a1863
```

Запуск на локальной машине

Ansible Container позволяет запустить несколько контейнеров локально, в точности как Docker Compose. Файл *container.yml* имеет такой же формат, как файл *docker-compose.yml*. Он показан в примере 15.11.

Пример 15.11 ❖ *container.yml* для запуска на локальной машине

```
version: "2"
settings:
  conductor_base: debian:jessie
```

```
services:
  ac-nginx:
    from: nginx
    command: [nginx, -g, daemon off;]
    roles:
      - ghost-nginx
    ports:
      - "8443:443"
      - "8000:80"
    dev_overrides: ❶
      volumes:
        - $PWD/certs:/certs
      links:
        - ghost
  ghost: ❷
    from: ghost
    dev_overrides:
      volumes:
        - $PWD/ghostdata:/var/lib/ghost
registries: {}
```

Обратите внимание на различия между примерами 15.10 и 15.11.

❶ В описание службы `ac-nginx` добавлен раздел `dev_overrides`, содержащий данные, характерные для запуска на локальной машине (то есть они не используются для создания образов или развертывания в промышленном окружении). В данном случае выполняется монтирование локального каталога с сертификатами TLS и определяется связь данного контейнера с контейнером `ghost`.

❷ Добавлена служба `ghost`, содержащая приложение Ghost. Прежде в этом не было необходимости, потому что мы не создавали свой контейнер Ghost, а просто запускали официальный, неизмененный контейнер.

Обратите также внимание на различия в синтаксисе с Docker Compose. Например, Ansible Container использует выражение `from`, тогда как Docker Compose использует `image`, и в Docker Compose нет раздела `dev_overrides`.

Запустить контейнеры на локальной машине можно командой

```
$ ansible-container run
```

а остановить – командой

```
$ ansible-container stop
```

Остановить все контейнеры и удалить все созданные образы можно командой

```
$ ansible-container destroy
```

Публикация образов в реестрах

Получив образы, удовлетворяющие требованиям, вы можете сохранить их (опубликовать) в реестре, чтобы затем использовать для развертывания.

Для этого необходимо настроить раздел `registries`, указав в нем нужный реестр. В примере 15.12 показано, как можно изменить *container.yml*, чтобы созданные образы сохранялись в Docker-реестре в пространстве имен *ansiblebook*.

Пример 15.12 ❖ Раздел registries в файле *container.yml*

```
registries:
  docker:
    url: https://index.docker.io/v1/
    namespace: ansiblebook
```

Аутентификация

Сохраняя образ в первый раз, необходимо передать свое имя пользователя в аргументе командной стоки:

```
$ ansible-container push --username $YOUR_USERNAME
```

Сразу после запуска команды вам будет предложено ввести пароль. В первой попытке сохранить образ Ansible Container запомнит ваши учетные данные в *~/.docker/config.json*, и в последующем вам не придется повторно указывать имя пользователя или вводить пароль.

Вот как будет выглядеть вывод предыдущей команды:

```
Parsing conductor CLI args.
Engine integration loaded. Preparing push. engine=Docker™ daemon
Tagging ansiblebook/ans-con-ac-nginx
Pushing ansiblebook/ans-con-ac-nginx:20170430055647...
The push refers to a repository [docker.io/ansiblebook/ans-con-ac-nginx]
Preparing
Pushing
Mounted from library/nginx
Pushed
20170430055647: digest: sha256:50507495a9538e9865fe3038d56793a1620b9b372482667a
Conductor terminated. Cleaning up. command_rc=0 conductor_id=1d4cfa04a055c1040
```

Несколько реестров

Ansible Container позволяет определить несколько реестров. Например, вот как может выглядеть раздел `registries` с двумя реестрами, Docker Hub и Quay:

```
registries:
  docker:
    namespace: ansiblebook
    url: https://index.docker.io/v1/
  quay:
    namespace: ansiblebook
    url: https://quay.io
```

Чтобы сохранить образы только в один реестр, используйте флаг `--push-to`. Например, следующая команда сохранит образы в реестр Quay:

```
$ ansible-container push --push-to quay
```

Развертывание контейнеров в промышленном окружении

Хотя мы не рассматривали этого вопроса, тем не менее Ansible Container также поддерживает развертывание контейнеров в промышленном окружении, для чего можно воспользоваться командой `ansible-container deploy`. На момент написания этих строк Ansible Container поддерживал развертывание на двух платформах управления контейнерами: OpenShift и Kubernetes.

Если для запуска своих контейнеров вы ищете публичное облако, поддерживаемое Ansible Container, обратите внимание на облачные платформы OpenShift Online (основанная на OpenShift и управляемая компанией Red Hat) и Kubernetes (часть облачной платформы Google Compute Engine). Обе платформы также являются проектами с открытым исходным кодом, поэтому при наличии собственного аппаратного обеспечения вы можете бесплатно развернуть свое облако OpenShift или Kubernetes. Если потребуется развернуть свой проект на какой-то другой платформе (например, EC2 Container Service или Azure Container Service), вы не сможете использовать для этого Ansible Container.

Технология Docker ясно продемонстрировала широту своих возможностей. В этой главе мы узнали, как управлять образами, контейнерами и сетями Docker. Несмотря на то что мы не рассматривали тему создания образов Docker в сценариях Ansible, к тому моменту, когда вы будете читать эти строки, вы почти наверняка будете достаточно полно представлять, как это можно сделать.

Глава 16

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, который будет искать причину проблемы. Вот пример сообщения об ошибке, с которой мы столкнулись, работая над этой книгой:

```
TASK [check out the repository on the host] ************************************
fatal: [web]: FAILED! => {"changed": false, "cmd": "/usr/bin/git clone --origin o
rigin '' /home/vagrant/mezzanine/mezzanine_example", "failed": true, "msg": "Clon
ing into '/home/vagrant/mezzanine/mezzanine_example'...\nPermission denied (publi
ckey).\r\nfatal: Could not read from remote repository.\n\nPlease make sure you h
ave the correct access rights\nand the repository exists.", "rc": 128, "stderr":
"Cloning into '/home/vagrant/mezzanine/mezzanine_example'...\nPermission 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"
: ["Cloning into '/home/vagrant/mezzanine/mezzanine_example'...", "Permission den
ied (publickey).", "fatal: Could not read from remote repository.", "", "Please m
ake sure you have the correct access rights", "and the repository exists."], "std
out": "", "stdout_lines": []}
```

Как упоминалось в главе 10, плагин обратного вызова `debug` может привести это сообщение к более удобочитаемому виду:

```
TASK [check out the repository on the host] ************************************
fatal: [web]: FAILED! => {
    "changed": false,
    "cmd": "/usr/bin/git clone --origin origin '' /home/vagrant/mezzanine/mezzani
    ne_example",
    "failed": true,
    "rc": 128
}

STDERR:
```

```
Cloning into '/home/vagrant/mezzanine/mezzanine_example'...
Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

MSG:

Cloning into '/home/vagrant/mezzanine/mezzanine_example'...
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
```

Отладка ошибок с SSH-подключением

Иногда Ansible не удается установить SSH-соединение с хостом. В этом случае полезно проверить, какие аргументы Ansible передает SSH-клиенту, и воспроизвести действие вручную в командной строке.

Если вызвать `ansible-playbook` с аргументом `-vvv`, можно увидеть, как именно Ansible вызывает SSH. Это может пригодиться для отладки.

В примере 16.1 показано, что вывела Ansible, попытавшись вызвать модуль, чтобы скопировать файл.

Пример 16.1 ❖ Пример вывода при передаче флага, включающего подробный вывод

```
TASK: [copy TLS key] ******************************************************
task path: /Users/lorin/dev/ansiblebook/ch15/playbooks/playbook.yml:5
Using module file /usr/local/lib/python2.7/site-packages/ansible/modules/core/
files/stat.py
<127.0.0.1> SSH: EXEC ssh -C -o ControlMaster=auto -o ControlPersist=60s -o
StrictHostKeyChecking=no -o Port=2222 -o 'IdentityFile=".vagrant/machines/default/
virtualbox/private_key"' -o KbdInteractiveAuthentication=no -o
PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey -o
PasswordAuthentication=no -o User=vagrant -o ConnectTimeout=10 -o ControlPath=
/Users/lorin/.ansible/cp/ansible-ssh-%h-%p-%r 127.0.0.1 '/bin/sh -c '"'"'( umask
77 && mkdir -p "` echo ~/.ansible/tmp/ansible-tmp-1487128449.23-168248620529755 `"
&& echo ansible-tmp-1487128449.23-168248620529755="` echo ~/.ansible/tmp/ansibletmp-
1487128449.23-168248620529755 `" ) && sleep 0'"'"''
<127.0.0.1> PUT /var/folders/g_/523vq6g1037d10231mmbx1780000gp/T/tmpyOxLAA TO
/home/vagrant/.ansible/tmp/ansible-tmp-1487128449.23-168248620529755/stat.py
<127.0.0.1> SSH: EXEC sftp -b - -C -o ControlMaster=auto -o ControlPersist=60s -o
StrictHostKeyChecking=no -o Port=2222 -o 'IdentityFile=".vagrant/machines/default/
virtualbox/private_key"' -o KbdInteractiveAuthentication=no -o
```

```
PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey -o
PasswordAuthentication=no -o User=vagrant -o ConnectTimeout=10 -o ControlPath=
/Users/lorin/.ansible/cp/ansible-ssh-%h-%p-%r '[127.0.0.1]'
```

Иногда, при отладке проблем с подключением, может даже понадобиться использовать флаг -vvvv, чтобы увидеть сообщение об ошибке, возвращаемое SSH-клиентом. Например, на хосте не запущен сервер SSH, вы увидите примерно такую ошибку:

```
testserver | FAILED => SSH encountered an unknown error. The output was:
OpenSSH_6.2p2, OSSLShim 0.9.8r 8 Dec 2011
debug1: Reading configuration data /etc/ssh_config
debug1: /etc/ssh_config line 20: Applying options for *
debug1: /etc/ssh_config line 102: Applying options for *
debug1: auto-mux: Trying existing master
debug1: Control socket "/Users/lorin/.ansible/cp/ansible-ssh-127.0.0.1-
2222-vagrant" does not exist
debug2: ssh_connect: needpriv 0
debug1: Connecting to 127.0.0.1 [127.0.0.1] port 2222.
debug2: fd 3 setting O_NONBLOCK
debug1: connect to address 127.0.0.1 port 2222: Connection refused
ssh: connect to host 127.0.0.1 port 2222: Connection refused
```

Если включена проверка ключей хостов и выявится несоответствие ключа хоста с ключом в *~/.ssh/known_hosts*, аргумент -vvvv поможет обнаружить эту ошибку:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
c3:99:c2:8f:18:ef:68:fe:ca:86:a9:f5:95:9e:a7:23.
Please contact your system administrator.
Add correct host key in /Users/lorin/.ssh/known_hosts to get rid of this
message.
Offending RSA key in /Users/lorin/.ssh/known_hosts:1
RSA host key for [127.0.0.1]:2222 has changed and you have requested strict
checking.
Host key verification failed.
```

Если дело в этом, удалите старую запись из файла *~/.ssh/known_hosts*.

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 16.2.

Пример 16.2 ❖ Модуль debug в действии

```
- debug: var=myvariable
- debug: msg="The value of myvariable is {{ var }}"
```

Как уже говорилось в главе 4, можно вывести значения всех переменных, связанных с текущим хостом, как показано ниже:

```
- debug: var=hostvars[inventory_hostname]
```

Интерактивный отладчик сценариев

В Ansible 2.1 была добавлена поддержка интерактивного отладчика. Чтобы включить режим отладки, добавьте `strategy: debug` в свою операцию, например:

```
- name: an example play
  strategy: debug
  tasks:
    ...
```

Когда включен режим отладки, Ansible запускает отладчика в случае появления ошибки в задаче:

```
TASK [try to apt install a package] ********************************************
fatal: [localhost]: FAILED! => {"changed": false, "cmd": "apt-get update",
"failed": true, "msg": "[Errno 2] No such file or directory", "rc": 2}
Debugger invoked
(debug)
```

В табл. 16.1 перечислены команды, поддерживаемые отладчиком.

Таблица 16.1. Команды отладчика

Команда	Описание
`p var`	Вывести значение переменой
`task.args[key]=value`	Изменить аргумент задачи, допустившей ошибку
`vars[key]=value`	Изменить значение переменной
`r`	Перезапустить задачу
`c`	Продолжить выполнение операции
`q`	Прервать операцию и завершить работу отладчика
`help`	Показать справку

В табл. 16.2 перечислены переменные, поддерживаемые отладчиком.

Таблица 16.2. Переменные, поддерживаемые отладчиком

Команда	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

Вот пример сеанса работы с отладчиком:

```
(debug) p task
TASK: try to apt install a package
(debug) p task.args
{u'name': u'foo'}
(debug) p result
{'_ansible_no_log': False,
 '_ansible_parsed': True,
 'changed': False,
 u'cmd': u'apt-get update',
 u'failed': True,
 'invocation': {u'module_args': {u'allow_unauthenticated': False,
                                 u'autoremove': False,
                                 u'cache_valid_time': 0,
                                 u'deb': None,
                                 u'default_release': None,
                                 u'dpkg_options': u'force-confdef,force-confold',
                                 u'force': False,
                                 u'install_recommends': None,
                                 u'name': u'foo',
                                 u'only_upgrade': False,
                                 u'package': [u'foo'],
                                 u'purge': False,
                                 u'state': u'present',
                                 u'update_cache': False,
                                 u'upgrade': None},
                'module_name': u'apt'},
 u'msg': u'[Errno 2] No such file or directory',
 u'rc': 2}
(debug) p vars['inventory_hostname']
u'localhost'
(debug) p vars
{u'ansible_all_ipv4_addresses': [u'192.168.86.113'],
 u'ansible_all_ipv6_addresses': [u'fe80::f89b:ffff:fe32:5e5%awdl0',
                                 u'fe80::3e60:8f83:34b5:fc17%utun0',
                                 u'fe80::9679:241b:e93:8b7f%utun2'],
 u'ansible_architecture': u'x86_64',
 ...
```

Вывод значений переменных – одна из самых полезных возможностей, однако отладчик позволяет также изменять переменные и аргументы задачи, потерпевшей неудачу. За более подробной информацией обращайтесь к документации с описанием отладчика Ansible (*http://bit.ly/2lvAm8B*).

Модуль assert

Модуль `assert` завершает выполнение с ошибкой при невыполнении заданного условия. Например, сценарий завершится с ошибкой, если не будет найден сетевой интерфейс `eth1`:

```
- name: assert that eth1 interface exists
  assert:
    that: ansible_eth1 is defined
```

Такая проверка тех или иных условий может очень пригодиться при отладке сценария.

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"
```

К сожалению, движок Jinja2 не поддерживает встроенную функцию `len`. Вместо нее следует использовать Jinja2-фильтр `length`:

```
assert:
  that: "ports|length == 1"
```

Чтобы проверить статус файла в файловой системе хоста, можно сначала вызвать модуль `stat` и добавить проверку возвращаемого модулем значения:

```
- name: stat /opt/foo
  stat: path=/opt/foo
  register: st
- name: assert that /opt/foo 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. 16.3.

Таблица 16.3. Возвращаемые значения модуля stat

Поле	Описание
`atime`	Время последнего доступа к файлу в формате меток времени Unix
`ctime`	Время создания в формате меток времени Unix
`dev`	Числовой идентификатор устройства, где находится данный индексный узел
`exists`	`True`, если путь существует
`gid`	Числовой идентификатор 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 установлен бит set-group-ID
`islnk`	`True`, если файл – символическая ссылка
`isreg`	`True`, если файл – обычный файл
`issock`	`True`, если файл – сокет

Окончание табл. 16.3

Поле	Описание
isuid	True, если установлен бит set-user-ID
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
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
size	Размер файла в байтах, если это обычный файл
uid	Числовой идентификатор пользователя владельца
wgrp	True, если дано разрешение на запись для группы
woth	True, если дано разрешение на запись для остальных
wusr	True, если дано разрешение на запись для пользователя
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.

Проверка синтаксиса

Как показано в примере 16.3, флаг --syntax-check включает проверку допустимости синтаксиса сценария, но не запускает его.

Пример 16.3 ❖ Проверка синтаксиса

```
$ ansible-playbook --syntax-check playbook.yml
```

Список хостов

Как показано в примере 16.4, флаг --list-hosts выводит список хостов, на которых будет выполняться сценарий, но не запускает его.

Пример 16.4 ❖ Список хостов

```
$ ansible-playbook --list-hosts playbook.yml
```

Иногда можно получить ошибку:

```
ERROR: provided hosts list is empty
```

В реестре явно должен быть указан хотя бы один хост, иначе Ansible вернет эту ошибку, 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 следующую строку:

```
localhost ansible_connection=local
```

Список задач

Как показано в примере 16.5, флаг `--list-tasks` выводит список задач, которые запускает сценарий, но не запускает его.

Пример 16.5 ❖ Список задач

```
$ ansible-playbook --list-tasks playbook.yml
```

Мы уже использовали этот флаг в примере 6.1 для вывода списка задач в нашем первом сценарии для развертывания Mezzanine.

Проверка режима

Флаги `-C` и `--check` запускают Ansible в режиме проверки (также известном как *dry-run* – холостой запуск), который показывает, изменила бы каждая задача состояние хоста, но при этом не выполняют никаких изменений.

```
$ ansible-playbook -C playbook.yml
$ ansible-playbook --check playbook.yml
```

Одна из сложностей использования режима проверки состоит в том, что успех выполнения последующих частей сценария зависит от выполнения предыдущих. Если запустить в режиме проверки сценарий из примера 6.28, он вернет признак ошибки, как показано в примере 16.6, потому что данная задача зависит от предыдущей, устанавливающей программу Git на хост.

Пример 16.6 ❖ Ошибка при выполнении сценария в режиме проверки

```
GATHERING FACTS ***************************************************************
ok: [web]

TASK: [install apt packages] **************************************************
changed: [web] => (item=git,libjpeg-dev,libpq-dev,memcached,nginx,postgresql,py
thon-dev,python-pip,python-psycopg2,python-setuptools,python-virtualenv,supervi
sor)

TASK: [check out the repository on the host] **********************************
failed: [web] => {"failed": true}
msg: Failed to find required executable git

FATAL: all hosts have already failed -- aborting
```

В главе 12 уже рассказывалось, как модули реализуют режим проверки.

Вывод изменений в файлах

Флаги `-D` и `-diff` выводят информацию об изменениях, выполненных в любых 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`, `template` и `lineinfile`), изменения будут отображены в формате *.diff*:

```
TASK: [set the gunicorn config file] ***************************************
--- before: /home/vagrant/mezzanine-example/project/gunicorn.conf.py
+++ after: /Users/lorin/dev/ansiblebook/ch06/playbooks/templates/gunicor
n.conf.py.j2

@@ -1,7 +1,7 @@
 from __future__ import unicode_literals
 import multiprocessing

 bind = "127.0.0.1:8000"
 workers = multiprocessing.cpu_count() * 2 + 1
-loglevel = "error"
+loglevel = "warning"
 proc_name = "mezzanine-example"
```

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.

Пошаговое выполнение

Флаг `--step`, показанный в примере 16.7, заставляет Ansible запрашивать подтверждение на запуск каждой задачи:

```
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`).

Пример 16.7 ❖ Пошаговое выполнение

```
$ ansible-playbook --step playbook.yml
```

Выполнение с указанной задачи

Флаг `--start-at-task taskname`, показанный в примере 16.8, 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.

Пример 16.8 ❖ Выполнение с указанной задачи

```
$ ansible-playbook --start-at-task="install packages" playbook.yml
```

Теги

Ansible позволяет добавлять теги к задачам и операциям. Например, следующая операция отмечена тегом `foo`, а задача – тегами `bar` и `quux`:

```
- hosts: myservers
  tags:
    - foo
  tasks:
    - name: install editors
      apt: name={{ item }}
      with_items:
        - vim
        - emacs
        - nano

  - name: run arbitrary command
    command: /opt/myprog
    tags:
      - bar
      - quux
```

Добавив в команду флаг `-t` *имена_тегов* или `--tags` *имена_тегов*, можно потребовать от Ansible выполнить только операции и задачи, отмеченные определенными тегами. Добавив флаг `--skip-tags`, можно потребовать пропустить операции и задачи, отмеченные указанными тегами. Взгляните на пример 16.9.

Пример 16.9 ❖ Использование тегов

```
$ ansible-playbook -t foo,bar playbook.yml
$ ansible-playbook --tags=foo,bar playbook.yml
$ ansible-playbook --skip-tags=baz,quux playbook.yml
```

Глава 17

Управление хостами Windows

Ansible часто называют «системой управления конфигурациями на стероидах». Исторически 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 – таких как Windows для Linux – заключалась не только в реализации механизмов подключения к Windows, но и в использовании более универсальных имен (например, в переименовании переменной `ansible_ssh_host` в `ansible_host` и выражения `sudo` в `become`).

Поддержка Microsoft Windows впервые появилась в версии Ansible 1.7, но она вышла из статуса «бета» только в версии 2.1. Кроме того, запустить Ansible на хосте с Windows (то есть использовать его в качестве управляющей машины) можно только при использовании Windows Subsystem for Linux (WSL).

Также следует отметить, что богатство библиотеки модулей для Windows уступает богатству библиотеки модулей для Linux.

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. Ansible использует интегрированный механизм удаленного управления Windows Remote Management (WinRM), поддерживающий SOAP-подобный протокол.

WinRM – это наша главнейшая зависимость в Windows, и для взаимодействий с этим механизмом из Python нужно установить соответствующий пакет на управляющий хост:

```
$ sudo pip install pywinrm
```

По умолчанию система 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:

```
[windows]
win01.example.com
win02.example.com
```

После этого нужно добавить в *group_var/windows* конфигурацию подключения, чтобы все хосты в группе унаследовали ее.

В 2015 г. компания Microsoft объявила в своем блоге (*https://blogs.msdn.microsoft.com/powershell/2015/06/03/looking-forward-microsoft-support-for-secure-shell-ssh/*) о начале работ по реализации встроенной поддержки Secure Shell (SSH). Это означает, что в будущем системе Ansible не нужно будет использовать особую конфигурацию для подключения к хостам с Windows.

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.

```
ansible_user: Administrator
ansible_password: 2XLL43hDpQ1z
ansible_connection: winrm
```

Чтобы использовать другой порт для HTTPS или HTTP, настройте его и схему, как показано ниже:

```
ansible_winrm_scheme: https
ansible_port: 5999
```

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 г. компания Microsoft открыла исходный код PowerShell на условиях лицензии MIT. Исходные коды и двоичные пакеты последних версий для macOS, Ubuntu и CentOS можно найти на GitHub (*https://github.com/PowerShell/PowerShell*). На момент написания этих строк последней стабильной была версия PowerShell 5.1.

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.

На управляющую машину, то есть 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, в Ansible имеется сценарий (https://github.com/ansible/ansible/blob/devel/examples/scripts/ConfigureRemotingForAnsible.ps1).

Установить и запустить его можно парой команд, представленной в примере 17.1. Сценарий ничего не нарушит, если запустить его несколько раз.

Пример 17.1 ❖ Установка в Windows поддержки Ansible

```
wget http://bit.ly/1rHMn7b -OutFile .\ansible-setup.ps1
.\ansible-setup.ps1
```

`wget` – это псевдоним для *Invoke-WebRequest* из PowerShell.

Чтобы узнать установленную версию PowerShell, выполните следующую команду в консоли PowerShell:

```
$PSVersionTable
```

Вы должны увидеть вывод, показанный на рис. 17.1.

Мы настроили механизм подключения, а теперь для начала проверим доступность хоста с Windows, выполнив команду `win_ping`. Похожая на команду `ping` в GNU/Linux или Unix, она не использует протокол ICMP, а проверяет возможность установки соединения с Ansible:

```
$ ansible windows -i hosts -m win_ping
```

Если в ответ появится сообщение об ошибке, как показано в примере 17.2, необходимо или получить действительный публичный сертификат TLS/SSL, или добавить доверительную цепочку для существующего внутреннего удостоверяющего центра (Certificate Authority, CA).

Пример 17.2 ❖ Ошибка, вызванная недействительным сертификатом

```
$ ansible -m win_ping -i hosts windows
win01.example.com | UNREACHABLE! => {
    "changed": false,
    "msg": "ssl: (\"bad handshake: Error([('SSL routines', 'tls_process_server_certi
ficate', 'certificate verify failed')],)\",)",
    "unreachable": true
}
```

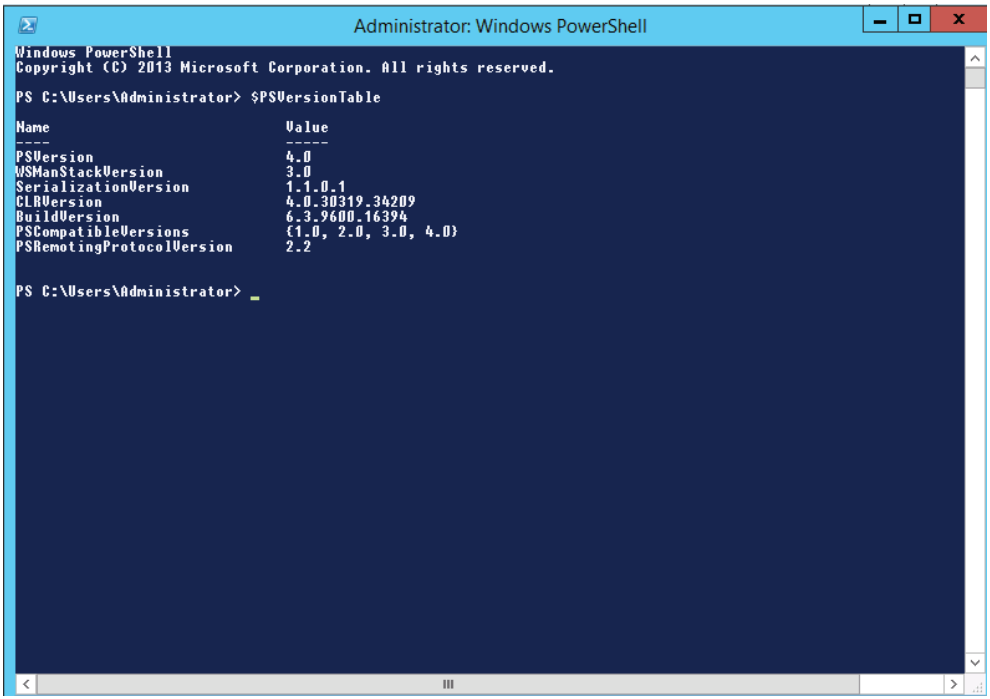

Рис. 17.1 ❖ Определение версии PowerShell

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 17.3, значит, проверка подключения выполнилась успешно.

Пример 17.3 ❖ Результат успешной проверки подключения

```
$ ansible -m win_ping -i hosts windows
win01.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Модули поддержки Windows

Имена модулей Ansible для Windows начинаются с префикса `win_`. На момент написания этих строк существовало более 40 таких модулей, из которых 19 считаются базовыми. Краткий обзор всех модулей для Windows можно найти в онлайн-документации (http://docs.ansible.com/ansible/latest/list_of_windows_modules.html).

В отношении имен модулей есть одно исключение: для сбора фактов из Windows модуль должен запускаться с именем `setup`, а не `win_setup`: `ansible -m setup -i hosts windows`.

Наш первый сценарий

Теперь, когда у нас есть хост с Windows, добавим его в нашу систему мониторинга. Для этого напишем сценарий Ansible, который будет использовать некоторые модули для Windows.

Для мониторинга была выбрана хорошо известная открытая система Zabbix, соответственно, мы должны установить *zabbix-agentd* на наш хост с Windows. Давайте создадим простой сценарий (см. пример 17.4), в котором опишем установку Zabbix Agent.

Пример 17.4 ❖ Сценарий для установки Zabbix Agent в Windows

```
---
- hosts: windows
  gather_facts: yes
  tasks:
    - name: install zabbix-agent
    win_chocolatey: ❶
      name: zabbix-agent

    - name: configure zabbix-agent
      win_template:
        src: zabbix_agentd.conf.j2
        dest: "C:\ProgramData\zabbix\zabbix_agentd.conf"
      notify: zabbix-agent restart

    - name: zabbix-agent restart
      win_service:
        name: Zabbix Agent
        state: started
  handlers:
    - name: zabbix-agent restart
      win_service:
        name: Zabbix Agent
        state: restarted
```

❶ `win_chocolatey` использует *chocolatey* – открытый диспетчер пакетов для Windows, распространяемый на условиях лицензии Apache License 2.0.

Сценарий в примере 17.4 не сильно отличается от того, что мы написали бы для Linux, разница только в используемых модулях.

Для установки программного обеспечения мы использовали диспетчер пакетов *chocolatey* (https://chocolatey.org/). Вместо него также можно было бы применить модуль `win_package`. Для конфигурирования мы задействовали модуль `win_template`, вместе с которым использовали факты (например, `ansible_hostname`).

Конечно, *zabbix_agentd.conf* необходимо скопировать с хоста Windows, прежде чем создавать его шаблон. Язык шаблонов идентичен используемому модулем `template`: Jinja2.

Последний модуль, задействованный в сценарии, – `win_service` – не требует пояснений.

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 17.5. 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 17.5 ❖ Сценарий для установки обновлений безопасности

```
---
- hosts: windows
  gather_facts: yes
  serial: 1 ❶
  tasks:
    - name: install software security updates
      win_updates:
        category_names:
          - SecurityUpdates
          - CriticalUpdates
      register: update_result

    - name: reboot windows if needed
      win_reboot:
        shutdown_timeout_sec: 1200 ❷
        msg: "Due to security updates this host will be rebooted in 20 minutes." ❸
      when: update_result.reboot_required
```

❶ Использовать `serial` для накатывания обновлений.
❷ Дать некоторое время системе для установки всех обновлений.
❸ Сообщить пользователям, что система будет перезагружена.

Давайте посмотрим, как он работает (см. пример 17.6).

Пример 17.6 ❖ Результаты работы сценария установки обновлений

```
$ ansible-playbook security-updates.yml -i hosts -v
No config file found; using defaults

PLAY [windows] *****************************************************************

TASK [Gathering Facts] *********************************************************
ok: [win01.example.com]

TASK [install software security updates] ***************************************
ok: [win01.example.com] => {"changed": false, "found_update_count": 0, "install
ed_update_count": 0, "reboot_required": false, "updates": {}} ❶

TASK [reboot windows if needed] ************************************************
skipping: [win01.example.com] => {"changed": false, "skip_reason": "Conditional
result was False", "skipped": true} ❷

PLAY RECAP *********************************************************************
win01.example.com :    ok=2    changed=0    unreachable=0    failed=0
```

❶ `win_updates` вернул `false` в значении `reboot_required`.
❷ Задача пропускается, потому что не выполнено условие `when: update_result.reboot_required`.

Все получилось! К сожалению, в данный момент у нас не было под рукой никаких обновлений безопасности, поэтому задача `reboot` была пропущена.

Добавление локальных пользователей

В этой части главы мы посмотрим, как создавать учетные записи пользователей и группы 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 17.7 создает две группы и две учетные записи пользователей, согласно списку словарей. В промышленном окружении словарь с пользователями мог бы находиться в `group_vars` или `host_vars`, но для удобочитаемости мы поместили его в сценарий.

Пример 17.7 ❖ Управление локальными группами и пользователями в Windows

```
- hosts: windows
  gather_facts: no
  tasks:
    - name: create user groups
      win_group:
```

```
      name: "{{ item }}"
    with_items:
      - application
      - deployments

  - name: create users
    win_user:
      name: "{{ item.name }}"
      password: "{{ item.password }}"
      groups: "{{ item.groups }}"
      password_expired: "{{ item.password_expired | default(false) }}" ❶
      groups_action: "{{ item.groups_action | default('add') }}" ❷
    with_items:
      - name: gil
        password: t3lCj1hU2Tnr
        groups:
          - Users
          - deployments
      - name: sarina
        password: S3cr3t!
        password_expired: true ❸
        groups:
          - Users
          - application
```

❶ По умолчанию срок действия пароля неограничен, если в словаре явно не указано иное.

❷ 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.

❸ Мы указали, что срок действия пароля Сабрины истек. Она должна будет выбрать новый пароль при следующей попытке входа.

Запустим его:

```
$ ansible-playbook users.yml -i hosts

PLAY [windows] *****************************************************************

TASK [create user groups] ******************************************************
changed: [win01.example.com] => (item=application)
changed: [win01.example.com] => (item=deployments)

TASK [create users] ************************************************************
changed: [win01.example.com] => (item={u'password': u't3lCj1hU2Tnr', u'name':
u'gil', u'groups': [u'Users', u'deployments']})
```

```
changed: [win01.example.com] => (item={u'password_expired': True, u'password':
u'S3cr3t!', u'name': u'sarina', u'groups': [u'Users', u'application']})

PLAY RECAP *********************************************************************
win01.example.com :    ok=2    changed=2    unreachable=0    failed=0
```

Как будто все работает, но давайте проверим.

Как можно видеть на рис. 17.2, группы были благополучно созданы. Отлично!

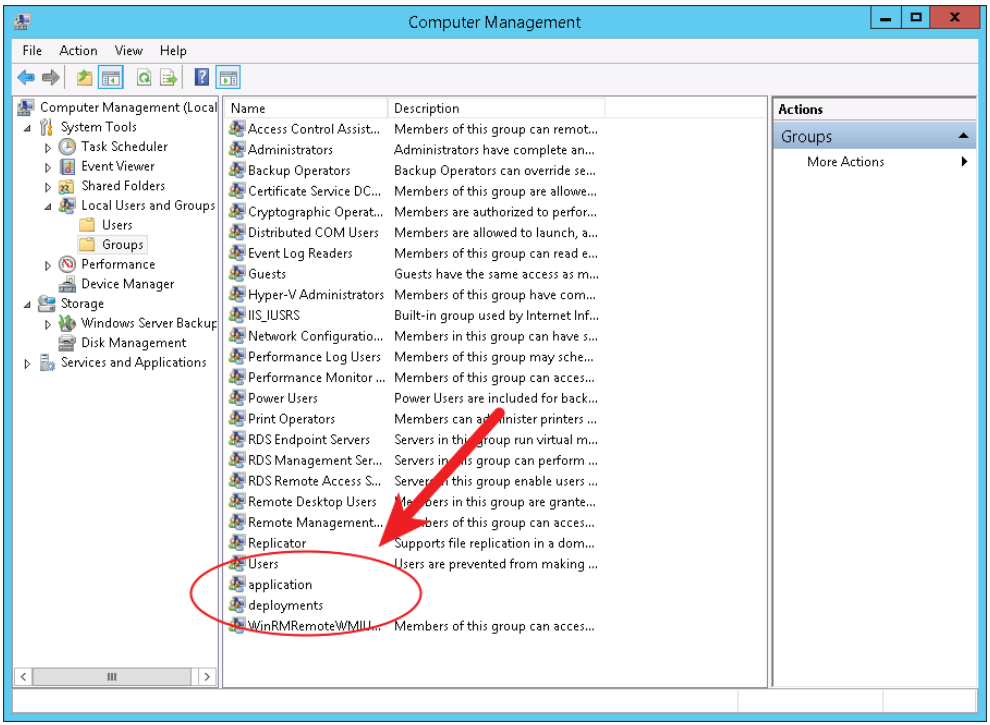

Рис. 17.2 ❖ Новые группы созданы

Проверим также учетные записи пользователей и посмотрим, какие настройки они получили. На рис. 17.3 можно видеть, что Ansible создала учетные записи пользователей и для `sarina` установлено требование сменить пароль при следующей попытке входа.

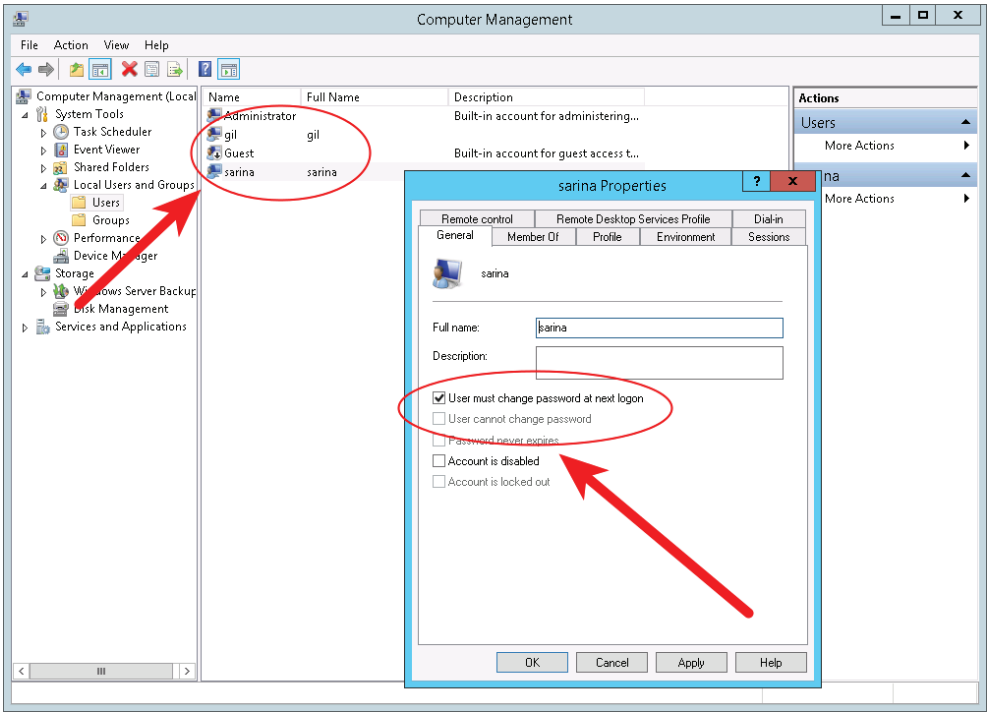

Рис. 17.3 ❖ Учетные записи новых пользователей созданы

Итоги

Ansible делает управление хостами с Microsoft Windows таким же простым, как управление хостами Linux и Unix.

Механизм Microsoft WinRM прекрасно работает, хотя и действует медленнее, чем протокол SSH. Будет интересно посмотреть, как улучшится производительность при использовании встроенной поддержки SSH для Windows и PowerShell.

Модули для Windows позволяют выполнять с помощью Ansible достаточно широкий круг задач, даже притом, что сообщество вокруг них пока еще невелико. Тем не менее Ansible – уже самый простой инструмент для управления парком хостов с разными операционными системами.

Глава 18

Ansible для сетевых устройств

Управление сетевыми устройствами и их настройка всегда вызывают у меня ностальгию. Вход с консоли через *telnet*, ввод нескольких команд, сохранение конфигурации – и работа сделана. Последовательность действий не меняется для этих устройств. Ладно-ладно, соглашусь, что некоторые изменения все же произошли, например появилась поддержка SSH.

Долгое время мы использовали две основные стратегии управления сетевыми устройствами:

- приобретение дорогостоящего патентованного программного обеспечения для настройки этих устройств;
- разработка 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 я заметил, – 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.

Как мне кажется, Ansible является одним из самых перспективных решений для задачи управления сетевыми устройствами по трем причинам:

- поддерживает сетевые устройства с консольным доступом через SSH и не ограниченные прикладными интерфейсами производителей;
- любой сетевой администратор может освоить этот стиль управления за час или даже меньше, потому что создание модулей мало чем отличается от привычной ему работы;
- Ansible – открытое программное обеспечение; мы можем использовать его здесь и сейчас!

Статус сетевых модулей

Прежде чем двинуться дальше, должен предупредить вас: сетевые модули все еще относительно новые – они продолжают развиваться и в настоящий момент находятся в стадии предварительных версий, предназначенных только для ознакомления. С течением времени ситуация изменится к лучшему. Но это не должно удерживать нас; мы с успехом можем использовать то, что уже имеется.

Список поддерживаемых производителей сетевого оборудования

Первый вопрос, который вы, скорее всего, зададите: «Поддерживается ли выбранный мной производитель сетевого оборудования или операционной системы?» Вот неполный, но довольно внушительный список поддерживаемых производителей и операционных систем:

- Cisco ASA, IOS, IOS XR, NX-OS;
- Juniper Junos OS;
- Dell Networking OS 6, 9 и 10;
- Cumulus;
- A10 Networks;
- F5 Networks;
- Arista EOS;
- VyOS.

Если вы не нашли своего производителя в списке, загляните в документацию, возможно, он уже поддерживается, потому что разработка сетевых модулей идет очень быстрыми темпами! На момент написания этих строк в состав Ansible входило около 200 модулей для взаимодействий с сетевыми устройствами.

Подготовка сетевого устройства

Прежде чем начать экспериментировать с сетевыми модулями, необходимо иметь, как вы уже поняли, само сетевое устройство.

Работая над книгой, я выпросил сетевое устройство. Этим устройством оказался не самый плохой, но довольно устаревший коммутатор Cisco Catalyst 2960G Series Layer 2, действующий под управлением IOS. Устройство было снято с производства в 2013 г. В этом устройстве нет ничего примечательного, кроме того что эта древняя штуковина может управляться с помощью Ansible!

Итак, прежде чем переходить к конфигурированию коммутатора с помощью Ansible, нужно проверить возможность соединения с ним. И тут нас поджидало первое препятствие – с заводскими настройками устройство принимало

соединения только по протоколу *telnet*. Мы должны привести его в состояние, когда оно будет принимать SSH-соединения, – нельзя использовать протокол *telnet* в промышленном окружении.

Ansible не поддерживает соединения с сетевыми устройствами через *telnet*.

Возможно, кто-то из вас уже настроил поддержку подключения по SSH в своих коммутаторах. Я не могу назвать себя опытным сетевым инженером; мне потребовалось время, чтобы узнать, как настроить поддержку SSH в моем коммутаторе Catalyst.

Настройка аутентификации через SSH

Для включения поддержки SSH необходимо выполнить несколько шагов. Команды, которые мы будем использовать, должны работать на большинстве устройств с IOS, но могут немного отличаться. Однако это не причина для волнений, потому что всегда остается возможность получить список допустимых параметров, введя в консоли знак вопроса (?).

Я сбросил настройки своего коммутатора Cisco в исходное состояние и перевел его в режим *Express Setup*. Так как все операции я выполнял в Linux, подключение к устройству через *telnet* не составило никакого труда (см. пример 18.1).

Пример 18.1 ❖ Вход через telnet

```
$ telnet 10.0.0.1
Trying 10.0.0.1...
Connected to 10.0.0.1.
Escape character is '^]'.
Switch#
```

Для настройки устройства его необходимо перевести в *режим настройки*, как показано в примере 18.2. Очевидный шаг, верно?

Пример 18.2 ❖ Перевод устройства в режим настройки

```
switch1#configure
Configuring from terminal, memory, or network [terminal]? terminal
Enter configuration commands, one per line. End with CNTL/Z.
```

Первое, что нужно сделать, – настроить IP-адрес, как показано в примере 18.3, чтобы можно было подключиться к устройству по окончании настройки.

Пример 18.3 ❖ Настройка статического IP-ареса

```
switch1(config)#interface vlan 1
switch1(config-if)#ip address 10.0.0.10 255.255.255.0
```

Чтобы сгенерировать ключ RSA, нужно присвоить устройству имя хоста и доменное имя, как показано в примере 18.4.

Пример 18.4 ❖ Настройка имени хоста и доменного имени

```
switch(config)#hostname switch1
switch1(config)#ip domain-name example.net
switch1(config)#
```

Теперь можно сгенерировать ключ *шифрования*, как показано в примере 18.5. Когда я писал эти строки, документация не рекомендовала генерировать ключи RSA с размером меньше 2048 бит.

Пример 18.5 ❖ Генерирование ключа RSA – это может потребовать некоторого времени

```
switch1(config)#crypto key generate rsa
The name for the keys will be: switch1.example.net
Choose the size of the key modulus in the range of 360 to 4096 for your
  General Purpose Keys. Choosing a key modulus greater than 512 may take
  a few minutes.

How many bits in the modulus [512]: 4096
% Generating 4096 bit RSA keys, keys will be non-exportable...
[OK] (elapsed time was 164 seconds)

switch1(config)#
```

Возможно, вы обратили внимание, что мы подключились к устройству по протоколу *telnet* без ввода учетных данных. В отличие от *telnet*, SSH всегда требует указывать имя пользователя и пароль.

Поэтому следующим шагом, который показан в примере 18.6, мы добавим нового пользователя, а также дадим ему уровень привилегий 15 (высший уровень).

Пароль можно установить двумя способами, как `secret` и как `password`. Пароль, установленный как `password`, хранится в открытом текстовом виде, тогда как `secret` сохранит пароль в виде хэш-суммы, тип которой зависит от устройства и версии прошивки.

Пример 18.6 ❖ Добавление нового пользователя admin

```
switch1(config)#username admin privilege 15 secret s3cr3t
```

Последний шаг, показанный в примере 18.7, – настройка модели аутентификации. Мой коммутатор по умолчанию использует *старую модель*. В этом режиме он запрашивает только пароль.

Но нам требуется, чтобы устройство запрашивало не только пароль, но также имя пользователя; это называется *новой моделью авторизации, аутентификации и учета* (Authentication, Authorization and Accounting – AAA).

Пример 18.7 ❖ Настройка модели аутентификации

```
switch1(config)#aaa new-model
```

Дополнительно установим пароль для привилегированного режима, как показано в примере 18.8, только чтобы показать, что Ansible также поддерживает эту особенность.

Пример 18.8 ❖ Установка пароля для привилегированного режима

```
switch1(config)#enable secret 3n4bl3s3cr3t
```

Теперь все готово и можно отключить простой текстовый и небезопасный протокол *telnet*, как показано в примере 18.9, на любом из 16 виртуальных терминалов.

Пример 18.9 ❖ Отключение поддержки telnet на устройстве

```
switch1(config)#line vty 0 15
switch1(config-line)#transport input ?
  all     All protocols
  none    No protocols
  ssh     TCP/IP SSH protocol
  telnet  TCP/IP Telnet protocol

switch1(config-line)#transport input ssh
switch1(config-line)#exit
```

Вот и все. Теперь сохраним конфигурацию и выйдем из режима настройки, как показано в примере 18.10. Имейте в виду, что после этого шага соединение с устройством может быть разорвано, но это не проблема.

Пример 18.10 ❖ Сохранение конфигурации в качестве используемой на запуске

```
switch1#copy running-config startup-config
Destination filename [startup-config]?
```

Теперь убедимся, что поддержка *telnet* выключена, а поддержка SSH включена, как показано в примере 18.11.

Пример 18.11 ❖ Вход через SSH

```
$ telnet 10.0.0.10
Trying 10.0.0.10...
telnet: Unable to connect to remote host: Connection refused
$ ssh admin@10.0.0.10
Password:
switch01>
```

Все работает!

Как работают модули

Прежде чем приступить к первому сценарию, вернемся немного назад и поговорим о том, как работают модули Ansible. Если говорить простыми словами, когда выполняется сценарий Ansible, модуль, используемый задачей, копируется на целевую машину и выполняется там.

В отношении сетевых модулей и сетевых устройств эта процедура не работает. Обычно на сетевых устройствах отсутствует интерпретатор Python или, по крайней мере, недоступен для нас. Именно поэтому сетевые модули действуют несколько иначе.

Их можно сравнить с модулями, взаимодействующими с HTTP API. Модули Ansible, использующие HTTP API, обычно выполняются локально – их код на языке Python взаимодействует с удаленным API по протоколу HTTP. Сетевые модули действуют примерно так же, только взаимодействуют не с HTTP API, а с консолью!

Наш первый сценарий

Я постараюсь сохранить этот первый сценарий максимально простым и реализую в нем только изменение имени хоста.

Так как наше сетевое устройство действует под управлением операционной системы Cisco IOS, мы используем модуль `ios_config`, который управляет конфигурационными разделами Cisco IOS.

Создадим первую задачу, `ios_config`, в сценарии, как показано в примере 18.12.

Пример 18.12 ❖ Изменение имени хоста в Cisco Catalyst

```
---
- hosts: localhost
  gather_facts: no
  connection: local ❶
  tasks:
    - name: set a hostname
      ios_config:
        lines: hostname sw1
        provider:
          host: 10.0.0.10 ❷
          username: admin ❸
          password: s3cr3t ❹
          authorize: true ❺
          auth_pass: 3n4bl3s3cr3t ❻
```

❶ Установить тип соединения `local`, чтобы все задачи обрабатывались системой Ansible как простые локальные действия.
❷ Доменное имя или IP-адрес сетевого устройства.
❸ Имя пользователя для входа на устройство через SSH.
❹ Пароль для входа на устройство.
❺ Выражением `authorize: true` мы сообщаем модулю, что команда должна выполняться в привилегированном режиме.
❻ Пароль для входа в привилегированный режим.

Вместо передачи аргументов `username`, `password`, `authorize` и `auth_pass` в каждой задаче можно определить следующие переменные окружения, которые автоматически будут использоваться взамен: `ANSIBLE_NET_USERNAME`, `ANSIBLE_NET_PASSWORD`, `ANSIBLE_NET_AUTHORIZE` и `ANSIBLE_NET_AUTH_PASS`.

Это может помочь уменьшить объем шаблонного кода в каждой задаче. Имейте в виду, что эти переменные окружения будут использоваться несколь-

кими сетевыми модулями. Однако каждую переменную всегда можно переопределить, явно передав модулю необходимые аргументы, как мы только что сделали это.

И это все? Да, это все. Давайте выполним сценарий:

```
$ ansible-playbook playbook.yml -v
No config file found; using defaults
[WARNING]: Host file not found: /etc/ansible/hosts

[WARNING]: provided hosts list is empty, only localhost is available

PLAY [localhost] ***************************************************************

TASK [set a hostname] **********************************************************
changed: [localhost] => {"changed": true, "updates": ["hostname sw1"],
"warnings": []}

PLAY RECAP *********************************************************************
localhost :            ok=1    changed=1    unreachable=0    failed=0
```

На первый взгляд все получилось, но, чтобы убедиться, попробуем зайти на устройство:

```
$ ssh admin@10.0.0.10
Password:
sw1>
```

Действительно все получилось! Мы благополучно выполнили свой первый сценарий, настраивающий Cisco Catalyst.

Сетевые модули пишутся с учетом поддержки идемпотентного выполнения. Мы можем выполнить сценарий сколько угодно раз, ничего при этом не нарушив!

Реестр и переменные для сетевых модулей

Возможно, вы заметили, что в последнем сценарии целевой хост был обозначен как `localhost`. Если бы у нас имелась ферма коммутаторов Cisco Catalyst, нам пришлось бы написать для каждого из них отдельный сценарий с целевым хостом `localhost`, потому что для каждого устройства нужны свои настройки и свои переменные.

Но давайте сделаем еще один шаг вперед и используем с сетевыми модулями уже знакомый нам прием: создадим статический файл реестра сетевых устройств, как показано в примере 18.13, и сохраним его с именем *./network_hosts*.

Пример 18.13 ❖ Файл хостов с сетевыми устройствами

```
[ios_switches]
sw1.example.com
```

Теперь можно поменять цель в сценарии на `ios_switches`, как показано в примере 18.14.

Пример 18.14 ❖ Изменение имени хоста в Cisco Catalyst

```
---
- hosts: ios_switches ❶
  gather_facts: no
  connection: local
  tasks:
    - name: set a hostname
      ios_config:
        lines: hostname sw1
        provider:
          host: 10.0.0.10
          username: admin
          password: s3cr3t
          authorize: true
          auth_pass: 3n4bl3s3cr3t
```

❶ Использовать `ios_switches` как цель.

Теперь, поскольку у нас есть реестр, можно использовать некоторые внутренние переменные Ansible. Переменная `inventory_hostname_short` содержит имя хоста из элемента в реестре (например, *sw1* из элемента *sw1.example.com*). То есть можно упростить сценарий, как показано в примере 18.15.

Пример 18.15 ❖ Использование `inventory_hostname_short` для настройки

```
---
- hosts: ios_switches
  gather_facts: no
  connection: local
  tasks:
  - name: set a hostname
    ios_config:
      lines: hostname {{ inventory_hostname_short }} ❶
      provider:
        host: 10.0.0.10
        username: admin
        password: s3cr3t
        authorize: true
        auth_pass: 3n4bl3s3cr3t
```

❶ Теперь можно использовать переменную `inventory_hostname_short`.

Локальное подключение

Как правило, сценарии для сетевых устройств должны выполняться с локальным подключением.

Вынесем этот параметр из сценария и поместим в файл *group_vars/ios_switches*, как показано в примере 18.16.

Пример 18.16 ❖ Файл с групповыми переменными для `ios_switches`

```
---
ansible_connection: local
```

Подключение к хосту

Из сценария в примере 18.15 также можно убрать конфигурационные параметры для модуля `ios_config`, которые наверняка будут отличаться для разных сетевых устройств (например, адрес `host` для подключения).

По аналогии с *hostname* мы можем использовать внутреннюю переменную; на этот раз `inventory_hostname`. В нашем случае `inventory_hostname` соответствует полному квалифицированному доменному имени *sw1.example.com*. Нам достаточно было бы, чтобы это доменное имя правильно распознавалось нашим сервером имен. Но на этапе разработки конфигурации дела могут обстоять иначе.

Чтобы не полагаться на DNS, мы добавим немного гибкости и создадим переменную `net_host`, которая будет использоваться для подключения. На крайний случай, если переменная `net_host` не будет определена, используем `inventory_hostname`.

На первый взгляд это условие может показаться сложным, но в действительности реализуется оно очень просто. Взгляните на пример 18.17.

Пример 18.17 ❖ Использование переменной для подключения

```
---
- hosts: ios_switches
  gather_facts: no
  tasks:
  - name: set a hostname
    ios_config:
      lines: hostname {{ inventory_hostname_short }}
      provider:
        host: "{{ net_host | default(inventory_hostname) }}" ❶
        username: admin
        password: s3cr3t
        authorize: true
        auth_pass: 3n4bl3s3cr3t
```

❶ Использовать для подключения переменную `net_host`, а если она не определена – переменную `inventory_hostname`.

Такие переменные обычно принято помещать в файл *hosts_vars*.

Так как эта переменная имеет некоторое отношение к подключению, возможно, лучше будет поместить ее в файл реестра *./network_hosts*, как показано в примере 18.18.

Пример 18.18 ❖ Добавление переменной `net_host` в соответствующую запись в реестре

```
[ios_switches]
sw1.example.com net_host=10.0.0.10
```

Переменные для аутентификации

Как последний шаг используем переменные для настройки параметров аутентификации. Это обеспечит нам максимальную гибкость.

Параметры аутентификации, общие для всех устройств в группе, можно определить в файле *group_vars*. Именно так мы и поступим (см. пример 18.19).

Пример 18.19 ❖ Файл с групповыми переменными для `ios_switches`

```
---
ansible_connection: local
net_username: admin
net_password: s3cr3t
net_authorize: true
net_auth_pass: 3n4bl3s3cr3t
```

Если для некоторых устройств используются иные параметры аутентификации, их можно переопределить на уровне *hosts_vars*.

Сохранение конфигурации

Пришло время реализовать сохранение конфигурации, чтобы гарантировать ее вступление в силу после следующей перезагрузки устройства. К счастью, для этого достаточно добавить в задачу `ios_config` параметр `save` со значением `true`.

Для любителей делать резервные копии Ansible предоставляет такую возможность. Если добавить параметр `backup` со значением `true`, задача сохранит резервную копию перед применением изменений.

Резервная копия конфигурации будет сохранена на управляющей машине в каталоге *backup*, рядом со сценарием. Если каталог *backup* отсутствует, Ansible автоматически создаст его:

```
$ ls backup/
switch1_config.2017-02-19@17:14:00
```

Файл резервной копии будет содержать действующую конфигурацию, а не начальную.

Новая версия сценария представлена в примере 18.20.

Пример 18.20 ❖ Окончательная версия сценария, устанавливающего имя хоста на устройстве Catalyst

```
---
- hosts: ios_switches
  gather_facts: no
  tasks:
  - name: set a hostname
    ios_config:
      lines: hostname {{ inventory_hostname_short }}
      provider:
        host: "{{ net_host | default(inventory_hostname) }}"
        username: "{{ net_username | default(omit) }}" ❶
        password: "{{ net_password | default(omit) }}" ❶
        authorize: "{{ net_authorize | default(omit) }}" ❶
```

```
    auth_pass: "{{ net_auth_pass | default(omit) }}" ❶
  backup: true ❷
  save: true ❸
```

❶ Все эти переменные можно определить на уровне *group_vars* или *host_vars*.
❷ Сохранит резервную копию действующей конфигурации в *./backup*.
❸ Сохранит `running-config` в `startup-config` на устройстве.

Параметры `backup` и `save` обрабатываются как отдельные операции. Они выполняются, даже если никаких изменений не было произведено. Я также заметил, что операция создания резервной копии не возвращает `changed=True`, а существующие резервные копии удаляются перед созданием новых.

Использование конфигураций из файлов

Параметр `lines` хорошо подходит для случаев, когда требуется изменить лишь несколько настроек. Однако я привык использовать подход, заключающийся в копировании конфигурации, хранящейся в локальном файле. Для этого я выполняю настройки в локальном файле и затем копирую его на устройство.

К счастью, `ios_config` принимает еще один параметр, позволяющий копировать конфигурационные файлы на устройства: параметр `src`. Благодаря этому параметру можно создать статический конфигурационный файл *ios_init_template.conf*, как показано в примере 18.21.

Пример 18.21 ❖ Пример статического конфигурационного файла для IOS

```
no service pad
service timestamps debug datetime msec
service timestamps log datetime msec
service password-encryption
boot-start-marker
boot-end-marker
aaa new-model
!
clock timezone CET 1 0
clock summer-time CEST recurring last Sun Mar 2:00 last Sun Oct 3:00
!
system mtu routing 1500
!
vtp mode transparent
!
ip dhcp snooping vlan 10-20
ip dhcp snooping
no ip domain-lookup
!
!
spanning-tree mode rapid-pvst
spanning-tree extend system-id
!
vlan internal allocation policy ascending
```

```
!
interface Vlan1
no ip address
no ip route-cache
shutdown
!
ip default-gateway 10.0.0.1
no ip http server
no ip http secure-server
!
snmp-server community private
snmp-server community public RO
snmp-server location earth
snmp-server contact admin@example.com
!
ntp server 10.123.0.5
ntp server 10.100.222.12
!
```

Не волнуйтесь! Я не собираюсь рассказывать, что значат все эти настройки. Вместо этого мы вернемся к нашему сценарию из предыдущего раздела и добавим в него задачу для копирования статического конфигурационного файла, как показано в примере 18.22.

Теперь в нашем сценарии две задачи настройки сетевого устройства. Использование параметра `backup` в двух задачах может привести к появлению большого количества промежуточных резервных копий, тогда как нам достаточно одной резервной копии действующей конфигурации, созданной перед любыми изменениями.

Поэтому добавим в начало сценария еще одну задачу, специально для создания резервной копии. По той же причине добавим обработчик, который будет вызывать `save`, только если произошли какие-то изменения.

Пример 18.22 ❖ Использование `src` для передачи статического конфигурационного файла

```
---
- hosts: ios_switches
  gather_facts: no
  tasks:
  - name: backup the running config
    ios_config:
      backup: true
      provider:
        host: "{{ net_host | default(inventory_hostname) }}"
        username: "{{ net_username | default(omit) }}"
        password: "{{ net_password | default(omit) }}"
        authorize: "{{ net_authorize | default(omit) }}"
        auth_pass: "{{ net_auth_pass | default(omit) }}"
  - name: init the static config
    ios_config:
      src: files/ios_init_config.conf ❶
```

```
    provider:
      host: "{{ net_host | default(inventory_hostname) }}"
      username: "{{ net_username | default(omit) }}"
      password: "{{ net_password | default(omit) }}"
      authorize: "{{ net_authorize | default(omit) }}"
      auth_pass: "{{ net_auth_pass | default(omit) }}"
  notify: save the running config ❷
- name: set a hostname
  ios_config:
    lines: hostname {{ inventory_hostname_short }}
    provider:
      host: "{{ net_host | default(inventory_hostname) }}"
      username: "{{ net_username | default(omit) }}"
      password: "{{ net_password | default(omit) }}"
      authorize: "{{ net_authorize | default(omit) }}"
      auth_pass: "{{ net_auth_pass | default(omit) }}"
    notify: save the running config ❷

handlers:
- name: save the running config
  ios_config:
    save: true
    provider:
      host: "{{ net_host | default(inventory_hostname) }}"
      username: "{{ net_username | default(omit) }}"
      password: "{{ net_password | default(omit) }}"
      authorize: "{{ net_authorize | default(omit) }}"
      auth_pass: "{{ net_auth_pass | default(omit) }}"
```

❶ Читает конфигурационный файл IOS *files/ios_init_config.conf*.
❷ Посылает уведомление обработчику сохранить конфигурацию.

Теперь мы умеем смешивать статические и динамические настройки. Конечно, мы можем продолжить идти тем же путем и расширить сценарий, добавив в него другие динамические настройки. Однако можно поступить иначе.

Но, прежде чем продолжить, обратите внимание, что сценарии от задачи к задаче повторяют немаленькие блоки `provider`. Мы можем устранить это повторение, как показано в примере 18.23.

Пример 18.23 ❖ Использование src для передачи статического конфигурационного файла

```
---
- hosts: ios_switches
  gather_facts: no
  vars:
    provider: ❶
      host: "{{ net_host | default(inventory_hostname) }}"
      username: "{{ net_username | default(omit) }}"
      password: "{{ net_password | default(omit) }}"
      authorize: "{{ net_authorize | default(omit) }}"
      auth_pass: "{{ net_auth_pass | default(omit) }}"
```

```
  tasks:
  - name: init the static config with backup before
    ios_config:
      backup: true ❷
      src: files/ios_init_config.conf
      provider: "{{ provider }}" ❸
    notify: save the running config

  - name: set a hostname
    ios_config:
      lines: hostname {{ inventory_hostname_short }}
      provider: "{{ provider }}" ❸
    notify: save the running config

  handlers:
  - name: save the running config
    ios_config:
      save: true
      provider: "{{ provider }}" ❸
```

❶ Выражение vars объявляет переменную provider для общего использования.
❷ Так как у нас только одна задача затрагивает конфигурацию, мы переместили параметр backup в эту задачу.
❸ Использование переменной provider variable.

Модуль ios_config можно вызвать с единственным параметром backup, чтобы получить начальный шаблон конфигурации.

Шаблоны, шаблоны, шаблоны

Теперь мы знаем, что параметр src модуля ios_config можно использовать для передачи статических файлов конфигурации. А можно ли использовать шаблоны Jinja2? К счастью, ios_config имеет встроенную поддержку шаблонов, как показано в примере 18.24.

Пример 18.24 ❖ Использование src для передачи статического конфигурационного файла и конфигурирование с помощью шаблона

```
---
- hosts: ios_switches
  gather_facts: no
  vars:
    provider:
      host: "{{ net_host | default(inventory_hostname) }}"
      username: "{{ net_username | default(omit) }}"
      password: "{{ net_password | default(omit) }}"
      authorize: "{{ net_authorize | default(omit) }}"
      auth_pass: "{{ net_auth_pass | default(omit) }}"
  tasks:
  - name: copy the static config
    ios_config:
```

```
    backup: true
    src: files/ios_init_config.conf.j2 ❶
    provider: "{{ provider }}"
  notify: save the running config

handlers:
- name: save the running config
  ios_config:
    save: true
    provider: "{{ provider }}"
```

❶ Мы создали шаблон из предыдущего статического конфигурационного файла и сохранили его в *files/ios_init_config.conf.j2*, следуя принятым соглашениям.

Мы превратили свой сценарий в адаптивный сценарий Ansible для настройки сетевых устройств, действующих под управлением IOS. Любые конфигурации сетевых устройств, статические или динамические, можно обрабатывать с помощью шаблона, показанного в примере 18.25.

Пример 18.25 ❖ Шаблон конфигурации IOS, включающий динамические настройки VLAN и интерфейсов

```
hostname {{ inventory_hostname_short }}

no service pad

service timestamps debug datetime msec
service timestamps log datetime msec
service password-encryption

boot-start-marker
boot-end-marker

clock timezone CET 1 0
clock summer-time CEST recurring last Sun Mar 2:00 last Sun Oct 3:00

ip dhcp snooping
no ip domain-lookup

spanning-tree mode rapid-pvst
spanning-tree extend system-id

vlan internal allocation policy ascending

!
{% if vlans is defined %} ❶
{% for vlan in vlans %}
vlan {{ vlan.id }}
name {{ vlan.name }}
!
{% endfor %}
{% endif %}

{% if ifaces is defined %} 
{% for iface in ifaces %}
```

```
interface {{ iface.name}}
  description {{ iface.descr }}
{% if iface.vlans is defined %}
{% endif %}
  switchport access vlan {{ iface.vlans | join(',') }}
  spanning-tree portfast
!
{% endfor %}
{% endif %}

no ip http server
no ip http secure-server

snmp-server community public RO
snmp-server location earth
snmp-server contact admin@example.com
! add more configs here...
```

❶ Пример использования динамических настроек в файле шаблона

Так как это обычный шаблон, в нем можно использовать все возможности механизма шаблонов Jinja2, включая наследование шаблонов и макросы. На момент написания эти строк `--diff` не возвращал различий между файлами в формате diff.

Давайте попробуем выполнить этот сценарий:

```
$ ansible-playbook playbook.yml -i network_hosts

PLAY [ios_switches] ************************************************************

TASK [copy the static config] **************************************************
changed: [switch1]

RUNNING HANDLER [save the running config] **************************************
changed: [switch1]

PLAY RECAP *********************************************************************
switch1 : ok=2 changed=2 unreachable=0 failed=0
```

Просто, не правда ли?

Сбор фактов

Сбор фактов из сетевых устройств реализует отдельный модуль – в данном случае `ios_facts`.

Устанавливайте параметр `gather_facts: false` в операциях с сетевыми устройствами в своих сценариях.

В предыдущем разделе мы уже подготовили все настройки соединения и теперь готовы перейти к сценарию, представленному в примере 18.26.

Модуль ios_facts имеет только один необязательный параметр: gather_subset. Этот параметр используется для ограничения желательных или фильтрации нежелательных фактов (с добавлением восклицательного знака). По умолчанию этот параметр принимает значение !config, что соответствует *всем фактам, кроме конфигурации*.

Пример 18.26 ❖ Сбор фактов из устройства IOS

```
---
- hosts: ios_switches
  gather_facts: no
  tasks:
  - name: gathering IOS facts
    ios_facts:
    gather_subset: hardware ❶
    host: "{{ net_host | default(inventory_hostname) }}"
    provider:
      username: "{{ net_username | default(omit) }}"
      password: "{{ net_password | default(omit) }}">
      authorize: "{{ net_authorize | default(omit) }}"
      auth_pass: "{{ net_auth_pass | default(omit) }}"
  - name: print out the IOS version
    debug:
      var: ansible_net_version ❷
```

❶ Собирать только факты об оборудовании.
❷ Все факты о сети начинаются с префикса ansible_net_.

Факты сохраняются в переменных хоста и не требуют регистрации (например, register: result) на уровне задач.

Попробуем запустить сценарий:

```
$ ansible-playbook facts.yml -i network_hosts -v
No config file found; using defaults

PLAY [ios_switches] ************************************************************

TASK [get some facts] **********************************************************
ok: [switch1] => {"ansible_facts": {"ansible_net_filesystems": ["flash:"], "ansi
ble_net_gather_subset": ["hardware", "default"], "ansible_net_hostname": "sw1",
"ansible_net_image": "flash:c2960-lanbasek9-mz.150-1.SE/c2960-lanbasek9-mz.150-1
.SE.bin", "ansible_net_memfree_mb": 17292, "ansible_net_memtotal_mb": 20841,
"ansible_net_model": null, "ansible_net_serialnum": "FOC1132Z0ZA", "ansible_net_
version": "15.0(1)SE"}, "changed": false, "failed_commands": []}

TASK [print out the IOS version] ***********************************************
ok: [switch1] => {
    "ansible_net_version": "15.0(1)SE"
}

PLAY RECAP *********************************************************************
switch1 : ok=2 changed=0 unreachable=0 failed=0
```

Итоги

Теперь вы получили первое представление, как управлять сетевыми устройствами, настраивать их и извлекать факты с помощью Ansible. Модули `ios_config` и `ios_facts` – обычные модули из множества других, аналогичных им, предназначенных для поддержки сетевых устройств с разными операционными системами (например, `dellos10_config` для Dell EMC Networking OS10 или `eos_config` для Arista EOS).

Но в зависимости от операционной системы и интерфейса, поддерживаемого сетевым устройством, количество и разнообразие модулей могут значительно отличаться. За подробной информацией о других модулях я рекомендую обращаться к документации (http://bit.ly/2uvBe2f).

Глава 19

Ansible Tower: Ansible для предприятий

Ansible Tower – коммерческий программный продукт, первоначально созданный в Ansible, Inc., а ныне предлагаемый компанией Red Hat. Ansible Tower реализован как классическая локальная веб-служба, действующая поверх Ansible. Эта служба поддерживает более тонкое управление пользователями, а политики доступа на основе ролей объединены с пользовательским веб-интерфейсом, пример которого показан на рис. 19.1, и RESTful API.

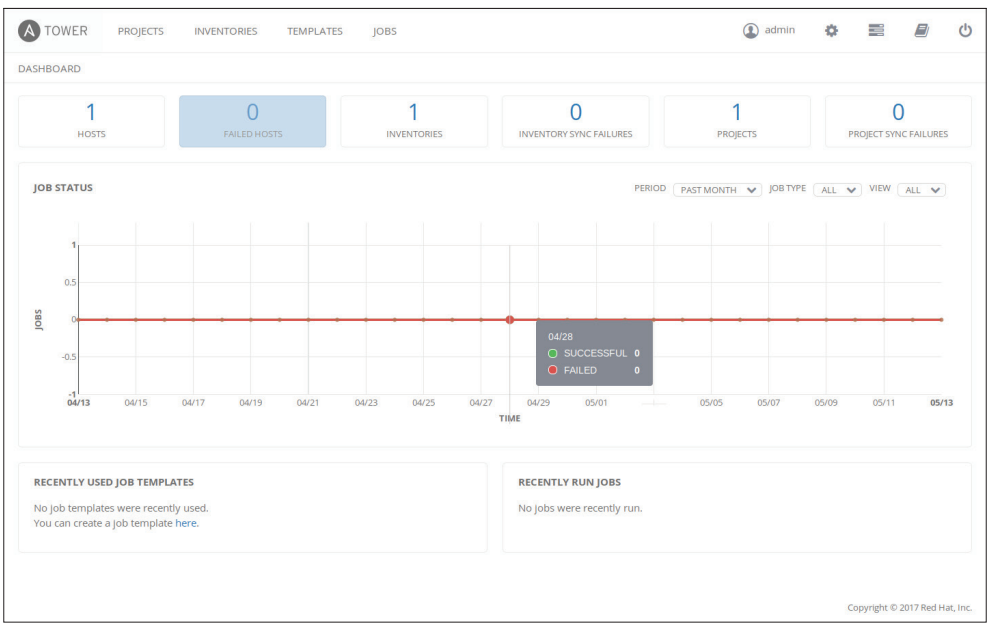

Рис. 19.1 ❖ Панель управления Ansible Tower

Модели подписки

Red Hat предлагает поддержку в виде ежегодной подписки трех типов, 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 Tower. Модель самостоятельной поддержки ограничивается 250 хостами и не включает следующих возможностей:

- изменения стандартного приветствия, отображаемого при входе;
- аутентификации через SAML, RADIUS и LDAP;
- поддержки нескольких организаций;
- потоков действий и систем протоколирования.

После того как в 2015 г. Red Hat приобрела Ansible, Inc., Red Hat выразила намерение продолжить разработку открытой версии Ansible Tower. Но на момент написания этих строк никакой дополнительной информации и никаких графиков выпуска не было обнародовано.

Пробная версия Ansible Tower

Red Hat предоставляет свободную пробную лицензию (https://www.ansible.com/license) с набором возможностей из модели подписки самостоятельной поддержки, до 10 управляемых хостов без ограничения срока использования.

Для быстрой оценки этой версии можно воспользоваться Vagrant:

```
$ vagrant init ansible/tower
$ vagrant up --provider virtualbox
$ vagrant ssh
```

После входа через SSH появится текст с приветствием, представленный в примере 19.1, где можно увидеть URL веб-интерфейса, имя пользователя и пароль.

Пример 19.1 ❖ Текст приветствия

```
Welcome to Ansible Tower!

Log into the web interface here:

  https://10.42.0.42/

  Username: admin
  Password: JSKYmEBJATFn

The documentation for Ansible Tower is available here:

  http://www.ansible.com/tower/

For help, visit http://support.ansible.com/
```

После входа в веб-интерфейс будет предложено заполнить форму для получения файла лицензии по электронной почте.

Если машина Vagrant недоступна по адресу 10.42.0.42, попробуйте выполнить следующую команду внутри нее, чтобы запустить сетевой интерфейс, связанный с этим IP-адресом:

```
$ sudo systemctl restart network.service
```

Какие задачи решает Ansible Tower

Ansible Tower – не просто веб-интерфейс к Ansible. Ansible Tower добавляет в Ansible некоторые дополнительные возможности. Рассмотрим их поближе в этом разделе.

Управление доступом

В крупных организациях с большим количеством отделов Ansible Tower помогает автоматизировать управление группами служащих с использованием ролей, наделяя их правами для управления хостами и устройствами, насколько это необходимо для выполнения служебных обязанностей.

Ansible Tower действует как защита для хостов. При использовании Ansible Tower ни одна группа и ни один работник не должны иметь прямого доступа к управляемым хостам. Это снижает сложность и увеличивает безопасность. На рис. 19.2 показан веб-интерфейс Ansible Tower для настройки прав доступа пользователей.

Подключение Ansible Tower к существующей системе аутентификации, такой как LDAP, может снизить затраты на администрирование пользователей.

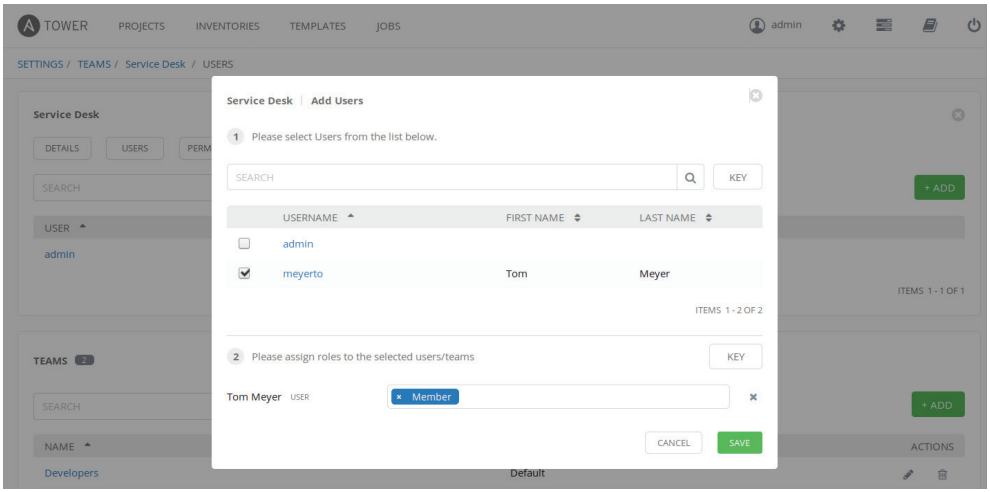

Рис. 19.2 ❖ Веб-интерфейс для настройки прав доступа пользователей

Проекты

Проектом в терминологии Ansible Tower называется пакет логически связанных сценариев и ролей.

В классических проектах Ansible вместе со сценариями и ролями часто можно видеть статические реестры. Ansible Tower осуществляет инвентаризацию иначе. 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 Tower 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 файловой системе, где проект будет храниться локально, на сервере Ansible Tower.

Так как проекты имеют свойство развиваться с течением времени, исходный код сценариев на сервере Ansible Tower должен синхронизироваться с содержимым системы управления версиями. Для этого в Ansible Tower имеется множество решений.

Например, гарантировать использование последних версий проектов в Ansible Tower можно, установив флажок «Update on Launch» (обновление на запуске) в параметрах проекта, как показано на рис. 19.3. Также можно настроить задания обновления проектов по расписанию. Наконец, проекты можно обновлять вручную, если вы хотите сами управлять обновлением.

Управление инвентаризацией

Ansible Tower позволяет управлять реестрами как самостоятельными ресурсами, включая управление доступом к этим реестрам. Типичный шаблон – определить разные реестры с хостами для эксплуатации, разработки и тестирования.

В каждом из реестров можно определять свои переменные по умолчанию и вручную добавлять группы и хосты. Кроме того, как показано на рис. 19.4, Ansible Tower позволяет запрашивать список хостов динамически из некоторого ресурса (например, из VMware vCenter) и помещать их в группу.

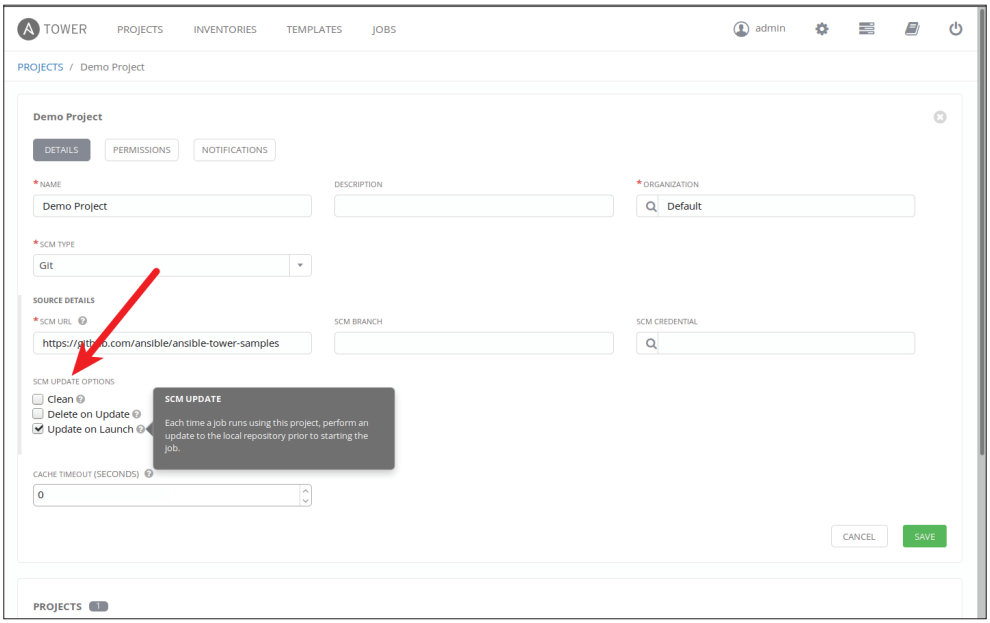

Рис. 19.3 ❖ Параметры настройки обновления проекта из системы управления версиями в Ansible Tower

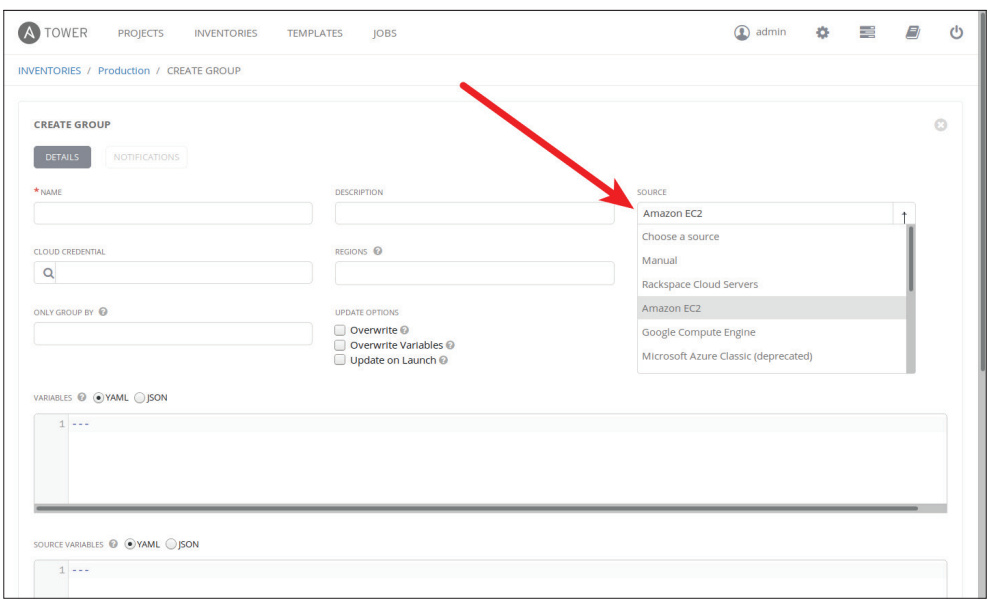

Рис. 19.4 ❖ Выбор источника информации о хостах в Ansible Tower

С помощью специальной формы можно добавлять переменные групп и хостов и переопределять значения по умолчанию.

Также есть возможность временно отключать хосты, щелкая на кнопках, как показано на рис. 19.5, и тем самым исключать их из обработки.

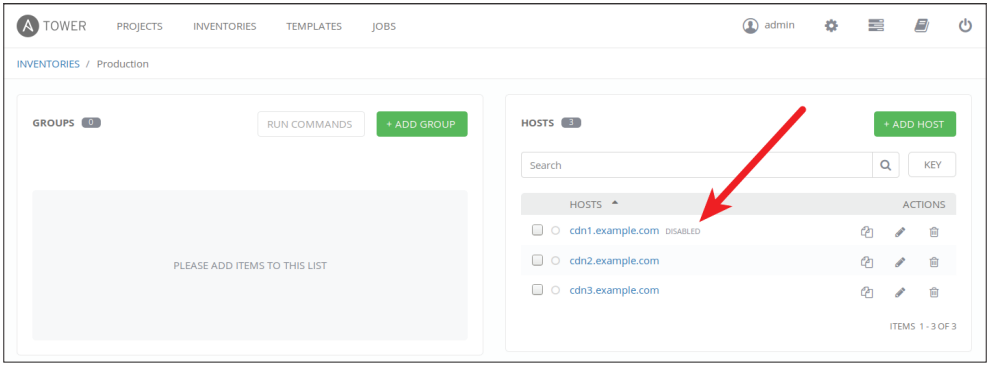

Рис. 19.5. Исключение хостов из обработки в Ansible Tower

Запуск заданий из шаблонов

Шаблоны заданий, как показано на рис. 19.6, связывают проекты с реестрами. Они определяют, как пользователи смогут запускать сценарии из проекта на определенных хостах из выбранного реестра.

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 задания создается новая запись, как показано на рис. 19.7.

В детальном обзоре каждой записи, как показано на рис. 19.8, 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.

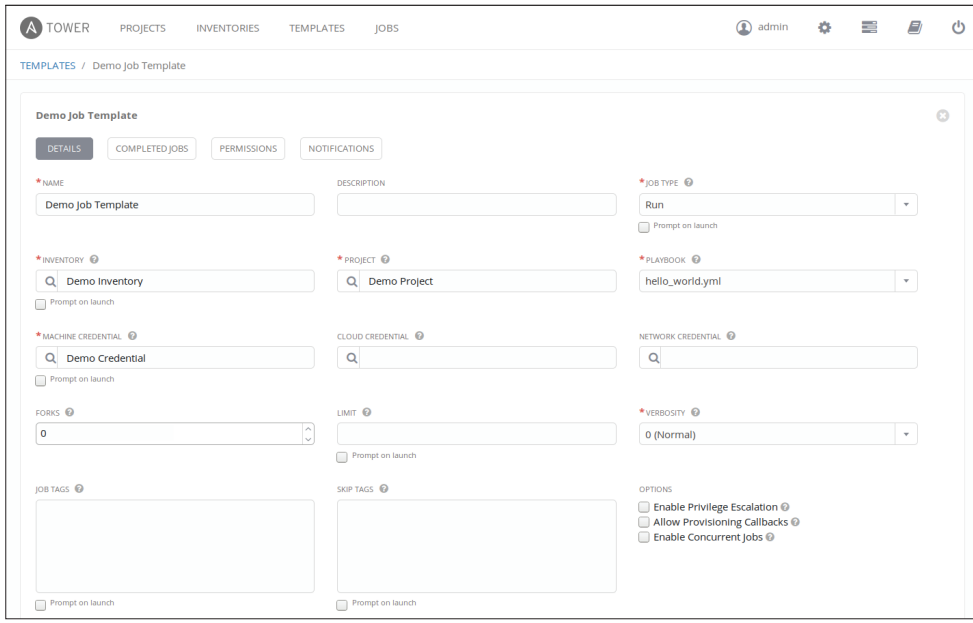

Рис. 19.6 ❖ Шаблоны заданий в Ansible Tower

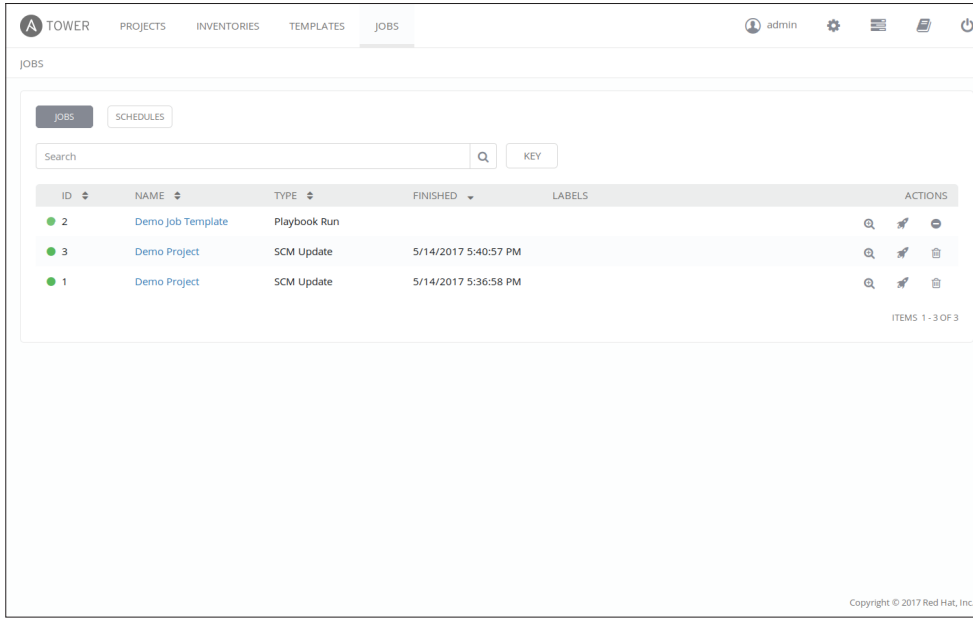

Рис. 19.7 ❖ Записи заданий в Ansible Tower

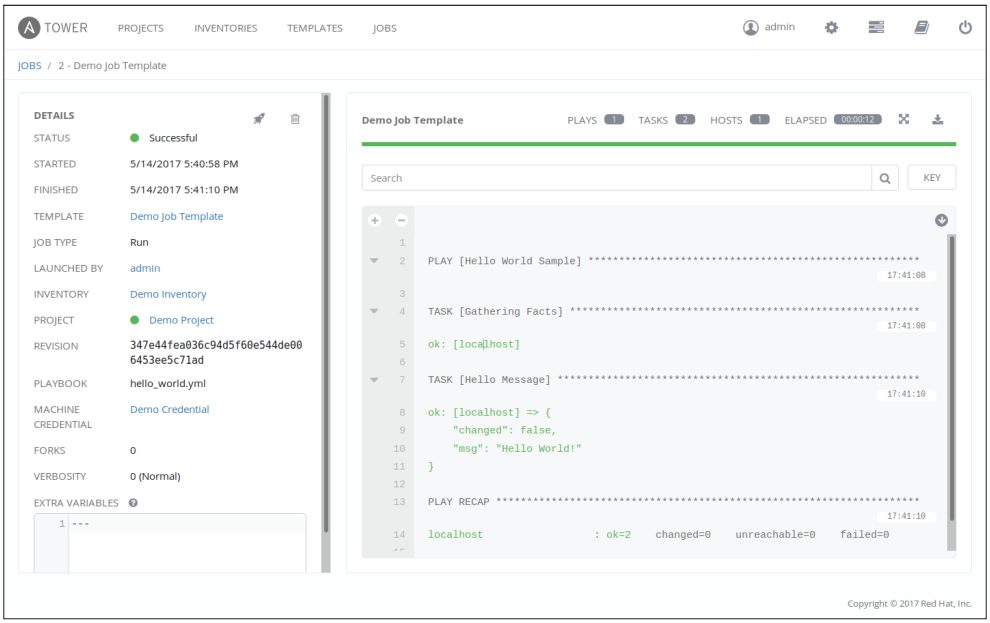

Рис. 19.8 ❖ Подробный обзор результатов задания в Ansible Tower

RESTful API

Сервер Ansible Tower 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*:

```
$ firefox https://10.42.0.42/api
```

На момент написания этих строк последней версией API была версия v1. Щелкнув на соответствующей ссылке или просто дополнив URL до *http://<tower_server>/api/v1*, можно получить список всех доступных ресурсов, как показано на рис. 19.9.

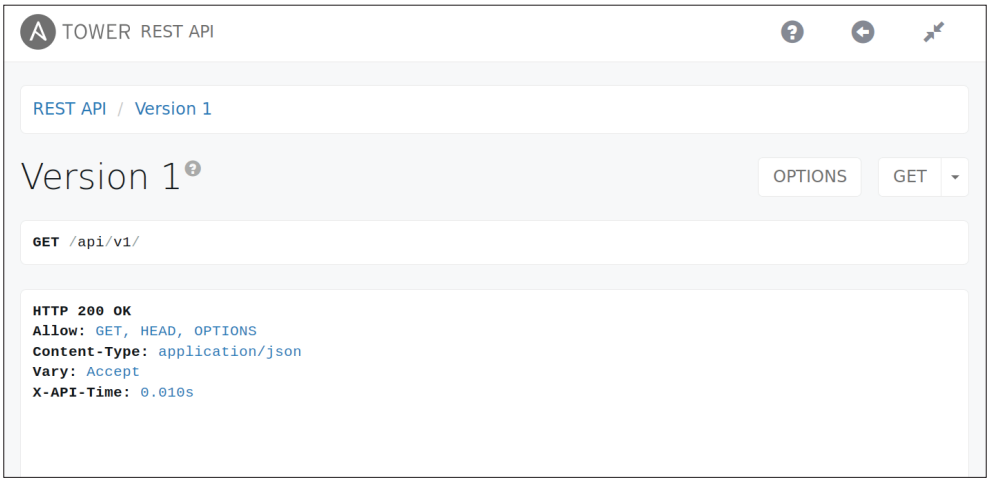

Рис. 19.9 ❖ Ansible Tower API версии 1

Самую свежую документацию с описанием API можно найти по адресу: http://docs.ansible.com/ansible-tower/.

Интерфейс командной строки Ansible Tower

Как создать нового пользователя или запустить задание, используя только программный интерфейс Ansible Tower? Конечно, можно было бы ограничиться лишь инструментом cURL для выполнения этих и других операций из командной строки по протоколу HTTP, но в Ansible имеется намного более удобный инструмент: `tower-cli`.

В отличие от приложения Ansible Tower, клиент командной строки `tower-cli` является открытым программным обеспечением и доступен в репозитории GitHub на условиях лицензии Apache 2.0.

Установка

Чтобы установить `tower-cli`, воспользуемся диспетчером пакетов для Python, `pip`.

Клиента `tower-cli` можно установить на уровне системы, если имеются привилегии root, или, как в данном случае, для локального пользователя Linux:

```
$ pip install ansible-tower-cli
```

Если установка выполняется с привилегиями обычного пользователя, клиент будет установлен в каталог *~/.local/bin/*. Не забудьте добавить путь *~/.local/bin* в переменную окружения PATH.

```
$ echo 'export PATH=$PATH:$HOME/.local/bin' >> $HOME/.profile
$ source $HOME/.profile
```

Прежде чем взаимодействовать с API, нужно настроить параметры учетной записи:

```
$ tower-cli config host 10.42.0.42
$ tower-cli config username admin
$ tower-cli config password JSKYmEBJATFn
```

Поскольку Ansible Tower использует самоподписанный сертификат SSL/TLS, просто пропустим его проверку:

```
$ tower-cli config verify_ssl false
```

Вывод по умолчанию содержит ровно столько информации, сколько необходимо. Но если у вас появится желание получить более подробный вывод по умолчанию, выберите формат yaml как формат по умолчанию. Впрочем, точно так же можно добавлять ключ `--format [human|json|yaml]` в конец команды для переопределения настроек по умолчанию:

```
$ tower-cli config format yaml
```

Для проверки просто выполните эту команду:

```
$ tower-cli config
```

Создание пользователя

Попробуем создать нового пользователя с помощью команды `tower-cli user`, как показано в примере 19.2. Если просто ввести эту команду без дополнительных параметров, она выведет список доступных действий.

Пример 19.2 ❖ Доступные действия клиента командной строки Ansible Tower CLI

```
$ tower-cli user
Usage: tower-cli user [OPTIONS] COMMAND [ARGS]...

  Manage users within Ansible Tower.

Options:
  --help Show this message and exit.

Commands:
  create  Create a user.
  delete  Remove the given user.
  get     Return one and exactly one user.
  list    Return a list of users.
  modify  Modify an already existing user.
```

RESTful API поддерживает типичные действия для этого вида программного интерфейса, с некоторыми исключениями. Главное отличие – доступные параметры и флаги, которые можно использовать при обращении к ресурсу. Если выполнить команду `tower-cli user create --help`, она выведет все доступные параметры.

Для создания пользователя требуется указать несколько параметров:

```
$ tower-cli user create \
--username guy \
--password 's3cr3t$' \
--email 'guy@example.com' \
--first-name Guybrush \
--last-name Threepwood
```

Клиент `tower-cli` обладает некоторой внутренней логикой, и с настройками по умолчанию его можно запустить несколько раз подряд, не рискуя получить сообщение об ошибке. `tower-cli` запросит ресурс, опираясь на ключевые поля, и вернет информацию о только что созданном пользователе, как показано в примере 19.3.

Пример 19.3 ❖ Вывод `tower-cli` после создания или обновления учетной записи пользователя

```
changed: true
id: 2
type: user
url: /api/v1/users/2/
related:
  admin_of_organizations: /api/v1/users/2/admin_of_organizations/
  organizations: /api/v1/users/2/organizations/
  roles: /api/v1/users/2/roles/
  access_list: /api/v1/users/2/access_list/
  teams: /api/v1/users/2/teams/
  credentials: /api/v1/users/2/credentials/
  activity_stream: /api/v1/users/2/activity_stream/
  projects: /api/v1/users/2/projects/
created: '2017-02-05T11:15:37.275Z'
username: guy
first_name: Guybrush
last_name: Threepwood
email: guy@example.com
is_superuser: false
is_system_auditor: false
ldap_dn: ''
external_account: null
auth: []
```

Однако `tower-cli` не обновит учетную запись, если попытаться изменить какие-то ее поля, например адрес электронной почты. Чтобы внести изменения, нужно или добавить флаг `--force-on-exists`, или явно указать действие `modify` вместо `create`.

Запуск задания

Первое, что нам наверняка понадобится автоматизировать, – это запуск задания из шаблона после успешной сборки на сервере непрерывной интеграции.

Клиент `tower-cli` существенно упрощает эту задачу. Достаточно лишь знать идентификатор или имя шаблона задания, которое требуется запустить. Чтобы узнать имя шаблона, можно воспользоваться действием `list`:

```
$ tower-cli job_template list --format human
== ================ ========= ======= ==============
id       name        inventory project    playbook
== ================ ========= ======= ==============
 5 Demo Job Template         1       4 hello_world.yml
 7 Deploy App ..             1       5 app.yml
== ================ ========= ======= ==============
```

В данный момент у нас имеются только два шаблона, и мы без труда можем сделать выбор. В больших промышленных окружениях порой имеются огромные коллекции шаблонов, и сделать правильный выбор намного труднее. Чтобы упростить задачу, `tower_cli` поддерживает возможность фильтрации вывода (например, по проекту `--project <id>` или по реестру `--inventory`).

Более сложные правила фильтрации больших коллекций шаблонов заданий (например, «вывести все шаблоны, имеющие определенное слово в имени с учетом регистра») можно определять с помощью параметра `--query`.

Например, вот как выглядит URL, сгенерированный клиентом для параметра `--query` с двумя аргументами – `name__icontains` и `deploy`:

```
https://10.42.0.42/api/v1/job_templates/?name__icontains=deploy
```

Все доступные фильтры можно найти в документации с описанием API (http://docs.ansible.com/ansible-tower/latest/html/towerapi/filtering.html).

Вызов действия `list` с желаемыми фильтрами вернет следующий результат:

```
$ tower-cli job_template list --query name__icontains deploy --format human
== ============= ========= ======= ==============
id      name      inventory project    playbook
== ============= ========= ======= ==============
 7 Deploy App xy         1       4 hello_world.yml
== ============= ========= ======= ==============
```

Отыскав требуемый шаблон, его можно запустить, как показано в примере 19.4, указав действие `job launch`, аргумент `--job-template` и имя или идентификатор выбранного шаблона.

Пример 19.4 ❖ Запуск задания с помощью tower-cli

```
$ tower-cli job launch --job-template 'Deploy App xy' --format human
Resource changed.
== ============ ======================= ======= =======
id job_template         created          status  elapsed
```

```
== ============ ====================== ======= =======
11             7 2017-02-05T14:08:05.022Z pending
== ============ ====================== ======= =======
```

Для мониторинга выполняющегося задания команда `tower-cli job` поддерживает действие `monitor` с аргументом – идентификатором задания. Эта команда запустится и будет ждать завершения задания.

```
tower-cli job monitor 11 --format human
Resource changed.
== ============ ====================== ========== =======
id job_template         created            status   elapsed
== ============ ====================== ========== =======
11             5 2017-02-05T13:57:30.504Z successful   6.486
== ============ ====================== ========== =======
```

Используя немного волшебства командной строки и утилиту `jq`, можно объединить запуск и мониторинг задания в одну команду:

```
tower-cli job monitor $(tower-cli job launch --job-template 5 --format json | jq '.id')
```

Послесловие

С окончанием этой главы подходит к концу и наше совместное путешествие. Но ваше путешествие с Ansible только начинается. Я надеюсь, что вам, так же как и мне, понравится работать с ним, и в следующий раз, столкнувшись с коллегами, нуждающимися в инструменте автоматизации, вы расскажете им о том, как Ansible может облегчить жизнь.

Приложение A

SSH

В качестве транспортного механизма Ansible использует протокол SSH, поэтому важно знать и понимать некоторые особенности SSH, чтобы успешно использовать этот протокол в работе с Ansible.

«Родной» SSH

По умолчанию Ansible использует SSH-клиент, установленный в операционной системе. Это значит, что Ansible может пользоваться всеми основными функциями SSH, включая Kerberos и SSH-шлюзы (jump hosts). Если у вас имеется свой файл *~/.ssh/config* с настройками SSH, Ansible будет использовать их.

SSH-агент

Существует программа с именем `ssh-agent`, которая позволяет упростить работу с приватными ключами SSH.

Когда на машине запущен `ssh-agent`, приватные ключи можно добавлять командой `ssh-add`.

```
$ ssh-add /path/to/keyfile.pem
```

Должна быть определена переменная окружения `SSH_AUTH_SOCK`, иначе команда `ssh-add` не сможет взаимодействовать с `ssh-agent`. Подробности смотрите в разделе «Запуск ssh-agent» ниже.

Получить список добавленных ключей можно командой `ssh_add` с ключом `-L` или `-l`, как показано в примере A.1. В данном случае были добавлены два ключа.

Пример A.1 ❖ Вывод ключей в агенте

```
$ ssh-add -l
2048 SHA256:o7H/I9rRZupXHJ7JnDi10RhSzeAKYiRVrlH9L/JFtfA /Users/lorin/.ssh/id_rsa
2048 SHA256:xLTmHqvHHDIdcrHiHdtoOXxq5sm9DOEVi+/jnObkKKM insecure_private_key
```

```
$ ssh-add -L
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDWAfog5tz4W9bPVbPDlNC8HWMfhjTgKOhpSZYI+clc
 e3/pz5viqsHDQIjzSImoVzIOTV0tOIfE8qMkqEYk7igESccCy0zN9VnD6EfYVkEx1C+xqkCtZTEVuQn
 d+4qyo222EAVkHm6bAhgyoA9nt9Um9WFO0045yHZL2Do9Z7KXTS4xOqeGF5vv7SiuKcsLjORPcWcYqC
 fYdrdUdRD9dFq7zFKmpCPJqNwDQDrXbgaTOe+H6cu2f4RrJLp88WY8voB3zJ7avv68eOgah82dovSgw
 hcsZp4SycZSTy+WqZQhzLogaifvtdgdzaooxNtsm+qRvQJyHkwdoXR6nJgt /Users/lorin/.ssh/i
 d_rsa
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA6NF8iallvQVp22WDkTkyrtvp9eWW6A8YVr+kz4TjGYe7
 gHzIw+niNltGEFHzD8+v1I2YJ6oXevct1YeS0o9HZyN1Q9qgCgzUFtdOKLv6IedplqoPkcmF0aYet2P
 kEDo3MlTBckFXPITAMzF8dJSIFo9D8HfdOV0IAdx4O7PtixWKn5y2hMNG0zQPyUecp4pzC6kivAIhyf
 HilFR61RGL+GPXQ2MWZWFYbAGjyiYJnAmCP3NOTd0jMZEnDkbUvxhMmBYSdETk1rRgm+R4LOzFUGaHq
 HDFIPKcF96hrucXzcWyLbIbEgE98OHlnVYCzRdK8jlqm8tehUc9c9WhQ== insecure_private_key
```

При попытке подключиться к удаленному хосту, когда запущен `ssh-agent`, клиент SSH попытается использовать для аутентификации ключи, хранящиеся в `ssh-agent`.

Использование SSH-агента дает несколько преимуществ:

- SSH-агент упрощает работу с зашифрованными приватными ключами SSH. При использовании зашифрованного приватного SSH-ключа файл, содержащий этот ключ, защищен паролем. При использовании ключа для установки SSH-соединения с хостом вам будет предложено ввести пароль. Даже если кто-либо получит доступ к вашему приватному SSH-ключу, его будет невозможно использовать без пароля. При использовании зашифрованного приватного SSH-ключа без участия SSH-агента каждое использование приватного ключа будет требовать ввода пароля шифрования. При использовании SSH-агента вводить пароль приватного ключа потребуется только во время добавления ключа в агента;
- если Ansible используется для управления хостами с разными SSH-ключами, применение SSH-агента упрощает конфигурирование Ansible – вам не придется явно определять `ansible_ssh_private_key_file` на хостах, как мы это делали в примере 1.1;
- если понадобится установить соединение SSH между удаленным и другим хостами (например, для клонирования приватного репозитория Git посредством SSH), можно воспользоваться преимуществом *перенаправления агента* (agent forwarding) и избавиться от необходимости копировать приватный SSH-ключ на удаленный хост. Далее я поясню суть перенаправления агента.

Запуск ssh-agent

Способ запуска SSH-агента зависит от операционной системы.

macOS

macOS уже настроена на автоматический запуск `ssh-agent`, поэтому вам не нужно предпринимать каких-либо действий.

Linux

В Linux необходимо запустить `ssh-agent` и проверить правильность определений переменных окружения. Если запустить команду `ssh-agent` непосредственно, она выведет список переменных окружения, которые необходимо определить. Например:

```
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-YI7PBGlkOteo/agent.2547; export SSH_AUTH_SOCK;
SSH_AGENT_PID=2548; export SSH_AGENT_PID;
echo Agent pid 2548;
```

Вы можете автоматически экспортировать эти переменные, вызвав `ssh-agent`, как показано ниже:

```
$ eval $(ssh-agent)
```

Вы также должны гарантировать, что в каждый конкретный момент времени выполняется только один экземпляр `ssh-agent`. В Linux имеется множество разных инструментов, таких как *Keychain* и *Gnome Keyring*, обеспечивающих автоматический запуск `ssh-agent`. Для этого также можно отредактировать файл *.profile*. Тема настройки поддержки `ssh-agent` в учетной записи выходит далеко за рамки этой книги, поэтому за дополнительной информацией я рекомендую обратиться к документации с описанием вашего дистрибутива Linux.

Agent Forwarding

При клонировании репозитория Git через SSH необходимо использовать приватный SSH-ключ, распознаваемый сервером Git. Я стараюсь избегать копирования приватных SSH-ключей на хосты, чтобы минимизировать потенциальный ущерб, если вдруг хост будет взломан.

Для этого на локальной машине можно использовать программу `ssh-agent` с функцией перенаправления агента (agent forwarding). Если вы установили SSH-соединение между вашим ноутбуком и хостом A при включенном перенаправлении агента, то сможете установить SSH-соединение между хостами A и B, используя приватный ключ, хранящийся на ноутбуке.

На рис. A.1 показан пример работы функции перенаправления агента. Допустим, вам нужно получить исходный код из репозитория GitHub через SSH. На вашем ноутбуке запущена утилита `ssh-agent`, и вы добавили приватный ключ командой `ssh-add`.

Если SSH-соединение с севером приложений установлено вручную, вы сможете вызвать команду `ssh` с ключом `-A`, который активирует перенаправление агента:

```
$ ssh -A myuser@myappserver.example.com
```

На сервере выполняется команда клонирования репозитория Git:

```
$ git clone git@github.com:lorin/mezzanine-example.git
```

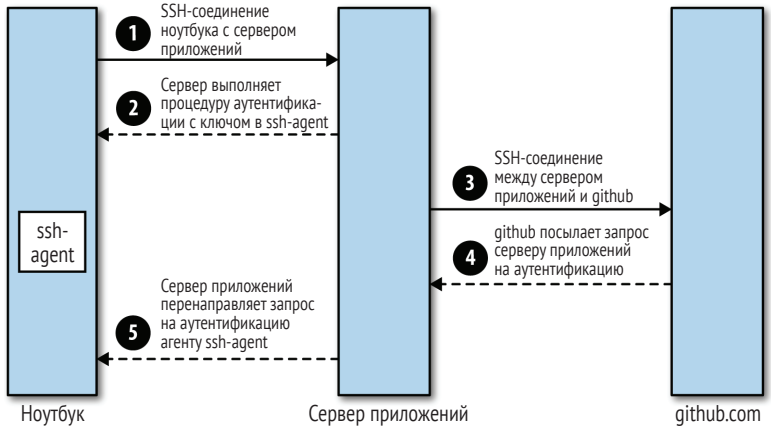

Рис. А.1 ❖ Перенаправление агента в действии

Git установит SSH-соединение с GitHub. SSH-сервер на GitHub попытается выполнить аутентификацию SSH-клиента на сервере приложений. Приватный ключ отсутствует на сервере приложений, однако, так как было включено перенаправление агента, SSH-клиент на сервере приложений подключится к агенту `ssh-agent`, запущенному на ноутбуке, и произведет аутентификацию.

При использовании функции перенаправления агента с Ansible нужно помнить о некоторых проблемах.

Во-первых, вы должны сообщить Ansible о необходимости включить перенаправление агента при установке соединения с удаленными машинами, поскольку SSH не включает его по умолчанию. Для включения перенаправления агента на всех узлах, с которыми устанавливается SSH-соединение, можно добавить следующие строки в файл *~/.ssh/config* на управляющей машине:

```
Host *
    ForwardAgent yes
```

Также можно включить перенаправление для конкретного сервера:

```
Host appserver.example.com
    ForwardAgent yes
```

Если нужно включить перенаправление агента только для Ansible, добавьте в файл *ansible.cfg* параметр `ssh_args` в раздел `ssh_connection`:

```
[ssh_connection]
ssh_args = -o ControlMaster=auto -o ControlPersist=60s -o ForwardAgent=yes
```

Здесь я использовал более длинный флаг `-o ForwardAgent=yes` вместо короткого `-A`, но оба они действуют совершенно идентично.

Параметры `ControlMaster` и `ControlPersist` необходимы для оптимизации работы – *мультиплексирования SSH*. По умолчанию они включены, но когда вы-

полняется переопределение переменной ssh_args, их необходимо указать явно, иначе можно лишиться этой функции. Мультиплексирование SSH обсуждается в главе 11.

Команда sudo и перенаправление агента

При включении перенаправления агента удаленная машина устанавливает переменную окружения SSH_AUTH_SOCK, куда записывает путь к сокету домена Unix (например, */tmp/ssh-FShDVu5924/agent.5924*). Однако, когда выполняется команда sudo, переменная SSH_AUTH_SOCK не переносится в новое окружение, если только вы не настроили такого поведения sudo явно.

Чтобы обеспечить перенос переменной SSH_AUTH_SOCK в окружение пользователя root, можно добавить следующую строку в файл */etc/sudoers* или в свой файл */etc/sudoers.d* (для дистрибутивов на основе Debian, таких как Ubuntu).

```
Defaults>root env_keep+=SSH_AUTH_SOCK
```

Сохраните этот файл с именем *99-keep-ssh-auth-sock-env* в каталоге *files* на локальной машине.

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: visudo -cf %s
```

При наличии выражения validate Ansible копирует файл сначала во временный каталог, а потом запускает указанную программу проверки. Если программа завершится успешно (0), Ansible скопирует файл из временного каталога в постоянное местоположение. Если программа вернет результат, отличный от нуля, Ansible выведет сообщение об ошибке:

```
failed: [myhost] => {"checksum": "ac32f572f0a670c3579ac2864cc3069ee8a19588",
"failed": true}
msg: failed to validate: rc:1 error:

FATAL: all hosts have already failed -- aborting
```

Поскольку файл *sudoers* с ошибками может нарушить доступ к привилегиям пользователя root, его всегда полезно проверить с помощью программы *visudo*. Для понимания проблем, которые несут файлы *sudoers*, предлагаю вашему вниманию статью участника проекта Ansible Жан-Пита Мэна (Jan-Piet Men) «Don't try this at the office: /etc/sudoers» (*http://bit.ly/1DfeQY7*).

```
- name: copy the sudoers file so we can do agent forwarding
  copy:
    src: files/99-keep-ssh-auth-sock-env
```

```
    dest: /etc/sudoers.d/99-keep-ssh-auth-sock-env
    owner: root group=root mode=0440
    validate: visudo -cf %s
```

К сожалению, на данный момент невозможно вызвать sudo с привилегиями обычного пользователя и использовать функцию перенаправления агента от имени другого непривилегированного пользователя. Например, представьте, что вам нужно вызвать sudo от имени пользователя ubuntu, чтобы выполнять команды от имени пользователя deploy. Проблема в том, что сокет домена Unix, ссылка на который хранится в SSH_AUTH_SOCK, принадлежит пользователю ubuntu и недоступен для чтения или изменения пользователю deploy.

Как вариант можно вызвать модуль git с привилегиями root и изменить разрешения с помощью модуля file, как показано в примере A.2.

Пример A.2 ❖ Копирование пользователем root и изменение разрешений

```
- name: verify the config is valid sudoers file
  local_action: command visudo -cf files/99-keep-ssh-auth-sock-env
  sudo: True

- name: copy the sudoers file so we can do agent forwarding
  copy:
    src: files/99-keep-ssh-auth-sock-env
    dest: /etc/sudoers.d/99-keep-ssh-auth-sock-env
    owner: root
    group: root
    mode: "0440"
    validate: 'visudo -cf %s'
  sudo: True

- name: check out my private git repository
  git:
    repo: git@github.com:lorin/mezzanine-example.git
    dest: "{{ proj_path }}"
  sudo: True

- name: set file ownership
  file:
    path: "{{ proj_path }}"
    state: directory
    recurse: yes
    owner: "{{ user }}"
    group: "{{ user }}"
  sudo: True
```

Ключи хоста

Каждый хост, где выполняется сервер SSH, имеет свой ключ хоста. Ключ хоста играет роль подписи, уникально идентифицирующей хост. Ключи хоста помогают предотвратить атаки типа «человек в середине». При клонировании ре-

позитория с GitHub через SSH никогда нельзя знать наверняка, действительно ли сервер, объявляющий себя *github.com*, является сервером GitHub или же это мошеннический сервер, подделывающий доменное имя. Ключи хоста позволяют убедиться, что сервер, объявляющий себя как *github.com*, действительно им является. Это значит, что вы должны иметь ключ хоста (копию подписи) до попытки установить соединение с этим хостом.

По умолчанию Ansible проверяет ключ хоста, но эту проверку можно отключить в файле *ansible.cfg*:

```
[defaults]
host_key_checking = False
```

Проверка ключа хоста используется вместе с модулем `git`. Вспомните, как в главе 6 модуль `git` использовал параметр `accept_hostkey`:

```
- name: check out the repository on the host
  git: repo={{ repo_url }} dest={{ proj_path }} accept_hostkey=yes
```

Модуль `git` может зависнуть при клонировании репозитория Git через SSH, если проверка ключа хоста отключена, а SSH-ключ хоста сервера Git текущему хосту неизвестен.

Самое простое решение – использовать параметр `accept_hostkey`, чтобы сообщить Git о необходимости автоматически принять ключ хоста, если он неизвестен. Этот подход мы использовали в примере 6.6.

Многие просто принимают ключ хоста и не заботятся о вероятности таких атак. Именно так мы и поступили в сценарии, определив аргумент `accept_hostkey=yes` модуля `git`. Однако если вы относитесь к безопасности с большей ответственностью и не хотите автоматически принимать ключ хоста, можете вручную извлечь и провести проверку ключа, а затем добавить его в системный файл */etc/ssh/known_hosts* для всей системы или в пользовательский файл *~/.ssh/known_hosts* для конкретного пользователя.

Чтобы вручную проверить SSH-ключ хоста, необходимо получить отпечаток SSH-ключа хоста альтернативным способом. При использовании GitHub в качестве сервера Git отпечаток SSH-ключа можно найти на сайте GitHub по ссылке http://bit.ly/1DffcxK.

На момент написания книги отпечаток SHA256 RSA сервера GitHub в формате base64 (новейший формат)[1] имел вид `16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48`, но лучше не верить мне на слово и проверить на сайте.

Далее необходимо извлечь полный SSH-ключ хоста. Для этого можно использовать программу `ssh-keyscan`, которая извлечет ключ хоста с именем *github.com*. Я предпочитаю помещать файлы, с которыми работает Ansible, в каталог *files*. Давайте сделаем это:

---

[1] Формат по умолчанию изменился в версии OpenSSH 6.8 с прежнего MD5 на base64 SHA256.

```
$ mkdir files
$ ssh-keyscan github.com > files/known_hosts
```

Результат будет выглядеть так:

```
github.com ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+PXYPCPy6rbTrTtw7PHkccK
rpp0yVhp5HdEIcKr6pLlVDBfOLX9QUsyCOV0wzfjIJNlGEYsdlLJizHhbn2mUjvSAHQqZETYP81eFzLQ
NnPHt4EVVUh7VfDESU84KezmD5QlWpXLmvU31/yMf+Se8xhHTvKSCZIFImWwoG6mbUoWf9nzpIoaSjB+
weqqUUmpaaasXVal72J+UX2B+2RPW3RcT0eOzQgqlJL3RKrTJvdsjE3JEAvGq3lGHSZXy28G3skua2Sm
Vi/w4yCE6gbODqnTWlg7+wC604ydGXA8VJiS5ap43JXiUFFAaQ==
```

Повысить уровень безопасности поможет ключ `-H`, поддерживаемый командой `ssh-keyscan`. Благодаря ему имя хоста не появится в файле *known_hosts*. Даже если кто-то попытается получить доступ к вашему файлу со списком хостов, он не сможет определить имена хостов. При использовании этого ключа результат будет выглядеть так:

```
|1|BI+Z8H3hzbcmTWna9R4orrwrNrg=|wCxJf50pTQ83JFzyXG4aNLxEmzc= ssh-rsa AAAAB3NzaC1y
c2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+PXYPCPy6rbTrTtw7PHkccKrpp0yVhp5HdEI
cKr6pLlVDBfOLX9QUsyCOV0wzfjIJNlGEYsdlLJizHhbn2mUjvSAHQqZETYP81eFzLQNnPHt4EVVUh7Vf
DESU84KezmD5QlWpXLmvU31/yMf+Se8xhHTvKSCZIFImWwoG6mbUoWf9nzpIoaSjB+weqqUUmpaaasXVa
l72J+UX2B+2RPW3RcT0eOzQgqlJL3RKrTJvdsjE3JEAvGq3lGHSZXy28G3skua2SmVi/w4yCE6gbODqnT
Wlg7+wC604ydGXA8VJiS5ap43JXiUFFAaQ==
```

Далее необходимо проверить, совпадает ли ключ хоста в файле *files/known_hosts* с отпечатком, полученным с сайта GitHub. Это можно сделать с помощью программы `ssh-keygen`:

```
$ ssh-keygen -lf files/known_hosts
```

Ее вывод должен совпадать с отпечатком RSA, представленным на сайте:

```
2048 SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8 github.com (RSA)
```

Теперь, когда вы уверены в достоверности ключа сервера Git, можно с помощью модуля `copy` скопировать его в каталог */etc/ssh/known_hosts*.

```
- name: copy system-wide known hosts
  copy: src=files/known_hosts dest=/etc/ssh/known_hosts owner=root group=root
  mode=0644
```

Или в каталог конкретного пользователя *~/.ssh/known_hosts*. В примере А.3 показано, как скопировать файл со списком хостов с управляющей машины на удаленные хосты.

Пример А.3 ❖ Добавление известного хоста

```
- name: ensure the ~/.ssh directory exists
  file: path=~/.ssh state=directory
- name: copy known hosts file
  copy: src=files/known_hosts dest=~/.ssh/known_hosts mode=0600
```

Неправильный ключ хоста может вызвать проблемы даже при отключенной проверке ключа

Если вы отключили проверку ключа хоста в Ansible, определив в файле *ansible.cfg* параметр `host_key_checking` со значением `false`, а ключ для хоста, с которым Ansible пытается установить связь, не соответствует ключу в файле *~/.ssh/known_hosts*, функция перенаправления агента работать не будет. Попытка клонировать репозиторий Git в этом случае завершится ошибкой:

```
TASK: [check out the repository on the host] *********************************
failed: [web] => {"cmd": "/usr/bin/git ls-remote git@github.com:lorin/
mezzanine- example.git -h refs/heads/HEAD", "failed": true, "rc": 128}
stderr: Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

msg: Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

FATAL: all hosts have already failed -- aborting
```

Это может произойти, если вы использовали и удалили машину Vagrant, а затем создали новую, потому что в этом случае ключ хоста изменится. Проверить работу перенаправления агента можно следующим образом:

```
$ ansible web -a "ssh-add -l"
```

Если функция работает, результат будет выглядеть так:

```
web | success | rc=0 >>
2048 SHA256:ScSt41+elNd0YkvRXW2nGapX6AZ8MP1J1UNg/qalBUs /Users/lorin/.ssh
/id_rsa (RSA)
```

Если не работает, результат будет такой:

```
web | FAILED | rc=2 >>
Could not open a connection to your authentication agent.
```

Если это случится, удалите соответствующий элемент из файла *~/.ssh/known_hosts*. Обратите внимание, что при использовании мультиплексирования SSH Ansible поддерживает соединение с хостом открытым в течение 60 секунд. Поэтому вы должны дождаться истечения срока соединения, иначе нельзя будет увидеть эффект внесения изменений в файл *known_hosts*.

Очевидно, что проверка достоверности SSH-ключа хоста требует гораздо больше усилий, чем простой его прием не глядя. Как всегда, это компромисс между безопасностью и удобством.

Приложение B

Использование ролей IAM для учетных данных EC2

Если вы собираетесь запускать Ansible внутри VPC, можете воспользоваться поддержкой ролей службы *идентификации и управления доступом* (Identity and Access Management, IAM) в Amazon, чтобы избавиться от необходимости определять переменные окружения для передачи учетных данных EC2 в экземпляр. IAM-роли в Amazon позволяют определять пользователей и группы и управлять их разрешениями в EC2 (например, получать информацию о запущенных экземплярах, создавать экземпляры, создавать образы). IAM-роли также можно присваивать запущенным экземплярам, чтобы, например, заявить: «Этому экземпляру позволено запускать другие экземпляры».

Когда вы посылаете запрос в EC2 с помощью клиентской программы, поддерживающей IAM-роли, и экземпляру предоставлены соответствующие ролям разрешения, клиент извлекает учетные данные из службы метаданных экземпляра EC2 (http://amzn.to/1Cu0fTl) и использует их для отправки запроса конечной точке EC2.

IAM-роли можно создавать в консоли управления Amazon Web Services (AWS) или из командной строки, с помощью клиента командной строки AWS (AWS CLI, http://aws.amazon.com/cli/).

Консоль управления AWS

Посмотрим, как с помощью консоли управления AWS создавать IAM-роли, предоставляющие привилегированный доступ «Power User Access», позволяющий делать практически все, что угодно, кроме изменения пользователей и групп IAM.

1. Зайдите в консоль управления AWS (https://console.aws.amazon.com).
2. Щелкните на ссылке **Identity & Access Management**.
3. Щелкните на кнопке **Roles** (Роли) слева.
4. Щелкните на кнопке **Create New Role** (Создать новую роль).

5. Дайте роли имя и затем щелкните на кнопке **Next** (Далее). Я предпочитаю использовать имя роли `ansible` для экземпляра, на котором будет запущена Ansible.
6. В меню **AWS Service Roles** (Роли AWS Service) выберите пункт **Amazon EC2**.
7. Выберите привилегии **PowerUserAccess**. Щелкните на кнопке **Next Step** (Следующий шаг).
8. Щелкните на кнопке **Create Role** (Создать роль).

Если после создания роли выбрать ее и щелкнуть на кнопке **Show Policy** (Показать разрешения), на экране должен появиться документ JSON, как показано в примере B.1.

Пример B.1 ❖ Документ с разрешениями IAM-роли «Power User»

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "NotAction": ["iam:*", "organizations:*"],
      "Resource": "*"
    },{
      "Effect": "Allow",
      "Action": "organizations:DescribeOrganization",
      "Resource": "*"
    }
  ]
}
```

При создании роли в веб-интерфейсе AWS также автоматически создаст *профиль экземпляра* с именем роли (например, `ansible`), а еще свяжет роль с именем профиля экземпляра. При создании экземпляра с помощью модуля `ec2` и передаче имени профиля экземпляра в параметре `instance_profile_name` созданный экземпляр будет обладать разрешениями роли.

Командная строка

Роль и профиль экземпляра можно также создать с помощью клиента командной строки AWS CLI. Но для этого потребуется чуть больше усилий:

1. Создайте роль, определив политику безопасности. Политика безопасности описывает объекты, которые могут принять на себя роль и условия доступа роли.
2. Создайте политику, описывающую разрешения для роли. В данном случае нам нужно создать эквивалент привилегированного пользователя, чтобы обладатель роли смог производить любые действия с AWS, кроме операций с ролями и группами IAM.
3. Создайте профиль экземпляра.
4. Свяжите роль с профилем экземпляра.

Сначала нужно создать два файла с политиками IAM. Они должны иметь формат JSON. Политика безопасности представлена в примере B.2. Это та же политика, которая автоматически генерируется AWS при создании роли через веб-интерфейс.

Политика роли определяет ее возможности и показана в примере B.3.

Пример B.2 ❖ trust-policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Пример B.3 ❖ power-user.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "NotAction": "iam:*",
      "Resource": "*"
    }
  ]
}
```

В примере B.4 показано, как создать профиль экземпляра из командной строки, когда файлы, приведенные в примерах B.2 и B.3, уже созданы.

Пример B.4 ❖ Создание профиля экземпляра из командной строки

```
# Файлы trust-policy.json и power-user.json должны находиться
# в текущем каталоге, иначе измените аргументы file://,
# указав в них полный путь

$ aws iam create-role --role-name ansible --assume-role-policy-document \
  file://trust-policy.json
$ aws iam put-role-policy --role-name ansible --policy-name \
  PowerUserAccess-ansible-20170214 --policy-document file://power-user.json
$ aws iam create-instance-profile --instance-profile-name ansible
$ aws iam add-role-to-instance-profile --instance-profile-name ansible \
  --role-name ansible
```

Как видите, работать с веб-интерфейсом гораздо проще. Но для автоматизации проще использовать командную строку. За дополнительной информацией

по IAM обращайтесь к руководству пользователя AWS Identity and Access Management (http://docs.aws.amazon.com/IAM/latest/UserGuide/).

После создания профиля можно запустить экземпляр EC2 с этим профилем, например с помощью модуля ec2, использовав параметр instance_profile_name:

```
- name: launch an instance with iam role
  ec2:
    instance_profile_name: ansible
    # Другие параметры не показаны
```

При установке SSH-соединения с экземпляром можно запросить у службы метаданных EC2 подтверждение, что он связан с профилем Ansible. Результат должен выглядеть приблизительно так:

```
$ curl http://169.254.169.254/latest/meta-data/iam/info
{
  "Code" : "Success",
  "LastUpdated" : "2014-11-17T02:44:03Z",
  "InstanceProfileArn" : "arn:aws:iam::549704298184:instance-profile/ansible",
  "InstanceProfileId" : "AIPAINM7F44YGDNIBHPYC"
}
```

Можно, конечно, детально изучить учетные данные, но, вообще говоря, в этом нет никакой необходимости. Библиотека Boto автоматически извлечет их при выполнении модуля ec2 или сценария динамической инвентаризации:

```
$ curl http://169.254.169.254/latest/meta-data/iam/security-credentials/ansible
{
  "Code" : "Success",
  "LastUpdated" : "2015-02-09T21:45:20Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" : "ASIAIYXCUETJPY42AC2Q",
  "SecretAccessKey" : "ORp9gldiymIKH9+rFtWEx8BjGRteNTQSRnLnlmWq",
  "Token" : "AQoDYXdzEGca4AMPC5W69pvtENpXjw79oH9...",
  "Expiration" : "2015-02-10T04:10:36Z"
}
```

Эти учетные данные являются временными. Amazon постоянно изменяет их.

Теперь вы можете использовать этот экземпляр в качестве управляющей машины без определения учетных данных в переменных окружения. Модули ec2 автоматически извлекут их из службы метаданных.

Глоссарий

Группа – набор хостов, обладающий названием.

Операция – сопоставляет группу хостов со списком задач, назначенных для выполнения на данных хостах.

Декларативный – тип языка программирования, когда программист описывает желаемый результат, а не процесс его достижения. Сценарии Ansible являются декларативными. Язык SQL является еще одним примером декларативного языка. Напротив, такие языки, как Java и Python, являются *процедурными*.

Динамический реестр (динамическая инвентаризация) – источник, снабжающий Ansible информацией о хостах и группах во время исполнения сценария.

Задача – единица работы в операциях Ansible. Задача определяет модуль и его аргументы, а также дополнительные имя и параметры.

Зарегистрированная переменная – переменная, созданная с помощью выражения `register` в задаче.

Идемпотентный – действие является идемпотентным, если многократное его выполнение дает тот же результат, что и однократное.

Конвергентность – свойство системы управления конфигурациями, когда система запускается на сервере несколько раз для приведения сервера в желаемое состояние, с каждым разом приближая к нему. Конвергенция наиболее тесно ассоциируется с системой управления конфигурациями CFEngine. Ansible не обладает свойством конвергентности, поскольку желаемое состояние достигается после первого запуска.

Контейнер – форма виртуализации, которая осуществляется на уровне операционной системы, когда экземпляр виртуальной машины использует то же ядро, что и система-носитель. Docker является наиболее известной технологией контейнеров.

Модуль – сценарий Ansible, выполняющий определенную задачу. Примером модуля может послужить создание учетной записи пользователя, установка пакета или запуск службы. Большинство модулей Ansible является идемпотентными.

Мультиплексирование SSH – особенность SSH-клиента OpenSSH, позволяющая сократить время на установке SSH-соединений, когда требуется установить несколько SSH-соединений с одной машиной. Ansible использует мультиплексирование SSH для повышения производительности.

Обработчик – напоминает задачу, но выполняется только в ответ на уведомление, посылаемое задачей.

Оркестрация (согласование) – выполнение серии задач в строго определенном порядке на группе серверов. Оркестрация часто необходима для развертывания.

Подстановки – код, выполняемый на управляющей машине для получения конфигурационных данных, необходимых Ansible во время работы со сценарием.

Псевдоним (Alias) – имя хоста в реестре, отличающееся от действительного.

Развертывание – процесс установки программного обеспечения в работающую систему.

Реестр (инвентаризация) – список хостов и групп.

Режим проверки (Check mode) – особый режим запуска сценария. В этом режиме сценарии Ansible не производят изменений на удаленных хостах. Вместо этого формируется отчет о возможных изменениях состояния хоста при исполнении каждой задачи. Иногда упоминается как режим «dry run» (холостой ход).

Роль – механизм Ansible, служащий для объединения задач, обработчиков, файлов, шаблонов и переменных.

Например, роль `nginx` может содержать задачи по установке пакета Nginx, созданию конфигурационного файла для Nginx, копированию файлов сертификата TLS и запуску службы Nginx.

Составные аргументы – аргументы модулей, имеющие вид списка или словаря.

Сценарий (Playbook) – определяет список операций и группу хостов, на которых выполняются данные операции.

Транспорт (Transport) – протокол и механизм, используемые в Ansible для подключения к удаленному хосту. По умолчанию роль транспорта выполняет протокол SSH.

Управление конфигурациями (Configuration management) – процесс поддержания серверов в рабочем состоянии. Под *рабочим состоянием* подразумевается, что файлы конфигурации хранят допустимые настройки, имеются все необходимые файлы, запущены нужные службы, имеются в наличии ожидаемые учетные записи пользователей, установлены корректные разрешения и т. д.

Управляющая машина – компьютер, на котором установлена система Ansible, осуществляющая управление удаленными хостами.

Управляющий сокет – сокет домена Unix, который используется SSH-клиентом для соединения с удаленным хостом при включенном мультиплексировании SSH.

Факт – переменная с информацией об определенном хосте.

Хост – удаленный сервер, управляемый Ansible.

Шаблон – синтаксис Ansible для описания хостов, на которых выполняется операция.

AMI (Amazon Machine Image) – образ виртуальной машины в облаке Amazon Elastic Compute Cloud, также известном как *EC2*.

Ansible, Inc. – компания, осуществляющая контроль над проектом Ansible.

Ansible Galaxy – репозиторий (https://galaxy.ansible.com/) ролей Ansible, разработанных сообществом.

Ansible Tower – платная веб-система и интерфейс REST для управления Ansible, продается компанией Ansible, Inc.

CIDR (Classless Inter-Domain Routing, бесклассовая адресация) – правило определения диапазона IP-адресов, используемых в группах безопасности Amazon EC2.

ControlPersist – синоним мультиплексирования SSH.

DevOps – жаргонный профессиональный термин в IT, ставший популярным в середине 2010-х гг. (https://ru.wikipedia.org/wiki/DevOps).

Dry run – смотрите *Режим проверки* (*Check mode*).

DSL (Domain Specific Language) – предметно-ориентированный язык. В системах, использующих предметно-ориентированные языки, пользователь взаимодействует с системой, создавая и выполняя файлы на таких языках. Предметно-ориентированные языки не обладают такой же широтой возможностей, как универсальные языки программирования, но (если правильно сконструированы) они проще читаются и на них легче писать управляющие программы. Ansible поддерживает предметно-ориентированный язык, использующий синтаксис YAML.

EBS (Elastic Block Store) – блочное хранилище. В терминах Amazon EC2 под EBS подразумевается дисковое пространство, которое может быть закреплено за экземплярами.

Glob – шаблон, используемый оболочками Unix для выбора файлов по именам. Например, шаблон `*.txt` соответствует всем файлам с расширением `.txt`.

IAM (Identity and Access Management) – служба облака Elastic Compute Cloud компании Amazon, позволяющая управлять разрешениями пользователей и групп.

Ohai – инструмент, используемый Chef для извлечения информации о хосте. Если Ohai установлен, Ansible запускает его в процессе сбора фактов о хосте.

TLS – протокол защиты транспортного уровня (Transport Layer Security). Используется для защиты взаимодействий между веб-серверами и браузерами. TLS заменил более ранний протокол защищённых сокетов (Secure Sockets Layer, SSL). Многие неправильно упоминают TLS как SSL.

Vault – механизм, используемый в Ansible для шифрования конфиденциальных данных на диске. Обычно применяется для безопасного хранения секретных данных в системах управления версиями.

Vagrant – инструмент для управления виртуальными машинами. Используется разработчиками для создания повторяемых окружений разработки.

Virtualenv – механизм для установки пакетов Python в виртуальные окружения, которые можно включать и выключать. Позволяет пользователю устанавливать пакеты Python, не обладая правами пользователя root и не засоряя глобальную библиотеку пакетов Python на машине.

VPC (Virtual Private Cloud) – используется Amazon EC2 для описания изолированной сети, которую можно создать для экземпляров EC2.

Библиография

1. *Hashimoto M*. Vagrant: Up and Running. O'Reilly Media, 2013.
2. *Hunt A., Thomas D.* The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, 1999[1].
3. *Jaynes M*. Taste Test: Puppet, Chef, Salt, Ansible. Publisher, 2014[2].
4. *Kleppmann M*. Designing Data-Intensive Applications. O'Reilly Media, 2015.
5. *Kurniawan Y*. Ansible for AWS. Leanpub, 2016.
6. *Limoncelli T. A.*, *Hogan C. J.*, *Chalup S. R.* The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems. Addison-Wesley Professional, 2014.
7. *Mell P.*, *Grance T*. The NIST Definition of Cloud Computing. NIST Special Publication 800-145, 2011.
8. OpenSSH/Cookbook/Multiplexing, Wikibooks. URL: http://bit.ly/1bpeV0y. October 28, 2014.
9. *Shafer A. C.* Agile Infrastructure in Web Operations: Keeping the Data on Time. O'Reilly Media, 2010.

---

[1] *Хант Э., Томас Д., Алексашин А.* Программист-прагматик. Путь от подмастерья к мастеру. Лори, 2016. ISBN 0-201-61622-x. – *Прим. перев.*

[2] *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. СПб.: Питер, 2018. ISBN 978-5-4461-0512-0. – *Прим. перев.*

Предметный указатель

Об авторах

Лорин Хохштейн (Lorin Hochstein) родился и вырос в Монреале, провинция Квебек, хотя по его акценту вы никогда не догадались бы, что он канадец, разве что по его привычке неправильно строить некоторые фразы. Занимался научной деятельностью – два года работал доцентом на кафедре информационных технологий в университете Небраска-Линкольн и еще четыре – научным сотрудником в институте информационных технологий Южно-Калифорнийского университета. Получил степень бакалавра в области информационных технологий в университете Макгилла, степень магистра в области проектирования электрических систем в Бостонском университете, защитил докторскую диссертацию в университете Мериленда. В настоящее время работает старшим разработчиком в Netflix, где практикует методику Chaos Engineering.

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 является членом основной команды разработчиков Ansible и написал более 40 модулей для Ansible. Также является членом комитета Project Management Committee Apache CloudStack. В настоящее время работает системным инженером в SWISS TXT.

Колофон

На обложке «Установка и работа с Ansible» изображена корова голштино-фризской породы, которую в Северной Америке часто называют голштинской, а в Европе – фризской. Она была выведена в Европе, в Нидердандах, 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 килограммов; зрелые голштинцы могут достигать в весе 580 килограммов и в холке до 1,5 метра. Половая зрелость у этой породы наступает в возрасте 13–15 месяцев; срок беременности длится 9,5 месяца.

Коровы этой породы дают в среднем 7600 литров молока в год; продуктивность племенных животных может достигать 8100 литров в год, а в течение жизни могут производить до 26 000 литров.

В сентябре 2000 г. 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. Чтобы узнать, чем вы можете помочь, посетите сайт *animals.oreilly.com*.

Изображение для обложки взято из второго тома энциклопедии Лидеккера (Lydekker) «Royal Natural History». Текст на обложке набран шрифтами URW Typewriter и Guardian Sans. Текст книги набран шрифтом Adobe Minion Pro; текст заголовков – шрифтом Adobe Myriad Condensed; а фрагменты программного кода – шрифтом Ubuntu Mono, созданным Далтоном Магом (Dalton Maag).

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
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.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Лорин Хохштейн, Рене́ Мозер

Запускаем Ansible

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Научный редактор *Маркелов А. А.*
Перевод *Филонов Е. В., Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 21,25. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**