EXPERT INSIGHT

Python для сетевых инженеров

Автоматизация сети, программирование и DevOps

Третье издание

Эрик Чоу

Packt

Mastering Python Networking

Third Edition

Your one-stop solution to using Python for network automation, programmability, and DevOps

Eric Chou

BIRMINGHAM – MUMBAI

Python для сетевых инженеров

Третье издание

Автоматизация сети,
программирование и DevOps

Эрик Чоу

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

Эрик Чоу

Python для сетевых инженеров. Автоматизация сети, программирование и DevOps

Серия «Для профессионалов»

Перевел с английского *С. Черников*

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*А. Киселев*
Литературный редактор	*П. Лебедева*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Е. Павлович*

ББК 32.973.2-018.1
УДК 004.43

Чоу Эрик

Ч-75 Python для сетевых инженеров. Автоматизация сети, программирование и DevOps. — СПб.: Питер, 2023. — 528 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1769-7

Сети образуют основу для развертывания, поддержки и обслуживания приложений. Python — идеальный язык для сетевых инженеров, предлагающий инструменты, которые ранее были доступны только системным инженерам и разработчикам приложений. Прочитав эту книгу, вы из обычного сетевого инженера превратитесь в сетевого разработчика, подготовленного ко встрече с сетями следующего поколения.

Третье издание полностью переработано и обновлено для использования Python 3. Помимо новых глав, посвященных анализу сетевых данных с помощью стека ELK (Elasticsearch, Logstash, Kibana и Beats) и Azure Cloud Networking, в него включены сведения по использованию Ansible и фреймворков pyATS и Nornir. Кроме того, были обновлены примеры для лучшего понимания концепций и обеспечения совместимости.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1839214677 англ.

ISBN 978-5-4461-1769-7

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 08.2022. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 08.07.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 42,570. Тираж 700. Заказ 0000.

Краткое содержание

Оглавление

Моей жене Джоанне и детям Микаелин и Эсми.

Моим родителям, которые зажгли во мне страсть много лет назад.

Предисловие

Многие думают (или кто-то так им сказал), что изучение программирования и языка Python пойдет им на пользу. «Навыки программирования пользуются спросом, поэтому вы должны стать программистом». Это неплохой совет. Но лучше ответить на вопрос: как, имея определенный опыт в какой-то области, опередить своих коллег за счет автоматизации и расширения своих умений с помощью навыков разработки ПО? Именно эта цель ставится в данной книге. Вы будете знакомиться с Python в контексте настройки, администрирования и мониторинга сети.

Если вам надоело постоянно заходить на свои серверы и вводить кучу команд для настройки сети; если вы хотите быть уверенными в надежности и воспроизводимости настроек своей сети; если хотите в реальном времени следить за всем происходящим в ней, то Python — это то, что вам нужно.

Вы уже, наверное, пришли к тому, что вам необходимо овладеть навыками программирования, которые можно применить для управления сетями. В конце концов, такие термины, как *программно-определяемые сети (Software-Defined Networking, SDN)*, в последние несколько лет у всех на слуху. Но почему Python? Может быть, лучше выучить JavaScript, Go или какой-то другой язык? Возможно, стоит сделать упор на Bash и освоить разработку сценариев на языке командной оболочки?

Есть две причины, по которым Python отлично подходит для сетевых технологий.

Во-первых, на страницах этой книги Эрик покажет, что для Python написано множество библиотек (иногда их называют пакетами), предназначенных специально для работы с сетью. Простой поиск на https://pypi.org по слову *network* дает более 500 различных библиотек для автоматизации и мониторинга сети. С помощью таких библиотек, как Ansible, вы сможете создавать сложные сетевые и серверные конфигурации декларативным способом, используя простые конфигурационные файлы.

Используя Pexpect или Paramiko, вы сможете управлять устаревшими удаленными системами, как если бы у них был свой API поддержки сценариев. Если у настраиваемого вами устройства есть свой API, то для работы с ним, скорее всего, уже существует специальная библиотека на Python. Поэтому данный язык, несомненно, подходит для таких задач.

Во-вторых, Python занимает особое место среди языков программирования. Я называю его *языком полного спектра* и определяю этот термин так: очень простой в освоении язык (`print("hello world")` — как вам?) и очень мощная технология, лежащая в основе невероятных программных комплексов, таких как youtube.com.

Это редкое явление. Существуют хорошие языки для начинающих, позволяющие быстро начать программировать. Сразу вспоминается Visual Basic, а также MATLAB и другие коммерческие языки. Но у них ограниченные возможности применения. Можете ли вы себе представить Linux, Firefox или видеоигры, написанные на любом из них? Конечно же нет.

На другом конце спектра находятся очень мощные языки, такие как C++, .NET, Java и многие другие. C++ используется для создания некоторых модулей ядра Linux и крупных программных проектов, таких как Firefox. Однако эти языки не для новичков. Чтобы начать писать код, вам придется разобраться в указателях, компиляторах, компоновщиках, заголовочных файлах, классах, модификаторах доступа (public/private) и т. д.

Python совмещает в себе лучшее из этих двух миров. С одной стороны, на нем очень легко написать что-то полезное, уложившись в несколько строчек кода и использовав простые концепции программирования. С другой — его все чаще применяют в весьма значительных проектах: YouTube, Instagram, Reddit и т. д. Компания Microsoft реализовала на Python *интерфейс командной строки* (Command Line Interface, CLI) для Azure (хотя для работы с ним не нужно знать или использовать Python).

Итак, подытожим. Умение программировать — это суперсила, способная вывести ваши инженерные навыки на новый уровень. Python — один из самых быстроразвивающихся и популярных языков программирования в мире. Кроме того, для Python имеется множество высококачественных сетевых библиотек. Книга «Python для сетевых инженеров» освещает все вышеперечисленное и изменит ваше представление о работе с сетями. В добрый путь!

Майкл Кеннеди, основатель подкаста Talk Python,
Портленд, штат Орегон, США

Введение

В 2014 году на конференции Cisco Live я провел первый семинар Coding 101 по Python и REST API для сетевых инженеров в DevNet Zone. В помещении было полно известных сетевых инженеров и архитекторов, многие из которых в этот день сделали свой первый API-вызов. После этого мне посчастливилось работать с сетевыми инженерами со всего мира, которые решили добавить в число своих навыков программирование.

Отделы информационных технологий и администрирования начинают меняться. Я считаю, что в будущем сетевые инженеры и разработчики ПО будут объединяться в единые команды. Для развертывания современных приложений необходимы масштабируемые, сложные и безопасные сети. И чтобы сделать управление этими сетями воспроизводимым, надежным и динамичным, требуется автоматизация.

Сетевые инженеры имеют большой опыт решения всевозможных проблем. Если в их набор инструментов управления сетями добавить Python, автоматизацию и умение работать с API, то получится потрясающая комбинация. С помощью этих дополнительных технологий инженеры смогут по-новому подходить к решению имеющихся проблем и браться за немыслимые ранее задачи. Эта книга — ценный ресурс как для сетевых инженеров, которые хотят освоить программирование, так и для разработчиков ПО, желающих воспользоваться преимуществами новой программируемой инфраструктуры.

Инженеры часто меня спрашивают: «С чего начать?» Мой совет: начинайте с простого. Выберите «извечные» проблемы, с которыми сталкивается ваша команда, и попытайтесь автоматизировать диагностику и сбор информации. Затем попробуйте автоматизировать передачу собранного в систему тикетов или в приложение мониторинга. Вскоре у вас начнет вырисовываться рабочий процесс. Такое постепенное внедрение нововведений поможет укрепить доверие к автоматизации в команде и позволит ее членам освоиться с новыми инструментами.

Сначала сосредоточьтесь на приобретении базовых навыков программирования, которые пригодятся вам в любом проекте, включая программирование на Python и REST API. Также уделите внимание таким инструментам, как Git и GitHub, которые помогут вам управлять своим исходным кодом и организовать совместную работу. Уделите время подготовке среды разработки. Попробуйте разные

редакторы для написания кода и инструменты для исследования API, такие как Postman и curl. Изучите приемы обработки JSON и XML. Начните исследовать методологии разработки программного обеспечения, такие как *разработки через тестирование* (*Test-Driven Development*, *TDD*) и основные принципы DevOps[1].

Эта книга отлично подойдет для приобретения этих навыков и поможет вам изучить все эти темы в контексте работы с сетью. Она начинается с использования Python для простых взаимодействий с сетевыми устройствами посредством CLI и API, а затем переходит к фреймворку автоматизации общего назначения — и все это с точки зрения сетевых инженеров. Попутно Эрик демонстрирует примеры кода на Python для обеспечения сетевой безопасности, мониторинга и построения своего API с помощью фреймворка Flask. Знакомит с облачными сетевыми технологиями на примере AWS и Azure и общепринятыми инструментами DevOps, такими как Git, Jenkins и TDD. В своих объяснениях Эрик использует прагматичный подход, основанный на практическом опыте. Это поможет вам внедрить изученные концепции в свою работу.

DevOps и облачные технологии трансформируют нашу индустрию. Команды, занимающиеся обслуживанием сетей, разработкой ПО и администрированием, начинают по-новому взаимодействовать, разделяя ответственность за достижение общих бизнес-целей. Сетевые инженеры с навыками программирования помогут определить ход этой трансформации.

Найдите себе какой-нибудь проект и начните оттачивать эти навыки в его контексте. Выберите какое-то простое действие, которое вам бы хотелось надежно воспроизводить, и попытайтесь его автоматизировать. Начните писать код с самого начала — и делайте это почаще. Найдите или организуйте себе рабочее место для экспериментов. Чем больше вы будете экспериментировать, тем быстрее пойдет обучение.

Инновации, которые станут результатом совместной работы сетевых инженеров и разработчиков в ближайшие пять лет, будут революционными.

Сделайте первые шаги навстречу будущему и не забудьте отпраздновать первый успешный ответ `200 OK` своего API.

Удачного программирования!

Мэнди Уэйли,
старший директор по поддержке разработчиков, Cisco DevNet

---

[1] DevOps — интеграция разработки и эксплуатации. Форма организации труда, главной целью которой является налаживание сотрудничества службы эксплуатации (Ops) и команды разработчиков (Dev). — *Примеч. ред.*

Об авторе

Эрик Чоу — IT-технолог с более чем двадцатилетним опытом. Имел дело с крупнейшими сетями в индустрии, работая в Amazon, Azure и других компаниях из списка Fortune 500. Эрик страстно увлекается автоматизацией сетей и языком Python и помогает организациям в создании эффективных механизмов безопасности.

Он также соавтор книги *Distributed Denial of Service (DDoS): Practical Detection and Defense* (O'Reilly Media).

Еще на счету Эрика два американских патента в сфере IP-телефонии. Своим глубоким интересом к технологиям он делится в книгах, лекциях и блоге, а также участвует в некоторых популярных проектах с открытым кодом на языке Python.

Я бы хотел поблагодарить членов сообщества и разработчиков открытого ПО, сетевых архитектур и Python, которые щедро делятся своими знаниями и кодом. Без них многие проекты, упоминаемые в этой книге, никогда бы не увидели свет. Надеюсь, я тоже внес свой скромный вклад.

Благодарю команду издательства Packt: Тушара, Тома, Иэна, Алекса, Джона и многих других за возможность совместной работы над третьим изданием этой книги. Особая благодарность Рикарду Кёрке за согласие вычитать мой текст.

Спасибо вам, Мэнди и Майкл, что написали предисловие к этой книге. Мне сложно выразить словами, насколько я вам признателен. Вы молодцы!

Моим родителям и моей семье: ваши постоянные поддержка и ободрение сделали меня тем, кто я есть. Я вас люблю.

О научном редакторе

Рикард Кёрке работает консультантом по NetDevOps в компании SDNit в группе опытных технических специалистов, проявляющих глубокий интерес и внимание к перспективным сетевым технологиям.

Программист-самоучка, которого в основном интересует Python. В число его ежедневных обязанностей входит управление сетевыми устройствами с помощью таких средств оркестрации, как Ansible.

Он также был научным редактором книги Марка Г. Собеля *A Practical Guide to Linux Commands, Editors and Shell Programming. Third Edition*.

Вступление

Как написал Чарльз Диккенс в «Повести о двух городах», «это было лучшее изо всех времен, это было худшее изо всех времен; это был век мудрости, это был век глупости». Эти на первый взгляд противоречивые слова идеально описывают хаос и настроения, царящие во времена перемен и преобразований. Мы, несомненно, испытываем нечто подобное в связи со стремительными изменениями, происходящими в области сетевых технологий. По мере того как разработка ПО все глубже проникает во все аспекты сетей, традиционный интерфейс командной строки и вертикально интегрированные сетевые методики перестают быть оптимальными способами управления современными сетями. С точки зрения сетевых инженеров, изменения, которые мы наблюдаем, волнующи: они открывают новые возможности, но в то же время создают трудности, особенно для тех, кому приходится быстро адаптироваться, чтобы поспевать за новшествами. Эта книга написана с целью облегчить переход для сетевых специалистов и содержит практическое руководство, в котором объясняется, как внедрить в традиционную платформу методики из мира разработки ПО.

В качестве языка программирования для решения задач управления сетями мы будем использовать Python. Это легкий в изучении высокоуровневый язык, который помогает сетевым инженерам эффективно реализовывать их творческие способности и применять навыки решения проблем и облегчает их повседневную работу. Python становится неотъемлемой частью многих крупномасштабных сетей, и с помощью этой книги я попытаюсь поделиться с вами усвоенными мною уроками.

С момента выхода первого и второго изданий я провел интересные и содержательные беседы со многими моими читателями. Я был польщен успехом этой книги и серьезно отнесся к полученным отзывам. В третьем издании я попытался охватить много новых библиотек и обновил примеры, использовав самое свежее программное и аппаратное обеспечение. Я также добавил две главы, которые, как мне кажется, будут важны для сетевых инженеров в сегодняшних условиях.

Времена перемен открывают отличные возможности для технического прогресса. Концепции и инструменты, описанные в этой книге, очень помогли моей карьере, и я надеюсь, что ту же роль они сыграют и в вашей жизни.

Кому подойдет эта книга

Эта книга идеально подойдет ИТ-специалистам и инженерам, которые занимаются администрированием сетевых устройств и хотят расширить свои знания о Python и других инструментах для решения сетевых проблем. Желательно иметь хотя бы базовое понимание сетевых технологий и Python.

Какие темы здесь освещаются

Глава 1. Обзор протоколов TCP/IP и Python. Содержит краткий обзор фундаментальных технологий, из которых состоят современные интернет-коммуникации, начиная с OSI и клиент-серверной модели и заканчивая протоколами TCP, UDP и IP. Здесь мы поговорим об основах языка Python: типах данных, операторах, циклах, функциях и пакетах.

Глава 2. Низкоуровневое взаимодействие с сетевыми устройствами. На практических примерах иллюстрирует использование Python для выполнения команд в контексте сетевого устройства. Также обсуждает трудности автоматизации с одним лишь CLI-интерфейсом. В примерах используются библиотеки Pexpect, Paramiko, Netmiko и Nornir.

Глава 3. API и IDN-сети. Обсуждает новейшие сетевые устройства с поддержкой *интерфейсов прикладного программирования (Application Programming Interface, API)* и других высокоуровневых методов взаимодействий. Здесь также рассматриваются инструменты, позволяющие абстрагировать низкоуровневые задачи и сосредоточиться на общих целях, которые стоят перед сетевыми инженерами. Обсуждение и примеры будут касаться Cisco NX-API, Meraki, Juniper PyEZ, Arista Pyeapi и Vyatta VyOS.

Глава 4. Фреймворк автоматизации на Python: основы Ansible. Обсуждает основы Ansible, открытого фреймворка автоматизации на Python. Ansible — это дальнейшее развитие идеи API с акцентом на декларативное описание задач. В этой главе описываются преимущества системы Ansible и ее высокоуровневой архитектуры, а также демонстрируются примеры использования с устройствами Cisco, Juniper и Arista.

Глава 5. Фреймворк автоматизации на Python: следующий уровень. Продолжает предыдущую главу и охватывает более сложные аспекты Ansible. Здесь мы поговорим об условных выражениях, циклах, шаблонах, переменных, Ansible Vault и ролях, а также затронем основы создания собственных модулей.

Глава 6. Сетевая безопасность с использованием Python. Представляет несколько инструментов, написанных на Python, которые помогут вам защитить вашу сеть. Обсуждает использование Scapy для тестирования безопасности, Ansible для быстрой реализации списков доступа и Python для ретроспективного анализа инцидентов в сети.

Глава 7. Сетевой мониторинг с использованием Python: часть 1. Охватывает мониторинг сетей с помощью различных инструментов. Содержит примеры использования SNMP и PySNMP для получения информации из устройств. Демонстрирует примеры использования Matplotlib и Pygal для представления результатов в графическом виде. В конце показан пример использования сценария на Python в роли источника данных для Cacti.

Глава 8. Сетевой мониторинг с использованием Python: часть 2. Охватывает дополнительные инструменты мониторинга. Вначале мы создадим графическую диаграмму сети из LLDP-данных с использованием Graphviz. Затем перейдем к пассивному сетевому мониторингу на основе Netflow и других технологий. Мы декодируем поток пакетов с помощью Python и применим ntop для визуализации результатов. Здесь вы также найдете краткий обзор системы Elasticsearch и применения для мониторинга сети.

Глава 9. Создание сетевых веб-сервисов с помощью Python. Здесь мы покажем, как создать собственный API для автоматизации управления сетью с помощью веб-фреймворка Python Flask. Преимущества таких API: сокрытие деталей организации сети от запрашивающей стороны, объединение и настройка операций, а также обеспечение повышенной безопасности за счет ограничения доступа к поддерживаемым операциям.

Глава 10. Облачные сетевые технологии AWS. Показывает, как с помощью AWS можно создать функциональную и устойчивую виртуальную сеть. Охватывает технологии виртуальных приватных облаков, такие как CloudFormation, таблицы маршрутизации VPC, списки доступа, Elastic IP, шлюзы NAT, Direct Connect и другие связанные с этим темы.

Глава 11. Облачные сетевые технологии Azur. Рассказывает о сетевых сервисах от Azur и о том, как в этом облаке можно создавать собственные сервисы. Здесь мы обсудим Azure VNet, Express Route and VPN, сетевые балансировщики нагрузки Azure и другие связанные с этим технологии.

Глава 12. Анализ сетевых данных с помощью Elastic Stack. Показывает, как использовать Elastic Stack в роли набора тесно интегрированных инструментов для анализа и мониторинга сети. Здесь обсуждается множество тем, от установки, настройки, импортирования данных с помощью Logstash и Beats и поиска данных с применением Elasticsearch до визуализации с использованием Kibana.

Глава 13. Работа с Git. Иллюстрирует роль Git для организации совместной работы и управления версиями кода. Демонстрирует примеры использования Git в повседневной практике администрирования сетей.

Глава 14. Непрерывная интеграция с помощью Jenkins. Рассказывает о применении Jenkins для автоматического создания конвейеров операций, которые могут сэкономить наше время и повысить надежность.

Глава 15. TDD для сетей. Объясняет, как с помощью Python-пакетов `unittest` и `pytest` создавать простые тесты для проверки нашего кода. Демонстрирует примеры написания тестов, проверяющих достижимость, сетевые задержки, безопасность и сетевые транзакции. Здесь вы также увидите, как внедрить тесты в инструменты непрерывной интеграции, такие как Jenkins.

Как извлечь максимум из этой книги

Чтобы получить максимальную пользу, читателю желательно иметь базовый опыт сетевого администрирования и знать Python. Главы можно читать в произвольном порядке, за исключением 4-й и 5-й. Помимо основных программных и аппаратных средств, представленных в начале книги, в каждой главе будут описываться дополнительные новые инструменты.

Настоятельно рекомендуем выполнять показанные здесь примеры в своей экспериментальной сетевой среде.

Загрузка файлов с примерами кода

Архив с примерами кода из этой книги также доступен на GitHub по адресу https://github.com/PacktPublishing/Mastering-Python-Networking-Third-Edition. Любые обновления будут вноситься в этот репозиторий.

У нас есть и другие архивы с кодом для нашей богатой коллекции книг и видеороликов, доступные по адресу https://github.com/PacktPublishing/. Загляните туда!

Загрузка полноцветных иллюстраций

Мы также предоставляем PDF-файл с цветными изображениями оригинальных снимков экранов и диаграмм, использованных в этой книге. Он доступен по адресу https://static.packt-cdn.com/downloads/9781839214677_ColorImages.pdf.

Условные обозначения

В этой книге используется ряд соглашений по оформлению текста.

Моноширинным шрифтом выделены код в тексте, имена таблиц баз данных, папок, файлов, расширений файлов, фиктивные URL, ввод пользователя и теги Twitter. Например, «В автоматически сгенерированной конфигурации также открыт доступ к `vty` по протоколам telnet и SSH».

Блоки кода оформляются так:

```
# Это комментарий
print("hello world")
```

Курсивом выделяются новые термины и важные слова. Например: «В следующем разделе мы продолжим обсуждать SNMP, но уже в контексте полнофункциональной системы мониторинга сети под названием *Cacti*».

Так обозначаются предупреждения и важные замечания.

Так обозначаются советы и подсказки.

От издательства

Обратите внимание: виртуальная лаборатория VIRL от Cisco, подробно описанная в главе 2 и используемая в примерах книги, на момент выхода данного издания уже не существует в представленном автором виде. Начиная с релиза 2.0, этот инструмент носит название Cisco Modeling Labs. Тем не менее книга, которую вы держите в руках, не теряет от этого свою ценность, а проекты на github и другие продукты Cisco, упоминаемые автором в контексте VIRL, актуальны и для новой версии лаборатории.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Обзор TCP/IP и Python

Добро пожаловать в век новых и удивительных сетевых технологий! Когда на рубеже тысячелетий я начинал работать сетевым инженером, моя роль тогда определенно отличалась от роли сетевого инженера сегодня. В то время сетевые инженеры управляли и обслуживали локальные и глобальные сети с помощью интерфейса командной строки. Иногда нам приходилось пересекать границу, разделяющую профессии сетевого инженера и разработчика, чтобы решать свои задачи, но вообще от нас не требовалось писать код или понимать концепции программирования. Сегодня все иначе.

С годами DevOps, *программно-определяемые сети (Software-Defined Networking, SDN)* и прочие технологии существенно размыли границы между ролями сетевого, системного инженера и разработчика.

Раз вас заинтересовала эта книга — значит, вы либо уже внедрили у себя DevOps для управления сетями, либо рассматриваете возможность перехода на программируемые сети. Возможно, вы, как и я, много лет работали сетевым инженером и захотели разобраться в языке программирования Python, о котором столько разговоров. А может быть, вы уже свободно владеете этим языком, но хотите узнать, как его применять в сетевых технологиях.

Если вы принадлежите к любой из этих категорий или просто интересуетесь языком Python в контексте сетевых технологий, я убежден, что эта книга — для вас (рис. 1.1).

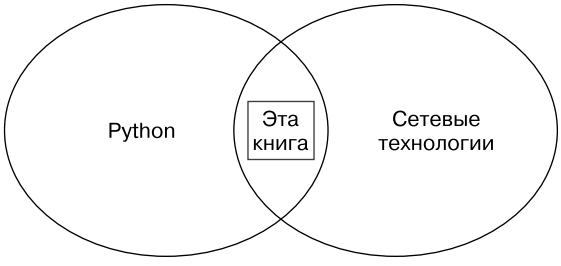


Рис. 1.1. Как пересекаются Python и сетевые технологии

Во многих учебниках сетевые технологии и язык Python рассматриваются как отдельные темы. Я выбрал другой подход. Предполагается, что у читателей этой книги есть некоторый практический опыт управления сетями и общее представление о сетевых протоколах. Знакомство с Python как языком тоже не помешает, но его основы будут рассмотрены далее в этой главе. Чтобы извлечь максимум из этой книги, вовсе не нужно быть специалистом по Python или в сетевых технологиях. Мы будем отталкиваться от базовых понятий, чтобы читатель мог изучить и опробовать различные приемы, которые могут упростить его жизнь.

В этой главе мы рассмотрим некоторые базовые понятия сетевых технологий и Python, а затем определим, какими знаниями нужно обладать, чтобы получить максимальную выгоду от этой книги. Если вам потребуется освежить свои знания, то в интернете есть множество бесплатных и недорогих ресурсов, которые помогут вам в этом. Могу посоветовать бесплатные лекции на Khan Academy (https://www.khanacademy.org/) и руководства по Python (https://www.python.org).

Мы кратко пробежимся по темам, касающимся сетевых технологий. Как показывает мой опыт работы в этой области, типичный сетевой инженер или разработчик может не помнить наизусть все состояния *протокола передачи данных (Transmission Control Protocol, TCP)* — я так точно не помню — и это не мешает ему решать повседневные задачи. Однако такой инженер обычно знаком с основами модели *взаимодействий открытых систем (Open Systems Interconnection, OSI)*, принципами работы протоколов TCP и *передачи пользовательских датаграмм (User Datagram Protocol, UDP)*, различными полями в IP-заголовках и другими базовыми понятиями.

Кроме того, бегло рассмотрим Python, чтобы читатели, которые не пишут на этом языке регулярно, могли себя уверенно чувствовать при чтении следующих глав.

В частности, в этой главе вас ждет краткий обзор:

- устройства интернета;
- модели OSI и архитектуры «клиент — сервер»;
- набора протоколов TCP, UDP и IP;
- синтаксиса, типов данных, операторов и циклов в языке Python;
- приемов расширения Python с помощью функций, классов и пакетов.

Конечно, представленная здесь информация не является исчерпывающей, поэтому при необходимости обращайтесь к дополнительным материалам по приводимым ссылкам.

Работу сетевых инженеров затрудняют масштаб и сложность сетей. Мы имеем дело и с небольшими домашними сетями, и со средними сетями в малом бизнесе, и с крупными сетями корпораций международного масштаба. Самая большая из всех сетей, конечно, интернет. Без него у нас не было бы электронной почты, веб-сайтов, служб, потокового мультимедиа или облачной обработки данных. Поэтому, прежде чем погружаться в подробности протоколов и Python, поговорим об интернете.

Краткий обзор интернета

Что такое интернет? Ответ на этот, казалось бы, простой вопрос зависит от того, в какой области вы работаете. Люди вкладывают в это слово разный смысл; молодежь, старики, студенты, учителя, предприниматели, поэты имеют собственное мнение на этот счет.

С точки зрения сетевого инженера, интернет — это глобальная вычислительная сеть, объединяющая мелкие и крупные сети. Иными словами, это сеть сетей без центрального управления. Возьмите, к примеру, свою домашнюю сеть. Она состоит из устройства, выполняющего функции маршрутизатора, коммутатора и беспроводной точки доступа, и позволяет взаимодействовать вашим смартфонам, планшетам, компьютерам и смарт-телевизорам. Это ваша *локальная вычислительная сеть (Local Area Network, LAN)*.

Когда вашей домашней сети нужно обратиться к внешнему миру, она передает информацию из LAN в более крупную сеть, принадлежащую вашему *поставщику услуг интернета*, или *интернет-провайдеру (Internet Service Provider, ISP)*. Обычно это компания, которой вы платите за доступ к интернету. Провайдер

подключает мелкие сети к сетям большего масштаба, которые он обслуживает. Сеть провайдера часто содержит граничные узлы, которые направляют трафик в магистральную сеть. Задача магистральной сети — связать эти граничные узлы с помощью высокоскоростных соединений.

Специальные граничные узлы соединяют сети разных провайдеров, чтобы ваш трафик мог попасть к адресату. Ответ, отправленный вашему домашнему компьютеру, планшету или смартфону, может идти по другому маршруту, однако отправитель и получатель остаются неизменными.

Рассмотрим компоненты, из которых состоят сети.

Серверы, хосты и сетевые компоненты

Хост — это конечный узел сети, который взаимодействует с другими узлами. В современных реалиях хостом может быть обычный компьютер, смартфон, планшет или телевизор. С появлением *интернета вещей (Internet of Things, IoT)* определение хоста расширилось и теперь охватывает IP-камеры, ресиверы цифрового телевидения и всевозможные датчики, которые применяются в сельском хозяйстве, автомобилях и т. д. Учитывая взрывной рост количества хостов, подключенных к интернету, все эти устройства нужно как-то адресовать, маршрутизировать и администрировать. Мы наблюдаем беспрецедентный спрос на зрелые сетевые технологии.

Большую часть времени, которое мы проводим в интернете, занимает выполнение запросов к сервисам. Это может быть просмотр веб-страницы, отправка или получение электронной почты, передача файлов и т. д. За работу этих сервисов отвечают *серверы*. Сервер предоставляет услуги множеству узлов и обычно обладает мощной аппаратной конфигурацией. В каком-то смысле серверы — это суперузлы сети, расширяющие возможности остальных узлов. Мы еще вернемся к ним, когда будем обсуждать клиент-серверную модель.

Если представить серверы и хосты в виде мегаполисов и небольших городов, *сетевые компоненты* будут играть роль дорог и автострад, соединяющих их. При описании сетевых компонентов, которые передают по всему миру постоянно растущие потоки битов и байтов, вспоминается термин *«информационная магистраль»*. В модели OSI, которую мы рассмотрим далее, этими компонентами являются устройства с первых трех уровней, иногда и с четвертого. Это, к примеру, маршрутизаторы и коммутаторы канального и сетевого уровней, которые направляют трафик, а также транспортная инфраструктура вроде оптоволоконных и коаксиальных кабелей, витой пары, а иногда и оборудова-

ния для *спектрального уплотнения каналов (Dense Wavelength Division Multiplexing, DWDM)*.

В совокупности хосты, серверы, хранилища данных и сетевые компоненты составляют то, что принято считать современным интернетом.

Появление дата-центров

В предыдущем разделе мы обсудили разные роли, которые играют серверы, хосты и сетевые компоненты в межсетевых взаимодействиях. Ввиду повышенных требований к аппаратным ресурсам серверы зачастую размещаются в каком-то одном месте, чтобы ими можно было эффективно управлять. Такое место называется центром обработки данных (ЦОД), или просто дата-центром.

Корпоративные дата-центры

Бизнес-нужды типичной компании — это электронная почта, хранилище документов, система отслеживания продаж, система управления заказами, инструменты для отдела кадров и внутренняя сеть для обмена знаниями. Для поддержки этих сервисов нужны файловый и почтовый серверы, серверы баз данных и веб-серверы. В отличие от рабочих станций это, как правило, высокопроизводительные компьютеры, потребляющие много электроэнергии и требующие хорошей системы охлаждения и быстрых сетевых соединений. К тому же такое оборудование зачастую создает много шума, что недопустимо для обычных условий труда. Серверы, как правило, размещаются в каком-то одном помещении в здании компании, которое называется *главным распределительным пунктом (Main Distribution Frame, MDF)*; это позволяет подвести к нему необходимое электропитание, установить резервный генератор, предусмотреть охлаждение и соединить все это в сеть.

Прежде чем попасть в MDF, пользовательский трафик обычно агрегируется недалеко от самого пользователя, в месте, которое иногда называют *промежуточным распределительным пунктом (Intermediate Distribution Frame, IDF)*. Нередко структура IDF-MDF повторяет план помещений в здании компании или на территории учебного заведения. Например, на каждом этаже может находиться свой пункт IDF, который собирает трафик и направляет его в MDF, размещенный в другой части здания. Если предприятие размещено в нескольких зданиях, может потребоваться дополнительная агрегация трафика с последующей передачей его в корпоративный дата-центр.

Корпоративные дата-центры обычно имеют трехуровневую сетевую архитектуру, включающую уровень доступа, уровень распределения и системный уровень. Конечно, как в любой архитектуре, здесь нет жестких правил и универсальных решений; это всего лишь общий подход. Например, если наложить эту трехуровневую архитектуру на модель «пользователь — IDF — MDF», то получится, что уровень доступа — это порты, к которым подключается каждый пользователь, IDF можно считать уровнем распределения, а системный уровень состоит из соединений, ведущих к MDF и корпоративным дата-центрам. Это, конечно же, обобщенный взгляд на корпоративные сети; некоторые компании используют другие модели.

Облачные дата-центры

С появлением облачных вычислений и программного обеспечения, объединяемых термином *«инфраструктура как услуга» (Infrastructure as a Service, IaaS)*, облачные провайдеры стали строить по-настоящему огромные дата-центры, которые иногда называют гипермасштабными. Под облачными вычислениями имеются в виду вычислительные ресурсы, доступ к которым осуществляется по мере необходимости и которые не требуют ручного управления со стороны пользователя. Такие услуги, к примеру, предоставляются компаниями Amazon, Microsoft и Google.

Учитывая количество серверов, которое необходимо вместить, облачные дата-центры обычно имеют куда более высокие требования к электропитанию, охлаждению и пропускной способности сети по сравнению с корпоративными ЦОД. Даже после многих лет использования облачных сервисов я не перестаю удивляться масштабу этих дата-центров, когда мне приходится посещать их лично. Они настолько большие и потребляют столько электроэнергии, что их обычно строят рядом с электростанциями, чтобы получить самый низкий тариф и минимизировать потери в электросети. При выборе места строительства также учитываются высочайшие требования к охлаждению. Например, компания Facebook выбрала для своего дата-центра город Лулео на севере Швеции (113 километров южнее полярного круга) отчасти из-за низкой температуры воздуха. В любой поисковой системе можно найти поразительные цифры, связанные с наукой проектирования и обслуживания облачных дата-центров для таких компаний, как Amazon, Microsoft, Google и Facebook. Например, ЦОД компании Microsoft в Вест-Де-Мойне, штат Айова, занимает 110 000 квадратных метров плюс еще 80 гектаров окружающей территории; чтобы обеспечить его строительство, городу пришлось вложить в общественную инфраструктуру 65 миллионов долларов (рис. 1.2).

Рис. 1.2. Дата-центр в штате Юта
(источник: https://en.wikipedia.org/wiki/Utah_Data_Center)

В масштабах облачных провайдеров размещение сервисов на одном сервере экономически невыгодно или просто невозможно. Сервисы распределяют между множеством серверов, иногда размещенных в разных зданиях, чтобы обеспечить избыточность и гибкость для владельцев сервисов.

Требования ко времени отклика и избыточности, а также физический разброс серверов создают огромную нагрузку на сеть. Для соединения множества серверов между собой требуется огромное количество сетевого оборудования, такого как кабели, коммутаторы и маршрутизаторы. И каждый из этих компонентов нужно поместить в стойку, подключить и затем обслуживать. Архитектура типичного дата-центра имеет вид многокаскадной сети Клоза (рис. 1.3).

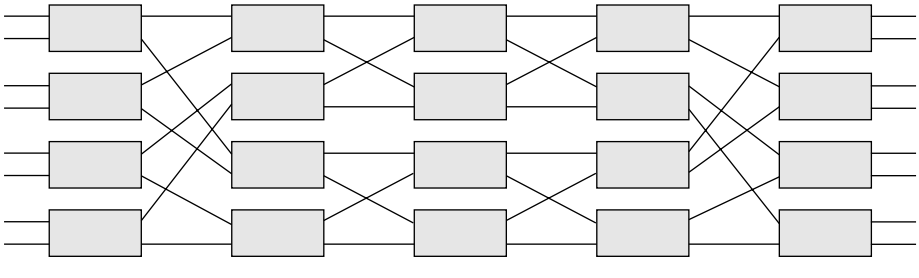


Рис. 1.3. Сеть Клоза

Для обеспечения производительности, гибкости и надежности облачных дата-центров часто требуется автоматизация сети. Традиционные способы управления сетевыми устройствами с помощью терминала и интерфейса командной

строки не позволяют вводить сервисы в работу в разумные сроки. Не говоря уже о том, что выполнение рутинных задач вручную чревато ошибками, неэффективно и по сути является пустой тратой инженерных талантов. Ситуация усложняется еще и тем, что быстро меняющиеся потребности бизнеса часто требуют оперативного изменения конфигурации сети.

Мой собственный путь к автоматизации сетей с помощью Python начался несколько лет назад именно в облачных дата-центрах, и я еще ни разу об этом не пожалел.

Граничные дата-центры

Если у нас достаточно вычислительных мощностей на уровне дата-центра, зачем размещать наше оборудование где-то еще? Клиентские соединения со всего мира можно направить к серверам в этом дата-центре, и этого будет достаточно? Ответ, конечно, зависит от ситуации. Самое серьезное ограничение заключается в задержках при маршрутизации запросов и сеансов между клиентом и крупным дата-центром: большие задержки превращают сеть в узкое место.

Конечно, любой, кто знаком с элементарной физикой, понимает, что задержка сети не может быть нулевой: даже свету в вакууме нужно какое-то время на преодоление расстояния. А в реальных условиях задержка будет куда выше. Почему? Потому что сетевой пакет должен пройти через множество сетей, а иногда и по подводному кабелю, через далекий спутник связи, сотовые сети 3G или 4G, Wi-Fi-соединения и т. д.

Как уменьшить задержку сети? Одно из решений: сократить количество сетей, через которые проходят клиентские запросы. Граница вашей сети должна находиться как можно ближе к конечному пользователю и обладать достаточными для обслуживания запросов ресурсами. Такой подход часто применяется для раздачи медиаконтента — музыки и видео.

Представьте, что вы занимаетесь созданием сервиса видеовещания следующего поколения. Чтобы порадовать своих клиентов идеальным видеопотоком, вам следует разместить свой сервер раздачи потокового видео как можно ближе к ним — либо внутри сети провайдера, либо рядом с ней. Также, чтобы зарезервировать ресурсы и повысить скорость соединения, исходящий сетевой канал серверной фермы должен быть подключен к как можно большему числу интернет-провайдеров, чтобы уменьшить количество транзитных участков. Все соединения должны иметь достаточную пропускную способность, чтобы не увеличивать задержки в периоды пиковых нагрузок. Эти требования привели к появлению граничных дата-центров для пирингового обмена данными

между крупными интернет- и контент-провайдерами. Сетевых устройств в них не так много, как в облачных ЦОД, но они тоже могут извлечь пользу из автоматизации сетей в плане надежности, гибкости, безопасности и наблюдаемости.

Далее в книге мы еще вернемся к вопросам безопасности (см. главу 6) и наблюдаемости (см. главы 7 и 8).

Многие сложные системы делят на меньшие части для уменьшения сложности, аналогично в основу сетевых взаимодействий была положена концепция уровней. С годами было разработано несколько сетевых моделей. В этой книге мы поговорим о двух самых важных и начнем с модели OSI.

Модель OSI

Никакую книгу по сетям нельзя считать полной без обзора модели OSI. В этой основополагающей модели телекоммуникационные функции делятся на уровни. Всего имеется семь независимых уровней, располагающихся один поверх другого и обладающих четкой структурой и характеристиками.

Например, поверх различных протоколов канального уровня, таких как Ethernet и Frame Relay, располагается протокол IP сетевого уровня. Эталонная модель OSI — отличное средство разделения разнообразных технологий по понятным группам, ни у кого не вызывающим возражений. Благодаря этому люди могут сосредоточиться на своих задачах в рамках одного уровня, не слишком заботясь о совместимости (рис. 1.4).

Изначально модель OSI разрабатывалась в конце 1970-х и позже была опубликована *Международной организацией по стандартизации (International Organization for Standardization, ISO)*, известной ныне под названием *Международного консультационного комитета по телефонии и телеграфии (Telecommunication Standardization Sector of the International Telecommunication Union, ITU-T)*. Это общепринятый стандарт, на основе которого обычно создаются новые телекоммуникационные технологии.

Примерно в то же время происходило становление интернета. Эталонную модель, которую использовал его создатель, часто называют TCP/IP. TCP и IP — это протоколы, составлявшие оригинальную архитектуру интернета. Они перекликаются с моделью OSI в том смысле, что передача данных между узлами в них разделена на абстрактные уровни.

Модель OSI			
Уровень		**Тип данных протокола (PDU)**	**Функции**
Уровни хоста	7. Прикладной	Данные	Высокоуровневые API, включая совместное использование ресурсов, доступ к удаленным файлам
	6. Представления		Передача данных между сетевыми сервисами и приложениями, включая преобразование кодировок символов, сжатие данных и шифрование/расшифровывание
	5. Сеансовый		Управление сеансами связи, например организация непрерывного обмена информацией в форме многократной передачи туда-обратно между двумя узлами
	4. Транспортный	Сегменты (TCP)/ датаграммы (UDP)	Надежная передача сегментов данных между узлами в сети, включая сегментирование, подтверждение получения и мультиплексирование
Уровни среды передачи данных	3. Сетевой	Пакеты	Структурирование и управление сетями с несколькими узлами, включая адресацию, маршрутизацию и управление трафиком
	2. Канальный	Биты/ кадры	Надежная передача кадров данных между двумя узлами, связанными на физическом уровне
	1. Физический	Биты	Передача и прием двоичных данных через физическую среду

Рис. 1.4. Модель OSI

Разница в том, что TCP/IP объединяет уровни 5–7 модели OSI в *прикладной* уровень, а *физический* и *канальный* — в *уровень сетевого доступа* (канальный) (рис. 1.5).

И OSI, и TCP/IP можно использовать в качестве стандартов для сквозной передачи данных. В этой книге основное внимание уделяется модели TCP/IP, так как именно на ней основан интернет. Но мы будем иногда вспоминать модель OSI — например, в ходе обсуждения веб-фреймворка в следующих главах. Помимо моделей транспортного уровня, существуют также модели управления сетевыми взаимодействиями на прикладном уровне. В современных сетях большинство приложений основано на клиент-серверной архитектуре. О ней и пойдет речь в следующем разделе.

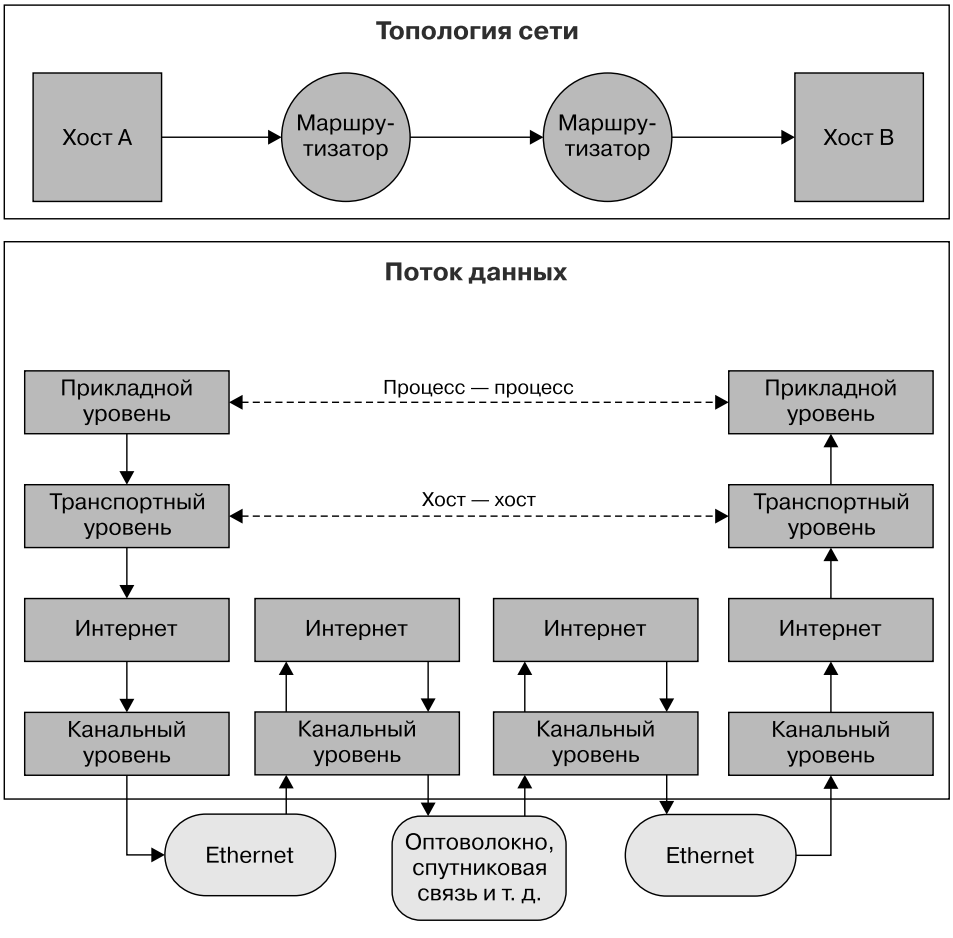


Рис. 1.5. Набор интернет-протоколов

Клиент-серверная модель

Эталонная клиент-серверная модель определяет стандартный способ обмена данными между двумя узлами. Конечно, сегодня ни для кого не секрет, что не все узлы равнозначны. Даже во времена *ARPANET (Advanced Research Projects Agency Network)* одни узлы играли роль рабочих станций, а другие — серверов, снабжающих рабочие станции данными. Серверные узлы обычно обладают более мощной аппаратной конфигурацией и находятся под более пристальным вниманием инженеров. Они предоставляют ресурсы и услуги

другим узлам сети, поэтому их вполне обоснованно называют серверами (от англ. server — «обслуживающее устройство»). Серверы, как правило, пассивно ждут, когда к ним обратятся за ресурсами. Эту модель распределенных ресурсов, которые запрашиваются клиентами, называют клиент-серверной архитектурой.

Почему это важно? Если задуматься, отношения между клиентами и серверами являются ярким свидетельством значимости сетевых технологий. Если бы клиенту и серверу не нужно было обмениваться услугами (сервисами), у нас не было бы необходимости во взаимных сетевых соединениях. Именно тот факт, что клиент и сервер посылают друг другу биты и байты, иллюстрирует важность сетевых технологий. Всем известно, как самая крупная из сетей, интернет, изменила мир и теперь играет важную роль в нашей жизни.

Вы можете спросить, как узлы определяют время, скорость, адрес отправителя и получателя каждый раз, когда им нужно обменяться данными? Это подводит нас к вопросу о сетевых протоколах.

Наборы сетевых протоколов

На ранних этапах развития компьютерных сетей протоколы были закрытыми и тесно контролировались компаниями, которые разрабатывали методы передачи данных. Хосты, использующие протокол *IPX/SPX* от Novell, не могли общаться с хостами, использующими *AppleTalk* от Apple, и наоборот. Закрытые наборы протоколов обычно делились на уровни по аналогии с эталонной моделью OSI и следовали архитектуре «клиент — сервер», но не были совместимы между собой. Они в основном работали в изолированных локальных сетях, не связанных с внешним миром. Когда трафик нужно было передать за пределы локальной сети, обычно использовали устройство вроде маршрутизатора, которое транслировало данные из одного протокола в другой. Например, чтобы подключить к интернету сеть на основе AppleTalk, маршрутизатор преобразовывал этот протокол в IP. Это промежуточное преобразование, как правило, было неидеальным, но, поскольку большая часть взаимодействий в те дни происходила внутри локальной сети, сетевые администраторы считали такой способ приемлемым.

Появление необходимости в межсетевых взаимодействиях вызвало острую потребность в стандартизации наборов сетевых протоколов. В конечном итоге закрытые протоколы уступили стандартизированным, таким как TCP, UDP и IP, что существенно улучшило способность сетей сообщаться между собой.

Корректная работа интернета, самой большой и важной сети, зависит от этих протоколов. Далее мы рассмотрим каждый из них.

Протокол управления передачей (TCP)

TCP (Transmission Control Protocol) — один из основных протоколов современного интернета. Открывая веб-страницу или отправляя электронное письмо, вы так или иначе используете его. Он находится на 4-м, транспортном уровне модели OSI и отвечает за надежную передачу сегментов данных между двумя узлами с проверкой ошибок. Пакет TCP состоит из 160-битного заголовка, который, помимо прочего, содержит номера портов отправителя и получателя, порядковый номер, номер подтверждения, управляющие флаги и контрольную сумму (рис. 1.6).

Рис. 1.6. TCP-заголовок

Функции и характеристики TCP

Для установления соединения между хостами протокол TCP использует сокеты или порты датаграмм. *Администрация адресного пространства интернета (Internet Assigned Numbers Authority, IANA)* закрепляет хорошо известные порты за определенными сервисами — например, порт 80 для HTTP (веб), а порт 25 — для SMTP (почта). Сервер в клиент-серверной модели обычно прослушивает один из этих стандартных портов в ожидании клиентских запросов. Операционная система управляет TCP-соединением с помощью сокета — локального представления конечной точки соединения.

Протокол действует подобно конечному автомату, определяя, когда принимать новые запросы на соединение, когда обслуживать действующий сеанс и когда освобождать ресурсы после закрытия соединения. Каждое TCP-соединение проходит через цепочку состояний, таких как `Listen`, `SYN-SENT`, `SYN-RECEIVED`, `ESTABLISHED`, `FIN-WAIT`, `CLOSE-WAIT`, `CLOSING`, `LAST-ACK`, `TIME-WAIT` и `CLOSED`.

TCP-сообщения и передача данных

Важнейшее отличие протокола TCP от его близкого «родственника» UDP: TCP передает данные упорядоченным и надежным образом. Благодаря гарантиям доставки TCP часто называют протоколом, ориентированным на соединения. Чтобы установить соединение, TCP сначала выполняет трехэтапную процедуру «рукопожатия», `SYN`, `SYN-ACK` и `ACK`, синхронизируя порядковый номер между отправителем и получателем.

Для наблюдения за передачей сегментов данных используются подтверждения. В конце одна из сторон посылает сообщение `FIN`, а другая подтверждает прием сообщением `ACK` и дополнительно посылает свое сообщение `FIN`. После чего инициатор завершения соединения посылает сообщения `ACK`, подтверждая получение сообщения `FIN`.

Те, кто занимался диагностикой TCP-соединений, знают, что это взаимодействие может оказаться довольно сложным. Но, к счастью, в большинстве случаев оно происходит незаметно, в фоновом режиме.

О протоколе TCP можно написать целый том; и на самом деле существует множество замечательных книг на эту тему.

Если вам недостаточно краткого обзора, вот отличный бесплатный источник, который даст вам более глубокое понимание этого материала: The TCP/IP Guide (http://www.tcpipguide.com/).

Протокол пользовательских датаграмм (UDP)

UDP (User Datagram Protocol) — еще один из основных протоколов интернета. Как и TCP, он работает на 4-м, транспортном уровне модели OSI и отвечает за обмен сегментами данных между прикладным и сетевым уровнями. В отличие от TCP, его заголовок занимает всего 64 бита, включая номера портов отправителя и получателя, длину и контрольную сумму. Легковесные заголовки делают этот протокол идеальным для ситуаций, когда требуется быстрая передача данных без организации сеанса связи между двумя хостами и без гарантий доставки. В эпоху быстрых интернет-соединений сложно представить, что во времена `X.21` и Frame Relay размер заголовков существенно влиял на скорость передачи данных.

Помимо скорости, UDP отличается от TCP отсутствием поддержки различных состояний, что тоже экономит вычислительные ресурсы на обеих сторонах соединения (рис. 1.7).

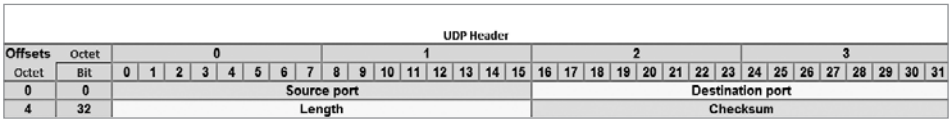


Рис. 1.7. UDP-заголовок

Вам, наверное, интересно, зачем UDP вообще используется в современных условиях. Учитывая отсутствие гарантий доставки данных, не лучше ли использовать во всех соединениях надежный протокол TCP с его проверкой ошибок? Но, например, при передаче потокового видео или звонках по Skype легковесные заголовки позволяют передавать датаграммы максимально быстро, что и требуется в подобных приложениях. Можно также вспомнить *DNS*-запросы, основанные на UDP: низкая задержка перевешивает потребность в точности.

Преобразование адреса, вводимого в браузере, в идентификатор, понятный компьютеру, желательно выполнять максимально быстро, потому что оно выполняется еще до того, как до вас дойдет первый бит вашего любимого веб-сайта.

И снова в одном разделе книги нельзя охватить все нюансы UDP, поэтому, если вас заинтересовала тема, исследуйте ее с помощью различных ресурсов.

Отличной отправной точкой для изучения UDP может послужить статья в Википедии: https://ru.wikipedia.org/wiki/UDP.

Межсетевой протокол (IP)

Сетевые инженеры, по их собственному признанию, работают на 3-м, сетевом уровне модели OSI. Протокол *IP (Internet Protocol)*, помимо прочего, устанавливает правила адресации и маршрутизации трафика между конечными узлами. Адресное пространство разделено на две части: сеть и хост. Какая часть сетевого адреса относится к сети, а какая к хосту, определяется маской подсети: часть, относящаяся к сети, обозначается единицами, а часть, относящаяся к хосту, — нулями. В IPv4 принята форма записи адресов через точки — например, `192.168.0.1`.

Маску подсети можно записать либо в формате с точками (`255.255.255.0`), либо с косой чертой, которая указывает, сколько битов отводится сетевой части (`/24`) (рис. 1.8).

IPv4 Header Format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

Рис. 1.8. Заголовок IPv4

Заголовок IPv6, следующего поколения протокола IP, имеет фиксированную часть и различные расширения (рис. 1.9).

Fixed Header format

Позиция	Октет	0								1								2								3							
Октет	Бит	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Версия*				*Класс трафика*								*Метка потока*																			
4	32	*Длина полезной нагрузки*																*Следующий заголовок*								*Число переходов*							
8	64	*IP-адрес отправителя*																															
12	96																																
16	128																																
20	160																																
24	192	*IP-адрес получателя*																															
28	224																																
32	256																																
36	288																																

Рис. 1.9. Заголовок IPv6

Поле Следующий заголовок в фиксированной части может указывать на расширенный заголовок с дополнительной информацией. Он также может определять протокол более высокого уровня, такой как TCP или UDP. Дополнительные заголовки могут содержать сведения о маршрутизации и фрагменте. Несмотря на огромное желание разработчиков протоколов перейти на IPv6, адресация в современном интернете в основном происходит с помощью IPv4, а IPv6 чаще используется для внутренней адресации в сетях поставщиков услуг.

Преобразование сетевых адресов (NAT) и сетевая безопасность

NAT (Network Address Translation) обычно используется для преобразования диапазона приватных IPv4-адресов в публично доступные. Но этот термин может также означать трансляцию IPv4 в IPv6; например, провайдер может использовать IPv6 во внутренней сети, но, когда пакеты покидают сеть, они должны преобразовываться в IPv4. Иногда по соображениям безопасности также применяется NAT6:6.

Обеспечение безопасности — это непрерывный процесс, охватывающий все аспекты сетевых технологий, включая автоматизацию и Python. В книге я хочу показать, как Python может помочь в управлении сетью; о безопасности речь пойдет в следующих главах, где мы обсудим использование Python для реализации списков доступа, поиска в журналах признаков несанкционированного вторжения и т. п. Мы также рассмотрим, как с помощью Python и других инструментов можно реализовать наблюдаемость сети — например, динамически создать графическое представление ее топологии на основе информации о сетевых устройствах.

Концепции IP-маршрутизации

IP-маршрутизация заключается в передаче пакетов между двумя конечными точками через промежуточные устройства в зависимости от IP-заголовков. Промежуточные устройства участвуют в передаче пакетов при любом взаимодействии в интернете. Как уже упоминалось, к ним относятся маршрутизаторы, коммутаторы, оптические устройства и другие компоненты, которые не выходят за пределы сетевого и транспортного уровней. Представьте, что вы направляетесь из Сан-Диего, штат Калифорния, в Сиэтл, штат Вашингтон. В этой аналогии Сан-Диего выступает в роли исходного IP-адреса, а Сиэтл — конечного. В своем путешествии вы будете проезжать через другие города: Лос-Анджелес, Сан-Франциско, Портленд; их можно считать промежуточными маршрутизаторами и коммутаторами между начальной и конечной точками.

Почему это важно? Данная книга в каком-то смысле посвящена администрированию и настройке этих промежуточных устройств. В век огромных дата-центров, занимающих несколько футбольных полей, эффективные, динамичные и экономные методы управления сетями становятся важным конкурентным преимуществом для многих компаний. В следующих главах мы подробно поговорим о том, какую роль в этом играет программирование на Python.

Итак, мы рассмотрели эталонные сетевые модели и наборы сетевых протоколов. Теперь можно переходить непосредственно к Python. Начнем с обзора этого языка.

Обзор языка Python

Эта книга писалась для того, чтобы упростить жизнь нам, сетевым инженерам. Но что такое Python и почему этот язык предпочитают многие инженеры

DevOps? Приведу цитату из краткого описания этого языка от Python Foundation (https://www.python.org/doc/essays/blurb/):

> *«Python — это интерпретируемый, объектно-ориентированный язык программирования высокого уровня с динамической семантикой. Высокоуровневая организация встроенных типов данных в сочетании с динамической типизацией и динамической привязкой делает этот язык очень привлекательным для быстрой разработки приложений, а также для использования в качестве языка сценариев или языка для связывания воедино существующих компонентов. Простой и легкий в освоении синтаксис Python делает упор на удобочитаемость и тем самым снижает затраты на обслуживание программ».*

Если вы новичок в программировании, то такие понятия, как «объектно-ориентированный» или «динамическая семантика», наверное, мало о чем вам говорят. Но такие характеристики, как «быстрая разработка приложений» и «простой, легкий в освоении синтаксис», как мне кажется, выглядят заманчиво. Python — интерпретируемый язык; это означает, что перед выполнением код почти (или вообще) не компилируется, благодаря чему время на написание, тестирование и редактирование программ на этом языке существенно сокращается. Если ваш сценарий завершается с ошибками, то для его отладки зачастую достаточно оператора `print`.

Использование интерпретатора также позволяет легко переносить программы на разные операционные системы, включая Windows и Linux. Программа, написанная для одной ОС, может использоваться в другой с минимальными изменениями кода (или вовсе без таковых).

Функции, модули и пакеты поощряют повторное использование кода за счет разбиения больших программ на простые фрагменты, пригодные для многократного использования. Объектно-ориентированная природа Python позволяет пойти дальше и группировать компоненты в объекты. На самом деле все файлы в Python являются модулями, которые можно повторно использовать или импортировать в других программах на том же языке. Это облегчает обмен кодом между инженерами и поощряет его повторное использование. Один из девизов Python: *батарейки — в комплекте*; это означает, что для выполнения распространенных задач достаточно самого языка и вам не нужно загружать никаких дополнительных пакетов. Чтобы код при этом не был слишком раздутым, вместе с интерпретатором Python устанавливается набор модулей, известных как стандартная библиотека. Для выполнения таких рутинных действий, как обработка регулярных выражений, вычисление математических функций

и декодирование JSON, достаточно добавить инструкцию `import`, и интерпретатор подключит необходимый код к вашей программе. Я бы назвал эту особенность Python одной из ключевых.

Наконец, для сетевых инженеров очень удобно, что разработка на Python может начаться с относительно небольшого сценария с несколькими строчками кода и вырасти в полноценную промышленную систему. Как многим из нас известно, сети часто развиваются естественным путем, без какого-то генерального плана. Язык, способный расти вместе с вашей сетью, бесценен. Вас может удивить, что язык, который нарекли языком сценариев, применяется многими передовыми компаниями в настоящих промышленных системах (список организаций, использующих Python: https://wiki.python.org/moin/OrganizationsUsingPython).

Если вам приходилось работать в условиях, когда требовалось часто переключаться между разными коммерческими платформами, такими как Cisco IOS и Juniper Junos, то вы знаете, насколько болезненным может быть переход с одного синтаксиса на другой в рамках одного проекта. Язык Python легко приспособить как для мелких, так и для крупных программ, поэтому переключение контекста некритично. В разных задачах, больших и маленьких, используется тот же код на Python!

В оставшейся части этой главы мы пройдемся по основам языка Python на случай, если вам нужно освежить свои знания. Если вы уже знакомы с ним, можете просто быстро просмотреть этот материал или сразу перейти к следующей главе.

Версии Python

Как, наверное, известно многим читателям, в последние несколько лет язык Python переходит от второй к третьей версии. Python 3 был выпущен в 2008 году, более десяти лет назад, и последняя версия имеет номер 3.10. К сожалению, Python 3 не имеет обратной совместимости с Python 2.

На момент подготовки третьего издания этой книги большая часть сообщества Python уже перешла на версию 3. На самом деле жизненный цикл Python 2 официально завершился 1 января 2020 года (https://pythonclock.org/). Последняя версия Python 2.x, 2.7, вышла больше восьми лет назад, в середине прошлого десятилетия. К счастью, обе эти версии могут сосуществовать на одном компьютере. Учитывая завершение жизненного цикла и уже состоявшееся прекращение поддержки, мы все должны перейти на Python 3. Вот пример вызова Python 2

и Python 3 на компьютере под управлением Ubuntu Linux (подробнее об использовании интерпретатора Python — в следующем разделе):

```
$ python2
Python 2.7.15+ (default, Jul 9 2019, 16:51:35)
[GCC 7.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
$ python3.7
Python 3.7.4 (default, Sep 2 2019, 20:47:34)
[GCC 7.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

С завершением жизненного цикла версии 2.7 большинство фреймворков теперь поддерживают Python 3. У Python 3 также есть множество прекрасных возможностей, таких как асинхронный ввод/вывод, который можно использовать для оптимизации кода. В примерах этой книги по умолчанию используется Python 3. Мы также стараемся указывать на различия между версиями 2 и 3, если таковые имеются.

Если какие-то конкретные библиотеки или фреймворки лучше подходят для Python 2 (например, Ansible; см. примечание ниже), это будет оговорено отдельно и в таких случаях будет использоваться Python 2. Но вообще Python 2 следует применять только там, где нет иного выхода, а в остальных случаях отдавать предпочтение Python 3.

На момент написания этих строк версии Ansible 2.8 и выше совместимы с Python 3. До версии 2.5 поддержка Python 3 считалась предварительной. Учитывая, что поддержка Python 3 была реализована не так давно, многие модули, разрабатываемые сообществом, все еще находятся в процессе перехода. Подробнее об Ansible и Python 3 читайте на странице https://docs.ansible.com/ansible/2.5/dev_guide/developing_python_3.html.

Операционные системы

Как уже упоминалось, Python — это кросс-платформенный язык. Написанные на нем программы могут работать в Windows, Mac и Linux. На самом деле для обеспечения совместимости со всеми платформами нужно принимать определенные меры — например, следует учитывать, что Windows в путях к файлам использует обратную косую черту, — и активировать виртуальную среду в разных системах. Эта книга предназначена для DevOps, системных и сетевых инженеров, поэтому предпочтительной платформой является Linux (особенно

в промышленных условиях). Код, который здесь приводится, проверен в Linux Ubuntu 18.04 LTS. Я также позабочусь о том, чтобы все примеры работали одинаково в Windows и macOS.

Подробнее о моей системе, если это вас интересует:

```
$ uname -a

Linux network-dev-2 4.18.0-25-generic #26~18.04.1-Ubuntu SMP Thu Jun 27
07:28:31 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
```

Выполнение программы на Python

Программы, написанные на Python, выполняются интерпретатором. Это означает, что интерпретатор получает ваш код, выполняет его и выводит результат. Существует несколько реализаций интерпретатора, созданных сообществом разработчиков Python, таких как IronPython и Jython. В этой книге мы используем самую популярную из них на сегодняшний день, CPython. Упоминая в этой книге Python, я имею в виду CPython (если не указано иное).

Для экспериментов с Python можно использовать интерактивную оболочку. Это особенно удобно, когда нужно быстро проверить какой-то фрагмент кода или концепцию без написания целой программы.

Обычно для этого достаточно ввести в терминале команду `python`:

```
$ python3.7
Python 3.7.4 (default, Sep 2 2019, 20:47:34)
[GCC 7.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
```

В Python 3 инструкция print является функцией, поэтому ее аргументы необходимо заключать в скобки. В Python 2 скобки можно опустить.

Интерактивный режим — одна из самых полезных возможностей Python. В интерактивной оболочке можно ввести любую корректную инструкцию или последовательность инструкций и сразу же получить результат. Я обычно поступаю так при исследовании какой-то незнакомой мне возможности или библиотеки. Интерактивный режим можно также применять для более сложных задач, для выяснения особенностей поведения структур данных, например разницы между изменяемым и неизменяемым типами. И правда, зачем откладывать такое удовольствие?

Если при вводе этой команды в Windows у вас не запустилась командная оболочка Python, это может означать, что соответствующая программа отсутствует в системных путях поиска. Программа установки Python в Windows предоставляет флажок для добавления интерпретатора в системный путь поиска; не забудьте установить его во время установки. Также можно добавить интерпретатор Python в путь поиска вручную, перейдя в настройки Environment Variables (Переменные среды).

Однако более традиционный способ выполнения программы на Python: сохранение ее в файл и последующий запуск с помощью интерпретатора. Так можно избежать многократного ввода одних и тех же выражений, как в случае с интерактивной оболочкой. Код на языке Python хранится в обычных текстовых файлах, как правило, с расширением `.py`.

В мире Unix в начало файла можно добавить так называемую строку *шебанг* (*shebang*; начинающуюся с пары символов `#!`), чтобы указать интерпретатор, который должен использоваться для выполнения файла. Символ `#` можно использовать для добавления комментариев, которые игнорируются интерпретатором. Например, файл `helloworld.py` содержит следующие инструкции:

```
# Это комментарий
print("hello world")
```

Его можно выполнить следующим образом:

```
$ python helloworld.py
hello world
$
```

Встроенные в Python типы данных

В Python реализована динамическая (утиная) типизация. Это означает, что язык пытается автоматически определить тип объекта при его объявлении. В интерпретатор Python встроено несколько стандартных типов:

- *числа:* `int`, `float`, `complex` и `bool` (подвид `int` с двумя возможными значениями: `True` и `False`);
- *последовательности:* `str`, `list`, `tuple` и `range`;
- *отображения:* `dict`;
- *множества:* `set` и `frozenset`;
- *None:* объект `null`.

Тип None

Тип `None` — это объект без значения. Он возвращается функциями, которые ничего явно не возвращают. Этот тип также используется в аргументах функций, для которых вызывающая сторона не предоставила никаких значений.

Числа

Числа в Python — это самые обычные числа. Если не считать булевы значения, все числовые типы (`int`, `long`, `float` и `complex`) имеют знак, то есть могут быть положительными и отрицательными. Тип `bool` является подтипом `int` и имеет только два возможных значения: 1 (`True`) и 0 (`False`). На практике для проверки булевых значений вместо 1 и 0 почти всегда используются `True` и `False`. Остальные числовые типы различаются по точности представления чисел; тип `int` в Python 3 не имеет максимального значения, а в Python 2 числа этого типа ограничены определенным диапазоном. Для чисел типа `float` используется представление двойной точности (64-разрядное).

Последовательности

Последовательности — это упорядоченные множества объектов с целочисленными неотрицательными индексами. В этом и следующих нескольких разделах для иллюстрации различных типов используется интерактивный интерпретатор.

Вы можете повторять те же примеры на своем компьютере.

Иногда люди удивляются тому, что тип `string` на самом деле является последовательностью. Но если задуматься, то строки — это и есть последовательность символов. Строки заключаются в одинарные, двойные или тройные кавычки.

Обратите внимание, что в следующих примерах виды открывающих и закрывающих кавычек должны совпадать и что тройные кавычки позволяют вводить многострочный текст:

```
>>> a = "networking is fun"
>>> b = 'DevOps is fun too'
>>> c = """what about coding?
... super fun!"""
>>>
```

Два других распространенных типа последовательностей: списки и кортежи. Список — это последовательность произвольных объектов. Их можно создавать,

заключая объекты в квадратные скобки. Как и строки, списки индексируются неотрицательными целыми числами, начиная с нуля. Значения элементов списка извлекаются по индексу:

```
>>> vendors = ["Cisco", "Arista", "Juniper"]
>>> vendors[0]
'Cisco'
>>> vendors[1]
'Arista'
>>> vendors[2]
'Juniper'
```

Кортежи похожи на списки, но для их создания используются круглые скобки. Значения их элементов тоже извлекаются по индексу, но, в отличие от списков, кортежи нельзя изменять после создания:

```
>>> datacenters = ("SJC1", "LAX1", "SFO1")
>>> datacenters[0]
'SJC1'
>>> datacenters[1]
'LAX1'
>>> datacenters[2]
'SFO1'
```

Некоторые операции являются общими для всех типов последовательностей. Например, извлечение элемента по индексу или создание среза:

```
>>> a
'networking is fun'
>>> a[1]
'e'
>>> vendors
['Cisco', 'Arista', 'Juniper']
>>> vendors[1]
'Arista'
>>> datacenters
('SJC1', 'LAX1', 'SFO1')
>>> datacenters[1]
'LAX1'
>>>
>>> a[0:2]
'ne'
>>> vendors[0:2]
['Cisco', 'Arista']
>>> datacenters[0:2]
('SJC1', 'LAX1')
>>>
```

Помните, что отсчет индексов начинается с 0. Поэтому индекс 1 на самом деле принадлежит второму элементу последовательности.

Существуют также стандартные функции, которые можно применять к последовательностям. Например, вот как можно узнать количество элементов, а также минимальное и максимальное значения:

```
>>> len(a)
17
>>> len(vendors)
3
>>> len(datacenters)
3
>>>
>>> b = [1, 2, 3, 4, 5]
>>> min(b)
1
>>> max(b)
5
```

Некоторые методы применимы только к строкам, что неудивительно. Стоит отметить, что эти методы не изменяют данные, с которыми они работают, и всегда возвращают новую строку. Если коротко, то изменяемые объекты, такие как списки и словари, можно модифицировать после их создания, а неизменяемые объекты, такие как строки, нельзя. Если вы хотите использовать результат изменения, вам необходимо принять возвращаемое значение и присвоить его другой переменной:

```
>>> a
'networking is fun'
>>> a.capitalize()
'Networking is fun'
>>> a.upper()
'NETWORKING IS FUN'
>>> a
'networking is fun'
>>> b = a.upper()
>>> b
'NETWORKING IS FUN'
>>> a.split()
['networking', 'is', 'fun']
>>> a
'networking is fun'
>>> b = a.split()
>>> b
['networking', 'is', 'fun']
>>>
```

Вот некоторые часто используемые методы для списков. Тип данных `list` — очень полезная структура, позволяющая объединять и последовательно перебирать несколько элементов. Например, можно составить список коммутаторов в дата-центре и применить к ним единый список доступа, перебирая их один за

другим. Поскольку значение списка можно изменять после создания (в отличие от кортежей), то мы можем расширять и сокращать списки по мере выполнения программы:

```
>>> routers = ['r1', 'r2', 'r3', 'r4', 'r5']
>>> routers.append('r6')
>>> routers
['r1', 'r2', 'r3', 'r4', 'r5', 'r6']
>>> routers.insert(2, 'r100')
>>> routers
['r1', 'r2', 'r100', 'r3', 'r4', 'r5', 'r6']
>>> routers.pop(1)
'r2'
>>> routers
['r1', 'r100', 'r3', 'r4', 'r5', 'r6']
```

Списки отлично подходят для хранения данных, однако следить за местоположением конкретных элементов в них может быть немного проблематично. В таких случаях удобнее использовать отображения.

Отображения

Отображения в Python реализованы в виде *словаря*. Словарь напоминает мне упрощенную базу данных, так как он содержит объекты, на которые можно ссылаться по ключу. В других языках программирования эти структуры часто называют *ассоциативными массивами* или *хеш-таблицами*. Если вы уже имели дело с чем-то подобным в других языках, то уже знаете, что это довольно мощный тип, позволяющий обращаться к объектам с помощью удобочитаемых ключей. Использование ключей вместо числовых индексов поможет бедолаге, которому приходится обслуживать или отлаживать чей-то код.

Таким бедолагой можете оказаться и вы, когда вам придется в два часа ночи искать проблему в собственном коде, написанном несколько месяцев назад. Элемент словаря может быть другим сложным объектом, таким как список. Вы уже знаете, что списки создаются с помощью квадратных скобок, а кортежи — круглых, поэтому на долю словарей остаются фигурные скобки:

```
>>> datacenter1 = {'spines': ['r1', 'r2', 'r3', 'r4']}
>>> datacenter1['leafs'] = ['l1', 'l2', 'l3', 'l4']
>>> datacenter1
{'leafs': ['l1', 'l2', 'l3', 'l4'], 'spines': ['r1',
'r2', 'r3', 'r4']}
>>> datacenter1['spines']
['r1', 'r2', 'r3', 'r4']
>>> datacenter1['leafs']
['l1', 'l2', 'l3', 'l4']
```

Словари в Python — одни из моих любимых контейнеров данных, которые я использую в своих сетевых сценариях. Существуют и другие контейнеры, которые могут вам пригодиться. Множества — один из них.

Множества

Множества используются для хранения неупорядоченных коллекций объектов. В отличие от списков и кортежей множества не поддерживают порядок следования элементов и числовую индексацию. Но одна особенность делает их полезными: элементы множества никогда не повторяются. Представьте, что у вас есть набор IP-адресов, которые нужно добавить в список доступа, но проблема в том, что многие элементы в этом наборе повторяются.

Теперь подумайте, сколько строк кода потребуется написать, чтобы путем последовательного циклического перебора этих IP-адресов удалить повторяющиеся значения. Встроенный тип `set` позволяет сделать то же самое одной строкой. Если честно, я не так часто применяю этот тип, но в ситуациях, когда он действительно необходим, я очень рад, что он существует. Множества можно сравнивать между собой с помощью методов `union`, `intersection` и `difference`:

```
>>> a = "hello"
# Используем встроенную функцию set() для преобразования строки в множество
>>> set(a)
{'h', 'l', 'o', 'e'}
>>> b = set([1, 1, 2, 2, 3, 3, 4, 4])
>>> b
{1, 2, 3, 4}
>>> b.add(5)
>>> b
{1, 2, 3, 4, 5}
>>> b.update(['a', 'a', 'b', 'b'])
>>> b
{1, 2, 3, 4, 5, 'b', 'a'}
>>> a = set([1, 2, 3, 4, 5])
>>> b = set([4, 5, 6, 7, 8])
>>> a.intersection(b)
{4, 5}
>>> a.union(b)
{1, 2, 3, 4, 5, 6, 7, 8}
>>> a.difference(b)
{1, 2, 3}
>>>
```

Итак, мы рассмотрели разные типы данных. Теперь сделаем обзор операторов языка Python.

Операторы в Python

В Python есть привычные *числовые операторы*, такие как +, – и т. д.; обратите внимание, что деление с усечением (//, известное также как *деление нацело*) усекает результат по десятичной запятой и возвращает целую его часть. Оператор деления с остатком (%) возвращает величину остатка от деления нацело:

```
>>> 1 + 2
3
>>> 2 - 1
1
>>> 1 * 5
5
>>> 5 / 1 # возвращает float
5.0
>>> 5 // 2 # // деление нацело
2
>>> 5 % 2 # деление с остатком
1
```

Есть также *операторы сравнения*. Обратите внимание, что для сравнения предназначен двойной знак равенства, а одинарный служит для присваивания значений:

```
>>> a = 1
>>> b = 2
>>> a == b
False
>>> a > b
False
>>> a < b
True
>>> a <= b
True
```

С помощью двух популярных *операторов членства* можно проверить присутствие объекта в последовательности:

```
>>> a = 'hello world'
>>> 'h' in a
True
>>> 'z' in a
False
>>> 'h' not in a
False
>>> 'z' not in a
True
```

Операторы в языке Python помогают эффективно выполнять простые операции. В следующем подразделе мы посмотрим, как эти операции можно повторять с помощью средств управления потоком выполнения.

Средства управления потоком выполнения в Python

Инструкции `if`, `else` и `elif` делают возможным условное выполнение кода. В отличие от некоторых других языков программирования в Python для структурирования блоков используются отступы. Условные инструкции имеют следующий формат:

```
if условие:
    делаем что-то
elif условие:
    делаем что-то, если условие выполняется
elif условие:
    делаем что-то, если условие выполняется
...
else:
    инструкция
```

Вот простой пример:

```
>>> a = 10
>>> if a > 1:
...     print("a is larger than 1")
... elif a < 1:
...     print("a is smaller than 1")
... else:
...     print("a is equal to 1")
...
a is larger than 1
>>>
```

Цикл `while` выполняется, пока условное выражение не вернет `False`, поэтому следите за этим выражением, чтобы цикл не стал бесконечным:

```
while условие:
    делаем что-то
```

```
>>> a = 10
>>> b = 1
>>> while b < a:
...     print(b)
...     b += 1
...
```

```
1
2
3
4
5
6
7
8
9
>>>
```

Цикл `for` работает с любыми объектами, которые поддерживают итерации; это означает, что он совместим со всеми встроенными типами последовательностей, такими как `list`, `tuple` и `string`. Буква `i` в следующем цикле обозначает переменную цикла — вы можете выбрать любое другое имя, имеющее смысл в контексте вашего кода:

```
for i in последовательность:
    делаем что-то

>>> a = [100, 200, 300, 400]
>>> for number in a:
...     print(number)
...
100
200
300
400
```

Мы рассмотрели типы данных, операторы и средства управления потоком выполнения в Python. Теперь все это можно объединить в многоразовые фрагменты кода, которые называют функциями.

Функции в Python

В большинстве случаев, когда приходится копировать и вставлять какие-то фрагменты кода, их лучше разбить на автономные блоки — функции. Это делает код модульным и более простым в облуживании, облегчая его повторное использование. Для определения функций в Python используется ключевое слово `def`, за которым следуют имя функции и ее параметры. Тело функции состоит из инструкций, которые следует выполнить. В конце можно вернуть значение вызывающей стороне; если этого не сделать, то по умолчанию функция вернет объект `None`:

```
def имя(параметр1, параметр2):
    инструкции
    return значение
```

В следующих главах вы увидите множество разных функций, поэтому здесь мы ограничимся простым примером. В листинге ниже используются позиционные параметры, то есть первый аргумент всегда передается функции в первом параметре. На параметры можно также ссылаться по именам и определять значения по умолчанию: `def subtract(a=10, b=5)`.

```
>>> def subtract(a, b):
...     c = a - b
...     return c
...
>>> result = subtract(10, 5)
>>> result
5
>>>
```

Функции отлично подходят для объединения задач. А можно ли объединить функции в еще более крупный блок многоразового кода? Конечно. Для этого в Python есть классы.

Классы в Python

Python — язык *объектно-ориентированного программирования (ООП)*. Объекты создаются с помощью ключевого слова `class`, и обычно их представляют как коллекции функций (методов), переменных и атрибутов (свойств). После определения класса можно создавать его экземпляры. Класс служит чертежом, или прообразом, последующих экземпляров.

Тема ООП выходит за рамки этой главы, поэтому приведу простой пример с определением объекта `router`:

```
>>> class router(object):
... def __init__(self, name, interface_number, vendor):
...     self.name = name
...     self.interface_number = interface_number
...     self.vendor = vendor
...
>>>
```

Определив этот класс, вы сможете создать любое число его экземпляров:

```
>>> r1 = router("SFO1-R1", 64, "Cisco")
>>> r1.name
'SFO1-R1'
>>> r1.interface_number
64
>>> r1.vendor
'Cisco'
```

```
>>>
>>> r2 = router("LAX-R2", 32, "Juniper")
>>> r2.name
'LAX-R2'
>>> r2.interface_number
32
>>> r2.vendor
'Juniper'
>>>
```

Конечно, это только первое знакомство с ООП и объектами в Python. В следующих главах вы увидите еще множество примеров.

Модули и пакеты в Python

Исходный файл на языке Python можно использовать в качестве модуля. Это изначально и есть модуль, и любые функции/классы, которые в нем определены, доступны для повторного использования. Чтобы подключить модуль, используйте ключевое слово `import`. При импортировании файла происходит следующее.

1. Для объектов, определенных в импортируемом файле, создается новое пространство имен.
2. Импортируемый модуль выполняется.
3. В пространстве имен вызывающей стороны создается имя, ссылающееся на импортированный модуль. Это имя совпадает с именем модуля.

Помните функцию `subtract()`, которую вы определили в интерактивной оболочке? Чтобы сделать ее доступной для повторного использования, ее можно поместить в файл `subtract.py`:

```
def subtract(a, b):
    c = a - b
    return c
```

В каталоге, где находится файл `subtract.py`, можно запустить интерпретатор Python и импортировать эту функцию:

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import subtract
>>> result = subtract.subtract(10, 5)
>>> result
5
```

Этот прием сработал, потому что по умолчанию Python ищет модули сначала в текущем каталоге. Помните стандартную библиотеку, упоминавшуюся выше? Как вы могли догадаться, это обычные файлы на языке Python, которые используются в качестве модулей.

Если модуль находится в другом каталоге, путь к нему можно вручную добавить в список путей поиска с помощью sys.path.

Пакеты позволяют объединять модули в коллекции. Это еще один уровень организации, обеспечивающий дополнительную защиту пространств имен и возможность повторного использования кода. Чтобы определить пакет, нужно создать каталог с именем, совпадающим с именем пакета, и поместить в него исходные файлы модулей.

Чтобы интерпретатор Python распознавал этот каталог как пакет, добавьте в него файл `__init__.py`. Его можно оставить пустым. Создадим каталог `math_stuff` и поместим в него файл `__init__.py` и модуль `subtract.py` из предыдущего примера:

```
echou@pythonicNeteng:~/Master_Python_Networking/Chapter1$ mkdir math_stuff
echou@pythonicNeteng:~/Master_Python_Networking/Chapter1$ touch math_stuff/
                          __init__.py
echou@pythonicNeteng:~/Master_Python_Networking/Chapter1$ tree
.
 ├── helloworld.py
 └── math_stuff
  ├── __init__.py
  └── subtract.py
1 directory, 3 files
echou@pythonicNeteng:~/Master_Python_Networking/Chapter1$
```

Теперь, чтобы обратиться к модулю, нужно указать имя пакета, а за ним через точку имя модуля. Например, `math_stuff.subtract`:

```
>>> from math_stuff.subtract import subtract
>>> result = subtract(10, 5)
>>> result
5
>>>
```

Как видите, модули и пакеты отлично подходят для организации больших коллекций исходных файлов и существенно упрощают повторное использование кода в Python.

Резюме

В этой главе мы рассмотрели модель OSI и наборы сетевых протоколов, такие как TCP, UDP и IP. Они действуют как уровни, определяющие порядок адресации и согласования соединения между двумя хостами. Эти протоколы разрабатывались с учетом возможности расширения, благодаря чему они почти не изменились с момента их создания. Это неординарное достижение, учитывая взрывообразное развитие интернета.

Мы также вкратце рассмотрели язык Python, в том числе встроенные типы данных, операторы, средства управления потоком выполнения, функции, классы, модули и пакеты. Python — мощный язык, готовый к использованию в промышленном окружении, который легко читается. Благодаря этому Python идеально подходит для автоматизации сетей. Сетевые инженеры могут начать с простых сценариев и постепенно переходить к использованию более продвинутых особенностей этого языка.

В главе 2 мы начнем обсуждать приемы использования Python для взаимодействия с сетевым оборудованием.

2

Низкоуровневое взаимодействие с сетевыми устройствами

В главе 1 мы рассмотрели теоретические основы сетевых протоколов и их характеристики, а также познакомились в общих чертах с языком Python. В этой главе мы погрузимся в тему программного управления сетевыми устройствами. В частности, рассмотрим разные способы применения Python для взаимодействия с сетевыми маршрутизаторами и коммутаторами прошлого поколения.

Что я имею в виду под прошлым поколением? В наши дни сложно представить какое-либо новое сетевое устройство, с которым нельзя было бы общаться посредством *прикладного программного интерфейса (Application Program Interface, API)*, однако раньше API в таких устройствах отсутствовал. Для администрирования использовался *интерфейс командной строки (Command Line Interface, CLI)* в виде консольных программ, рассчитанных на ручное управление. Инженер должен был интерпретировать данные, полученные из устройства, и принять соответствующее решение. Понятно, что с увеличением количества сетевых устройств и усложнением сетей такое администрирование становилось все более проблемным.

Для Python написано несколько отличных библиотек и фреймворков, помогающих решать такие задачи. Среди них — Pexpect, Paramiko, Netmiko, NAPALM и Nornir. У некоторых из этих проектов есть общий код, зависимости и разработчики. Например, библиотеку Netmiko создал Кирк Байерс в 2014 году на

основе другой библиотеки, Paramiko SSH. В 2017 году Кирк объединился с Дэвидом Барросо из проекта NAPALM и другими программистами для создания фреймворка Nornir, обеспечивающего средства сетевой автоматизации на чистом Python.

Эти библиотеки по большей части можно использовать совместно. Например, Ansible (см. главы 4 и 5) применяет в своих сетевых модулях обе библиотеки, Paramiko и Ansible-NAPALM.

На сегодняшний день существует так много библиотек, что рассказать обо всех на страницах одной книги просто невозможно. В этой главе мы сначала рассмотрим Pexpect и затем перейдем к примерам использования библиотеки Paramiko. Уяснив основы и принцип работы последней, вы без труда сможете освоить другие проекты, такие как Netmiko и NAPALM.

Эта глава охватывает такие темы, как:

- трудности работы с CLI;
- создание виртуальной лаборатории;
- библиотека Python Pexpect;
- библиотека Python Paramiko;
- примеры использования обеих библиотек;
- недостатки Pexpect и Paramiko.

Я уже упоминал о недостатках управления сетевыми устройствами с помощью интерфейса командной строки. Этот подход показал себя неэффективным в работе с сетями среднего размера. В этой главе представлены библиотеки для Python, которые помогут обойти это ограничение. Но сначала давайте подробнее обсудим трудности работы с CLI.

Трудности работы с CLI

В 2014 году на выставке Interop в Лас-Вегасе генеральный директор Big Switch Networks Дуглас Мюррей проиллюстрировал изменения, произошедшие в *сетевых технологиях дата-центров* за предыдущие 20 лет, с 1993 по 2013 год, слайдом, приведенным на рис. 2.1.

Он показал очевидное: за два десятилетия в управлении сетевыми устройствами изменилось не так уж много. Возможно, это было немного несправедливо по отношению к производителям оборудования того времени, но суть вывода мало

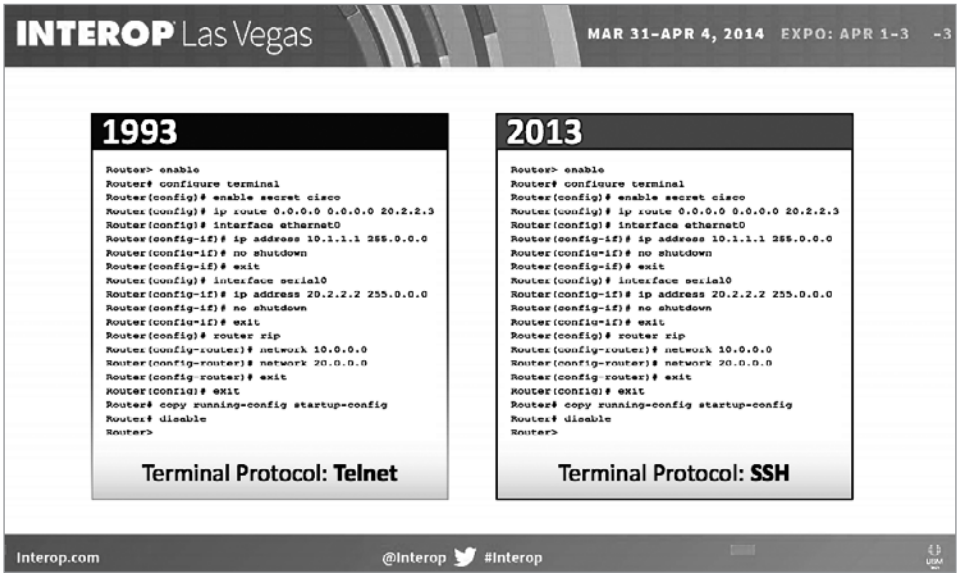


Рис. 2.1. Изменения в сетевых технологиях дата-центров (источник: https://www.bigswitch.com/sites/default/files/presentations/murraydouglasstartuphotseatpanel.pdf)

у кого вызывала возражения. По мнению автора презентации, в управлении маршрутизаторами и коммутаторами за предыдущие 20 лет изменился только протокол: вместо Telnet стали использовать более безопасный SSH.

Именно в это время, ориентировочно в 2014 году, мы стали наблюдать в нашей индустрии растущий консенсус о явной необходимости перехода от утилит командной строки, управляемых вручную, к программной автоматизации на основе API. Конечно, нам по-прежнему приходится обращаться к устройствам напрямую при проектировании, создании прототипов и начальном развертывании топологии. Но после этого этапа управление сетью обычно сводится к надежному и согласованному внесению изменений во все сетевые устройства. Это позволяет избежать ошибок и сделать процесс воспроизводимым, благодаря чему можно не отвлекать и не утомлять инженеров. И это одна из тех задач, которые логично передать компьютеру и нашему любимому языку Python.

Но вернемся к слайду и представим, что сетевыми устройствами можно управлять лишь из командной строки. Основная трудность возникает при имитации взаимодействия администратора с маршрутизатором с помощью компьютерной программы. В командной строке маршрутизатор будет выводить какую-то информацию и ожидать от нас ручного ввода неких команд, основанных на нашей интерпретации полученного вывода. Например, в устройстве Cisco под

управлением *IOS (Internetwork Operating System — межсетевая операционная система)* нужно ввести команду `enable`, чтобы войти в привилегированный режим; затем, получив приглашение к вводу с символом `#`, вы должны ввести `configure terminal`, чтобы перейти в режим конфигурации. Далее можно перейти в режим конфигурации интерфейса или протокола маршрутизации. Это полная противоположность программного подхода. Когда компьютер выполняет какую-то отдельную задачу (скажем, назначает IP-адрес интерфейсу), он структурирует всю необходимую информацию и передает ее маршрутизатору целиком, ожидая от того четкого ответа об успешном или неуспешном выполнении задачи.

Решение, реализованное в Pexpect и в Paramiko, основано на запуске интерактивного дочернего процесса и наблюдении за тем, как этот процесс общается с целевым устройством. Проверив полученное значение, родительский процесс предпринимает последующие действия (если таковые требуются).

Уверен, вам уже не терпится перейти к библиотекам для Python, но сначала давайте организуем нашу сетевую лабораторию, чтобы у нас было где проверить наш код. Есть несколько вариантов.

Создание виртуальной лаборатории

Прежде чем погрузиться в библиотеки и фреймворки для Python, давайте сначала организуем небольшую учебную лабораторию. Мастером, как известно, можно стать, только практикуясь. Нам понадобится изолированная среда, которая стерпит все наши ошибки, а также позволит нам опробовать новые подходы и повторить уже пройденное, чтобы закрепить в памяти понятия. Установить Python и пакеты для управления хостом легко, но что насчет маршрутизаторов и коммутаторов, которые мы хотим имитировать?

Сетевую лабораторию можно создать на основе либо физических, либо виртуальных устройств. Рассмотрим преимущества и недостатки этих вариантов.

Физические устройства

Физические сетевые устройства можно увидеть и пощупать. Счастливчики могут даже воспроизвести в лаборатории точную структуру своей промышленной среды.

- **Преимущества.** Плавный переход от лаборатории к промышленным условиям. Вашим руководителям и коллегам-инженерам будет легче разобраться с топологией; при необходимости они всегда могут взглянуть на устройства и выполнить с ними какие-то действия. Если коротко, то привычность физических устройств обеспечивает чрезвычайно высокий уровень удобства.
- **Недостатки.** Использование устройств сугубо в экспериментальных целях — относительно дорогое удовольствие. Создание лаборатории на их основе потребует времени, и поменять что-то потом будет непросто.

Виртуальные устройства

Это эмуляторы, или имитаторы, настоящих сетевых устройств. Такие устройства предоставляются либо производителями, либо сообществом открытого ПО.

- **Преимущества.** Виртуальные устройства легче настраивать. Они относительно дешевые и позволяют быстро вносить изменения в топологию сети.
- **Недостатки.** Это обычно упрощенные версии физических аналогов, иногда с урезанными функциями.

Конечно, выбор между виртуальной и физической лабораторией следует делать, исходя из своих возможностей и задач, пытаясь найти разумный компромисс между стоимостью, простотой реализации и риском расхождений между тестовой и промышленной средами. В некоторых проектах, над которыми я работал, виртуальная лаборатория использовалась для создания начального прототипа, а ближе к завершающей стадии проектирования мы переходили на физические устройства.

По моему мнению, при обучении следует выбирать виртуальную лабораторию, учитывая, что все больше производителей начинают выпускать виртуальное оборудование. Нехватка функциональности в виртуальных устройствах относительно небольшая и подробно задокументирована, особенно если эти устройства предоставляются самим производителем. Они недорогие по сравнению с физическими альтернативами. К тому же это зачастую обычные программы, и поэтому с ними требуется меньше времени на построение лаборатории.

Показывать примеры я буду с помощью и физических, и виртуальных устройств, отдавая предпочтение последним. Разница должна быть незаметной. Если при

использовании виртуальных и физических устройств в достижении стоящих перед нами целей будет какая-то разница, я постараюсь ее объяснить.

Как вы вскоре увидите, в примерах в этой книге я пытаюсь максимально упростить сетевую топологию, но так, чтобы она позволяла показать необходимые идеи. Каждая виртуальная сеть обычно состоит всего из нескольких узлов, и мы часто будем использовать одни и те же сети в разных упражнениях.

Благодаря этому читатели предыдущих изданий могли использовать многие популярные лаборатории с виртуальными сетями, такие как GNS3, Eve-NG и другие виртуальные машины.

Для примеров я выбирал виртуальные машины от разных поставщиков, таких как Juniper и Arista. На момент написания этих строк Arista vEOS можно загрузить бесплатно на сайте Arista. Платформа Juniper JunOS Olive, которую использую я, не поддерживается официально, но Juniper предлагает для vMX бесплатную пробную лицензию, которую можно заменять. Я также применяю программу сетевой лаборатории от Cisco под названием *Virtual Internet Routing Lab*. Она платная, но в следующих разделах я объясню, почему я считаю ее хорошим вариантом для лабораторий с виртуальными сетями.

Хочу подчеркнуть, что не обязательно покупать программу VIRL; вы можете использовать бесплатные альтернативы. Но для выполнения упражнений из этой книги я настоятельно рекомендую обзавестись каким-то лабораторным оборудованием.

Cisco VIRL

Вспоминаю, как я начинал готовиться к экзамену для получения *сертификата специалиста по межсетевым технологиям Cisco (Cisco Certified Internetwork Expert, CCIE)*. Мне пришлось купить подержанное оборудование Cisco на eBay. Даже со скидкой маршрутизатор и коммутатор стоили сотни долларов, поэтому, чтобы сэкономить, я купил очень старые маршрутизаторы, произведенные в 1980-х годах (если хотите посмеяться, можете ввести в поисковой системе Cisco AGS). У них были очень скромные возможности и производительность даже по стандартам лаборатории. После их включения я имел занимательную беседу с членами моей семьи (устройства сильно шумели), а монтаж этого оборудования был не из приятных. Оно было тяжелым и громоздким; подключение всех кабелей требовало много усилий, а для имитации разрыва соединения мне нужно было буквально вытаскивать кабель из разъема.

Несколькими годами позже появился эмулятор Dynamips, и я не мог нарадоваться тому, насколько просто в нем было создавать различные сетевые конфигурации. Это было особенно важно при изучении новых концепций. Загрузив образы IOS на сайте Cisco и тщательно сконфигурировав файлы топологии, я мог с легкостью создавать виртуальные сети для проверки своих навыков. У меня была целая папка с сетевыми топологиями, готовыми конфигурациями и версиями образов для разных ситуаций. А благодаря добавлению клиентской части GNS3 моя лаборатория приобретала прекрасный графический интерфейс. С GNS3 можно было несколькими щелчками кнопкой мыши создавать соединения и устройства и даже распечатать топологию сети для начальства или клиента прямо с панели проектирования.

Единственный недостаток: этот инструмент не получил официальной поддержки производителя, то есть Cisco. Из-за этого он считался недостаточно надежным.

В 2015 году сообщество Cisco решило исправить ситуацию и выпустило Cisco VIRL. Я предпочитаю использовать эту платформу для разработки и отладки кода на языке Python как для примеров в этой книге, так и в своей работе.

По состоянию на 14 ноября 2019 года лицензия для личного использования с поддержкой 20 узлов стоила всего $199,99 в год.

Даже учитывая денежные затраты, платформа VIRL, как мне кажется, имеет несколько преимуществ по сравнению с альтернативами.

- **Простота в использовании.** Как уже упоминалось, все образы для IOSv, IOS-XRv, CSR1000v, NX-OSv и ASAv собраны в едином архиве, доступном для загрузки.
- **Официальный статус (почти).** Поддержка предоставляется сообществом, но этот инструмент широко применяется в самой компании Cisco. Благодаря его популярности ошибки в нем быстро исправляются, новые возможности тщательно документируются, а между его пользователями происходит активный обмен полезными знаниями.
- **Возможность миграции в облако.** На случай, если вашей виртуальной среде перестанет хватать аппаратных ресурсов, проект предлагает логичный способ миграции в облако, такое как Cisco dCloud (https://dcloud.cisco.com/) и Cisco DevNet (https://developer.cisco.com/). Это важная возможность, о которой иногда забывают.
- **Имитация сетевого соединения и управляющего уровня.** Этот инструмент умеет имитировать такие характеристики физических сетей, как

задержки, флуктуации и потеря пакетов в каждом отдельном соединении. Существует также генератор трафика управляющего уровня для внедрения внешних маршрутов.

- **Другое.** Среди прочих интересных возможностей можно выделить проектирование топологий и управление симуляцией с помощью VM Maestro, автоматическую генерацию конфигурационных файлов с использованием AutoNetKit и управление пользовательскими рабочими пространствами для разделяемых серверов. Существуют также открытые проекты, такие как virlutils (https://github.com/CiscoDevNet/virlutils), в рамках которых сообщество активно работает над улучшением инструментов.

Мы не используем все возможности VIRL в этой книге. Это относительно новый инструмент, и если вы все же выберете именно его, я бы хотел дать несколько советов относительно его конфигурации.

И снова хочу подчеркнуть, что для выполнения упражнений из этой книги вам потребуется лаборатория. Но это может быть не Cisco VIRL, а что-то другое. Примеры кода, которые вы здесь найдете, должны работать на любом экспериментальном устройстве — главное, чтобы оно имело подходящий тип и версию программного обеспечения.

Советы относительно VIRL

На веб-сайте Cisco DevNet вы найдете обилие справочных руководств, инструкций по подготовке лаборатории и документации. Также хочу отметить, что у сообщества пользователей VIRL можно получить быстрые и точные советы. Я не стану повторять информацию, которую можно найти в этих двух местах; однако некоторые готовые конфигурации будут использоваться в нашей лаборатории.

Моя лаборатория VIRL содержит два виртуальных интерфейса Ethernet для соединений. Первый предназначен для *преобразования сетевых адресов (Network Address Translation, NAT)* на соединении хоста с интернетом, а второй служит интерфейсом локального управления (в следующем примере он называется VMnet2). Для запуска кода на языке Python я использую отдельную виртуальную машину с похожей сетевой конфигурацией: первый и основной порт Ethernet используется для подключения к интернету, а второй — к интерфейсу VMnet2 и предназначен для управления лабораторными сетевыми устройствами (рис. 2.2).

Рис. 2.2. Первый адаптер Ethernet VIRL заменен на NAT

VMnet2 — это пользовательская сеть, созданная для соединения хоста под управлением Ubuntu с виртуальной машиной VIRL (рис. 2.3).

1. На вкладке Topology (Топология) в списке Management Network (Управляющая сеть) я выбрал пункт Shared flat network (Разделяемая плоская сеть), чтобы управлять виртуальными маршрутизаторами из сети VMnet2 (рис. 2.4).
2. На вкладке Node (Узел) нужно задать статический управляющий IP-адрес. Я предпочитаю статические IP-адреса вместо динамических, которые назначаются программным обеспечением. Это делает доступ к сети более предсказуемым (рис. 2.5).

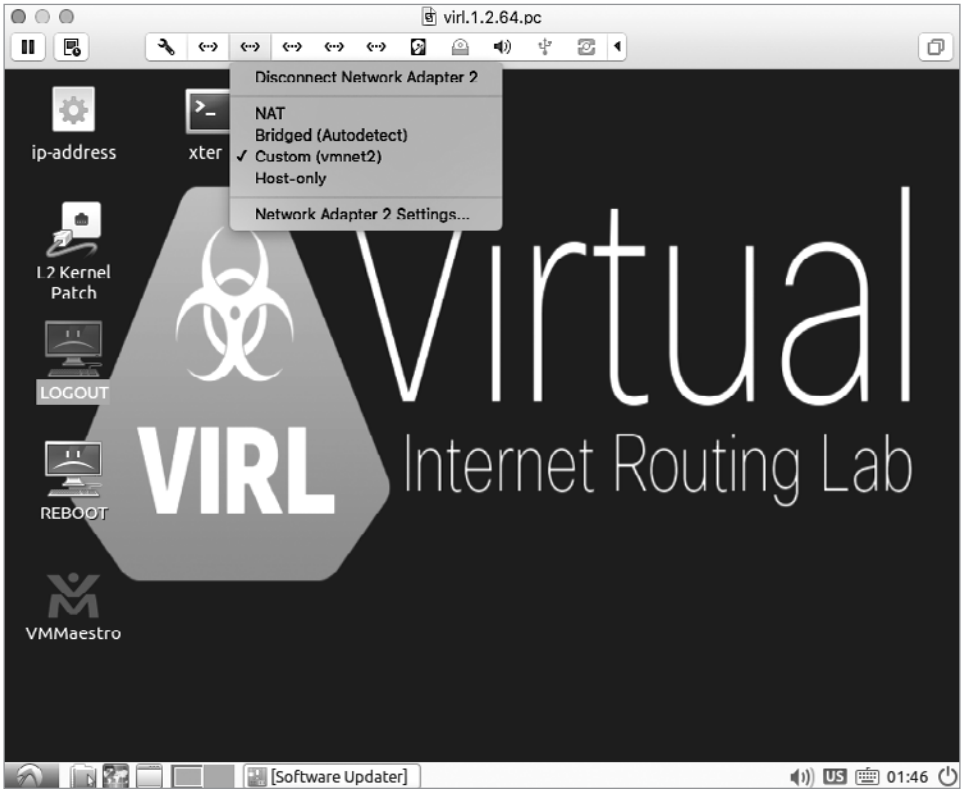


Рис. 2.3. Второй адаптер Ethernet VIRL соединен с VMnet2

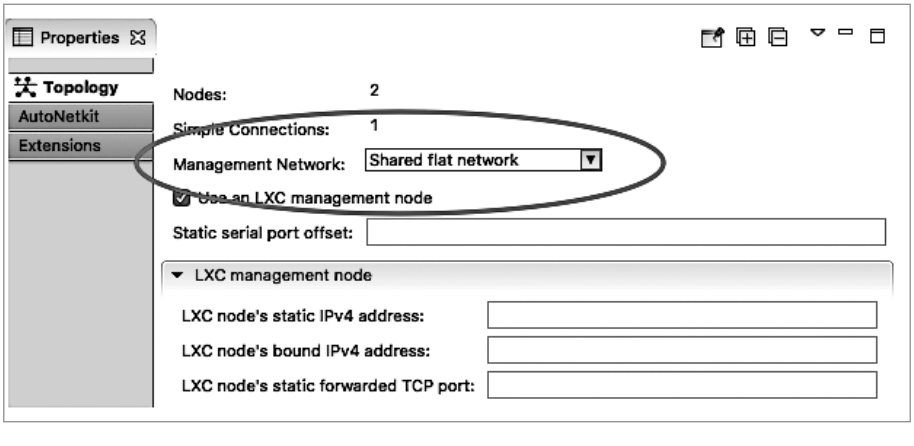


Рис. 2.4. Используйте плоскую разделяемую сеть в качестве управляющей сети VIRL

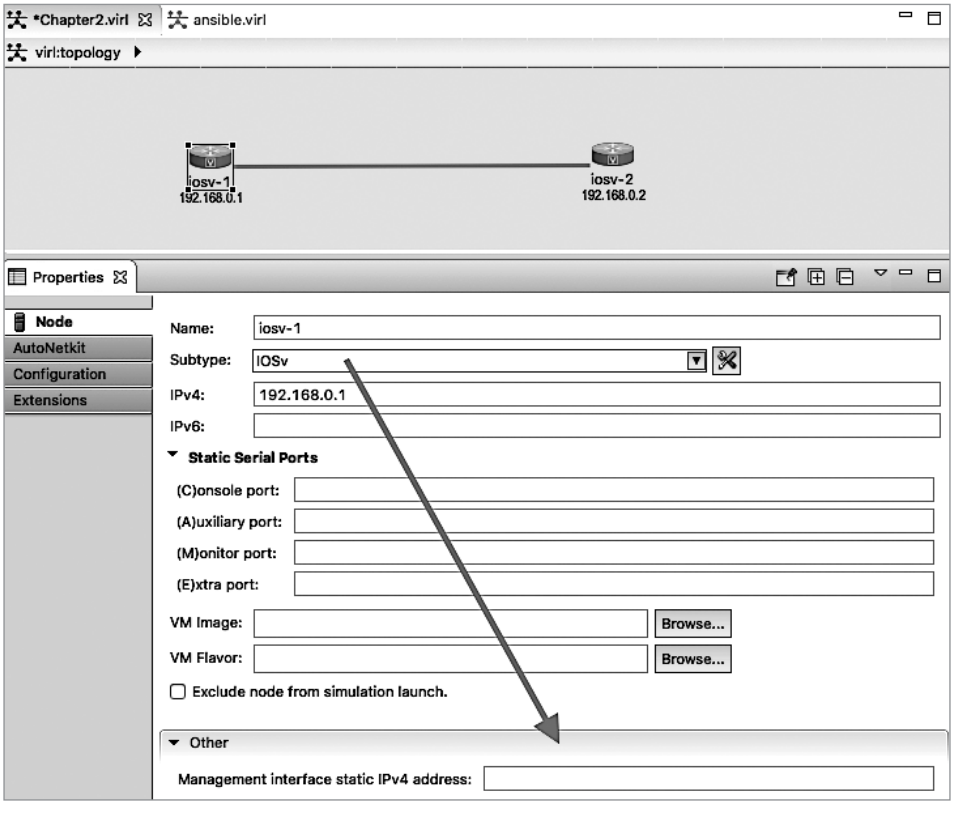


Рис. 2.5. IOSv1 со статическим управляющим IP-адресом

Топология лаборатории VIRL предоставляется вместе с примерами кода для каждой главы.

Cisco DevNet и dCloud

У Cisco есть два других замечательных бесплатных (на момент написания этой книги) инструмента для экспериментов с оборудованием этого производителя. Оба требуют наличия учетной записи *Cisco Connection Online (CCO)*. Это по-настоящему хорошие продукты, особенно за свою цену (равную нулю!).

Первый инструмент, изолированная среда Cisco DevNet (https://developer.cisco.com/), включает учебные руководства, полноценную документацию, удаленную лабораторию и многое другое. Некоторые лаборатории всегда доступны, а другие необходимо резервировать. Доступность лаборатории зависит от ее востре-

бованности. Это отличная альтернатива для тех, у кого нет собственного оборудования для экспериментов. Вы непременно должны воспользоваться этим продуктом независимо от того, есть у вас локальный хост с VIRL или нет (рис. 2.6).

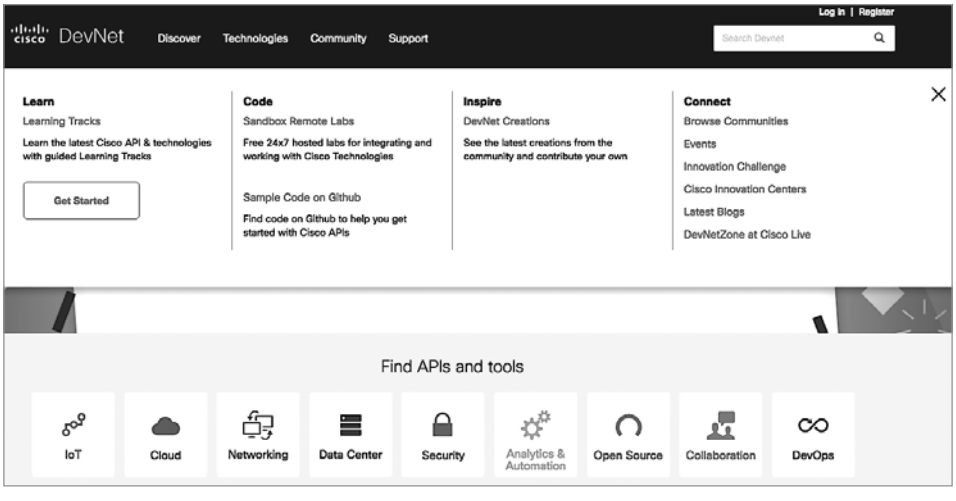


Рис. 2.6. Cisco DevNet

С момента своего создания сайт Cisco DevNet стал фактически каталогом любых материалов, касающихся программирования и автоматизации сетей в Cisco. Например, в июне 2019 года компания Cisco анонсировала на DevNet много новых курсов сертификации, https://developer.cisco.com/certification/.

Еще одна бесплатная онлайн-лаборатория для Cisco: https://dcloud.cisco.com/. dCloud можно считать копией VIRL, запущенной на чужих серверах и не требующей администрирования или оплаты соответствующих ресурсов. Похоже, что компания Cisco позиционирует dCloud и как самостоятельный продукт, и как расширение к VIRL. Например, dCloud можно применять для расширения локальной лаборатории, если потребуется запустить дополнительные экземпляры IOS-XR или NX-OS.

Это относительно новый инструмент, но он определенно заслуживает внимания (рис. 2.7).

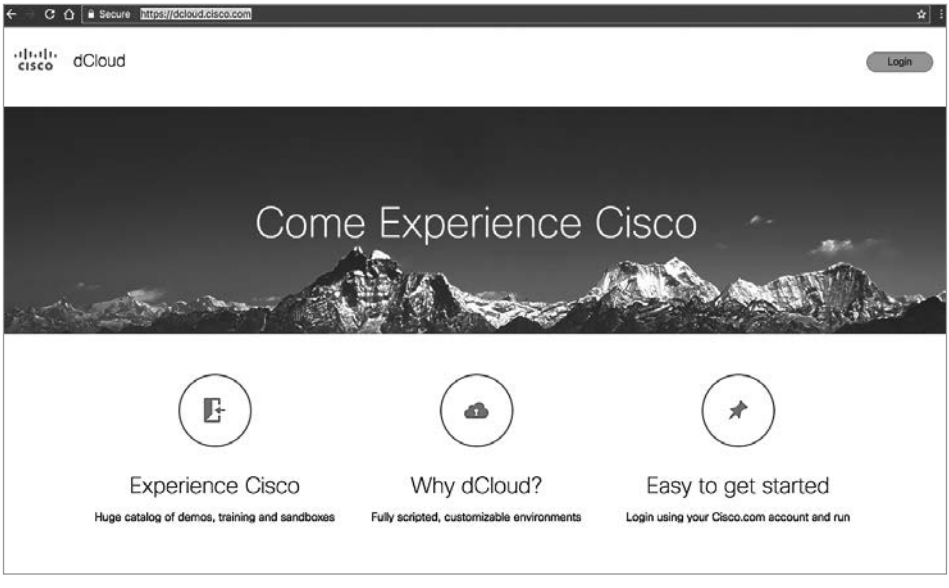


Рис. 2.7. Cisco dCloud

GNS3

Есть еще несколько виртуальных лабораторий, которые я использую в других проектах. Одна из них — GNS3 (рис. 2.8).

Как уже говорилось, GNS3 — это среда, которую многие используют для подготовки к сертификационным тестам и для практических занятий. Когда-то она была всего лишь клиентской частью для Dynamips, но со временем превратилась в настоящий коммерческий продукт. Один из недостатков таких инструментов, как VIRL, DevNet и dCloud: они поддерживают только технологии компании Cisco. И хотя лабораторные устройства могут взаимодействовать с внешним миром, в том числе и с оборудованием других производителей, это требует выполнения не совсем очевидных шагов. Среда GNS3 не привязана ни к какому конкретному поставщику и позволяет использовать в лаборатории виртуализованную платформу с поддержкой многих производителей. Обычно это подразумевает клонирование образа сетевого устройства (такого как Arista vEOS) или его запуск непосредственно под управлением сторонних гипервизоров (таких как эмулятор Juniper Olive). GNS3 подходит для случаев, когда в одной лаборатории нужно совмещать технологии разных поставщиков.

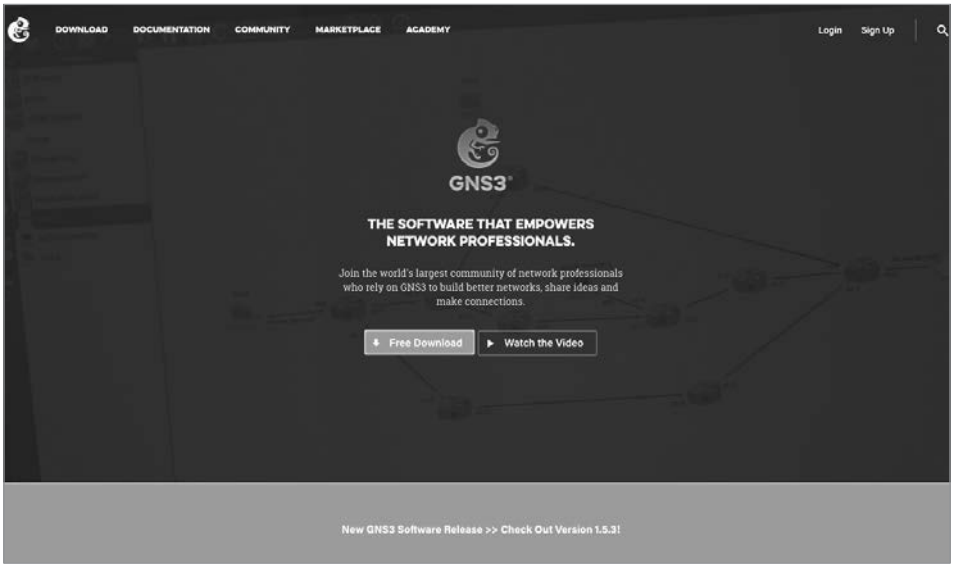


Рис. 2.8. Веб-сайт GNS3

Еще одна среда эмуляции сетей с поддержкой разных поставщиков, которая получила множество хвалебных отзывов, — *Eve-NG (Emulated Virtual Environment Next Generation — эмулируемая виртуальная среда следующего поколения)*: http://www.eve-ng.net/. Лично у меня почти нет опыта работы с этим инструментом, но многие мои друзья и коллеги используют его в своих сетевых лабораториях.

Существуют также виртуализованные платформы: Arista vEOS (https://eos.arista.com/tag/veos/), Juniper vMX (https://www.juniper.net/ru/ru/products/routers/mx-series/vmx-virtual-router-software.html) и vSRX (https://www.juniper.net/ru/ru/products/security/srx-series/vsrx-virtual-firewall.html), которые можно применять как самостоятельные виртуальные устройства при тестировании. Это отличные вспомогательные средства для проверки специфических возможностей платформы; например, вы можете узнать, чем различаются версии API. Многие из этих продуктов для большей доступности предоставляются в качестве платных услуг в магазинах облачных провайдеров. Они имеют те же возможности, что и их физические аналоги.

Итак, мы подготовили сетевую лабораторию. Теперь можно поэкспериментировать с библиотеками для Python, которые могут помочь с администрированием и автоматизацией. Начнем с библиотеки Pexpect.

Библиотека Python Pexpect

Pexpect — это модуль, написанный на чистом Python. Он предназначен для создания дочерних процессов, управления ими и выполнения действий на основе ожидаемых закономерностей в их выводе. Модуль Pexpect действует подобно библиотеке Expect Дона Лайбса. Он позволяет вашему сценарию запускать дочерние программы и управлять ими так, будто команды вручную вводит человек.

Документация Pexpect доступна по адресу https://pexpect.readthedocs.io/en/stable/index.html.

Подобно оригинальному модулю `Expect`, который Дон Лайбс написал на языке *Tcl*, Pexpect запускает или создает новый процесс и управляет им, контролируя взаимодействия. Модуль Expect изначально разрабатывался для автоматизации таких интерактивных процессов, как FTP, Telnet и rlogin, но позже был расширен для выполнения сетевой автоматизации. В отличие от своего прообраза, Pexpect полностью написан на Python и не требует наличия Tcl или компиляции расширений на языке C. Это позволяет использовать в нашем коде знакомый нам синтаксис Python и его богатую стандартную библиотеку.

Виртуальная среда Python

Давайте начнем с настройки виртуальной среды Python, что позволит нам использовать разные версии пакетов в разных проектах. Для этого мы будем создавать виртуальные изолированные окружения Python и устанавливать в них свои пакеты. При этом не нужно беспокоиться о том, что мы повлияем на совместимость с пакетами, установленными глобально или в других виртуальных средах. Для начала установим утилиту `pip`:

```
$ sudo apt update
$ sudo apt install python3-pip
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $
(venv) $ which python
/home/echou/venv/bin/python
(venv) $ deactivate
```

Здесь мы используем пакет `venv` из стандартной библиотеки Python 3, создаем каталог для нашей среды и затем активируем ее. Когда виртуальная среда активируется, вы увидите метку `(venv)` перед именем хоста, показывающую, что

вы находитесь в виртуальной среде. Чтобы покинуть виртуальную среду, выполните команду `deactivate`. Больше по этой теме читайте на странице https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/#installing-virtualenv.

В Python 2 виртуальная среда устанавливается и используется немного по-другому. В интернете можно найти ряд аналогичных руководств для этой версии Python.

Установка Pexpect

Установить Pexpect просто:

```
(venv) $ pip install pexpect
```

Если вы устанавливаете пакеты для Python в глобальной среде, вам понадобятся привилегии суперпользователя root, например: sudo pip install pexpect.

Проверим доступность пакетов после установки; для этого запустите интерактивную оболочку Python из виртуальной среды:

```
(venv) $ python
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pexpect
>>> dir(pexpect)
['EOF', 'ExceptionPexpect', 'Expecter', 'PY3', 'TIMEOUT', '__all__',
'__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__path__', '__revision__', '__spec__', '__
version__', 'exceptions', 'expect', 'is_executable_file', 'pty_spawn',
'run', 'runu', 'searcher_re', 'searcher_string', 'spawn', 'spawnbase',
'spawnu', 'split_command_line', 'sys', 'utils', 'which']
```

Краткий обзор Pexpect

Нашей первой лабораторной работой будет создание простой сети с двумя устройствами IOSv, соединенными напрямую (рис. 2.9).

У каждого устройства есть локальный адрес из диапазона `192.168.0.x/24` и управляющий IP-адрес из диапазона `172.16.1.x/24`. Файл топологии VIRL есть в архиве с примерами для этой книги, а также в репозитории GitHub (https://

github.com/PacktPublishing/Mastering-Python-Networking-Third-Edition). Вы можете импортировать его в свою среду VIRL. Если у вас нет VIRL, найдите необходимую информацию, открыв файл топологии в текстовом редакторе. Это обычный XML-файл с данными о каждом узле внутри элементов node (рис. 2.10).

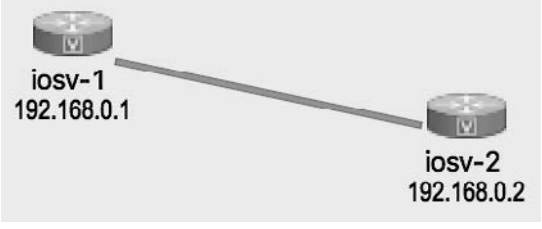


Рис. 2.9. Топология сетевой лаборатории

Рис. 2.10. Информация об узлах

Итак, мы подготовили устройства. Теперь давайте посмотрим, как можно обратиться к маршрутизатору с помощью telnet:

```
(venv) $ telnet 172.16.1.20
Trying 172.16.1.20...
Connected to 172.16.1.20.
Escape character is '^]'.
```

```
<опущено>
User Access Verification

Username: cisco
Password:
```

Я автоматически сгенерировал начальную конфигурацию маршрутизаторов с помощью VIRL AutoNetKit, в которой по умолчанию используются имя пользователя `cisco` и пароль `cisco`. Обратите внимание, что пользователь уже в привилегированном режиме, так как в конфигурации ему назначены соответствующие привилегии:

```
iosv-1#sh run | i cisco
enable password cisco
username cisco privilege 15 secret 5 $1$SXY7$Hk6z8OmtloIzFpyw6as2G.
 password cisco
 password cisco
```

В автоматически сгенерированной конфигурации также открыт доступ к `vty` по протоколам telnet и SSH:

```
line con 0
 password cisco
line aux 0
line vty 0 4
 exec-timeout 720 0
 password cisco
 login local
 transport input telnet ssh
!
```

Рассмотрим пример использования Pexpect в интерактивной оболочке Python:

```
>>> import pexpect
>>> child = pexpect.spawn('telnet 172.16.1.20')
>>> child.expect('Username')
0
>>> child.sendline('cisco')
6
>>> child.expect('Password')
0
>>> child.sendline('cisco')
6
>>> child.expect('iosv-1#')
0
>>> child.sendline('show version | i V')
19
>>> child.before
b": \r\n*************************************************************
***********\r\n* IOSv is strictly limited to use for evaluation,
demonstration and IOS  *\r\n* education. IOSv is provided as-is and
is not supported by Cisco's      *\r\n* Technical Advisory Center. Any
use or disclosure, in whole or in part, *\r\n* of the IOSv Software
```

```
or Documentation to any third party for any       *\r\n* purposes is
expressly prohibited except as otherwise authorized by     *\r\n* Cisco
in writing.                                                    *\r\n***
**************************************************************
*******\r\n"
>>> child.sendline('exit')
5
>>> exit()
```

Начиная с версии 4.0 в Pexpect появилась поддержка Windows. Но, как отмечается в официальной документации, она пока носит экспериментальный характер.

В предыдущем интерактивном примере Pexpect создает дочерний процесс и наблюдает за ним. Здесь видно два важных метода, `expect()` и `sendline()`. В вызов метода `expect()` передается строка, которую ищет процесс Pexpect, она служит признаком получения ожидаемого ответа — ожидаемым шаблоном. В этом примере мы знали, что перед выводом строки приглашения (`iosv-1#`) маршрутизатор отправил всю необходимую информацию. Методу `sendline()` передается строка для отправки удаленному устройству в качестве команды. Есть также похожий метод `send()`, но `sendline()` добавляет в конец символ перевода строки, который воспринимается как нажатие клавиши Enter. С точки зрения маршрутизатора все выглядит так, будто кто-то ввел текст в терминале. Иными словами, маршрутизатор «думает», что взаимодействует с человеком, хотя на самом деле все команды отдает компьютер.

Свойствам `before` и `after` присваивается текст, который выводит дочерний процесс. В свойство `before` записывается текст, предшествующий ожидаемому шаблону. В свойство `after` — сам шаблон. В данном случае в `before` попадет вывод между двумя ожидаемыми совпадениями (`iosv-1#`), включая команду `show version`, а в `after` — строка приглашения маршрутизатора:

```
>>> child.sendline('show version | i V')
19
>>> child.expect('iosv-1#')
0
>>> child.before
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\r\
nProcessor board ID 9Y0KJ2ZL98EQQVUED5T2Q\r\n'
>>> child.after
b'iosv-1#'
```

b' перед возвращенным текстом — это индикатор байтовой строки в Python (https://docs.python.org/3.7/library/stdtypes.html).

Что произойдет, если попробовать ожидать неправильный ответ? Например, если после создания дочернего процесса попробовать ждать строку `username` (с маленькой буквы u) вместо `Username`, то Pexpect будет ждать появления именно `username`. А поскольку маршрутизатор никогда не возвращает это слово, Pexpect просто зависнет. Рано или поздно сеанс завершится по тайм-ауту, но вы можете прервать его вручную, нажав Ctrl+C.

Метод `expect()` ждет, пока дочерний процесс не вернет заданную строку, поэтому, если регистр первой буквы u не должен учитываться, то можно использовать такое выражение:

```
>>> child.expect('[Uu]sername')
```

Квадратная скобка играет роль операции `or` (или), которая сообщает дочернему процессу, что тот должен ожидать маленькую или большую букву u, за которой следует строка `sername`. Таким образом, процесс будет ожидать любую из строк: `Username` или `username`.

Больше о регулярных выражениях в Python читайте на странице https://docs.python.org/3.7/library/re.html.

Методу `expect()` можно также передать список вариантов вместо одной строки, которые к тому же могут быть регулярными выражениями. Попробуем в примере выше передать список, чтобы учесть две возможные строки:

```
>>> child.expect(['Username', 'username'])
```

В общем случае методу `expect` предпочтительнее передать одну строку с регулярным выражением, определяющим все варианты, например, написания имени хоста; но если нужно учесть совершенно разные ответы маршрутизатора, как, например, отклонение пароля, то используйте список вариантов. Например, если вы применяете разные пароли для входа в систему, вам следует перехватить строку `% Login invalid`, а также приглашение командной строки устройства.

Важное отличие регулярных выражений Pexpect от регулярных выражений Python: первые выполняют так называемое нежадное сопоставление, то есть при использовании специальных символов они захватывают наименьший участок строки. Поскольку регулярные выражения Pexpect применяются к потоку, вы не можете заглядывать вперед, так как вам неизвестно, закончил ли работу дочерний процесс, генерирующий этот поток. Это означает, что знак доллара `$`, соответствующий концу строки, бесполезен, так как шаблон `.+` всегда обнаруживает совпадение, но ничего не возвращает, а шаблон `.*` захватывает наимень-

шую часть строки. Просто помните об этом и старайтесь максимально подробно описывать ожидаемые строки.

Рассмотрим ситуацию:

```
>>> child.sendline('show run | i hostname')
22
>>> child.expect('iosv-1')
0
>>> child.before
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\r\
nProcessor board ID 9Y0KJ2ZL98EQQVUED5T2Q\r\n'
>>>
```

Хм-м... Что-то здесь не так. Сравните это с предыдущим консольным выводом; здесь ожидалось получить `hostname iosv-1`:

```
iosv-1#sh run | i hostname
hostname iosv-1
```

Но присмотритесь к ожидаемой строке. Мы упустили решетку (`#`) после имени хоста `iosv-1`. В результате родительский процесс принял вторую часть вывода за ожидаемую строку:

```
>>> child.sendline('show run | i hostname')
22
>>> child.expect('iosv-1#')
0
>>> child.before
b'#show run | i hostname\r\nhostname iosv-1\r\n'
```

После нескольких примеров с Pexpect вырисовывается закономерность. Пользователь описывает последовательность взаимодействий между процессом Pexpect и дочерним приложением. С переменными и циклами Python мы можем создать полезную программу, которая поможет нам собирать информацию и вносить изменения в сетевые устройства.

Наша первая программа на основе Pexpect

Наша первая программа, `chapter2_1.py`, основана на экспериментах из предыдущего раздела, но с дополнительным кодом:

```
#!/usr/bin/env python
import pexpect
devices = {'iosv-1': {'prompt': 'iosv-1#', 'ip': '172.16.1.20'},
           'iosv-2': {'prompt': 'iosv-2#', 'ip': '172.16.1.21'}}
```

```
username = 'cisco'
password = 'cisco'

for device in devices.keys():
    device_prompt = devices[device]['prompt']
    child = pexpect.spawn('telnet ' + devices[device]['ip'])
    child.expect('Username:')
    child.sendline(username)
    child.expect('Password:')
    child.sendline(password)
    child.expect(device_prompt)
    child.sendline('show version | i V')
    child.expect(device_prompt)
    print(child.before)
    child.sendline('exit')
```

В строке 3 мы используем вложенный словарь:

```
devices = {'iosv-1': {'prompt': 'iosv-1#', 'ip': '172.16.1.20'},
           'iosv-2': {'prompt': 'iosv-2#', 'ip': '172.16.1.21'}}
```

Он позволяет ссылаться на устройство (такое как `iosv-1`) с соответствующим IP-адресом и приглашением к вводу. Позже в цикле эти значения можно передать методу `expect()`.

Для каждого устройства на экран выводится результат выполнения `show version | i V`:

```
(venv) $ python chapter2_1.py

b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\r\
nProcessor board ID 9Y0KJ2ZL98EQQVUED5T2Q\r\n'

b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\r\n'
```

Итак, мы рассмотрели простой пример работы с Pexpect. Теперь подробнее обсудим возможности этой библиотеки.

Другие возможности Pexpect

Возможности Pexpect из этого раздела пригодятся в разных ситуациях.

Вы можете изменить время ожидания метода `expect()` (которое по умолчанию равно 30 секундам) в зависимости от скорости вашего соединения с устройством. Для этого предусмотрен аргумент `timeout`:

```
>>> child.expect('Username', timeout=5)
```

С помощью метода `interact()` можно дать пользователю возможность самому ввести команду. Это полезно, когда вам нужно автоматизировать лишь определенные участки исходной задачи:

```
>>> child.sendline('show version | i V')
19
>>> child.expect('iosv-1#')
0
>>> child.before
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor
board ID 9MM4BI7B0DSWK40KV1IIRrn'
>>> child.interact()
show version | i V
Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version
15.6(3)M2, RELEASE SOFTWARE (fc2)
Processor board ID 9Y0KJ2ZL98EQQVUED5T2Q
iosv-1#sh run | i hostname
hostname iosv-1
iosv-1#exit
Connection closed by foreign host.
>>>
```

Об объекте `child.spawn` можно получить много информации, если вывести его в виде строки:

```
>>> str(child)
"<pexpect.pty_spawn.spawn object at 0x7f95f25ff780>\ncommand: /usr/
bin/telnet\nargs: ['/usr/bin/telnet', '172.16.1.20']\nbuffer (last 100
chars): b''\nbefore (last 100 chars): b'                        *\\r\\n*****
**************************************************
**********\\r\\n'\nafter: b'iosv-1#'\nmatch: <_sre.SRE_Match object;
span=(612, 619), match=b'iosv-1#'>\nmatch_index: 0\nexitstatus: 1\
nflag_eof: False\npid: 5676\nchild_fd: 5\nclosed: False\ntimeout: 30\
ndelimiter: <class 'pexpect.exceptions.EOF'>\nlogfile: None\nlogfile_
read: None\nlogfile_send: None\nmaxread: 2000\nignorecase: False\
nsearchwindowsize: None\ndelaybeforesend: 0.05\ndelayafterclose: 0.1\
ndelayafterterminate: 0.1"
>>>
```

Самое полезное средство отладки для Pexpect — запись вывода в файл:

```
>>> child = pexpect.spawn('telnet 172.16.1.20')
>>> child.logfile = open('debug', 'wb')
```

Используйте child.logfile = open('debug', 'w') в Python 2. В Python 3 по умолчанию используются байтовые строки. Больше о возможностях Pexpect читайте на странице https://pexpect.readthedocs.io/en/stable/api/index.html.

До сих пор мы в своих примерах использовали telnet, благодаря чему все взаимодействия выполнялись с помощью обычного текста. В современных сетях для управления обычно применяют *безопасные командные оболочки (Secure Shell, SSH)*. В следующем разделе мы поговорим о работе с Pexpect через SSH.

Pexpect и SSH

Если попытаться реализовать предыдущий пример с использованием SSH вместо telnet, впечатления будут не самыми приятными. SSH всегда требует включать имя пользователя в сеансе, отвечать на вопросы при генерации нового ключа и выполнять много других рутинных действий. Сеанс SSH можно наладить разными способами, но, к счастью, в Pexpect есть подкласс `pxssh`, специально предназначенный для установления SSH-соединений. Этот класс представляет методы для входа и выхода из системы и ряд неочевидных механизмов для улаживания проблем с аутентификацией.

Сгенерируем ключ для работы с `iosv-1` по SSH:

```
iosv-1(config)#crypto key generate rsa general-keys
The name for the keys will be: iosv-1.virl.info
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]: 2048
% Generating 2048 bit RSA keys, keys will be non-exportable...
[OK] (elapsed time was 2 seconds)
```

В остальном процедура почти не меняется, если не считать вызовов `login()` и `logout()`:

```
>>> from pexpect import pxssh
>>> child = pxssh.pxssh()
>>> child.login('172.16.1.20', 'cisco', 'cisco', auto_prompt_reset=False)
True
>>> child.sendline('show version | i V')
19
>>> child.expect('iosv-1#')
0
>>> child.before
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2) Processor
board ID 9MM4BI7B0DSWK40KV1IIRrn'
>>> child.logout()
>>>
```

Обратите внимание на аргумент `auto_prompt_reset=False` в вызове метода `login()`. По умолчанию для синхронизации вывода `pxssh` использует приглашение командной оболочки. Но, так как в большинстве случаев bash-shell и c-shell используют переменную окружения `PS1`, этот способ вызовет ошибку в Cisco и других сетевых устройствах.

Итоговая программа на основе Pexpect

В заключение воплотим все, что было сказано о Pexpect, в сценарии. Код в виде сценария легче использовать в промышленной среде и делиться им с коллегами. Это будет наш второй сценарий, `chapter2_2.py`.

Сценарий доступен в репозитории GitHub для этой книги по ссылке https://github.com/PacktPublishing/Mastering-Python-Networking-Third-Edition.

Если вы еще не сгенерировали SSH-ключ на другом маршрутизаторе, `iosv-2`, сделайте это сейчас:

```
iosv-2(config)#crypto key generate rsa general-keys
The name for the keys will be: iosv-2.virl.info
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]: 2048
% Generating 2048 bit RSA keys, keys will be non-exportable...
[OK] (elapsed time was 2 seconds)
```

Взгляните на следующий код:

```
#!/usr/bin/env python

import getpass
from pexpect import pxssh

devices = {'iosv-1': {'prompt': 'iosv-1#', 'ip': '172.16.1.20'},
           'iosv-2': {'prompt': 'iosv-2#', 'ip': '172.16.1.21'}}
commands = ['term length 0', 'show version', 'show run']

username = input('Username: ')
password = getpass.getpass('Password: ')

# Цикл перебора устройств
for device in devices.keys():
    outputFileName = device + '_output.txt'
    device_prompt = devices[device]['prompt']
    child = pxssh.pxssh()
```

```
    child.login(devices[device]['ip'], username.strip(), password.
strip(), auto_prompt_reset=False)
    # Цикл перебора команд с записью вывода
    with open(outputFileName, 'wb') as f:
        for command in commands:
            child.sendline(command)
            child.expect(device_prompt)
            f.write(child.before)

    child.logout()
```

Этот сценарий дополняет нашу первую программу на основе Pexpect некоторыми возможностями:

- использует SSH вместо telnet;
- поддерживает не одну, а несколько команд в виде списка (строка 8) и перебирает эти команды в цикле (начиная со строки 20);
- имя и пароль не прописаны в коде сценария — их должен ввести пользователь;
- записывает вывод для дальнейшего анализа в два файла, `iosv-1_output.txt` и `iosv-2_output.txt`.

Для приема пользовательского ввода в Python 2 используйте raw_input() вместо input() и режим открытия файла w вместо wb.

Библиотека Python Paramiko

Paramiko — это реализация протокола SSHv2 для Python. Как и подкласс `pxssh` из библиотеки Pexpect, Paramiko упрощает организацию взаимодействий с удаленными устройствами по SSHv2. Но, в отличие от `pxssh`, Paramiko не имеет поддержки Telnet. И эта библиотека предоставляет как клиентские, так и серверные операции.

Paramiko — это низкоуровневый SSH-клиент в составе высокоуровневого фреймворка автоматизации под названием Ansible. Последний будет рассмотрен в главах 4 и 5. А пока сосредоточимся на Paramiko.

Установка Paramiko

Пакет Paramiko легко установить с помощью утилиты `pip`. Однако у него есть жесткая зависимость от криптографической библиотеки низкоуровневых алгоритмов шифрования для протокола SSH, написанной на языке C.

Инструкции по установке для Windows, Mac и других разновидностей Linux можно найти на странице https://cryptography.io/en/latest/installation/.

Ниже показан пошаговый процесс установки Paramiko на нашей виртуальной машине под управлением Ubuntu 18.04:

```
sudo apt-get install build-essential libssl-dev libffi-dev python3-dev
pip install cryptography
pip install paramiko
```

Проверим корректность установки этой библиотеки, импортировав ее с помощью интерпретатора Python:

```
$ python
Python 3.6.8 (default, Aug 20 2019, 17:12:48)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramiko
>>> exit()
```

А теперь перейдем к краткому обзору Paramiko в следующем разделе.

Краткий обзор Paramiko

Рассмотрим небольшой пример с Paramiko в интерактивной оболочке Python 3:

```
>>> import paramiko, time
>>> connection = paramiko.SSHClient()
>>> connection.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> connection.connect('172.16.1.20', username='cisco', password='cisco',
look_for_keys=False, allow_agent=False)
>>> new_connection = connection.invoke_shell()
>>> output = new_connection.recv(5000)
>>> print(output) b"r
\n\*************************************************************
**********
***rn* IOSv is strictly limited to use for evaluation, demonstration
and IOS *rn* education. IOSv is provided as-is and is not supported by
Cisco's
*rn* Technical Advisory Center. Any use or disclosure, in whole or in
part,
*rn* of the IOSv Software or Documentation to any third party for any
*rn* purposes is expressly prohibited except as otherwise authorized by
*rn* Cisco in writing.
*rn************************************************************
***********
**rniosv-1#"
```

```
>>> new_connection.send("show version | i V\n")
19
>>> time.sleep(3)
>>> output = new_connection.recv(5000)
>>> print(output)
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor
board ID 9MM4BI7B0DSWK40KV1IIRrniosv-1#'
>>> new_connection.close()
>>>
```

Функция time.sleep() приостанавливает сценарий, чтобы перехватить весь вывод. Это особенно полезно при работе с медленными сетевыми соединениями или перегруженными устройствами. Данная команда не обязательна, ее рекомендуется применять по ситуации.

Даже если вы никогда прежде не видели Paramiko в действии, элегантность языка Python и его четкий синтаксис позволят вам предположить, что эта программа пытается сделать:

```
>>> import paramiko
>>> connection = paramiko.SSHClient()
>>> connection.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> connection.connect('172.16.1.20', username='cisco', password='cisco',
look_for_keys=False, allow_agent=False)
```

В первых четырех строках создается экземпляр класса `SSHClient` из библиотеки Paramiko. Следующая строка задает политику работы с ключами на клиентской стороне; в данном случае строки `iosv-1` нет ни среди системных ключей хоста, ни среди ключей приложения. В нашем примере ключ добавляется автоматически в объект приложения `HostKeys`. На этом этапе при входе в маршрутизатор вы увидите дополнительный сеанс входа от Paramiko:

```
iosv-1#who
Line User Host(s) Idle Location
*578 vty 0 cisco idle 00:00:00 172.16.1.1
579 vty 1 cisco idle 00:01:30 172.16.1.173
Interface User Mode Idle Peer Address
iosv-1#
```

Следующие несколько строк вызывают в контексте соединения интерактивную командную оболочку, после чего происходит отправка команд и извлечение вывода. В конце соединение закрывается.

Некоторые читатели с опытом работы с Paramiko могут предпочесть метод `exec_command()` вызову интерактивной командной оболочки. Почему мы сдела-

ли такой выбор? К сожалению, в Cisco IOS метод `exec_command()` позволяет выполнить лишь одну команду. Рассмотрим пример, где в контексте соединения вызывается `exec_command()`:

```
>>> connection.connect('172.16.1.20', username='cisco', password='cisco',
look_for_keys=False, allow_agent=False)
>>> stdin, stdout, stderr = connection.exec_command('show version | i V\n')
>>> stdout.read()
b'Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor board ID
9MM4BI7B0DSWK40KV1IIRrn'
>>>
```

Все прекрасно работает; но, если вы проверите количество сеансов на устройстве Cisco, то заметите, что соединение было разорвано самим устройством, без вашего участия:

```
iosv-1#who
Line User Host(s) Idle Location
*578 vty 0 cisco idle 00:00:00 172.16.1.1
Interface User Mode Idle Peer Address
iosv-1#
```

Поскольку SSH-сеанс уже неактивен, `exec_command()` вернет ошибку при попытке послать удаленному устройству новые команды:

```
>>> stdin, stdout, stderr = connection.exec_command('show version | i V\n')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/local/lib/python3.5/dist-packages/paramiko/client.py", line
435, in exec_command
chan = self._transport.open_session(timeout=timeout)
File "/usr/local/lib/python3.5/dist-packages/paramiko/transport.py", line
711, in open_session
timeout=timeout)
File "/usr/local/lib/python3.5/dist-packages/paramiko/transport.py", line
795, in open_channel
raise SSHException('SSH session not active') paramiko.ssh_exception.
SSHException: SSH session not active
>>>
```

В предыдущем примере команда `new_connection.recv()` вывела содержимое буфера и автоматически его очистила. Что произойдет, если не очистить полученный буфер? Его содержимое будет перезаписываться поступающим выводом:

```
>>> new_connection.send("show version | i V\n")
19
>>> new_connection.send("show version | i V\n")
19
```

```
>>> new_connection.send("show version | i V\n")
19
>>> new_connection.recv(5000)
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor
board ID 9MM4BI7B0DSWK40KV1IIRrniosv-1#show version | i VrnCisco IOS
Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T,
RELEASE SOFTWARE (fc2)rnProcessor board ID 9MM4BI7B0DSWK40KV1IIRrniosv-
1#show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor
board ID 9MM4BI7B0DSWK40KV1IIRrniosv-1#'
>>>
```

Чтобы вывод всегда оставался предсказуемым, его следует извлекать из буфера после выполнения каждой команды.

Наша первая программа, написанная с использованием Paramiko

Наша первая программа будет иметь ту же структуру, что и итоговый пример с Pexpect. Мы точно так же пройдемся по спискам устройств и команд, но вместо Pexpect используем Paramiko. В итоге вы поймете различия этих двух библиотек.

Загрузите файл `chapter2_3.py` из репозитория GitHub для этой книги по адресу https://github.com/PacktPublishing/Mastering-Python-Networking-Third-Edition, если вы этого еще не сделали. Ниже перечислены заметные различия:

```
devices = {'iosv-1': {'ip': '172.16.1.20'}, 'iosv-2': {'ip': '172.16.1.21'}}
```

С Paramiko не нужно искать строку приглашения устройства, поэтому словарь устройств можно упростить:

```
commands = ['show version', 'show run']
```

В Paramiko нет эквивалента операции отправки строки, поэтому мы вручную вставляем перевод строки в каждую команду:

```
def clear_buffer(connection):
    if connection.recv_ready():
        return connection.recv(max_buffer)
```

Мы добавили новый метод для очистки буфера, через который отправляются команды, такие как `terminal length 0` или `enable`, поскольку нам не нужен вывод этих команд. Мы просто хотим очистить буфер и перейти к приглашению

командной строки. Позже эта функция будет использована в цикле (строка 25 в сценарии):

```
output = clear_buffer(new_connection)
```

Оставшаяся часть программы понятна; мы уже видели подобный код выше в этой главе.

И последнее: ввиду интерактивности этой программы между отправкой буфера и извлечением вывода должна быть задержка, чтобы удаленное устройство успело выполнить команду:

```
time.sleep(5)
```

После очистки буфера, в промежутке между выполнением команд, мы ждем пять секунд. Благодаря этому, если устройство слишком занято, у него будет достаточно времени на ответ.

Другие возможности Paramiko

Мы еще вернемся к библиотеке Paramiko в главе 4, при обсуждении Ansible, так как она служит внутренним транспортным механизмом для многих сетевых модулей. А в этом разделе мы рассмотрим дополнительные возможности Paramiko.

Paramiko для серверов

Paramiko можно также использовать для управления серверами по SSHv2. Рассмотрим пример. В нашем SSH-сеансе аутентификация будет выполняться с помощью ключа.

Для этого примера я использовал еще одну виртуальную машину под управлением Ubuntu с тем же гипервизором, что и на удаленном сервере. Подойдет и сервер в симуляторе VIRL или одном из публичных облаков, такой как Amazon AWS EC2.

Сгенерируем открытый и закрытый ключи для нашего хоста с Paramiko:

```
ssh-keygen -t rsa
```

По умолчанию эта команда генерирует в домашнем каталоге пользователя `~/.ssh` открытый и закрытый ключи с именами `id_rsa.pub` и `id_rsa` соответственно. Закрытый ключ, как и пароль, следует держать в тайне от всех. Сообщение будет

зашифровано локально с помощью закрытого ключа и расшифровано на удаленным хосте с использованием открытого. Открытый ключ необходимо скопировать на удаленный сервер. В промышленных условиях это лучше сделать без использования сети, с помощью USB-накопителя; в условиях нашей лаборатории можно просто скопировать открытый ключ в файл `~/.ssh/authorized_keys` на удаленном хосте. Откройте окно терминала и скопируйте содержимое `~/.ssh/id_rsa.pub` на своем управляющем хосте, где у вас установлена библиотека Paramiko:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa <ваш открытый ключ>
```

Затем сохраните его в пользовательском каталоге на удаленном хосте; в данном случае я использую на обеих сторонах `echou`:

```
<Удаленный хост>$ vim ~/.ssh/authorized_keys
ssh-rsa <ваш открытый ключ>
```

Все готово к управлению удаленным хостом посредством Paramiko. Обратите внимание: в этом примере мы используем закрытый ключ для аутентификации и отправляем команды с помощью `exec_command()`:

```
>>> import paramiko
>>> key = paramiko.RSAKey.from_private_key_file('/home/echou/.ssh/id_rsa')
>>> client = paramiko.SSHClient()
>>> client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> client.connect('192.168.199.182', username='echou', pkey=key)
>>> stdin, stdout, stderr = client.exec_command('ls -l')
>>> stdout.read()
b'total 44ndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14 Desktopndrwxr-xr-x 2
echou echou 4096 Jan 7 10:14 Documentsndrwxr-xr-x 2 echou echou 4096 Jan 7
10:14 Downloadsn-rw-r--r-- 1 echou echou 8980 Jan 7 10:03
examples.desktopndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14 Musicndrwxrxr-x
echou echou 4096 Jan 7 10:14 Picturesndrwxr-xr-x 2 echou echou 4096 Jan
7 10:14 Publicndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14 Templatesndrwxrxr-x
2 echou echou 4096 Jan 7 10:14 Videosn'
>>> stdin, stdout, stderr = client.exec_command('pwd')
>>> stdout.read()
b'/home/echoun'
>>> client.close()
>>>
```

Обратите внимание, что в примере с сервером нам не понадобилось создавать интерактивный сеанс для выполнения серии команд. Теперь вы можете выключить аутентификацию на основе пароля в настройках SSHv2 своего удаленного хоста, чтобы применять более безопасный метод автоматической аутентификации по ключу. Некоторые сетевые устройства, такие как коммутаторы Cumulus и Vyatta, тоже поддерживают этот метод.

Итоговая программа на основе Paramiko

Теперь сделаем нашу программу на основе Paramiko более универсальной. Сценарий имеет один недостаток: его необходимо модифицировать, чтобы добавить новые или удалить старые хосты или чтобы заменить команды, которые мы хотим выполнить удаленно.

Это объясняется тем, что адреса хостов и команды прописаны вручную в самом сценарии. И это повышает риск ошибок. К тому же коллеги, которым вы передадите этот сценарий, могут плохо ориентироваться в Python, Paramiko или Linux.

Поместив адреса хостов и команды в отдельные файлы, которые будут передаваться сценарию в качестве параметров, мы устраним некоторые из этих проблем. Пользователи (и вы сами в будущем) смогут просто отредактировать эти текстовые файлы, если им понадобится поменять хост или команды.

Эти нововведения реализованы в сценарии `chapter2_4.py`.

Вместо того чтобы прописывать команды в коде, мы вынесли их в отдельный файл `commands.txt`. До сих пор мы использовали команду `show` для вывода информации; в этом примере мы внесем изменения в конфигурацию. В частности, поменяем размер буфера журнала, увеличив его до 30 000 байт:

```
$ cat commands.txt
config t
logging buffered 30000
end
copy run start
```

Информация об устройствах будет храниться в файле `devices.json`. Мы выбрали формат JSON, потому что Python с легкостью преобразует его в словарь:

```
$ cat devices.json
{
"iosv-1": {"ip": "172.16.1.20"},
"iosv-2": {"ip": "172.16.1.21"}
}
```

Изменения в сценарии:

```
with open('devices.json', 'r') as f:
    devices = json.load(f)

with open('commands.txt', 'r') as f:
    commands = f.readlines()
```

Вот его сокращенный вывод:

```
(venv) $ python chapter2_4.py
Username: cisco
Password:
b'terminal length 0\r\niosv-1#config t\r\nEnter configuration commands,
one per line.  End with CNTL/Z.\r\niosv-1(config)#'
b'logging buffered 30000\r\niosv-1(config)#'
b'end\r\niosv-1#'
<опущено>
```

Проверим изменения в `running-config` и `startup-config`:

```
iosv-1#sh run | i logging
logging buffered 30000
iosv-1#sh start | i logging
logging buffered 30000

iosv-2#sh run | i logging
logging buffered 30000
iosv-2#sh start | i logging
logging buffered 30000
```

Paramiko — это библиотека общего назначения для работы с интерактивными консольными программами. Для сетевого администрирования есть другая библиотека, Netmiko, написанная на основе Paramiko специально для управления сетевыми устройствами. Рассмотрим этот инструмент в следующем разделе.

Библиотека Netmiko

Paramiko отлично подходит для низкоуровневого взаимодействия с Cisco IOS и устройствами других производителей. Но, как вы могли заметить в предыдущих примерах, аутентификация и выполнение команд на устройствах `iosv-1` и `iosv-2` во многом совпадают. Чем больше команд мы будем автоматизировать, тем чаще нам придется повторять одни и те же действия по захвату и преобразованию вывода в удобный нам формат. Было бы здорово, если бы кто-нибудь написал библиотеку для Python, которая упрощает низкоуровневые шаги и позволяет делиться ими с другими сетевыми инженерами!

Начиная с 2014 года Кирк Байерс (https://github.com/ktbyers) работает над открытыми проектами, призванными упростить управление сетевыми устройствами. Мы рассмотрим пример использования созданной им библиотеки Netmiko (https://github.com/ktbyers/netmiko).

Для начала установим пакет Netmiko с помощью `pip`:

```
(venv) $ pip install netmiko
```

Возьмем пример с сайта Кирка, https://pynet.twb-tech.com/blog/automation/netmiko.html, и адаптируем его к нашим лабораторным работам. Для начала импортируем библиотеку и ее класс `ConnectHandler`. Затем определим параметр `device` в виде словаря и передадим его этому классу. Обратите внимание, что в параметре `device` определены поля `device_type` и `cisco_ios`.

```
>>> from netmiko import ConnectHandler
>>> ios_v1 = {'device_type': 'cisco_ios', 'host': '172.16.1.20',
'username': 'cisco', 'password': 'cisco'}
>>> net_connect = ConnectHandler(**ios_v1)
```

С этого места наш код становится проще. Как видите, библиотека автоматически определяет приглашение командной строки устройства и форматирует вывод, возвращаемый командой `show`:

```
>>> net_connect.find_prompt()
'iosv-1#'
>>> output = net_connect.send_command('show ip int brief')
>>> print(output)
Interface                  IP-Address      OK? Method Status
Protocol
GigabitEthernet0/0         172.16.1.20     YES NVRAM  up
up
GigabitEthernet0/1         10.0.0.5        YES NVRAM  up
up
Loopback0                  192.168.0.1     YES NVRAM  up
up
```

Рассмотрим еще один пример, но уже для второго устройства Cisco IOS. На этот раз определим параметр `iosv-2` при инициализации объекта `ConnectHandler`, а вместо `show` отправим команду `configuration`. Стоит отметить, что атрибут `command` — это список, в котором может быть несколько команд:

```
>>> net_connect_2 = ConnectHandler(device_type='cisco_ios',
host='172.16.1.21', username='cisco', password='cisco')
>>> output = net_connect_2.send_config_set(['logging buffered 19999'])
>>> print(output)
config term
Enter configuration commands, one per line.  End with CNTL/Z.
iosv-2(config)#logging buffered 19999
iosv-2(config)#end
iosv-2#
>>> exit()
```

Библиотека Netmiko используется многими сетевыми инженерами и может сэкономить много времени. В следующем разделе речь пойдет о фреймворке Nornir (https://github.com/nornir-automation/nornir), упрощающем низкоуровневые взаимодействия.

Фреймворк Nornir

Nornir (https://nornir.readthedocs.io/en/latest/) — это фреймворк для автоматизации, он написан на чистом Python и рассчитан на использование из этого языка. Другой фреймворк для автоматизации на языке Python под названием Ansible мы обсудим в главах 4 и 5. В этой главе я представляю Nornir в качестве альтернативного способа автоматизации низкоуровневых взаимодействий с устройствами. Но, если вы только осваиваетесь в этой области, к фреймворкам лучше переходить после прочтения главы 5. Можете пролистать этот пример и вернуться к нему позже.

Как уже повелось, начнем с установки пакета Nornir в нашей среде:

```
(venv) $ pip install nornir
```

Для работы с Nornir нужно определить файл `hosts.yaml` с информацией об устройствах в формате YAML. Эта информация не отличается от той, которую мы уже оформляли в виде словаря в примере с Netmiko:

```
---
iosv-1:
    hostname: '172.16.1.20'
    port: 22
    username: 'cisco'
    password: 'cisco'
    platform: 'cisco_ios'

iosv-2:
    hostname: '172.16.1.21'
    port: 22
    username: 'cisco'
    password: 'cisco'
    platform: 'cisco_ios'
```

Для взаимодействия с нашим устройством можно воспользоваться плагином `netmiko` из библиотеки Nornir, как показано в файле chapter2_5.py:

```
from nornir import InitNornir
from nornir.plugins.tasks.networking import netmiko_send_command
from nornir.plugins.functions.text import print_result
```

```
nr = InitNornir()

result = nr.run(
    task=netmiko_send_command,
    command_string="show arp"
)

print_result(result)
```

Ниже показан вывод сценария:

```
(venv) $ python chapter2_5.py
netmiko_send_
command************************************************************
* iosv-1 ** changed : False ***********************************************
*******
vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvv
vvvvvvv INFO
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.0.0.5               -   fa16.3e0e.a3a3  ARPA
GigabitEthernet0/1
Internet  10.0.0.6              40   fa16.3ed7.1041  ARPA
GigabitEthernet0/1
^^^^ END netmiko_send_command ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^
* iosv-2 ** changed : False ***********************************************
*******
vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvv
vvvvvvv INFO
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.0.0.5              40   fa16.3e0e.a3a3  ARPA
GigabitEthernet0/1
Internet  10.0.0.6               -   fa16.3ed7.1041  ARPA
GigabitEthernet0/1
^^^^ END netmiko_send_command ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^
```

Nornir поддерживает и другие плагины, помимо Netmiko: например, популярную библиотеку NAPALM (https://github.com/napalm-automation/napalm). Актуальный список доступных плагинов смотрите на странице проекта Nornir по адресу https://nornir.readthedocs.io/en/latest/plugins/index.html.

Другие фреймворки для автоматизации, pyATS и Genie, мы рассмотрим в главе 15 о тестировании.

В этой главе мы сделали довольно большой шаг в направлении автоматизации нашей сети с помощью Python. Однако некоторые методы больше напоминали обходные решения. Мы пытались убедить удаленные устройства в том, что на другом конце соединения находится живой человек. Этот подход не поменялся

и при работе с библиотеками вроде Netmiko или фреймворком Nornir. Тот факт, что кто-то потрудился и спрятал от нас всю грязную работу, связанную с низкоуровневым взаимодействием, вовсе не означает, что недостатки устройств, поддерживающих только CLI, нас больше не касаются.

В следующих разделах мы обсудим слабые стороны Pexpect и Paramiko и сравним их с другими инструментами. А уже в следующей главе рассмотрим методики на основе API.

Недостатки Pexpect и Paramiko по сравнению с другими инструментами

Самый главный недостаток нашего подхода к автоматизации удаленных устройств, поддерживающих только CLI, заключается в том, что эти устройства не возвращают структурированных данных. Их вывод идеально подходит для отображения в терминале, где его может прочитать человек, но не для компьютерной программы. Человеческий глаз легко замечает пустые строки, отделяющие блоки данных, а компьютер видит только символы перевода строки.

Более удачные методы будут представлены в следующей главе. Для начала же обсудим принцип идемпотентности.

Идемпотентное взаимодействие с сетевыми устройствами

Значение термина *«идемпотентность»* зависит от контекста. В этой книге он означает, что при выполнении одних и тех же вызовов к удаленному устройству клиент всегда получает один и тот же результат. Это ценное свойство, и вряд ли кто-то с этим поспорит. Представьте ситуацию, когда при каждом выполнении ваш сценарий дает разные результаты. Меня такая возможность пугает. Как можно в таком случае доверять своему сценарию? Наши попытки наладить автоматизацию оказались бы тщетными, так как нам пришлось бы готовиться к обработке разного вывода.

Pexpect и Paramiko генерируют последовательности команд в интерактивном режиме, что повышает вероятность неидемпотентного взаимодействия. И если учесть, что нужные данные нужно извлекать из возвращаемого текста, риск получения разных выводов только увеличивается. Между написанием сценария и его сотым выполнением на другом конце соединения может что-то измениться. Например, если в следующей модели устройства производитель

изменит выводимый текст, то сценарий придется обновить, иначе он перестанет работать.

Сценарий, применяемый в промышленных условиях, должен быть как можно более идемпотентным.

Плохая автоматизация усугубляет негативные последствия

Плохая автоматизация способствует совершению ошибок. Компьютеры выполняют намного больше задач за одно и то же время, чем мы, инженеры. Вручную команды выполняются всегда медленнее, чем в сценарии. Однако сценарию не хватает надежной обратной связи между процедурами. В интернете полно историй о том, как люди нажимали клавишу Enter и тут же об этом жалели.

Мы должны свести к минимуму вероятность сбоев и серьезного ущерба. Все мы ошибаемся; тщательно тестируйте свои сценарии перед выполнением любых задач в промышленных условиях и ограничивайте возможность получения негативных последствий — эти два ключевых правила помогут вам выявлять ошибки до того, как случится беда. Конечно, от ошибок полностью не застрахован ни человек, ни программный инструмент, но мы можем их минимизировать. Как мы уже видели, несмотря на наличие отличных библиотек, метод взаимодействия на основе CLI несовершенен и ошибки вероятны. В следующей главе мы рассмотрим альтернативный подход с использованием API, который лишен некоторых из этих недостатков.

Резюме

В этой главе мы обсудили низкоуровневые методы прямого взаимодействия с сетевыми устройствами. Если устройство не предоставляет программных средств управления и внесения изменений, это исключает автоматизацию. Мы рассмотрели две библиотеки для автоматического администрирования оборудования, предоставляющего лишь CLI-интерфейс. Это полезные инструменты, но не совсем надежные, так как соответствующее сетевое оборудование предназначено для управления человеком, а не компьютером.

В главе 3 речь пойдет о сетевых устройствах, поддерживающих API и сети, ориентированные на намерения (intent-driven networking, IDN-сети).

3

API и IDN-сети

В главе 2 мы обсудили взаимодействие с сетевыми устройствами с помощью Pexpect и Paramiko. Оба эти инструмента используют непрерывный сеанс с имитацией ручного ввода команд, как будто человек сидит за терминалом. Во многом этот подход работает. Мы отправляем устройству команды и перехватываем результаты. Но когда объем вывода превышает несколько строк, его сложно интерпретировать в компьютерной программе. Pexpect и Paramiko возвращают последовательности символов, предназначенные для чтения человеком. Этот многострочный вывод с пробелами понятен живому пользователю, но компьютерная программа плохо справляется с его анализом.

Для автоматизации многих задач с использованием компьютерных программ мы должны как-то интерпретировать возвращаемые результаты и выполнять на их основе последующие действия. Если результаты не поддаются точной и предсказуемой интерпретации, мы не можем с уверенностью выполнить следующую команду.

К счастью, эта проблема была решена интернет-сообществом. Представьте себе то, как компьютер и человек читают веб-страницу. Человек видит слова, картинки, отступы, интерпретированные браузером; компьютер видит HTML-код, символы юникода и двоичные файлы. Но что, если веб-сайт необходимо превратить в веб-сервис для другого компьютера? Одни и те же веб-ресурсы должны быть рассчитаны как на живых пользователей, так и на компьютерные программы. Не напоминает ли вам это проблему, с которой мы уже сталкивались?

Решением является *интерфейс прикладного программирования (Application Program Interface, API)*. Необходимо отметить, что это концепция, понятие, а не какая-то конкретная технология или фреймворк. Из Википедии:

> *«В программировании интерфейс прикладного программирования (API) — это набор определений подпрограмм, протоколов и инструментов для создания прикладного программного обеспечения. В общем и целом — это набор четко определенных способов взаимодействий между различными программными компонентами. Хороший API упрощает разработку компьютерной программы, определяя все строительные блоки, которые затем используются программистом».*

В нашем случае набор четко определенных методов взаимодействий относится к программе на языке Python, с одной стороны, и к целевому устройству — с другой. API наших сетевых устройств предоставляют отдельный механизм управления для компьютерных программ. Конкретная реализация API зависит от производителя устройства. Одни производители предпочитают XML, другие JSON; одни могут использовать HTTPS в качестве транспортного протокола, а другие — предлагать готовые библиотеки-обертки для Python. В этой главе есть примеры каждого из этих вариантов.

Несмотря на разную реализацию, API следуют общей идее: это метод взаимодействия, оптимизированный для других компьютерных программ.

В этой главе мы рассмотрим такие темы, как:

- *инфраструктура как код (Infrastructure as Code, IaC)*, сети, ориентированные на намерения и моделирование данных;
- Cisco NX-API и *инфраструктура, ориентированная на приложения (Application Centric Infrastructure, ACI)*;
- *протокол конфигурации сети (Network Configuration Protocol, NETCONF)* и PyEZ;
- Arista eAPI и pyeapi.

Для начала попробуем объяснить, зачем к инфраструктуре следует относиться как к коду.

Инфраструктура как код

В идеальном мире сетевые инженеры и архитекторы, занимающиеся проектированием и администрированием сетей, должны заботиться не о взаимодействиях на уровне отдельных устройств, а о задачах, которые стоят перед сетью. Но все мы знаем, что мир неидеален. Много лет назад, когда я был наивным

и впечатлительным стажером в компании интернет-провайдера второго уровня, одним из моих первых заданий была установка маршрутизатора у нашего клиента, чтобы подключить его сеть Frame Relay (помните такую технологию?). «Как же я это сделаю?» — спросил я. Мне выдали стандартный набор инструкций по настройке соединений на основе Frame Relay.

Я пришел к клиенту, бездумно ввел нужные команды, взглянул на мигающие зеленые светодиоды, с удовлетворением упаковал свои вещи и поздравил себя с успешно проделанной работой. Это было захватывающее задание, но я не до конца понимал, что делаю. Я просто следовал инструкциям, не задумываясь о последствиях выполнения вводимых мною команд. Как бы я диагностировал проблему, если бы загорелся красный светодиод? Мне бы, наверное, пришлось звонить в офис своей компании и умолять о помощи (может, даже со слезами на глазах).

Конечно, сетевое проектирование нельзя свести к вводу команд в устройство. Это скорее построение инфраструктуры, которая позволяет доставлять услуги (или сервисы) из точки А в точку Б максимально легко. Команды, которые мы используем, и вывод, который мы интерпретируем, — это всего лишь средства достижения цели. То есть мы должны сосредоточиться на том, для чего нам нужна наша сеть. *Задачи, которые должна выполнять сеть, намного важнее синтаксиса команд, который мы используем для управления устройствами.* Если сделать эту идею еще более абстрактной и описать наши намерения с помощью программного кода, мы сможем определить всю нашу инфраструктуру как какое-то конкретное состояние. И чтобы обеспечивать это состояние, мы будем описывать нашу инфраструктуру с помощью необходимого ПО и фреймворков.

Сети, ориентированные на намерения

С момента выхода первого издания этой книги такие термины, как *сети на основе намерений (Intent-Based Networking, IBN)* и *сети, ориентированные на намерения (Intent-Driven Networking, IDN)*, стали намного популярней: производители начали использовать их для описания своих устройств следующего поколения. В целом эти понятия имеют одно значение. *По моему мнению, идея IDN-сети состоит в определении состояния, в котором она должна находиться, и приведении ее в это состояние с помощью программного кода.* Например, если мне нужно сделать порт 80 недоступным снаружи, это будет намерением сети, о котором я должен объявить. Его реализацией будет заниматься внутреннее ПО, которое знает синтаксис конфигурации и применяет требуемый список доступа на соответствующем граничном маршрутизаторе. Конечно, IDN — это лишь идея, воплощать которую можно по-разному. Для реализации объявлен-

ного намерения можно использовать библиотеку, фреймворк или готовый продукт, приобретенный у производителя.

API, как мне кажется, приближает нас к воплощению этой идеи. Если коротко, то инкапсуляция конкретных команд, которые выполняются на целевом устройстве, позволяет сосредоточиться на нашем намерении, а не на отдельных действиях. Если вернуться к примеру с блокированием порта 80, в Cisco для этого предусмотрены списки и группы доступа, а в Juniper — списки фильтрации. Но если использовать API, наша программа сможет запросить у исполнителя его намерение, скрыв от него тип физического устройства. Мы даже можем воспользоваться более высокоуровневым декларативным фреймворком, таким как Ansible (подробнее о нем — в главе 4). Но пока что давайте сосредоточимся на сетевых API.

Консольный вывод и структурированные результаты API-запроса

Представьте типичную ситуацию, когда нам нужно войти в сетевое устройство и убедиться, что все его интерфейсы находятся в состоянии `up/up` (то есть индикаторы состояния и протокола показывают `up`). Для сетевого инженера не составит труда войти в Cisco NX-OS, выполнить в терминале команду `show ip interface brief` и определить по ее выводу, какие интерфейсы активны:

```
nx-osv-2# show ip int brief
IP Interface Status for VRF "default"(1) Interface IP Address Interface
Status
Lo0 192.168.0.2 protocol-up/link-up/admin-up
Eth2/1 10.0.0.6 protocol-up/link-up/admin-up
nx-osv-2#
```

Человеческий глаз легко различает переводы строк, пробельные символы и первую строку с заголовками столбцов. На самом деле все это нужно, только чтобы помочь нам быстро скользнуть взглядом, скажем, по IP-адресам сверху вниз. С точки зрения компьютера все эти пробелы и переводы строк лишь отвлекают от действительно важной части вывода — от информации о состоянии интерфейсов. Чтобы это проиллюстрировать, приведу вывод Paramiko для этой же операции:

```
>>> new_connection.send('show ip int brief/n')
16
>>> output = new_connection.recv(5000)
>>> print(output)
b'sh ip int briefrrnIP Interface Status for VRF "default"(1)r\nInterface
IP Address Interface StatusrnLo0 192.168.0.2 protocol-up/link-up/admin-up
rnEth2/1 10.0.0.6 protocol-up/link-up/admin-up r\nrnx- osv-2# '
>>>
```

Ниже представлен псевдокод (то есть упрощенный вариант настоящего кода, который я бы для этого написал), анализирующий данные в переменной `output` и извлекающий из них нужную мне информацию.

1. Разделим текст на отдельные строки по символу перевода строки.
2. Первая строка с командой `show ip interface brief` нам не нужна, поэтому мы ее отбрасываем.
3. Во второй строке захватываем весь текст вплоть до VRF и сохраняем его в переменную, так как нас интересует состояние VRF в выводе.
4. Мы не знаем, сколько всего интерфейсов у устройства, поэтому для анализа остальных строк воспользуемся регулярным выражением; нас интересуют строки, начинающиеся с названия интерфейса: `lo` (loopback) или `Eth` (Ethernet).
5. Эту строку нужно разделить по пробелам на три части: название интерфейса, IP-адрес и состояние.
6. Состояние интерфейса нужно разбить по слешу (`/`), чтобы получить протокол, а также состояние соединения и средства администрирования.

Ух, сколько работы — и все для того, чтобы получить ответ, который человек может определить с одного взгляда! Этот код можно оптимизировать и сделать компактнее; но в целом при захвате плохо структурированного консольного вывода придется выполнить все эти шаги. У такого метода множество недостатков, но я бы выделил следующие основные проблемы.

- **Масштабируемость.** Мы потратили столько времени на кропотливое написание инструкций для анализа вывода каждой команды. Сложно представить, сколько усилий потребуется, чтобы проделать то же самое для сотен других команд, с которыми мы обычно имеем дело.
- **Предсказуемость.** Нет никакой гарантии, что при обновлении ПО этот вывод не изменится. Даже небольшое изменение может сделать наши усилия по сбору информации бесполезными.
- **Привязка к поставщику ПО.** Наверное, это самая большая проблема. После долгого и кропотливого анализа вывода для устройства конкретного поставщика с конкретной версией ПО (в данном случае Cisco NX-OS) этот же процесс придется повторить для устройств других производителей. Не знаю, как вы, но я бы очень скептически относился к внедрению нового оборудования, если бы для этого требовалось переписывать заново весь код, захватывающий и анализирующий консольный вывод.

Теперь для сравнения посмотрим, что возвращает NX-API для той же команды `show ip interface brief`. Подробнее о том, как получить этот вывод из вашего

устройства, мы поговорим далее в этой главе. А пока сравним вывод с текстом, захваченным в предыдущем примере:


```
{
"ins_api":{
"outputs":{
"output":{
"body":{ "TABLE_intf":[
{
"ROW_intf":{
"admin-state":"up",
"intf-name":"Lo0",
"iod":84,
"ip-disabled":"FALSE",
"link-state":"up",
"prefix":"192.168.0.2",
"proto-state":"up"
}
},
{
"ROW_intf":{
"admin-state":"up",
"intf-name":"Eth2/1",
"iod":36,
"ip-disabled":"FALSE",
"link-state":"up",
"prefix":"10.0.0.6",
"proto-state":"up"
}
}
],
"TABLE_vrf":[
{
"ROW_vrf":{
"vrf-name-out":"default"
}
},
{
"ROW_vrf":{
"vrf-name-out":"default"
}
}
]
},
"code":"200",
"input":"show ip int brief",
"msg":"Success"
}
},
"sid":"eoc",
"type":"cli_show",
"version":"1.2"
}
}
```

NX-API умеет возвращать вывод в форматах XML и JSON. Здесь мы имеем дело с форматом JSON. Сразу видно, что вывод структурирован и что в языке Python его легко сохранить в виде словаря. После преобразования данных в словарь их не нужно анализировать, а можно просто извлекать значения по ключу. В этом выводе мы также видим метаданные, такие как признак успешного выполнения команды. Если команда завершается неудачей, ее отправитель получит сообщение о причине сбоя. Вам больше не нужно запоминать команду, которую выполняет программа, так как она указана в поле `input`. Здесь есть и другие метаданные, такие как версия NX-API.

Такого рода обмен информацией упрощает жизнь как производителям, так и операторам оборудования. Производители могут с легкостью передавать конфигурацию и информацию о состоянии. Добавлять новые поля, если им нужно сделать доступными дополнительные сведения в рамках той же структуры данных. Операторы могут с легкостью извлекать информацию и выстраивать вокруг нее свою инфраструктуру. Автоматизация и программируемость сетей нужны как производителям, так и операторам сетевого оборудования — и это понятно всем. Разногласия в основном возникают вокруг формата и структуры этой автоматизации. Как вы увидите далее в этой главе, понятие API охватывает много конкурирующих технологий. Например, в качестве транспортного протокола можно использовать REST API, NETCONF, RESTCONF и т. д.

В будущем рынок может прийти к некоторому единому формату данных. А пока каждый из нас может составить собственное мнение и помочь отрасли двигаться вперед.

Моделирование данных для IaC

Согласно Википедии (https://en.wikipedia.org/wiki/Data_model), у модели данных следующее определение:

> *«Модель данных — это абстрактная модель, которая организует элементы данных и стандартизирует их связи друг с другом и со свойствами объектов реального мира. Например, в модели данных может быть указано, что элемент, представляющий автомобиль, состоит из множества других элементов, которые, в свою очередь, определяют цвет и размеры автомобиля, а также его владельца».*

Процесс моделирования данных проиллюстрирован на рис. 3.1.

Рис. 3.1. Процесс моделирования данных

В нашем контексте концепция модели данных абстрактна и описывает сеть, будь то дата-центр, территория университета или WAN. Если взять структуру настоящего дата-центра, Ethernet-коммутатор канального уровня можно считать устройством с таблицей MAC-адресов, привязанных к каждому порту. Модель данных этого коммутатора описывает, как MAC-адреса должны храниться в таблице вместе с ключами и дополнительными характеристиками (представьте себе публичные и приватные VLAN) и т. д. Мы также можем подняться на уровень выше и смоделировать весь дата-центр целиком. Можем начать с количества устройств на уровне доступа, распределения и системном уровне, описать то, как они соединены и как должны вести себя в промышленном окружении. Например, если наша сеть имеет вид дерева, в модели можно указать количество соединений в каждом магистральном маршрутизаторе, сколько маршрутов они должны содержать и количество следующих переходов для каждого префикса.

Эти характеристики можно представить в формате, который будет описывать эталонное состояние, и сверяться с ними с помощью компьютерных программ.

YANG и NETCONF

YANG (*Yet Another Next Generation* — еще одно поколение next; вопреки распространенному мнению, в некоторых рабочих группах IETF присутствует чувство юмора) — это относительно новый язык моделирования сетевых данных, который в последнее время набирает обороты. Впервые он был представлен в документе RFC 6020 в 2010 году и с тех пор приобрел популярность среди производителей и операторов.

На момент написания этой книги уровень поддержки YANG у разных производителей сильно отличался. Темпы его внедрения в промышленные среды остаются относительно низкими. Но если сравнивать с другими форматами моделирования данных, он пользуется наибольшим вниманием.

YANG применяется для моделирования конфигурации сетевых устройств. Он также может представлять данные о состоянии, которыми управляет протокол NETCONF, удаленные вызовы процедур NETCONF и уведомления NETCONF. Задача этого языка — обеспечить общий уровень абстракции между используемыми протоколами, такими как NETCONF, и внутренним синтаксисом для конфигурации и администрирования, зависящим от производителя. Примеры использования YANG будут рассмотрены далее в этой главе.

Итак, мы обсудили общие концепции управления устройствами и моделирования данных с использованием API. Теперь обсудим некоторые API и платформу ACI от Cisco.

API и платформа ACI от Cisco

Компания Cisco Systems, тяжеловес на рынке сетевых технологий, не пропустила тенденции, связанные с автоматизацией сетей. У них есть свои наработки в этой области, усовершенствованные продукты, налаженные партнерские отношения и внешние приобретения. Их семейство продуктов охватывает маршрутизаторы, коммутаторы, брандмауэры, серверные решения (Unified Computing System), беспроводное оборудование, программно-аппаратные комплексы, аналитическое ПО и многое другое. Даже не знаю, с чего начать.

Эта книга посвящена Python и сетевым технологиям, поэтому в данном разделе мы ограничимся основными сетевыми продуктами Cisco, а именно:

- автоматизацией устройств Nexus с помощью NX-API;
- примерами с Cisco NETCONF и YANG;

- Cisco ACI для дата-центров;
- Cisco ACI для предприятий.

Для примеров с NX-API и NETCONF, представленных в этой главе, можно использовать либо всегда доступные лабораторные устройства Cisco DevNet (см. главу 2), либо локальную виртуальную лабораторию Cisco VIRL. Платформа ACI — отдельный продукт Cisco, поэтому она лицензируется вместе с физическими коммутаторами. Для следующих примеров с ACI я бы посоветовал использовать лаборатории DevNet или dCloud, чтобы познакомиться с ними поближе. Если вы один из счастливых сетевых инженеров с частной лабораторией ACI под рукой, используйте ее там, где это уместно.

Мы возьмем сетевую топологию из главы 2, но на этот раз одно из устройств будет работать под управлением *NX-OSv* (рис. 3.2).

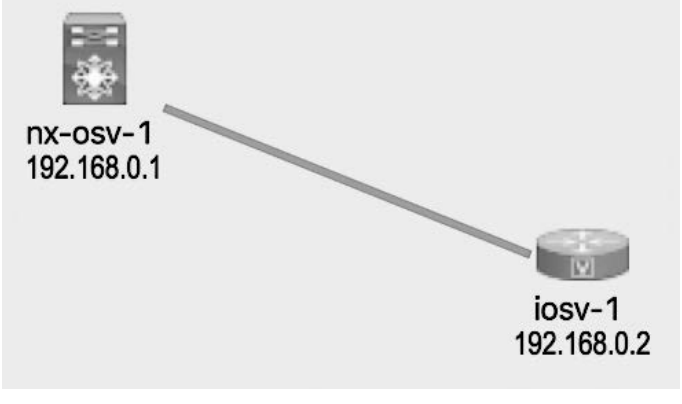


Рис. 3.2. Топология сетевой лаборатории

Рассмотрим NX-API.

Cisco NX-API

Nexus — это основная линейка коммутаторов Cisco, предназначенных для дата-центров. С помощью NX-API (http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/6-x/programmability/guide/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide_chapter_011.html) инженеры взаимодействуют с коммутатором удаленно по разным транспортным протоколам — например, SSH, HTTP и HTTPS.

Установка лабораторного ПО и подготовка устройства

Установим пакеты для Ubuntu, которые перечислены ниже. `pip` и `git` могли остаться у вас после предыдущих глав:

```
(venv) $ sudo apt-get install -y python3-dev libxml2-dev libxslt1-dev
libffi-dev libssl-dev zlib1g-dev python3-pip git python3-requests
```

Пакеты для Python 2: sudo apt-get install -y python-dev libxml2-dev libxslt1-dev libffi-dev libssl-dev zlib1g-dev python-pip git python-requests.

`ncclient` (https://github.com/ncclient/ncclient) — это клиентская библиотека NETCONF для Python. Активируйте виртуальную среду, созданную в предыдущей главе, если вы этого еще не сделали. Мы установим `ncclient` из репозитория GitHub, чтобы получить последнюю версию:

```
(venv) $ git clone github.com/ncclient/ncclient
(venv) $ cd ncclient/
(venv) $ python setup.py install
```

Поддержка NX-API на устройствах Nexus по умолчанию выключена, поэтому ее нужно включить. Для процедур NETCONF можно использовать существующую учетную запись (если применяется автоконфигурация VIRL) или создать новую:

```
feature nxapi
username cisco password 5 $1$Nk7ZkwH0$fyiRmMMfIheqE3BqvcL0C1 role
network- operator
username cisco role network-admin
username cisco passphrase lifetime 99999 warntime 14 gracetime 3
```

Мы включим в лаборатории как HTTP, так и конфигурацию изолированного окружения (в промышленных условиях они должны быть выключены!):

```
nx-osv-2(config)# nxapi http port 80
nx-osv-2(config)# nxapi sandbox
```

И далее — наш первый пример с NX-API.

Примеры с NX-API

Изолированная среда NX-API отлично подходит для экспериментов с командами, данными и форматами; мы даже можем скопировать сценарий на Python прямо с веб-страницы. Мы включили ее в конце предыдущего раздела в учебных целях. Напомню, что в промышленных условиях изолированная среда должна быть выключена.

Откроем в веб-браузере управляющий IP-адрес устройства Nexus и рассмотрим разные форматы сообщений, запросы и ответы с помощью уже знакомых нам консольных команд (рис. 3.3).

В следующем примере я выбрал `JSON-RPC` и тип `CLI` для команды `show version`. Нажмите кнопку POST (Отправить) — и вы увидите как запрос, так и ответ (рис. 3.4).

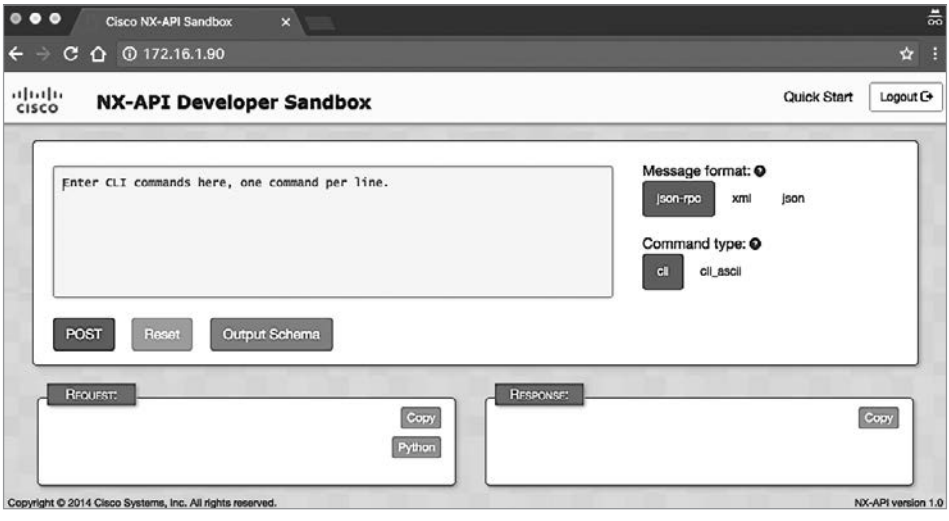


Рис. 3.3. Отладочная изолированная среда NX-API

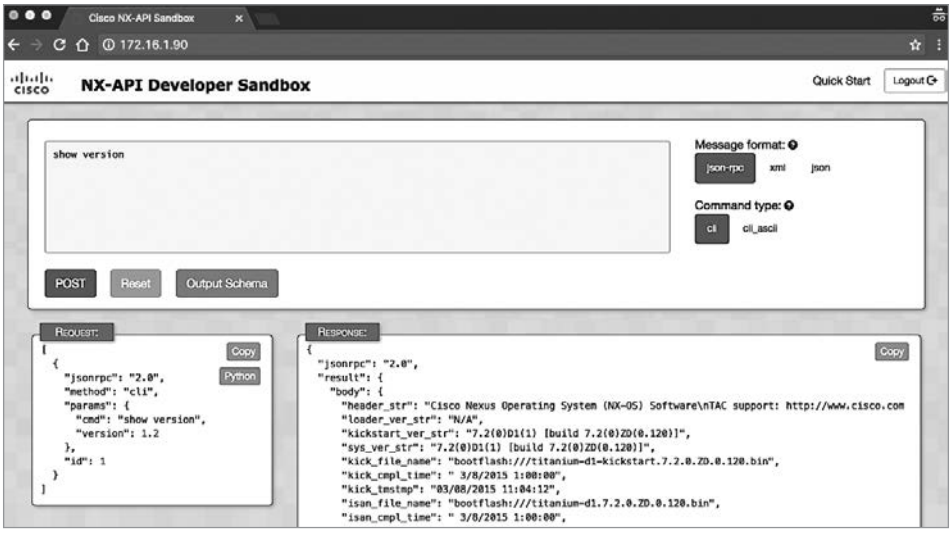


Рис. 3.4. Вывод команды в отладочной изолированной среде Cisco NX-API

Изолированная среда может помочь выяснить, поддерживается ли тот или иной формат и какие ключи можно использовать для извлечения данных из ответа в своем коде.

В первом примере, `cisco_nxapi_1.py`, мы просто подключимся к устройству Nexus и посмотрим его возможности, которые оно перечисляет, как только соединение будет установлено:

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

from ncclient import manager

conn = manager.connect(
        host='172.16.1.90',
        port=22,
        username='cisco',
        password='cisco',
        hostkey_verify=False,
        device_params={'name': 'nexus'},
        look_for_keys=False
        )

for value in conn.server_capabilities:
    print(value)

conn.close_session()
```

Думаю, параметры соединения (хост, порт, имя пользователя и пароль) не нуждаются в пояснениях. В поле `device_params` указан тип устройства, к которому подключается клиент. При обсуждении Juniper NETCONF вы увидите, что эта же библиотека может возвращать другой ответ. Параметр `hostkey_verify` позволяет обойти требование `known_host` для SSH; если его не указать, адрес вашего хоста следует записать в файл `~/.ssh/known_hosts`. Параметр `look_for_keys` отключает аутентификацию с помощью открытого и закрытого ключей и вместо них использует имя пользователя и пароль.

У некоторых пользователей возникают проблемы с Python 3 и Paramiko: https://github.com/paramiko/paramiko/issues/748. К моменту выхода второго издания эта проблема уже должна быть исправлена на стороне Paramiko.

В выводе видим, что данная версия NX-OS поддерживает XML и NETCONF:

```
(venv) $ python cisco_nxapi_1.py
urn:ietf:params:xml:ns:netconf:base:1.0
urn:ietf:params:netconf:base:1.0
urn:ietf:params:netconf:capability:validate:1.0
urn:ietf:params:netconf:capability:writable-running:1.0
urn:ietf:params:netconf:capability:url:1.0?scheme=file
```

```
urn:ietf:params:netconf:capability:rollback-on-error:1.0
urn:ietf:params:netconf:capability:candidate:1.0
urn:ietf:params:netconf:capability:confirmed-commit:1.0
```

ncclient и NETCONF по SSH приближают нас к исходной реализации и синтаксису. Эту библиотеку мы будем применять и дальше. Для работы с NX-API подходят также HTTPS и JSON-RPC. На первом снимке экрана с отладочной изолированной средой NX-API можно видеть панель REQUEST (Запрос) с кнопкой Python — ее нажатие сгенерирует сценарий на Python, использующий библиотеку `requests`.

В следующем сценарии применяется очень популярная внешняя библиотека для Python под названием requests. Ее девиз: «HTTP для людей». Она используется такими компаниями и агентствами, как Amazon, Google, NSA и др.

Для примера с командой `show version` изолированная среда NX-API сгенерировала следующий сценарий на Python. Привожу его без изменений:

```
"""
NX-API-BOT
"""
import requests
import json

"""
Modify these please
"""
url='YOURIP/ins'
switchuser='USERID'
switchpassword='PASSWORD'

myheaders={'content-type':'application/json-rpc'}
payload=[
  {
    "jsonrpc": "2.0",
    "method": "cli",
    "params": {
      "cmd": "show version",
      "version": 1.2
    },
    "id": 1
  }
]
response = requests.post(url,data=json.dumps(payload), headers=myheaders,
auth=(switchuser,switchpassword)).json()
```

Я воспользуюсь этим кодом в сценарии `cisco_nxapi_2.py` и заменю лишь URL, имя пользователя и пароль. Вот его вывод (я оставил только версию ПО):

```
(venv) $ python cisco_nxapi_2.py
7.3(0)D1(1)
```

Главное преимущество этого метода: одна и та же синтаксическая структура подходит для обеих команд, `configuration` и `show`. Это проиллюстрировано в файле `cisco_nxapi_3.py`, который изменяет имя хоста устройства с помощью командной строки. После выполнения команды вы увидите, что имя хоста поменялось с `nx-osv-1` на `nx-osv-1-new`:

```
nx-osv-1-new# sh run | i hostname
hostname nx-osv-1-new
```

Для выполнения нескольких операций их можно упорядочить с помощью поля ID. Это показано в `cisco_nxapi_4.py`. Следующее сообщение меняет описание интерфейса Ethernet 2/12 в режиме конфигурации:

```
{
  "jsonrpc": "2.0",
  "method": "cli",
  "params": {
    "cmd": "interface ethernet 2/12",
    "version": 1.2
  },
  "id": 1
},
{
  "jsonrpc": "2.0",
  "method": "cli",
  "params": {
    "cmd": "description foo-bar",
    "version": 1.2
  },
  "id": 2
},
{
  "jsonrpc": "2.0",
  "method": "cli",
  "params": {
    "cmd": "end",
    "version": 1.2
  },
  "id": 3
},
{
  "jsonrpc": "2.0",
  "method": "cli",
  "params": {
    "cmd": "copy run start",
```

```
    "version": 1.2
  },

  "id": 4
  }
]
```

Чтобы проверить результат работы этого сценария, выведем текущую конфигурацию устройства Nexus:

```
hostname nx-osv-1-new
...
interface Ethernet2/12
description foo-bar
shutdown
no switchport
mac-address 0000.0000.002f
```

В следующем разделе рассмотрим примеры использования Cisco NETCONF и модели YANG.

Модель Cisco YANG

Рассмотрим примеры того, как можно описать сеть с помощью языка моделирования данных YANG.

Модель YANG определяет только тип схемы, передаваемой по протоколу NETCONF, не уточняя, какими должны быть данные. А NETCONF — это отдельный протокол, как уже говорилось в разделе про NX-API. YANG — молодой язык, который поддерживается не всеми производителями и не во всех семействах продуктов. Например, если запустить наш сценарий, возвращающий список возможностей, для Cisco CSR 1000v под управлением IOS-XE, окажется, что это устройство поддерживает другую модель YANG:

```
urn:cisco:params:xml:ns:yang:cisco-virtual-service?module=cisco- virtual-
service&revision=2015-04-09
tail-f.com/ns/mibs/SNMP-NOTIFICATION-MIB/200210140000Z?module=SNMP-
NOTIFICATION-MIB&revision=2002-10-14
urn:ietf:params:xml:ns:yang:iana-crypt-hash?module=iana-crypt-
hash&revision=2014-04-04&features=crypt-hash-sha-512,crypt-hash-sha-
256,crypt-hash-md5
urn:ietf:params:xml:ns:yang:smiv2:TUNNEL-MIB?module=TUNNEL-
MIB&revision=2005-05-16
urn:ietf:params:xml:ns:yang:smiv2:CISCO-IP-URPF-MIB?module=CISCO-IP-URPF-
MIB&revision=2011-12-29
urn:ietf:params:xml:ns:yang:smiv2:ENTITY-STATE-MIB?module=ENTITY-STATE-
MIB&revision=2005-11-22
```

```
urn:ietf:params:xml:ns:yang:smiv2:IANAifType-MIB?module=IANAifType-
MIB&revision=2006-03-31
<опущено>
```

Сравните это с выводом для NX-OS. IOS-XE имеет более богатую поддержку модели YANG, чем NX-OS.

Поддержку общеотраслевого моделирования сетевых данных, несомненно, было бы полезно внедрить во всех ваших устройствах; это поспособствовало бы автоматизации сети. Но, учитывая неоднородную поддержку этой технологии со стороны производителей, она, по моему мнению, еще далека от того, чтобы стать единым решением для промышленных сетей. Взгляните на сценарий `cisco_yang_1.py` для контроллера Cisco APIC-EM; в нем показано, как извлечь полезные данные из вывода NETCONF XML с помощью фильтров YANG `urn:ietf:params:xml:ns:yang:ietf-interfaces`, чтобы получить список имеющихся тегов.

Последнюю информацию о поддержке YANG разными производителями можно получить на странице проекта на GitHub (https://github.com/YangModels/yang/tree/master/vendor).

Cisco ACI и APIC-EM

Технология Cisco ACI предназначена для предоставления централизованного доступа ко всем сетевым компонентам. В контексте дата-центра это означает, что у нас есть центральный контроллер, который знает о магистральных, боковых и верхних коммутаторах в стойке, управляет ими, а также поддерживает все возможности обслуживания сети. Управление может осуществляться через графический, консольный API. ACI — это ответ компании Cisco на появление более общих, программно-определяемых сетей на основе контроллеров.

Иногда бывает не совсем понятно, чем ACI отличается от APIC-EM. Если коротко, то технология ACI предназначена для использования в дата-центрах, тогда как прерогатива APIC-EM — корпоративные сети. Обе технологии позволяют централизованно просматривать и администрировать сетевые компоненты, но каждая имеет свою направленность и набор инструментов. Например, в крупных дата-центрах нечасто развертывают беспроводную, клиентоориентированную инфраструктуру, но в современных корпоративных сетях беспроводной доступ крайне важен. Еще один пример — разные подходы к сетевой безопасности. Безопасность важна в любой сети, но в дата-центрах для масштабируемости многие политики безопасности передаются на пограничные узлы.

В корпоративной среде политики безопасности часто разделяются между сетевыми устройствами и серверами.

В отличие от NETCONF RPC, API ACI следует модели REST, которая использует команды HTTP (`GET`, `POST` и `DELETE`) для описания поддерживаемых операций.

Рассмотрим файл `cisco_apic_em_1.py` — модифицированную версию демонстрационного кода из `lab2-1-get-network-device-list.py` (https://github.com/CiscoDevNet/apicem-1.3-LL-sample-codes/blob/master/basic-labs/lab2-1-get-network-device-list.py). Он в общих чертах иллюстрирует процесс взаимодействия с контроллерами ACI и APIC-EM.

Ниже представлена сокращенная версия этого кода без комментариев и пустых строк.

Первая функция, `getTicket()`, посылает контроллеру запрос HTTPS POST с путем `/api/v1/ticket` и с именем пользователя и паролем, встроенными в заголовок. Эта функция возвращает обработанный ответ с тикетом, действительный в течение ограниченного времени:

```
def getTicket():
    url = "https://" + controller + "/api/v1/ticket"
    payload = {"username":"usernae","password":"password"}
    header = {"content-type": "application/json"}
    response= requests.post(url,data=json.dumps(payload),
    headers=header, verify=False)
    r_json=response.json()
    ticket = r_json["response"]["serviceTicket"]
    return ticket
```

Затем вторая функция обращается к другому пути, `/api/v1/network-devices`, передает только что полученный тикет, встроенный в заголовок, и разбирает результаты:

```
url = "https://" + controller + "/api/v1/network-device"
header = {"content-type": "application/json", "X-Auth-Token":ticket}
```

Это распространенный подход к взаимодействию по API. При первом запросе клиент аутентифицируется на сервере и получает временный токен. Этот токен будет использоваться в последующих запросах, подтверждая прохождение аутентификации.

В ответ возвращаются документ в формате JSON и обработанная таблица. Здесь показан частичный вывод, полученный от лабораторного контроллера DevNet:

```
Network Devices =
 {
  "version": "1.0",
  "response": [
  {
   "reachabilityStatus": "Unreachable",
   "id": "8dbd8068-1091-4cde-8cf5-d1b58dc5c9c7",
   "platformId": "WS-C2960C-8PC-L",
<опущено> "lineCardId": null,
  "family": "Wireless Controller",
  "interfaceCount": "12",
  "upTime": "497 days, 2:27:52.95"
  }
 ]
}
8dbd8068-1091-4cde-8cf5-d1b58dc5c9c7 Cisco Catalyst 2960-C Series
Switches
cd6d9b24-839b-4d58-adfe-3fdf781e1782 Cisco 3500I Series Unified Access
Points
<опущено>
55450140-de19-47b5-ae80-bfd741b23fd9 Cisco 4400 Series Integrated
Services Routers
ae19cd21-1b26-4f58-8ccd-d265deabb6c3 Cisco 5500 Series Wireless LAN
Controllers
```

Как видите, обращаясь лишь к одному контроллеру, мы получаем общую информацию обо всех сетевых устройствах, которые ему известны. В данном случае мы сможем продолжить исследование коммутатора Catalyst 2960-C, точек доступа Cisco 3500I, маршрутизатора 4400 ISR и контроллера беспроводного доступа Cisco 5500. Недостаток этого подхода в том, что на сегодняшний день технологию ACI поддерживают только устройства Cisco.

Cisco IOS-XE

Сценарии Cisco IOS-XE по своим возможностям во многом похожи на тот код, который мы написали для NX-OS. Однако IOS-XE имеет дополнительные возможности, включая встроенный Python и гостевую командную оболочку, которые могут положительно повлиять на программируемость сетей: https://developer.cisco.com/docs/ios-xe/#!on-box-python-and-guestshell-quick-start-guide/onbox-python.

Помимо ACI, у Cisco есть другой продукт, Meraki, который представляет собой централизованный хост, обеспечивающий наблюдаемость в разных проводных и беспроводных сетях. Но, в отличие от контроллера ACI, Meraki размещается не локально, а в облаке. Некоторые возможности Cisco Meraki и примеры использования этого решения рассматриваются в следующем разделе.

Контроллер Cisco Meraki

Cisco Meraki — это облачный, беспроводной централизованный контроллер, который упрощает администрирование сетевых устройств. В нем используется примерно тот же подход, что и в APIC, только сам контроллер находится в облаке и имеет публичный URL. Обычно пользователь получает ключ в веб-интерфейсе и затем с его помощью извлекает ID организации в сценарии на Python:

```
#!/usr/bin/env python3
import requests
import pprint

myheaders={'X-Cisco-Meraki-API-Key': <опущено>}
url = 'https://dashboard.meraki.com/api/v0/organizations'
response = requests.get(url, headers=myheaders, verify=False)
pprint.pprint(response.json())
```

Выполним сценарий:

```
(venv) $ python cisco_meraki_1.py
 [{'id': '681155',
  'name': 'DeLab',
  'url': 'n6.meraki.com/o/49Gm_c/manage/organization/overview'},
 {'id': '865776',
  'name': 'Cisco Live US 2019',
  'url': 'n22.meraki.com/o/CVQqTb/manage/organization/overview'},
 {'id': '549236',
  'name': 'DevNet Sandbox',
  'url': 'n149.meraki.com/o/t35Mb/manage/organization/overview'},
 {'id': '52636',
  'name': 'Forest City - Other',
  'url': 'n42.meraki.com/o/E_utnd/manage/organization/overview'}]
```

По ID организации можно получить дополнительные сведения: список устройств, информацию о сети и т. д.

```
#!/usr/bin/env python3
import requests
import pprint

myheaders={'X-Cisco-Meraki-API-Key': <опущено>}
orgId = '549236'
url = 'https://dashboard.meraki.com/api/v0/organizations/' + orgId +
'/networks'
response = requests.get(url, headers=myheaders, verify=False)
pprint.pprint(response.json())

(venv) $ python cisco_meraki_2.py
```

```
<опущено>
[{'disableMyMerakiCom': False,
  'disableRemoteStatusPage': True,
  'id': 'L_646829496481099586',
  'name': 'DevNet Always On Read Only',
  'organizationId': '549236',
  'productTypes': ['appliance', 'switch'],
  'tags': ' Sandbox ',
  'timeZone': 'America/Los_Angeles',
  'type': 'combined'},
 {'disableMyMerakiCom': False,
  'disableRemoteStatusPage': True,
  'id': 'N_646829496481152899',
  'name': 'test - mx65',
  'organizationId': '549236',
  'productTypes': ['appliance'],
  'tags': None,
  'timeZone': 'America/Los_Angeles',
  'type': 'appliance'},
<опущено>
```

Если у вас нет лабораторного устройства Meraki, воспользуйтесь, как я, бесплатной лабораторией DevNet по адресу https://developer.cisco.com/learning/tracks/meraki.

Итак, мы рассмотрели примеры с контроллерами NX-API, ACI и Meraki для устройств Cisco. В следующем разделе посмотрим, как работать с устройствами Juniper Networks с помощью Python.

API на языке Python для Juniper Networks

Компания Juniper Networks всегда была популярна в кругах поставщиков услуг. Если сделать шаг назад и взглянуть на вертикаль этих поставщиков, сразу станет понятно, почему автоматизация сетевого оборудования у них в приоритете. До расцвета облачных дата-центров больше всего сетевых устройств было у поставщиков услуг. Типичная корпоративная сеть могла иметь несколько резервных интернет-соединений в штаб-квартире компании и несколько удаленных площадок, соединенных с центральным офисом звездообразным образом с помощью приватной *MPLS-сети* (*Multiprotocol Label Switching* — многопротокольная коммутация по меткам), которую предоставлял поставщик услуг. И именно поставщикам приходилось создавать, выделять, администрировать и диагностировать соединения и соответствующие сети. Их бизнес-модель основывалась

на продаже пропускной способности и дополнительных удаленных услуг. Логично, что поставщики вкладывали деньги в автоматизацию: это позволяло им минимизировать временны́е затраты на поддержание сетей в рабочем состоянии. Сетевая автоматизация стала их ключевым конкурентным преимуществом.

По моему мнению, разница потребностей сети поставщика услуг и облачного дата-центра в том, что первая традиционно агрегирует больше сервисов в каждом устройстве. Хороший пример — технология MPLS, которую предоставляют почти все крупные поставщики, но которая редко используется в корпоративных сетях и дата-центрах. Компания Juniper очень успешна в том числе и потому, что ей удалось увидеть необходимость программируемости сетей и удовлетворить потребности поставщиков в автоматизации. Давайте рассмотрим некоторые API для автоматизации от Juniper.

Juniper и NETCONF

NETCONF — это стандарт IETF, который был впервые опубликован в 2006 году в документе `RFC 4741` и последний раз обновлен в `RFC 6241`. Компания Juniper Networks сделала огромный вклад в разработку этих двух спецификаций (на самом деле она единственный автор `RFC 4741`). Поэтому логично, что устройства этого производителя получили полноценную поддержку NETCONF; к тому же эта технология играет роль внутреннего слоя для большинства инструментов и фреймворков автоматизации от Juniper. Важнейшие характеристики стандарта NETCONF:

1. Использует *расширяемый язык разметки (eXtensible Markup Language, XML)* в качестве формата данных.
2. Использует *удаленный вызов процедур (Remote Procedure Call, RPC)*, поэтому при выборе HTTP(S) в качестве транспортного протокола URL конечной точки остается неизменным, а нужная операция указывается в теле запроса.
3. На концептуальном уровне состоит из вертикально размещенных слоев, таких как содержимое, операции, сообщения и механизм передачи данных (транспорт) (рис. 3.5).

В технической библиотеке компании Juniper Networks есть обширное руководство для разработчиков по использованию протокола управления NETCONF XML (https://www.juniper.net/documentation/en_US/junos13.2/information-products/pathway-pages/netconf-guide/netconf.html#overview). Посмотрим на эту технологию в действии.

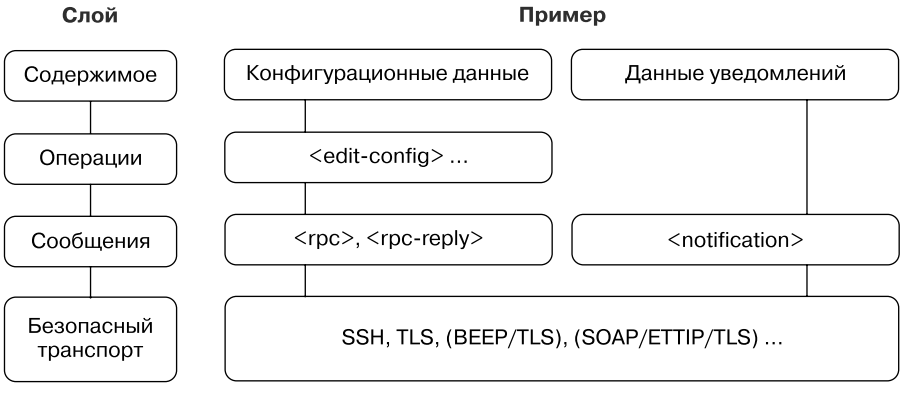


Рис. 3.5. Модель NETCONF

Подготовка устройства

Чтобы использовать NETCONF, сначала нужно создать отдельную учетную запись и включить необходимые сервисы:

```
set system login user netconf uid 2001
set system login user netconf class super-user
set system login user netconf authentication encrypted-password "$1$0EkA.
XVf$cm80A0GC2dgSWJIYWv7Pt1"
set system services ssh
set system services telnet
set system services netconf ssh port 830
```

В качестве лаборатории с устройствами Juniper я использую старую, неподдерживаемую платформу под названием JunOS Olive. Она предназначена сугубо для экспериментов. В интернете можно найти любопытные факты из истории этого продукта.

Конфигурацию устройства Juniper всегда можно посмотреть либо в виде плоского файла, либо в формате XML. Первый вариант предпочтительнее, когда необходимо изменить конфигурацию с помощью однострочной команды:

```
netconf@foo> show configuration | display set
set version 12.1R1.9
set system host-name foo set system domain-name bar
<опущено>
```

Формат XML удобен для просмотра конфигурации в структурированном виде:

```
netconf@foo> show configuration | display xml
<rpc-reply xmlns:junos="xml.juniper.net/junos/12.1R1/junos">
<configuration junos:commit-seconds="1485561328" junos:commit-
```

```
localtime="2017-01-27 23:55:28 UTC" junos:commit-user="netconf">
<version>12.1R1.9</version>
<system>
<host-name>foo</host-name>
<domain-name>bar</domain-name>
```

Инструкции по установке необходимых Linux-библиотек и пакета `ncclient` для Python перечислены в подразделе «Установка лабораторного ПО и подготовка устройства» раздела «Cisco NX-API» ранее в этой главе. Если вы их еще не установили, сделайте это сейчас.

Теперь мы готовы перейти к первому примеру с Juniper NETCONF.

Примеры с Juniper NETCONF

Вначале рассмотрим простой пример с выполнением команды `show version`. Назовем этот файл `junos_netconf_1.py`:

```
#!/usr/bin/env python3
from ncclient import manager

conn = manager.connect(
    host='192.168.24.252',
    port='830',
    username='netconf',
    password='juniper!',
    timeout=10,
    device_params={'name':'junos'},
    hostkey_verify=False)

result = conn.command('show version', format='text')
print(result.xpath('output')[0].text)
conn.close_session()
```

Названия полей в этом сценарии говорят сами за себя; одно исключение — поле `device_params`. Оно было добавлено в ncclient 0.4.1 для задания разных производителей и платформ. Например, в параметре `name` можно указать Juniper, CSR, Nexus или Huawei. Мы также добавили `hostkey_verify=False`, так как используем самозаверенный сертификат из устройства Juniper.

В ответ приходит XML-документ `rpc-reply` с элементом `output`:

```
<rpc-reply message-id="urn:uuid:7d9280eb-1384-45fe-be48- b7cd14ccf2b7">
<output>
Hostname: foo
Model: olive
JUNOS Base OS boot [12.1R1.9]
JUNOS Base OS Software Suite [12.1R1.9]
```

```
< опущено >
JUNOS Runtime Software Suite [12.1R1.9] JUNOS Routing Software Suite
[12.1R1.9]
</output>
</rpc-reply>
```

Можно извлечь из этого вывода содержимое `output`:

```
print(result.xpath('output')[0].text)
```

В файле `junos_netconf_2.py` внесем изменения в конфигурацию устройства. Сначала импортируем новые модули для создания XML-элементов и объекта, который будет управлять соединением:

```
#!/usr/bin/env python3
from ncclient import manager
from ncclient.xml_ import new_ele, sub_ele

conn = manager.connect(host='192.168.24.252', port='830',
username='netconf', password='juniper!', timeout=10, device_
params={'name':'junos'}, hostkey_verify=False)
```

Заблокируем конфигурацию и внесем в нее изменения:

```
# блокируем конфигурацию и вносим в нее изменения
conn.lock()

# формируем конфигурацию
config = new_ele('system')
sub_ele(config, 'host-name').text = 'master'
sub_ele(config, 'domain-name').text = 'python'
```

В блоке кода для формирования конфигурации мы создаем новый элемент `system` с дочерними элементами `host-name` и `domain-name`. Если вам интересно, как выглядит эта иерархия, ниже показана структура узлов XML-документа, в которой `system` — это родитель для `host-name` и `domain-name`:

```
<system>
    <host-name>foo</host-name>
    <domain-name>bar</domain-name>
...
</system>
```

Сформировав конфигурацию, сценарий передаст ее устройству и зафиксирует изменения. Рекомендованная процедура внесения изменений в конфигурацию Juniper включает шаги: `lock`, `configure`, `unlock` и `commit`:

```
# отправляем, проверяем и фиксируем конфигурацию
conn.load_configuration(config=config)
```

```
conn.validate()
commit_config = conn.commit()
print(commit_config.tostring)

# разблокируем конфигурацию
conn.unlock()

# завершаем сеанс
conn.close_session()
```

В целом этапы работы с NETCONF довольно точно соответствуют аналогичным консольным командам. В файле `junos_netconf_3.py` представлен более универсальный код. Следующий пример объединяет шаги, описанные выше, в несколько функций на Python:

```
# создаем объект соединения
def connect(host, port, user, password):
    connection = manager.connect(host=host, port=port,
        username=user, password=password, timeout=10,
        device_params={'name':'junos'}, hostkey_verify=False)
    return connection

# выполняем команду show
def show_cmds(conn, cmd):
    result = conn.command(cmd, format='text')
    return result

# отправляем конфигурацию
def config_cmds(conn, config):
    conn.lock()
    conn.load_configuration(config=config)
    commit_config = conn.commit()
    return commit_config.tostring
```

Этот файл можно использовать как самостоятельный сценарий или импортировать из других сценариев на Python.

Juniper также предоставляет библиотеку PyEZ для Python, которая позволяет работать с устройствами этого производителя. В следующем разделе рассмотрим несколько примеров ее использования.

Juniper PyEZ для разработчиков

PyEZ — это высокоуровневая библиотека, которую можно легко интегрировать в код на Python. Ее API служит оберткой для исходной конфигурации, что позволяет выполнять рутинные операции и изменять настройки устройства без обширных знаний интерфейса командной строки Junos.

Исчерпывающее руководство по Junos PyEZ для разработчиков есть в технической библиотеке Juniper: https://www.juniper.net/documentation/us/en/software/junos-pyez/junos-pyez-developer/index.html. Если вы заинтересовались PyEZ, я бы настоятельно рекомендовал вам хотя бы просто ознакомиться с освещенными в нем темами.

Установка и подготовка

Инструкции по установке PyEZ для каждой операционной системы можно найти по ссылке https://www.juniper.net/documentation/en_US/junos-pyez/topics/task/installation/junos-pyez-server-installing.html. Мы покажем, как установить эту библиотеку в Ubuntu 18.04.

Ниже перечислены пакеты-зависимости, многие из них должны были у вас остаться после предыдущих примеров:

```
(venv) $ sudo apt-get install -y python3-pip python3-dev libxml2-dev
libxslt1-dev libssl-dev libffi-dev
```

Пакеты PyEZ можно установить через `pip`:

```
(venv) $ pip install junos-eznc
```

Для работы с PyEZ на устройстве Juniper нужно сконфигурировать NETCONF в качестве внутреннего API формата XML:

```
set system services netconf ssh port 830
```

Для аутентификации используйте либо пароль, либо SSH-ключ. Создать локальную учетную запись просто:

```
set system login user netconf uid 2001
set system login user netconf class super-user
set system login user netconf authentication encrypted-password "$1$0EkA.
XVf$cm80A0GC2dgSWJIYWv7Pt1"
```

Сначала сгенерируйте пару SSH-ключей на своем управляющем хосте, если вы еще этого не сделали в главе 2:

```
$ ssh-keygen -t rsa
```

Открытый и закрытый ключи по умолчанию сохраняются в каталоге `~/.ssh/`; первый называется `id_rsa.pub`, а второй — `id_rsa`. Относитесь к закрытому ключу как к паролю, который вы никому не показываете. Открытый ключ можно свободно распространять. Здесь мы скопируем открытый ключ в каталог `/tmp`

и включим модуль HTTP-сервера из состава Python 3, чтобы создать URL, доступный снаружи:

```
(venv) $ cp ~/.ssh/id_rsa.pub /tmp
(venv) $ cd /tmp
(venv) $ python3 -m http.server
(venv) Serving HTTP on 0.0.0.0 port 8000 ...
```

Для Python 2 — команда python -m SimpleHTTPServer.

Теперь можно создать учетную запись для устройства Juniper и связать с ней наш открытый ключ, предварительно загруженный с веб-сервера из Python 3:

```
netconf@foo# set system login user echou class super-user authentication
load-key-file <IP-адрес управляющего хоста>:8000/id_rsa.pub
/var/home/netconf/...transferring.file........100% of 394 B 2482 kBps
```

Теперь, если зайти на устройство с управляющего хоста и указать наш закрытый ключ, пользователь будет аутентифицирован:

```
(venv) $ ssh -i ~/.ssh/id_rsa <Juniper device ip>
--- JUNOS 12.1R1.9 built 2012-03-24 12:52:33 UTC
echou@foo>
```

Убедимся в том, что PyEZ поддерживает оба метода аутентификации. Опробуем аутентификацию по имени пользователя и паролю:

```
>>> from jnpr.junos import Device
>>> dev = Device(host='<IP-адрес устройства Juniper, в нашем случае
192.168.24.252>',
user='netconf', password='juniper!')
>>> dev.open()
Device(192.168.24.252)
>>> dev.facts
{'serialnumber': '', 'personality': 'UNKNOWN', 'model': 'olive', 'ifd_
style': 'CLASSIC', '2RE': False, 'HOME': '/var/home/juniper', 'version_
info': junos.version_info(major=(12, 1), type=R, minor=1, build=9),
'switch_style': 'NONE', 'fqdn': 'foo.bar', 'hostname': 'foo', 'version':
'12.1R1.9', 'domain': 'bar', 'vc_capable': False}
>>> dev.close()
```

А теперь попытаемся аутентифицироваться по SSH-ключу:

```
>>> from jnpr.junos import Device
>>> dev1 = Device(host='192.168.24.252', user='echou', ssh_private_key_
file='/home/echou/.ssh/id_rsa')
>>> dev1.open()
Device(192.168.24.252)
>>> dev1.facts
```

```
{'HOME': '/var/home/echou', 'model': 'olive', 'hostname': 'foo', 'switch_
style': 'NONE', 'personality': 'UNKNOWN', '2RE': False, 'domain': 'bar',
'vc_capable': False, 'version': '12.1R1.9', 'serialnumber': '', 'fqdn':
'foo.bar', 'ifd_style': 'CLASSIC', 'version_info': junos.version_
info(major=(12, 1), type=R, minor=1, build=9)}
>>> dev1.close()
```

Отлично! Теперь все готово для выполнения примеров с PyEZ.

Примеры с PyEZ

В предыдущем интерактивном примере мы видели, что после подключения к устройству объект автоматически извлекает несколько фактов о нем. В нашем первом сценарии, `junos_pyez_1.py`, мы подключались к устройству и выполняли RPC-вызов с командой `show interface em1`:

```
#!/usr/bin/env python3
from jnpr.junos import Device
import xml.etree.ElementTree as ET
import pprint

dev = Device(host='192.168.24.252', user='juniper', passwd='juniper!')

try:
    dev.open()
except Exception as err:
    print(err)
    sys.exit(1)

result = dev.rpc.get_interface_information(interface_name='em1',
terse=True)
pprint.pprint(ET.tostring(result))

dev.close()
```

У класса `Device` есть свойство `rpc`, которое содержит все доступные команды. Это крайне удобно, так как CLI и API предоставляют одни и те же возможности. Хитрость в том, что нам нужно найти тег `xml rpc`, который соответствует нашей команде. Если взять первый пример, то откуда мы узнали, что `show interface em1` эквивалентно `get_interface_information`? Получить эту информацию мы можем тремя путями.

1. Свериться со справочником *Junos XML API Operational Developer Reference*.
2. Вывести с помощью CLI эквивалент XML RPC и подставить подчеркивания (`_`) вместо дефисов (`-`) между словами.
3. Воспользоваться библиотекой PyEZ.

Я обычно пользуюсь вторым способом:

```
netconf@foo> show interfaces em1 | display xml rpc
<rpc-reply xmlns:junos="xml.juniper.net/junos/12.1R1/junos">
 <rpc>
 <get-interface-information>
 <interface-name>em1</interface-name>
 </get-interface-information>
 </rpc>
 <cli>
 <banner></banner>
 </cli>
</rpc-reply>
```

А вот пример использования PyEZ (третий способ):

```
>>> dev1.display_xml_rpc('show interfaces em1', format='text')
'<get-interface-information>/n <interface-name>em1</interface- name>/n</
get-interface-information>/n'
```

И конечно, мы можем внести изменения в конфигурацию. Импортируем в `junos_pyez_2.py` дополнительный метод `Config()` из PyEZ:

```
#!/usr/bin/env python3
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
```

Используем тот же код для подключения к устройству:

```
dev = Device(host='192.168.24.252', user='juniper',
    passwd='juniper!')

try:
    dev.open()
except Exception as err:
    print(err)
    sys.exit(1)
```

Новый метод `Config()` загрузит XML-данные и внесет изменения в конфигурацию:

```
config_change = """
<system>
  <host-name>master</host-name>
  <domain-name>python</domain-name>
</system>
"""
cu = Config(dev)
cu.lock()
cu.load(config_change)
cu.commit()
cu.unlock()

dev.close()
```

Примеры здесь намеренно выбраны простые. Надеюсь, вы поняли, как можно использовать эту библиотеку для автоматизации устройств Junos. В следующем примере вы научитесь работать с сетевыми устройствами Arista с помощью библиотек для Python.

API на языке Python для устройств Arista

Компания *Arista Networks* всегда уделяла основное внимание сетям крупномасштабных дата-центров. На ее корпоративной странице (https://www.arista.com/en/company/company-overview) говорится следующее:

> *«Arista Networks была создана для разработки и предоставления программных облачных сетевых решений для крупных дата-центров и вычислительных окружений».*

Обратите внимание, что в этой цитате явно упоминаются *крупные дата-центры*, у которых, как мы знаем, стремительно увеличивается количество серверов, баз данных и, конечно же, сетевых устройств. И логично, что автоматизация всегда была для Arista одним из приоритетных направлений. На самом деле фирменная операционная система этой компании основана на Linux, благодаря чему в ней напрямую доступны стандартные консольные команды и интерпретатор Python. В Arista с самого начала делали акцент на том, что возможности Linux и Python должны быть доступны сетевым операторам.

Arista, как и другие производители, позволяет взаимодействовать со своими устройствами непосредственно через eAPI или с помощью их библиотеки для Python. Мы рассмотрим оба эти варианта. А в следующих главах вы узнаете, как Arista работает с фреймворком Ansible.

Работа с eAPI от Arista

Компания Arista впервые представила eAPI несколько лет назад в EOS 4.12. Это интерфейс, который передает список информационных и конфигурационных команд по HTTP или HTTPS и возвращает ответ в формате JSON. Его отличительная черта: использование RPC и *JSON-RPC* вместо чистого RESTFul API, который работает поверх HTTP или HTTPS. Вся разница в том, что запросы направляются к конечной точке с одним и тем же URL и HTTP-методом (`POST`). Но вместо использования HTTP-команд (`GET`, `POST`, `PUT`, `DELETE`) для выражения

действия мы просто указываем наши намерения в теле запроса. В случае с eAPI мы указываем ключ `method` со значением `runCmds`.

В следующих примерах я использую физический коммутатор Arista под управлением EOS 4.16.

Подготовка eAPI

Агент eAPI на устройствах Arista по умолчанию выключен, поэтому, прежде чем начинать, нам нужно его включить:

```
arista1(config)#management api http-commands
arista1(config-mgmt-api-http-cmds)#no shut
arista1(config-mgmt-api-http-cmds)#protocol https port 443
arista1(config-mgmt-api-http-cmds)#no protocol http
arista1(config-mgmt-api-http-cmds)#vrf management
```

Как видите, мы отключили протокол HTTP и передаем данные исключительно по HTTPS. Управляющие интерфейсы по умолчанию находятся в VRF с названием *management*. В моей топологии доступ к устройствам происходит через управляющий интерфейс, поэтому для работы с eAPI я указал VRF. Состояние управляющего API можно узнать с помощью команды `show management api http-commands`:

```
arista1#sh management
api http-commands Enabled: Yes
HTTPS server: running, set to use port 443 HTTP server: shutdown, set to
use port 80
Local HTTP server: shutdown, no authentication, set to use port 8080
Unix Socket server: shutdown, no authentication
VRF: management
Hits: 64
Last hit: 33 seconds ago Bytes in: 8250
Bytes out: 29862
Requests: 23
Commands: 42
Duration: 7.086
seconds SSL Profile: none
QoS DSCP: 0
User Requests Bytes in Bytes out Last hit
---------- ------------- ------------- --------------- -----------
admin 23 8250 29862 33 seconds ago

URLs
----------------------------------------
Management1 : https://192.168.199.158:443

arista1#
```

После включения агента мы получим доступ к странице обозревателя eAPI; ее можно открыть в веб-браузере, указав IP-адрес устройства. Если вы поменяли порт доступа по умолчанию, просто добавьте его в конце. Здесь применяется тот же метод аутентификации, что и в самом коммутаторе. Мы возьмем имя пользователя и пароль, сконфигурированные локально на устройстве. По умолчанию используется самозаверенный сертификат (рис. 3.6).

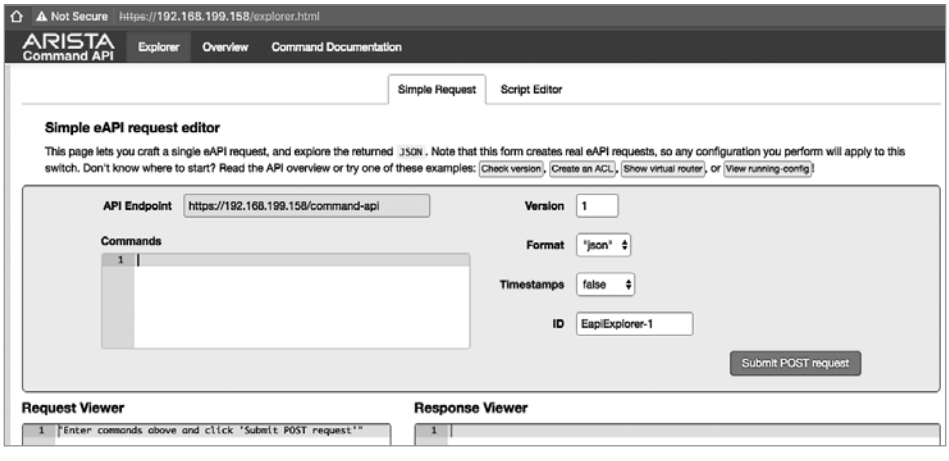


Рис. 3.6. Обозреватель Arista EOS

У вас откроется страница обозревателя, где можно ввести консольную команду и получить структурированный вывод с телом запроса. Например, если вам интересно, как создать тело запроса для команды `show version`, смотрите вывод на рис. 3.7.

На вкладке Overview (Обзор) вы найдете примеры и справочную информацию, а документация поможет вам ориентироваться в командах `show`. Для каждой команды дается название поля с возвращаемым значением и краткое описание. В сценариях, которые приводятся в справочнике Arista, применяется библиотека `jsonrpclib` (https://github.com/joshmarshall/jsonrpclib/), и мы тоже будем использовать ее в наших примерах. Однако на момент написания этой книги она не поддерживала Python 3, поэтому следующие сценарии работают с Python 2.7.

С момента выхода этой книги ситуация могла поменяться. Пожалуйста, ознакомьтесь с запросом на включение внесенных изменений (https://github.com/joshmarshall/jsonrpclib/issues/38) и с содержимым файла README на GitHub (https://github.com/joshmarshall/jsonrpclib/), чтобы узнать последние новости.

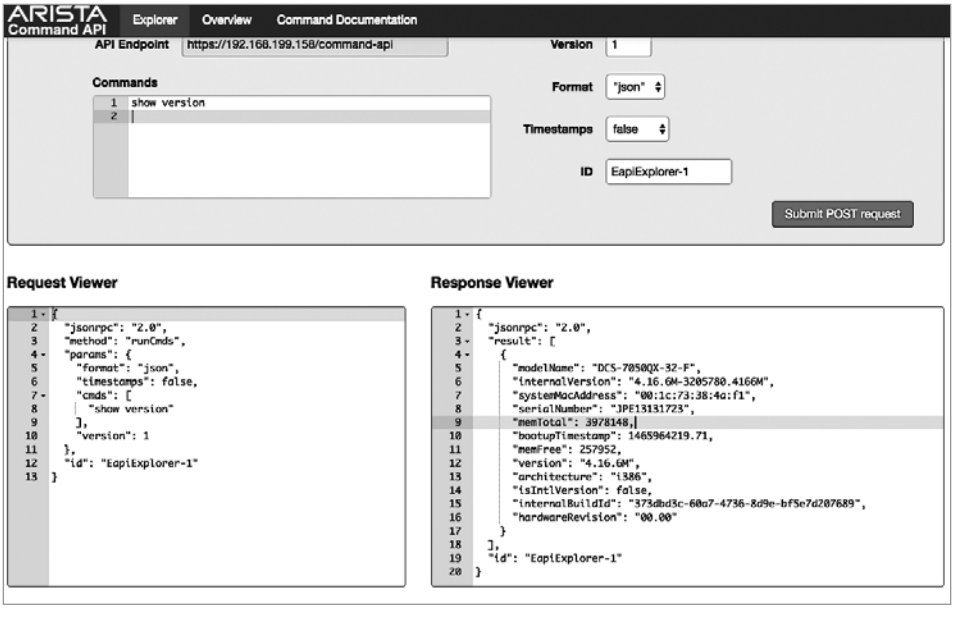


Рис. 3.7. Просмотр вывода в обозревателе Arista EOS

Для простоты установки используем `easy_install` или `pip`:

```
(venv) $ pip install jsonrpclib
```

Примеры работы с eAPI

Напишем простую программу для вывода текста ответа. Назовем ее `eapi_1.py`:

```
#!/usr/bin/python2

from __future__ import print_function
from jsonrpclib import Server
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
switch = Server("https://admin:arista@192.168.199.158/command-api")
response = switch.runCmds( 1, [ "show version" ] )
print('Serial Number: ' + response[0]['serialNumber'])
```

Поскольку это Python 2, я использую from __future__ import print_function, чтобы упростить будущую миграцию. Код, использующий ssl, требует Python версии 2.7.9 и выше. Подробнее об этом см. https://www.python.org/dev/peps/pep-0476/.

Вот какой ответ возвращает предыдущий вызов метода `runCmds()`:

```
[{u'memTotal': 3978148, u'internalVersion': u'4.16.6M- 3205780.4166M',
u'serialNumber': u'<omitted>', u'systemMacAddress': u'<omitted>',
u'bootupTimestamp': 1465964219.71, u'memFree': 277832, u'version':
u'4.16.6M', u'modelName': u'DCS-7050QX-32-F', u'isIntlVersion':
False, u'internalBuildId': u'373dbd3c-60a7-4736-8d9e-bf5e7d207689',
u'hardwareRevision': u'00.00', u'architecture': u'i386'}]
```

Как видите, это список с единственным элементом — словарем. Если нам нужно извлечь серийный номер, мы можем просто указать порядковый номер элемента и ключ:

```
print('Serial Number: ' + response[0]['serialNumber'])
```

В выводе будет только серийный номер:

```
$ python eapi_1.py
Serial Number: <опущено>
```

Чтобы больше узнать о командах, я советую перейти по ссылке Command Documentation (Документация по командам) на странице eAPI и сравнить свой вывод с выводом `show version`, который приводится в документации.

Как уже отмечалось, клиент JSON-RPC, в отличие от REST, использует для вызова серверных ресурсов конечную точку с тем же URL. В предыдущем примере видно, что метод `runCmds()` принимает список команд. Тот же способ подходит и для команд изменения конфигурации.

Ниже показан файл `eapi_2.py` с примером выполнения конфигурационных команд. Мы написали функцию, которая принимает объект `switch_object` и список команд в качестве атрибутов:

```
#!/usr/bin/python2

from __future__ import print_function
from jsonrpclib import Server
import ssl, pprint

ssl._create_default_https_context = ssl._create_unverified_context

# Выполняет команды Arista через eAPI
def runAristaCommands(switch_object, list_of_commands):
    response = switch_object.runCmds(1, list_of_commands)
    return response

switch = Server("https://admin:arista@192.168.199.158/command-api")

commands = ["enable", "configure", "interface ethernet 1/3",
```

```
"switchport access vlan 100", "end", "write memory"]

response = runAristaCommands(switch, commands)
pprint.pprint(response)
```

Вывод этого сценария:

```
$ python2 eapi_2.py
[{}, {}, {}, {}, {}, {u'messages': [u'Copy completed successfully.']}]
```

Теперь проверим, выполнилась ли команда на коммутаторе:

```
arista1#sh run int eth 1/3
interface Ethernet1/3
    switchport access vlan 100
arista1#
```

В целом интерфейс eAPI довольно понятный и простой в использовании. Для большинства языков программирования есть библиотеки наподобие `jsonrpclib`, инкапсулирующие внутренние механизмы JSON-RPC. Вы можете начать интегрировать автоматизацию Arista EOS в свою сеть с помощью всего нескольких команд.

Библиотека Arista Pyeapi

Pyeapi (http://pyeapi.readthedocs.io/en/master/index.html) — это клиентская библиотека для Python, которая служит оберткой eAPI. Она предоставляет набор функций для конфигурации узлов Arista EOS. Но зачем она нам нужна, если у нас уже есть eAPI? Ответ зависит от конкретной ситуации. Если вы уже используете Python для автоматизации, выбор между Pyeapi и eAPI — это вопрос личных предпочтений.

Но если вы не используете Python, то вам, наверное, лучше выбрать eAPI. Как видно по примерам, для работы с eAPI достаточно клиента с поддержкой JSON-RPC. Поэтому этот интерфейс доступен в большинстве языков программирования. Когда я только начинал работать в этой области, доминирующим языком для написания сценариев и автоматизации сети был Perl. Многие организации до сих пор используют сценарии на Perl в качестве основного средства автоматизации. Если компания уже вложила много ресурсов и ее кодовая база написана не на Python, хорошим выбором будет eAPI с JSON-RPC.

Но для тех, кто предпочитает программировать на Python, библиотека на этом языке позволит сделать написание кода более естественным. Это, несомненно, упрощает внедрение поддержки узлов EOS в программы на Python и обновле-

ние кода с учетом последних изменений в этом языке. Например, с Pyeapi можно использовать Python 3!

На момент написания этой книги работа над поддержкой Python 3 (3.4+) еще не завершена: http://pyeapi.readthedocs.io/en/master/requirements.html. Подробности ищите в документации.

Установка Pyeapi

Установку легко выполнить с помощью `pip`:

```
(venv) $ pip install pyeapi
```

Стоит отметить, что вместе с зависимостями для Pyeapi, перечисленными на странице http://pyeapi.readthedocs.io/en/master/requirements.html, будет установлена библиотека netaddr.

Клиент Pyeapi по умолчанию ищет в домашнем каталоге скрытый файл `eapi.conf` (с точкой в начале имени) в формате INI. Путь к этому файлу можно переопределить вручную. Вообще учетные данные для соединения рекомендуется выносить из сценария. Со списком полей в файле `eapi.conf` можно ознакомиться в документации Arista Pyeapi (http://pyeapi.readthedocs.io/en/master/configfile.html#configfile).

Я использую в своей лаборатории файл:

```
cat ~/.eapi.conf
[connection:Arista1]
host: 192.168.199.158
username: admin
password: arista
transport: https
```

Первая строка, `[connection:Arista1]`, содержит имя, которое мы будем использовать в Pyeapi-соединении; остальные поля говорят сами за себя. Вы можете сделать так, чтобы пользователю, который выполняет сценарий, этот файл был доступен только для чтения:

```
$ chmod 400 ~/.eapi.conf
$ ls -l ~/.eapi.conf
-r-------- 1 echou echou 94 Jan 27 18:15 /home/echou/.eapi.conf
```

Итак, библиотека Pyeapi установлена. Перейдем к примерам.

Примеры с Pyeapi

У нас все готово к работе с Pyeapi. Для начала подключимся к узлу EOS, создав объект в интерактивной командной оболочке Python:

```
>>> import pyeapi
>>> arista1 = pyeapi.connect_to('Arista1')
```

Пошлем этому узлу команду `show` и получим ее вывод:

```
>>> import pprint
>>> pprint.pprint(arista1.enable('show hostname'))
[{'command': 'show hostname',
'encoding': 'json',
'result': {'fqdn': 'arista1', 'hostname': 'arista1'}}]
```

Методу `config()` можно передать одну команду или список команд:

```
>>> arista1.config('hostname arista1-new')
[{}]
>>> pprint.pprint(arista1.enable('show hostname'))
[{'command': 'show hostname',
 'encoding': 'json',
 'result': {'fqdn': 'arista1-new', 'hostname': 'arista1-new'}}]
>>> arista1.config(['interface ethernet 1/3', 'description my_link'])
[{}, {}]
```

Сокращенные команды (например, `show run` вместо `show running-config`) и некоторые расширения не будут работать:

```
>>> pprint.pprint(arista1.enable('show run'))
Traceback (most recent call last):
...
File "/usr/local/lib/python3.5/dist-packages/pyeapi/eapilib.py", line
396, in send
raise CommandError(code, msg, command_error=err, output=out) pyeapi.
eapilib.CommandError: Error [1002]: CLI command 2 of 2 'show run' failed:
invalid command [incomplete token (at token 1: 'run')]
>>>
>>> pprint.pprint(arista1.enable('show running-config interface ethernet
1/3'))
Traceback (most recent call last):
...
pyeapi.eapilib.CommandError: Error [1002]: CLI command 2 of 2 'show
running-config interface ethernet 1/3' failed: invalid command
[incomplete token (at token 2: 'interface')]
```

Но вы всегда можете перехватить результат и извлечь нужное значение:

```
>>> result = arista1.enable('show running-config')
>>> pprint.pprint(result[0]['result']['cmds']['interface Ethernet1/3'])
{'cmds': {'description my_link': None, 'switchport access vlan 100':
None}, 'comments': []}
```

До сих пор мы делали то же, что и с eAPI для команд `show` и `configuration`. Pyeapi предоставляет различные API, которые упрощают этот процесс. В следующем примере мы подключаемся к узлу, вызываем VLAN API и работаем с параметрами VLAN этого устройства:

```
>>> import pyeapi
>>> node = pyeapi.connect_to('Arista1')
>>> vlans = node.api('vlans')
>>> type(vlans)
<class 'pyeapi.api.vlans.Vlans'>
>>> dir(vlans)
[...'command_builder', 'config', 'configure', 'configure_interface',
'configure_vlan', 'create', 'default', 'delete', 'error', 'get', 'get_
block', 'getall', 'items', 'keys', 'node', 'remove_trunk_group', 'set_
name', 'set_state', 'set_trunk_groups', 'values']
>>> vlans.getall()
{'1': {'vlan_id': '1', 'trunk_groups': [], 'state': 'active', 'name':
'default'}}
>>> vlans.get(1)
{'vlan_id': 1, 'trunk_groups': [], 'state': 'active', 'name': 'default'}
>>> vlans.create(10) True
>>> vlans.getall()
{'1': {'vlan_id': '1', 'trunk_groups': [], 'state': 'active', 'name':
'default'}, '10': {'vlan_id': '10', 'trunk_groups': [], 'state':
'active', 'name': 'VLAN0010'}}
>>> vlans.set_name(10, 'my_vlan_10') True
```

Убедимся в том, что на устройстве был создан интерфейс VLAN 10:

```
arista1#sh vlan
VLAN Name Status Ports
----- -------------------------------- --------- -------------------
-----
1 default active
10 my_vlan_10 active
```

API объекта EOS для Python — это пример превосходства Pyeapi над eAPI. Он предоставляет низкоуровневые атрибуты, которые инкапсулируют устройства в виде объектов, делая код более аккуратным и читаемым.

Полный и постоянно растущий список API Pyeapi есть в официальной документации: http://pyeapi.readthedocs.io/en/master/api_modules/_list_of_modules.html.

В заключение этого раздела предположим, что нам приходится часто выполнять приведенные выше шаги и что для экономии времени можно написать еще один класс на Python.

Сценарий `pyeapi_1.py`:

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

import pyeapi

class my_switch():

    def __init__(self, config_file_location, device):
        # загрузка файла конфигурации
        pyeapi.client.load_config(config_file_location)
        self.node = pyeapi.connect_to(device)
        self.hostname = self.node.enable('show hostname')[0]['result']
        ['hostname']
        self.running_config = self.node.enable('show running-config')

    def create_vlan(self, vlan_number, vlan_name):
        vlans = self.node.api('vlans')
        vlans.create(vlan_number)
        vlans.set_name(vlan_number, vlan_name)
```

Как видите, мы автоматически подключаемся к узлу и затем, установив соединение, задаем имя хоста и загружаем `running_config`. У этого класса также есть метод, который создает интерфейс VLAN с помощью VLAN API. Опробуем этот сценарий в интерактивной оболочке:

```
>>> import pyeapi_1
>>> s1 = pyeapi_1.my_switch('/tmp/.eapi.conf', 'Arista1')
>>> s1.hostname
'arista1'
>>> s1.running_config
[{'encoding': 'json', 'result': {'cmds': {'interface Ethernet27':
{'cmds':
{}, 'comments': []}, 'ip routing': None, 'interface face Ethernet29':
{'cmds': {}, 'comments': []}, 'interface Ethernet26': {'cmds': {},
'comments': []}, 'interface Ethernet24/4': h.':
<опущено>
'interface Ethernet3/1': {'cmds': {}, 'comments': []}}, 'comments': [],
'header': ['! device: arista1 (DCS-7050QX-32, EOS-4.16.6M)n!n']},
'command': 'show running-config'}]
>>> s1.create_vlan(11, 'my_vlan_11')
>>> s1.node.api('vlans').getall()
{'11': {'name': 'my_vlan_11', 'vlan_id': '11', 'trunk_groups': [],
'state':
'active'}, '10': {'name': 'my_vlan_10', 'vlan_id': '10', 'trunk_groups':
[], 'state': 'active'}, '1': {'name': 'default', 'vlan_id': '1', 'trunk_
groups': [], 'state': 'active'}}
>>>
```

Итак, мы рассмотрели сценарии на Python для взаимодействия с сетевым оборудованием трех крупнейших производителей: Cisco Systems, Juniper Networks

и Arista Networks. В следующем разделе речь пойдет о сетевой операционной системе с открытым исходным кодом, которая набирает популярность в той же области.

Пример работы с VyOS

VyOS — это полностью открытая сетевая ОС, совместимая с широким спектром оборудования, разными виртуальными машинами и облачными провайдерами (https://vyos.io/). VyOS широко поддерживают в сообществе Open Source. Многие открытые проекты используют VyOS в качестве стандартной платформы для тестирования. Рассмотрим небольшой пример с этой ОС.

Образ VyOS доступен в разных форматах: https://wiki.vyos.net/wiki/Installation. После его загрузки и инициализации установим на наш управляющий хост соответствующую библиотеку для Python:

```
(venv) $ pip install vymgmt
```

Ниже — очень простой демонстрационный сценарий `vyos_1.py`:

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

import vymgmt

vyos = vymgmt.Router('192.168.2.116', 'vyos', password='vyos')
vyos.login()
vyos.configure()
vyos.set("system domain-name networkautomationnerds.net")
vyos.commit()
vyos.save()
vyos.exit()
vyos.logout()
```

С его помощью можно изменить доменное имя системы:

```
(venv) $ python vyos_1.py
We can log in to the device to verify the change:
vyos@vyos:~$ show configuration | match domain
domain-name networkautomationnerds.net
```

В примере метод работы с VyOS напоминает методы, которые мы применяли в сценариях для закрытых устройств. Это сделано для того, чтобы нам было легче перейти на VyOS с оборудования других поставщиков.

Мы приближаемся к концу главы, но остается еще несколько библиотек, которые заслуживают внимания и за развитием которых стоит следить. О них — в следующем разделе.

Другие библиотеки

В заключение главы перечислю некоторые проекты замечательных библиотек, не привязанных к конкретному производителю. Это Nornir (https://nornir.readthedocs.io/en/stable/index.html), Netmiko (https://github.com/ktbyers/netmiko) и NAPALM (https://github.com/napalm-automation/napalm). Примеры с ними были в предыдущей главе. Почти все они немного отстают в поддержке новых платформ и возможностей. Но благодаря независимости от производителей эти инструменты пригодятся в том случае, если привязка к конкретному оборудованию нежелательна. Еще одно их преимущество: открытый исходный код, это позволит вам участвовать в разработке новых возможностей и исправлении ошибок.

Компания Cisco недавно выпустила фреймворк pyATS (https://developer.cisco.com/pyats/) и одноименную библиотеку (ранее известную как Genie). pyATS и Genie будут рассмотрены в главе 15.

Резюме

В этой главе мы обсудили различные способы взаимодействия с сетевыми устройствами от Cisco, Juniper, VyOS и Arista и методы их администрирования. Мы рассмотрели как прямое взаимодействие с помощью NETCONF и REST, так и взаимодействие с использованием фирменных библиотек PyEZ и Pyeapi. Это разные уровни абстракции, предназначенные для программного управления сетевыми устройствами без человеческого вмешательства.

В главе 4 речь пойдет о высокоуровневом фреймворке Ansible. Это открытый фреймворк автоматизации общего назначения, написанный на Python и не привязанный к конкретному производителю оборудования. С его помощью можно автоматизировать серверы, сетевые устройства, балансировщики нагрузки и многое другое. Конечно, в этой книге нас в первую очередь интересует автоматизация сетевых устройств.

4

Основы Ansible

В предыдущих двух главах вы познакомились с разными способами взаимодействия с сетевыми устройствами. В главе 2 мы обсудили библиотеки Pexpect и Paramiko, которые управляют взаимодействиями в рамках интерактивного сеанса. В главе 3 мы посмотрели на сеть с точки зрения API и намерений. Вы познакомились с API для выполнения команд, которые позволяют получать от устройства четко структурированную обратную связь. Мы также задумались о том, чего именно хотим от сети, и начали выражать свои намерения в коде.

В этой главе мы расширим идею преобразования наших намерений в требования к сети. Если вы когда-либо занимались проектированием сетей, самой сложной частью этого процесса, скорее всего, была не работа с сетевым оборудованием, а отбор бизнес-требований и их воплощение в архитектуре сети. Ваша архитектура должна решать бизнес-задачи. Например, вы можете работать в большой инфраструктурной команде, обслуживающей успешный интернет-магазин, время ответа которого возрастает в часы пик. Как определить, сеть ли причина проблемы? Если плохая отзывчивость в самом деле вызвана перегрузкой сети, какую ее часть следует обновить? Выиграет ли другая часть системы от повышения пропускной способности?

На рис. 4.1 приведена простая последовательность шагов преобразования бизнес-требований в сетевую архитектуру.

По моему мнению, автоматизацию сетей нельзя свести к выбору более быстрой конфигурации. Мы должны сосредоточиться на решении бизнес-задач, а также на точном и надежном воплощении наших намерений в работу устройства.

Рис. 4.1. Логика развертывания сети с точки зрения бизнеса

Это те цели, о которых необходимо помнить, занимаясь сетевой автоматизацией. В этой главе мы начнем знакомство с фреймворком на основе Python под названием *Ansible*, который позволяет декларировать наши намерения и еще сильнее абстрагироваться от интерфейсов API и CLI.

Эта глава охватывает такие темы, как:

- введение в Ansible;
- небольшой пример работы с Ansible;
- архитектура Ansible;
- модули Ansible для Cisco с примерами;
- модули Ansible для Juniper с примерами;
- модули Ansible для Arista с примерами.

Ansible: более декларативный фреймворк

Представьте: однажды утром вы просыпаетесь в холодном поту от кошмара о потенциальной дыре в безопасности вашей сети. Вы знаете, что ваша сеть

содержит ценные цифровые активы, которые необходимо беречь. Вы добросовестно выполняли свою работу сетевого администратора, поэтому безопасность должна быть на уровне, но вам на всякий случай хочется принять дополнительные меры по защите сетевых устройств.

Вы разбиваете эту задачу на два этапа:

- обновление ПО устройств до последней версии, что потребует:
  1) выгрузить образ в устройство;
  2) настроить устройство на загрузку с нового образа;
  3) перезагрузить устройство;
  4) проверить, как устройство работает под управлением нового образа;
- настройка подходящего списка доступа на сетевых устройствах в два шага:
  1) сформировать список доступа на устройстве;
  2) применить список доступа к сетевым интерфейсам, что в большинстве случаев подразумевает использование соответствующего раздела конфигурации.

Будучи сетевым инженером, сосредоточенным на автоматизации, вы решаете написать сценарии, чтобы надежно сконфигурировать устройства и получить результаты выполнения операций. Вы начинаете искать подходящие команды и API для каждого из перечисленных шагов, проверяете их в своей лаборатории и, наконец, развертываете их в промышленной среде. Потратив много усилий на обновление ОС и применение ACL, вы надеетесь, что написанные вами сценарии будут совместимы и со следующим поколением устройств.

Было бы здорово, если бы существовал такой инструмент, который мог бы укоротить этот цикл проектирования-разработки-развертывания.

В этой и следующей главах мы будем работать с открытым средством автоматизации под названием Ansible. Это фреймворк, который может упростить процесс перевода бизнес-логики в сетевые команды. Он умеет конфигурировать системы, развертывать программное обеспечение и управлять комбинациями задач.

Фреймворк Ansible написан на Python, это один из ведущих инструментов автоматизации для разработчиков на этом языке, и у него самый высокий уровень поддержки со стороны производителей сетевого оборудования. В опросе Python-разработчиков за 2018 год, спонсированном фондом Python Software Foundation, проект Ansible занял первое место в категории «управление конфигурацией» (рис. 4.2).

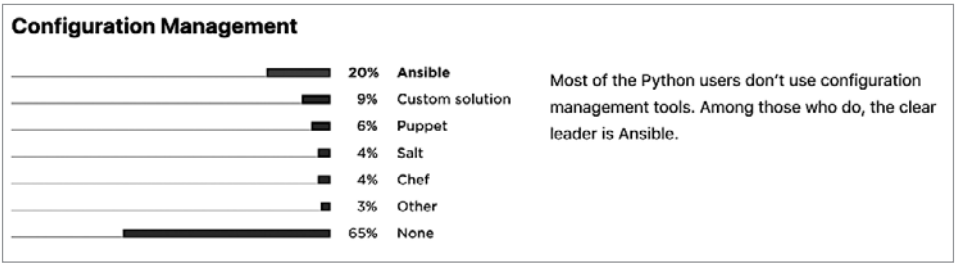


Рис. 4.2. Результаты опроса от Python Software Foundation в категории «управление конфигурацией» (источник: https://www.jetbrains.com/research/python-developers-survey-2018/)

На момент работы над этим, третьим, изданием версию Ansible 2.8 можно использовать на любом компьютере с Python 2 (версия 2.7) или Python 3 (версия 3.5 и выше). Многие полезные возможности Ansible разрабатываются сообществом в виде модулей расширения (в этом фреймворк похож на Python). И хотя основные модули уже поддерживают Python 3, многие сторонние расширения и промышленные системы все еще используют Python 2. По этой причине здесь мы будем работать с Python 2.7 и Ansible 2.8.

Версия Ansible 2.5, выпущенная в марте 2018 года, стала важной вехой в автоматизации сетей. Начиная с нее, в Ansible появилось много сетевых модулей с новыми методами соединения, синтаксисом описания задач и рекомендуемыми методиками. Учитывая, что это произошло относительно недавно, многие развернутые системы еще используют более старые версии. Чтобы обеспечить обратную совместимость, в некоторых примерах сначала демонстрируется старый формат (до версии 2.5), а затем приводится код для последней версии. В процессе будут подчеркиваться различия между версиями. К тому же переход от старого стиля к новому позволит вам лучше понять логику нововведений.

Самая актуальная информация о поддержке Python 3 в Ansible — на странице https://docs.ansible.com/ansible/latest/reference_appendices/python_3_support.html.

Я считаю, что учиться лучше на практических примерах. Как и в случае с языком Python, синтаксические конструкции Ansible легко понять даже тем, кто никогда раньше не работал с этой системой. Если вы уже знакомы с YAML или Jinja2, вы догадаетесь, какую процедуру описывает тот или иной код. Начнем с примера.

Короткий пример с Ansible

Как и другие средства автоматизации, проект Ansible изначально был нацелен на управление серверами, но со временем в число его возможностей вошло администрирование сетевого оборудования. Модули и сценарии Ansible (иногда сценарии Ansible называют плейбуками — от англ. playbook) для серверов и сетей имеют лишь небольшие различия. В этой главе мы рассмотрим пример с серверной задачей и затем сравним его с сетевыми модулями.

Установка управляющего узла

Для начала определимся с терминологией в контексте Ansible. Виртуальная машина с установленным фреймворком Ansible будет называться управляющим сервером или управляющим узлом; администрируемые компьютеры мы будем называть целевыми серверами или управляемыми узлами. Фреймворк Ansible поддерживает работу с сетевыми устройствами, на которых установлен Python версии 2.7 и выше, поэтому его можно установить в большинстве систем семейства Unix. В настоящее время официальная поддержка операционной системы Windows в качестве управляющего сервера отсутствует. Но хостами с Windows все же можно управлять с помощью Ansible; они просто не могут быть управляющими.

В Windows 10 появилась подсистема для Linux, поэтому вскоре Ansible может «научиться» работать в этой ОС. Подробности об этом можно узнать в официальной документации Ansible для Windows (https://docs.ansible.com/ansible/latest/user_guide/windows_faq.html).

Что касается требований к управляемому узлу, в документации иногда упоминается Python 2.7 и выше. Это актуально для целевых узлов с такими операционными системами, как Linux, но, конечно, не всякое сетевое оборудование поддерживает Python. Позже вы увидите, как это требование можно обойти с использованием сетевых модулей, выполняющихся на управляющем узле.

В Windows модули Ansible реализованы на PowerShell. В основном и дополнительном репозитории эти модули размещаются в подкаталоге Windows; если хотите, взгляните на них.

Установим Ansible в виртуальную машину под управлением Ubuntu. Инструкции по установке для других операционных систем смотрите в документации

(https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html). Ниже перечислены этапы установки программных пакетов:

```
$ sudo apt update
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get install ansible
```

Ansible можно установить и с помощью pip, командой pip install ansible. Лично я предпочитаю использовать системные диспетчеры пакетов, такие как Apt в Ubuntu.

Проверим успешность установки:

```
$ ansible --version
ansible 2.8.5
  config file = /etc/ansible/ansible.cfg
```

Теперь поговорим о том, как использовать разные версии Ansible на одном и том же управляющем узле. Это будет полезно тем, кто хочет поэкспериментировать с последними, еще не выпущенными изменениями, но без установки на постоянной основе. Этот метод подойдет и тогда, когда Ansible необходимо установить на управляющем узле, на котором у нас нет привилегий суперпользователя.

Установка разных версий Ansible из исходного кода

Вы можете установить Ansible из исходного кода, получив его из репозитория (об использовании Git в качестве механизма управления версиями мы поговорим в главе 13):

```
$ git clone github.com/ansible/ansible.git --recursive
$ cd ansible/
$ source ./hacking/env-setup
...
Setting up Ansible to run out of checkout...
$ ansible --version
ansible 2.10.0.dev0
  config file = /etc/ansible/ansible.cfg
...
```

В этом примере мы установили и запустили версию Ansible 2.10.0.dev0, которая отличается от системной версии 2.8.5. Чтобы установить другую версию, нужно просто выполнить команду `git checkout`, указав соответствующую ветку или тег, и снова выполнить настройку окружения:

```
$ git branch -a
$ git tag --list
$ git checkout v2.5.6
...
HEAD is now at 0c985fee8a New release v2.5.6
$ source ./hacking/env-setup
$ ansible --version
ansible 2.5.6 (detached HEAD 0c985fee8a) last updated 2019/09/23 07:05:28
(GMT -700)
  config file = /etc/ansible/ansible.cfg
```

Не исключено, что команды Git кажутся вам немного непривычными — в главе 13 вы сможете поближе познакомиться с этим инструментом.

Переключимся на версию Ansible 2.2 и обновим основной модуль:

```
$ git checkout v2.2.3.0-1
HEAD is now at f5be18f409 New release v2.2.3.0-1
$ source ./hacking/env-setup
$ ansible --version
ansible 2.2.3.0 (detached HEAD f5be18f409) last updated 2019/09/23
07:09:11 (GMT -700)
```

Git позволяет авторам проекта включать в свой репозиторий другие репозитории, которые называются подмодулями. Обновите подмодуль, чтобы синхронизироваться с текущим выпуском:

```
$ git submodule update --init --recursive Submodule 'lib/ansible/modules/
core'
(github.com/ansible/ansible-modules-core) registered for path
'lib/ansible/modules/core'
```

А теперь рассмотрим топологию лаборатории, которую мы будем использовать в этой и следующей главах.

Подготовка лаборатории

Мы будем использовать в нашей лаборатории управляющий узел с Ubuntu 18.04 и Ansible. Он имеет доступ к сетевым устройствам VIRL, а именно IOSv и NX-OSv. У нас также будет отдельная виртуальная машина с Ubuntu для примера со сценарием Ansuble, в котором целевым сервером выступает хост под управлением Linux (рис. 4.3).

Теперь рассмотрим первый пример сценария Ansible.

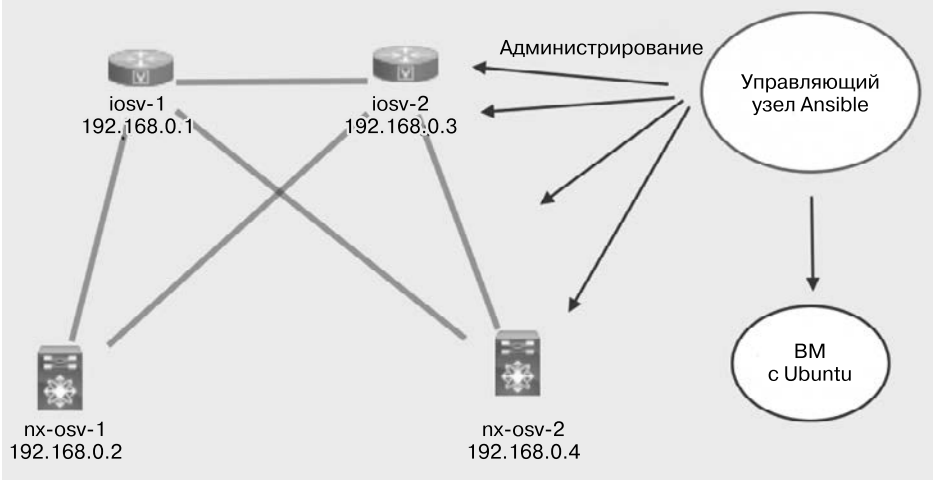


Рис. 4.3. Топология лаборатории

Ваш первый сценарий Ansible

Сценарий Ansible будет использоваться между управляющим узлом и хостом с Ubuntu. Мы выполним следующие шаги.

1. Позаботимся о том, чтобы управляющий узел мог использовать авторизацию на основе ключа.
2. Создадим файл реестра (файл со списком устройств).
3. Создадим сценарий Ansible.
4. Выполним и протестируем его.

Авторизация с открытым ключом

Первым делом нужно скопировать открытый SSH-ключ с управляющего сервера на целевой хост. Я не стану подробно описывать инфраструктуру открытых ключей (Public Key Infrastructure, PKI), но приведу инструкции для управляющего узла:

```
$ ssh-keygen -t rsa <<<< сгенерирует пару с открытым и закрытым ключами
на хосте, если вы этого еще не сделали

$ cat ~/.ssh/id_rsa.pub <<<< скопирует содержимое вывода и вставит его
в файл ~/.ssh/authorized_keys на целевом хосте в домашнем каталоге того же
```

```
пользователя; создайте файл в текстовом редакторе, таком как VI или Emac,
если он отсутствует.
```

Подробнее о PKI — на странице https://ru.wikipedia.org/wiki/Инфраструктура_открытых_ключей.

Аутентификация на основе ключа более безопасна и позволяет отключить вход по паролю на удаленном узле. Теперь управляющий узел сможет подключаться к удаленному хосту по SSH, используя закрытый ключ, и при этом у него не будут требовать пароль.

Можно ли автоматизировать начальное копирование открытого ключа? Да, но это зависит от вашей конкретной ситуации, требований и окружения. Это сравнимо с начальной настройкой сетевого оборудования в консоли для обеспечения доступности IP-адресов. Автоматизируете ли вы этот процесс и чем это обусловлено?

В следующем разделе вы увидите, как сделать целевые серверы управляемыми с помощью Ansible.

Файл реестра

Зачем нам Ansible, если у нас нет удаленных устройств, которые нужно администрировать? Все начинается с того, что нам нужно выполнить какую-то задачу на удаленном хосте. В Ansible для перечисления таких хостов используется специальный файл реестра. Можно использовать файл по умолчанию `/etc/ansible/hosts`, а можно указать другой, передав его имя в параметре `-i`. Лично я предпочитаю хранить этот файл в одном каталоге со сценарием.

В принципе, у этого файла может быть любое допустимое имя. Но традиционно его называют hosts. Используя общепринятое название, вы избавите себя и своих коллег от лишней головной боли.

Файл реестра имеет простой формат INI (https://ru.wikipedia.org/wiki/.ini), в котором перечисляются целевые хосты — либо полные доменные имена, либо IP-адреса:

```
$ cat hosts
192.168.2.122
```

В данном случае `192.168.2.122` — это IP-адрес Linux-сервера, к которому может подключиться управляющий хост Ansible. Для проверки Ansible и файла `hosts` можно использовать параметры командной строки:

```
$ ansible -i hosts 192.168.2.122 -m ping
192.168.2.122 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

По умолчанию в Ansible предполагается, что пользователь, выполняющий сценарий, имеет одноименную учетную запись на удаленном хосте. Например, мой сценарий Ansible выполняется локально от имени echou; такой же пользователь есть и на моем удаленном сервере. Чтобы использовать другую учетную запись, передайте параметр `-u` при запуске сценария Ansible, например `-u` УДАЛЕННЫЙ_ПОЛЬЗОВАТЕЛЬ.

Эта команда прочитает файл реестра и выполнит модуль `ping` на хосте с IP-адресом `192.168.2.122`. `Ping` — это элементарный тестовый модуль, который подключается к удаленному узлу, проверяет наличие подходящей версии Python и в случае успеха возвращает сообщение pong.

Если у вас возникают какие-либо вопросы о модулях, входящих в состав Ansible, можете ознакомиться с их постоянно растущим списком по ссылке https://docs.ansible.com/ansible/latest/collections/index_module.html.

Если ключ удаленного сервера не указан в файле `~/.ssh/known_hosts` управляющего хоста, вам будет предложено его добавить — ответьте `'yes'`. Чтобы отключить эту проверку, добавьте в файл `/etc/ansible/ansible.cfg` или `~/.ansible.cfg` следующий код:

```
[defaults]
host_key_checking = False
```

Итак, мы проверили пакет Ansible и файл реестра. Теперь можно создать сценарий Ansible.

Ваш первый сценарий Ansible

Сценарии описывают действия, которые вы бы хотели выполнить с хостами с помощью модулей Ansible. Именно на их написание мы, использующие Ansible,

будем тратить львиную долю нашего времени. Если провести аналогию со строительством дома, то сценарий можно сравнить с практическим руководством, модули — с инструментами, а файл реестра — с перечнем элементов дома, к которым вы будете применять свои инструменты.

Сценарии Ansible записываются на языке YAML и рассчитаны на то, чтобы их мог прочитать человек. Часто используемые синтаксические конструкции будут рассмотрены в разделе, посвященном архитектуре Ansible. А пока что сосредоточимся на примере сценария, чтобы получить представление об этой системе.

Считается, что изначально аббревиатура YAML расшифровывалась как Yet Another Markup Language («еще один язык разметки»), но теперь, согласно сайту http://yaml.org/, акроним YAML расшифровывается как Ain't Markup Language («YAML — это не язык разметки»).

Взгляните на этот простой сценарий Ansible из шести строк, `df_playbook.yml`:

```
---
- hosts: 192.168.2.122

  tasks:
    - name: check disk usage
      shell: df > df_temp.txt
```

Каждый сценарий состоит из одного или нескольких разделов. У нас всего один раздел (строки со второй по шестую) с одной задачей (строки с четвертой по шестую). Поле `name` описывает назначение задачи на понятном для человека языке, а `shell` определяет модуль, который будет использоваться. Наш модуль принимает один аргумент, `df`, интерпретирует его как команду и выполняет ее на удаленном хосте. В этом примере выполняется команда `df`, чтобы проверить использование диска, и ее вывод копируется в файл `df_temp.txt`.

Для запуска сценария Ansible можно использовать следующий код:

```
$ ansible-playbook -i hosts df_playbook.yml
PLAY [192.168.2.122] ***************************************************
*******
TASK [setup] ***********************************************************
*******
ok: [192.168.2.122]
TASK [check disk usage] ************************************************
*******
changed: [192.168.2.122]
PLAY RECAP *************************************************************
*******
192.168.2.122 :            ok=2       changed=1        unreachable=0
failed=0
```

Если зайти на управляемый хост (в нашем случае `192.168.2.122`), можно увидеть файл `df_temp.txt` с выводом команды `df`. Классно, правда?

В выводе можно заметить, что на самом деле было выполнено две задачи, хотя в сценарии указана только одна; модуль `setup` добавляется по умолчанию. Ansible выполняет его для сбора информации об удаленном хосте, которая может пригодиться позже. Например, модуль `setup` определяет тип операционной системы. Для чего это делается? Эти сведения можно использовать в качестве условий для дополнительных задач в том же сценарии. Например, вы можете предусмотреть задачу, устанавливающую пакеты. Узнав тип ОС, Ansible сможет установить нужные пакеты с помощью `apt`, если это система из семейства Debian, или `yum`, если система основана на Red Hat.

Если вам интересно увидеть вывод модуля setup, чтобы узнать, какую информацию собирает Ansible, выполните $ ansible -i hosts <host> -m setup.

При выполнении нашей простой задачи внутри Ansible происходит несколько интересных вещей. В ходе выполнения сценария управляющий узел копирует модуль на языке Python на удаленный хост, выполняет его и сохраняет вывод во временный файл, который впоследствии удаляется после захвата вывода. Мы можем спокойно игнорировать эти детали, пока они нам не понадобятся.

Мы должны разобраться в процессе, через который только что прошли, так как мы еще будем возвращаться к его элементам в этой главе. Я сознательно выбрал пример с сервером, потому что это поможет вам лучше понять, как работают сетевые модули, когда нам придется применять немного другой подход (я уже упоминал, что интерпретатора Python, скорее всего, нет на том сетевом оборудовании, которое мы хотим администрировать).

Поздравляю вас с созданием первого действующего сценария Ansible! Позже мы подробнее рассмотрим архитектуру этой системы, а пока поговорим о том, благодаря чему она подходит для сетевого администрирования. Как вы помните, модули Ansible написаны на Python; это можно считать преимуществом для сетевого инженера-питониста, не так ли?

Преимущества Ansible

Помимо Ansible, существует множество других фреймворков для автоматизации инфраструктуры — а именно, Chef, Puppet и SaltStack. Каждый из них предлагает уникальные возможности и модели; нет какого-то универсального реше-

ния, которое бы подошло для любой организации. В этом разделе я бы хотел перечислить некоторые преимущества Ansible по сравнению с альтернативами и объяснить, почему я считаю, что это хорошее средство автоматизации сетей.

При описании преимуществ Ansible я не буду рассматривать другие фреймворки. Они могут использовать некоторые принципы или определенные аспекты Ansible, но редко какая альтернатива может похвастаться всеми возможностями, которые я здесь опишу. Как мне кажется, именно сочетание следующих характеристик и философии делает Ansible идеальным решением для автоматизации сетей.

Отсутствие агентов

В отличие от других подобных инструментов Ansible не полагается на жесткую модель «ведущий — ведомый». На клиентский компьютер не нужно устанавливать никакого ПО или агентов, которые общаются с сервером. Если не считать интерпретатора Python, который по умолчанию есть на многих платформах, вам не понадобятся дополнительные программные средства.

Для применения необходимых изменений на удаленном хосте Ansible использует в своих модулях сетевой автоматизации не агенты, а SSH или API. Это только снижает необходимость в интерпретаторе Python. Данный факт крайне важен в контексте управления сетевыми устройствами, потому что производители, как правило, неохотно внедряют стороннее программное обеспечение в свои платформы. А вот SSH, к примеру, уже присутствует на любом сетевом оборудовании. В последние несколько лет ситуация немного изменилась, но в целом SSH является общим знаменателем для всех сетевых устройств, в отличие от агентов управления конфигурацией. Как вы можете помнить из главы 3, более новые сетевые устройства предоставляют слой API, который Ansible тоже может использовать.

Поскольку на удаленном хосте нет агента, Ansible самостоятельно доставляет изменения на устройство, не дожидаясь, пока клиентское ПО загрузит их с главного сервера. Эта модель, по моему мнению, более предсказуемая, так как все действия инициируются управляющим хостом. Для сравнения клиенты могут запрашивать изменения с разной частотой, что приводит к расхождениям между ожидаемым и фактическим временем выполнения операций.

Важность отсутствия агентов сложно переоценить, когда речь идет о работе с уже имеющимся сетевым оборудованием. Именно по этой причине многие сетевые администраторы и производители внедряют поддержку Ansible.

Идемпотентность

Согласно «Википедии», идемпотентность — это свойство объекта или операции в математике и информатике при многократном применении операции давать один и тот же результат (https://ru.wikipedia.org/wiki/Идемпотентность). Это означает, что выполнение процедуры меняет систему только в первый раз. Платформа Ansible пытается быть идемпотентной, что полезно для сетевых операций, которые должны выполняться в определенном порядке.

Преимущества идемпотентности очевидны, если вспомнить сценарии Pexpect и Paramiko, которые мы писали. Эти сценарии отправляли команды так, будто за терминалом сидел живой инженер. Если выполнить их десять раз, они внесут десять изменений. Если реализовать ту же задачу в виде сценария Ansible, она будет выполнена только в том случае, если эти изменения еще не вносились. Для этого сначала проверяется текущая конфигурация устройства. Если выполнить сценарий Ansible десять раз, изменения будут применены только при первом выполнении, а в остальных девяти случаях система их проигнорирует.

Идемпотентность позволяет многократно выполнять сценарий Ansible без риска внести ненужные изменения. Это важно, поскольку нам нужно автоматически обеспечивать согласованность состояния без лишних затрат.

Простота и расширяемость

Фреймворк Ansible написан на Python, а сценарии для него пишутся на YAML. Оба эти языка считаются простыми для изучения. Помните синтаксис Cisco IOS? Это предметно-ориентированный язык (Domain-Specific Language, DSL), который годится только для управления устройствами с Cisco IOS или другим оборудованием подобного рода; это не язык общего назначения, и его нельзя применять за рамками данной области. К счастью, в отличие от некоторых других средств автоматизации, Ansible не требует изучения дополнительных DSL, поскольку YAML и Python — языки общего назначения и широко применяются.

Как вы видели в предыдущем примере, даже не зная YAML, легко догадаться, для чего предназначен тот или иной сценарий. Ansible также использует Jinja2 в качестве механизма шаблонов; это популярный инструмент, который применяется в веб-фреймворках на Python, например Django и Flask. Таким образом, приобретенные здесь знания пригодятся и в других областях.

Еще раз хочу обратить ваше внимание на расширяемость Ansible. Как упоминалось в предыдущем примере, этот проект изначально задумывался для

автоматизации управления конфигурациями серверов (в основном Linux). Затем появилась поддержка серверов под управлением Windows, реализованная с помощью PowerShell. И чем больше людей в этой индустрии начали внедрять Ansible, тем актуальнее становилась тема автоматизации сетей.

Этому способствовали три факта: компания Ansible наняла подходящих людей, сетевые специалисты заинтересовались этой технологией и клиенты начали требовать от поставщиков ее поддержки. С выходом Ansible 2.0 к автоматизации сетей начали относиться не менее серьезно, чем к администрированию серверов. Экосистема жива и здорова и с каждым выпуском улучшается.

У Ansible, как и у Python, есть дружественное сообщество, открытое для новых пользователей и идей. Я когда-то сам был новичком и пытался разобраться в процедуре добавления своего кода в репозиторий в надежде написать модули, которые включили бы в основную ветку Ansible. Все мои идеи были встречены с энтузиазмом и уважением.

Простота и расширяемость очень полезны при создании прототипов. Мир технологий стремительно меняется, и мы постоянно пытаемся к нему адаптироваться. Было бы здорово, если бы мы могли выучить какую-то технологию и применять ее вне зависимости от последних тенденций. Конечно, никто не может предсказывать будущее, но, как показывает история Ansible, эта система не должна испытывать проблем с внедрением будущих технологий.

Поддержка от производителей сетевого оборудования

Посмотрим правде в глаза: мы живем не в вакууме. В нашей индустрии часто шутят, что в модель OSI должно входить еще два уровня — деньги и политика. Ежедневно мы работаем с сетевым оборудованием разных производителей.

Возьмем, к примеру, интеграцию API. В предыдущих главах мы видели, чем API отличаются от Pexpect. Очевидно, они имеют преимущество, когда речь идет об автоматизации сети. Однако производителям их внедрение обходится недешево. Каждый производитель вкладывает время, деньги и инженерные ресурсы в интеграцию API. К счастью, как видно по непрерывно растущему списку доступных сетевых модулей, Ansible поддерживают все крупные поставщики (https://docs.ansible.com/ansible/2.9/modules/list_of_network_modules.html).

Почему производители поддерживают Ansible охотнее, чем другие средства автоматизации? Один из факторов, несомненно, отсутствие агентов, так как

наличие единственной зависимости в виде SSH существенно снижает порог вхождения. Инженеры, работавшие на стороне производителей, знают, что процесс добавления новой возможности обычно растягивается на месяцы и встречает множество преград. Каждая новая функция требует времени на регрессионное тестирование, проверку совместимости, анализ интеграции и многое другое. Упрощение этой процедуры обычно служит первым шагом на пути к получению поддержки производителей.

Еще в этом процессе важно, что система Ansible основана на Python — языке, любимом многими сетевыми специалистами. Такие компании, как Juniper и Arista, которые уже вложили ресурсы в PyEZ и Pyeapi, могут с легкостью применять существующие модули на Python и быстро интегрировать свои возможности в Ansible. В главе 5 будет показано, как можно легко писать свои модули, используя знания о Python.

Еще до переориентации на сетевые технологии в Ansible было много модулей, разработанных сообществом. Уже в некоторой степени налажен процесс присоединения к проекту — по крайней мере настолько, насколько это возможно в мире открытого ПО. Основная команда Ansible привыкла к работе с сообществом и охотно принимает изменения со стороны.

Еще одна причина поддержки со стороны производителей сетевого оборудования в том, что Ansible позволяет им демонстрировать их сильные стороны в контексте модулей.

В следующем разделе вы увидите, что модули Ansible можно выполнять не только по SSH, но и локально и они способны взаимодействовать с устройствами с помощью API. Благодаря этому производители могут реализовывать свои самые свежие и лучшие возможности сразу после того, как те становятся доступными через API. С точки зрения сетевых специалистов это означает, что при использовании Ansible в качестве платформы автоматизации вы можете выбирать самое передовое оборудование.

Мы подробно обсудили поддержку со стороны поставщиков, так как мне кажется, что этот аспект Ansible получает недостаточно внимания. Производители оборудования готовы сделать ставку на определенный инструмент — и вы как сетевой инженер можете спать спокойно и быть уверены в том, что следующая прорывная сетевая технология с большой вероятностью будет поддерживать Ansible и, когда вашу сеть придется расширять, вы не будете привязаны к вашему текущему производителю.

Итак, мы обсудили преимущества системы Ansible. Теперь исследуем ее архитектуру.

Архитектура Ansible

Ansible состоит из сценариев, разделов и задач. Рассмотрим файл `df_playbook.yml`, который мы использовали ранее (рис. 4.4).

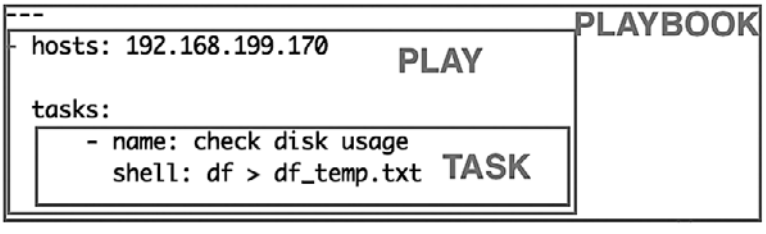


Рис. 4.4. Сценарий Ansible

Этот файл в целом называется сценарием и содержит один или несколько разделов. Каждый раздел может состоять из одной или нескольких задач. В нашем простом примере всего один раздел с одной задачей. Далее мы рассмотрим компоненты и термины, относящиеся к Ansible; некоторые вы уже видели.

- **YAML.** Этот формат активно используется в Ansible для описания сценариев и переменных.
- **Файл реестра.** Файл, в котором можно указывать и группировать хосты вашей инфраструктуры. При желании в этом файле можно также задать переменные `host` и `group`.
- **Переменные.** Каждое сетевое устройство уникально. У него есть сетевое имя, IP-адрес, отношения с соседними узлами и т. д. Переменные позволяют учитывать эти различия при создании стандартного набора задач.
- **Шаблоны.** В мире сетевых технологий шаблоны — это не новинка. Вы наверняка используете такой, даже не подозревая об этом. Что мы обычно делаем, когда нам нужно выделить новое устройство или заменить бракованное оборудование? Мы копируем старую конфигурацию и редактируем такие ее участки, как сетевое имя и IP-адреса. Ansible предлагает стандартный формат шаблонов на основе Jinja2, который мы подробно рассмотрим позже.

В главе 5 мы углубимся в такие темы, как условные выражения, циклы, блоки, обработчики, роли в сценариях, и посмотрим, как все это можно интегрировать в сетевое администрирование.

YAML

Язык YAML используется в сценариях и некоторых других файлах Ansible. В документации YAML есть полная спецификация его синтаксиса. Вот ее краткая версия в контексте ее типичного применения в Ansible:

- файл YAML начинается с трех дефисов (`---`);
- как и в Python, отступы используются для обозначения структур путем их выравнивания друг относительно друга;
- комментарии начинаются с решетки (`#`);
- каждый элемент списка занимает отдельную строку и начинается с дефиса (`-`);
- списки также можно оформлять с помощью квадратных скобок (`[]`), разделяя элементы запятыми (`,`);
- словари имеют вид пар «ключ – значение» с двоеточием в качестве разделителя;
- словари можно оформлять с помощью фигурных скобок, разделяя элементы запятыми (`,`);
- строки могут заключаться в одинарные или двойные кавычки, но не обязательно.

YAML легко преобразуется в JSON и Python. Если перевести `df_playbook.yml` в `df_playbook.json`, результат будет выглядеть так:

```
[
  {
    "hosts": "192.168.199.170",
    "tasks": [
      {"name": "check disk usage"},
      {"shell": "df > df_temp.txt"}
    ]
  }
]
```

Это, конечно, не настоящий сценарий Ansible, но с его помощью мы можем сориентироваться в формате YAML, сравнивая его с JSON. Сценарии Ansible в основном состоят из комментариев (`#`), списков (`-`) и словарей («ключ – значение»).

Файлы реестров

По умолчанию Ansible ищет хосты, указанные в сценарии, в файле `/etc/ansible/hosts`. Как уже упоминалось, я считаю, что явная передача файла с помощью

параметра `-i` выглядит более выразительно. Именно этот подход мы использовали до сих пор. Вернемся к предыдущему примеру и запишем файл реестра с хостами:

```
[ubuntu]
192.168.2.122
[nexus]
172.16.1.142
172.16.1.143

[nexus:vars]
username=cisco
password=cisco

[nexus_by_name]
switch1 ansible_host=172.16.1.142
switch2 ansible_host=172.16.1.143
```

Как вы уже могли догадаться, в квадратных скобках указаны названия групп, на которые можно ссылаться в сценариях Ansible. Например, в `cisco_1.yml` и `cisco_2.yml` я могу работать со всеми хостами, перечисленными в группе `nexus`:

```
---
- name: Configure SNMP Contact
  hosts: "nexus"
  gather_facts: false
  connection: local
  <опущено>
```

Хост может входить сразу в несколько групп. Группы могут включать другие группы, если обозначить их как `children`:

```
[cisco]
router1
router2

[arista]
switch1
switch2

[datacenter:children]
cisco
arista
```

В этом примере группа `datacenter` включает все члены групп `cisco` и `arista` — всего четыре устройства.

В следующем разделе речь пойдет о переменных. Переменные можно объявлять в нескольких местах, и вы уже видели некоторые примеры с ними. В нашем первом файле реестра мы объявили переменные как для хостов, так и для групп.

В [nexus:vars] определяются переменные для всей группы nexus, а ansible_host определяет переменную для хоста, указанного в той же строке.

Больше о файлах реестров читайте в официальной документации (https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html).

Переменные

Мы начали обсуждать переменные. Но зачем они нам нужны? Наши управляемые узлы не совсем идентичные, поэтому нужно как-то учитывать различия между ними. Имена переменных могут состоять из букв, цифр и подчеркиваний и должны начинаться с буквы. Переменные обычно определяются в трех местах:

- в сценарии Ansible;
- в файле реестра;
- в отдельных файлах, которые подключаются к другим файлам и ролям.

Рассмотрим пример определения переменных в сценарии cisco_1.yml:

```
---
- name: Configure SNMP Contact
  hosts: "nexus"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ inventory_hostname }}"
      username: cisco
      password: cisco
      transport: cli

  tasks:
    - name: configure snmp contact
      nxos_snmp_contact:
        contact: TEST_1
        state: present
        provider: "{{ cli }}"

      register: output

    - name: show output
      debug:
        var: output
```

В разделе vars определяется переменная cli, на которую ссылается задача nxos_snmp_contact, используя для этого двойные фигурные скобки ("{{ cli }}").

Подробности о модуле nxos_snmp_contact ищите в онлайн-документации (https://docs.ansible.com/ansible/latest/collections/cisco/nxos/nxos_snmp_contact_module.html).

На переменную можно сослаться с помощью двойных фигурных скобок; этот синтаксис принят в языке шаблонов Jinja2. Ссылку в фигурных скобках можно не заключать в кавычки, если с нее не начинается значение, но я все равно это делаю, чтобы не отвлекаться на эти отличия.

Вы также видели ссылку `{{ inventory_hostname }}`, которая не была объявлена в сценарии. Это одна из стандартных переменных, которые Ansible вставляет автоматически. Она ссылается на IP-адрес или имя хоста (прописанное в DNS) в файле реестра. Иногда в документации такие переменные называют волшебными (англ. magic).

Волшебных переменных не так много. Их список смотрите в документации (http://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#magic-variables-and-how-to-access-information-about-other-hosts).

Мы уже объявляли переменные в файле реестра в предыдущем разделе:

```
[nexus:vars]
username=cisco
password=cisco

[nexus_by_name]
switch1 ansible_host=172.16.1.142
switch2 ansible_host=172.16.1.143
```

Чтобы не объявлять их в сценарии, добавим в том же файле переменные для группы `[nexus_by_name]`:

```
[nexus_by_name]
switch1 ansible_host=172.16.1.142
switch2 ansible_host=172.16.1.143

[nexus_by_name:vars]
username=cisco
password=cisco
```

Затем сошлемся на них, как показано в следующей версии сценария `cisco_2.yml`, представленной ниже:

```
---
- name: Configure SNMP Contact
  hosts: "nexus_by_name"
```

```
gather_facts: false
connection: local

vars:
  cli:
    host: "{{ ansible_host }}"
    username: "{{ username }}"
    password: "{{ password }}"
    transport: cli

tasks:
  - name: configure snmp contact
    nxos_snmp_contact:
      contact: TEST_1
      state: present
      provider: "{{ cli }}"

    register: output

  - name: show output
    debug:
      var: output
```

Обратите внимание: в этом примере мы ссылаемся на группу `[nexus_by_name]` из файла реестра, на переменную хоста `ansible_host`, а также на групповые переменные `username` и `password`.

Можно вынести имя пользователя и пароль в отдельный файл, защищенный от записи, и улучшить тем самым безопасность.

Дополнительные примеры использования переменных смотрите в документации Ansible (https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html).

Для доступа к сложным переменным, организованным в виде вложенной структуры данных, можно использовать два метода. В задаче `nxos_snmp_contact` мы сохраняем вывод в переменной и выводим его на экран с помощью модуля debug.

При выполнении сценария вы увидите примерно такой вывод:

```
$ ansible-playbook -i hosts cisco_2.yml
TASK [show output] ************************************************
*******
ok: [switch1] => {
    "output": {
        "changed": false,
        "end_state": {
            "contact": "TEST_1"
        },
        "existing": {
```

```
            "contact": "TEST_1"
        },
        "proposed": {
            "contact": "TEST_1"
        },
        "updates": []
    }
}
```

Для доступа к вложенным данным можно использовать следующий синтаксис, как показано в файле `cisco_3.yml`:

```
tasks:
  - name: configure snmp contact
    nxos_snmp_contact:
      contact: TEST_1
      state: present
      provider: "{{ cli }}"

    register: output

  - name: show output in output["end_state"]["contact"]
    debug:
      msg: '{{ output["end_state"]["contact"] }}'

  - name: show output in output.end_state.contact
    debug:
      msg: '{{ output.end_state.contact }}'
```

Этот сценарий выведет только указанное значение:

```
$ ansible-playbook -i hosts cisco_3.yml
TASK [show output in output["end_state"]["contact"]]
***************************
ok: [switch1] => {
    "msg": "TEST_1"
}
ok: [switch2] => {
    "msg": "TEST_1"
}

TASK [show output in output.end_state.contact] **********************
***********
ok: [switch1] => {
    "msg": "TEST_1"
}
ok: [switch2] => {
    "msg": "TEST_1"
}
```

И напоследок: как мы уже говорили, переменные можно хранить в отдельном файле. Но прежде, чем показать их использование в подключаемых файлах или

ролях, рассмотрим еще парочку примеров, поскольку это не самые простые приемы и с них лучше не начинать. Больше примеров с ролями — в главе 5.

Шаблоны Jinja2

В предыдущем разделе для ссылки на переменные мы использовали синтаксис Jinja2 вида `{{ переменная }}`. Jinja2 позволяет делать много сложных вещей, но, к счастью, для знакомства с шаблонами Ansible нам хватит основных возможностей этого механизма шаблонов.

Jinja2 (http://jinja.pocoo.org/) — это мощный механизм шаблонов, разработанный сообществом Python. Он широко используется в таких веб-фреймворках на этом языке, как Django и Flask.

Пока что просто запомните, что Ansible использует Jinja2 как механизм шаблонов. По мере необходимости мы будем знакомиться с фильтрами, тестами и инструкциями поиска, которые поддерживает Jinja2.

Подробнее о Jinja2 в Ansible читайте на странице https://docs.ansible.com/ansible/latest/user_guide/playbooks_templating.html.

На этом мы завершаем краткий обзор архитектуры Ansible. В следующем разделе рассмотрим модули для работы с сетью, с помощью которых решается бóльшая часть сетевых задач в Ansible.

Сетевые модули Ansible

Проект Ansible создавался для управления узлами с полноценными операционными системами, такими как Linux и Windows, и уже позже в нем появилась поддержка сетевого оборудования. Вы уже могли заметить расхождения в сценариях для сетевых устройств, например в строках `gather_facts: false` и `connection: local`. В следующих разделах мы разберем эти различия.

В документации Ansible есть хорошая статья об отличиях автоматизации сетей: https://docs.ansible.com/ansible/latest/network/getting_started/network_differences.html.

Локальные соединения и факты

Модули Ansible — это код на языке Python, который по умолчанию выполняется на удаленном хосте. Большая часть сетевого оборудования не предоставляет прямого доступа к интерпретатору Python или попросту его не содержит, поэтому сценарии Ansible почти всегда выполняются локально, на управляющем узле. Это означает, что сначала сценарий интерпретируется в локальной системе, а затем необходимые команды или изменения в конфигурации отправляются целевому устройству.

Факты об удаленном узле в нашем примере с сервером собирались с помощью модуля `setup`, добавленным по умолчанию. Поскольку сценарий Ansible выполняется локально, этот модуль собирает информацию не об удаленном, а о локальном хосте. Это явно не то, что нам нужно, поэтому при локальном соединении мы можем опустить этот шаг, присвоив полю `gather_facts` значение `no` или `false`. Начиная с версии 2.5, для каждой платформы предоставляется свой модуль сбора фактов. Ознакомьтесь с примером `fact-demo.yml` по ссылке https://docs.ansible.com/ansible/latest/network/user_guide/network_best_practices_2.5.html#step-2-creating-the-playbook.

Сетевые модули выполняются локально — и те из них, которые поддерживают резервное копирование, сохраняют свои файлы, в том числе и на управляющем узле.

Важное нововведение в Ansible 2.5 — поддержка разных сетевых протоколов (https://docs.ansible.com/ansible/latest/network/getting_started/network_differences.html#multiple-communication-protocols).

В качестве метода соединения теперь можно указать `network_cli`, `netconf`, `httpapi` или `local`. Если сетевое устройство использует CLI-интерфейс через SSH, в одной из переменных для этого устройства следует выбрать метод `network_cli`. Полезно ориентироваться как в старом, так и в новом синтаксисе соединений. В целом синтаксис, появившийся в версии 2.5, выглядит более лаконичным.

Переменная provider

Как мы уже видели в главах 2 и 3, к сетевому оборудованию можно подключаться через SSH и API в зависимости от платформы и версии программного обеспечения. Все основные сетевые модули предоставляют переменную `provider` с набором значений, описывающих метод подключения к сетевому устройству. Некоторые модули поддерживают только `cli`, другие поддерживают дополни-

тельные значения; например, Arista поддерживает eAPI, а Cisco — NX-API (на платформе Nexus).

С выходом Ansible 2.5 метод подключения рекомендуется указывать в переменной `connection`. В будущих выпусках Ansible аргумент `provider` будет выходить из употребления. Как можно видеть в примере с модулем `ios_command` (https://docs.ansible.com/ansible/latest/modules/ios_command_module.html#ios-command-module), этот аргумент по-прежнему работает, хотя и считается устаревшим. Позже в этой главе вы увидите, как его использовать.

Метод соединения `provider` поддерживает следующие основные параметры:

- `host` — определяет удаленный хост;
- `port` — определяет порт подключения;
- `username` — имя пользователя для аутентификации;
- `password` — пароль для аутентификации;
- `transport` — транспортный протокол для соединения;
- `authorize` — позволяет повышать уровень привилегий на устройствах, которые этого требуют;
- `auth_pass` — определяет пароль для повышения уровня привилегий.

Как видите, в переменной `provider` не обязательно указывать все аргументы. Например, в наших предыдущих сценариях Ansible пользователь всегда получает при входе администраторские привилегии, поэтому мы можем опустить аргументы `authorize` и `auth_pass`.

Эти аргументы — обычные переменные, поэтому на них распространяются правила старшинства. Представьте, к примеру, что мы отредактировали файл `cisco_3.yml` следующим образом (теперь это `cisco_4.yml`):

```
---
- name: Configure SNMP Contact
  hosts: "nexus_by_name"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
      password: "{{ password }}"
      transport: cli

  tasks:
    - name: configure snmp contact
```

```
    nxos_snmp_contact:
      contact: TEST_1
      state: present
      username: cisco123 #новая строка
      password: cisco123 #новая строка
      provider: "{{ cli }}"

    register: output

  - name: show output in output["end_state"]["contact"]
    debug:
      msg: '{{ output["end_state"]["contact"] }}'

  - name: show output in output.end_state.contact
    debug:
      msg: '{{ output.end_state.contact }}'
```

Имя пользователя и пароль на уровне задачи переопределят аналогичные значения на уровне сценария. При попытке соединения сценарий столкнется с ошибкой, поскольку такого пользователя на устройстве нет:

```
PLAY [Configure SNMP Contact] ***************************************
*******

TASK [configure snmp contact] ***************************************
*******
fatal: [switch2]: FAILED! => {"changed": false, "failed": true, "msg":
"failed to connect to 172.16.1.143:22"}
fatal: [switch1]: FAILED! => {"changed": false, "failed": true, "msg":
"failed to connect to 172.16.1.142:22"}
to retry, use: --limit @/home/echou/Mastering_Python_Networking_third_
edition/Chapter04/cisco_4.retry

PLAY RECAP **********************************************************
*******
switch1    : ok=0    changed=0    unreachable=0    failed=1
switch2    : ok=0    changed=0    unreachable=0    failed=1
```

В следующем разделе мы подробнее рассмотрим примеры администрирования устройств Cisco.

Пример Ansible с устройствами Cisco

Ansible поддерживает такие операционные системы Cisco, как IOS, IOS-XR и NX-OS. Мы уже имели дело с NX-OS, поэтому теперь попробуем поуправлять устройствами на основе IOS.

Наш файл реестра включает два хоста, `ios-r1` и `ios-r2`:

```
[ios-devices]
ios-r1 ansible_host=172.16.1.134
ios-r2 ansible_host=172.16.1.135

[ios-devices:vars]
username=cisco
password=cisco
```

Сценарий `cisco_5.yml` выполняет произвольные команды `show` с помощью модуля `ios_command`:

```
---
- name: IOS Show Commands
  hosts: "ios-devices"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
      password: "{{ password }}"
      transport: cli

  tasks:
    - name: ios show commands
      ios_command:
        commands:
          - show version | i IOS
          - show run | i hostname
        provider: "{{ cli }}"

      register: output

    - name: show output in output["end_state"]["contact"]
      debug:
          var: output
```

В результате мы ожидаемо получили вывод команд `show version` и `show run`:

```
$ ansible-playbook -i hosts cisco_5.yml

PLAY [IOS Show Commands] ********************************************
*******

TASK [ios show commands] ********************************************
*******
ok: [ios-r1]
ok: [ios-r2]

TASK [show output in output["end_state"]["contact"]]
**************************
ok: [ios-r1] => {
```

```
    "output": {
        "changed": false,
        "stdout": [
            "Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\nROM: Bootstrap program is
IOSv\nCisco IOSv (revision 1.0) with  with 460033K/62464K bytes of memory.",
            "hostname iosv-1"
        ],
        "stdout_lines": [
            [
                "Cisco IOS Software, IOSv Software (VIOSADVENTERPRISEK9-M),
                 Version 15.6(3)M2, RELEASE SOFTWARE (fc2)",
                "ROM: Bootstrap program is IOSv",
                "Cisco IOSv (revision 1.0) with  with 460033K/62464Kbytes
                of memory."
            ],
            [
                "hostname iosv-1"
            ]
        ],
        "warnings": []
    }
}
ok: [ios-r2] => {
    "output": {
        "changed": false,
        "stdout": [
            "Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
             Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\nROM: Bootstrap
             program is IOSv\nCisco IOSv (revision 1.0) with 460033K/62464K
             bytes of memory.",
            "hostname iosv-2"
        ],
        "stdout_lines": [
            [
                "Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
                 Version 15.6(3)M2, RELEASE SOFTWARE (fc2)",
                "ROM: Bootstrap program is IOSv",
                "Cisco IOSv (revision 1.0) with with 460033K/62464Kbytes
                of memory."
            ],
            [
                "hostname iosv-2"
            ]
        ],
        "warnings": []
    }
}

PLAY RECAP *********************************************************
*******
ios-r1     : ok=2    changed=0    unreachable=0    failed=0
ios-r2     : ok=2    changed=0    unreachable=0    failed=0
```

В этом примере я хочу обратить ваше внимание на несколько моментов:

- сценарии для NX-OS и IOS во многом идентичны;
- модули `nxos_snmp_contact` и `ios_command` имеют похожий синтаксис, не считая их аргументов;
- устройство под управлением IOS довольно старое и не поддерживает API, но это не влияет на синтаксис использования модуля.

Таким образом, сценарии мало чем отличаются с точки зрения синтаксиса; несущественные расхождения наблюдаются при использовании разных модулей для выполнения определенных задач.

Пример сценария для Ansible 2.8

Я уже упоминал об изменениях в настройках сетевых соединений в сценариях для Ansible 2.5. Вместе с этой версией был выпущен перечень рекомендаций по использованию сетевых устройств: https://docs.ansible.com/ansible/latest/network/user_guide/network_best_practices_2.5.html. Мы напишем на основе этих рекомендаций еще один сценарий. Файлы для этого примера размещены в отдельном каталоге: `ansible_2-8_example`.

Для его опробования используйте версию Ansible, установленную в системе, или вновь установите версию 2.8 из исходного кода в репозитории Git, как было показано выше:

```
$ ansible --version
ansible 2.8.5
```

В предыдущих примерах мы помещали в файл реестра и список хостов, и связанные с ними переменные. В этом примере мы вынесем переменные в отдельный каталог `host_vars`:

```
$ tree .
.
├── hosts
├── host_vars
│   ├── iosv-1
│   └── iosv-2
└── my_playbook.yml
1 directory, 4 files
```

Файл реестра теперь содержит только группу и имена хостов:

```
$ cat hosts
[ios-devices]
iosv-1
iosv-2
```

В каталоге `host_vars` два файла с именами, соответствующими именам хостов в файле реестра:

```
$ ls host_vars/
iosv-1
iosv-2
```

Файл с переменными хостов содержит то, что раньше было значением переменной `CLI`. Дополнительная переменная `ansible_connection` задает `network_cli` в качестве метода соединения:

```
$ cat host_vars/iosv-1
---
ansible_host: 172.16.1.134
ansible_user: cisco
ansible_ssh_pass: cisco
ansible_connection: network_cli
ansible_network_os: ios
ansbile_become: yes
ansible_become_method: enable
ansible_become_pass: cisco

$ cat host_vars/iosv-2
---
ansible_host: 172.16.1.135
ansible_user: cisco
ansible_ssh_pass: cisco
ansible_connection: network_cli
ansible_network_os: ios
ansbile_become: yes
ansible_become_method: enable
ansible_become_pass: cisco
```

Наш сценарий использует модуль `ios_config` со включенной функцией резервного копирования. Обратите внимание на условие `when`; благодаря ему задача не будет выполняться на хостах с другой операционной системой:

```
$ cat ansible2-8_playbook.yml
---
- name: Chapter 4 Ansible 2.8 Best Practice Demonstration
  connection: network_cli
  gather_facts: false
  hosts: all
  tasks:
    - name: backup
```

```
    ios_config:
      backup: yes
    register: backup_ios_location
    when: ansible_network_os == 'ios'
```

При выполнении этот сценарий должен создать новую папку с резервными копиями конфигураций обоих хостов:

```
$ ansible-playbook -i hosts ansible2-8_playbook.yml
PLAY [Chapter 4 Ansible 2.8 Best Practice Demonstration] ****************
*****************
TASK [backup] ***********************************************************
*****************
changed: [iosv-2]
changed: [iosv-1]
PLAY RECAP **************************************************************
*****************
iosv-1    : ok=1    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
iosv-2    : ok=1    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

Как видите, он действительно создал новый каталог `backup` с двумя файлами внутри:

```
$ tree
.
├── ansible2-8_playbook.yml
├── backup
│   ├── iosv-1_config.2019-09-24@10:40:36
│   └── iosv-2_config.2019-09-24@10:40:36
├── hosts
└── host_vars
    ├── iosv-1
    └── iosv-2
2 directories, 6 files

$ head -20 backup/iosv-1_config.2019-09-24@10\:40\:36
Building configuration...

Current configuration : 4598 bytes
!
! Last configuration change at 17:02:29 UTC Sun Sep 22 2019
!
version 15.6
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
!
hostname iosv-1
!
boot-start-marker
boot-end-marker
```

```
!
!
vrf definition Mgmt-intf
 !
```

Мы увидели, как использовать переменную `network_connection` и структуру каталогов, которая соответствует официальным рекомендациям. О переносе переменных в каталог `host_vars` и об условных выражениях мы поговорим в главе 5. Эту же структуру можно использовать и в примерах с Juniper и Arista из данной главы. Для работы с другими устройствами достаточно изменить значения `network_connection`, а как это сделать, вы узнаете в следующем разделе на примере Juniper.

Пример Ansible с устройствами Juniper

Модуль Ansible для поддержки Juniper требует наличия пакета PyEZ и NETCONF. Если вы опробовали пример с API из главы 3, у вас уже должен быть этот пакет. В противном случае вернитесь к тому разделу и выполните инструкции по установке; там же вы найдете сценарии на Python для проверки работоспособности PyEZ. Еще вам понадобится пакет `jxmlease` для Python:

```
(venv) $ pip install jxmlease
```

Укажем в файле со списком хостов целевое устройство и переменные соединения:

```
[junos_devices]
J1 ansible_host=192.168.24.252

[junos_devices:vars]
username=juniper
password=juniper!
```

В нашем сценарии для Juniper будет использоваться модуль `junos_facts`, который собирает основные сведения об устройстве. Это аналог модуля `setup`; он полезен в ситуациях, когда нужно выполнить какое-то действие в зависимости от возвращаемого значения. Обратите внимание на изменившиеся значения полей `transport` и `port`:

```
---
- name: Get Juniper Device Facts
  hosts: "junos_devices"
  gather_facts: false
  connection: local

  vars:
    netconf:
```

```
        host: "{{ ansible_host }}"
        username: "{{ username }}"
        password: "{{ password }}"
        port: 830
        transport: netconf

    tasks:
    - name: collect default set of facts
      junos_facts:
        provider: "{{ netconf }}"

      register: output

    - name: show output
      debug:
          var: output
```

При выполнении мы получим следующий вывод от устройства Juniper:

```
PLAY [Get Juniper Device Facts]
*********************************************

TASK [collect default set of facts]
***************************************** ok: [J1]

TASK [show output]
************************************************************
ok: [J1] "
<опущено>

PLAY RECAP
********************************************************************** J1
: ok=2 changed=0 unreachable=0 failed=0
```

Итак, мы рассмотрели примеры администрирования устройств Cisco и Juniper. Теперь сравним их с другими примерами, в которых целевые хосты — устройства Arista.

Пример Ansible с устройствами Arista

В заключительном примере рассмотрим модуль `eos_command` для Arista. Мы уже хорошо знакомы с синтаксисом и структурой сценариев Ansible. В качестве метода соединения с устройством Arista можно использовать `cli` или `eapi`. В этом примере мы выбрали `cli`.

Вот файл `hosts`:

```
[eos-devices]
arista1 ansible_host=192.168.199.158
```

Этот сценарий похож на предыдущие:

```
---
- name: EOS Show Commands
  hosts: "eos_devices"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "arista"
      password: "arista"
      authorize: true
      transport: cli

  tasks:
    - name: eos show commands
      eos_command:
        commands:
          - show version | i Arista
        provider: "{{ cli }}"

      register: output

    - name: show output
      debug:
          var: output
```

Как видите, с точки зрения структуры он почти не отличается от примеров с Cisco и Juniper. Это сильная сторона Ansible: даже если речь идет о новом производителе, для работы с его оборудованием можно использовать привычную структуру.

Резюме

В этой главе мы сделали обзор открытого фреймворка автоматизации Ansible. В отличие от Pexpect и API, Ansible предоставляет более высокий уровень абстракции для автоматизации сетевых устройств.

Проект Ansible изначально был нацелен на управление серверами, но позже обзавелся поддержкой сетевых устройств. Поэтому мы сначала рассмотрели пример с сервером, а затем сравнили его с примерами администрирования сетевых устройств. Потом мы создали сценарии для устройств на основе Cisco IOS, Juniper JUNOS и Arista EOS. Мы также затронули официальные рекомендации для Ansible версии 2.8.

В главе 5 мы перейдем к более продвинутым возможностям Ansible: групповым переменным, шаблонам и условным выражениям.

5

Ansible: следующий уровень

В главе 4 мы рассмотрели некоторые базовые структуры для работы с Ansible. Мы использовали файлы реестров, переменные и сценарии. Также рассмотрели примеры использования сетевых модулей для устройств Cisco, Juniper и Arista.

В этой главе вы расширите уже приобретенные знания и углубитесь в изучение продвинутых возможностей Ansible. Об этом фреймворке написано много книг, и его нельзя полностью охватить в двух главах. Однако мы постараемся максимально сократить кривую обучения и познакомиться с большинством возможностей и функций Ansible, которые, как мне кажется, пригодятся вам как сетевому инженеру.

Если вам не совсем понятны какие-то моменты из главы 4, сейчас самое время к ним вернуться, так как без этого вам будет сложно разобраться в материалах этой главы.

Эта глава охватывает следующие темы:

- условные выражения в Ansible;
- циклы в Ansible;
- шаблоны;
- переменные группы и хоста;
- Ansible Vault;

- роли в Ansible;
- написание собственных модулей.

Нам предстоит многое изучить, поэтому начнем!

Подготовка лаборатории

Мы используем лабораторную топологию из главы 4. Также продолжим следовать рекомендациям и вынесем переменные хоста в каталог `host_vars`. А еще воспользуемся несколькими параметрами в `ansible.cfg`, чтобы отключить `host_key_checking` и обнаружение интерпретатора Python.

Больше информации о host_key_checking и обнаружении интерпретатора Python ищите на страницах https://docs.ansible.com/ansible/2.5/user_guide/intro_getting_started.html#host-key-checking.

Ниже проиллюстрированы версия и структура каталогов Ansible, которые мы будем использовать:

```
$ ansible --version
ansible 2.8.5
$ tree host_vars/
host_vars/
├── ios-r1
└── ios-r2
<опущено>

$ cat ansible.cfg
[defaults]
host_key_checking=False
interpreter_python=auto
```

Условные выражения в Ansible

Условные выражения в Ansible похожи на применяемые в языках программирования. В главе 1 мы видели, что инструкции `if`, `then` и `while` в Python позволяют выполнить определенный фрагмент кода только при определенном условии. В Ansible условные выражения используются для выбора задач, которые должны выполняться. Во многих случаях выполнение задач или целого их списка может зависеть от значения факта, переменной или результата преды-

дущей задачи. Представьте, к примеру, список задач для обновления образов маршрутизатора; в этом случае, прежде чем переходить к следующему списку задач с перезагрузкой маршрутизатора, следует убедиться, что новый образ действительно находится на устройстве.

В этом разделе мы обсудим выражение `when`, которое поддерживается во всех модулях, а также уникальные условные состояния, характерные для сетевых командных модулей Ansible. Некоторые из доступных условий:

- равно (`eq`);
- не равно (`neq`);
- больше (`gt`);
- больше или равно (`ge`);
- меньше (`lt`);
- меньше или равно (`le`);
- содержит (`contains`).

Посмотрим на выражение `when` в действии.

Выражение when

Выражение `when` подходит для случаев, когда перед выбором следующего действия нужно проверить значение переменной или результат выполнения предыдущей задачи. Мы уже видели небольшой пример с этим выражением, когда рассматривали рекомендуемую структуру каталогов для Ansible 2.8. Как вы, наверное, помните, задача там выполнялась, только если устройство работало под управлением операционной системы Cisco IOS. Рассмотрим еще один такой пример в файле `chapter5_1.yml`:

```
---
- name: IOS Command Output
  hosts: "ios-devices"
  gather_facts: false
  connection: network_cli

  tasks:
    - name: show hostname
      ios_command:
        commands:
            - show run | i hostname

      register: output
```

```
    - name: show output with when conditions
      when: '"iosv-2" in "{{ output.stdout }}"'
      debug:
        msg: '{{ output }}'
```

Все элементы этого сценария, вплоть до завершения первой задачи, уже знакомы нам по главе 4. Во второй задаче выражение `when` проверяет присутствие в выводе ключевого слова `iosv-2`. Если присутствует, то задача продолжает выполняться и использует модуль `debug` для отображения перехваченного вывода. Запустив этот сценарий, мы получим следующий вывод:

```
$ ansible-playbook -i hosts chapter5_1.yml
<опущено>
TASK [show output with when conditions] ********************************
*****************************************
skipping: [ios-r1]
ok: [ios-r2] => {
    "msg": {
        <опущено>
        "failed": false,
        "stdout": [
            "hostname iosv-2"
        ],
        "stdout_lines": [
            [
                "hostname iosv-2"
            ]
        ]
    }
}
```

Устройство `iosv-r1` отсутствует в выводе, так как для него условие не выполняется. Расширим этот пример в файле `chapter5_2.yml` и сделаем так, чтобы изменения в конфигурации применялись только при выполнении условия:

```
<опущено>
  tasks:
    - name: show hostname
      ios_command:
        commands:
            - show run | i hostname

      register: output

    - name: config example
      when: '"iosv-2" in "{{ output.stdout }}"'
      ios_config:
        lines:
            - logging buffered 30000
```

Ниже вывод этого сценария:

```
$ ansible-playbook -i hosts chapter5_2.yml
<опущено>
TASK [config example] ***********************************************
************************
skipping: [ios-r1]

changed: [ios-r2]

PLAY RECAP ***********************************************************
**************
ios-r1     : ok=1    changed=0    unreachable=0    failed=0    skipped=1
rescued=0    ignored=0
ios-r2     : ok=2    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

Опять же обратите внимание, что изменения были применены только к `ios-r2`, а устройство `ios-r1` было пропущено. В данном случае размер буфера журнала изменился только в `ios-r2`:

```
iosv-2#sh run | i logging
logging buffered 30000
```

Выражение `when` также крайне полезно при использовании модулей `setup` или `facts`: вы можете действовать в зависимости от определенных фактов, собранных вначале. Например, благодаря следующему условному выражению воздействию подвергнутся только хосты с Ubuntu версии 16 или выше:

```
when: ansible_os_family == "Debian" and ansible_lsb.major_release|int >= 16
```

Более подробную информацию об использовании условных выражений в сценариях вы найдете в документации Ansible (https://docs.ansible.com/ansible/latest/user_guide/playbooks_conditionals.html).

В следующем разделе мы увидим, как Ansible собирает сведения (факты) о сетевых устройствах и использует их в контексте сетевых сценариев.

Факты о сетевых устройствах в Ansible

До версии 2.5 система Ansible поставлялась вместе с набором сетевых модулей для сбора фактов об устройствах разных производителей. И модули для каждого производителя назывались и использовались по-разному. Но в Ansible 2.5 началась стандартизация этих модулей. Теперь они собирают информацию из систем и хранят результаты в виде фактов с префиксом `ansible_net_`. Данные,

собираемые этими модулями, перечислены в их документации в разделе с описанием *возвращаемых значений*. Это большой шаг вперед для сетевых модулей Ansible, так как теперь они по умолчанию скрывают от вас сложный процесс сбора фактов.

Воспользуемся рекомендованной для Ansible 2.8 структурой каталогов, которую мы видели в главе 4, но немного ее расширим, чтобы увидеть, как модуль `ios_facts` собирает факты. Наш файл реестра содержит два хоста с IOS, переменные которых находятся в каталоге `host_vars`:

```
$ cat hosts
[ios-devices]
iosv-1
iosv-2
$ cat host_vars/iosv-1
---
ansible_host: 172.16.1.134
ansible_user: cisco
ansible_ssh_pass: cisco
ansible_connection: network_cli
ansible_network_os: ios
ansbile_become: yes
ansible_become_method: enable
ansible_become_pass: cisco
```

Сценарий будет содержать три задачи. Первая собирает сведения об обоих сетевых устройствах с помощью модуля `ios_facts`. Вторая — выводит определенные факты о каждом из этих устройств, собранные и сохраненные ранее. Вы увидите, что отображаемые факты — это стандартные факты `ansible_net`, а не зарегистрированная переменная из первой задачи.

Третья задача выводит всю информацию, собранную для хоста `iosv-1`:

```
$ cat ios_facts_playbook.yml
---
- name: Chapter 5 Ansible 2.8 network facts
  connection: network_cli
  gather_facts: false
  hosts: all
  tasks:
    - name: Gathering facts via ios_facts module
      ios_facts:
      when: ansible_network_os == 'ios'

    - name: Display certain facts
      debug:
        msg: "The hostname is {{ ansible_net_hostname }} running
        {{ansible_net_version }}"
```

```
    - name: Display all facts for a host
      debug:
        var: hostvars['iosv-1']
```

Первые две задачи дают ожидаемые результаты:

```
$ ansible-playbook -i hosts ios_facts_playbook.yml
PLAY [Chapter 5 Ansible 2.8 network facts] ******************************
*******************************************
TASK [Gathering facts via ios_facts module] *****************************
*******************************************
ok: [iosv-2]
ok: [iosv-1]
TASK [Display certain facts] ********************************************
************************************************
ok: [iosv-1] => {
    "msg": "The hostname is iosv-1 running 15.6(3)M2"
}
ok: [iosv-2] => {
    "msg": "The hostname is iosv-2 running 15.6(3)M2"
}
```

Третья задача выводит факты обо всех устройствах IOS. Это огромный набор данных, который поможет вам в автоматизации сети; мы видим его в выводе третьей задачи:

```
TASK [Display all facts for a host] *************************************
************************************************
ok: [iosv-1] => {
    "hostvars['iosv-1']": {
        "ansbile_become": true,
        "ansible_become_method": "enable",
        "ansible_become_pass": "cisco",
        "ansible_check_mode": false,
        "ansible_connection": "network_cli",
        "ansible_diff_mode": false,
        "ansible_facts": {
            "discovered_interpreter_python": "/usr/bin/python",
            "net_all_ipv4_addresses": [
                "10.0.0.13",
                "10.0.0.5",
                "10.0.0.17",
                "172.16.1.134",
                "192.168.0.1"
            ],
            "net_all_ipv6_addresses": [],
            "net_api": "cliconf",
            "net_filesystems": [
                "flash0:"
            ],
<опущено>
```

Обновление модуля для сбора сведений о сетевых устройствах в Ansible 2.5 существенно упростило рабочий процесс и вывело этот модуль на один уровень с другими серверными модулями.

Условные выражения в сетевых модулях

Давайте рассмотрим еще один пример с условными выражениями. На этот раз мы используем ключевое слово, представляющее операцию сравнения, упоминавшееся в начале этой главы. Воспользуемся тем фактом, что IOSv и Arista EOS возвращают вывод команды `show` в формате JSON. Например, мы можем проверить состояние интерфейса:

```
veos01#sh int eth 1 | json
{
    "interfaces": {
        "Ethernet1": {
            "lastStatusChangeTimestamp": 1569573423.6540787,
            "name": "Ethernet1",
            "interfaceStatus": "disabled",
            "autoNegotiate": "off",
            "loopbackMode": "loopbackNone",
            "interfaceStatistics": {
<опущено>
```

Представим, что у нас есть операция, которая должна выполняться, только если интерфейс Ethernet1 неактивен. Ниже в сценарии `chapter5_3.yml` приводятся задачи, которые проверяют выполнение заданного условия перед продолжением работы. Первая из них использует модуль `eos_command` для получения информации об интерфейсе и перед переходом к следующей задаче проверяет состояние этого интерфейса с помощью ключевых слов `wait_for` и `eq`:

```
<опущено>
  tasks:
    - name: "sh int ethernet 1 | json"
      eos_command:
        commands:
            - "show interface ethernet 1 | json"
        wait_for:
          - "result[0].interfaces.Ethernet1.interfaceStatus eq disabled"

      register: output
    - name: show output
      debug:
        msg: "Interface Disabled, Safe to Proceed"
```

Вторая задача запускается только в случае выполнения условия:

```
$ ansible-playbook -i hosts chapter5_3.yml
<опущено>
TASK [sh int ethernet 1 | json] ************************************
*********************************
ok: [arista1]
TASK [show output] *************************************************
*********************
ok: [arista1] => {
    "msg": "Interface Disabled, Safe to Proceed"
}
```

Как показано в следующем выводе, если интерфейс активен, мы получим ошибку. Вывод, следующий за первой задачей, сигнализирует, что ввиду невыполнения условия изменение не внесено:

```
TASK [sh int ethernet 1 | json] ************************************
*********************************

fatal: [arista1]: FAILED! => {<опущено>"changed": false, "failed_
conditions": ["result[0].interfaces.Ethernet1.interfaceStatus eq
disabled"], "msg": "One or more conditional statements have not been
satisfied"}

PLAY RECAP *********************************************************
**************

arista1                    : ok=0    changed=0    unreachable=0
failed=1    skipped=0    rescued=0    ignored=0
```

Познакомьтесь самостоятельно с условными выражениями `contains`, `greater than` и `less than`, если они подходят для вашего конкретного случая. Они позволяют сценарию вести себя более «разумно»: выполнять действия в зависимости от состояния устройства.

В следующем разделе мы поговорим о циклах в Ansible и о том, как с помощью нескольких строк кода организовать автоматическое выполнение задач для набора элементов.

Циклы в Ansible

Сценарии Ansible поддерживают несколько видов циклов, включая стандартные, перебор файлов и подэлементов, `do-until` и многие другие. В этом разделе мы рассмотрим два наиболее распространенных вида: стандартные циклы и перебор значений хеш-таблицы.

Стандартные циклы

Стандартные циклы в сценариях часто используются для многократного выполнения похожих задач. Они имеют очень простой синтаксис: переменная `{{ item }}` содержит текущий элемент списка. В нашем примере, `chapter5_4.yml`, мы пройдемся по элементам списка `with_items`, выполняя команду `echo` на нашем локальном хосте.

```
$ cat chapter5_4.yml
---
- name: Echo Loop Items
  hosts: "localhost"
  gather_facts: false

  tasks:
    - name: echo loop items
      command: echo "{{ item }}"
      with_items:
        - 'r1'
        - 'r2'
        - 'r3'
        - 'r4'
        - 'r5'
```

Прежде чем запустить сценарий, скопируем наш собственный открытый ключ из `~/.ssh/id_rsa.pub` и вставим его в `~/.ssh/authorized_keys`:

```
$ ansible-playbook -i hosts chapter5_4.yml
<опущено>
TASK [echo loop items] *********************************************
*************************

changed: [localhost] => (item=r1)
changed: [localhost] => (item=r2)
changed: [localhost] => (item=r3)
changed: [localhost] => (item=r4)
changed: [localhost] => (item=r5)
```

В сценарии `chapter5_5.yml` стандартный цикл объединен с сетевым командным модулем. Таким способом можно добавить в устройство сразу несколько интерфейсов VLAN:

```
---
- name: Add Multiple Vlans
  hosts: "nxos-r1"
  gather_facts: false
  connection: network_cli

  vars:
    vlan_numbers: [100, 200, 300]
```

```
  tasks:
    - name: add vlans
      nxos_config:
        lines:
            - vlan {{ item }}
      with_items: "{{ vlan_numbers }}"
      register: output
```

В этом примере видно, что список `with_items` можно прочитать и из переменной, что делает структуру нашего сценария более гибкой:

```
vars:
  vlan_numbers: [100, 200, 300]
```

Выполним этот сценарий и проверим, появились ли на устройстве `nxos-r1` соответствующие интерфейсы VLAN:

```
$ ansible-playbook -i hosts chapter5_5.yml
<опущено>
TASK [add vlans] ***********************************************************
*******************
changed: [nxos-r1] => (item=100)
changed: [nxos-r1] => (item=200)
changed: [nxos-r1] => (item=300)
PLAY RECAP *****************************************************************
**************
nxos-r1                    : ok=1    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0

nx-osv-1# sh vlan

VLAN Name                              Status    Ports
---- --------------------------------- --------- -------------------------
------
1    default                           active
100  VLAN0100                          active
200  VLAN0200                          active
300  VLAN0300                          active
```

Стандартный цикл экономит много времени при выполнении в сценарии повторяющихся действий. Он также делает сценарий более удобочитаемым, сокращая количество строк в коде задачи.

В Ansible 2.5 было добавлено ключевое слово loop для замены большинства циклов вида with_<lookup> (https://docs.ansible.com/ansible/2.8/user_guide/playbooks_loops.html). Ключевое слово with_<lookup> ввиду его популярности будет поддерживаться и дальше. Но вы должны знать о новом ключевом слове и изменениях в связи с ним.

В следующем разделе мы займемся циклическим перебором словарей.

Циклический перебор словарей

Перебор простого списка — это хорошо, но зачастую нам приходится иметь дело с сущностями, у которых больше одного атрибута. Если взять пример с VLAN из предыдущего раздела, за каждым интерфейсом закреплено несколько уникальных атрибутов, таких как описание, IP-адрес шлюза и др. Нередко представить сущности с множественными атрибутами можно с помощью словаря.

Следующий сценарий, `chapter5_6.yml`, — продолжение примера с VLAN в `chapter5_5.yml`. Мы определили словарь `vlans` со значениями для трех интерфейсов в виде вложенных словарей с описанием и IP-адресом:

```
---
- name: Add Multiple Vlans
  hosts: "nxos-r1"
  gather_facts: false
  connection: network_cli

  vars:
    vlans: {
        "100": {"description": "floor_1", "ip": "192.168.10.1"},
        "200": {"description": "floor_2", "ip": "192.168.20.1"},
        "300": {"description": "floor_3", "ip": "192.168.30.1"}
      }

  tasks:
    - name: add vlans
      nxos_config:
        lines:
          - vlan {{ item.key }}
      with_dict: "{{ vlans }}"

    - name: configure vlans
      nxos_config:
        lines:
          - description {{ item.value.description }}
          - ip address {{ item.value.ip }}/24
        parents: interface vlan {{ item.key }}
      with_dict: "{{ vlans }}"
```

В данном сценарии первая задача добавляет интерфейсы VLAN, читая ключи словаря, а вторая конфигурирует эти интерфейсы, используя значения, хранящиеся в каждом элементе. Обратите внимание: мы используем параметр `parents` для однозначной идентификации раздела, с которым должны сверяться команды. Это продиктовано тем, что описание и IP-адрес указываются в конфигурации в подразделе `interface vlan <number>`.

Перед выполнением команды убедимся, что на устройстве Nexus включен интерфейс третьего уровня:

```
nx-osv-1# sh run | i interface-vlan
feature interface-vlan
```

При выполнении вы увидите, как происходит перебор элементов словаря в цикле:

```
$ ansible-playbook -i hosts chapter5_6.yml
<опущено>
TASK [add vlans] ***************************************************
************************************************
changed: [nxos-r1] => (item={'value': {u'ip': u'192.168.30.1',
u'description': u'floor_3'}, 'key': u'300'})
changed: [nxos-r1] => (item={'value': {u'ip': u'192.168.20.1',
u'description': u'floor_2'}, 'key': u'200'})
changed: [nxos-r1] => (item={'value': {u'ip': u'192.168.10.1',
u'description': u'floor_1'}, 'key': u'100'})

TASK [configure vlans] *********************************************
************************************************
changed: [nxos-r1] => (item={'value': {u'ip': u'192.168.30.1',
u'description': u'floor_3'}, 'key': u'300'})
changed: [nxos-r1] => (item={'value': {u'ip': u'192.168.20.1',
u'description': u'floor_2'}, 'key': u'200'})
changed: [nxos-r1] => (item={'value': {u'ip': u'192.168.10.1',
u'description': u'floor_1'}, 'key': u'100'})
<опущено>
```

Проверим, была ли применена к устройству нужная нам конфигурация:

```
nx-osv-1# sh run int vlan 100

!Command: show running-config interface Vlan100
!Time: Fri Sep 27 18:00:24 2019

version 7.3(0)D1(1)

interface Vlan100
  description floor_1
  ip address 192.168.10.1/24
```

О других видах циклов в Ansible читайте в документации (https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html).

С циклическим перебором словарей нужно немного попрактиковаться. Но, как и в случае со стандартными циклами, владение этим инструментом бесценно. Циклы экономят время и делают сценарии более понятными. В следующем

разделе мы поговорим о шаблонах Ansible, которые позволяют методично вносить изменения в текстовые файлы и в основном используются для конфигурации сетевых устройств.

Шаблоны

С первых дней работы сетевым инженером я использовал системы шаблонов. По моему опыту, у многих сетевых устройств есть идентичные участки конфигурации, особенно если они выполняют одну и ту же функцию в вашей сети.

В большинстве случаев, когда нужно настроить новое устройство, мы копируем на него уже готовый шаблон конфигурации, предварительно заменив нужные поля. Ansible позволяет автоматизировать этот процесс с помощью модуля `template` (https://docs.ansible.com/ansible/2.5/modules/template_module.html).

Мы используем базовый файл шаблона, написанный на языке шаблонов Jinja2 (http:/jinja.pocoo.org/docs/). В главе 4 мы упоминали этот язык, а здесь рассмотрим его подробнее. Jinja2, как и Ansible, имеет свой синтаксис с поддержкой циклов и условных выражений; к счастью, нам будет достаточно знания основных понятий. Шаблоны Ansible — важный инструмент, мы будем применять их в повседневных задачах, и этот раздел посвящен им. Чтобы изучить их синтаксис, мы возьмем простой сценарий Ansible и будем постепенно его расширять.

Базовый синтаксис использования шаблонов крайне простой; достаточно указать исходный файл и место, куда его скопировать.

Создадим новый каталог `Templates` и приступим к разработке сценариев. Начнем с пустого файла:

```
$ mkdir Templates
$ cd Templates/
$ touch file1
```

Затем воспользуемся сценарием `chapter5_7.yml`, чтобы скопировать `file1` в `file2`. Обратите внимание: этот сценарий выполняется только на управляющем компьютере:

```
---
- name: Template Basic
  hosts: localhost

  tasks:
    - name: copy one file to another
```

```
      template:
        src=/home/echou/Mastering_Python_Networking_third_edition/
Chapter05/Templates/file1
        dest=/home/echou/Mastering_Python_Networking_third_edition/
Chapter05/Templates/file2
```

Выполнение сценария приведет к созданию нового файла:

```
$ ansible-playbook -i hosts chapter5_7.yml
TASK [copy one file to another] ****************************************
**********************************

changed: [localhost]

$ ls file*
file1
file2
```

Исходные файлы могут иметь любое расширение, но поскольку они обрабатываются механизмом шаблонов Jinja2, создадим для шаблона текстовый файл с именем `nxos.j2`. В нем мы будем соблюдать требования синтаксиса Jinja2, согласно которому ссылки на переменные помещаются в двойные фигурные скобки, а команды — в одинарные фигурные скобки со знаком процента:

```
hostname {{ item.value.hostname }}

feature telnet
feature ospf
feature bgp
feature interface-vlan

{% if item.value.netflow_enable %}
feature netflow
{% endif %}

username {{ item.value.username }} password {{ item.value.password }}
role network-operator

{% for vlan_num in item.value.vlans %}
vlan {{ vlan_num }}
{% endfor %}

{% if item.value.l3_vlan_interfaces %}
{% for vlan_interface in item.value.vlan_interfaces %}
interface {{ vlan_interface.int_num }}
  ip address {{ vlan_interface.ip }}/24
{% endfor %}
{% endif %}
```

Теперь напишем сценарий, создающий конфигурацию на основе файла шаблона `nxos.j2`.

Переменные в шаблонах Jinja2

Сценарий `chapter5_8.yml` расширяет предыдущий пример следующим образом.

1. Исходный файл называется `nxos.j2`.
2. Имя конечного файла теперь задается переменной, которая сама извлекается из словаря `nexus_devices`, объявленного в сценарии.
3. Для каждого устройства в `nexus_devices` объявляются переменные, которые подменяются или циклически перебираются внутри шаблона.

Этот сценарий может показаться сложнее предыдущего, но если убрать участок с объявлениями переменных, различия окажутся не такими уж большими:

```
---
- name: Template Looping
  hosts: localhost

  vars:
    nexus_devices: {
        "nx-osv-1": {
            "hostname": "nx-osv-1",
            "username": "cisco",
            "password": "cisco",
            "vlans": [100, 200, 300],
            "l3_vlan_interfaces": True,
            "vlan_interfaces": [
                {"int_num": "100", "ip": "192.168.10.1"},
                {"int_num": "200", "ip": "192.168.20.1"},
                {"int_num": "300", "ip": "192.168.30.1"}
            ],
            "netflow_enable": True
        },
        "nx-osv-2": {
            "hostname": "nx-osv-2",
            "username": "cisco",
            "password": "cisco",
            "vlans": [100, 200, 300],
            "l3_vlan_interfaces": False,
            "netflow_enable": False
        }
    }
  tasks:
    - name: create router configuration files
      template:
        src=/home/echou/Mastering_Python_Networking_third_edition/
        Chapter05/Templates/nxos.j2
        dest=/home/echou/Mastering_Python_Networking_third_edition/
        Chapter05/Templates/{{ item.key }}.conf
      with_dict: "{{ nexus_devices }}"
```

Пока не будем его выполнять; сначала обсудим условные инструкции `if` и циклы `for`, которые в этом шаблоне Jinja2 заключены в `{% %}`.

Циклы в Jinja2

В шаблоне `nxos.j2` есть два цикла `for`: один перебирает список `vlans`, а другой — интерфейсы `vlan`:

```
{% for vlan_num in item.value.vlans %}
vlan {{ vlan_num }}
{% endfor %}

{% if item.value.l3_vlan_interfaces %}
{% for vlan_interface in item.value.vlan_interfaces %}
interface {{ vlan_interface.int_num }}
  ip address {{ vlan_interface.ip }}/24
{% endfor %}
{% endif %}
```

Как вы, наверное, помните, циклы в Jinja2 позволяют перебирать как списки, так и словари. В нашем примере переменная `vlans` — это список, а `vlan_interfaces` — список словарей.

Цикл по элементам `vlan_interfaces` находится внутри условного выражения. И это последнее, на что следует обратить внимание перед выполнением сценария.

Условные выражения в Jinja2

Jinja2 поддерживает проверку условий с помощью `if`. В шаблоне `nxos.j2` она применяется дважды — для переменных `netflow` и `l3_vlan_interfaces`. Код внутри блока выполняется, только если условие истинно:

```
<опущено>
{% if item.value.netflow_enable %}
feature netflow
{% endif %}
<опущено>
{% if item.value.l3_vlan_interfaces %}

<опущено>
{% endif %}
```

В сценарии параметру `netflow_enable` присваивается значение `True`, если это устройство `nx-os-v1`, и `False`, если `nx-os-v2`:

```
vars:
  nexus_devices: {
      "nx-osv-1": {
        <опущено>
        "netflow_enable": True
    },
    "nx-osv-2": {
        <опущено>
        "netflow_enable": False
    }
}
```

Наконец, запускаем сценарий:

```
$ ansible-playbook -i hosts chapter5_8.yml
PLAY [Template Looping] ********************************************
**************************************************
TASK [Gathering Facts] *********************************************
**************************************************
ok: [localhost]
TASK [create router configuration files] ***************************
**************************************************
changed: [localhost] => (item={'value': {u'username': u'cisco',
u'hostname': u'nx-osv-2', u'l3_vlan_interfaces': False, u'vlans': [100,
200, 300], u'password': u'cisco', u'netflow_enable': False}, 'key': u'nxosv-
2'})
changed: [localhost] => (item={'value': {u'username': u'cisco', u'vlan_
interfaces': [{u'int_num': u'100', u'ip': u'192.168.10.1'}, {u'int_
num': u'200', u'ip': u'192.168.20.1'}, {u'int_num': u'300', u'ip':
u'192.168.30.1'}], u'hostname': u'nx-osv-1', u'l3_vlan_interfaces': True,
u'vlans': [100, 200, 300], u'password': u'cisco', u'netflow_enable':
True}, 'key': u'nx-osv-1'})

PLAY RECAP *********************************************************
**************************************************
localhost                  : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Помните, что имена конечных файлов имеют вид `{{ элемент.ключ }}.conf`? Мы создали два файла с именами устройств:

```
$ ls nx-os*
nx-osv-1.conf

nx-osv-2.conf
```

Посмотрим сходства и различия между этими двумя конфигурационными файлами и проверим, содержат ли они нужные изменения. В обоих файлах должны присутствовать статические элементы, такие как `feature ospf`, вместо

сетевых имен и других переменных — подставлены подходящие значения, а функция `netflow` и конфигурация интерфейса третьего уровня `vlan` должны быть только в `nx-osv-1.conf`:

```
$ cat nx-osv-1.conf
hostname nx-osv-1

feature telnet
feature ospf
feature bgp
feature interface-vlan

feature netflow

username cisco password cisco  role network-operator

vlan 100
vlan 200
vlan 300

interface 100
  ip address 192.168.10.1/24
interface 200
  ip address 192.168.20.1/24
interface 300
  ip address 192.168.30.1/24


$ cat nx-osv-2.conf
hostname nx-osv-2

feature telnet
feature ospf
feature bgp
feature interface-vlan

username cisco password cisco  role network-operator

vlan 100
vlan 200
vlan 300
```

Здорово, правда? Это точно сэкономит уйму времени в ситуациях, когда требуется многократное копирование. Лично для меня знакомство с модулем `template` стало переломным моментом. Одно это мотивировало меня на изучение и использование Ansible несколько лет назад.

Наш сценарий разросся. В следующем разделе вы увидите, как его оптимизировать переносом файлов с переменными в группы и каталоги.

Переменные групп и хостов

Обратите внимание: в предыдущем сценарии, `chapter5_8.yml`, мы продублировали переменные с именем пользователя и паролем для двух устройств в словаре `nexus_devices`:

```
vars:
  nexus_devices: {
       "nx-osv-1": {
           "hostname": "nx-osv-1",
           "username": "cisco",
           "password": "cisco",
<опущено>
       "nx-osv-2": {
           "hostname": "nx-osv-2",
           "username": "cisco",
           "password": "cisco",
<опущено>
```

Это не лучшее решение. Если вы захотите обновить эти значения, изменения придется вносить в двух местах. Это затрудняет администрирование и повышает вероятность ошибки. В документации Ansible рекомендуется выносить переменные из сценариев в каталоги `group_vars` и `host_vars`.

Больше рекомендаций по использованию Ansible ищите по ссылке http://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html.

Дальше мы познакомимся с групповыми переменными. Для этого создадим в нашей рабочей папке новый каталог `group_host_vars`, куда поместим код примера для следующего раздела. Он будет основан на уже написанном нами сценарии `chapter5_8.yml`. Скопируем файл `chapter5_8.yml` в этот новый каталог и переименуем его в `chapter5_9.yml`. Попробуем добавить в него переменные.

Переменные группы

По умолчанию Ansible ищет групповые переменные в каталоге `group_vars`, находящемся в одной папке со сценарием. Эти переменные применяются к группе. Имя файла по умолчанию должно совпадать с именем группы, указанной в файле реестра. Например, если у нас есть группа `[nexus-devices]`, то в каталог `group_vars` можно поместить файл `nexus-devices` со всеми переменными для этой группы.

Можно также создать специальный файл `all` с переменными для всех групп.

Мы воспользуемся этой возможностью для хранения переменных с именем пользователя и паролем. Для начала создадим каталог `group_vars`:

```
$ mkdir group_vars
```

Затем создадим YAML-файл `all` с определениями переменных `username` и `password`:

```
$ cat group_vars/all
---
username: cisco
password: cisco
```

Теперь используем групповые переменные в сценарии `chapter5_9.yml`:

```
"nexus_devices":
  "nx-osv-1":
    "hostname": "nx-osv-1"
    "username": "{{ username }}"
    "password": "{{ password }}"
<опущено>
  "nx-osv-2":
    "hostname": "nx-osv-2"
    "username": "{{ username }}"
    "password": "{{ password }}"
<опущено>
```

Групповые переменные нужны в ситуациях, когда одни и те же значения подходят для нескольких устройств. В нашем сценарии на устройствах `nx-osv-1` и `nx-osv-2` были настроены одинаковые имя пользователя и пароль. В следующем примере мы посмотрим, как использовать переменные с разными значениями для разных хостов.

Переменные хоста

Переменные хоста, как и переменные групп, тоже можно вынести из сценария. Именно так мы поступали в примерах для Ansible 2.8 в этой и предыдущей главах:

```
$ mkdir host_vars
```

Мы выполняем команду локально, поэтому файл внутри `host_vars` должен иметь соответствующее имя, `host_vars/localhost`. В этом же файле можно переопределять переменные, объявленные в `group_vars`:

```
$ cat host_vars/localhost
---
"nexus_devices":
  "nx-osv-1":
    "hostname": "nx-osv-1"
    "username": "{{ username }}"
    "password": "{{ password }}"
    "vlans": [100, 200, 300]
    "l3_vlan_interfaces": True
    "vlan_interfaces": [
        {"int_num": "100", "ip": "192.168.10.1"},
        {"int_num": "200", "ip": "192.168.20.1"},
        {"int_num": "300", "ip": "192.168.30.1"}
     ]
    "netflow_enable": True

  "nx-osv-2":
    "hostname": "nx-osv-2"
    "username": "{{ username }}"
    "password": "{{ password }}"
    "vlans": [100, 200, 300]
    "l3_vlan_interfaces": False
    "netflow_enable": False
```

После переноса переменных наш сценарий стал намного проще, он содержит только логику нашей операции:

```
---
- name: Ansible Group and Host Varibles
  hosts: localhost

  tasks:
    - name: create router configuration files
      template:
        src=/home/echou/Mastering_Python_Networking_third_edition/
        Chapter05/Group_Host_Vars/nxos.j2
        dest=/home/echou/Mastering_Python_Networking_third_edition/
       Chapter05/Group_Host_Vars/{{ item.key }}.conf
      with_dict: "{{ nexus_devices }}"
```

Каталоги `group_vars` и `host_vars` не только оптимизируют наши операции, но и позволяют собрать всю конфиденциальную информацию в нескольких файлах и обезопасить ее с помощью Ansible Vault, о чем и пойдет речь дальше.

Ansible Vault

Как видно из предыдущего раздела, в большинстве случаев переменные Ansible содержат конфиденциальную информацию, такую как имя пользователя и пароль. Было бы неплохо их защитить. Ansible Vault шифрует файлы, чтобы они

не хранились в виде обычного текста (https://docs.ansible.com/ansible/2.8/user_guide/vault.html).

Все функции Ansible Vault начинаются с команды `ansible-vault`. Вы можете вручную создать зашифрованный файл с помощью параметра `create`. Вам будет предложено ввести пароль. После этого, открыв данный файл, вы не увидите в нем понятного текста. На случай, если вы загрузили примеры для этой книги, в качестве пароля я использовал слово `password`:

```
$ ansible-vault create secret.yml
Vault password: <password>
$ cat secret.yml
$ANSIBLE_VAULT;1.1;AES256 336564626462373962326635326361323639323635353630
06466656564303532613837376 23
<опущено>6535373338373838636365303564646230323334323861393033356632623962
```

Для просмотра или редактирования зашифрованного файла используйте параметры `view` или `edit` соответственно:

```
$ ansible-vault edit secret.yml
Vault password: <password>
```

Зашифруем файлы с переменными `group_vars/all` и `host_vars/localhost`:

```
$ ansible-vault encrypt group_vars/all host_vars/localhost
New Vault password:
Confirm New Vault password:
```

Теперь, если запустить сценарий, мы получим сообщение о неудавшейся расшифровке:

```
$ ansible-playbook chapter5_10.yml
PLAY [Ansible Group and Host Varibles] ********************************
****************************************
ERROR! Attempting to decrypt but no vault secrets found
```

Для запуска этого сценария используем параметр `--ask-vault-pass`:

```
$ ansible-playbook chapter5_10.yml --ask-vault-pass
Vault password:
<опущено>
```

Любые зашифрованные файлы, которые используются сценарием, будут расшифровываться в памяти.

Раньше все файлы в Ansible Vault шифровались с одним и тем же паролем. Но с выходом Ansible 2.4 стало можно указывать разные файлы с паролями с помощью параметра --vault-id: https://docs.ansible.com/ansible/2.8/user_guide/vault.html.

Пароль также можно сохранить в файл и ограничить доступ к нему:

```
$ chmod 400 ~/.vault_password.txt
$ ls -lia ~/.vault_password.txt
809496 -r-------- 1 echou echou 9 Feb 18 12:17
/home/echou/.vault_password.txt
```

Выполним этот сценарий с параметром `--vault-password-file`:

```
$ ansible-playbook chapter5_10.yml --vault-password-file
~/.vault_password.txt
```

Зашифруем отдельную строку и встроим ее в сценарий с помощью параметра `encrypt_string` (https://docs.ansible.com/ansible/2.8/user_guide/vault.html#use-encrypt-string-to-create-encrypted-variables-to-embed-in-yaml):

```
$ ansible-vault encrypt_string New
New Vault password:
Confirm New Vault password:
!vault |
          $ANSIBLE_VAULT;1.1;AES256
          36313139616164343861633937336334663435336434313132373463336334
           53038366230653839
          33656362636431386437383665363734653761306631346l0a3637376539656
           13432333335626432
          64653437383730323737646266363561653431366334646265376161303137
           43164343538633765
          36633032326263383l0a33623832336438333835383561316265376366343864
           36137653865646261
          3234
Encryption successful
```

Затем эту строку можно поместить в файл сценария в виде переменной. В следующем разделе мы еще больше оптимизируем наш сценарий с помощью инструкции `include` и ролей.

Подключение файлов и роли в Ansible

Сложные задачи лучше разбивать на простые. Конечно, этот подход применяется как в Python, так и в проектировании сетей. В Python мы разбиваем сложный код на функции, классы, модули и пакеты, а сложные сети делим на стойки, ряды, кластеры и дата-центры. В Ansible для сегментации и организации сложных сценариев в виде множества файлов применяются роли и инструкции

include. Это упрощает структуру сценариев, так как каждому файлу отводится меньшее число задач. Это также делает возможным повторное использование разделов сценариев.

Инструкции include в Ansible

С увеличением размера сценария в какой-то момент становится очевидно: многие его задачи можно использовать также в других сценариях. В Ansible есть инструкция include, подобная тем, которые используются во многих конфигурационных файлах в Linux; она позволяет подключать внешние файлы так, будто их содержимое вписано непосредственно в конфигурацию. Инструкцией include можно подключать и целые сценарии, и отдельные задачи. Рассмотрим простой пример расширения нашей задачи.

Представьте, что мы хотим отобразить вывод двух разных сценариев. Создадим отдельный YAML-файл show_output.yml с дополнительной задачей:

```
---
- name: show output
  debug:
    var: output
```

И затем используем эту задачу в разных сценариях. Например, файл chapter5_11_1.yml почти идентичен последнему сценарию, если не считать регистрацию вывода и инструкции include в конце:

```
---
- name: Ansible Group and Host Varibles
  hosts: localhost

  tasks:
    - name: create router configuration files
      template:
        src=./nxos.j2
        dest=./{{ item.key }}.conf
      with_dict: "{{ nexus_devices }}"
      register: output

    - include: show_output.yml
```

Чтобы продемонстрировать повторное использование задачи с помощью include, рассмотрим еще один сценарий, chapter5_11_2.yml, который тоже может подключить show_output.yml:

```
---
- name: show users
  hosts: localhost

  tasks:
    - name: show local users
      command: who
      register: output

    - include: show_output.yml
```

Обратите внимание: оба сценария используют переменную с одним и тем же именем, `output`, потому что для простоты примера мы прописали это имя непосредственно в коде `show_output.yml`. Если вам это не подходит, можете передавать переменные в подключаемый файл.

Роли Ansible

Ansible поддерживает определение ролей, которые могут играть физические хосты в вашей сети. Например, можно создать такие роли, как «магистральный», «листовой» и «основной» или же Cisco, Juniper и Arista. Один и тот же физический хост может иметь несколько ролей; например, устройству можно назначить роли «основной» и Juniper. Это позволит, к примеру, обновить все устройства Juniper, не заботясь об их уровне сети.

Роли Ansible могут автоматически загружать определенные переменные, задачи и обработчики с учетом заранее определенной структуры файлов. Ansible уже знает эту структуру, поэтому файлы подключаются автоматически. На самом деле роли можно считать аналогами инструкций `include`, предварительно созданными самой системой Ansible.

В документации Ansible (https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html) перечислены каталоги, которые можно использовать для определения ролей, такие как `tasks`, `handlers`, `files`, `templates`, `vars`, `defaults` и `meta`. Нам не обязательно использовать их все. В нашем следующем примере мы используем только папки `tasks` и `vars`. Но вообще полезно знать обо всех доступных вариантах в структуре каталогов для ролей Ansible.

Ниже показана структура примера для демонстрации ролей:

```
$ tree .
.
├── chapter5_12.yml
├── chapter5_13.yml
├── hosts
```

```
└── roles
    ├── cisco_nexus
    │   ├── tasks
    │   │   └── main.yml
    │   └── vars
    │       └── main.yml
    └── spines
        ├── tasks
        │   └── main.yml
        └── vars
            └── main.yml

7 directories, 7 files
```

Как видите, файл `hosts` и сценарии находятся на верхнем уровне. Там же — папка `roles`, внутри которой размещены две папки с ролями: `cisco_nexus` и `spines`. В этом примере из всех доступных каталогов ролей используются только `tasks` и `vars`. Внутри каждого каталога находится файл `main.yml`, который описывает поведение по умолчанию; это наша точка входа, которая автоматически подключается к сценарию, если в нем указана соответствующая роль. `main.yml` можно разбить на несколько файлов с помощью инструкции `include`.

Вот что мы собираемся сделать.

- У нас есть два устройства Cisco Nexus, `nxos-r1` и `nxos-r2`. Для каждого мы настроим журнальный сервер и журналирование состояния канала, используя роль `cisco_nexus`.
- `nxos-r1` — магистральное устройство, и на нем желательно организовать более подробное журналирование, так как оно занимает более важное место в сети.

Для роли `cisco_nexus` в файле `roles/cisco_nexus/vars/main.yml` предусмотрены следующие переменные:

```
roles/cisco_nexus/vars/main.yml:

   ---
   cli:
     host: "{{ ansible_host }}"
     username: cisco
     password: cisco
     transport: cli
```

В `roles/cisco_nexus/tasks/main.yml` находятся такие конфигурационные задачи:

```
roles/cisco_nexus/tasks/main.yml:

   ---
```

```
    - name: configure logging parameters
      nxos_config:
        lines:
          - logging server 191.168.1.100
          - logging event link-status default
        provider: "{{ cli }}"
```

Сценарий `chapter5_12.yml` получился чрезвычайно простым. Он требует лишь перечислить хосты, которые нужно сконфигурировать согласно роли `cisco_nexus`:

```
---
- name: playbook for cisco_nexus role
  hosts: "cisco_nexus"
  gather_facts: false
  connection: local

  roles:
    - cisco_nexus
```

При выполнении этот сценарий подключит задачи и переменные, определенные в роли `cisco_nexus`, и сконфигурирует устройства соответствующим образом:

```
$ ansible-playbook -i hosts chapter5_12.yml
<опущено>
TASK [cisco_nexus : configure logging parameters] **********************
**************************************************
changed: [nxos-r1]
changed: [nxos-r2]
```

Для роли `spine` у нас предусмотрена дополнительная задача, которая настраивает более подробное журналирование (`roles/spines/tasks/main.yml`):

```
---
- name: change logging level
  nxos_config:
    lines:
      - logging level local7 7
    provider: "{{ cli }}"
```

В сценарии `chapter5_13.yml` можно указать роли `cisco_nexus` и `spines`:

```
---
- name: playbook for spine role
  hosts: "spines"
  gather_facts: false
  connection: local

  roles:
    - cisco_nexus
    - spines
```

Роли выполняются в порядке подключения: сначала `cisco_nexus`, а затем `spines`:

```
$ ansible-playbook -i hosts chapter5_13.yml
<опущено>
TASK [cisco_nexus : configure logging parameters] **********************
************************************************
changed: [nxos-r1]

TASK [spines : change logging level] ***********************************
*************************************
changed: [nxos-r1]
<опущено>
```

Роли Ansible отличаются гибкостью и масштабируемостью. В этом они похожи на функции и классы Python. Когда ваш код достигает определенного размера, его почти всегда имеет смысл разбить на части поменьше, чтобы упростить сопровождение.

Больше примеров с ролями ищите в Git-репозитории Ansible: https://github.com/ansible/ansible-examples.

Ansible Galaxy (https://docs.ansible.com/ansible/latest/reference_appendices/galaxy.html) — это сайт, поддерживаемый сообществом, который позволяет искать и совместно работать над ролями, а также обмениваться ими. На рис. 5.1 показан пример сетей Juniper на основе роли Ansible на сайте Ansible Galaxy.

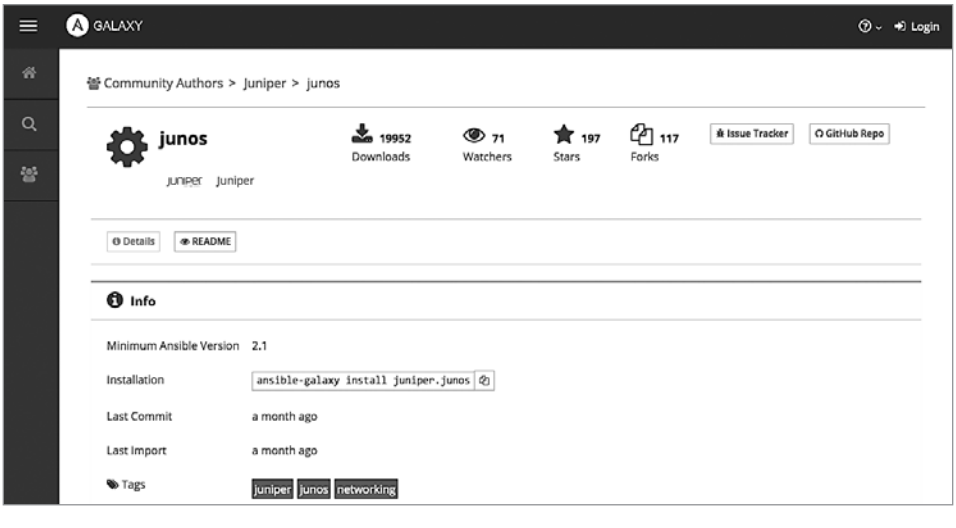


Рис. 5.1. Роль JUNOS на сайте Ansible Galaxy (https://galaxy.ansible.com/Juniper/junos)

В следующем разделе мы поговорим о том, как написать собственный модуль для Ansible.

Написание собственного модуля

К этому моменту у вас могло сложиться впечатление, что сетевое администрирование в Ansible во многом зависит от того, удастся ли вам найти подходящий модуль для своей задачи. В этом, несомненно, большая доля правды.

Модули дают возможность инкапсулировать взаимодействие между управляемым и управляющим хостами и позволяют нам сосредоточиться на логике операций. Мы уже оценили широкий спектр модулей, предоставляемых такими крупными производителями, как Cisco, Juniper и Arista.

Если взять в качестве примера модули Cisco Nexus, то, помимо специфических задач, таких как управление BGP-соседями (`nxos_bgp`) и AAA-серверами (`nxos_aaa_server`), большинство производителей предоставляют средства для выполнения произвольных информационных (`nxos_command`) и конфигурационных команд (`nxos_config`). Это покрывает большую часть сценариев использования.

В Ansible 2.5 упростился процесс именования и использования модулей для сбора фактов о сетевых устройствах.

Но что, если для используемого вами устройства нет модуля, выполняющего нужную вам задачу? В этом разделе вы увидите, как исправить эту ситуацию, написав собственный модуль.

Ваш первый модуль

Собственный модуль не обязательно должен быть сложным; на самом деле можно даже обойтись без Python. Но, поскольку мы уже знакомы с этим языком, мы будем применять его в своих модулях. Предположим, что модуль предназначен только для внутреннего использования и не будет включен в проект Ansible. Поэтому на данном этапе мы частично проигнорируем документацию и рекомендации по форматированию.

Если вы заинтересованы в разработке модулей, которые впоследствии могут стать частью Ansible, то ознакомьтесь с соответствующим руководством (https://docs.ansible.com/ansible/latest/dev_guide/developing_modules.html).

Если в папке со сценарием создать каталог `library`, Ansible автоматически включит его в список путей поиска модулей. Поэтому мы можем разместить наш модуль в этом каталоге и использовать его в сценарии. К пользовательским модулям предъявляется одно простое требование: все они должны возвращать вывод в формате JSON.

Как вы, наверное, помните, в главе 3 мы написали сценарий на Python `cisco_nxapi_2.py`, использующий NXAPI для взаимодействия с устройством NX-OS:

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

import requests
import json

url='http://172.16.1.90/ins'
switchuser='cisco'
switchpassword='cisco'

myheaders={'content-type':'application/json-rpc'}
payload=[
  {
    "jsonrpc": "2.0",
    "method": "cli",
    "params": {
      "cmd": "show version",
      "version": 1.2
    },
    "id": 1
  }
]
response = requests.post(url,data=json.dumps(payload), headers=myheade
rs,auth=(switchuser,switchpassword)).json()
print(response['result']['body']['sys_ver_str'])
```

Он просто выводит версию системы. Мы можем изменить последнюю строку, чтобы она выводила результат в формате JSON:

```
version = response['result']['body']['sys_ver_str']
print json.dumps({"version": version})
```

Для нормальной работы этого модуля в конфигурации устройства нужно включить поддержку nxapi (если вы этого еще не сделали в главе 3):

```
nx-osv-1(config)# feature nxapi
nx-osv-1(config)# nxapi http port 80
nx-osv-1(config)# nxapi sandbox
```

Поместим этот файл в папку `library`:

```
$ ls -a library/
.... custom_module_1.py
```

Чтобы вызвать этот модуль в сценарии `chapter5_14.yml`, воспользуемся плагином `action` (https://docs.ansible.com/ansible/latest/dev_guide/developing_plugins.html):

```
---
- name: Your First Custom Module
  hosts: localhost
  gather_facts: false
  connection: local

  tasks:
    - name: Show Version
      action: custom_module_1
      register: output

    - debug:
      var: output
```

Обратите внимание: как и в случае с SSH-соединением, этот модуль выполняется локально, отправляя исходящие API-вызовы. Этот сценарий выведет следующее:

```
$ ansible-playbook chapter5_14.yml
PLAY [Your First Custom Module] ************************************
***************************

TASK [Show Version] ************************************************
****************
ok: [localhost]

TASK [debug] *******************************************************
*************************************************
ok: [localhost] => {
    "output": {
        "changed": false,
        "failed": false,
        "version": "7.3(0)D1(1)"
    }
}
<опущено>
```

Таким образом, вы можете написать любой модуль, совместимый с API устройства, и Ansible примет любой сгенерированный им вывод в формате JSON.

Ваш второй модуль

Давайте расширим предыдущий модуль, воспользовавшись шаблонным кодом, который можно найти в документации для разработчиков Ansible (https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html). Создадим файл `custom_module_2.py`, который будет принимать ввод из сценария Ansible.

Для начала импортируем шаблонный код из `ansible.module_utils.basic`:

```
from ansible.module_utils.basic import AnsibleModule
if __name__ == '__main__':
    main()
```

Дальше определим главную функцию, где будет находиться наш код. Модуль `AnsibleModule`, который мы уже импортировали, предоставляет много универсальных операций для обработки возвращаемых значений и разбора аргументов. В следующем примере мы разберем три аргумента, `host`, `username` и `password`, и сделаем их обязательными полями:

```
def main():
    module = AnsibleModule(
      argument_spec = dict(
        host = dict(required=True),
        username = dict(required=True),
        password = dict(required=True)
      )
    )
```

Эти значения можно извлечь и использовать в нашем коде:

```
device = module.params.get('host')
username = module.params.get('username')
password = module.params.get('password')
url='http://' + host + '/ins'
switchuser=username
switchpassword=password
```

Наконец, добавим код, завершающий выполнение и возвращающий значение:

```
module.exit_json(changed=False, msg=str(data))
```

Этот сценарий Ansible почти ничем не отличается от предыдущего, только теперь мы можем передавать ему значения для разных устройств:

```
tasks:
  - name: Show Version
    action: custom_module_2 host="172.16.1.142" username="cisco"
password="cisco"
    register: output
```

Этот сценарий выведет все то же самое, что и предыдущий. Но поскольку наш модуль теперь принимает аргументы, его могут использовать другие люди, не зная о нем никаких подробностей. Они могут указать в своем сценарии Ansible собственные имя пользователя, пароль и IP-адрес хоста.

Это вполне рабочий, но незаконченный модуль. Например, мы не проверяем ошибки и не предоставляем инструкций по его использованию. Но мы увидели, насколько просто написать собственный модуль. Еще один результат: мы теперь знаем, как превратить существующие сценарии на Python в модули для Ansible.

Резюме

В этой главе мы много всего обсудили. Например, вы познакомились с условными выражениями, циклами и шаблонами, узнали, как улучшить масштабируемость сценариев Ansible с помощью переменных хостов и групп, инструкций `include` и ролей, как защитить сценарии с помощью Ansible Vault. И в конце мы написали модули на Python.

Ansible — очень гибкий фреймворк на языке Python, его можно использовать для автоматизации сетей. Он предоставляет еще один уровень абстракции, имеющий мало общего со сценариями на основе Pexpect и API. Он декларативный по своей природе, позволяет выразительнее описывать наши намерения и может стать для вас идеальным фреймворком, помогающим сэкономить время и силы.

В главе 6 мы поговорим о сетевой безопасности в Python.

6

Сетевая безопасность с использованием Python

Писать на тему сетевой безопасности очень непросто. И проблема не в технической сложности, а в правильном выборе охватываемой области. Сфера сетевой безопасности чрезвычайно обширна и пронизывает все семь уровней модели OSI. Речь может идти и о перехвате информации на физическом уровне, и об уязвимостях в транспортных протоколах, и об атаках вида «человек посередине» на прикладном уровне. Эта проблема усложняется обнаружением все новых уязвимостей, которые, как иногда может показаться, становятся чем-то обыденным. И это не говоря уже о таком аспекте сетевой безопасности, как социальная инженерия.

По этой причине я бы хотел сразу определить рамки обсуждения в этой главе. Как и прежде, основное внимание будет уделено использованию Python для защиты сетевых устройств на сетевом и транспортном уровнях модели OSI. Мы рассмотрим инструменты на языке Python, подходящие для обеспечения безопасности отдельных сетевых устройств, а также использование Python для связывания разных компонентов. Надеюсь, применение Python на разных уровнях модели OSI позволит нам выработать общий подход к сетевой безопасности.

Эта глава охватывает такие темы, как:

- подготовка лаборатории;
- тестирование безопасности с помощью Python Scapy;
- списки доступа;

- ретроспективный анализ на основе Syslog и *UFW (Uncomplicated Firewall)* с использованием Python;
- другие инструменты: списки фильтрации MAC-адресов, приватные интерфейсы VLAN и Python-пакет для работы с Iptables.

Подготовка лаборатории

Здесь нам понадобятся устройства несколько другого рода, чем те, с которыми мы имели дело ранее. В предыдущих главах мы изолировали устройство, чтобы сосредоточиться на теме. В этой главе наша лаборатория будет содержать чуть больше устройств, что позволит нам проиллюстрировать возможности инструментов. Важна будет информация о соединении и операционной системе, так как на ее основе мы выбираем средства безопасности. Например, если мы хотим применить список доступа, чтобы защитить сервер, нам нужно иметь представление о топологии сети и о том, откуда устанавливается клиентское соединение. Соединения с хостами под управлением Ubuntu немного отличаются от тех, которые мы видели до сих пор, поэтому позже при необходимости вы можете сверяться с этим разделом.

Мы используем тот же инструмент Cisco VIRL с четырьмя узлами: двумя хостами и двумя сетевыми устройствами. Если вам нужно освежить навыки работы с Cisco VIRL, перечитайте главу 2, в которой мы познакомились с этой системой (рис. 6.1).

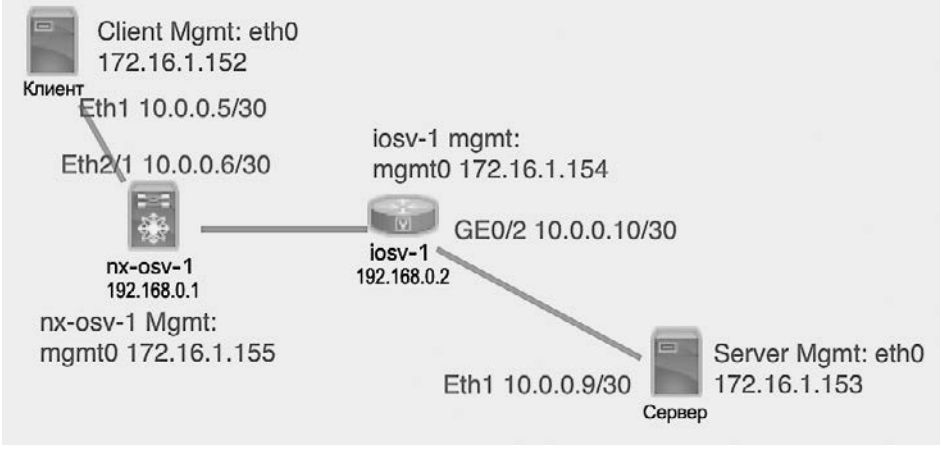


Рис. 6.1. Топология лаборатории

IP-адреса в вашей лаборатории могут отличаться от указанных выше. Мы привели их для того, чтобы вам было легче ориентироваться в примерах кода, представленных в этой главе.

Следуя этой иллюстрации, мы переименуем верхний и нижний хосты в Client и Server соответственно. Это похоже на то, как интернет-клиент пытается обратиться к корпоративному серверу внутри нашей сети. Мы снова выберем пункт Shared flat network (Разделяемая плоская сеть) в списке Management Network (Управляющая сеть), чтобы получить возможность управлять устройствами по дополнительному каналу (рис. 6.2).

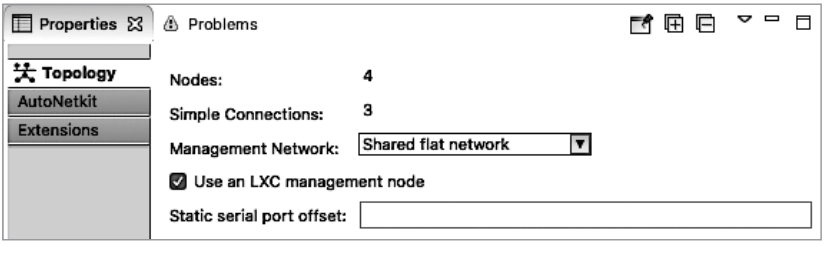


Рис. 6.2. Вариант управляющей сети в лаборатории

В этом примере клиентский хост должен быть доступен извне через VMnet2. В своей лаборатории я использую внешний USB-интерфейс, подключенный к моему хосту ESXi, предоставляющему такое соединение. Вам также может понадобиться добавить внешние DNS-серверы в /etc/resolveconf/resolv.conf.d/base:

```
cisco@Client:~$ cat /etc/resolvconf/resolv.conf.d/
base
nameserver 8.8.8.8
nameserver 8.8.4.4
```

В двух коммутаторах в качестве протокола внутреннего шлюза (Interior Gateway Protocol, IGP) используется алгоритм маршрутизации *OSPF (Open Shortest Path First)*, и оба устройства находятся в зоне `0`. Протокол BGP включен по умолчанию, и в обоих случаях используется AS 1.

Согласно сгенерированной конфигурации, интерфейсы, соединенные с хостами под управлением Ubuntu, находятся в OSPF-зоне `1`, поэтому они представлены как межзональные маршруты (inter-area routes). Ниже показана конфигурация для NX-OSv (устройства IOSv имеют аналогичную конфигурацию и вывод):

```
interface Ethernet2/1
  description to Client
  no switchport
  mac-address fa16.3e00.0001
  ip address 10.0.0.6/30
  ip router ospf 1 area 0.0.0.0
  no shutdown
!
interface Ethernet2/2
  description to iosv-1
  no switchport
  mac-address fa16.3e00.0002
  ip address 10.0.0.14/30
  ip router ospf 1 area 0.0.0.0
  no shutdown
!
nx-osv-1# sh ip route
<опущено>
10.0.0.8/30, ubest/mbest: 1/0
    *via 10.0.0.13, Eth2/2, [110/41], 14:10:28, ospf-1, intra
192.168.0.2/32, ubest/mbest: 1/0
    *via 10.0.0.13, Eth2/2, [110/41], 14:10:28, ospf-1, intra
<опущено>
```

Ниже можно видеть OSPF-соседа и вывод BGP для NX-OSv (IOSv имеет похожий вывод):

```
nx-osv-1# sh ip ospf neighbors
 OSPF Process ID 1 VRF default
 Total number of neighbors: 1
 Neighbor ID     Pri State            Up Time  Address          Interface
 192.168.0.2       1 FULL/DR          14:12:31 10.0.0.13        Eth2/2
!
nx-osv-1# sh ip bgp summary
BGP summary information for VRF default, address family IPv4 Unicast
BGP router identifier 192.168.0.1, local AS number 1
BGP table version is 5, IPv4 Unicast config peers 1, capable peers 1
2 network entries and 2 paths using 288 bytes of memory
BGP attribute entries [2/288], BGP AS path entries [0/0]
BGP community entries [0/0], BGP clusterlist entries [0/0]

Neighbor        V    AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down
State/PfxRcd
192.168.0.2     4     1     936     857        5    0    0 14:12:33 1
```

Хосты в нашей сети работают под управлением ОС Ubuntu 16.04; она похожа на Ubuntu VM 18.04, которую мы использовали до этого момента:

```
cisco@Server:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 16.04.3 LTS
Release: 16.04
Codename: xenial
```

Оба хоста с Ubuntu оснащены двумя сетевыми интерфейсами, `eth0` и `eth1`. Первый соединен с управляющей сетью (172.16.1.0/24), а второй — с сетевыми устройствами (10.0.0.x/30). Маршруты, ведущие к петлевому адресу устройства, напрямую соединены с сетевым блоком, и сети удаленных хостов статически направлены к `eth1` так, чтобы маршрут по умолчанию вел к управляющей сети:

```
cisco@Client:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref   Use
Iface
0.0.0.0         172.16.1.254    0.0.0.0         UG    0      0     0
eth0
10.0.0.0        10.0.0.6        255.0.0.0       UG    0      0     0
eth1
10.0.0.4        0.0.0.0         255.255.255.252 U     0      0     0
eth1
172.16.1.0      0.0.0.0         255.255.255.0   U     0      0     0
eth0
192.168.0.0     10.0.0.6        255.255.255.248 UG    0      0     0
eth1
```

Чтобы проверить путь от клиента к серверу, используем утилиту `ping` и отследим маршрут. Так мы убедимся в том, что трафик между нашими хостами проходит через сетевые устройства, а не по стандартному маршруту:

```
# IP-адрес серверного интерфейса Eth1 10.0.0.9/30
cisco@Server:~$ ifconfig eth1
eth1      Link encap:Ethernet  HWaddr fa:16:3e:68:bb:ce
          inet addr:10.0.0.9  Bcast:10.0.0.11  Mask:255.255.255.252
<опущено>

# эхо-запрос с IP-адреса клиентского интерфейса Eth1 (10.0.0.5/30) на сервер
cisco@Client:~$ ifconfig eth1
eth1      Link encap:Ethernet  HWaddr fa:16:3e:7c:75:ec
          inet addr:10.0.0.5  Bcast:10.0.0.7  Mask:255.255.255.252
<опущено>

cisco@Client:~$ ping -c 1 10.0.0.9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=1 ttl=62 time=7.00 ms
--- 10.0.0.9 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.007/7.007/7.007/0.000 ms

# Трассируем маршрут от клиента к серверу
cisco@Client:~$ traceroute 10.0.0.9
traceroute to 10.0.0.9 (10.0.0.9), 30 hops max, 60 byte packets
 1  10.0.0.6 (10.0.0.6)  5.694 ms  10.632 ms  10.599 ms
 2  10.0.0.13 (10.0.0.13)  13.078 ms  19.132 ms  19.101 ms
 3  10.0.0.9 (10.0.0.9)  14.929 ms  19.026 ms  19.004 ms
```

Отлично! Мы настроили лабораторию; теперь познакомимся с некоторыми средствами и мерами безопасности на основе Python.

Python Scapy

Scapy (https://scapy.net) — это мощная интерактивная программа на языке Python для генерации сетевых пакетов. Если не считать некоторые дорогие коммерческие аналоги, существует не так много инструментов с похожими возможностями, насколько мне известно. Это один из моих любимых проектов в мире Python.

Основное преимущество программы Scapy: она позволяет создать собственный сетевой пакет на очень низком уровне. По словам создателя Scapy:

> *«Scapy — мощная интерактивная программа для управления пакетами. Она способна конструировать или декодировать пакеты самых разных протоколов, отправлять их по сети, перехватывать, сопоставлять запросы и ответы и многое другое... большинство других инструментов не позволяет создавать то, что не предусмотрено их авторами. Эти инструменты были созданы для конкретной цели и не могут отклоняться от нее».*

Познакомимся с этим инструментом.

Установка Scapy

В попытке внедрить поддержку Python 3 проект Scapy пошел по интересному пути. В 2015 году для реализации поддержки Python 3 от Scapy 2.2.0 отпочковалась независимая версия, получившая название Scapy3k. В этой книге мы используем основную кодовую базу оригинального проекта Scapy. Если вы читали предыдущие издания и использовали версию Scapy, совместимую только с Python 2, то посмотрите, как разные выпуски этого инструмента поддерживают Python 3 (рис. 6.3).

Поскольку мы хотим генерировать пакеты на клиенте и отправлять их на сервер, Scapy следует установить на клиентской стороне:

```
cisco@Client:~$ git clone github.com/secdev/scapy.git
cisco@Client:~$ cd scapy/
cisco@Client:~/scapy$ sudo python3 setup.py install
```

Вслед за установкой мы запустим интерактивную оболочку Scapy, введя в командной строке `scapy` (рис. 6.4).

Python versions support

	Python 2		Python 3		
Scapy version	Python 2.5-2.6	Python 2.7	Python 3.4-3.6	Python 3.7	Python 3.8
2.2.X					
2.3.3					
2.4.0					
2.4.2					

Рис. 6.3. Поддержка версий Python (источник: https://scapy.net/download/)

Подробнее о том, как поддержка Python 3 появилась в Scapy. Все началось с ответвления независимой версии от Scapy 2.2.0 в 2015 году. Проект был назван Scapy3k. Это ответвление разошлось с основной кодовой базой Scapy. В первом издании данной книги история на этом заканчивалась. Позже возникло недоразумение по поводу пакета python3-scapy в PyPI и официальной поддержки Scapy. Основная цель данной главы — изучение этого инструмента. Поэтому я решил использовать более старую его версию, основанную на Python 2.

```
cisco@Client:~/scapy$ sudo scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
INFO: Can't import python-cryptography v1.7+. Disabled WEP decryption/encryption. (Dot11)
INFO: Can't import python-cryptography v1.7+. Disabled IPsec encryption/authentication.
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

                    aSPY//YASa
             apyyyyCY//////////YCa       |
            sY//////YSpcs  scpCY//Pp     | Welcome to Scapy
 ayp ayyyyyyySCP//Pp           syY//C    | Version 2.4.3.dev61
 AYAsAYYYYYYYY///Ps              cY//S   |
         pCCCCY//p          cSSps y//Y   | https://github.com/secdev/scapy
         SPPPP///a          pP///AC//Y   |
              A//A            cyP////C   | Have fun!
              p///Ac            sC///a   |
              P////YCpc           A//A   | Craft packets before they craft
       scccccp///pSP///p          p//Y   | you.
      sY/////////y  caa           S//P   |                      -- Socrate
       cayCyayP//Ya              pY/Ya   |
        sY/PsY////YCc           aC//Yp
         sc  sccaCY//PCypaapyCP//YSs
                  spCPY//////YPSps
                       ccaacs

>>>
```

Рис. 6.4. Тестирование Python Scapy

Простая проверка доступности библиотеки Scapy в Python 3:

```
cisco@Client:~$ python3
Python 3.5.2 (default, Jul 10 2019, 11:58:48)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> exit()
```

Отлично! Пакет Scapy установлен и доступен для использования в интерпретаторе Python. В следующем разделе вы увидите, как можно использовать интерактивную оболочку.

Интерактивные примеры

В первом примере мы сгенерируем на клиенте ICMP-пакет и отправим его серверу. На серверной стороне мы воспользуемся утилитой `tcpdump` с фильтром хостов, чтобы увидеть этот пакет:

```
## Сторона клиента
cisco@Client:~/scapy$ sudo scapy
>>> send(IP(dst="10.0.0.9")/ICMP())
.
Sent 1 packets.

# Сторона сервера
cisco@Server:~$ sudo tcpdump -i eth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
17:19:24.812184 IP 10.0.0.5 > 10.0.0.9: ICMP echo request, id 0, seq 0,
length 8
17:19:24.812205 IP 10.0.0.9 > 10.0.0.5: ICMP echo reply, id 0, seq 0,
length 8
```

Как видите, это очень простой процесс. Scapy позволяет сформировать пакет, добавляя заголовки протоколов через косую черту (`/`).

Функция `send` работает на сетевом уровне, поэтому вам не нужно беспокоиться о маршрутизации и физической адресации. У нее есть альтернатива, `sendp()`, которая работает на канальном уровне; это означает, что вам нужно будет указать интерфейс и протокол соединения.

Попробуем захватить ответный пакет с помощью функции `send-request (sr)`. Воспользуемся специальной версией `sr` с именем `sr1`, которая возвращает только один пакет:

```
>>> p = sr1(IP(dst="10.0.0.9")/ICMP())
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
>>> p
<IP  version=4 ihl=5 tos=0x0 len=28 id=44710 flags= frag=0 ttl=62
proto=icmp chksum=0xba2d src=10.0.0.9 dst=10.0.0.5 |<ICMP  type=echoreply
code=0 chksum=0xffff id=0x0 seq=0x0 |>>
```

Функция `sr()` возвращает кортеж со списками пакетов, на которые был и не был получен ответ:

```
>>> p = sr(IP(dst="10.0.0.9")/ICMP())
.Begin emission:
.....Finished sending 1 packets.
*
Received 7 packets, got 1 answers, remaining 0 packets
>>> type(p)
<class 'tuple'>
```

Заглянем внутрь этого кортежа:

```
>>> ans, unans = sr(IP(dst="10.0.0.9")/ICMP())
.Begin emission:
...Finished sending 1 packets.
..*
Received 7 packets, got 1 answers, remaining 0 packets
>>> type(ans)
<class 'scapy.plist.SndRcvList'>
>>> type(unans)
<class 'scapy.plist.PacketList'>
```

Если вывести список пакетов, на которые был получен ответ, то мы увидим, что это еще один кортеж, содержащий отправленные и полученные пакеты:

```
>>> for i in ans:
...     print(type(i))
...
<class 'tuple'>
>>>
>>>
>>> for i in ans:
...     print(i)
...
(<IP  frag=0 proto=icmp dst=10.0.0.9 |<ICMP  |>>, <IP  version=4 ihl=5
tos=0x0 len=28 id=19027 flags= frag=0 ttl=62 proto=icmp chksum=0x1e81
src=10.0.0.9 dst=10.0.0.5 |<ICMP  type=echo-reply code=0 chksum=0xffff
id=0x0 seq=0x0 |>>)
```

Scapy также поддерживает конструирование пакетов для протоколов прикладного уровня, таких как DNS-запросы. В следующем примере мы обращаемся

к публичному DNS-серверу, чтобы получить IP-адрес доменного имени `www.google.com`:

```
>>> p = sr1(IP(dst="8.8.8.8")/UDP()/DNS(rd=1,qd=DNSQR(qname="www.google.
com")))
Begin emission:
......Finished sending 1 packets.
..........*
Received 17 packets, got 1 answers, remaining 0 packets
>>> p
<IP  version=4 ihl=5 tos=0x20 len=76 id=17713 flags= frag=0 ttl=121
proto=udp chksum=0x28c5 src=8.8.8.8 dst=192.168.2.211 |<UDP  sport=domain
dport=domain len=56 chksum=0xa9db |<DNS  id=0 qr=1 opcode=QUERY aa=0
tc=0 rd=1 ra=1 z=0 ad=0 cd=0 rcode=ok qdcount=1 ancount=1 nscount=0
arcount=0 qd=<DNSQR  qname='www.google.com.' qtype=A qclass=IN |>
an=<DNSRR  rrname='www.google.com.' type=A rclass=IN ttl=274 rdlen=None
rdata=216.58.217.36 |> ns=None ar=None |>>>
```

Рассмотрим другие возможности Scapy. Начнем с захвата пакетов.

Захват пакетов с помощью Scapy

В процессе устранения неполадок нам, сетевым инженерам, постоянно приходится захватывать пакеты, проходящие через соединение. Обычно для этого используется Wireshark или похожие инструменты, но Scapy тоже позволяет легко выполнять захват пакетов:

```
>>> a = sniff(filter="icmp", count=5)
>>> a.show()
0000 Ether / IP / ICMP 192.168.2.211 > 8.8.8.8 echo-request 0 / Raw
0001 Ether / IP / ICMP 8.8.8.8 > 192.168.2.211 echo-reply 0 / Raw
0002 Ether / IP / ICMP 192.168.2.211 > 8.8.8.8 echo-request 0 / Raw
0003 Ether / IP / ICMP 8.8.8.8 > 192.168.2.211 echo-reply 0 / Raw
0004 Ether / IP / ICMP 192.168.2.211 > 8.8.8.8 echo-request 0 / Raw
```

Можно просматривать пакеты более подробно, даже в исходном формате:

```
>>> for packet in a:
...     print(packet.show())
...
###[ Ethernet ]###
  dst= 70:4f:57:94:7f:86
  src= 5e:00:00:02:00:00
  type= IPv4
###[ IP ]###
     version= 4
     ihl= 5
     tos= 0x0
     len= 84
```

```
      id= 1856
      flags= DF
      frag= 0
      ttl= 64
      proto= icmp
      chksum= 0x5fde
      src= 192.168.2.211
      dst= 8.8.8.8
      \options\
###[ ICMP ]###
         type= echo-request
         code= 0
         chksum= 0x4616
         id= 0x4a84
         seq= 0x1
###[ Raw ]###
            load= 'k\x9a\x8f]\x00\x00\x00\x00\xac\x99\x01\x00\x00\x00\
            x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\
            x1d\x1e\x1f!"#$%&\'()*+,-./01234567'
<опущено>
```

Итак, мы познакомились с основными принципами работы Scapy. Теперь посмотрим, как этот инструмент применяется для проверки безопасности сети.

Сканирование TCP-портов

Любая попытка взлома почти всегда начинается с поиска в сети открытых сервисов; это позволяет атаковать определенную цель. Конечно, чтобы обслуживать наших клиентов, нам необходимо открыть некоторые порты; это риск, который мы должны принять. Вместе с тем мы должны закрыть все остальные порты, которые могли бы послужить дополнительными целями для атак. Мы можем использовать Scapy для сканирования нашего собственного хоста на предмет открытых TCP-портов.

Отправим пакет `SYN` и посмотрим, вернет ли сервер в ответ `SYN-ACK`. Начнем с TCP-порта 23, принадлежащего Telnet:

```
>>> p = sr1(IP(dst="10.0.0.9")/TCP(sport=666,dport=23,flags="S"))
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
>>> p.show()
###[ IP ]###
  version= 4
  ihl= 5
  tos= 0x0
  len= 40
```

```
  id= 14089
  flags= DF
  frag= 0
  ttl= 62
  proto= tcp
  chksum= 0xf1b9
  src= 10.0.0.9
  dst= 10.0.0.5
  \options\
###[ TCP ]###
     sport= telnet
     dport= 666
     seq= 0
     ack= 1
     dataofs= 5
     reserved= 0
     flags= RA
     window= 0
     chksum= 0x9911
     urgptr= 0
     options= []
```

Обратите внимание, что в данном случае на попытку установить соединение с TCP-портом 23 сервер ответил пакетом `RESET+ACK`, потому что на хосте нет Telnet. Однако TCP-порт 22 (SSH) открыт, поэтому для него возвращается пакет `SYN-ACK`:

```
>>> p = sr1(IP(dst="10.0.0.9")/TCP(sport=666,dport=22,flags="S"))
>>> p = sr1(IP(dst.show()
###[ IP ]###
  version= 4
<опущено>
  proto= tcp
  chksum= 0x28bf
  src= 10.0.0.9
  dst= 10.0.0.5
  \options\
###[ TCP ]###
     sport= ssh
     dport= 666
     seq= 1671401418
     ack= 1
     dataofs= 6
     reserved= 0
     flags= SA
<опущено>
```

Мы также можем просканировать диапазон портов от `20` до `22`; заметьте, что мы отправляем/принимаем сразу несколько пакетов, поэтому здесь используется функция `sr()`, а не `sr1()` (которая отправляет/принимает одиночные пакеты):

```
>>> ans,unans = sr(IP(dst="10.0.0.9")/TCP(sport=666,dport=(20,22),flags=
"S"))
>>> for i in ans:
...     print(i)
...
(<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666 dport=ftp_data
flags=S |>>, <IP  version=4 ihl=5 tos=0x0 len=40 id=59720 flags=DF
frag=0 ttl=62 proto=tcp chksum=0x3f7a src=10.0.0.9 dst=10.0.0.5 |<TCP
sport=ftp_data dport=666 seq=0 ack=1 dataofs=5 reserved=0 flags=RA
window=0 chksum=0x9914 urgptr=0 |>>)
(<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666 dport=ftp
flags=S |>>, <IP  version=4 ihl=5 tos=0x0 len=40 id=59721 flags=DF
frag=0 ttl=62 proto=tcp chksum=0x3f79 src=10.0.0.9 dst=10.0.0.5 |<TCP
sport=ftp dport=666 seq=0 ack=1 dataofs=5 reserved=0 flags=RA window=0
chksum=0x9913 urgptr=0 |>>)
(<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666 dport=ssh flags=S
|>>, <IP  version=4 ihl=5 tos=0x0 len=44 id=0 flags=DF frag=0 ttl=62
proto=tcp chksum=0x28bf src=10.0.0.9 dst=10.0.0.5 |<TCP  sport=ssh
dport=666 seq=3932520059 ack=1 dataofs=6 reserved=0 flags=SA window=29200
chksum=0xa666 urgptr=0 options=[('MSS', 1460)] |>>)
>>>
```

Вместо отдельного хоста можно просканировать целую сеть. Как видите, в диапазоне `10.0.0.8/29` ответ `SA` вернули хосты `10.0.0.9`, `10.0.0.10` и `10.0.0.14`; это два сетевых устройства и сервер:

```
>>> ans,unans = sr(IP(dst="10.0.0.8/29")/TCP(sport=666,dport=(22),flags=
"S"))
>>> for i in ans:
... print(i)
...
(<IP  frag=0 proto=tcp dst=10.0.0.14 |<TCP  sport=666 dport=ssh flags=S
|>>, <IP  version=4 ihl=5 tos=0x0 len=44 id=7289 flags= frag=0 ttl=64
proto=tcp chksum=0x4a41 src=10.0.0.14 dst=10.0.0.5 |<TCP  sport=ssh
dport=666 seq=1652640556 ack=1 dataofs=6 reserved=0 flags=SA window=17292
chksum=0x9029 urgptr=0 options=[('MSS', 1444)] |>>)
(<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666 dport=ssh flags=S
|>>, <IP  version=4 ihl=5 tos=0x0 len=44 id=0 flags=DF frag=0 ttl=62
proto=tcp chksum=0x28bf src=10.0.0.9 dst=10.0.0.5 |<TCP  sport=ssh
dport=666 seq=898054835 ack=1 dataofs=6 reserved=0 flags=SA window=29200
chksum=0x9f0d urgptr=0 options=[('MSS', 1460)] |>>)
(<IP  frag=0 proto=tcp dst=10.0.0.10 |<TCP  sport=666 dport=ssh flags=S
|>>, <IP  version=4 ihl=5 tos=0x0 len=44 id=38021 flags= frag=0 ttl=254
proto=tcp chksum=0x1438 src=10.0.0.10 dst=10.0.0.5 |<TCP  sport=ssh
dport=666 seq=371720489 ack=1 dataofs=6 reserved=0 flags=SA window=4128
chksum=0x5d82 urgptr=0 options=[('MSS', 536)] |>>)
>>>
```

Основываясь на этих результатах, напишем простой сценарий, `scapy_tcp_scan_1.py`, пригодный для многократного использования:

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

from scapy.all import *
import sys

def tcp_scan(destination, dport):
    ans, unans = sr(IP(dst=destination)/TCP(sport=666,dport=dport,flags="S"))
    for sending, returned in ans:
        if 'SA' in str(returned[TCP].flags):
            return destination + " port " + str(sending[TCP].dport) + " is
open."
        else:
            return destination + " port " + str(sending[TCP].dport) + " is
not open."

def main():
    destination = sys.argv[1]
    port = int(sys.argv[2])
    scan_result = tcp_scan(destination, port)
    print(scan_result)

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

Сначала мы импортируем `scapy` и модуль `sys` для приема аргументов. Функция `tcp_scan()` похожа на код, который мы использовали до сих пор; разница лишь в том, что мы можем получить ввод из аргументов командной строки и затем вызвать `tcp_scan()` внутри `main()`.

Помните, что доступ к низкоуровневой сети требует привилегий суперпользователя, поэтому сценарий нужно запускать с помощью `sudo`. Просканируем с его помощью порты `22` (SSH) и `80` (HTTP):

```
cisco@Client:~$ sudo python3 scapy_tcp_scan_1.py "10.0.0.14" 22
Begin emission:
......Finished sending 1 packets.
*
Received 7 packets, got 1 answers, remaining 0 packets
10.0.0.14 port 22 is open.

cisco@Client:~$ sudo python3 scapy_tcp_scan_1.py "10.0.0.14" 80
Begin emission:
...Finished sending 1 packets.
*
Received 4 packets, got 1 answers, remaining 0 packets
10.0.0.14 port 80 is not open.
```

Это был относительно длинный пример сценария для сканирования TCP-портов, который показал, чего можно достичь, генерируя собственные пакеты с помощью Scapy. Мы протестировали отдельные шаги в интерактивной оболочке и в конце оформили их в виде простого сценария. Теперь рассмотрим примеры использования Scapy для проверки безопасности.

Коллекция пакетов для проверки связи

Представьте, что наша сеть состоит из компьютеров под управлением Windows, Unix и Linux и пользователи могут подключать к ней собственные устройства в рамках политики *BYOD (Bring Your Own Device — «принеси свое собственное устройство»)*; мы не знаем, поддерживает ли тот или иной хост ICMP-пакеты эхо-запросов (ping). Мы можем написать сценарий, использующий три распространенных типа пакетов для проверки связи: ICMP, TCP и UDP. Назовем его `scapy_ping_collection.py`:

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

from scapy.all import *

def icmp_ping(destination):
    # обычный эхо-запрос ICMP
    ans, unans = sr(IP(dst=destination)/ICMP())
    return ans

def tcp_ping(destination, dport):
    ans, unans = sr(IP(dst=destination)/TCP(dport=dport,flags="S"))
    return ans

def udp_ping(destination):
    ans, unans = sr(IP(dst=destination)/UDP(dport=0))
    return ans

def answer_summary(ans):
    for send, recv in ans:
    print(recv.sprintf("%IP.src% is alive"))
```

Затем мы можем послать все три типа сетевых пакетов с помощью одного сценария:

```
def main():
    print("** ICMP Ping **")
    ans = icmp_ping("10.0.0.13-14")
    answer_summary(ans)
    print("** TCP Ping ***")
    ans = tcp_ping("10.0.0.13", 22)
    answer_summary(ans)
    print("** UDP Ping ***")
    ans = udp_ping("10.0.0.13-14")
    answer_summary(ans)

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

Возможность формировать собственные пакеты позволяет выполнять именно те операции и проверки, которые вы хотите. Создавать пакеты с помощью Scapy можно и с целью проверки безопасности сети.

Распространенные атаки

В этом примере вы узнаете, как сформировать пакет для классических атак *Ping of Death* (https://ru.wikipedia.org/wiki/Ping_of_death) и *LAND* (https://wikiboard.ru/wiki/LAND). Эти проверки позволяют выявить возможность проникновения в сеть, и раньше их можно было выполнять только с помощью платного коммерческого ПО. Scapy дает возможность выполнить аналогичные проверки с сохранением полного контроля за происходящим и добавлять новые проверки в будущем.

Первая атака заключается в отправке хосту поддельного IP-заголовка со значением длины, равным 2, или номером версии 3:

```
def malformed_packet_attack(host):
       send(IP(dst=host, ihl=2, version=3)/ICMP())
```

Функция `ping_of_death_attack` отправляет обычный ICMP-пакет с объемом полезной нагрузки больше 65 535 байт:

```
def ping_of_death_attack(host):
    # https://ru.wikipedia.org/wiki/Ping_of_death
    send(fragment(IP(dst=host)/ICMP()/("X"*60000)))
```

Функция `land_attack` пытается перенаправить клиенту его же ответ, чтобы у его хоста исчерпались ресурсы:

```
def land_attack(host):
    # https://wikiboard.ru/wiki/LAND
    send(IP(src=host, dst=host)/TCP(sport=135,dport=135))
```

Это довольно старые уязвимости, которым современные операционные системы больше не подвержены. Ни одна из этих атак не нарушит работу нашего хоста с Ubuntu 16.04. Однако всегда обнаруживаются новые дыры в безопасности, и Scapy — отличный инструмент для тестирования собственных сетей и хостов, благодаря которому вам больше не нужно ждать, пока ваш поставщик предоставит вам подходящий инструмент для проверки. Это особенно актуально для так называемых атак нулевого дня (публикуемых без предупреждения), которые, похоже, становятся все более частым явлением в интернете. Обширные возможности Scapy нельзя уместить в одну главу, но, к счастью, существует множество открытых ресурсов, которые мы можем порекомендовать.

Ресурсы о Scapy

В этой главе мы уделили довольно много внимания работе со Scapy. Это отчасти вызвано тем, что я высоко ценю этот инструмент. Надеюсь, вы согласитесь со

мной, что Scapy пригодится любому сетевому инженеру. Что самое приятное, этот проект постоянно развивается активным сообществом пользователей.

Я настоятельно рекомендую ознакомиться с практическим руководством по Scapy (http://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial), а также с любой интересующей вас документацией.

Конечно, обеспечение сетевой безопасности не ограничивается формированием пакетов и тестированием уязвимостей. В следующем разделе мы поговорим об автоматизации списков доступа, которые повсеместно используются для защиты внутренних конфиденциальных ресурсов.

Списки доступа

Списки доступа к сети обычно служат первой линией защиты от вторжений и атак снаружи. В целом маршрутизаторы и коммутаторы обрабатывают пакеты намного быстрее, чем серверы, используя оборудование с высокоскоростной памятью, такой как *TCAM (Ternary Content-Addressable Memory — троичная ассоциативная память)*. Им не нужна информация прикладного уровня. Чтобы определить, следует ли пропускать пакет дальше, они проверяют лишь заголовки сетевого и транспортного уровней. Поэтому защита сетевых ресурсов обычно начинается со списков доступа в сетевых устройствах.

Списки доступа, как правило, стараются размещать как можно ближе к источнику (клиенту). Также логично, что мы доверяем хостам внутри нашей сети и с подозрением относимся к клиентам за ее пределами. Поэтому список доступа обычно применяется к сетевым интерфейсам, которые принимают внешний трафик. В условиях нашей лаборатории это означает, что список доступа следует применить ко входящему трафику на интерфейсе Ethernet2/1 хоста `nx-osv-1`, который напрямую соединен с клиентом.

Если у вас нет уверенности относительно направления трафика и размещения списка доступа, вот несколько советов.

- Думайте о списке доступа с точки зрения сетевого устройства.
- Сводите пакеты к исходному и конечному IP-адресам, используя в качестве примера один хост.
- В нашей лаборатории трафик от сервера к клиенту будет иметь исходящий IP-адрес `10.0.0.9` и конечный `10.0.0.5`.

- Исходящим и конечным IP-адресами в трафике от клиента к серверу будут `10.0.0.5` и `10.0.0.9` соответственно.

Очевидно, что все сети разные, и составление списка доступа зависит от услуг, предоставляемых вашим сервером. Но если речь идет о списке доступа для внешнего входящего трафика, вы должны:

- запретить прием пакетов, исходящих со специальных адресов, таких как `127.0.0.0/8` (RFC 3030);
- запретить прием пакетов, исходящих из диапазонов адресов локальных сетей, таких как `10.0.0.0/8` (RFC 1918);
- запретить прием пакетов, исходящих из вашего адресного пространства (в данном случае `10.0.0.4/30`);
- разрешить входящий трафик на TCP-порты 22 (SSH) и 80 (HTTP) хоста `10.0.0.9`;
- запретить все остальное.

Вот хороший список немаршрутизируемых сетей, которые следует блокировать: https://ipinfo.io/bogon.

Но понимание того, из чего должен состоять список доступа, — это лишь полдела. В следующем разделе вы узнаете, как его можно реализовать с помощью Ansible.

Реализация списков доступа с помощью Ansible

Для реализации списка доступа проще всего использовать Ansible. Мы уже познакомились с этим инструментом в предыдущих двух главах, но еще раз перечислим его преимущества в этой ситуации.

- **Простота администрирования.** Длинные списки доступа для удобства можно разбивать на части с помощью инструкции `include`. Затем с этими отдельными списками могут работать другие команды или владельцы сервиса.
- **Идемпотентность.** Мы можем запланировать регулярное выполнение сценария Ansible, и при этом будут вноситься только необходимые изменения.
- **Явное описание каждой задачи.** Мы можем отделить конструирование списков, а также применять списки доступа к соответствующим интерфейсам.

- **Повторное использование.** В будущем, если у нас появятся другие интерфейсы, направленные наружу, будет достаточно добавить их в список устройств — и правила доступа применятся к ним автоматически.
- **Расширяемость.** Вы сами увидите, что один и тот же сценарий Ansible можно использовать как для формирования списка доступа, так и для его применения к нужному интерфейсу. Мы можем начать с малого и затем, по мере расширения, разбивать код на отдельные сценарии, если понадобится.

Файл `host` имеет привычный вид. Для простоты разместим в нем переменные хостов:

```
[nxosv-devices]
nx-osv-1 ansible_host=172.16.1.155 ansible_username=cisco ansible_
password=cisco

[iosv-devices]
iosv-1 ansible_host=172.16.1.154 ansible_username=cisco ansible_
password=cisco
```

Объявим эти переменные в сценарии:

```
---
- name: Configure Access List
  hosts: "nxosv-devices"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ ansible_username }}"
      password: "{{ ansible_password }}"
      transport: cli
```

Чтобы сэкономить место, мы проиллюстрируем только запрет приема пакетов из адресных пространств локальных сетей (RFC 1918). Запрет диапазона RFC 3030 и нашего собственного адресного пространства настраивается точно так же. Обратите внимание, что наш сценарий не запрещает диапазон `10.0.0.0/8`, поскольку в нашей конфигурации в настоящее время используется сеть `10.0.0.0`. Конечно, мы могли бы сначала разрешить доступ для отдельного хоста, а затем запретить `10.0.0.0/8` в одной из следующих записей, но в этом примере мы просто проигнорируем этот диапазон:

```
tasks:
  - nxos_acl:
      name: border_inbound
```

```
        seq: 20
        action: deny
        proto: tcp
        src: 172.16.0.0/12
        dest: any
        log: enable
        state: present
        provider: "{{ cli }}"
    - nxos_acl:
        name: border_inbound
        seq: 30
        action: deny
        proto: tcp
        src: 192.168.0.0/16
        dest: any
        state: present
        log: enable
        provider: "{{ cli }}"
<опущено>
```

Мы разрешаем прием трафика через соединения, инициированные самим сервером. В конце мы используем явную инструкцию `deny ip any` с большим порядковым номером (`1000`), чтобы позже можно было вставить новые записи.

Затем применим список доступа к подходящему интерфейсу:

```
- name: apply ingress acl to Ethernet 2/1
  nxos_acl_interface:
    name: border_inbound
    interface: Ethernet2/1
    direction: ingress
    state: present
    provider: "{{ cli }}"
```

Список доступа для VIRL NX-OSv поддерживается только управляющим интерфейсом. Вы увидите предупреждение: Warning: ACL may not behave as expected since only one management interface is supported if you configure this ACL via the CLI (Внимание: если вы настраиваете этот список доступа из командной строки, то он может дать эффект, отличный от ожидаемого, потому что поддерживается только один управляющий интерфейс). Это нормально, так как я пытался лишь показать, как можно автоматизировать конфигурацию списков доступа.

Довольно много работы для одного списка доступа, не так ли? Опытному инженеру будет проще зайти на устройство и сконфигурировать все вручную. Но помните, что этот сценарий можно использовать повторно, поэтому в долгосрочной перспективе он сэкономит вам время.

Как показывает мой опыт, в длинных списках доступа записи, соответствующие отдельным сервисам, чередуются друг с другом. Таким спискам свойственно разрастаться, и со временем бывает очень сложно определить происхождение и назначение той или иной записи. Но их можно разбить на части, что существенно упрощает администрирование списков доступа.

Теперь выполним сценарий на хосте `nx-osv-1` и проверим результат:

```
$ ansible-playbook -i hosts access_list_nxosv.yml
PLAY [Configure Access List] ***************************************
*******************************
TASK [nxos_acl] ****************************************************
******************
ok: [nx-osv-1]
<опущено>
TASK [apply ingress acl to Ethernet 2/1] ***************************
*****************************************
changed: [nx-osv-1]
We should log in to the nx-osv-1 devices to verify the changes:
nx-osv-1# sh ip access-lists border_inbound

IP access list border_inbound
        20 deny tcp 172.16.0.0/12 any log
        30 deny tcp 192.168.0.0/16 any log
        40 permit tcp any 10.0.0.9/32 eq 22 log
        50 permit tcp any 10.0.0.9/32 eq www log
        60 permit tcp any any established log
        1000 deny ip any any log

nx-osv-1# sh run int eth 2/1
!
interface Ethernet2/1
  description to Client
  no switchport
  mac-address fa16.3e00.0001
  ip access-group border_inbound in
  ip address 10.0.0.6/30
  ip router ospf 1 area 0.0.0.0
  no shutdown
```

Это реализация списка доступа с IP-адресами, которая проверяет информацию на сетевом уровне модели OSI. В следующем подразделе речь пойдет о том, как ограничить доступ к устройству на канальном уровне.

Списки доступа по MAC-адресам

Если вы работаете на канальном уровне или используете в своих Ethernet-интерфейсах другие протоколы, помимо IP, то ограничивать доступность сети

для внешних хостов можно по их MAC-адресам. Это похоже на наш предыдущий пример, только вместо IP-адресов в списке доступа будут проверяться MAC-адреса. Как вы, наверное, помните, в MAC-адресах (или физических адресах) первые шесть шестнадцатеричных символов составляют *уникальный идентификатор организации (Organizationally Unique Identifier, OUI)*. Поэтому один и тот же механизм сопоставления позволяет запретить доступ определенным группам хостов.

Мы выполняем этот пример на хосте с IOSv и модулем ios_config. В старых версиях Ansible изменения применяются при каждом выполнении сценария. В новых версиях управляющий узел сначала проверяет необходимость изменений и только потом применяет их.

Файл `host` и начало сценария похожи на пример со списком доступа по IP-адресам, однако в разделе `tasks` используются другие модули и аргументы:

```
<опущено>
 tasks:
    - name: Deny Hosts with vendor id fa16.3e00.0000
      ios_config:
        lines:
          - access-list 700 deny fa16.3e00.0000 0000.00FF.FFFF
          - access-list 700 permit 0000.0000.0000 FFFF.FFFF.FFFF
        provider: "{{ cli }}"
    - name: Apply filter on bridge group 1
      ios_config:
        lines:
          - bridge-group 1
          - bridge-group 1 input-address-list 700
        parents:
          - interface GigabitEthernet0/1
        provider: "{{ cli }}"
```

Выполним этот плейбук на устройстве IOSv-1 и посмотрим результат:

```
$ ansible-playbook -i hosts access_list_mac_iosv.yml

TASK [Deny Hosts with vendor id fa16.3e00.0000] ************************
************************************************
changed: [iosv-1]

TASK [Apply filter on bridge group 1] **********************************
************************************************
changed: [iosv-1]
```

Зайдем на устройство, как мы это делали раньше, и убедимся, что изменения внесены:

```
iosv-1#sh run int gig 0/1
!
interface GigabitEthernet0/1
 description to nx-osv-1
 <опущено>
 bridge-group 1
 bridge-group 1 input-address-list 700
end
```

С популяризацией виртуальных сетей информация сетевого уровня иногда становится прозрачной для базовых виртуальных каналов. В таких ситуациях, если вам нужно ограничить доступ к этим интерфейсам, хорошим решением будет список доступа по MAC-адресам. В этом разделе для реализации таких списков на канальном и сетевом уровнях мы использовали Ansible. Теперь давайте поговорим о другом аспекте сетевой безопасности: как извлечь необходимую информацию из системного журнала (Syslog) с помощью Python.

Поиск в Syslog

Нам известно о множестве дыр в сетевой безопасности, остававшихся незакрытыми на протяжении длительного периода. В то время мы часто обнаруживали признаки подозрительной активности. Их можно было обнаружить в журнальных записях серверов и сетевых устройств. Эта активность не была обнаружена не из-за нехватки информации, а скорее по причине ее *избыточности*. Важные сведения, которые мы ищем, обычно теряются среди большого количества данных, в которых сложно разобраться.

Еще один хороший источник информации для серверов, помимо Syslog, — UFW. Это клиентский интерфейс для Iptables, серверного брандмауэра. UFW существенно упрощает управление правилами брандмауэра и записывает довольно много данных. Больше о UFW читайте в разделе «Другие инструменты».

В этом разделе мы попробуем использовать Python для поиска в Syslog интересующей нас активности. Конечно, искомый текст зависит от нашего устройства. Например, Cisco предоставляет перечень сообщений, по которым в Syslog можно найти любые нарушения правил, указанных в списке доступа. Он доступен по адресу http://www.cisco.com/c/en/us/about/security-center/identify-incidents-via-syslog.html.

Описание особенностей журналирования использования правил в списке доступа смотрите здесь: http://www.cisco.com/c/en/us/about/security-center/access-control-list-logging.html.

Для этого упражнения нам понадобится коммутатор Nexus с анонимизированным файлом Syslog, содержащим около 65 000 строк. Его можно взять в репозитории этой книги на GitHub:

```
$ wc -l sample_log_anonymized.log
65102 sample_log_anonymized.log
```

Мы вставили в него несколько сообщений из документации Cisco (http://www.cisco.com/c/en/us/support/docs/switches/nexus-7000-series-switches/118907-configure-nx7k-00.html) — именно их мы будем искать:

```
2014 Jun 29 19:20:57 Nexus-7000 %VSHD-5-VSHD_SYSLOG_CONFIG_I: Configured
from vty by admin on console0
2014 Jun 29 19:21:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1,
Dst IP: 172.16.10.10, Src Port: 0, Dst Port: 0, Src Intf: Ethernet4/1,
Pro tocol: "ICMP"(1), Hit-count = 2589

2014 Jun 29 19:26:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1, Dst IP: 172.16.10.10, Src Port: 0, Dst Port: 0, Src Intf:
Ethernet4/1, Protocol: "ICMP"(1), Hit-count = 4561
```

Мы рассмотрим простые примеры с регулярными выражениями. Если вы уже знакомы с модулем `re` из библиотеки Python, можете смело пропустить следующий раздел.

Поиск с помощью модуля регулярных выражений

Для поиска текста в нашем первом примере мы используем модуль поддержки регулярных выражений. Напишем простой цикл, который делает следующее:

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

import re, datetime

startTime = datetime.datetime.now()

with open('sample_log_anonymized.log', 'r') as f:
    for line in f.readlines():
        if re.search('ACLLOG-5-ACLLOG_FLOW_INTERVAL', line):
            print(line)

endTime = datetime.datetime.now()
elapsedTime = endTime - startTime
print("Time Elapsed: " + str(elapsedTime))
```

На поиск в этом файле журнала ушло примерно четыре сотых секунды:

```
$ python3 python_re_search_1.py
2014 Jun 29 19:21:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP: 10.1
0.10.1,

2014 Jun 29 19:26:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP: 10.1
0.10.1,

Time Elapsed: 0:00:00.047249
```

Для большей эффективности поисковое выражение рекомендуется скомпилировать. Но это не окажет существенного влияния, поскольку наш сценарий и так быстрый. А так как Python — интерпретируемый язык, это может даже замедлить сценарий. Однако при поиске по более крупным файлам компиляция может помочь, поэтому посмотрим, как это сделать:

```
searchTerm = re.compile('ACLLOG-5-ACLLOG_FLOW_INTERVAL')
with open('sample_log_anonymized.log', 'r') as f:
    for line in f.readlines():
        if re.search(searchTerm, line):
            print(line)
```

Время выполнения увеличилось:

```
Time Elapsed: 0:00:00.081541
```

Давайте немного расширим пример. Представьте, что у нас есть несколько файлов и поисковых выражений. Сделаем копию оригинального файла:

```
$ cp sample_log_anonymized.log sample_log_anonymized_1.log
```

Также добавим искомую строку `PAM: Authentication failure`. Чтобы выполнить поиск по обоим файлам, нам понадобится еще один цикл:

```
term1 = re.compile('ACLLOG-5-ACLLOG_FLOW_INTERVAL')
term2 = re.compile('PAM: Authentication failure')

fileList = ['sample_log_anonymized.log', 'sample_log_anonymized_1.
log']

for log in fileList:
    with open(log, 'r') as f:
        for line in f.readlines():
            if re.search(term1, line) or re.search(term2, line):
                print(line)
```

Поскольку поисковых выражений и сообщений стало больше, мы видим разницу в производительности:

```
$ python3 python_re_search_2.py
2016 Jun 5 16:49:33 NEXUS-A %DAEMON-3-SYSTEM_MSG: error: PAM:
Authentication failure for illegal user AAA from 172.16.20.170 -
sshd[4425]

2016 Sep 14 22:52:26.210 NEXUS-A %DAEMON-3-SYSTEM_MSG: error: PAM:
Authentication failure for illegal user AAA from 172.16.20.170 -
sshd[2811]

<опущено>

2014 Jun 29 19:21:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1,

2014 Jun 29 19:26:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1,
<опущено>

Time Elapsed: 0:00:00.330697
```

Конечно, любая оптимизация производительности — это стремление к недостижимому идеалу, и скорость работы сценария иногда зависит от вашего аппаратного обеспечения. Но суть в том, что вы должны регулярно проводить аудит своих файлов журналов с помощью Python, чтобы как можно раньше выявить признаки потенциального нарушения безопасности.

Мы рассмотрели некоторые ключевые способы улучшения сетевой безопасности с использованием Python, однако существуют и другие мощные инструменты, которые могут сделать этот процесс легче и эффективнее. В последнем разделе этой главы мы рассмотрим некоторые из них.

Другие инструменты

Существуют другие средства поддержки сетевой безопасности, которые можно использовать и автоматизировать с помощью Python. Давайте рассмотрим два самых распространенных.

Приватные VLAN

Виртуальные локальные компьютерные сети (Virtual Local Area Networks, VLAN) существуют с давних пор. Это, в сущности, широковещательный домен, в котором все хосты могут быть подключены к единому коммутатору, но при этом разделены на разные домены; это позволяет группировать их в зависимости

от того, какие из них видят друг друга через широковещательный канал. Давайте рассмотрим пример сети, поделенной на подсети. Например, если взять корпоративное здание, на каждом его этаже с большой долей вероятности будет отдельная подсеть: `192.168.1.0/24` на первом этаже, `192.168.2.0/24` на втором и т. д. В данном случае каждый этаж получает блок адресов `/24`. Это создает четкие границы в физической и логической сетях. Хосту, который хочет выйти за пределы собственной подсети, нужно сначала пройти через шлюз сетевого уровня, на котором в целях безопасности может быть список доступа.

Но что, если разные отделы находятся на одном этаже? Например, второй этаж отведен финансистам и менеджерам по продажам и вы не хотите размещать их хосты в одном широковещательном домене. Вы можете разделить подсеть, но это может оказаться утомительным процессом и нарушить уже готовую стандартную структуру. В этой ситуации поможет приватная сеть VLAN.

Приватные VLAN фактически позволяют разбить имеющуюся локальную сеть на подсети. Они делятся на три категории.

- **P-порт (Promiscuous port).** Этому порту позволено обмениваться кадрами канального уровня с любым другим портом VLAN; обычно этот порт соединен с маршрутизатором сетевого уровня.
- **I-порт (Isolated port).** Порту позволено общаться только с портами P, а они обычно принадлежат хостам, которые не должны взаимодействовать с другими хостами в той же локальной сети.
- **C-порт (Community port).** Этому порту позволено общаться с другими C-портами в том же «сообществе» (community), а также с портами P.

Мы снова можем обратиться к Ansible или любым другим инструментам на Python, подходящим для данной задачи. У вас уже должно хватать опыта и уверенности в своих силах, чтобы автоматизировать эту возможность, поэтому я не буду приводить здесь инструкции. Знакомство с VLAN пригодится в ситуациях, когда нужно дополнительно изолировать порты локальной сети на канальном уровне.

UFW и Python

Как уже говорилось, UFW — это клиентский интерфейс для Iptables на хостах с Ubuntu. Краткие инструкции по его установке и использованию:

```
$ sudo apt-get install ufw
$ sudo ufw status
```

```
$ sudo ufw default outgoing
$ sudo ufw allow 22/tcp
$ sudo ufw allow www
$ sudo ufw default deny incoming
```

Проверить состояние UFW можно так:

```
$ sudo ufw status verbose Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed) New
profiles: skip

To Action From
-- ------ ----
22/tcp ALLOW IN Anywhere
80/tcp ALLOW IN Anywhere
22/tcp (v6) ALLOW IN Anywhere (v6)
80/tcp (v6) ALLOW IN Anywhere (v6)
```

Преимущество UFW: это простой интерфейс для создания правил Iptables, которые было бы сложно описывать вручную. Чтобы сделать все еще проще, вместе с UFW можно использовать дополнительные инструменты на Python.

- Модуль Ansible UFW поможет оптимизировать наши операции. Подробности доступны по адресу https://docs.ansible.com/ansible/2.9/modules/ufw_module.html. Поскольку фреймворк Ansible написан на Python, мы можем даже заглянуть в исходный код модуля. Дополнительную информацию ищите на странице https://github.com/ansible-collections/community.general/blob/main/plugins/modules/system/ufw.py.
- Существуют модули-обертки UFW, предоставляющие API (https://gitlab.com/dhj/easyufw). Они могут упростить интеграцию, если вам нужно динамически изменять правила UFW в зависимости от определенных событий.
- Сам модуль UFW написан на Python. Следовательно, если вам нужно расширить имеющиеся наборы команд, вы можете применить свое знание этого языка. Больше информации здесь: https://launchpad.net/ufw.

UFW — хорошее средство защиты для ваших сетевых серверов.

Дополнительный материал

Python применяется во многих областях, связанных с безопасностью. Я хотел бы порекомендовать вам две книги:

- *O'Connor T. J.* Violent Python: A cookbook for hackers, forensic analysts, penetration testers and security engineers;
- *Джастин З., Тим А.* Black Hat Python. Программирование для хакеров и пентестеров. СПб.: Питер, 2022 (Black Hat Python: Python programming for hackers and pentesters; Justin Seitz).

Я сам активно использую Python в нашей исследовательской работе по *распределенным атакам на отказ в обслуживании (Distributed Denial of Service, DDoS)* в A10 Networks. Если хотите узнать больше, загрузите бесплатное руководство: https://www.a10networks.com/wp-content/uploads/A10-TPS-EB-Distributed_Denial_of_Service_DDoS_Practical_Detection_and_Defense.pdf.

Резюме

Эта глава была посвящена сетевой безопасности с Python. Мы использовали инструмент Cisco VIRL для создания лаборатории с серверами и сетевыми устройствами типа NX-OSv и IOSv. Мы также сделали краткий обзор проекта Scapy, который позволяет формировать пакеты с нуля.

Scapy можно применять в интерактивном режиме для быстрого тестирования. Шаги, выполненные в интерактивном режиме, можно сохранить в файл, чтобы сделать тестирование более масштабируемым. Этот инструмент подходит для проверки сетей на проникновение с поиском известных уязвимостей.

Мы также обсудили защиту сети с помощью списков доступа на основе IP- и MAC-адресов. Это обычно первая линия защиты в сетях. С помощью Ansible мы обеспечим быстрое и согласованное развертывание списков доступа на разные устройства.

Syslog и другие файлы журналов содержат полезную информацию, которую мы регулярно должны проверять, чтобы своевременно обнаружить признаки проникновения. С помощью регулярных выражений в языке Python можно систематически искать известные журнальные записи, сигнализирующие о событиях безопасности, которые требуют нашего внимания. Для дополнительной защиты можно использовать виртуальные локальные сети и UFW.

В главе 7 мы посмотрим, как с помощью Python организовать мониторинг сети. Мониторинг позволяет ориентироваться в том, что происходит в сети, и понимать, в каком состоянии она находится.

7

Сетевой мониторинг с использованием Python: часть 1

Представьте, что вам позвонили в два часа ночи из центра управления сетью вашей компании. Вы подняли трубку и слышите следующее: «Здравствуйте, у нас возникла серьезная проблема, которая сказывается на работе наших промышленных сервисов. Мы подозреваем, что она может быть связана с сетью. Можете проверить?» Что бы вы ответили в такой неотложной ситуации? В большинстве случаев на ум приходит следующее: какие изменения вносились перед тем, как возникли проблемы? Вы, скорее всего, откроете систему мониторинга и проверите, какие метрики поменялись за последние пару часов. А еще лучше, если вы уже получили оповещение, что определенные показатели вышли за рамки нормального диапазона.

На страницах книги мы постоянно обсуждаем способы систематического внесения предсказуемых изменений в сетевую конфигурацию, чтобы обеспечить максимально беспроблемную работу сети. Однако сети далеко не статические — это, наверное, одна из самых изменчивых частей вашей инфраструктуры. Они по определению соединяют разные инфраструктурные компоненты, постоянно передавая трафик туда-сюда.

Сеть может перестать работать так, как мы того от нее ожидаем, по множеству причин: аппаратные сбои, ошибки в программном обеспечении, ошибки, допущенные людьми (несмотря на благие намерения), и многое другое. Вопрос не в том, пойдет ли что-то не так, а скорее — когда это случится и что именно при этом сломается. Мы всегда должны следить за работой нашей сети, и желательно получать уведомления в случае возникновения проблем.

В этой и следующей главах мы рассмотрим разные способы организации мониторинга сети. Многие инструменты, которые мы уже рассмотрели, можно объединять или напрямую использовать с помощью Python. Как и многие другие механизмы, о которых мы узнали, сетевой мониторинг состоит из двух частей.

Первое: нам нужно понять, какую информацию наше оборудование способно передавать. И второе: мы должны определить, какие полезные сведения мы можем из нее извлечь.

Начнем эту главу с обзора инструментов для эффективного мониторинга сети, таких как:

- подготовка лаборатории;
- протокол *SNMP* (*Simple Network Management Protocol* — простой протокол сетевого управления) и сопутствующие библиотеки для работы с ним в Python;
- библиотеки визуализации для Python:
- Matplotlib и примеры;
- Pygal и примеры;
- интеграция Python с MRTG и Cacti для визуализации сети.

Это далеко не полный список. На рынке сетевого мониторинга явно нет недостатка в коммерческих решениях. Но для простых задач, которые мы будем рассматривать, хорошо подойдут как коммерческие, так и открытые инструменты.

Подготовка лаборатории

Здесь наша лаборатория — как в главе 6, но с одним отличием: оба сетевых устройства будут работать под управлением IOSv (рис. 7.1).

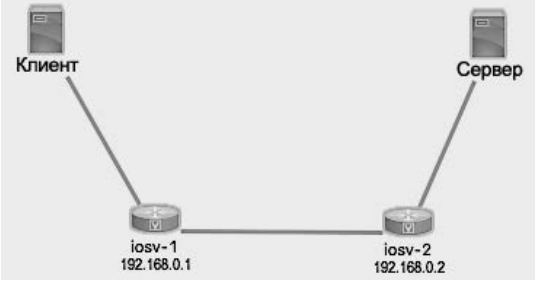


Рис. 7.1. Топология лаборатории

Два хоста с Ubuntu будут генерировать сетевой трафик, чтобы мы могли получить ненулевые метрики.

SNMP

SNMP — это стандартный протокол администрирования устройств и сбора сведений о них. По моему опыту, большинство сетевых администраторов используют SNMP только в качестве механизма получения информации. Поскольку SNMP работает поверх протокола UDP, который не поддерживает постоянных соединений, и учитывая относительно слабую защиту в версиях 1 и 2, внесение изменений в конфигурацию устройств с помощью SNMP вызывает у операторов сетей некоторое беспокойство. В версии 3 появилась поддержка шифрования, а также новые концепции и терминология, но ее внедрение зависит от поставщика оборудования.

Протокол SNMP был представлен в 1988 году, описан в RFC 1065 и широко применяется в сетевом мониторинге. Работать с ним просто: диспетчер шлет устройству GET- и SET-запросы, а устройство, если на нем установлен SNMP-агент, возвращает в ответ информацию. Самый распространенный стандарт — SNMPv2c, описанный в RFC 1901 — RFC 1908. Для защиты в нем применяется простой механизм безопасности, разработанный сообществом. В нем также появились новые возможности, такие как получение массивов информации. На рис. 7.2 показан общий принцип работы SNMP.

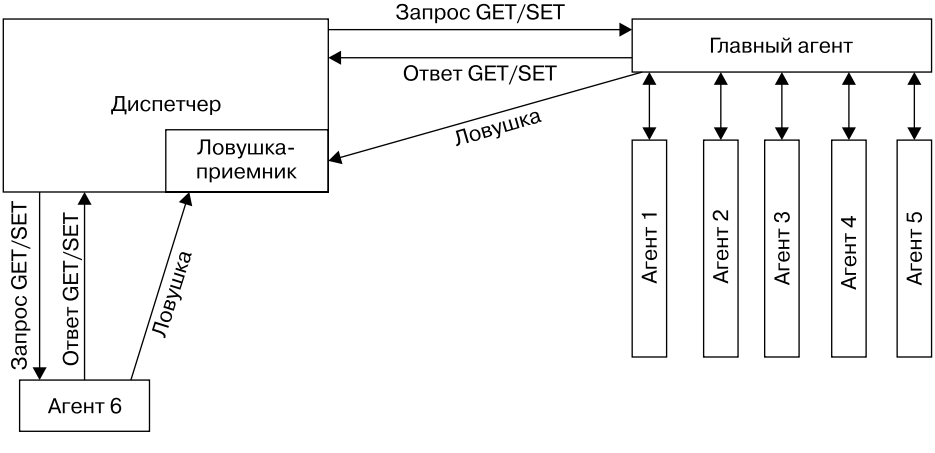


Рис. 7.2. Принцип работы SNMP

Информация, хранящаяся на устройстве, структурирована в *базе управляющей информации (Management Information Base, MIB)*. MIB использует иерархическое пространство имен с *идентификатором объекта (Object Identifier, OID)*, представляющее сведения, которые можно прочитать и вернуть запрашивающей стороне. Говоря об извлечении информации с помощью SNMP, мы на самом деле имеем в виду использование управляющего хоста для выполнения запроса по определенному OID, который возвращает нужные нам данные. Производители устройств используют общие OID для своих систем и интерфейсов. Но некоторые OID могут предоставляться специально для отдельных предприятий.

Чтобы можно было извлекать полезную информацию, администратор сети должен приложить некоторые усилия для систематизации данных в формате OID. Иногда этот процесс оказывается утомительным и заключается в поиске отдельных идентификаторов. Например, вы можете обратиться к OID устройства и получить в ответ значение 10 000. Что это такое? Трафик, проходящий через интерфейс? Это биты или байты? А может, количество пакетов? Откуда мы знаем? Чтобы ответить на эти вопросы, нам нужно свериться либо со стандартом, либо с документацией производителя. Этот процесс можно упростить с помощью таких инструментов, как MIB Browser, который предоставляет метаданные для конкретного значения. Но, по моему опыту, создание системы мониторинга на основе SNMP иногда выглядит как игра в кошки-мышки, когда вы пытаетесь найти то самое недостающее значение.

Несколько основных аспектов SNMP, о которых стоит помнить.

- Реализация во многом зависит от количества информации, которую способно предоставить устройство. А это, в свою очередь, зависит от того, как производитель относится к SNMP: как к неотъемлемой части системы или как к дополнительной возможности.
- Обычно для возвращения значения SNMP-агенты отнимают процессорное время у управляющего уровня. Из-за этого протокол SNMP неэффективен на устройствах, к примеру, с большими BGP-таблицами; что еще хуже, его нецелесообразно использовать для частого извлечения данных.
- Чтобы запросить данные, пользователь должен знать OID.

Поскольку протокол SNMP существует давно, я предполагаю, что вы уже с ним сталкивались. Приступим сразу к установке пакетов и нашему первому примеру с SNMP.

Подготовка

Для начала убедимся, что в нашей конфигурации есть управляющее устройство и агент с поддержкой SNMP. Пакет SNMP можно установить как на хостах (клиенте или сервере), так и на управляющем устройстве. Главное, чтобы диспетчер SNMP имел доступ к устройству и управляемое устройство принимало входящие соединения. В промышленных условиях установку следует делать только на управляющий хост, а SNMP-трафик должен быть разрешен только на управляющем уровне.

В этой лабораторной работе мы устанавливаем SNMP как на хост с Ubuntu в сети управления, так и на клиентский хост.

Организация внешнего доступа к хостам VIRL описывается в главе 6.

```
$ sudo apt-get update
$ sudo apt-get install snmp
```

Дальше нужно включить протокол SNMP и сконфигурировать его параметры на сетевых устройствах `iosv-1` и `iosv-2`. Нам доступно много дополнительных параметров: контакт, местоположение, ID шасси и размер пакетов SNMP. Их список зависит от устройства, поэтому вам следует свериться с документацией своего поставщика. На устройствах IOSv мы настроим список доступа, чтобы принимать запросы только от нужных нам хостов, и привяжем этот список к неизменяемой строке сообщества SNMP. В примере в качестве этой строки будет использоваться слово `secret`, а список доступа будет называться `permit_snmp`:

```
!
ip access-list standard permit_snmp
 permit 172.16.1.123 log
 deny   any log
!
snmp-server community secret RO permit_snmp
!
```

Строка сообщества SNMP играет роль пароля, разделяемого диспетчером и агентом; следовательно, ее необходимо включать в любой запрос, посылаемый устройству.

Как уже упоминалось в этой главе, поиск подходящего OID зачастую составляет значительную часть работы с SNMP. Для этого можно использовать Cisco IOS MIB Locator (http://tools.cisco.com/ITDIT/MIBS/servlet/index).

В качестве альтернативы мы можем пройтись по дереву SNMP для устройств Cisco, начиная с вершины: `.1.3.6.1.4.1.9`. Мы проведем обход, чтобы убедиться в корректной работе SNMP-агента и списка доступа:

```
$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.4.1.9
iso.3.6.1.4.1.9.2.1.1.0 = STRING: "
Bootstrap program is IOSv
"
iso.3.6.1.4.1.9.2.1.2.0 = STRING: "reload"
iso.3.6.1.4.1.9.2.1.3.0 = STRING: "iosv-1"
iso.3.6.1.4.1.9.2.1.4.0 = STRING: "virl.info"
<опущено>
```

Сделаем наш запрос по OID более конкретным:

```
$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.4.1.9.2.1.61.0
iso.3.6.1.4.1.9.2.1.61.0 = STRING: "cisco Systems, Inc.
170 West Tasman Dr.
San Jose, CA  95134-1706
U.S.A.
Ph +1-408-526-4000
Customer service 1-800-553-6387 or +1-408-526-7208
24HR Emergency 1-800-553-2447 or +1-408-526-7209
Email Address tac@cisco.com
World Wide Web www.cisco.com"
```

Что произойдет, если поменять в конце одну цифру, подставив `1` вместо `0`? Вот что мы увидим:

```
$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.4.1.9.2.1.61.1
iso.3.6.1.4.1.9.2.1.61.1 = No Such Instance currently exists at this OID
```

В отличие от API-вызовов этот вывод не предоставляет никакого сообщения или кода ошибки; он просто утверждает, что данный OID не существует. Иногда это может быть довольно неприятно.

Напоследок проверим список доступа, который должен отклонять нежелательные SNMP-запросы. Поскольку ключевое слово `log` в списке доступа указано как для разрешительных, так и для запретительных записей, запросы к устройствам сможет выполнять только хост `172.16.1.123`:

```
*Sep 29 16:39:19.857: %SEC-6-IPACCESSLOGNP: list permit_snmp permitted 0
172.16.1.123 -> 0.0.0.0, 1 packet
```

Как видите, основная трудность конфигурации SNMP состоит в поиске подходящего OID. Некоторые OID определены в стандартизованных базах MIB-2, а другие входят в ту часть дерева, которая относится к конкретному предприятию. Эту информацию лучше всего искать в документации поставщика. В этом

вам поможет ряд инструментов; например, вы можете ввести MIB (опять же предоставленные поставщиком) в MIB Browser и получить описание корпоративных OID. Существенную помощь в поиске идентификаторов объектов может оказать SNMP Object Navigator от Cisco (http://snmp.cloudapps.cisco.com/Support/SNMP/do/BrowseOID.do?local=en).

PySNMP

PySNMP — это кросс-платформенная реализация SNMP на чистом языке Python, разработанная Ильей Етингофом (https://github.com/etingof). Она инкапсулирует множество деталей протокола SNMP и поддерживает как Python 2, так и Python 3.

Для работы с PySNMP требуется пакет PyASN1. Вот что об этом написано в Википедии:

> *«ASN.1 — это стандарт и нотация описания правил и структур для представления, кодирования, передачи и декодирования данных в телекоммуникациях и компьютерных сетях».*

PyASN1 — это удобная обертка вокруг ASN.1 для Python. Сначала установим этот пакет. Обратите внимание, что все действия выполняются в виртуальном окружении, поэтому здесь используется интерпретатор Python из виртуального окружения:

```
(venv) $ cd /tmp
(venv) $ git clone github.com/etingof/pyasn1.git
(venv) $ git checkout 0.2.3
(venv) $ python setup.py install # обратите внимание на путь venv
```

Установим пакет PySNMP:

```
(venv) $ cd /tmp
(venv) $ git clone github.com/etingof/pysnmp
(venv) $ cd pysnmp/
(venv) $ git checkout v4.3.10
(venv) $ python setup.py install # обратите внимание на путь venv
```

Мы используем более старую версию PySNMP, так как в версии 5.0.0 был удален модуль pysnmp.entity.rfc3413.oneliner (https://github.com/etingof/pysnmp/blob/a93241007b970c458a0233c16ae2ef82dc107290/CHANGES.txt). Если вы устанавливаете пакеты с помощью pip, то эти примеры, скорее всего, не будут работать.

Посмотрим, как с помощью PySNMP получить ту же контактную информацию компании Cisco, которую мы использовали в предыдущем примере. Это слегка отредактированный пример со страницы https://pysnmp.readthedocs.io/en/latest/faq/response-values-mib-resolution.html. Сначала импортируем нужный нам модуль и создадим объект `CommandGenerator`:

```
>>> from pysnmp.entity.rfc3413.oneliner import cmdgen
>>> cmdGen = cmdgen.CommandGenerator()
>>> cisco_contact_info_oid = "1.3.6.1.4.1.9.2.1.61.0"
```

Мы можем выполнить SNMP-запрос с помощью метода `getCmd`. Ответ будет распакован в ряд переменных, из которых нас больше всего интересует `varBinds` с результатом запроса:

```
>>> errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
        cmdgen.CommunityData('secret'),
        cmdgen.UdpTransportTarget(('172.16.1.189', 161)),
        cisco_contact_info_oid)
>>> for name, val in varBinds:
        print('%s=%s' % (name.prettyPrint(), str(val)))

SNMPv2-SMI::enterprises.9.2.1.61.0=cisco Systems, Inc.
170 West Tasman Dr.
San Jose, CA  95134-1706
U.S.A.
Ph +1-408-526-4000
Customer service 1-800-553-6387 or +1-408-526-7208
24HR Emergency 1-800-553-2447 or +1-408-526-7209
Email Address tac@cisco.com
World Wide Web www.cisco.com
>>>
```

Обратите внимание: значения в ответе являются объектами PyASN1. Метод `prettyPrint()` может привести некоторые из них в удобочитаемый вид, но `varBinds` придется преобразовывать вручную. Мы превратили ее в строку.

Теперь на основе предыдущего интерактивного примера можно написать сценарий с проверкой ошибок. Назовем его `pysnmp_1.py`. И мы можем передать методу `getCmd()` сразу несколько OID:

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

from pysnmp.entity.rfc3413.oneliner import cmdgen

cmdGen = cmdgen.CommandGenerator()

system_up_time_oid = "1.3.6.1.2.1.1.3.0"
cisco_contact_info_oid = "1.3.6.1.4.1.9.2.1.61.0"

errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
```

```
    cmdgen.CommunityData('secret'),
    cmdgen.UdpTransportTarget(('172.16.1.189', 161)),
    system_up_time_oid,
    cisco_contact_info_oid
)

# Проверяем ошибки и выводим результаты
if errorIndication:
    print(errorIndication)
else:
    if errorStatus:
        print('%s at %s' % (
            errorStatus.prettyPrint(),
            errorIndex and varBinds[int(errorIndex)-1] or '?'
            )
        )
    else:
        for name, val in varBinds:
            print('%s = %s' % (name.prettyPrint(), str(val)))
```

Результаты будут распакованы, и мы получим значения двух OID:

```
$ python pysnmp_1.py
SNMPv2-MIB::sysUpTime.0 = 599083
SNMPv2-SMI::enterprises.9.2.1.61.0 = cisco Systems, Inc.
170 West Tasman Dr.
San Jose, CA  95134-1706
U.S.A.
Ph +1-408-526-4000
Customer service 1-800-553-6387 or +1-408-526-7208
24HR Emergency 1-800-553-2447 or +1-408-526-7209
Email Address tac@cisco.com
World Wide Web www.cisco.com
```

В следующем примере мы сохраним значения, полученные в ответ на запрос, для выполнения других действий с данными, таких как визуализация. Для вывода значений, относящихся к интерфейсам, воспользуемся `ifEntry` из MIB-2.

Вы можете найти ряд ресурсов с описанием дерева `ifEntry`; вот снимок страницы сайта Cisco SNMP Object Navigator, который мы использовали для `ifEntry` (рис. 7.3).

Выполним небольшую проверку, чтобы показать, какие OID имеют интерфейсы устройства:

```
$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.2.1.2.2.1.2
iso.3.6.1.2.1.2.2.1.2.1 = STRING: "GigabitEthernet0/0"
iso.3.6.1.2.1.2.2.1.2.2 = STRING: "GigabitEthernet0/1"
iso.3.6.1.2.1.2.2.1.2.3 = STRING: "GigabitEthernet0/2"
iso.3.6.1.2.1.2.2.1.2.4 = STRING: "Null0"
iso.3.6.1.2.1.2.2.1.2.5 = STRING: "Loopback0"
```

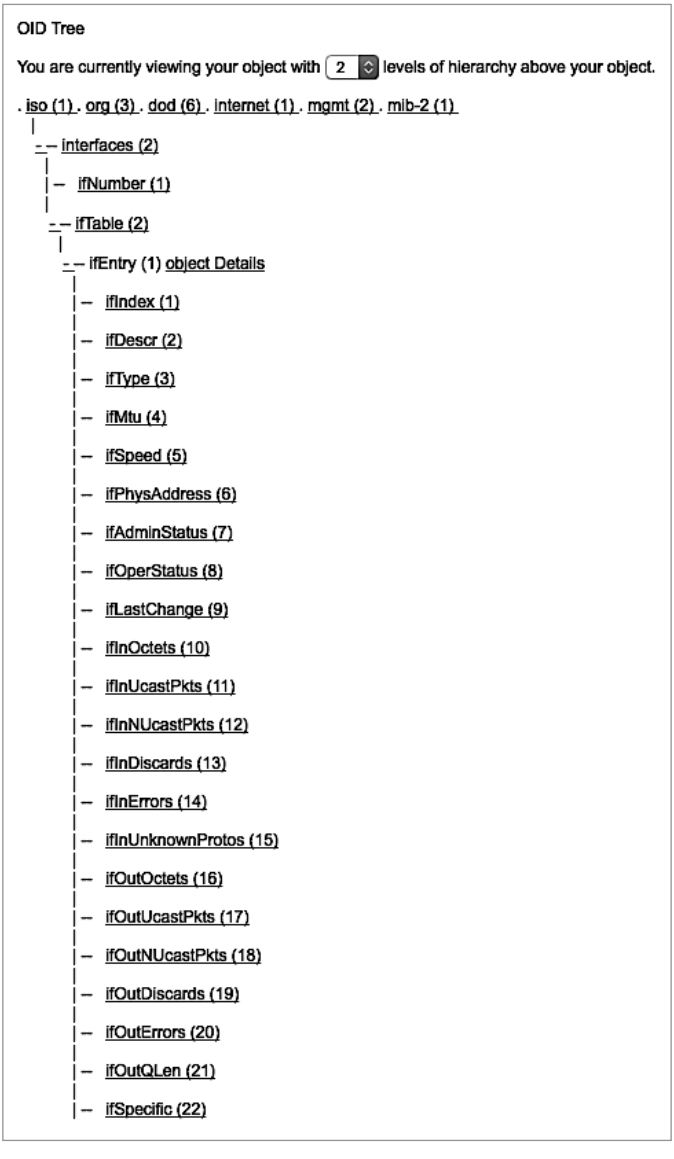


Рис. 7.3. Дерево ifEntry с перечнем OID

Согласно документации, значения `ifInOctets(10)`, `ifInUcastPkts(11)`, `ifOutOctets(16)` и `ifOutUcastPkts(17)` можно отобразить в соответствующие значения OID. Например, в документации для CLI и MIB сказано, что пакеты, выходящие из `GigabitEthernet0/0`, имеют OID `1.3.6.1.2.1.2.2.1.17.1`. Тот же

процесс можно повторить и для остальных OID. Когда вы проверяете пакеты между CLI и SNMP, помните, что значения должны быть близкими, но не обязательно одинаковыми, так как между получением вывода в терминале и отправкой SNMP-запроса по каналу связи может пройти какой-то трафик:

```
iosv-1#sh int gig 0/0 | i packets
  5 minute input rate 0 bits/sec, 0 packets/sec
  5 minute output rate 0 bits/sec, 0 packets/sec
     6872 packets input, 638813 bytes, 0 no buffer
     4279 packets output, 393631 bytes, 0 underruns

$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.2.1.2.2.1.17.1
iso.3.6.1.2.1.2.2.1.17.1 = Counter32: 4292
```

В промышленных условиях результаты, скорее всего, записываются в базу данных. Но поскольку это всего лишь пример, мы сохраним полученные значения в плоский файл. Напишем сценарий `pysnmp_3.py` для информационных запросов и определим в нем OID, которые нам нужно запрашивать:

```
# OID имени хоста
system_name = '1.3.6.1.2.1.1.5.0'

# OID интерфейса
gig0_0_in_oct = '1.3.6.1.2.1.2.2.1.10.1'
gig0_0_in_uPackets = '1.3.6.1.2.1.2.2.1.11.1'
gig0_0_out_oct = '1.3.6.1.2.1.2.2.1.16.1'
gig0_0_out_uPackets = '1.3.6.1.2.1.2.2.1.17.1'
```

Эти значения будут использоваться в функции `snmp_query()`, которая принимает на вход `host`, `community` и `oid`:

```
def snmp_query(host, community, oid):
    errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
        cmdgen.CommunityData(community),
        cmdgen.UdpTransportTarget((host, 161)),
        oid
    )
```

Все значения помещаются в словарь с разными ключами и записываются в файл с именем `results.txt`:

```
result = {}
result['Time'] = datetime.datetime.utcnow().isoformat()
result['hostname'] = snmp_query(host, community, system_name)
result['Gig0-0_In_Octet'] = snmp_query(host, community, gig0_0_in_oct)
result['Gig0-0_In_uPackets'] = snmp_query(host, community, gig0_0_in_
uPackets)
result['Gig0-0_Out_Octet'] = snmp_query(host, community, gig0_0_out_oct)
result['Gig0-0_Out_uPackets'] = snmp_query(host, community, gig0_0_
out_uPackets)
```

```
with open('/home/echou/Master_Python_Networking/Chapter7/results.txt',
'a') as f:
    f.write(str(result))
    f.write('\n')
```

В результате мы получим файл с пакетами, представленными в интерфейсе в момент запроса:

```
$ cat results.txt
{'Gig0-0_In_Octet': '3990616', 'Gig0-0_Out_uPackets': '60077', 'Gig0-
0_In_uPackets': '42229', 'Gig0-0_Out_Octet': '5228254', 'Time': '2017-03-
06T02:34:02.146245', 'hostname': 'iosv-1.virl.info'}
{'Gig0-0_Out_uPackets': '60095', 'hostname': 'iosv-1.virl.info', 'Gig0-0_
Out_Octet': '5229721', 'Time': '2017-03-06T02:35:02.072340', 'Gig0-0_In_
Octet': '3991754', 'Gig0-0_In_uPackets': '42242'}
<опущено>
```

Мы можем сделать этот сценарий выполняемым и запланировать его запуск раз в пять минут с помощью `cron`:

```
$ chmod +x pysnmp_3.py
# crontab configuration
*/5 * * * * /home/echou/Mastering_Python_Networking_third_edition/
Chapter07/pysnmp_3.py
```

Повторяю, что в промышленном окружении мы бы поместили информацию в базу данных. В реляционной БД уникальные идентификаторы можно использовать в качестве первичного ключа. В NoSQL первичным индексом (или ключом) может выступать время (так как оно никогда не повторяется), за которым следуют разные пары вида «ключ — значение».

Подождем, пока этот сценарий выполнится несколько раз, чтобы он записал нужные нам значения. Если вам не терпится, сократите интервал выполнения задания `cron` до 1 минуты. Когда в файле `results.txt` накопится достаточно данных для построения графика, можно переходить к следующему разделу, в котором вы увидите, как визуализировать данные с помощью Python.

Python для визуализации данных

Мы собираем сетевые данные для того, чтобы получить представление о работе сети. Визуализация — один из лучших способов узнать, что означают те или иные данные. Это относится почти к любым данным, но к последовательным особенно, если речь идет о сетевом мониторинге. Сколько трафика прошло по сети за предыдущую неделю? Какая его доля приходится на протокол TCP?

Механизмы сбора данных вроде SNMP помогут нам получить такую информацию, а библиотеки для Python — построить графики.

В этом разделе мы с помощью популярных библиотек Matplotlib и Pygal визуализируем данные, собранные ранее с помощью SNMP.

Matplotlib

Matplotlib (http://matplotlib.org/) — это библиотека для Python, которая позволяет строить двумерные графики с помощью пакета математических расширений NumPy. С ее помощью можно генерировать изображения типографского качества, включая графики, гистограммы и столбчатые диаграммы, всего несколькими строками кода.

NumPy — это расширение языка программирования Python, инструмент с открытым исходным кодом, который широко применяется в различных проектах для анализа данных. Подробнее о нем читайте на странице https://ru.wikipedia.org/wiki/NumPy.

Начнем с установки.

Установка

Для установки используйте диспетчер пакетов дистрибутива Linux или `pip`:

```
(venv) $ pip install matplotlib
```

Перейдем к первому примеру.

Matplotlib: первый пример

В этом примере итоговое изображение будет по умолчанию подаваться на стандартное устройство вывода, обычно это экран монитора. При разработке бывает проще сначала поэкспериментировать, возвращая результат в стандартный вывод, а затем оформить код в виде сценария. Если вы выполняли упражнения из этой книги на виртуальной машине, для вывода графиков рекомендуется использовать окно ВМ, а не SSH (иначе вы ничего не увидите). Если у вас нет доступа к стандартному выводу, можете сохранить изображение и загрузить его для просмотра (вскоре вы узнаете, как это делается). В некоторых примерах из этого раздела нужно установить переменную `$DISPLAY`.

Ниже показан снимок экрана системы Ubuntu Desktop, которая используется здесь в примерах визуализации. Сразу после ввода в терминале команды `plt.show()` на экране появится окно Figure 1. Закрыв это окно, вы сможете вернуться в командную оболочку (рис. 7.4).

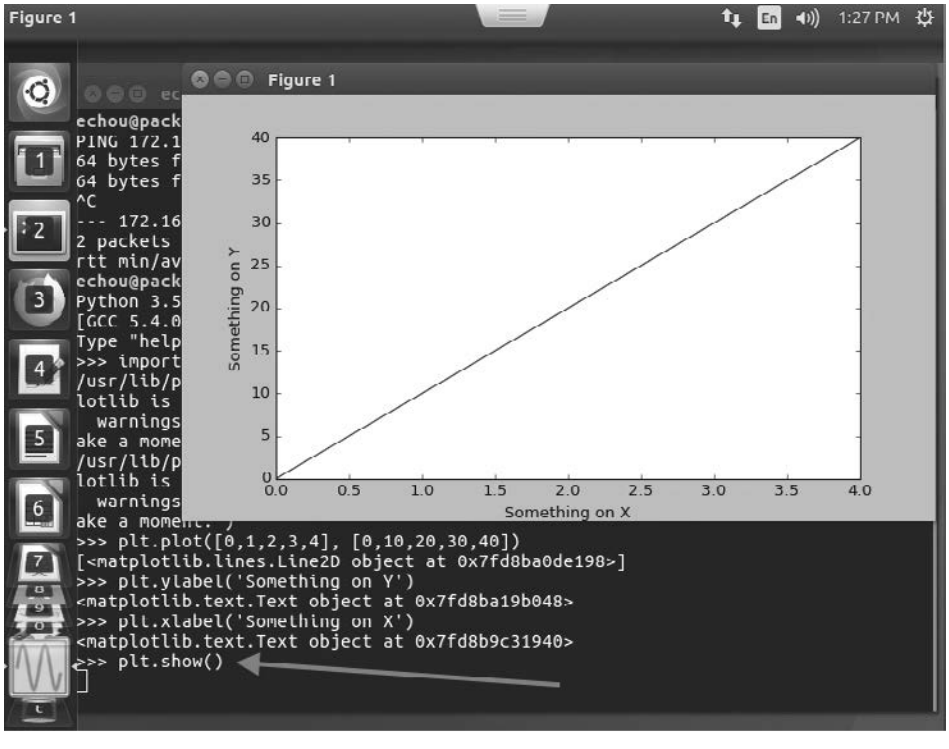


Рис. 7.4. Визуализация с помощью Matplotlib в Ubuntu Desktop

Сначала рассмотрим график прямой. Он получается из двух списков числовых значений по осям *X* и *Y*:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([0,1,2,3,4], [0,10,20,30,40])
[<matplotlib.lines.Line2D object at 0x7f932510df98>]
>>> plt.ylabel('Something on Y')
<matplotlib.text.Text object at 0x7f93251546a0>
>>> plt.xlabel('Something on X')
<matplotlib.text.Text object at 0x7f9325fdb9e8>
>>> plt.show()
```

График будет иметь вид прямой линии (рис. 7.5).

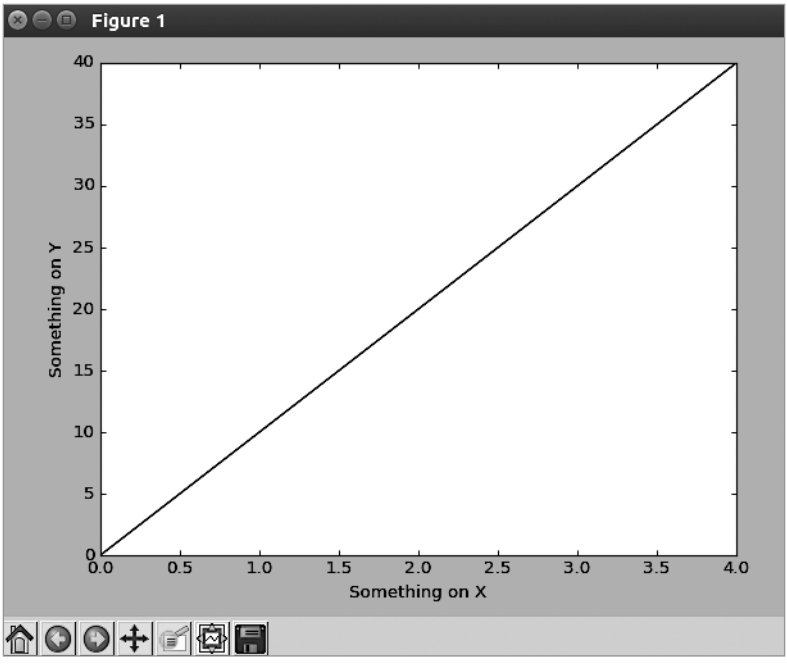


Рис. 7.5. График прямой в Matplotlib

Если у вас нет доступа к с стандартному выводу или вы не сохранили изображение с самого начала, воспользуйтесь методом `savefig()`:

```
>>> plt.savefig('figure1.png') or
>>> plt.savefig('figure1.pdf')
```

Вооружившись этими базовыми знаниями о построении графиков, мы сможем визуализировать результаты SNMP-запросов.

Matplotlib: вывод результатов SNMP

В нашем первом сценарии `matplotlib_1.py`, использующем Matplotlib, в дополнение к `pyplot` импортируем модуль `matplotlib.dates`; мы используем его вместо модуля `dates` из стандартной библиотеки Python.

В отличие от `dates` модуль `matplotlib.dates` автоматически преобразует значение даты в вещественный тип, как того требует библиотека Matplotlib:

```
import matplotlib.pyplot as plt
import matplotlib.dates as dates
```

Matplotlib имеет богатые возможности по выводу диаграмм с датами; подробнее об этом читайте по адресу http://matplotlib.org/api/dates_api.html.

В этом сценарии мы создадим два пустых списка для значений по осям *X* и *Y*. Обратите внимание: в строке 12 мы используем встроенную в Python функцию `eval()`, чтобы прочитать входные данные в виде словаря, а не обычной строки:

```
x_time = []
y_value = []

with open('results.txt', 'r') as f:
    for line in f.readlines():
        # eval(line) интерпретирует каждую строку как словарь, а не как
        # строку текста
        line = eval(line)
        # преобразуем во внутренний вещественный тип
        x_time.append(dates.datestr2num(line['Time']))
        y_value.append(line['Gig0-0_Out_uPackets'])
```

Чтобы перевести значение по оси *X* обратно в формат даты, понятный человеку, применим функцию `plot_date()` вместо `plot()`. Также слегка изменим размер изображения и сместим ось *X*, чтобы видеть все значения:

```
plt.subplots_adjust(bottom=0.3)
plt.xticks(rotation=80)
plt.plot_date(x_time, y_value)
plt.title('Router1 G0/0')
plt.xlabel('Time in UTC')
plt.ylabel('Output Unicast Packets')
plt.savefig('matplotlib_1_result.png')
plt.show()
```

Итоговый результат содержит заголовки Router1 G0/0 и Output Unicast Packets (рис. 7.6).

Если вам больше нравятся прямые линии, чем точки, добавьте третий параметр в функцию `plot_date()`:

```
plt.plot_date(x_time, y_value, "-")
```

Можно повторить эти шаги для остальных значений с выходными октетами, входными одноадресными пакетами и вводом в виде отдельных диаграмм. Но в следующем примере, `matplotlib_2.py`, мы посмотрим, как вывести несколько значений в одном временном диапазоне, и познакомимся с дополнительными параметрами Matplotlib.

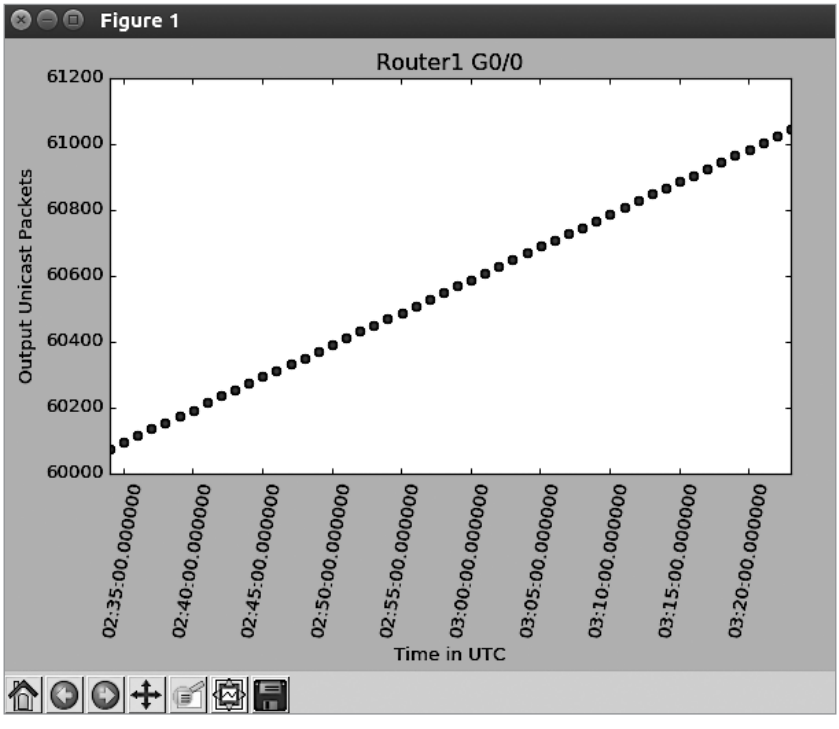


Рис. 7.6. График для Router1 в Matplotlib

Сейчас мы создадим дополнительные списки и заполним их значениями:

```
x_time = []
out_octets = []
out_packets = []
in_octets = []
in_packets = []
with open('results.txt', 'r') as f:
    for line in f.readlines():
        # eval(line) интерпретирует каждую строку как словарь,
а не как строку текста
        line = eval(line)
        # преобразуем во внутренний вещественный тип
        x_time.append(dates.datestr2num(line['Time']))
        out_packets.append(line['Gig0-0_Out_uPackets'])
        out_octets.append(line['Gig0-0_Out_Octet'])
        in_packets.append(line['Gig0-0_In_uPackets'])
        in_octets.append(line['Gig0-0_In_Octet'])
```

Поскольку значения по оси *X* совпадают, можно просто добавить на ту же диаграмму другие значения по оси *Y*:

```
# Используем plot_date, чтобы снова вывести ось X в формате даты
plt.plot_date(x_time, out_packets, '-', label='Out Packets')
plt.plot_date(x_time, out_octets, '-', label='Out Octets')
plt.plot_date(x_time, in_packets, '-', label='In Packets')
plt.plot_date(x_time, in_octets, '-', label='In Octets')
```

Добавим на диаграмму сетку и легенду:

```
plt.title('Router1 G0/0')
plt.legend(loc='upper left')
plt.grid(True)
plt.xlabel('Time in UTC')
plt.ylabel('Values')
plt.savefig('matplotlib_2_result.png')
plt.show()
```

В итоге получится единая диаграмма со всеми значениями. Обратите внимание: часть значений в левом верхнем углу оказались закрыты легендой. Чтобы этого избежать, можно изменить размер изображения и/или сместить/масштабировать графики (рис. 7.7).

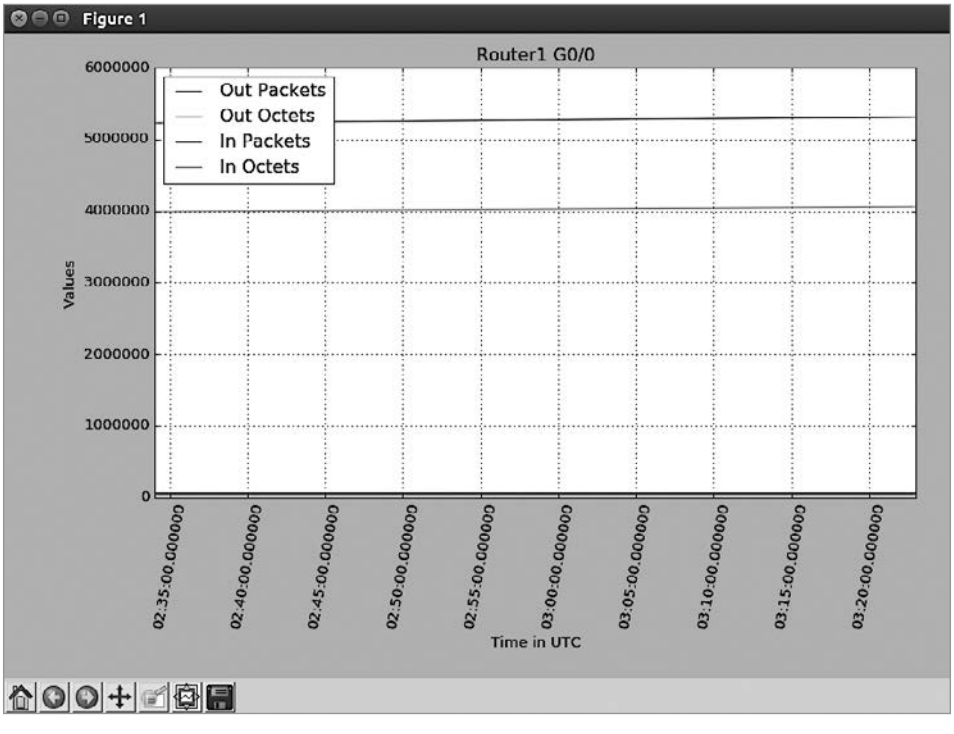


Рис. 7.7. Несколько графиков для Router1 в Matplotlib

Matplotlib поддерживает много других видов диаграмм; мы не ограничены обычными графиками. Например, в `matplotlib_3.py` те же фиктивные данные можно использовать для вывода соотношения разных типов трафика, который проходит через соединение:

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

# Пример взят из
# http://matplotlib.org/2.0.0/examples/pie_and_polar_charts/pie_demo_
features.html

import matplotlib.pyplot as plt

# Круговая диаграмма, секторы которой упорядочены и выводятся против часовой
стрелки:
labels = 'TCP', 'UDP', 'ICMP', 'Others'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # Выделяем UDP

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Благодаря пропорциональному отношению сторон диаграмма
                   # имеет форму круга.

plt.savefig('matplotlib_3_result.png')
plt.show()
```

После выполнения `plt.show()` этот код выведет круговую диаграмму, показанную на рис. 7.8.

В этом разделе мы с помощью Matplotlib создали на основе сетевых данных привлекательные диаграммы, которые отражают состояние сети. Мы можем строить столбчатые и круговые диаграммы, а также линейные графики — все, что подходит для отображения имеющихся данных. Matplotlib — это мощный открытый инструмент, который не ограничивается языком Python. Существует множество дополнительных ресурсов, которые помогут вам ближе познакомиться с Matplotlib.

Дополнительные источники информации о Matplotlib

Matplotlib — одна из лучших библиотек для построения диаграмм в сценариях на Python, она позволяет создавать изображения типографского качества. Как и Python, Matplotlib позволяет упрощать задачи. У Matplotlib более 10 000 звезд на GitHub (и их число растет); это один из самых популярных проектов с открытым исходным кодом.

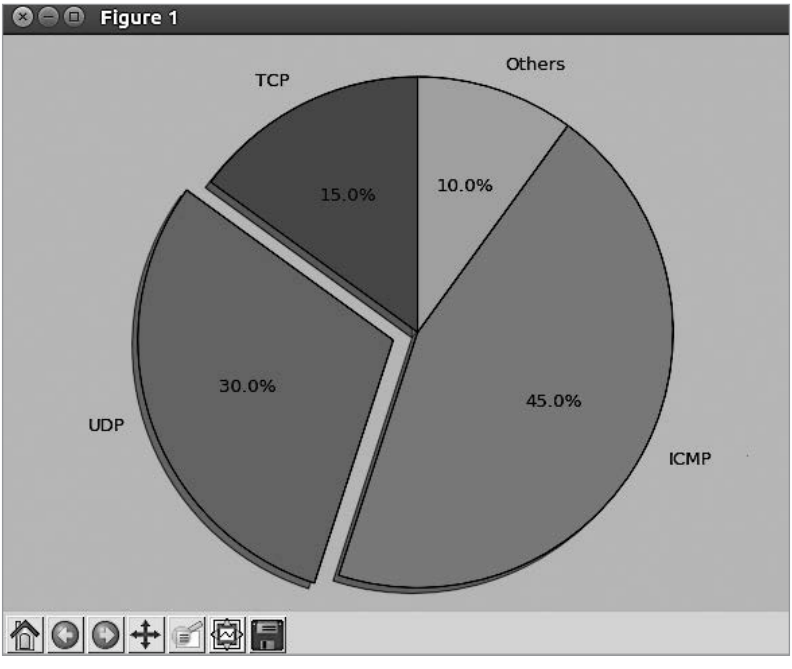


Рис. 7.8. Круговая диаграмма в Matplotlib

Прямое следствие популярности — оперативное исправление ошибок, дружественное сообщество, обширная документация и удобство использования. Изучение этого пакета требует определенных усилий, но оно того стоит.

В этом разделе мы едва затронули богатые возможности Matplotlib. Дополнительную информацию ищите на страницах http://matplotlib.org/2.0.0/index.html (страница проекта Matplotlib) и https://github.com/matplotlib/matplotlib (репозиторий Matplotlib на GitHub).

В следующем разделе речь пойдет еще об одной популярной библиотеке для создания диаграмм в Python — *Pygal*.

Pygal

Pygal (http://www.pygal.org/) — это библиотека для динамического создания SVG-диаграмм, написанная на Python. Ее основное преимущество, по моему мнению, состоит в том, что она позволяет легко генерировать изображения в формате *SVG (Scalable Vector Graphics — масштабируемая векторная графика)* с помощью

стандартных средств. У SVG множество преимуществ по сравнению с другими графическими форматами, но среди них можно выделить два главных: он хорошо поддерживается браузерами и позволяет масштабировать изображения без ухудшения качества. То есть вы можете просматривать полученные изображения в любом современном браузере и увеличивать их, не жертвуя детализацией. Я уже говорил, что все это делается с помощью нескольких строк кода на Python? Здорово, правда?

Установим Pygal и приступим к первому примеру.

Установка

Установка выполняется с помощью `pip`:

```
(venv)$ pip install pygal
```

Pygal: первый пример

Рассмотрим пример создания линейного графика, который приводится в документации Pygal (https://www.pygal.org/en/stable/documentation/types/line.html):

```
>>> import pygal
>>> line_chart = pygal.Line()
>>> line_chart.title = 'Browser usage evolution (in %)'
>>> line_chart.x_labels = map(str, range(2002, 2013))
>>> line_chart.add('Firefox', [None, None,    0, 16.6,   25,   31, 36.4,
45.5, 46.3, 42.8, 37.1])
<pygal.graph.line.Line object at 0x7f4883c52b38>
>>> line_chart.add('Chrome',  [None, None, None, None, None, None,    0,
3.9, 10.8, 23.8, 35.3])
<pygal.graph.line.Line object at 0x7f4883c52b38>
>>> line_chart.add('IE',      [85.8, 84.6, 84.7, 74.5,   66, 58.6, 54.7,
44.8, 36.2, 26.6, 20.1])
<pygal.graph.line.Line object at 0x7f4883c52b38>
>>> line_chart.add('Others',  [14.2, 15.4, 15.3,  8.9,    9, 10.4,   8.9,
5.8,  6.7,  6.8,  7.5])
<pygal.graph.line.Line object at 0x7f4883c52b38>
>>> line_chart.render_to_file('pygal_example_1.svg')
```

В этом примере мы создали объект линейного графика Line с 11 метками вдоль оси X (x_labels). В диаграмму можно добавлять любые объекты с метками и списками значений, например Firefox, Chrome или IE.

Интересно, что каждой точке графика соответствует отдельная метка на оси *X*. Если для какого-то года не существует значения (как для Chrome в период 2002–2007), то указывается значение `None`.

На рис. 7.9 показана итоговая диаграмма, открытая в браузере Firefox.

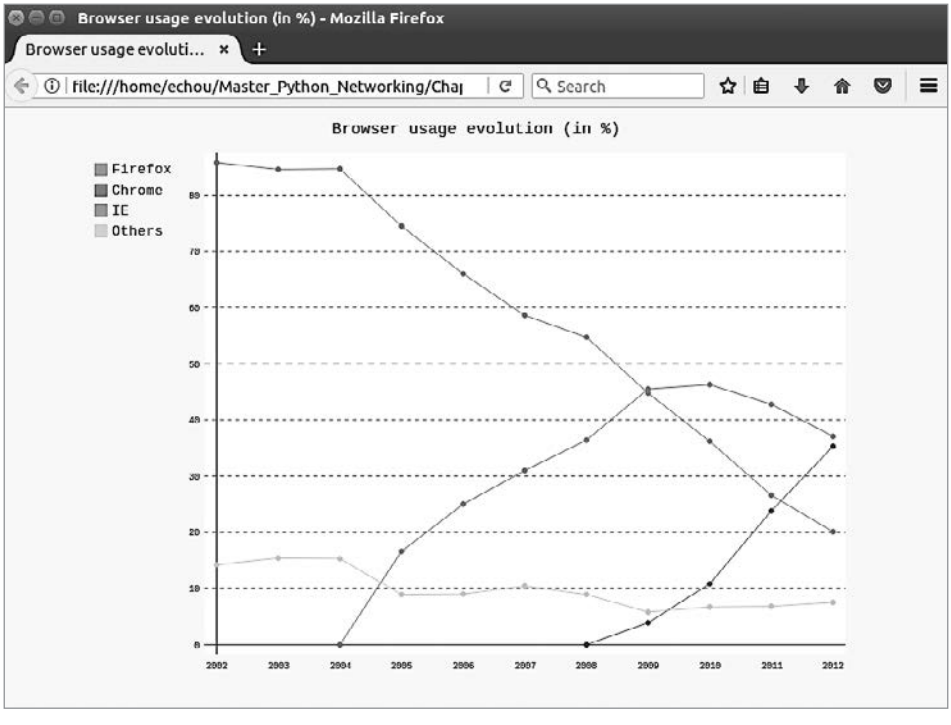


Рис. 7.9. Демонстрационная диаграмма Pygal

Познакомившись с общими принципами использования Pygal, применим их для вывода результатов SNMP. Этому посвящена следующая глава.

Вывод результатов SNMP с помощью Pygal

Для вывода графика с помощью Pygal используйте примерно тот же подход, что и в примере с Matplotlib, где мы создавали списки значений путем чтения файла. Нам больше не нужно переводить значения по оси *X* во внутренний вещественный формат, однако каждое числовое значение, которое мы получим, необходимо преобразовать в тип `float`:

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

import pygal

x_time = []
```

```
out_octets = []
out_packets = []
in_octets = []
in_packets = []

with open('results.txt', 'r') as f:
    for line in f.readlines():
        # eval(line) интерпретирует каждую строку как словарь,
        # а не как строку текста
        line = eval(line)
        x_time.append(line['Time'])
        out_packets.append(float(line['Gig0-0_Out_uPackets']))
        out_octets.append(float(line['Gig0-0_Out_Octet']))
        in_packets.append(float(line['Gig0-0_In_uPackets']))
        in_octets.append(float(line['Gig0-0_In_Octet']))
```

Для построения графика подойдет уже знакомый нам механизм:

```
line_chart = pygal.Line()
line_chart.title = "Router 1 Gig0/0"
line_chart.x_labels = x_time
line_chart.add('out_octets', out_octets)
line_chart.add('out_packets', out_packets)
line_chart.add('in_octets', in_octets)
line_chart.add('in_packets', in_packets)
line_chart.render_to_file('pygal_example_2.svg')
```

Результат похож на тот, который мы уже видели, однако график теперь в формате SVG, который легко отображается на веб-странице. Его можно просмотреть в современном браузере (рис. 7.10).

Pygal, как и Matplotlib, поддерживает многие виды диаграмм. Например, чтобы получить круговую диаграмму, которую мы видели в примере с Matplotlib, можно воспользоваться объектом `pygal.Pie()`, как показано в сценарии `pygal_2.py`:

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

import pygal

line_chart = pygal.Pie()
line_chart.title = "Protocol Breakdown"
line_chart.add('TCP', 15)
line_chart.add('UDP', 30)
line_chart.add('ICMP', 45)
line_chart.add('Others', 10)
line_chart.render_to_file('pygal_example_3.svg')
```

На рис. 7.11 — итоговый SVG-файл.

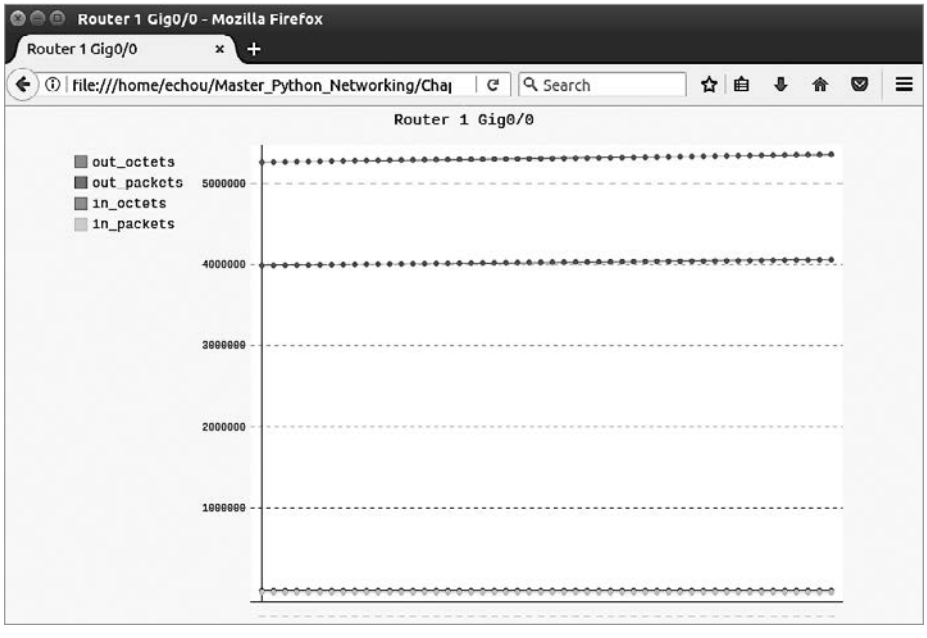


Рис. 7.10. Диаграмма с несколькими графиками для Router1 в Pygal

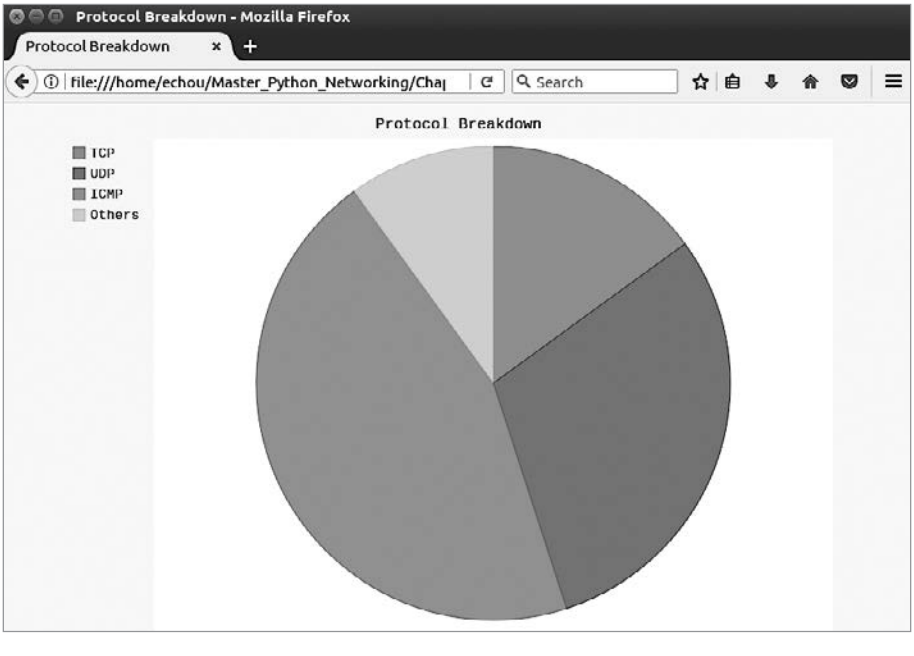


Рис. 7.11. Круговая диаграмма в Pygal

Pygal отлично подходит для генерации SVG-диаграмм типографского качества. Если вам нужен именно этот формат, не тратьте время на поиск альтернатив. В этом разделе мы рассмотрели примеры с Pygal для создания графиков на основе сетевых данных. Как и в случае с Matplotlib, для более глубокого изучения Pygal используйте дополнительные ресурсы.

Дополнительные источники информации о Pygal

Pygal предоставляет множество конфигурируемых возможностей и функций для построения диаграмм и визуализации данных, собираемых такими простыми средствами сетевого мониторинга, как SNMP. Мы показали здесь простой линейный график и круговую диаграмму. Больше информации об этом проекте ищите на сайтах:

- *документация Pygal:* http://www.pygal.org/en/stable/index.html;
- *страница проекта на GitHub:* https://github.com/Kozea/pygal.

В следующем разделе мы продолжим обсуждение SNMP, но уже в контексте полноценной системы сетевого мониторинга под названием *Cacti*.

Работа с Cacti в Python

Когда я только начинал свою карьеру, работая младшим сетевым инженером в компании регионального интернет-провайдера, мы использовали открытый кросс-платформенный инструмент *MRTG (Multi Router Traffic Grapher*; https://ru.wikipedia.org/wiki/MRTG) для проверки загруженности сетевых соединений. Это было наше основное и чуть ли не единственное средство сетевого мониторинга. Меня по-настоящему впечатлило, насколько качественным и полезным может оказаться проект с открытым исходным кодом. Это была одна из первых открытых высокоуровневых систем для мониторинга сетей, которая скрывала от сетевых инженеров подробности работы SNMP, базы данных и HTML. Затем появился пакет *RRDtool (Round-Robin Database tool*; https://ru.wikipedia.org/wiki/RRDtool). В первой версии, вышедшей в 1999 году, авторы RRDtool охарактеризовали его как MRTG Done Right («MRTG, реализованный как следует»). От MRTG его отличала существенно улучшенная производительность базы данных и опрашивающего механизма в серверной части.

В 2001 году вышел открытый веб-инструмент для сетевого мониторинга и визуализации под названием Cacti (https://ru.wikipedia.org/wiki/Cacti), который раз-

рабатывался как улучшенный клиент для RRDtool. Благодаря наследию MRTG и RRDtool вы получите знакомый вид диаграмм, привычные шаблоны и механизм сбора данных из SNMP. Этот инструмент устанавливается и используется как цельный пакет. Но при этом он позволяет выполнять запросы — и для этого мы можем использовать Python. В этом разделе вы увидите, как из сценариев на Python взаимодействовать с Cacti.

Но сначала займемся установкой.

Установка

Учитывая, что Cacti — самодостаточный инструмент, включающий веб-клиент, набор сценариев и базу данных, я рекомендую установить его на отдельную ВМ в вашей лаборатории (если у вас уже есть опыт его использования, вы можете этого не делать). Процесс установки Cacti на управляющем хосте с Ubuntu выглядит просто и понятно, если использовать APT:

```
$ sudo apt-get install cacti
```

Эта команда инициирует последовательность шагов по установке и настройке, включая установку базы данных MySQL, веб-сервера (Apache или lighttpd) и различные конфигурационные операции. По завершении откройте http://<ip>/cacti и войдите в систему, использовав имя пользователя и пароль по умолчанию (`admin`/`admin`); вам будет предложено поменять пароль на свой.

Если при установке вы в чем-то не уверены, выбирайте вариант по умолчанию и старайтесь не усложнять этот процесс.

После входа в систему можно добавить устройство и привязать к нему шаблон, следуя инструкциям в документации. Можете начать с готового шаблона для маршрутизатора Cisco. У Cacti есть хорошее практическое руководство по добавлению устройства и созданию первой диаграммы (http://docs.cacti.net/). Взглянем на снимки экрана, чтобы вы знали, чего ожидать (рис. 7.12).

Строка со временем работы устройства свидетельствует об успешном взаимодействии по SNMP (рис. 7.13).

Вы можете подключать к устройству диаграммы для просмотра трафика, проходящего через интерфейс, и другой статистики (рис. 7.14).

Через некоторое время вы начнете видеть трафик, как показано на рис. 7.15.

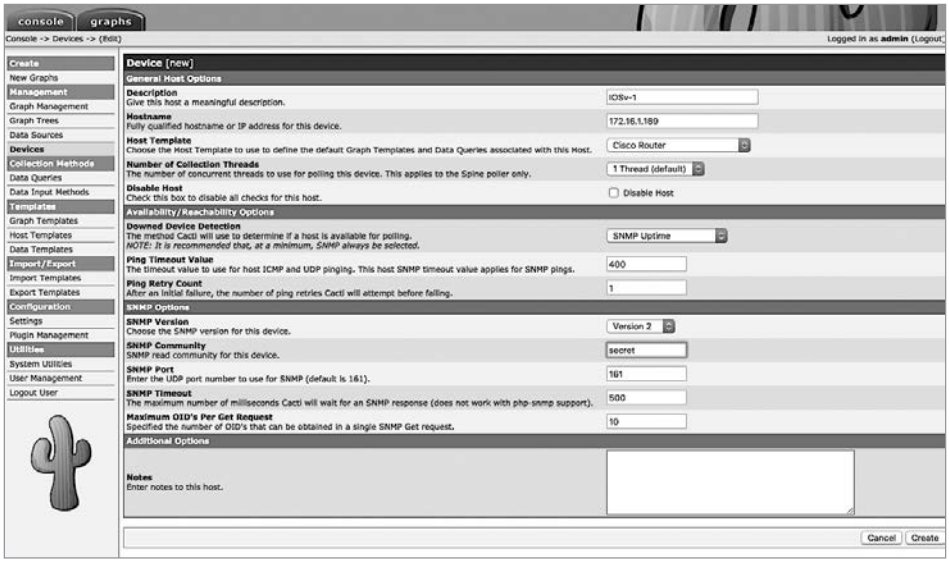


Рис. 7.12. Страница редактирования устройства в Cacti

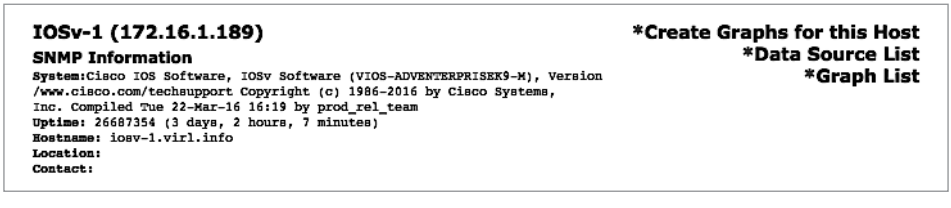


Рис. 7.13. Страница с результатами редактирования устройства

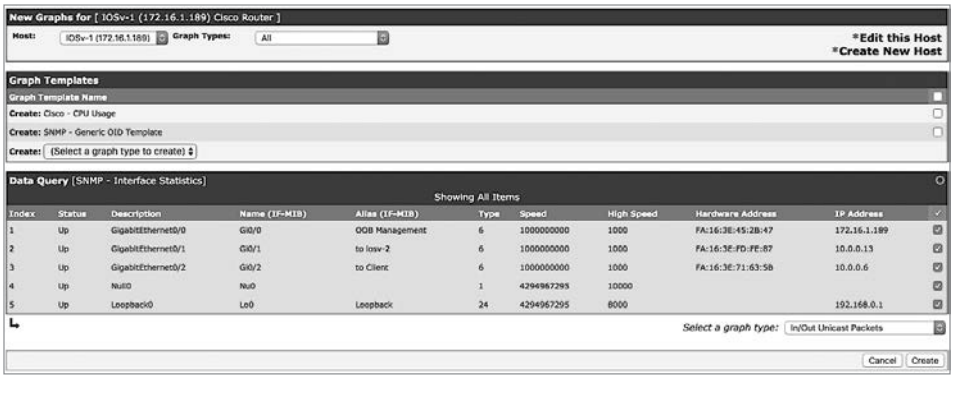


Рис. 7.14. Новые диаграммы для устройства

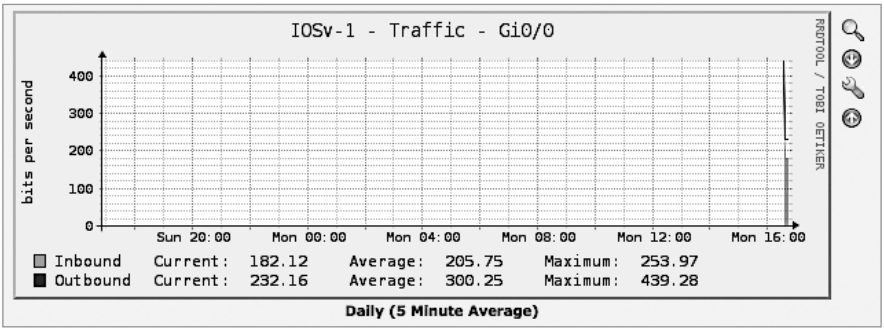


Рис. 7.15. График с усреднением за 5 минут

Теперь поговорим о расширении возможностей Cacti по сбору данных с помощью сценариев на Python.

Сценарий на Python в качестве источника данных

Прежде чем использовать сценарии в качестве источников данных, прочитайте два документа:

- *Data input methods* («Методы ввода данных»): http://www.cacti.net/downloads/docs/html/data_input_methods.html;
- *Making your scripts work with Cacti* («Как сделать, чтобы ваши сценарии работали с Cacti»): http://www.cacti.net/downloads/docs/html/making_scripts_work_with_cacti.html.

Вам, наверное, интересно узнать, в каких случаях могут пригодиться сценарии на Python в роли источников данных. Один из таких случаев — мониторинг ресурсов, не имеющих OID. Например, с помощью сценария можно построить график, показывающий, сколько раз список доступа `permit_snmp` позволил хосту `172.16.1.173` выполнить SNMP-запрос. Количество журнальных записей с этим событием можно узнать в командной строке:

```
iosv-1#sh ip access-lists permit_snmp | i 172.16.1.173 10 permit
172.16.1.173 log (6362 matches)
```

Однако у этого значения, скорее всего, нет собственного OID (или же мы можем представить, что его нет). В такой ситуации можно задействовать внешний сценарий, вывод которого сможет использовать хост Cacti.

Можно повторно использовать сценарий на основе Pexpect, `chapter1_1.py`, который обсуждался в главе 2. Переименуем его в `cacti_1.py`. Его содержимое останется прежним, только теперь мы выполним консольную команду и сохраним ее вывод:

```
<опущено>
for device in devices.keys():
...
    child.sendline('sh ip access-lists permit_snmp | i 172.16.1.173')
    child.expect(device_prompt)
    output = child.before
```

Вывод в своем исходном виде будет выглядеть так:

```
b'sh ip access-lists permit_snmp | i 172.16.1.173rn 10 permit
172.16.1.173 log (6428 matches)rn'
```

Воспользуемся функцией `split()`, чтобы получить только количество совпадений, и вернем результат в стандартный вывод сценария:

```
print(str(output).split('(')[1].split()[0])
```

Для проверки выполним сценарий несколько раз, чтобы увидеть, как увеличивается число попыток:

```
$ ./cacti_1.py
6428
$ ./cacti_1.py
6560
$ ./cacti_1.py
6758
```

Сценарий можно сделать выполняемым и поместить его в стандартный каталог Cacti для сценариев:

```
$ chmod a+x cacti_1.py
$ sudo cp cacti_1.py /usr/share/cacti/site/scripts/
```

В документации Cacti, доступной по адресу http://www.cacti.net/downloads/docs/html/how_to.html, есть инструкции по добавлению результатов работы сценария в выходной график. Там описывается добавление сценария как метода ввода данных, добавление метода ввода в источник данных и создание диаграммы, доступной для просмотра (рис. 7.16).

SNMP — распространенный механизм мониторинга, поддерживаемый сетевыми устройствами. RRDtool с Cacti в качестве клиента служит хорошей платформой для работы с любым сетевым оборудованием по SNMP. Для дополнения информации, собираемой с помощью SNMP, можно использовать сценарии на языке Python.

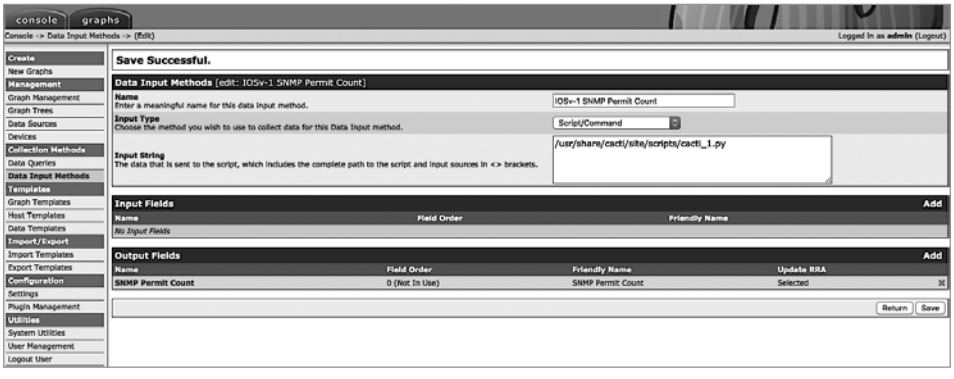


Рис. 7.16. Страница с результатами добавления метода ввода данных

Резюме

В этой главе вы увидели разные методы организации мониторинга сети посредством SNMP. Мы сконфигурировали на сетевых устройствах команды, относящиеся к SNMP, и использовали нашу управляющую ВМ с механизмом сбора информации для выполнения запросов к этим устройствам. Для упрощения и автоматизации SNMP-запросов мы использовали модуль PySNMP. Вы также научились сохранять результаты в плоский файл и базу данных для использования в дальнейших примерах.

Мы создали диаграммы на основе результатов SNMP, применяя два разных пакета визуализации для Python: Matplotlib и Pygal. У каждого из них свои преимущества. Matplotlib — зрелая, многофункциональная библиотека, которая широко используется в проектах по анализу данных. Pygal умеет самостоятельно генерировать диаграммы в формате SVG, которые отличаются гибкостью и совместимостью с браузерами. Вы научились генерировать графики и круговые диаграммы для сетевого мониторинга. Также вы познакомились с комплексной системой сетевого мониторинга под названием Cacti. Для сбора данных она в основном использует SNMP, но вы видели, как ее возможности можно расширять путем применения сценариев на Python в качестве источников данных; это полезно в ситуациях, когда на удаленном хосте нет SNMP OID.

В главе 8 мы продолжим обсуждение инструментов для мониторинга и проверки поведения наших сетей. Мы рассмотрим средства потокового мониторинга: NetFlow, sFlow и IPFIX. Мы также используем Graphviz для визуализации нашей сетевой топологии и обнаружения топологических изменений.

8

Сетевой мониторинг с использованием Python: часть 2

В главе 7 мы использовали SNMP для сбора информации о сетевом оборудовании. Для этого диспетчер SNMP выполнял запросы к SNMP-агенту, находящемуся на самом устройстве. Полученная информация была структурирована иерархически, значения объектов имели уникальные идентификаторы. В большинстве случаев нас интересуют числовые значения: загруженность процессора, объем занятой памяти или трафика, проходящего через интерфейс. На основе этих данных можно построить график с осью времени — и увидеть, как они меняются.

Подход, который применяется в SNMP, можно в целом охарактеризовать как *опрашивающий* (pull), так как мы постоянно обращаемся к устройству за определенным ответом. Это создает нагрузку на устройство, так как оно должно тратить процессорное время управляющего уровня на получение сведений о своих подсистемах, оформление ответа в виде SNMP-пакета и возвращение его запрашивающей стороне. Если вам знакома ситуация, когда на семейном торжестве ваш родственник без конца задает вам одни и те же вопросы, вы сможете представить, как диспетчер SNMP опрашивает управляемый узел.

Со временем, если у нас несколько диспетчеров, опрашивающих одно и то же устройство раз в 20 секунд (это случается на удивление часто), накладные расходы на администрирование станут довольно существенными. Если вернуться к аналогии с семейным торжеством, представьте, что вам приходится иметь дело сразу с несколькими родственниками, которые перебивают вас раз в полмину-

ты, чтобы задать какой-то вопрос. Не знаю, как вам, но мне бы это быстро надоело, даже если бы этот вопрос был простым (что еще хуже, все они могут спрашивать вас об одном и том же).

Чтобы обеспечить более эффективный мониторинг сети, можно поменять местами управляющие и управляемые хосты. Иными словами, сделать так, чтобы устройство само отправляло диспетчеру информацию в заранее согласованном формате. Именно на этой модели основан потоковый мониторинг, в рамках которого сетевое устройство направляет диспетчеру поток со сведениями о трафике. Для этого может использоваться закрытый формат Cisco NetFlow (версии 5 или 9), отраслевой стандарт IPFIX или открытый протокол sFlow. В этой главе вы познакомитесь с NetFlow, IPFIX и sFlow в контексте Python.

Не всякий мониторинг подразумевает сбор данных по времени. Конечно, при сильном желании в виде временных рядов можно представить даже топологию сети и журнал Syslog, но это будет не самым оптимальным решением. Мы можем периодически проверять, изменилась ли наша сетевая топология, с помощью Python, а для ее визуализации применять Graphviz с оберткой для Python. Как вы уже видели в главе 6, Syslog содержит информацию, связанную с безопасностью. В этой главе я покажу, как с помощью Elastic Stack (Elasticsearch, Logstash, Kibana и Beat) эффективно собирать и индексировать данные о безопасности сети и журнальные записи.

В частности, в этой главе мы рассмотрим следующие темы:

- открытый пакет визуализации Graphviz, с помощью которого можно быстро и эффективно создать графическое представление нашей сети;
- потоковый мониторинг, такой как NetFlow, IPFIX и sFlow;
- ntop для визуализации потоков данных.

Для начала посмотрим, как использовать Graphviz в качестве средства мониторинга изменений в топологии сети.

Graphviz

Graphviz — это открытое программное обеспечение для визуализации. Представьте, что нам нужно описать коллеге топологию нашей сети без изображений. Мы можем сказать, что наша сеть состоит из трех уровней: системного, распределительного и системы доступа.

Системный уровень содержит два маршрутизатора (дублирующие друг друга), каждый из которых соединен с четырьмя распределительными маршрутизаторами, а те, в свою очередь, подсоединены к маршрутизаторам доступа. Для внутренней маршрутизации используется протокол OSPF, а пиринг с внешним поставщиком услуг происходит по BGP. Этому описанию недостает некоторых подробностей, но вашему коллеге его должно быть достаточно, чтобы получить общее представление о вашей сети.

Система Graphviz работает похожим образом. Она поддерживает текстовый формат для описания графов под названием DOT (https://ru.wikipedia.org/wiki/DOT_(язык)). Получив текстовый файл с таким описанием, она может создать изображение графа. Конечно, у компьютера, в отличие от человека, нет воображения, поэтому мы должны выражаться очень точно.

С грамматикой языка DOT, характерной для Graphviz, познакомьтесь по адресу http://www.graphviz.org/doc/info/lang.html.

В этом разделе мы воспользуемся протоколом обнаружения топологии канального уровня *LLDP (Link Layer Discovery Protocol)* для обращения к соседним устройствам и создадим схему топологии сети с помощью Graphviz. Этот обширный пример продемонстрирует, как для решения интересной задачи (автоматическая визуализация текущей сетевой топологии) можно использовать сочетание новых и уже знакомых нам технологий (Graphviz и LLDP).

Подготовим лабораторию.

Подготовка лаборатории

Используем VIRL для создания лаборатории. Как и в предыдущих главах, у нас будет несколько маршрутизаторов, сервер и клиент.

Мы используем сетевые узлы под управлением IOSv и два хоста с Ubuntu (рис. 8.1).

Если вам интересно, почему мы выбрали IOSv, а не NX-OS или IOS-XR и почему у нас именно столько устройств, напомню несколько моментов, которые следует учитывать при создании лаборатории.

- Виртуализация узлов на основе NX-OS и IOS-XR требует намного больше памяти, чем на основе IOS.

- Я использую виртуальный диспетчер VIRL с 16 Гбайт оперативной памяти, чего на первый взгляд должно хватить, однако такая конфигурация может быть слегка нестабильной, если доступность узлов меняется произвольным образом.
- Если вы предпочитаете NX-OS, подумайте о NX-API или других API-вызовах, которые возвращают структурированные данные.

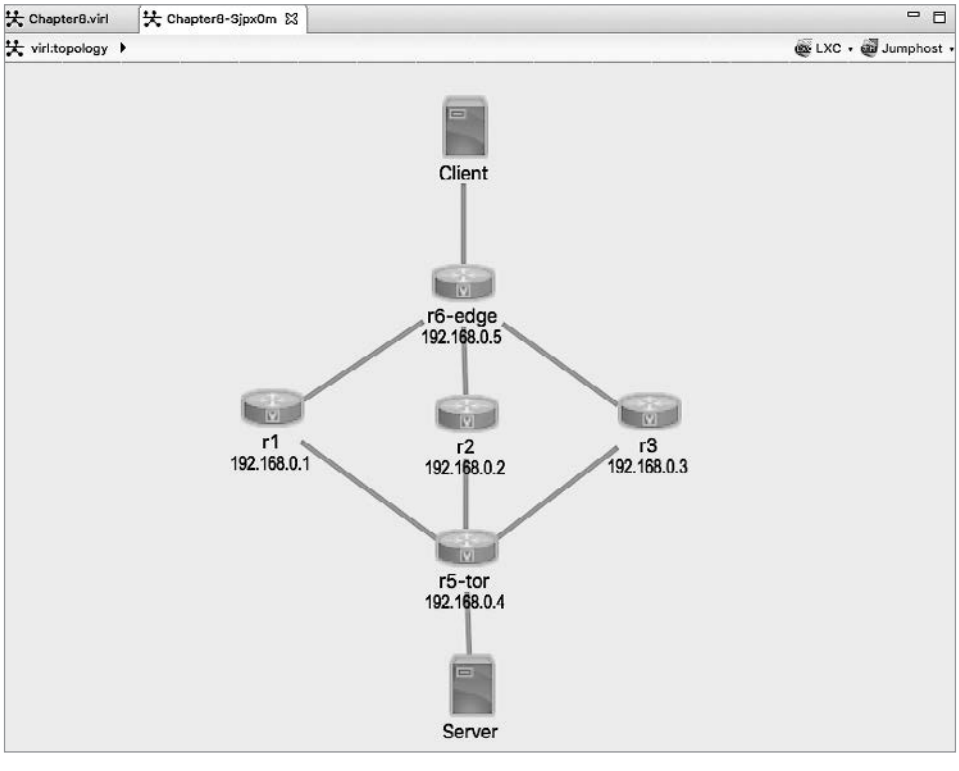


Рис. 8.1. Топология лаборатории

Для обнаружения соседних устройств на канальном уровне мы используем протокол LLDP, так как он не привязан ни к какому производителю. Обратите внимание, что VIRL позволяет автоматически включить CDP. Этот протокол, похожий по своим возможностям на LLDP, может сэкономить вам время; однако это закрытая технология компании Cisco, поэтому в нашей лаборатории она будет отключена (рис. 8.2).

Запустив сеть и убедившись в ее работоспособности, перейдем к установке необходимых программных пакетов.

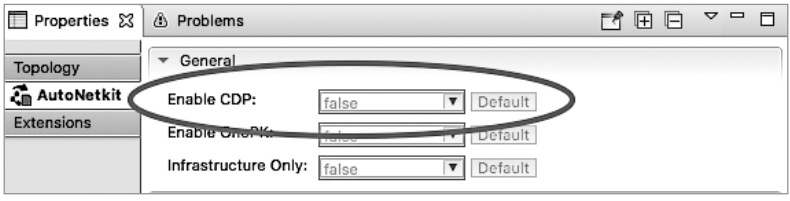


Рис. 8.2. Поддержка CDP в VIRL

Установка

Graphviz можно установить с помощью `apt`:

```
$ sudo apt-get install graphviz
```

После установки выполните проверку с помощью команды `dot`:

```
$ dot -V
dot - graphviz version 2.40.1 (20161225.0304)
```

Мы будем использовать обертку для Graphviz на языке Python, поэтому заодно установим и ее:

```
(venv)$ pip install graphviz
>>> import graphviz
>>> graphviz.__version__
'0.13'
>>> exit()
```

Теперь посмотрим, как работать с этим программным обеспечением.

Примеры работы с Graphviz

Как и большинство популярных проектов с открытым исходным кодом, Graphviz обладает обширной документацией (https://www.graphviz.org/documentation/). При знакомстве с программным продуктом зачастую сложнее всего сделать первый шаг. В примере мы сосредоточимся на построении ориентированного иерархического графа с помощью утилиты dot (не путайте с языком описания графов DOT).

Начнем с основных понятий:

- узлы представляют компоненты нашей сети, такие как маршрутизаторы, коммутаторы и серверы;
- ребра представляют соединения между компонентами сети;

- у графа, узлов и ребер есть атрибуты (https://www.graphviz.org/doc/info/attrs.html), которые можно менять;
- после описания сети мы можем сохранить ее граф (https://www.graphviz.org/doc/info/output.html) в формате PNG, JPEG или PDF.

Первым примером, `chapter8_gv_1.gv`, будет неориентированный граф из четырех узлов (`core`, `distribution`, `access1` и `access2`). Ребра, представленные дефисом (`-`), соединяют узел `core` с `distribution`, а `distribution` — с узлами `access1` и `access2`:

```
graph my_network {
    core -- distribution;
    distribution -- access1;
    distribution -- access2;
}
```

Граф можно вывести с помощью команды вида `dot -T<формат> source -o <выходной файл>`:

```
$ dot -Tpng chapter8_gv_1.gv -o output/chapter8_gv_1.png
```

Полученный граф можно просмотреть в папке `output` (рис. 8.3).

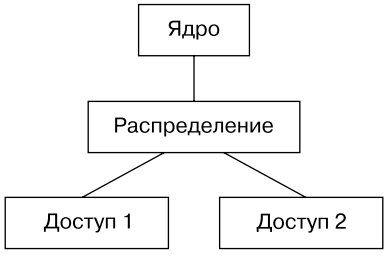


Рис. 8.3. Пример неориентированного графа, созданного с помощью Graphviz

Как и в главе 7, вам, наверное, будет проще работать с этими графами в окне виртуальной машины с настольной версией Linux; так вы сможете сразу их просматривать.

Чтобы сделать граф ориентированным, его можно описать как `digraph` (орграф), обозначив ребра стрелками (`->`). У ребер и узлов есть несколько атрибутов, которые можно изменять: форма узла, метки ребра и т. д. В файле `chapter8_gv_2.gv` показана видоизмененная версия предыдущего графа:

```
digraph my_network {
    node [shape=box];
```

```
    size = "50 30";
    core -> distribution [label="2x10G"];
    distribution -> access1 [label="1G"];
    distribution -> access2 [label="1G"];
}
```

На этот раз сохраним вывод в PDF-файл:

```
$ dot -Tpdf chapter8_gv_2.gv -o output/chapter8_gv_2.pdf
```

Посмотрите на стрелки в нашем новом графе (рис. 8.4).

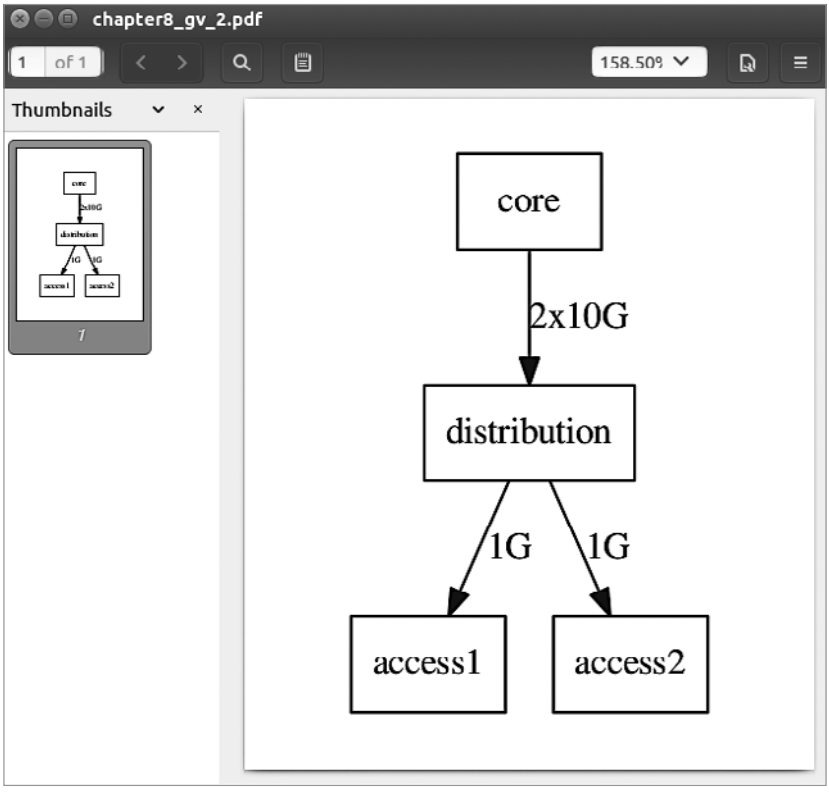


Рис. 8.4. Граф сети со стрелками и описанием соединений

Теперь рассмотрим обертку для Graphviz на языке Python.

Примеры с Graphviz и Python

Воспроизведем граф той же сети, что и в предыдущем примере, с помощью пакета Graphviz для Python. Создадим ту же трехуровневую сетевую топологию:

```
>>> from graphviz import Digraph
>>> my_graph = Digraph(comment="My Network")
>>> my_graph.node("core")
>>> my_graph.node("distribution")
>>> my_graph.node("access1")
>>> my_graph.node("access2")
>>> my_graph.edge("core", "distribution")
>>> my_graph.edge("distribution", "access1")
>>> my_graph.edge("distribution", "access2")
```

Этот код генерирует описание, которое обычно создается вручную на языке DOT, но делает это в манере Python. Перед генерацией графа просмотрим его исходный текст:

```
>>> print(my_graph.source)
// My Network
digraph {
    core
    distribution
    access1
    access2
    core -> distribution
    distribution -> access1
    distribution -> access2
}
```

Граф можно сгенерировать с помощью метода `render()`. Формат вывода по умолчанию — PDF:

```
>>> my_graph.render("output/chapter8_gv_3.gv")
'output/chapter8_gv_3.gv.pdf'
```

Обертка для Python точно повторяет все параметры API Graphviz. Документацию по этим параметрам можно найти на веб-сайте проекта (http://graphviz.readthedocs.io/en/latest/index.html). А просматривая исходный код Graphviz на GitHub (https://github.com/xflr6/graphviz), вы найдете дополнительную информацию. Теперь у нас все готово для создания схемы нашей сети.

Создание графа ближайших соседей с помощью LLDP

В этом разделе я покажу пример нахождения соседних устройств по LLDP. Он иллюстрирует подход к решению задач, который помогал мне годами.

1. По возможности разбейте каждую задачу на части. В нашем примере некоторые шаги можно объединить, но их разделение позволит нам использовать их повторно и упростит их обновление.
2. Используйте средства автоматизации для взаимодействия с сетевыми устройствами, но храните сложную логику отдельно, на управляющем хосте. Например, если маршрутизатор предоставил запутанный вывод с информацией о соседях, мы можем проанализировать его на управляющем хосте и извлечь нужные нам сведения с помощью сценария на Python.
3. Если одну и ту же задачу можно выполнить несколькими способами, выбирайте тот, который можно использовать повторно. В нашем примере для обращения к маршрутизаторам можно применить низкоуровневые сценарии на основе Pexpect и Paramiko или сценарии Ansible. По моему мнению, вариант с Ansible более универсальный, поэтому я выбрал его.

Протокол LLDP по умолчанию выключен, поэтому нам нужно сначала сконфигурировать наши маршрутизаторы. Вы уже знаете, что это можно сделать разными способами; в данном случае я выбрал сценарий Ansible с модулем `ios_config`. Файл `hosts` включает пять маршрутизаторов:

```
$ cat hosts
[devices]
r1
r2
r3
r5-tor
r6-edge
[edge-devices]
r5-tor
r6-edge
```

Для каждого хоста есть одноименный файл в папке `host_vars`. Вот, например, `r1`:

```
---
ansible_host: 172.16.1.218
ansible_user: cisco
ansible_ssh_pass: cisco
ansible_connection: network_cli
```

```
ansible_network_os: ios
ansbile_become: yes
ansible_become_method: enable
ansible_become_pass: cisco
```

Сценарий `cisco_config_lldp.yml` состоит из одного списка задач с модулем `ios_lldp`:

```
---
- name: Enable LLDP
  hosts: "devices"
  gather_facts: false
  connection: network_cli

  tasks:
    - name: enable LLDP service
      ios_lldp:
        state: present

      register: output

    - name: show output
      debug:
        var: output
```

Модуль ios_lldp входит в состав Ansible начиная с версии 2.5. Если у вас более старая версия, используйте вместо него модуль ios_config.

Выполните этот сценарий, чтобы включить LLDP:

```
$ ansible-playbook -i hosts cisco_config_lldp.yml
<опущено>
PLAY RECAP *********************************************************************
*************
r1                         : ok=2    changed=0   unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
r2                         : ok=2    changed=0   unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
r3                         : ok=2    changed=0   unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
r5-tor                     : ok=2    changed=0   unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
r6-edge                    : ok=2    changed=0   unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

По умолчанию интервал обновления LLDP составляет 30 секунд, поэтому мы должны немного подождать, пока устройства не обменяются LLDP-объявлениями. Убедимся в том, что в маршрутизаторе был включен протокол LLDP и что соседние устройства обнаружены:

```
r1#sh lldp

Global LLDP Information:
    Status: ACTIVE
    LLDP advertisements are sent every 30 seconds
    LLDP hold time advertised is 120 seconds
    LLDP interface reinitialisation delay is 2 seconds

r1#sh lldp neighbors
Capability codes:
   (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
   (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID          Local Intf     Hold-time  Capability      Port ID
r2.virl.info       Gi0/0          120        R               Gi0/0
r3.virl.info       Gi0/0          120        R               Gi0/0
r5.virl.info       Gi0/2          120        R               Gi0/1
r5.virl.info       Gi0/0          120        R               Gi0/0
r6.virl.info       Gi0/0          120        R               Gi0/0

Device ID          Local Intf     Hold-time  Capability      Port ID
r6.virl.info       Gi0/1          120        R               Gi0/1
Total entries displayed: 6
```

В своем выводе вы увидите, что `G0/0` является MGMT-интерфейсом, поэтому LLDP-соседи будут показаны так, будто они находятся в плоской управляющей сети. Нас в действительности интересуют только интерфейсы `G0/1` и `G0/2`, соединенные с другими устройствами. Эта информация пригодится при подготовке к анализу вывода и созданию графа нашей топологии.

Извлечение информации

Теперь воспользуемся еще одним сценарием Ansible, `cisco_discover_lldp.yml`, чтобы послать устройствам LLDP-команды и скопировать вывод каждого из них в каталог `tmp`.

Сначала создадим этот каталог:

```
$ mkdir tmp
```

Наш сценарий будет содержать три задачи. Первая выполнит команду поиска соседей `show lldp` на каждом из устройств, вторая отобразит вывод, а третья скопирует этот вывод в текстовый файл в каталоге `tmp`:

```
tasks:
  - name: Query for LLDP Neighbors
    ios_command:
      commands: show lldp neighbors

    register: output
```

```
  - name: show output
    debug:
      var: output

  - name: copy output to file
    copy: content="{{ output.stdout_lines }}" dest="./tmp/{{
      inventory_hostname }}_lldp_output.txt"
```

После выполнения сценария в каталоге `./tmp` появятся файлы с выводом каждого маршрутизатора (в котором перечислены их LLDP-соседи):

```
(venv) $ ls -l tmp
total 100
-rw-rw-r-- 1 echou echou   772 Oct   1 17:17 r1_lldp_output.txt
-rw-rw-r-- 1 echou echou   772 Oct   1 17:17 r2_lldp_output.txt
-rw-rw-r-- 1 echou echou   772 Oct   1 17:17 r3_lldp_output.txt
-rw-rw-r-- 1 echou echou   843 Oct   1 17:17 r5-tor_lldp_output.txt
-rw-rw-r-- 1 echou echou   843 Oct   1 17:17 r6-edge_lldp_output.txt
```

`r1_lldp_output.txt`, как и другие выходные файлы, содержит переменную `output.stdout_lines` из сценария Ansible с информацией о каждом устройстве:

```
$ cat tmp/r1_lldp_output.txt
[["Capability codes:", "    (R) Router, (B) Bridge, (T) Telephone,
(C) DOCSIS Cable Device", "    (W) WLAN Access Point, (P) Repeater,
(S) Station, (O) Other", "", "Device ID           Local Intf     Hold-
Time  Capability      Port ID", "veos01              Gi0/0          120
B               Ethernet1", "r2.virl.info        Gi0/0 120
R               Gi0/0", "r3.virl.info        Gi0/0 120
R               Gi0/0", "r5.virl.info        Gi0/2 120
R               Gi0/1", "r5.virl.info        Gi0/0 120
R               Gi0/0", "r6.virl.info        Gi0/0 120
R               Gi0/0", "r6.virl.info        Gi0/1 120                 R
Gi0/1", "", "Total entries displayed: 7"]]
```

До сих пор мы занимались извлечением информации из сетевых устройств. Теперь пришло время объединить все это в сценарии на Python.

Сценарий на Python для разбора вывода

Теперь с помощью сценария на Python разберем вывод со списком LLDP-соседей для каждого устройства и построим на основе полученных результатов граф сети. Это позволит автоматически проверять: исчезли соседние устройства из-за разрыва соединения или каких-то других проблем. Возьмем файл `cisco_graph_lldp.py` и посмотрим, как это делается.

Для начала импортируем необходимые пакеты и создадим пустой список, который будет заполнен кортежами с соединениями между узлами. Мы знаем, что

интерфейс `Gi0/0` каждого устройства соединен с управляющей сетью; следовательно, наше регулярное выражение будет искать в выводе `show LLDP neighbors` только строки вида `Gi0/[1234]`:

```
import glob, re
from graphviz import Digraph, Source

pattern = re.compile('Gi0/[1234]')

device_lldp_neighbors = []
```

Мы используем метод `glob.glob()`, чтобы обойти все файлы в каталоге `./tmp`, извлечь имена устройств и найти соседей, с которыми они соединены. Мы добавили в сценарий несколько инструкций `print`, которые выводят результат, как показано ниже; в итоговой версии их можно закомментировать.

```
$ python cisco_graph_lldp.py
device: r6-edge
  neighbors: r2
  neighbors: r1
  neighbors: r3
device: r2
  neighbors: r5
  neighbors: r6
device: r3
  neighbors: r5
  neighbors: r6
device: r5-tor
  neighbors: r3
  neighbors: r1
  neighbors: r2
device: r1
  neighbors: r5
  neighbors: r6
```

Ниже показан окончательный список ребер графа с кортежами, каждый из которых содержит имена устройства и его соседей:

```
Edges: [('r6-edge', 'r2'), ('r6-edge', 'r1'), ('r6-edge', 'r3'), ('r2',
'r5'), ('r2', 'r6'), ('r3', 'r5'), ('r3', 'r6'), ('r5-tor', 'r3'), ('r5-
tor', 'r1'), ('r5-tor', 'r2'), ('r1', 'r5'), ('r1', 'r6')]
```

Теперь можно создать граф сети, воспользовавшись пакетом Graphviz. Самая важная часть этого процесса — распаковка кортежей, которые представляют ребра между узлами:

```
my_graph = Digraph("My_Network")
my_graph.edge("Client", "r6-edge")
my_graph.edge("r5-tor", "Server")

# формируем ребра между узлами
```

```
for neighbors in device_lldp_neighbors:
    node1, node2 = neighbors
    my_graph.edge(node1, node2)
```

Полученный исходный DOT-файл будет точным представлением нашей сети:

```
digraph My_Network {

        Client -> "r6-edge"
        "r5-tor" -> Server
        "r6-edge" -> r2
        "r6-edge" -> r1
        "r6-edge" -> r3
        r2 -> r5
        r2 -> r6
        r3 -> r5
        r3 -> r6
        "r5-tor" -> r3
        "r5-tor" -> r1
        "r5-tor" -> r2
        r1 -> r5
        r1 -> r6
}
```

Некоторые соединения встречаются дважды; например, в предыдущей диаграмме два ребра, `r2 -> r5-tor` и `r5-tor -> r2`, по одному для каждого направления. Сетевые инженеры должны понимать, что иногда из-за физического сбоя соединение может стать однонаправленным; это нежелательно.

Если визуализировать эту диаграмму как есть, то узлы будут выглядеть немного странно. Местоположение узлов при отрисовке выбирается автоматически. На рис. 8.5 показан пример диаграммы с размещением элементов по умолчанию.

А вот результат, полученный с использованием движка `neato`, а именно `Digraph(My_Network, engine='neato')`. Этот движок пытается сделать неориентированный граф еще менее иерархическим (рис. 8.6).

Иногда стандартное размещение выглядит удовлетворительно, особенно если вас в первую очередь заботит обнаружение неполадок, а не внешний вид. Но давайте попробуем вставить в исходный файл элементы на языке DOT. Мы знаем, что команда `rank` позволяет разместить некоторые узлы на одном уровне. Но в API Graphviz для Python эта возможность отсутствует. К счастью, исходный DOT-файл представляет собой обычную строку, которую можно вставить как простой DOT-комментарий с помощью метода `replace()`:

```
source = my_graph.source
original_text = "digraph My_Network {"
new_text = 'digraph My_Network {\n{rank=same Client "r6-edge"}
\n{rank=same r1 r2 r3}\n'
```

```
new_source = source.replace(original_text, new_text)
print(new_source)
new_graph = Source(new_source)
new_graph.render("output/chapter8_lldp_graph.gv")
```

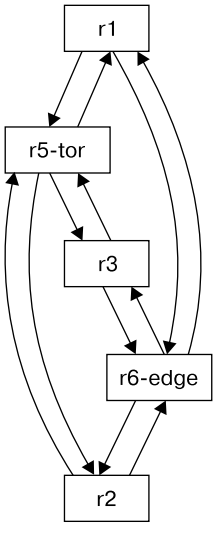


Рис. 8.5. Граф топологии 1

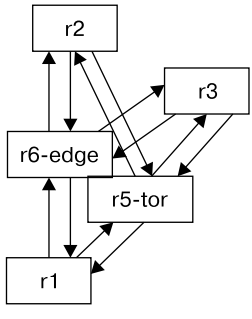


Рис. 8.6. Граф топологии 2

В результате получится новый исходный файл, на основе которого можно построить итоговый граф сети:

```
digraph My_Network {
{rank=same Client "r6-edge"}
{rank=same r1 r2 r3}

        Client -> "r6-edge"
        "r5-tor" -> Server
        "r6-edge" -> r2
        "r6-edge" -> r1
        "r6-edge" -> r3
        r2 -> r5
        r2 -> r6
        r3 -> r5
        r3 -> r6
        "r5-tor" -> r3
        "r5-tor" -> r1
        "r5-tor" -> r2
        r1 -> r5
        r1 -> r6
}
```

Теперь у нашего графа правильная иерархия (рис. 8.7).

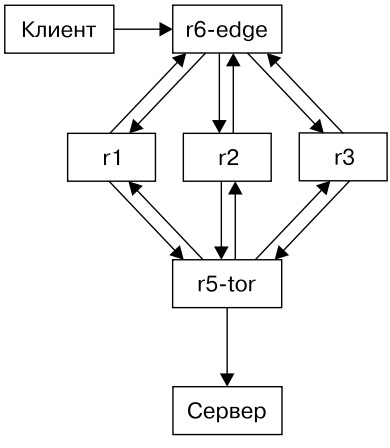


Рис. 8.7. Граф топологии 3

С помощью сценария на Python мы автоматически извлекли информацию с устройств и построили граф топологии. Это потребовало немало усилий, наградой стали согласованность и гарантия того, что граф всегда представляет фактическое состояние сети. Проверим, обнаружит ли наш сценарий изменения в топологии и отобразит ли их в виде графа.

Проверка сценария Ansible

Итак, посмотрим, сможет ли сценарий Ansible точно изобразить актуальную топологию при изменении сетевых соединений.

Для этого выключим интерфейсы `Gi0/1` и `Gi0/2` на устройстве `r6-edge`:

```
r6#confi t
Enter configuration commands, one per line.  End with CNTL/Z.
r6(config)#int gig 0/1
r6(config-if)#shut
r6(config-if)#int gig 0/2
r6(config-if)#shut
r6(config-if)#end
r6#
```

По окончании периода обновления LLDP эти интерфейсы исчезнут из таблицы на устройстве `r6-edge`:

```
r6#sh lldp neighbors
Capability codes:
    (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
    (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID           Local Intf     Hold-time    Capability    Port ID
r1.virl.info        Gi0/0          120          R             Gi0/0
r2.virl.info        Gi0/0          120          R             Gi0/0
r3.virl.info        Gi0/0          120          R             Gi0/0
r5.virl.info        Gi0/0          120          R             Gi0/0
r3.virl.info        Gi0/0          120          R             Gi0/1

Device ID           Local Intf     Hold-time    Capability    Port ID

Total entries displayed: 5
```

Если выполнить сценарии Ansible и Python, на графе будет видно, что маршрутизатор `r6-edge` соединен только с `r3` (рис. 8.8). И мы сможем приступить к поиску причин.

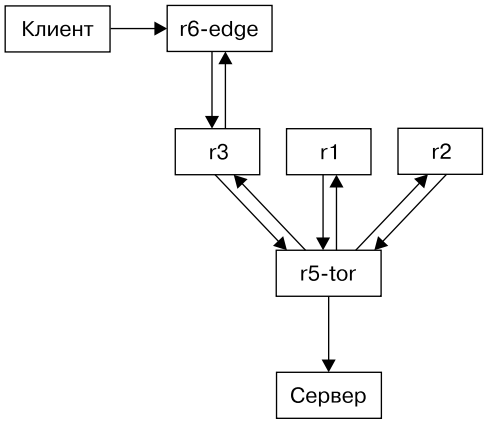


Рис. 8.8. Граф топологии 4

Этот длинный пример демонстрирует решение задачи с помощью нескольких знакомых нам инструментов, таких как Ansible и Python. Это позволило нам разбить задачи на части, которые можно использовать повторно.

Затем мы применили новый инструмент, Graphviz, для мониторинга информации, не привязанной ко времени, такой как взаимоотношения внутри сетевой топологии.

В следующем разделе мы обратимся к другой теме — мониторингу сети с помощью сбора сетевых потоков на наших сетевых устройствах.

Потоковый мониторинг

Как уже упоминалось в начале главы, помимо опрашивающих технологий, таких как SNMP, существуют стратегии, согласно которым сами устройства должны передавать информацию управляющему хосту. Примерами такого подхода являются технология NetFlow и ее близкие родственники IPFIX и sFlow. Можно сказать, что этот метод более надежный, так как устройство само решает, какие ресурсы нужно выделять для передачи информации. Если, к примеру, устройство чем-то занято, оно может отказаться от экспорта в пользу более важной задачи, например маршрутизации пакетов.

Согласно IETF (https://www.ietf.org/proceedings/39/slides/int/ip1394-background/tsld004.htm), поток — это последовательность пакетов, проходящих между отправляющим и принимающим приложениями. Если вернуться к модели OSI, поток представляет собой единицу взаимодействия между двумя приложениями. Любой поток состоит из какого-то количества пакетов: большого (как в случае с видеопотоком) или маленького (как в случае с HTTP-запросом). Если задуматься, то пакетами и кадрами озабочены только маршрутизаторы и коммутаторы, тогда как приложения и пользователей больше интересуют сетевые потоки.

Когда говорят о потоковом мониторинге, обычно имеют в виду NetFlow, IPFIX или sFlow.

- **NetFlow.** NetFlow v5 — это технология, которая позволяет сетевому устройству кешировать содержимое потоков и агрегировать пакеты, сопоставляя их с набором кортежей (по исходному интерфейсу, исходным IP/порту, конечным IP/порту и т. д.). По завершении потока устройство экспортирует его характеристики на управляющий хост, включая общее количество переданных байтов и пакетов.
- **IPFIX.** Это предложенный стандарт для структурированной потоковой передачи данных, похожий на NetFlow v9; также известен под названием Flexible NetFlow. В сущности, это механизм определяемого экспорта потоков, который дает возможность пользователю экспортировать практически все, что известно сетевому устройству. За такую гибкость часто приходится платить: IPFIX имеет более сложную конфигурацию по сравнению с традиционной технологией NetFlow v5. Из-за этого IPFIX не подходит для знакомства с потоками. Но если вы уже работали с NetFlow v5, у вас не должно возникнуть проблем с пониманием IPFIX; просто сопоставьте определения шаблонов.

- **sFlow.** На самом деле в sFlow нет понятий «поток» и «агрегация пакетов». Пакеты по этой технологии отбираются двумя методами: случайным образом (один из *n* пакетов) и периодически. Данные передаются управляющему хосту, а тот выводит на их основе информацию о сетевых потоках, учитывая тип полученной выборки и счетчики. Поскольку на сетевом устройстве не выполняется никакой агрегации, sFlow можно считать более масштабируемой технологией по сравнению с NetFlow и IPFIX.

Знакомство с этими технологиями лучше начать с примеров. И в следующем разделе мы рассмотрим примеры работы с сетевыми потоками.

Разбор NetFlow с помощью Python

Используем Python для разбора датаграммы NetFlow, передающейся по соединению. Это позволит нам подробно рассмотреть структуру пакетов NetFlow и научиться диагностировать любые проблемы с этой технологией.

Для начала сгенерируем трафик между клиентом и сервером в сети VIRL. В качестве сервера на хосте можно запустить модуль `http.server` из библиотеки Python. Откройте новое окно терминала на серверном хосте и запустите HTTP-сервер; оставьте это окно открытым:

```
cisco@Server:~$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
```

В Python 2 этот модуль называется SimpleHTTPServer. Например, python2 -m SimpleHTTPServer.

В отдельном окне терминала зайдите на клиентский хост по SSH. Создайте сценарий на Python с коротким циклом `while`, который будет непрерывно слать веб-серверу запросы HTTP GET:

```
cisco@Client:~$ cat http_get.py
import requests
import time

while True:
    r = requests.get("http://10.0.0.5:8000")
    print(r.text)
    time.sleep(5)
```

В ответ клиент должен получать раз в пять секунд очень простую HTML-страницу:

```
cisco@Client:~$ python3 http_get.py
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "www.w3.org/TR/
html4/strict.dtd">
<html>
<head>
<опущено>
</body>
</html>
```

В окне терминала с запущенным сервером мы увидим запросы, поступающие от клиента каждые пять секунд:

```
cisco@Server:~$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.0.9 - - [02/Oct/2019 00:55:57] "GET / HTTP/1.1" 200 -
10.0.0.9 - - [02/Oct/2019 00:56:02] "GET / HTTP/1.1" 200 -
10.0.0.9 - - [02/Oct/2019 00:56:07] "GET / HTTP/1.1" 200 –
```

Трафик, направленный от клиента к серверу, проходит через сетевые устройства; и с любого из этих промежуточных устройств можно экспортировать поток NetFlow. Мы выбрали маршрутизатор `r6-edge`, поскольку он является первым транзитным участком; экспортируем из него NetFlow на управляющий хост через порт 9995.

В этом примере используется всего одно устройство, поэтому мы вручную конфигурируем его с помощью команд. В следующем разделе мы включим NetFlow на всех устройствах и используем сценарий Ansible для их настройки.

Для экспорта NetFlow на устройствах под управлением Cisco IOS необходима конфигурация:

```
!
ip flow-export version 5
ip flow-export destination 172.16.1.123 9995 vrf Mgmt-intf
!
interface GigabitEthernet0/4
 description to Client
 ip address 10.0.0.10 255.255.255.252
 ip flow ingress
 ip flow egress
<опущено>
```

Теперь рассмотрим сценарий на Python, который поможет нам извлекать отдельные поля из потоков, которые передают сетевые устройства.

Модули Python socket и struct

Мы взяли за основу сценарий из статьи Брайана Рэка, см. http://blog.devicenull.org/2013/09/04/python-netflow-v5-parser. Большинство изменений в нашей версии, `netFlow_v5_parser.py`, в основном касаются совместимости с Python 3, а также разбора дополнительных полей NetFlow v5. Мы не стали использовать NetFlow v9, так как эта версия более сложная и описывает поля с помощью шаблонов, что затруднило бы знакомство с сетевыми потоками. Но версия v9 — это расширение v5, поэтому все, что будет показано в этом разделе, в равной степени относится и к ней.

Поскольку пакеты NetFlow передаются по соединению в виде байтов, мы переведем байты в стандартные типы данных Python с помощью модуля `struct`.

Больше об этих модулях читайте на страницах https://docs.python.org/3.7/library/socket.html и https://docs.python.org/3.7/library/struct.html.

Для начала с помощью модуля `socket` подключимся к порту в ожидании UDP-датаграмм. Параметры `socket.AF_INET` и `socket.SOCK_DGRAM` сигнализируют о том, что мы собираемся прослушивать сокет с IPv4-адресом и что нас интересуют UDP-датаграммы:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('0.0.0.0', 9995))
```

Запустим цикл, который в каждой итерации будет извлекать из соединения 1500 байт:

```
while True:
    buf, addr = sock.recvfrom(1500)
```

В следующей строке начинается разбор, или распаковка, полученного пакета. Знак восклицания в первом аргументе, `!HH`, говорит о том, что байты следуют в сетевом порядке (от старшего к младшему; big-endian), а два других символа обозначают формат языка C (`H` — это двухбайтное беззнаковое короткое целое):

```
(version, count) = struct.unpack('!HH',buf[0:4])
```

Первые четыре байта содержат версию и количество потоков, экспортированных в этом пакете. Если вы не можете с ходу вспомнить структуру заголовка

в NetFlow v5 (шучу; я сам читаю заголовки, только когда мне хочется побыстрее уснуть), вот небольшая шпаргалка (табл. 8.1).

Таблица 8.1. Формат заголовка в NetFlow v5 (оригинальный источник: http://www.cisco.com/c/en/us/td/docs/net_mgmt/netflow_collection_engine/3-6/user/guide/format.html#wp1006108)

Байты	Содержимое	Описание
0–1	version	Номер версии формата экспорта NetFlow
2–3	count	Количество потоков, экспортированных в этом пакете (1–30)
4–7	SysUptime	Текущее время в миллисекундах с момента загрузки экспортирующего устройства
8–11	unix_secs	Количество секунд, прошедших с начала 1970 года в UTC
12–15	unix_nsecs	Остаточные наносекунды с начала 1970 года в UTC
16–19	flow_sequence	Счетчик общего числа полученных потоков
20	engine_type	Тип механизма переключения потоков
21	engine_id	Номер механизма переключения потоков
22–23	sampling_ interval	Первые два бита хранят режим выборки; остальные 14 бит хранят длину интервала выборки

Оставшаяся часть заголовка разобирается аналогичным образом, с учетом расположения байтов и типа данных. Python позволяет распаковать сразу несколько полей заголовка в одной строке:

```
    (sys_uptime, unix_secs, unix_nsecs, flow_sequence) = struct.
unpack('!IIII', buf[4:20])
    (engine_type, engine_id, sampling_interval) = struct.
unpack('!BBH', buf[20:24])
```

Дальше идет цикл `while`, который поместит в словарь `nfdata` такие данные о потоке, как исходные и конечные адрес и порт, а также количество пакетов и байтов. Эта информация выводится ка экран:

```
nfdata = {}
for i in range(0, count):
    try:
        base = SIZE_OF_HEADER+(i*SIZE_OF_RECORD)
```

```
        data = struct.unpack('!IIIIHH',buf[base+16:base+36])
        input_int, output_int = struct.unpack('!HH', buf[base+12:base+16])
        nfdata[i] = {}
        nfdata[i]['saddr'] = inet_ntoa(buf[base+0:base+4])
        nfdata[i]['daddr'] = inet_ntoa(buf[base+4:base+8])
        nfdata[i]['pcount'] = data[0]
        nfdata[i]['bcount'] = data[1]
        nfdata[i]['stime'] = data[2]
        nfdata[i]['etime'] = data[3]
        nfdata[i]['sport'] = data[4]
        nfdata[i]['dport'] = data[5]
        print(i, " {0}:{1} -> {2}:{3} {4} packts {5} bytes".format(
            nfdata[i]['saddr'],
            nfdata[i]['sport'],
            nfdata[i]['daddr'],
            nfdata[i]['dport'],
            nfdata[i]['pcount'],
            nfdata[i]['bcount']),
            )
```

Вывод этого сценария позволяет легко визуализировать заголовок и содержимое потока. Ниже мы увидим управляющие пакеты BGP (TCP-порт 179) и HTTP-трафик (TCP-порт 8000), проходящие через `r6-edge`:

```
$ python3 netFlow_v5_parser.py
Headers:
NetFlow Version: 5
Flow Count: 6
System Uptime: 116262790
Epoch Time in seconds: 1569974960
Epoch Time in nanoseconds: 306899412
Sequence counter of total flow: 24930
0  192.168.0.3:44779 -> 192.168.0.2:179 1 packts 59 bytes
1  192.168.0.3:44779 -> 192.168.0.2:179 1 packts 59 bytes
2  192.168.0.4:179 -> 192.168.0.5:30624 2 packts 99 bytes
3  172.16.1.123:0 -> 172.16.1.222:771 1 packts 176 bytes
4  192.168.0.2:179 -> 192.168.0.5:59660 2 packts 99 bytes
5  192.168.0.1:179 -> 192.168.0.5:29975 2 packts 99 bytes
**********
Headers:
NetFlow Version: 5
Flow Count: 15
System Uptime: 116284791
Epoch Time in seconds: 1569974982
Epoch Time in nanoseconds: 307891182
Sequence counter of total flow: 24936
0  10.0.0.9:35676 -> 10.0.0.5:8000 6 packts 463 bytes
1  10.0.0.9:35676 -> 10.0.0.5:8000 6 packts 463 bytes
<опущено>
11  10.0.0.9:35680 -> 10.0.0.5:8000 6 packts 463 bytes
12  10.0.0.9:35680 -> 10.0.0.5:8000 6 packts 463 bytes
13  10.0.0.5:8000 -> 10.0.0.9:35680 5 packts 973 bytes
14  10.0.0.5:8000 -> 10.0.0.9:35680 5 packts 973 bytes
```

Стоит отметить, что в NetFlow v5 размер записи фиксированный — 48 байт, благодаря чему цикл и сценарий получились относительно простыми. Но в NetFlow v9 и IPFIX вслед за заголовком идет шаблон FlowSet (http://www.cisco.com/en/US/technologies/tk648/tk362/technologies_white_paper09186a00800a3db9.html), в котором указаны количество, типы и длина полей. Это позволяет сборщику разбирать данные, формат которых не известен заранее. Для работы с NetFlow v9 нам придется добавить в наш сценарий на Python дополнительную логику.

Разобрав данные NetFlow с помощью сценария, мы получили представление о полях этого формата, но это очень утомительный подход, который к тому же плохо масштабируется. Как вы уже, наверное, догадались, существуют другие инструменты, которые избавят нас от разбора каждой отдельной записи NetFlow. Один из таких инструментов (`ntop`) мы рассмотрим в следующем разделе.

Мониторинг трафика с помощью ntop

В этой и предыдущей главах мы видели сценарий, основанный на PySNMP, а в этой написали сценарий для разбора NetFlow. Точно так же сценарии на Python подходят для низкоуровневой работы с сетевыми соединениями. Но для тех же целей можно, к примеру, использовать многофункциональный открытый пакет Cacti, который включает в себя механизмы сбора информации, хранилище данных (RRD) и пользовательский веб-интерфейс для визуализации. Эти средства могут сэкономить вам много усилий за счет объединения часто используемых возможностей в одном пакете.

Для NetFlow существует ряд открытых и коммерческих инструментов сбора информации. Поискав среди самых популярных открытых проектов в этой области, можно найти много сравнительных исследований.

У каждого инструмента свои достоинства и недостатки; ваш выбор будет зависеть от личных предпочтений, платформы и необходимости тонкой настройки. Я бы посоветовал выбрать проект с поддержкой как v5, так и v9 (и, возможно, sFlow). Еще один фактор — знание языка, на котором написан инструмент; мне кажется, что возможность расширения сценариями на Python будет дополнительным плюсом.

Из открытых инструментов для работы с NetFlow, которые я использовал, мне нравятся NfSen (с NFDUMP в качестве серверного инструмента сбора информации) и `ntop` (или `ntopng`). Последний — более известный анализатор трафика;

он работает как в Windows, так и в Linux, и хорошо интегрируется с Python. Поэтому возьмем `ntop` в качестве примера в этом разделе.

Подобно Cacti, ntop — многофункциональный инструмент. В промышленных условиях его лучше устанавливать не на управляющем хосте.

Установка на нашем хосте с Ubuntu проста и понятна:

```
$ sudo apt-get install ntop
```

В процессе установки вам будет предложено выбрать прослушиваемый интерфейс и ввести пароль администратора. По умолчанию веб-интерфейс `ntop` прослушивает порт 3000, а сборщик прослушивает UDP-порт 5556. На сетевом устройстве нужно указать местоположение средства экспорта NetFlow:

```
!
ip flow-export version 5
ip flow-export destination 172.16.1.123 5556 vrf Mgmt-intf
!
```

IOSv по умолчанию создает запись VRF под названием Mgmt-intf и прописывает в ней Gi0/0.

В конфигурации устройства также нужно указать, какой трафик мы хотим экспортировать: входящий (`ingress`) или исходящий (`egress`):

```
interface GigabitEthernet0/0
...
ip flow ingress
ip flow egress
...
```

В архиве с примерами есть сценарий Ansible `cisco_config_netflow.yml`, который позволяет включить на устройстве экспорт NetFlow.

У устройств r5-tor и r6-edge есть два дополнительных интерфейса, в отличие от устройств r1, r2 и r3, и для включения этих интерфейсов предусмотрен еще один сценарий Ansible.

Выполните сценарий и убедитесь, что изменения применены на устройствах:

```
$ ansible-playbook -i hosts cisco_config_netflow.yml
TASK [configure netflow export station] ********************************
```

```
****************************************
changed: [r2]
changed: [r1]
changed: [r3]
changed: [r5-tor]
changed: [r6-edge]
TASK [configure flow export on Gi0/0] ********************************
****************************************
ok: [r1]
ok: [r3]
ok: [r2]
ok: [r5-tor]
ok: [r6-edge]
<опущено>
```

После выполнения сценария всегда имеет смысл проверить конфигурацию устройства, поэтому взглянем на `r2`:

```
r2#sh run
!
interface GigabitEthernet0/0
 description OOB Management
 vrf forwarding Mgmt-intf
 ip address 172.16.1.219 255.255.255.0
 ip flow ingress
 ip flow egress
<опущено>
!
ip flow-export version 5
ip flow-export destination 172.16.1.123 5556 vrf Mgmt-intf
!
```

После настройки и проверки откройте веб-интерфейс `ntop` и проверьте локальный IP-трафик (рис. 8.9).

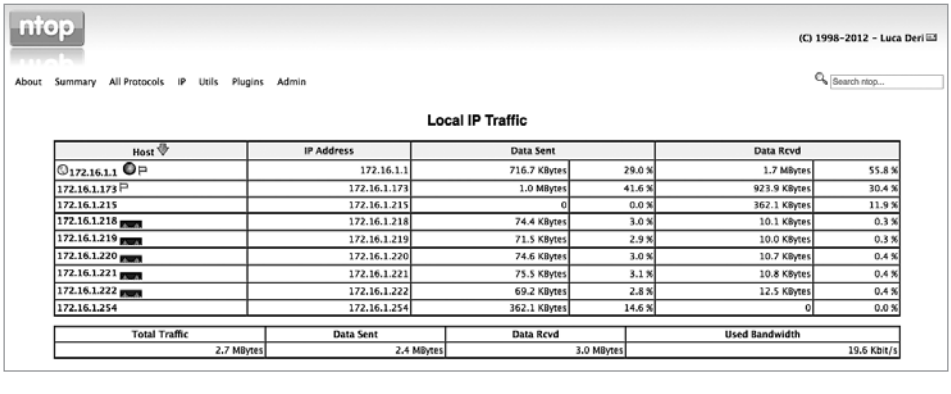


Рис. 8.9. Локальный IP-трафик в ntop

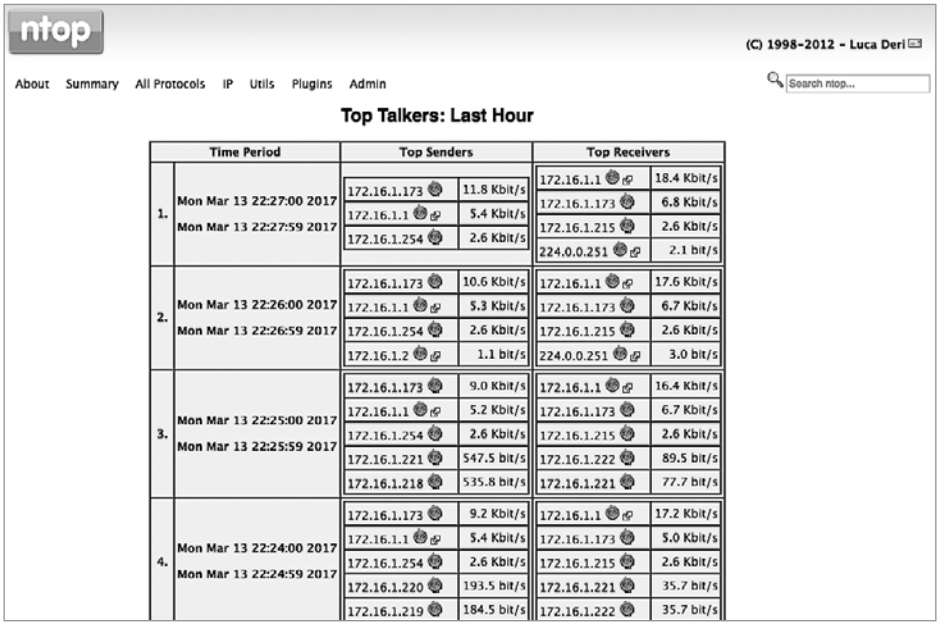


Рис. 8.10. Самые активные хосты в ntop

Одна из самых востребованных возможностей `ntop` — просмотр списка самых активных хостов (рис. 8.10).

Система генерации отчетов `ntop` написана на C; она быстрая и эффективная, но, даже просто чтобы поправить веб-интерфейс, требуется знание этого языка, поэтому данный инструмент плохо совместим с современными представлениями о гибкой разработке.

После нескольких неудачных попыток внедрения Perl в середине 2000-х годов разработчики `ntop` наконец решили выбрать Python в качестве механизма поддержки сценариев-расширений. Посмотрим, что из этого вышло.

Расширение ntop с помощью Python

Используем Python для расширения веб-сервера `ntop`. Веб-сервер `ntop` позволяет выполнять сценарии на Python для:

- доступа к состоянию `ntop`;
- обработки форм и URL-параметров с помощью модуля Python CGI;
- создания шаблонов, которые генерируют HTML-страницы.

Каждый сценарий на Python может читать из `stdin` и записывать в `stdout/stderr`. Вывод в `stdout` должен быть оформлен как HTML-страница.

Существует несколько ресурсов, которые могут пригодиться при интеграции с Python. В разделе About ▸ Show Configuration (О программе ▸ Показать конфигурацию) веб-интерфейса можно узнать версию интерпретатора Python и каталог для ваших сценариев (рис. 8.11).

Вы можете получить список каталогов, где должны храниться сценарии на Python (рис. 8.12).

Run time/Internal	
Web server URL	http://any:3000
GDBM version	GDBM version 1.8.3. 10/15/2002 (built Nov 16 2014 23:11:58)
Embedded Python	2.7.12 (default, Nov 19 2016, 06:48:10) [GCC 5.4.0 20160609]

Рис. 8.11. Версия Python

Directory (search) order	
Data Files	. /usr/share/ntop /usr/local/share/ntop
Config Files	. /usr/share/ntop /usr/local/etc/ntop /etc
Plugins	./plugins /usr/lib/ntop/plugins /usr/local/lib/ntop/plugins

Рис. 8.12. Каталоги для плагинов

В разделе About ▸ Online Documentation ▸ Python ntop Engine (О программе ▸ Онлайн-документация ▸ Поддержка Python в ntop) есть ссылки на API для Python и практическое руководство (рис. 8.13).

Как уже упоминалось, веб-сервер `ntop` автоматически выполняет сценарии на Python, размещенные в заданном каталоге:

```
$ pwd
/usr/share/ntop/python
```

Поместим в эту директорию наш первый сценарий, `chapter8_ntop_1.py`. Модуль `CGI` обрабатывает формы и разбирает URL-параметры:

```
# Импорт модулей для работы с CGI
import cgi, cgitb
import ntop
```

```
# Разбор URL
cgitb.enable();
```

ntop реализует три модуля для Python, у каждого — свое назначение:

- *ntop* — взаимодействует с ядром ntop;
- *Host* — нужен для получения информации об определенном хосте;
- *Interfaces* — представляет информацию об интерфейсах локального хоста.

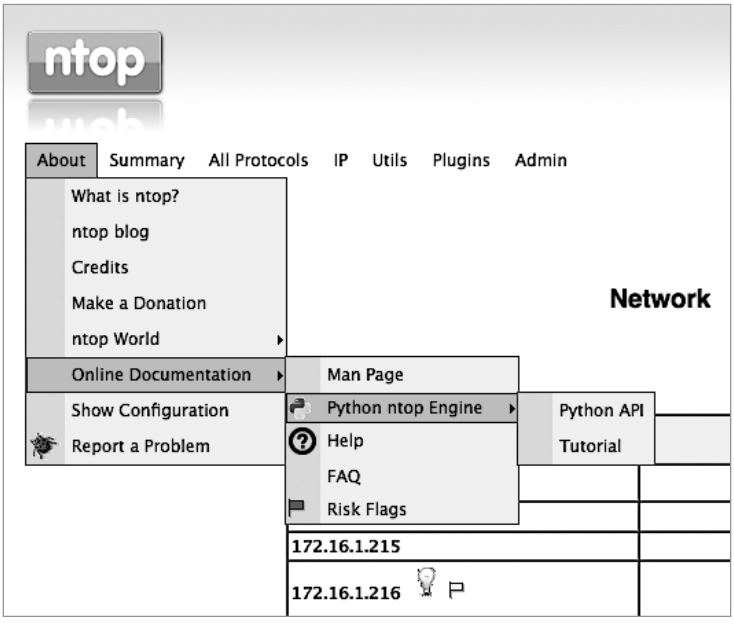


Рис. 8.13. Документация ntop для Python

Мы будем использовать модуль ntop для получения информации из ядра ntop, а также метод sendString() для отправки текста с телом HTML-документа:

```
form = cgi.FieldStorage();
name = form.getvalue('Name', default="Eric")

version = ntop.version()
os = ntop.os()
uptime = ntop.uptime()

ntop.printHTMLHeader('Mastering Python Networking', 1, 0) ntop.
sendString("Hello, "+ name +"<br>")
ntop.sendString("Ntop Information: %s %s %s" % (version, os, uptime))
ntop.printHTMLFooter()
```

Выполним этот сценарий `chapter8_ntop_1.py`, открыв в браузере адрес `http://<ip>:3000/python/<имя сценария>`. Вот результат его работы (рис. 8.14).

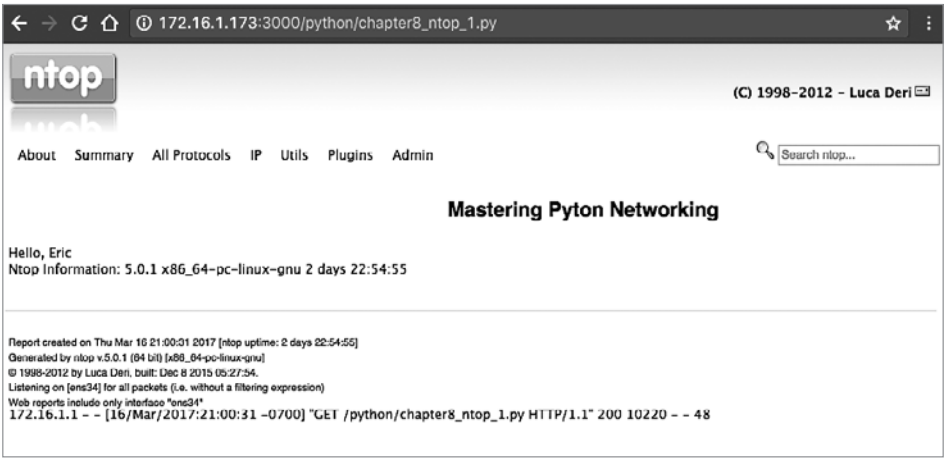


Рис. 8.14. Результат выполнения сценария ntop

Рассмотрим еще один пример — сценарий `chapter8_ntop_2.py`, который взаимодействует с модулем `interface` для перебора сетевых интерфейсов:

```
import ntop, interface, json

ifnames = []
try:
for i in range(interface.numInterfaces()):
    ifnames.append(interface.name(i))

except Exception as inst:
    print(type(inst)) # экземпляр исключения
    print(inst.args) # аргументы, хранящиеся в .args
    print(inst) # вывести args непосредственно
<опущено>
```

Он выведет на странице список интерфейсов `ntop` (рис. 8.15).

Проект `ntop` разрабатывается сообществом, но у него есть и несколько коммерческих продуктов. Активное открытое сообщество, коммерческая поддержка и расширяемость с помощью Python делает `ntop` хорошим выбором для мониторинга NetFlow.

Далее рассмотрим близкого родственника NetFlow — sFlow.

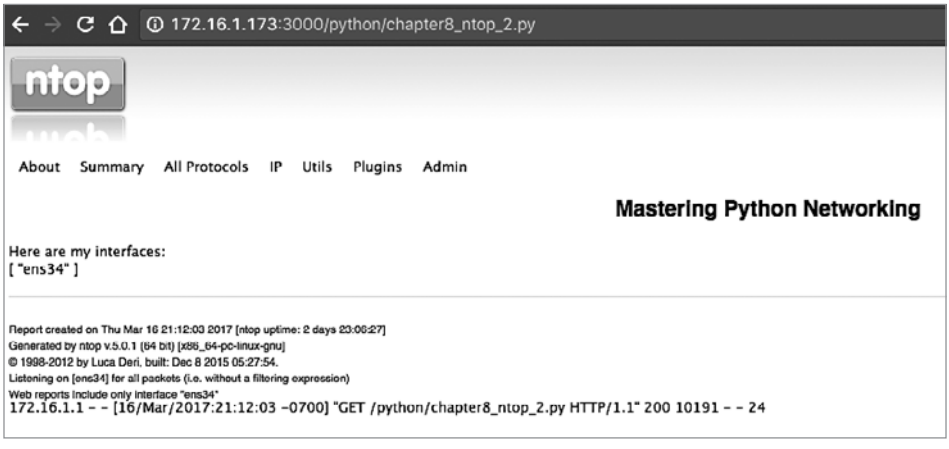


Рис. 8.15. Информация об интерфейсах ntop

sFlow

Проект sFlow (сокращенно от sampled flow — «дискретный поток») изначально разрабатывала компания InMon (http://inmon.com), и позже его стандартизировали, выпустив RFC. Текущей является версия 5. В нашей отрасли бытует мнение, что главное преимущество sFlow — в масштабируемости.

sFlow использует для оценки трафика случайную выборку пакетов (один из *n*) и выборку по интервалу; это оказывает меньшую нагрузку на процессоры сетевых устройств по сравнению с NetFlow. Функция статистической выборки sFlow интегрирована в оборудование и экспортирует необработанные данные в режиме реального времени.

По причинам масштабируемости и конкуренции новые производители, такие как Arista Networks, Vyatta и A10 Networks, отдают предпочтение sFlow перед NetFlow. Компания Cisco поддерживает sFlow только в своей линейке устройств Nexus.

Поддержка SFlowtool и sFlow-RT в Python

К сожалению, на сегодняшний день sFlow не поддерживается устройствами в нашей лаборатории VIRL (даже виртуальными коммутаторами NX-OSv). Подойдут Cisco Nexus 3000 либо коммутаторы других производителей с поддержкой sFlow, таких как Arista. Еще вариант — виртуальный сервер Arista vEOS. Лично у меня есть доступ к устройству Cisco Nexus 3048 версии 7.0 (3), которое я и использую в этом разделе для экспорта sFlow.

Сконфигурировать sFlow в Cisco Nexus 3000 просто:

```
Nexus-2# sh run | i sflow feature sflow
sflow max-sampled-size 256
sflow counter-poll-interval 10
sflow collector-ip 192.168.199.185 vrf management sflow agent-ip
192.168.199.148
sflow data-source interface Ethernet1/48
```

Для приема sFlow проще всего использовать `sflowtool`. Инструкции по установке этого инструмента ищите в документации по адресу http://blog.sflow.com/2011/12/sflowtool.html:

```
$ wget www.inmon.com/bin/sflowtool-3.22.tar.gz
$ tar -xvzf sflowtool-3.22.tar.gz
$ cd sflowtool-3.22/
$ ./configure
$ make
$ sudo make install
```

В своей лаборатории я использую более старый вариант sFlowtool. Новые версии работают точно так же.

После установки запустите `sflowtool` и просмотрите датаграмму, которую Nexus 3048 посылает в стандартный вывод:

```
$ sflowtool
startDatagram ================================
datagramSourceIP 192.168.199.148
datagramSize 88
unixSecondsUTC 1489727283
datagramVersion 5
agentSubId 100
agent 192.168.199.148
packetSequenceNo 5250248
sysUpTime 4017060520
samplesInPacket 1
startSample ----------------------
sampleType_tag 0:4 sampleType COUNTERSSAMPLE sampleSequenceNo 2503508
sourceId 2:1
counterBlock_tag 0:1001
5s_cpu 0.00
1m_cpu 21.00
5m_cpu 20.80
total_memory_bytes 3997478912
free_memory_bytes 1083838464 endSample ----------------------
endDatagram ================================
```

В репозитории `sflowtool` в GitHub есть хорошие примеры с этим инструментом (https://github.com/sflow/sflowtool); один из них демонстрирует прием ввода для

sflowtool и разбор его вывода. Мы можем использовать для этого сценарий на Python. В примере chapter8_sflowtool_1.py мы примем ввод с помощью sys.stdin.readline и используем регулярные выражения для поиска строк в пакетах sFlow, содержащих слово agent:

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

import sys, re

for line in iter(sys.stdin.readline, ''):
    if re.search('agent ', line):
        print(line.strip())
```

Сценарий можно подключить к sflowtool с помощью конвейера:

```
$ sflowtool | python3 chapter8_sflowtool_1.py
agent 192.168.199.148
agent 192.168.199.148
```

Есть и другие полезные примеры: работа с tcpdump, вывод в формате записей NetFlow v5 и компактный построчный вывод. Благодаря такой гибкости sflowtool подходит для разных окружений мониторинга.

ntop поддерживает sFlow; то есть вы можете экспортировать свой поток sFlow непосредственно в сборщик ntop. Если ваш сборщик поддерживает только NetFlow, то передайте sflowtool параметр -c, чтобы преобразовать вывод в формат NetFlow v5:

```
$ sflowtool --help
...
tcpdump output:
-t - (output in binary tcpdump(1) format)
-r file - (read binary tcpdump(1) format)
-x - (remove all IPV4 content)
-z pad - (extend tcpdump pkthdr with this many zeros
e.g. try -z 8 for tcpdump on Red Hat Linux 6.2)

NetFlow output:
-c hostname_or_IP - (netflow collector host)
-d port - (netflow collector UDP port)
-e - (netflow collector peer_as (default = origin_as))
-s - (disable scaling of netflow output by sampling rate)
-S - spoof source of netflow packets to input agent IP
```

Альтернативная система анализа sFlow — sFlow-RT от InMon (http://www.sflow-rt.com/index.php). Главная особенность sFlow-RT с точки зрения операторов — богатый RESTful API, который можно подстраивать под свои задачи. Из него можно извлекать различные метрики. Познакомьтесь с этим API на странице http://www.sflow-rt.com/reference.php.

Обратите внимание, что для работы sFlow-RT требуется Java:

```
$ sudo apt-get install default-jre
$ java -version
openjdk version "1.8.0_121"
OpenJDK Runtime Environment (build 1.8.0_121-8u121-b13-0ubuntu1.16.04.2-b13)
OpenJDK 64-Bit Server VM (build 25.121-b13, mixed mode)
```

Установив необходимые зависимости, вы легко загрузите и запустите sFlow-RT (https://sflow-rt.com/download.php):

```
$ wget www.inmon.com/products/sFlow-RT/sflow-rt.tar.gz
$ tar -xvzf sflow-rt.tar.gz
$ cd sflow-rt/
$ ./start.sh
2017-03-17T09:35:01-0700 INFO: Listening, sFlow port 6343
2017-03-17T09:35:02-0700 INFO: Listening, HTTP port 8008
```

Чтобы проверить успешность установки, введите в браузере адрес с HTTP-портом 8008 (рис. 8.16).

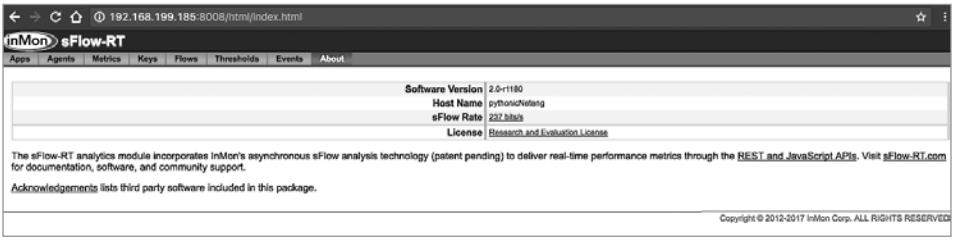


Рис. 8.16. Версия sFlow-RT

Как только sFlow-RT получит какие-либо пакеты sFlow, на странице появятся агенты и другие метрики (рис. 8.17).

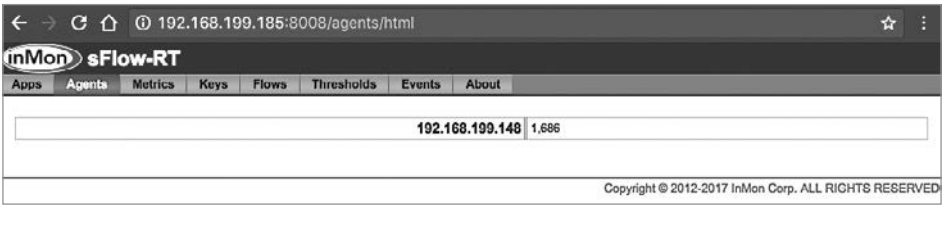


Рис. 8.17. IP-адрес агента sFlow-RT

Вот два примера использования Python-модуля `requests` для извлечения информации из REST API sFlow-RT:

```
>>> import requests
>>> r = requests.get("192.168.199.185:8008/version")
>>> r.text '2.0-r1180'
>>> r = requests.get("192.168.199.185:8008/agents/json")
>>> r.text
'{"192.168.199.148": {n "sFlowDatagramsLost": 0,n
"sFlowDatagramSource": ["192.168.199.148"],n "firstSeen": 2195541,n
"sFlowFlowDuplicateSamples": 0,n "sFlowDatagramsReceived": 441,n
"sFlowCounterDatasources": 2,n "sFlowFlowOutOfOrderSamples": 0,n
"sFlowFlowSamples": 0,n "sFlowDatagramsOutOfOrder": 0,n "uptime":
4060470520,n "sFlowCounterDuplicateSamples": 0,n "lastSeen":
3631,n "sFlowDatagramsDuplicates": 0,n "sFlowFlowDrops": 0,n
"sFlowFlowLostSamples": 0,n "sFlowCounterSamples": 438,n
"sFlowCounterLostSamples": 0,n "sFlowFlowDatasources": 0,n
"sFlowCounterOutOfOrderSamples": 0n}}'
```

О других доступных вам конечных точках REST можно узнать в документации.

В предыдущих изданиях книги в этой главе имелся раздел об ELK Stack. В данном издании этому пакету посвящена отдельная глава.

Мы рассмотрели примеры мониторинга на основе sFlow как в виде отдельных инструментов, так и в комплексе с `ntop`. sFlow — один из новых форматов сетевых потоков; его авторы попытались решить проблемы с масштабированием, присущие традиционным форматам NetFlow. Определенно стоит потратить время, чтобы понять, подходит ли этот инструмент для ваших задач. Мы приблизились к концу главы, поэтому вспомним, какие темы в ней обсуждались.

Резюме

В этой главе мы рассмотрели традиционные способы использования Python для улучшения механизмов мониторинга сети. Мы начали с пакета Graphviz для построения графов сетевой топологии на основе данных LLDP, возвращаемых сетевыми устройствами в режиме реального времени, и научились с легкостью выводить текущую топологию сети и находить разрывы соединений.

Затем мы использовали Python для анализа пакетов NetFlow v5, чтобы лучше понять потоки NetFlow и провести их диагностику. И увидели, как Python позволяет расширить возможности мониторинга NetFlow в `ntop`. sFlow — альтернативная технология выборки пакетов; результаты ее работы мы интерпретировали с помощью `sflowtool` и sFlow-RT.

В главе 9 мы поговорим о создании сетевых веб-сервисов с помощью веб-фреймворка Flask на языке Python.

9

Создание сетевых веб-сервисов с помощью Python

В предыдущих главах мы выступали потребителями API, предоставляемых кем-то другим. В главе 3 вы увидели, что по URL `http://<ip вашего маршрутизатора>/ins` можно послать HTTP-запрос типа `POST` с командой, встроенной в его тело, чтобы удаленно управлять устройством Cisco Nexus с поддержкой NX-API; в своем HTTP-ответе устройство возвращало результат выполнения команды. В главе 8 мы использовали HTTP-запрос типа `GET` с пустым телом, чтобы обратиться по адресу <ip вашего хоста>:8008/version и получить версию sFlow-RT, установленного на устройстве. Взаимодействие вида «запрос — ответ» — пример веб-сервисов типа RESTful.

Вот что написано в Википедии (https://ru.wikipedia.org/wiki/REST):

> *«Передача состояния представления (Representational State Transfer, REST) — один из способов организации взаимодействий между компьютерными системами в интернете. Веб-сервисы, совместимые с REST, позволяют запрашивающим системам читать и изменять текстовое представление веб-ресурсов, используя унифицированный и заранее определенный набор операций без сохранения состояния».*

Как уже отмечалось, веб-сервисы типа RESTful на основе протокола HTTP — это лишь один из многих методов обмена данными в интернете; существуют и другие виды веб-сервисов. Однако на сегодня это самый распространенный подход, в котором обмен информацией происходит с помощью предопределенных HTTP-методов, таких как `GET`, `POST`, `PUT` и `DELETE`.

В данном контексте HTTPS используется в качестве защищенного расширения HTTP (https://ru.wikipedia.org/wiki/HTTPS) и внутреннего протокола для RESTful API.

С точки зрения провайдера преимущество предоставления REST-сервисов — возможность скрывать от пользователей внутреннюю работу. Например, в случае с sFlow-RT, если мы зайдем на устройство, чтобы проверить версию установленного на нем ПО, нам понадобятся углубленные знания соответствующего инструмента, иначе мы не будем знать, где искать. Предоставляя ресурсы в виде URL, провайдер API может скрыть действия по проверке версии от запрашивающей стороны, делая данную операцию намного проще. Такого рода инкапсуляция дополнительно обеспечивает безопасность, так как провайдер может открывать конечные точки только по мере необходимости. Полностью контролируя свою сеть, вы получаете существенные преимущества от использования веб-сервисов типа RESTful, в том числе:

- можно скрыть от запрашивающей стороны внутреннюю работу сети. Например, предоставить веб-сервис для получения версии коммутатора, не требуя от пользователя знания подходящей консольной команды или API устройства;
- мы можем объединять и видоизменять операции, которые подходят именно для нашей сети. Это, к примеру, может быть ресурс для обновления всех наших TOR-коммутаторов;
- можно повысить уровень безопасности, предоставляя доступ только к необходимым операциям. Например, для основных сетевых устройств можно предусмотреть URL только для чтения (`GET`), а URL для коммутаторов уровня доступа могут поддерживать как чтение, так и запись (`GET` / `POST` / `PUT` / `DELETE`).

В этой главе мы воспользуемся одним из популярнейших веб-фреймворков на языке Python, *Flask*, чтобы создать собственный веб-сервис типа RESTful для нашей сети.

В этой главе мы обсудим следующие темы:

- сравнение веб-фреймворков для Python;
- введение в фреймворк Flask;
- операции для работы со статическим сетевым контентом;
- операции для работы с динамическим сетевым контентом;
- аутентификацию и авторизацию;
- запуск нашего веб-приложения в контейнерах.

Для начала рассмотрим доступные веб-фреймворки для Python и объясним, почему мы выбрали Flask.

Сравнение веб-фреймворков для Python

Язык Python славится широким выбором веб-фреймворков. В сообществе Python часто шутят, что разработчику на этом языке нельзя найти полноценную работу, где бы ему не пришлось иметь дело с каким-нибудь фреймворком. У одного из самых популярных фреймворков для Python, Django, даже есть ежегодная конференция под названием DjangoCon, которую регулярно посещают сотни участников. На конец 2019 года единственная конференция по Flask состоялась в Бразилии в 2018 году. Также стоит упомянуть конференцию более общего характера, PyConWeb, которую провели весной 2019 года. Я уже говорил, что у Python активное сообщество разработчиков?

На странице https://hotframeworks.com/languages/python можно ознакомиться с длинным списком фреймворков для Python (рис. 9.1).

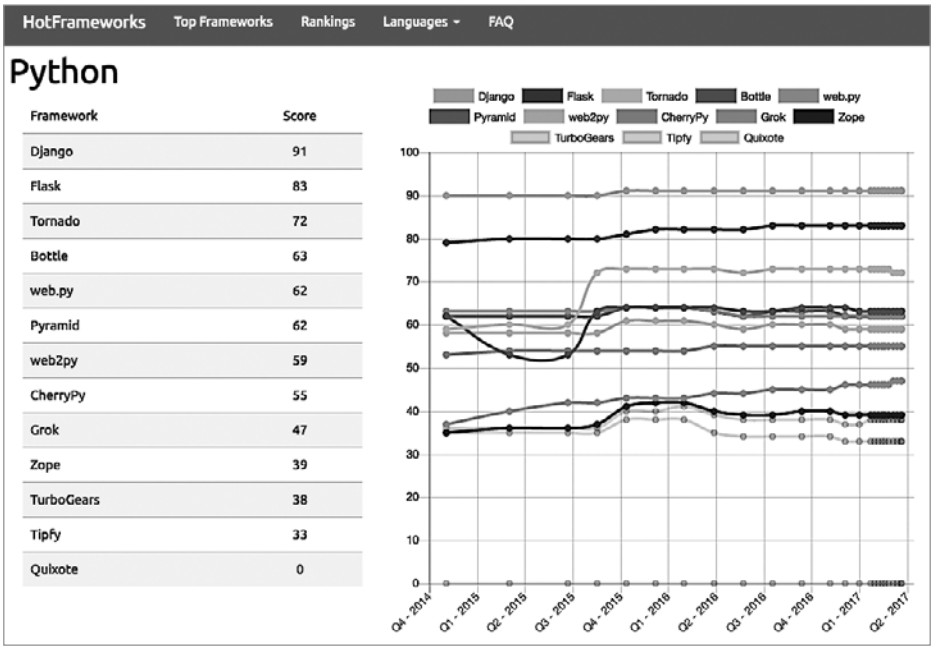


Рис. 9.1. Рейтинг веб-фреймворков на языке Python

На каком же из них нам остановиться, учитывая такой богатый выбор? Чтобы опробовать каждый, понадобится слишком много времени. Вопрос о лучшем веб-фреймворке вызывает бурные дискуссии среди веб-разработчиков. Если вы зададите его на каком-нибудь форуме вроде Quora или поищете по Reddit, будьте готовы к очень предвзятым ответам и жарким спорам.

Если уж речь зашла о Quora и Reddit, стоит упомянуть, что оба эти форума написаны на Python. И тот и другой используют Pylons (https://www.reddit.com/wiki/faq#wiki_so_what_python_framework_do_you_use.3F), только разработчики Quora заменили часть этого фреймворка собственным кодом (https://www.quora.com/What-languages-and-frameworks-are-used-to-code-Quora).

Конечно, у меня есть свои предпочтения в языках программирования (Python!) и веб-фреймворках (Flask!). В этом разделе я постараюсь объяснить, чем продиктован мой выбор. Возьмем два самых популярных фреймворка из списка HotFrameworks и сравним их.

- **Django.** Это крупный высокоуровневый веб-фреймворк, который поощряет высокие темпы разработки и имеет чистую и прагматичную архитектуру (https://www.djangoproject.com/). Как утверждают его авторы, это «веб-фреймворк для перфекционистов, ограниченных сжатыми сроками». Он содержит готовый код с панелью администрирования и встроенными средствами управления контентом.
- **Flask.** Это микрофреймворк для Python, основанный на Werkzeug, Jinja2 и других проектах (https://palletsprojects.com/p/flask/). Приставка «микро» означает, что у Flask компактное ядро, которое при необходимости можно расширять. И это не означает отсутствие какой-то функциональности или неготовность для промышленного использования.

Лично я использую Django для крупных проектов, а Flask — для создания прототипов. Django навязывает строго определенный подход к тому, как следует делать те или иные вещи; если вы от него отклонитесь, у вас возникнет ощущение того, что вы «боретесь с фреймворком». Например, в документации Django (https://docs.djangoproject.com/en/2.2/ref/databases/) говорится, что этот фреймворк поддерживает несколько разных баз данных. Однако все они реляционные: MySQL, PostgreSQL, SQLite и т. п.

Что, если вам нужна БД типа NoSQL, такая как MongoDB или CouchDB? Возможно, вам удастся ее внедрить, но многое придется делать вручную. Во фрейм-

ворках, которые навязывают определенные методы работы, нет ничего плохого — это дело вкуса.

Концепция компактного ядра, которое при необходимости можно расширить, нравится тем, кому нужно что-то простое и быстрое. Первый пример в документации Flask состоит всего из шести строк, и его легко понять, даже если вы незнакомы с этим фреймворком. Поскольку архитектура Flask изначально предусматривает расширяемость, написание собственных расширений, таких как декораторы, не вызывает сложностей. Flask — микрофреймворк, но его ядро содержит все необходимые компоненты, включая сервер для разработки, отладчик, интеграцию с модульными тестами, отправку REST-запросов и др. Вы можете использовать его без установки дополнительного ПО.

Как видите, Flask занимает второе место в рейтинге популярности среди всех фреймворков на языке Python, уступая только Django. Активное сообщество, хорошая поддержка и высокие темпы разработки способствуют широкому распространению этого проекта.

По причинам, перечисленным выше, я считаю, что Flask идеально подходит для знакомства с процессом создания сетевых веб-сервисов — а это именно то, что нам нужно.

Flask и подготовка лаборатории

В этой главе мы продолжим использовать виртуальное окружение, чтобы изолировать среду Python и сопутствующие зависимости. Можете создать новое окружение или продолжить работать в прежнем.

Нам нужно установить много Python-пакетов. Чтобы упростить этот процесс, я включил в репозиторий книги на GitHub файл `requirements.txt`; с его помощью можно установить все необходимые зависимости (не забудьте активировать свое виртуальное окружение). В конце вы увидите, как эти пакеты загружаются и успешно устанавливаются:

```
(venv) $ cat requirements.txt
Flask==1.1.1
Flask-HTTPAuth==3.3.0
Flask-SQLAlchemy==2.4.1
Jinja2==2.10.1
MarkupSafe==1.1.1
Pygments==2.4.2
```

```
SQLAlchemy==1.3.9
Werkzeug==0.16.0
httpie==1.0.3
itsdangerous==1.1.0
python-dateutil==2.8.0
requests==2.20.1
(venv) $ pip install -r requirements.txt
```

Мы используем простую сетевую топологию с четырьмя узлами (рис. 9.2).

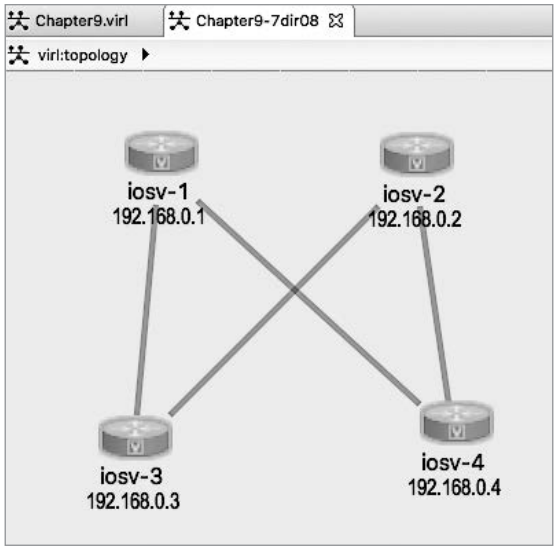


Рис. 9.2. Топология лаборатории

В следующем разделе мы познакомимся с Flask.

В дальнейшем я буду исходить из того, что вы выполняете все свои сценарии в виртуальном окружении и что у вас установлены все необходимые пакеты, указанные в файле requirements.txt.

Введение в фреймворк Flask

Как и у большинства популярных проектов с открытым исходным кодом, у Flask очень хорошая документация, она доступна по адресу https://flask.palletsprojects.com/en/1.1.x/. Если вы хотите познакомиться с Flask поближе, начните с нее.

Я также настоятельно рекомендую ознакомиться с материалами Мигеля Гринберга (https://blog.miguelgrinberg.com/) по Flask. Я многое почерпнул из его блога, книги и видеокурсов. Лекция Мигеля под названием Building Web APIs with Flask вдохновила меня на написание этой главы. Код, который он публикует, доступен на GitHub: https://github.com/miguelgrinberg/oreilly-flask-apis-video.

Наше первое приложение на основе Flask состоит из одного файла, `chapter9_1.py`:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_networkers():
    return 'Hello Networkers!'

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

Это чуть ли не шаблон проектирования, который мы будем использовать в следующих примерах с Flask. Мы создали экземпляр класса `Flask`, передав в первом аргументе имя модуля пакета приложения. В данном случае используется единственный модуль, который может запускаться как приложение; позже вы увидите, как его можно импортировать в виде пакета. Затем мы воспользовались декоратором `route`, чтобы фреймворк знал, какой URL должна обрабатывать функция `hello_networkers()`; в данном случае мы указали корневой путь. В конце выполняется обычная проверка пространства имен (https://docs.python.org/3.7/library/__main__.html), чтобы определить, когда модуль запускается как сценарий. Мы также добавили параметры `host` и `debug`, чтобы получить более подробный вывод и чтобы на хосте прослушивались все интерфейсы (по умолчанию прослушивается только петлевой интерфейс). Это приложение можно запустить с помощью сервера для разработки:

```
(venv) $ python chapter9_1.py
* Serving Flask app "chapter9_1" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production
  deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 141-973-077
```

Теперь можно проверить работоспособность сервера, обратившись к нему из HTTP-клиента.

Клиент HTTPie

Мы уже установили HTTPie (https://httpie.org/), когда использовали файл `requirements.txt`. HTTPie имеет улучшенную цветную подсветку синтаксиса для HTTP-транзакций. HTTPie также обеспечивает интуитивно понятное взаимодействие с REST-сервером из командной строки. Этот инструмент подойдет для проверки нашего первого приложения на основе Flask (далее будет больше примеров с HTTPie). Откроем на управляющем хосте второе окно терминала, активируем виртуальное окружение и введем следующее:

```
(venv) $ http http://192.168.2.123:5000
HTTP/1.0 200 OK
Content-Length: 17
Content-Type: text/html; charset=utf-8
Date: Tue, 08 Oct 2019 19:06:23 GMT
Server: Werkzeug/0.16.0 Python/3.6.8

Hello Networkers!
```

Для сравнения: чтобы получить тот же вывод с помощью curl, нужно указать параметр -i: curl -i http://192.168.2.123:5000.

В этой главе в качестве клиента мы выбрали HTTPie; давайте потратим несколько минут, чтобы узнать, как им пользоваться. Для демонстрации используем бесплатный веб-сайт HTTP Bin (https://httpbin.org/). Работа с HTTPie следует простому шаблону:

```
$ http [flags] [METHOD] URL [ITEM]
```

Как мы уже видели в примере с сервером для разработки Flask, отправка `GET`-запроса выглядит очень просто:

```
$ http GET https://httpbin.org/user-agent
<опущено>
{
    "user-agent": "HTTPie/1.0.3"
}
```

В качестве типа содержимого в HTTPie по умолчанию используется JSON. Если тело вашего HTML-документа содержит лишь текст, вам не нужно выполнять никаких других операций. Для передачи нестроковых полей JSON используй-

те := или другие задокументированные спецсимволы. В следующем примере мы передаем переменную "married" как булево значение, а не строковое:

```
$ http POST https://httpbin.org/post name=eric twitter=at_ericchou
married:=true
<опущено>
Content-Type: application/json
<опущено>

{
    <опущено>
    "headers": {
        "Accept": "application/json, */*",
        <опущено>
        "User-Agent": "HTTPie/1.0.3"
    },
    "json": {
        "married": true,
        "name": "eric",
        "twitter": "at_ericchou"
    },
    <опущено>
    "url": "https://httpbin.org/post"
}
```

Как видите, HTTPie имеет существенно улучшенный синтаксис по сравнению с curl, что делает тестирование REST API легким и приятным.

Дополнительные примеры использования доступны по адресу https://httpie.org/doc#usage.

Вернемся к нашей программе на основе Flask. Процесс создания API во многом полагается на маршрутизацию URL. Рассмотрим декоратор app.route().

Маршрутизация URL

В chapter9_2.py мы создали две дополнительные функции и связали их с подходящими маршрутами, используя app.route():

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'You are at index()'

@app.route('/routers/')
```

```
def routers():
    return 'You are at routers()'

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

В результате разные конечные точки передаются разным функциям. В этом можно убедиться с помощью двух HTTP-запросов:

```
# На стороне сервера
$ python chapter9_2.py
<опущено>
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

# На стороне клиента
$ http http://192.168.2.123:5000
<опущено>

You are at index()

$ http http://192.168.2.123:5000/routers/
<опущено>

You are at routers()
```

Запросы, поступающие с клиентской стороны, отображаются в командной строке сервера:

```
(venv) $ python chapter9_2.py
<опущено>
192.168.2.123 - - [08/Oct/2019 12:43:08] "GET / HTTP/1.1" 200 -
192.168.2.123 - - [08/Oct/2019 12:43:18] "GET /routers/ HTTP/1.1" 200 –
```

Как видите, разные конечные точки соответствуют разным функциям; сервер берет результат, который производит функция, и возвращает его запрашивающей стороне. Конечно, если все наши пути статические, маршрутизация будет ограниченной. Но мы можем передавать во Flask динамические переменные, используя URL; пример этого — в следующем подразделе.

URL-переменные

В примере `chapter9_3.py` показано, как в URL можно передавать динамические переменные:

```
<опущено>
@app.route('/routers/<hostname>')
def router(hostname):
    return 'You are at %s' % hostname
```

```
@app.route('/routers/<hostname>/interface/<int:interface_number>')
def interface(hostname, interface_number):
    return 'You are at %s interface %d' % (hostname, interface_number)
<опущено>
```

В момент, когда клиент выполняет запрос, этим двум функциям передается динамическая информация: имя хоста и номер интерфейса. Обратите внимание, что в URL `/routers/<hostname>` переменная `<hostname>` передается в виде строки, а в `/routers/<hostname>/interface/<int:interface_number>` переменная должна быть целочисленной (`int`). Запустим этот пример и выполним несколько запросов:

```
# На стороне сервера
(venv) $ python chapter9_3.py

(venv) # На стороне клиента
$ http http://192.168.2.123:5000/routers/host1
HTTP/1.0 200 OK
<опущено>

You are at host1

(venv) $ http http://192.168.2.123:5000/routers/host1/interface/1
HTTP/1.0 200 OK
<опущено>

You are at host1 interface 1
```

Если переменная `interface_number` НЕ является целым числом, мы получим ошибку:

```
(venv) $ http http://192.168.2.123:5000/routers/host1/interface/one
HTTP/1.0 404 NOT FOUND
<опущено>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL
manually please check your spelling and try again.</p>
```

Flask умеет преобразовывать целые и вещественные значения, а также пути (со слешами).

Помимо сопоставления статических маршрутов с динамическими переменными, мы можем генерировать URL при запуске приложения. Это очень полезно в ситуациях, когда конечная точка хранится в переменной или зависит от других условий, таких как значения, запрашиваемые из базы данных, и мы не знаем заранее, как она выглядит. Рассмотрим пример:

Генерация URL

Попробуем при запуске приложения `chapter9_4.py` динамически создать URL `/<hostname>/list_interfaces`, где `hostname` может быть `r1`, `r2` или `r3`. Как мы уже знаем, для этого можно сконфигурировать три маршрута с тремя соответствующими функциями. Но посмотрим, как сделать это динамически, во время запуска приложения:

```
from flask import Flask, url_for

app = Flask(__name__)

@app.route('/<hostname>/list_interfaces')
def device(hostname):
    if hostname in routers:
        return 'Listing interfaces for %s' % hostname
    else:
        return 'Invalid hostname'

routers = ['r1', 'r2', 'r3']
for router in routers:
    with app.test_request_context():
        print(url_for('device', hostname=router))

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

При выполнении мы получим несколько аккуратных, логичных URL, соответствующих списку маршрутизаторов, и при этом нам не пришлось статически определять каждый из них:

```
# сторона сервера
(venv) $ python chapter9_4.py
<опущено>
/r1/list_interfaces
/r2/list_interfaces
/r3/list_interfaces

# сторона клиента
(venv) $ http http://192.168.2.123:5000/r1/list_interfaces
<опущено>

Listing interfaces for r1

(venv) $ http http://192.168.2.123:5000/r2/list_interfaces
<опущено>

Listing interfaces for r2
```

```
# некорректный запрос
(venv) $ http http://192.168.2.123:5000/r1000/list_interfaces
<опущено>

Invalid hostname
```

Пока что `app.text_request_context()` можно считать фиктивным объектом запроса, необходимым в демонстрационных целях. Если вас интересует локальный контекст, ознакомьтесь с документацией http://werkzeug.pocoo.org/docs/0.14/local/. Динамическая генерация конечных точек существенно экономит время, делая код намного проще и понятнее.

Возвращение результата с помощью jsonify

Еще один механизм Flask, который сэкономит вам время, — `jsonify()`. Это обертка вокруг `json.dumps()`, которая превращает вывод в формате JSON в объект ответа с `application/json` в HTTP-заголовке `Content-Type`. Ниже показан сценарий `chapter9_5.py` — модифицированная версия `chapter9_3.py`:

```
from flask import Flask, jsonify
app = Flask(__name__)

@app.route('/routers/<hostname>/interface/<int:interface_number>')
def interface(hostname, interface_number):
    return jsonify(name=hostname, interface=interface_number)

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

Всего несколько строк кода, и вот мы уже получаем объект JSON с соответствующим заголовком:

```
(venv) $ http http://192.168.2.123:5000/routers/r1/interface/1
HTTP/1.0 200 OK
Content-Length: 38
Content-Type: application/json
Date: Tue, 08 Oct 2019 21:48:51 GMT
Server: Werkzeug/0.16.0 Python/3.6.8

{
    "interface": 1,
    "name": "r1"
}
```

Теперь создадим API для нашей сети, используя все эти возможности Flask.

API для сетевых ресурсов

В промышленном окружении у каждого сетевого устройства есть состояние и информация, которые имеет смысл хранить на постоянной основе, чтобы при необходимости легко их извлекать. Для этого часто используют базу данных. Мы уже видели примеры сохранения информации в главах о мониторинге.

Для нас может быть нежелательно давать прямой доступ к базе данных пользователям, которые не являются сетевыми администраторами, даже если им нужна эта информация, или мы и не хотим заставлять их учить сложный язык запросов SQL. В таких случаях можно предоставлять необходимую информацию через сетевой API с помощью фреймворка Flask и его расширения *Flask-SQLAlchemy*.

Больше о Flask-SQLAlchemy читайте по адресу https://flask-sqlalchemy.palletsprojects.com/en/2.x/.

Flask-SQLAlchemy

SQLAlchemy и расширение Flask-SQLAlchemy — это механизм абстрагирования базы данных и, соответственно, объектно-реляционного отображения. Проще говоря, эти инструменты позволяют работать с базой данных как с объектом на языке Python. Чтобы не усложнять, используем в нашем примере БД SQLite, управляющую плоским файлом, которая ведет себя как полноценная реляционная база данных. В сценарии `chapter9_db_1.py` показан пример использования Flask-SQLAlchemy для создания сетевой БД и добавления в нее нескольких табличных записей. Это многошаговый процесс, и мы рассмотрим каждый шаг отдельно.

Для начала напишем приложение на основе Flask и загрузим конфигурацию для SQLAlchemy, такую как путь к БД и ее название, а затем создадим объект `SQLAlchemy`, передав в него экземпляр приложения:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

# Создаем Flask-приложение, загружаем конфигурацию и создаем объект
SQLAlchemy
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
db = SQLAlchemy(app)
```

Мы можем создать объект базы данных вместе с соответствующим первичным ключом и различными столбцами:

```
# Это объект модели БД
class Device(db.Model):
    __tablename__ = 'devices'
    id = db.Column(db.Integer, primary_key=True)
    hostname = db.Column(db.String(120), index=True)
    vendor = db.Column(db.String(40))

    def __init__(self, hostname, vendor):
        self.hostname = hostname
        self.vendor = vendor

    def __repr__(self):
        return '<Device %r>' % self.hostname
```

Мы можем вызывать объект `database`, создавать записи и вставлять их в таблицу в базе данных. Важно: все, что мы добавляем в сеанс (`session`), нужно зафиксировать в БД, иначе эта информация не сохранится:

```
if __name__ == '__main__':
    db.create_all()
    r1 = Device('lax-dc1-core1', 'Juniper')
    r2 = Device('sfo-dc1-core1', 'Cisco')
    db.session.add(r1)
    db.session.add(r2)
    db.session.commit()
```

Запустим этот сценарий и проверим наличие файла базы данных:

```
(venv) $ python chapter9_db_1.py
(venv) $ ls -l network.db
-rw-r--r-- 1 echou echou 3072 Oct 8 15:38 network.db
```

Для просмотра табличных записей воспользуемся интерактивной оболочкой:

```
>>> from flask import Flask
>>> from flask_sqlalchemy import SQLAlchemy
>>> app = Flask(__name__)
>>> app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
>>> db = SQLAlchemy(app)
>>> from chapter9_db_1 import Device
>>> Device.query.all()
[<Device 'lax-dc1-core1'>, <Device 'sfo-dc1-core1'>]
>>> Device.query.filter_by(hostname='sfo-dc1-core1')
<flask_sqlalchemy.BaseQuery object at 0x7f09544a0e80>
>>> Device.query.filter_by(hostname='sfo-dc1-core1').first()
<Device 'sfo-dc1-core1'>
```

Аналогично можно создавать новые записи:

```
>>> r3 = Device('lax-dc1-core2', 'Juniper')
>>> db.session.add(r3)
>>> db.session.commit()
>>> Device.query.filter_by(hostname='lax-dc1-core2').first()
<Device 'lax-dc1-core2'>
```

Удалим файл `network.db`, чтобы он не конфликтовал с другими нашими примерами, в которых используется то же название БД:

```
(venv) $ rm network.db
```

Перейдем к созданию API для работы с содержимым нашей сети.

API для работы с содержимым сети

Прежде чем приступать к построению нашего API, подумаем о его будущей структуре. Планирование здесь больше похоже на искусство, чем на науку; оно сильно зависит от конкретной ситуации и наших предпочтений. Мой подход вовсе не единственный, но на начальных этапах постарайтесь повторять за мной.

Как вы помните из нашей диаграммы, у нас есть четыре устройства Cisco IOSv. Представим, что два из них, `iosv-1` и `iosv-2`, — магистральные, а два других, `iosv-3` и `iosv-4`, играют роль сетевых сервисов. Эта конфигурация выбрана произвольно, и позже ее можно поменять, но смысл в том, что нам нужно предоставлять данные о сетевых устройствах через API.

Упростим пример — создадим два API: для группы устройств и для отдельного устройства (рис. 9.3).

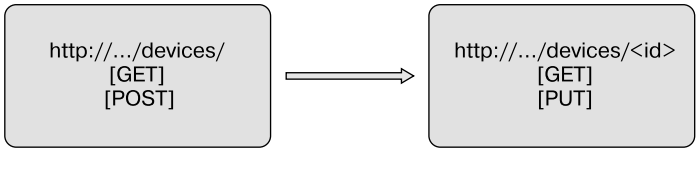


Рис. 9.3. API для работы с содержимым сети

Первый API будет иметь конечную точку `172.16.1.123/devices/` и поддерживать два метода: `GET` и `POST`. `GET`-запрос будет возвращать текущий список устройств, а `POST`-запрос с правильно сформированным телом в формате JSON — создавать новое устройство. Конечно, для создания и получения информации можно

предусмотреть разные конечные точки, но в этом примере операции будут различаться только методом HTTP.

Второй API относится к отдельному устройству и имеет вид `172.16.1.123/devices/<id устройства>`. `GET`-запрос будет выводить информацию об устройстве, введенную нами в базу данных, а `PUT`-запрос — обновлять имеющуюся запись. Обратите внимание на `PUT` вместо `POST`. Это типичный пример использования HTTP-методов; `PUT` применяется, когда нужно изменить существующие данные.

Чтобы вы лучше представили, как будет выглядеть API, я забегу немного вперед и покажу конечный результат. Если хотите выполнить этот пример вместе со мной, запустите `chapter9_6.py` в качестве Flask-сервера.

`POST`-запрос к API `/devices/` создает новую запись. В данном случае я хочу создать сетевое устройство с такими атрибутами, как имя хоста, локальный IP-адрес, управляющий IP-адрес, роль, поставщик и операционная система, под управлением которой оно работает:

```
(venv) $ http POST http://172.16.1.123:5000/devices/
'hostname'='iosv-1'
'loopback'='192.168.0.1'
'mgmt_ip'='172.16.1.225'
'role'='spine'
'vendor'='Cisco'
'os'='15.6'

HTTP/1.0 201 CREATED
Content-Length: 3
Content-Type: application/json
Date: Tue, 08 Oct 2019 23:15:31 GMT
Location: http://172.16.1.123:5000/devices/1
Server: Werkzeug/0.16.0 Python/3.6.8
{}
```

Повторяю предыдущий шаг, чтобы создать еще три устройства:

```
(venv) $ http POST http://172.16.1.123:5000/devices/
'hostname'='iosv-2'
'loopback'='192.168.0.2'
'mgmt_ip'='172.16.1.226'
'role'='spine'
'vendor'='Cisco'
'os'='15.6'

(venv) $ http POST http://172.16.1.123:5000/devices/
'hostname'='iosv-3',
'loopback'='192.168.0.3'
'mgmt_ip'='172.16.1.227'
'role'='leaf'
```

```
'vendor'='Cisco'
'os'='15.6'

(venv) $ http POST http://172.16.1.123:5000/devices/
'hostname'='iosv-4',
'loopback'='192.168.0.4'
'mgmt_ip'='172.16.1.228'
'role'='leaf'
'vendor'='Cisco'
'os'='15.6'
```

GET-запрос к той же конечной точке вернет список созданных нами сетевых устройств:

```
(venv) $ http GET http://172.16.1.123:5000/devices/
HTTP/1.0 200 OK
Content-Length: 192
Content-Type: application/json
Date: Tue, 08 Oct 2019 23:21:12 GMT
Server: Werkzeug/0.16.0 Python/3.6.8

{
    "device": [
        "http://172.16.1.123:5000/devices/1",
        "http://172.16.1.123:5000/devices/2",
        "http://172.16.1.123:5000/devices/3",
        "http://172.16.1.123:5000/devices/4"
    ]
}
```

Точно так же GET-запрос к /devices/<id> вернет информацию об отдельном устройстве:

```
(venv) echou@network-dev-2:~$ http GET http://172.16.1.123:5000/devices/1
<опущено>
{
    "hostname": "iosv-1",
    "loopback": "192.168.0.1",
    "mgmt_ip": "172.16.1.225",
    "os": "15.6",
    "role": "spine",
    "self_url": "http://172.16.1.123:5000/devices/1",
    "vendor": "Cisco"
}
```

Представьте, что мы решили понизить версию операционной системы на устройстве r1 с 15.6 до 14.6. Для обновления соответствующей записи используем PUT-запрос:

```
(venv) $ http PUT http://172.16.1.123:5000/devices/1
'hostname'='iosv-1'
'loopback'='192.168.0.1'
```

```
'mgmt_ip'='172.16.1.225'
'role'='spine'
'vendor'='Cisco'
'os'='14.6'
HTTP/1.0 200 OK
# Проверка
(venv) $ http GET http://172.16.1.123:5000/devices/1HTTP/1.0 200 OK
<опущено>

{
    "hostname": "iosv-1",
    "loopback": "192.168.0.1",
    "mgmt_ip": "172.16.1.225",
    "os": "14.6",
    "role": "spine",
    "self_url": "http://172.16.1.123:5000/devices/1",
    "vendor": "Cisco"
}
```

Теперь рассмотрим код сценария `chapter9_6.py`, который создает эти API. Мне нравится в этом примере, что все наши API, включая код для взаимодействия с БД, уместились в один файл. Позже, когда проект разрастется, мы вынесем отдельные компоненты, такие как класс базы данных, в отдельные файлы.

API для работы с устройствами

Файл `chapter9_6.py` начинается с импорта необходимых модулей. Обратите внимание: объект `request` импортируется из клиентской библиотеки Flask, а не из пакета `requests`, как в предыдущих главах:

```
from flask import Flask, url_for, jsonify, request
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
db = SQLAlchemy(app)
```

Мы объявили объект `database` с `id` в качестве первичного ключа и со строковыми полями `hostname`, `loopback`, `mgmt_ip`, `role`, `vendor` и `os`:

```
class Device(db.Model):
    __tablename__ = 'devices'
    id = db.Column(db.Integer, primary_key=True)
    hostname = db.Column(db.String(64), unique=True)
    loopback = db.Column(db.String(120), unique=True)
    mgmt_ip = db.Column(db.String(120), unique=True)
    role = db.Column(db.String(64))
    vendor = db.Column(db.String(64))
    os = db.Column(db.String(64))
```

Функция `get_url()` в классе `Device` возвращает URL из `url_for()`. Обратите внимание: функция `get_device()`, которая здесь вызывается, еще не определена для маршрута `/devices/<int:id>`:

```
def get_url(self):
    return url_for('get_device', id=self.id, _external=True)
```

Функции `export_data()` и `import_data()` решают противоположные задачи. Первая используется для предоставления пользователю информации из базы данных, когда тот посылает запрос `GET`, а вторая сохраняет в базе данных информацию, отправленную пользователем методом `POST` или `PUT`:

```
def export_data(self):
    return {
        'self_url': self.get_url(),
        'hostname': self.hostname,
        'loopback': self.loopback,
        'mgmt_ip': self.mgmt_ip,
        'role': self.role,
        'vendor': self.vendor,
        'os': self.os
    }

def import_data(self, data):
    try:
        self.hostname = data['hostname']
        self.loopback = data['loopback']
        self.mgmt_ip = data['mgmt_ip']
        self.role = data['role']
        self.vendor = data['vendor']
        self.os = data['os']
    except KeyError as e:
        raise ValidationError('Invalid device: missing ' + e.args[0])
    return self
```

Теперь у нас есть объект `database` и функции импорта и экспорта. Осталось реализовать маршрутизацию URL для работы с устройствами. В ответ на `GET`-запрос мы должны вернуть список со всеми записями из таблицы `devices` и URL для каждой из них. При получении метода `POST` будет вызвана функция `import_data()` с глобальным объектом `request` в аргументе; она добавляет устройство и записывает информацию о нем в базу данных:

```
@app.route('/devices/', methods=['GET'])
def get_devices():
    return jsonify({'device': [device.get_url()
                               for device in Device.query.all()]})

@app.route('/devices/', methods=['POST'])
def new_device():
    device = Device()
```

```
    device.import_data(request.json)
    db.session.add(device)
    db.session.commit()
    return jsonify({}), 201, {'Location': device.get_url()}
```

В ответ на запрос POST возвращается ответ с пустым документом JSON в теле и кодом состояния 201 (создано), а также дополнительные заголовки:

```
HTTP/1.0 201 CREATED
Content-Length: 2
Content-Type: application/json Date: ...
Location: http://172.16.1.173:5000/devices/4
Server: Werkzeug/0.9.6 Python/3.5.2
```

Рассмотрим API для получения информации об отдельных устройствах.

API для работы с отдельными устройствами

В маршруте для отдельных устройств указано, что идентификатор должен быть целочисленным; это может служить первой линией защиты от некорректных запросов. Здесь также используется одна конечная точка с двумя методами, в которых вызываются те же функции импорта и экспорта:

```
@app.route('/devices/<int:id>', methods=['GET'])
def get_device(id):
    return jsonify(Device.query.get_or_404(id).export_data())

@app.route('/devices/<int:id>', methods=['PUT'])
def edit_device(id):
    device = Device.query.get_or_404(id)
    device.import_data(request.json)
    db.session.add(device)
    db.session.commit()
    return jsonify({})
```

Стоит отметить, что метод `query_or_404()` служит удобным способом вернуть код 404 (не найдено) на случай, если в базе данных нет записей с указанным ID. Это довольно элегантный и компактный способ проверки запроса к БД.

Последний участок кода создает в базе данных таблицу и запускает сервер Flask для разработки:

```
if __name__ == '__main__':
    db.create_all()
    app.run(host='0.0.0.0', debug=True)
```

Это один из самых длинных сценариев на Python в книге, поэтому у нас ушло больше времени на его рассмотрение. Он позволяет проиллюстрировать работу

с базой данных на серверной стороне для отслеживания сетевых устройств, внешний доступ к которым осуществляется по API на основе Flask.

В следующем разделе вы узнаете, как использовать API для выполнения асинхронных задач как с отдельными устройствами, так и с группой устройств.

Динамические сетевые операции

Итак, наш API умеет предоставлять статическую информацию о сети; запрашивающей стороне можно вернуть все, что может быть сохранено в базе данных. Было бы здорово, если бы мы могли взаимодействовать с нашей сетью напрямую, запрашивать сведения об устройствах или вносить изменения в их конфигурацию.

Возьмем за основу сценарий для взаимодействия с устройством с помощью Pexpect, который мы уже видели в главе 2, и оформим его в виде функции, чтобы затем использовать в `chapter9_pexpect_1.py`:

```
import pexpect
def show_version(device, prompt, ip, username, password):
    device_prompt = prompt
    child = pexpect.spawn('telnet ' + ip)
    child.expect('Username:')
    child.sendline(username)
    child.expect('Password:')
    child.sendline(password)
    child.expect(device_prompt)
    child.sendline('show version | i V')
    child.expect(device_prompt)
    result = child.before
    child.sendline('exit')
    return device, result
```

Проверим новую функцию в интерактивной оболочке:

```
>>> from chapter9_pexpect_1 import show_version
>>> print(show_version('iosv-1', 'iosv-1#', '172.16.1.225', 'cisco',
'cisco'))
('iosv-1', b'show version | i V\r\nCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\r\
nProcessor board ID 9Z1DS4YEJWHZGVUM73HWA\r\n')
```

Убедитесь, что ваш сценарий на основе Pexpect работает. В следующем примере предполагается, что вы ввели в базу данных всю необходимую информацию из предыдущего раздела.

Добавим в chapter9_7.py новый API для получения версии устройства:

```
from chapter9_pexpect_1 import show_version
<опущено>
@app.route('/devices/<int:id>/version', methods=['GET'])
def get_device_version(id):
    device = Device.query.get_or_404(id)
    hostname = device.hostname
    ip = device.mgmt_ip
    prompt = hostname+"#"
    result = show_version(hostname, prompt, ip, 'cisco', 'cisco')
    return jsonify({"version": str(result)})
```

Запрашивающая сторона получит следующий результат:

```
(venv) $ http GET http://172.16.1.123:5000/devices/1/version
HTTP/1.0 200 OK
Content-Length: 212
Content-Type: application/json
Date: Tue, 08 Oct 2019 23:53:49 GMT
Server: Werkzeug/0.16.0 Python/3.6.8

{
    "version": "('iosv-1', b'show version | i V\\r\\nCisco IOS Software,
IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE
SOFTWARE (fc2)\\r\\nProcessor board ID 9Z1DS4YEJWHZGVUM73HWA\\r\\n')"
}
```

Теперь добавим еще одну конечную точку для выполнения пакетных операций с несколькими устройствами на основе их общих полей. В следующем примере конечная точка извлекает из URL атрибут device_role и отыскивает соответствующее устройство (или устройства):

```
@app.route('/devices/<device_role>/version', methods=['GET'])
def get_role_version(device_role):
    device_id_list = [device.id for device in Device.query.all()
                                 if device.role == device_role]
    result = {}
    for id in device_id_list:
        device = Device.query.get_or_404(id)
        hostname = device.hostname
        ip = device.mgmt_ip
        prompt = hostname + "#"
        device_result = show_version(hostname, prompt, ip, 'cisco', 'cisco')
        result[hostname] = str(device_result)
    return jsonify(result)
```

Конечно, перебирать все устройства в Device.query.all(), как в предыдущем примере, будет неэффективно. В промышленных условиях мы могли бы воспользоваться SQL-запросом, который отыщет устройства с заданной ролью.

С помощью RESTful API можно получить список сразу всех магистральных и периферийных устройств:

```
(venv) $ http GET http://172.16.1.123:5000/devices/spine/version
HTTP/1.0 200 OK
<опущено>
{
    "iosv-1": "('iosv-1', b'show version | i V\\r\\nCisco IOS Software,
IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE
SOFTWARE (fc2)\\r\\nProcessor board ID 9Z1DS4YEJWHZGVUM73HWA\\r\\n')",
    "iosv-2": "('iosv-2', b'show version | i V\\r\\nCisco IOS Software,
IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE
SOFTWARE (fc2)\\r\\nProcessor board ID 9BPSF4VEH068CWL8YVZGT\\r\\n')"
}
```

Как проиллюстрировано выше, конечные точки API обращаются к устройству (-ам) в режиме реального времени и возвращают результат запрашивающей стороне. Однако такой подход годится, только если операция всегда успевает ответить в пределах тайм-аута (по умолчанию 30 секунд) или если вы допускаете, что HTTP-сеанс может завершиться до конца операции. Чтобы решить проблему тайм-аута, задачи можно выполнять асинхронно. В следующем подразделе вы увидите, как это делается.

Асинхронные операции

По моему мнению, асинхронные операции Flask, когда задачи выполняются не в хронологическом порядке, — сложная тема. У Мигеля Гринберга (https://blog.miguelgrinberg.com/), которого я очень уважаю, на эту тему есть много статей в блоге и примеров в GitHub. В демонстрационном сценарии `chapter9_8.py` используется декоратор `background` на основе кода Мигеля (https://github.com/miguelgrinberg/oreilly-flask-apis-video/blob/master/camera/camera.py) для Raspberry Pi. Для начала импортируем несколько дополнительных модулей:

```
from flask import Flask, url_for, jsonify, request,\
    make_response, copy_current_request_context
from flask_sqlalchemy import SQLAlchemy
from chapter9_pexpect_1 import show_version
import uuid
import functools
from threading import Thread
```

Декоратор `background` принимает функцию и выполняет ее в фоновом режиме, используя поток и UUID в качестве идентификатора задачи. Декоратор воз-

вращает код 202 (принято) и путь к новым ресурсам, по которому должна обратиться запрашивающая сторона. Создадим новый URL для проверки состояния выполняющейся задачи:

```
@app.route('/status/<id>', methods=['GET'])
def get_task_status(id):
    global background_tasks
    rv = background_tasks.get(id)
    if rv is None:
        return not_found(None)
    if isinstance(rv, Thread):
        return jsonify({}), 202, {'Location': url_for('get_task_status',
               id=id)}
    if app.config['AUTO_DELETE_BG_TASKS']:
        del background_tasks[id]
    return rv
```

После извлечения ресурс удаляется. Для этого в верхней части приложения параметру `app.config['AUTO_DELETE_BG_TASKS']` присваивается `True`. Добавим этот декоратор в конечную точку `version`; в нем скрыта вся сложная логика, поэтому остальной код менять не нужно (здорово, правда?):

```
@app.route('/devices/<int:id>/version', methods=['GET'])
@background
def get_device_version(id):
    device = Device.query.get_or_404(id)
<опущено>
@app.route('/devices/<device_role>/version', methods=['GET'])
@background
def get_role_version(device_role):
    device_id_list = [device.id for device in Device.query.all() if
device.role == device_role]
<опущено>
```

Получение результата происходит в два этапа. Сначала выполняется `GET`-запрос к конечной точке, который возвращает ответ с заголовком `Location`:

```
(venv) $ http GET http://172.16.1.123:5000/devices/spine/version
HTTP/1.0 202 ACCEPTED
Content-Length: 3
Content-Type: application/json
Date: Tue, 08 Oct 2019 23:58:57 GMT
Location: http://172.16.1.123:5000/status/057c895371b448d2aad30525c31e
1c51
Server: Werkzeug/0.16.0 Python/3.6.8
{}
```

Затем, чтобы получить сам результат, выполняется второй запрос по указанному адресу:

```
(venv) $ http GET http://172.16.1.123:5000/status/057c895371b448d2aad3052
5c31e1c51
HTTP/1.0 200 OK
<опущено>
{
    "iosv-1": "('iosv-1', b'show version | i V\\r\\nCisco IOS Software,
IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE
SOFTWARE (fc2)\\r\\nProcessor board ID 9Z1DS4YEJWHZGVUM73HWA\\r\\n')",
    "iosv-2": "('iosv-2', b'show version | i V\\r\\nCisco IOS Software,
IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE
SOFTWARE (fc2)\\r\\nProcessor board ID 9BPSF4VEH068CWL8YVZGT\\r\\n')"
}
```

Убедимся, что в случае неготовности ресурса возвращается код состояния 202: используем для этого сценарий `chapter9_request_1.py`, который сразу выполняет запрос к новому ресурсу:

```
import requests, time

server = 'http://172.16.1.123:5000'
endpoint = '/devices/1/version'

# Первый запрос для получения нового ресурса
r = requests.get(server+endpoint)
resource = r.headers['location']
print("Status: {} Resource: {}".format(r.status_code, resource))

# Второй запрос для получения состояния ресурса
r = requests.get(resource)
print("Immediate Status Query to Resource: " + str(r.status_code))

print("Sleep for 2 seconds")
time.sleep(2)
# Третий запрос для получения состояния ресурса
r = requests.get(resource)
print("Status after 2 seconds: " + str(r.status_code))
```

Как видите, пока ресурс продолжает выполняться в фоне, в ответ на запрос возвращается код 202:

```
(venv) $ python chapter9_request_1.py
Status: 202 Resource: http://172.16.1.123:5000/status/6108048c6e9b40fbab5
a5b53c5817e7c
Immediate Status Query to Resource: 202
Sleep for 2 seconds
Status after 2 seconds: 200
```

Процесс создания API идет гладко! Сетевые ресурсы — это наша ценность, поэтому доступ к ним должен иметь только авторизованный персонал. В следующем разделе мы добавим в наш API элементарные механизмы безопасности.

Аутентификация и авторизация

Реализуем базовую аутентификацию пользователей с помощью модуля `httpauth` из фреймворка Flask, написанного Мигелем Гринбергом; также применим библиотеку Werkzeug для работы с паролями. В начале главы мы установили модуль `httpauth` в числе других перечисленных в файле `requirements.txt`. Новый сценарий, иллюстрирующий наши меры безопасности, называется `chapter9_9.py`. Он начинается с импорта дополнительных модулей:

```
from werkzeug.security import generate_password_hash, check_password_hash
from flask_httpauth import HTTPBasicAuth
```

Создадим объект `HTTPBasicAuth` и объект `User` для работы с базой данных. Обратите внимание: в процессе создания объекту передается значение пароля, но мы будем сохранять только его хеш, `password_hash`:

```
auth = HTTPBasicAuth()
<опущено>
class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True)
    password_hash = db.Column(db.String(128))

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

У объекта `auth` есть декоратор `verify_password`, который можно использовать вместе с глобальным контекстом Flask `g`, созданным при получении пользовательского запроса. Если сохранить в этот контекст запись о пользователе, она будет доступна на протяжении всей транзакции:

```
@auth.verify_password
def verify_password(username, password):
    g.user = User.query.filter_by(username=username).first()
    if g.user is None:
        return False
    return g.user.verify_password(password)
```

Это удобный обработчик `before_request`, который можно использовать перед вызовом любой конечной точки API. Применим его и декоратор `auth.login_required` ко всем нашим маршрутам:

```
@app.before_request
@auth.login_required
def before_request():
    pass
```

И наконец, воспользуемся обработчиком ошибок `unauthorized`, чтобы возвращать объект `response` с кодом состояния 401 (пользователь не авторизован):

```
@auth.error_handler
def unathorized():
    response = jsonify({'status': 401, 'error': 'unauthorized',
                        'message': 'please authenticate'})
    response.status_code = 401
    return response
```

Создадим пользователей в нашей базе данных — тогда мы сможем проверить аутентификацию:

```
>>> from chapter9_9 import db, User
>>> db.create_all()
>>> u = User(username='eric')
>>> u.set_password('secret')
>>> db.session.add(u)
>>> db.session.commit()
>>> exit()
```

После запуска своего сервера Flask для разработки попробуйте к нему обратиться, как мы это делали ранее. На этот раз вы увидите, что сервер отклонил ваш запрос и вернул ошибку 401 (пользователь не авторизован):

```
(venv) $ http GET http://172.16.1.123:5000/devices/
HTTP/1.0 401 UNAUTHORIZED
<опущено>
WWW-Authenticate: Basic realm="Authentication Required"

{
    "error": "unauthorized",
    "message": "please authenticate",
    "status": 401
}
```

Теперь, чтобы подключиться к серверу, запрос должен содержать заголовок аутентификации:

```
(venv) $ http --auth eric:secret GET http://172.16.1.123:5000/devices/
HTTP/1.0 200 OK
Content-Length: 192
Content-Type: application/json
Date: Wed, 09 Oct 2019 00:31:41 GMT
Server: Werkzeug/0.16.0 Python/3.6.8
```

```
{
    "device": [
        "http://172.16.1.123:5000/devices/1",
        "http://172.16.1.123:5000/devices/2",
        "http://172.16.1.123:5000/devices/3",
        "http://172.16.1.123:5000/devices/4"
    ]
}
```

У нас получился приличный RESTful API для нашей сети. Когда пользователю понадобится информация о сетевом устройстве, он сможет запросить у сети статическое содержимое. Он также может выполнять сетевые операции с отдельными или несколькими устройствами сразу. Мы предусмотрели и элементарные меры безопасности, чтобы данные из нашего API могли извлекать только созданные нами пользователи. И все это уместилось в одном файле длиной 250 строк (200, если не считать комментарии)!

Если вас интересуют подробности управления пользовательскими сеансами, их запоминание, а также вход и выход из системы, я рекомендую расширение Flask-Login (https://flask-login.readthedocs.io/en/latest/).

Мы инкапсулировали внутренний API поставщика в нашей сети, заменив его собственным RESTful API. Благодаря этому мы можем использовать на серверной стороне все, что нам нужно, включая Pexpect, а запрашивающая сторона при этом получает унифицированный клиентский интерфейс. Мы можем пойти еще дальше и заменить исходное сетевое устройство, не оказывая никакого влияния на пользователей, которые шлют нам свои запросы. Flask позволяет реализовать этот уровень абстракции просто и компактно. К тому же Flask требует меньше ресурсов, если использовать контейнеры.

Выполнение Flask в контейнерах

Контейнеры обрели популярность в последние несколько лет. Они дают дополнительные уровни абстракции и виртуализации по сравнению с виртуальными машинами на основе гипервизора. Обсуждение контейнеров выходит за рамки этой книги. Но для тех, кому интересно, мы предлагаем ознакомиться с простым примером: как приложение на основе Flask можно запустить в контейнере Doocker.

Наш пример основан на практическом руководстве по Docker от DigitalOcean, посвященном созданию контейнеров на компьютерах под управлением

Ubuntu 18.04 (https://www.digitalocean.com/community/tutorials/how-to-build-and-deploy-a-flask-application-using-docker-on-ubuntu-18-04). Если вы плохо знакомы с контейнерами, я рекомендую вам ознакомиться с этим руководством и затем вернуться к данному разделу.

Убедимся, что у нас установлен пакет Docker:

```
$ sudo docker --version
Docker version 19.03.2, build 6a30dfc
```

Создадим каталог с именем `TestApp`, в котором будет храниться наш код:

```
$ mkdir TestApp
$ cd TestApp/
```

В нем мы создадим еще один каталог с именем `app` и поместим в него файл `__init__.py`:

```
$ mkdir app
$ touch app/__init__.py
```

Логика нашего приложения будет находиться в каталоге `app`. Поскольку до сих пор мы хранили весь код в одном файле, скопируем содержимое `chapter9_6.py` в `app/__init__.py`:

```
$ cat app/__init__.py
from flask import Flask, url_for, jsonify, request
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
db = SQLAlchemy(app)

@app.route('/')
def home():
    return "Hello Python Netowrking!"
<опущено>
class Device(db.Model):
    __tablename__ = 'devices'
    id = db.Column(db.Integer, primary_key=True)
    hostname = db.Column(db.String(64), unique=True)
    loopback = db.Column(db.String(120), unique=True)
    mgmt_ip = db.Column(db.String(120), unique=True)
    role = db.Column(db.String(64))
    vendor = db.Column(db.String(64))
    os = db.Column(db.String(64))
<опущено>
```

В этот же каталог можно скопировать созданный нами файл базы данных SQLite:

```
$ tree app/
app/
├── __init__.py
├── network.db
```

Поместим файл `requirements.txt` в каталог `TestApp` и создадим сценарий `main.py`, который будет служить точкой входа. Также добавим INI-файл для `uwsgi`:

```
$ cat main.py
from app import app

$ cat uwsgi.ini
[uwsgi]
module = main
callable = app
master = true
```

Используем в качестве основы готовый образ Docker и создадим файл `Dockerfile`, чтобы собрать свой образ:

```
$ cat Dockerfile
FROM tiangolo/uwsgi-nginx-flask:python3.7-alpine3.7
RUN apk --update add bash vim
RUN mkdir /TestApp
ENV STATIC_URL /static
ENV STATIC_PATH /TestApp/static
COPY ./requirements.txt /TestApp/requirements.txt
RUN pip install -r /TestApp/requirements.txt
```

Наш сценарий командной оболочки `start.sh` соберет образ, запустит его в фоновом режиме и перенаправит порт 8000 в контейнер Docker:

```
$ cat start.sh
#!/bin/bash
app="docker.test"
docker build -t ${app} .
docker run -d -p 8000:80 \
  --name=${app} \
  -v $PWD:/app ${app}
```

Теперь можно использовать сценарий `start.sh` для сборки образа и запуска нашего контейнера:

```
$ sudo bash start.sh
Sending build context to Docker daemon  49.15kB
Step 1/7 : FROM tiangolo/uwsgi-nginx-flask:python3.7-alpine3.7
python3.7-alpine3.7: Pulling from tiangolo/uwsgi-nginx-flask
48ecbb6b270e: Pulling fs layer
692f29ee68fa: Pulling fs layer
<опущено>
```

Теперь наше приложение на основе Flask выполняется в контейнере, который доступен на нашем хосте на порте 8000:

```
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                              NAMES
ac5384e6b007        docker.test         "/entrypoint.sh /sta…"   55
minutes ago     Up 46 minutes       443/tcp, 0.0.0.0:8000->80/tcp
docker.test
```

Вот адресная строка с *IP-адресом управляющего хоста* (рис. 9.4).

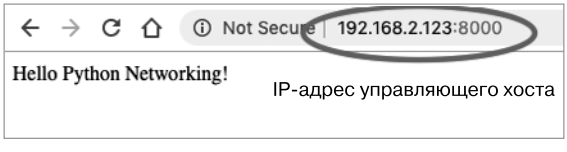


Рис. 9.4. IP-адрес управляющего хоста

Конечная точка API Flask выглядит так (рис. 9.5).

Рис. 9.5. Конечная точка API Flask

Мы можем остановить и удалить контейнер с помощью следующих команд:

```
$ sudo docker stop <container id>
$ sudo docker rm <containter id>
```

Мы также можем удалить образ Docker:

```
$ sudo docker images -a -q #find the image id
$ sudo docker rmi <image id>
```

Выполнение Flask в контейнере дает нам еще больше гибкости и возможность развертывать свои API в промышленном окружении. Конечно, контейнеры создают дополнительные сложности и прибавляют работы системным администраторам, поэтому при выборе методов развертывания следует взвесить все преимущества и издержки. Мы подошли к концу главы. Вспомним, что мы с вами успели сделать.

Резюме

Главу мы начали с того, что стали на путь построения RESTful API для нашей сети. Мы рассмотрели популярные веб-фреймворки на языке Python — Django и Flask — и сравнили их между собой. Выбор Flask позволил нам начать с малого и добавлять новые возможности за счет расширений этого фреймворка.

Мы использовали в нашей лаборатории виртуальное окружение, чтобы отделить установленный экземпляр Flask от глобальной коллекции пакетов Python. Наша сеть состояла из четырех узлов: двух магистральных маршрутизаторов и двух периферийных. Мы сделали обзор основных возможностей Flask и проверили конфигурацию нашего API с помощью простого клиента HTTPie.

Из множества возможностей Flask были особо выделены маршрутизация URL и переменные в маршрутах, так как они составляют базовую логику взаимодействий между запрашивающей стороной и нашим API. Вы увидели, как Flask-SQLAlchemy и SQLite можно использовать для хранения и возврата информации о статических элементах сети. Мы также создали конечные точки для административных задач, выполнение которых основано на обращении к другим программам, например Pexpect. Для совершенствования нашей конфигурации мы добавили в API асинхронную обработку и аутентификацию. В конце было показано, как запустить API на основе Flask в контейнере Docker.

В главе 10 мы сменим тему и рассмотрим облачные сетевые технологии на примере *Amazon Web Services (AWS)*.

10

Облачные сетевые технологии AWS

Облачные вычисления — одно из ведущих направлений в современном компьютерном мире, и эта тенденция сохраняется на протяжении многих лет. Публичные облачные провайдеры преобразили индустрию стартапов и изменили понимание того, что требуется для запуска нового сервиса с нуля. Нам больше не нужно строить собственную инфраструктуру; мы можем арендовать у публичного облачного провайдера часть его ресурсов для своих потребностей. В наши дни среди участников любой технологической конференции или встречи сложно увидеть человека без знаний или опыта использования/создания облачных сервисов. Облачные вычисления — это всерьез и надолго, и нам остается только свыкнуться с этим.

Существует несколько моделей облачных услуг, которые в целом можно разделить на *SaaS (Software-as-a-Service — программное обеспечение как услуга*; https://ru.wikipedia.org/wiki/Программное_обеспечение_как_услуга), *PaaS (Platform-as-a-Service — платформа как услуга*; https://ru.wikipedia.org/wiki/Платформа_как_услуга) и *IaaS (Infrastructure-as-a-Service — инфраструктура как услуга*; https://ru.wikipedia.org/wiki/Инфраструктура_как_услуга). Каждая предлагает отдельный уровень абстракции с точки зрения пользователя. В нашем случае сетевые технологии — это часть IaaS, и обсуждение именно этой модели будет главной темой данной главы.

Amazon Web Services (*AWS* — https://aws.amazon.com/) — первая компания, которая предложила публичную услугу IaaS, и по состоянию на 2019 год она

явный лидер в этой области, если смотреть на ее рыночную долю. Если под *программно-определяемыми сетями (Software-Defined Networking, SDN)* подразумевается группа программных сервисов, которые работают совместно для реализации таких сетевых концепций, как IP-адреса, списки доступа, балансировщики нагрузки, *NAT (Network Address Translation — преобразование сетевых адресов)*, можно утверждать, что AWS — крупнейший в мире провайдер SDN. Эта компания использует масштаб своей глобальной сети, а также свои дата-центры и серверы для предоставления потрясающего объема сетевых услуг.

Если вы хотите узнать больше о масштабе и сетевых технологиях Amazon, я настоятельно рекомендую посмотреть выступление Джеймса Гамильтона (James Hamilton) на конференции AWS re:Invent 2014: https://www.youtube.com/watch?v=JIQETrFC_SQ.

В этой главе мы обсудим сетевые услуги, предоставляемые облаком AWS, и особенности работы с ними из Python, такие как:

- подготовка к работе с AWS и краткий обзор сетевых технологий;
- виртуальное частное облако;
- Direct Connect и VPN;
- сервисы для масштабирования сетевых технологий;
- другие сетевые услуги AWS.

Приготовим все необходимое.

Подготовка к работе с AWS

Если у вас еще нет учетной записи AWS и вы хотите опробовать примеры из этой главы, то пройдите по адресу https://aws.amazon.com/ и зарегистрируйтесь. Регистрация проста и понятна; вам понадобится банковская карта и какой-то метод идентификации личности (например, мобильный телефон с возможностью принимать СМС).

Для начинающих AWS предлагает ряд бесплатных услуг (https://aws.amazon.com/free/), которыми можно пользоваться ограниченно. Например, в этой главе мы будем работать с Elastic Compute Cloud (EC2); бесплатный пакет EC2 включает 750 часов использования инстанса (экземпляра) t2.micro в месяц в первый год.

Я советую всегда начинать с бесплатной версии и затем при необходимости постепенно переходить на другие тарифные планы. Актуальные предложения смотрите на сайте AWS (рис. 10.1).

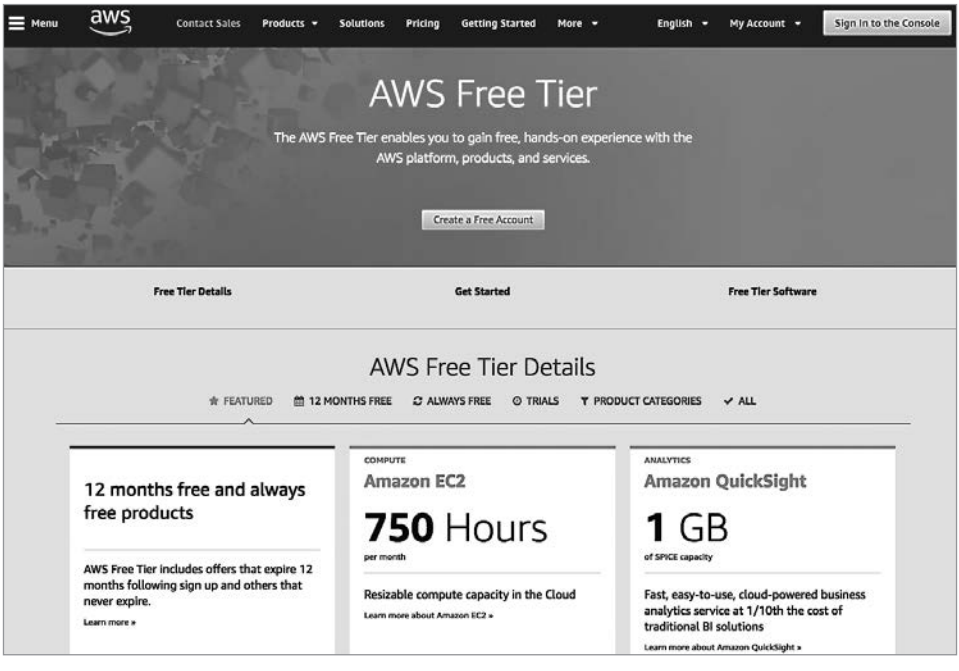


Рис. 10.1. Бесплатные услуги AWS

После регистрации вы можете войти в консоль управления AWS (https://console.aws.amazon.com/) и ознакомиться с различными сервисами, которые предлагает эта компания.

В консоли можно настроить все имеющиеся сервисы и просматривать ежемесячные счета (рис. 10.2).

Имея учетную запись, можно использовать утилиту AWS CLI и Python SDK для управления своими облачными ресурсами.

AWS CLI и Python SDK

Помимо консоли, для управления сервисами AWS можно использовать *интерфейс командной строки (Command Line Interface, CLI)* и различные SDK. AWS

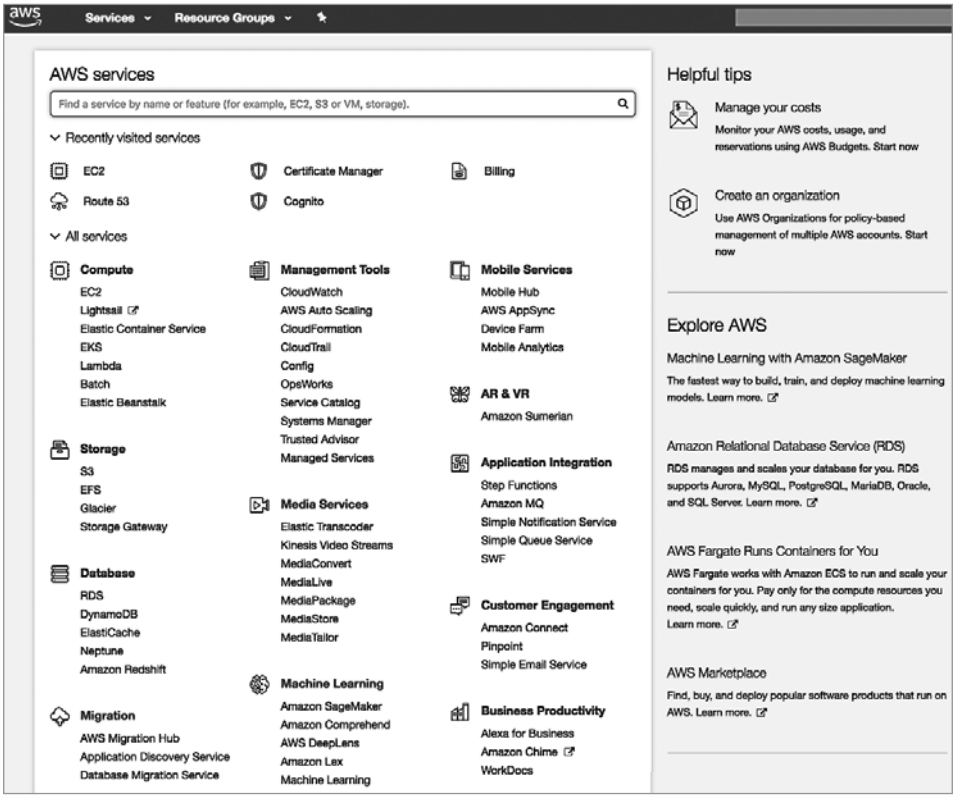


Рис. 10.2. Консоль AWS

CLI — это пакет на языке Python, который можно установить с помощью `pip` (https://docs.aws.amazon.com/cli/latest/userguide/installing.html). Давайте установим его на наш хост с Ubuntu:

```
(venv) $ pip install awscli
(venv) $ aws --version
aws-cli/1.16.259 Python/3.6.8 Linux/5.0.0-27-generic botocore/1.12.249
```

Чтобы упростить и обезопасить доступ, создадим нового пользователя и укажем его учетные данные для AWS CLI. Вернемся в консоль AWS и перейдем в раздел Identity and Access Management (IAM) (Управление учетными данными и доступом) (рис. 10.3).

Выберите Users (Пользователи) на левой панели, чтобы создать нового пользователя (рис. 10.4).

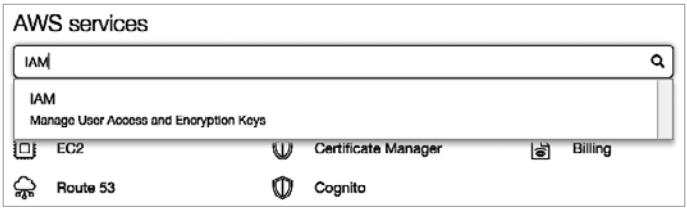


Рис. 10.3. AWS IAM

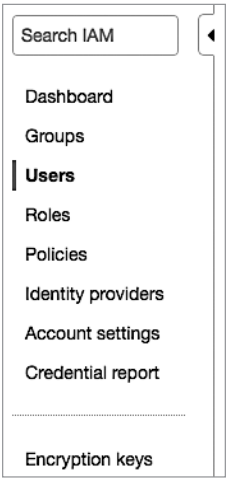


Рис. 10.4. Пользователи в AWS IAM

Выберите пункт Programmatic access (Программный доступ) и поместите пользователя в группу администраторов по умолчанию (рис. 10.5).

В результате мы получим Access key ID (ID ключа доступа) и Secret access key (Секретный ключ доступа). Скопируйте их в текстовый файл и сохраните в безопасном месте (рис. 10.6).

Закончим настройку аутентификации для AWS CLI в терминале с помощью команды `aws configure`. В следующем разделе мы рассмотрим разные регионы AWS, а пока остановимся на US-East-1, так как в этом регионе доступно больше всего сервисов. Позднее вы сможете вернуться к настройкам и выбрать другой регион:

```
$ aws configure
AWS Access Key ID [None]: <key>
AWS Secret Access Key [None]: <secret>
Default region name [None]: us-east-1
Default output format [None]: json
```

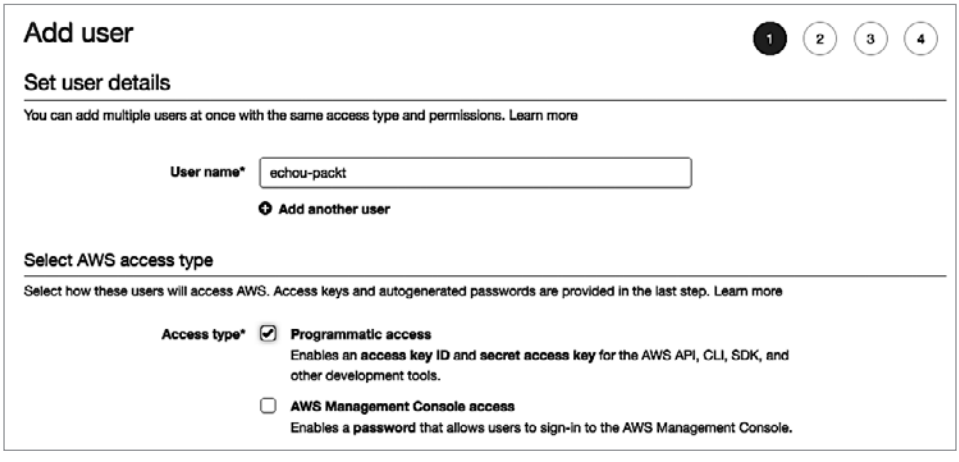


Рис. 10.5. Добавление пользователя в AWS IAM

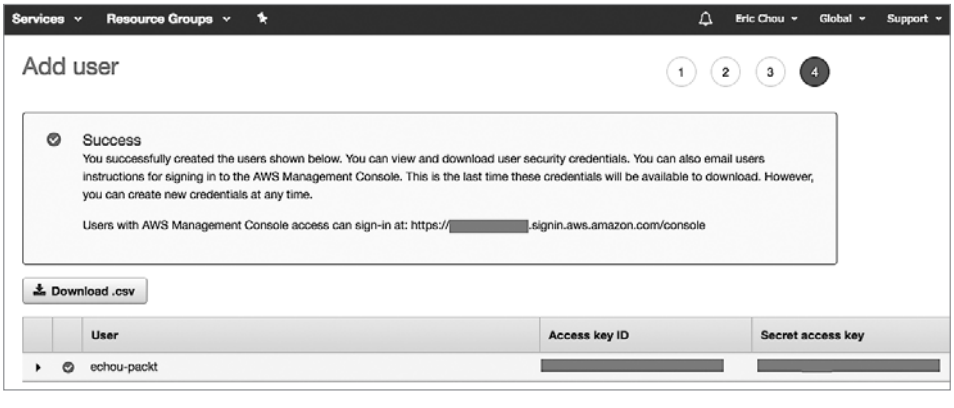


Рис. 10.6. Учетные данные для безопасного доступа в AWS IAM

Также установим AWS Python SDK — Boto3 (https://boto3.readthedocs.io/en/latest/):

```
(venv) $ pip install boto3
# проверка
(venv) $ python
Python 3.6.8 (default, Oct 7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto3
>>> exit()
```

Теперь мы готовы двинуться дальше и начнем с введения в сетевые сервисы облака AWS.

Обзор сети AWS

Начнем обсуждение сервисов AWS с самого высокого уровня — регионов и *зон доступности (availability zones, AZ)*. Они имеют большое значение для всех сервисов. На момент написания этой книги в AWS насчитывалось 22 географических региона и 69 *зон доступности (AZ)* по всему миру. Вот что говорится на странице AWS Global Cloud Infrastructure (https://aws.amazon.com/about-aws/global-infrastructure/):

> *«Облачная инфраструктура AWS построена вокруг регионов и зон доступности. Каждый регион предоставляет несколько физически разделенных и изолированных зон доступности, соединенных каналами связи с низкой задержкой, высокой пропускной способностью и большим количеством резервных ресурсов».*

На странице https://www.infrastructure.aws есть красивая визуальная карта сети AWS с возможностью фильтрации по региону и зоне доступности.

Некоторые услуги, которые предлагает AWS, — глобальные (такие как создание пользователей в IAM), но большинство доступно только в определенных регионах: US-East, US-West, EU-London, Asia-Pacific-Tokyo и т. д. В нашем контексте это означает, что инфраструктуру следует размещать как можно ближе к нашим потенциальным пользователям. Это уменьшит задержки сервисов для наших клиентов. Если целевая аудитория нашего сервиса находится на *Восточном побережье* Соединенных Штатов, то в качестве региона следует выбрать *US East (N. Virginia)* или *US East (Ohio)* (рис. 10.7).

Помимо задержек, которые будут наблюдать пользователи, регионы AWS имеют разные цены и доступные сервисы. Пользователи, которые только знакомятся с этим облаком, могут удивиться отсутствию некоторых сервисов в тех или иных регионах. Сервисы из этой главы доступны в большинстве регионов, но отдельные новые продукты — только в некоторых из них.

В следующем примере вы увидите, что Alexa for Business и Amazon Chime доступны только в регионе Северная Вирджиния, США (рис. 10.8). Помимо набора сервисов, в регионах могут быть и разные цены. Например, в сервисе EC2, который мы рассмотрим в этой главе, цена аренды инстанса *a1.medium* в *US East (N. Virginia)* составляет $0,0255 в час, а в *EU (Frankfurt)* — $0,0291 в час, то есть на 14 % выше (рис. 10.9, 10.10).

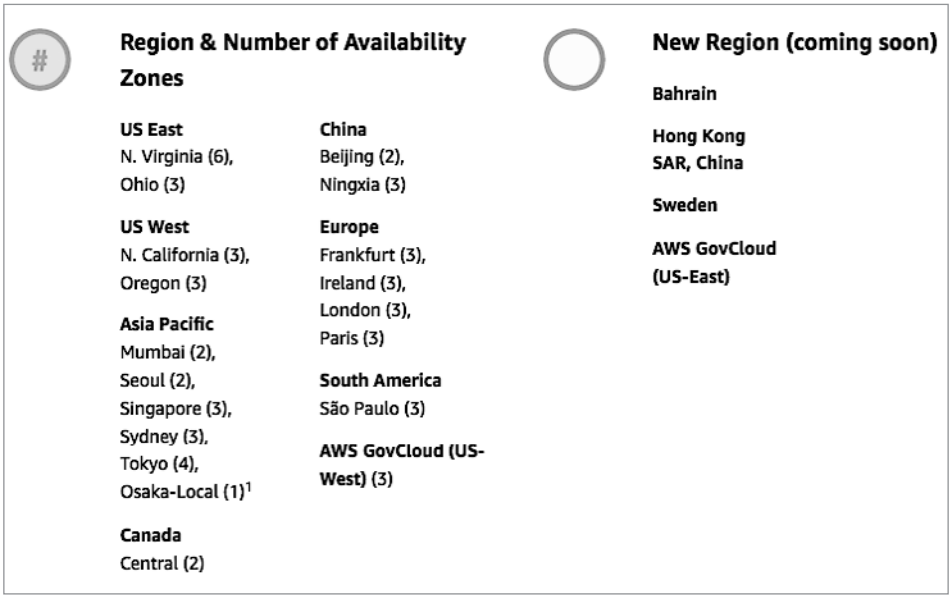


Рис. 10.7. Регионы AWS

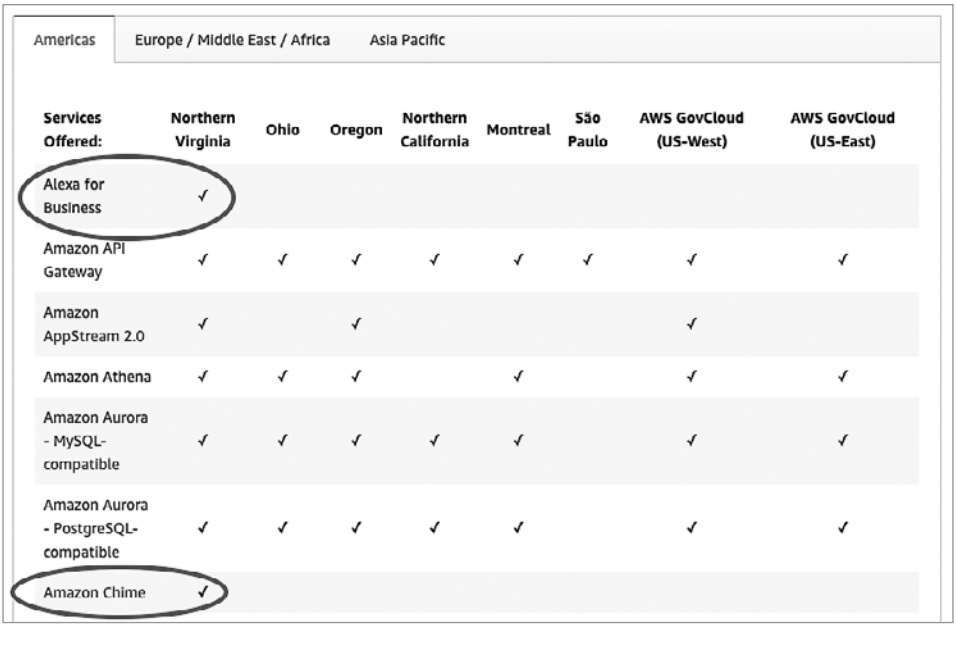


Рис. 10.8. Перечень сервисов AWS по регионам

Linux | RHEL | SLES | Windows | Windows with SQL Standard | Windows with SQL Web

Windows with SQL Enterprise | Linux with SQL Standard | Linux with SQL Web | Linux with SQL Enterprise

Region: US East (N. Virginia)

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
a1.medium	1	N/A	2 GiB	EBS Only	$0.0255 per Hour
a1.large	2	N/A	4 GiB	EBS Only	$0.051 per Hour
a1.xlarge	4	N/A	8 GiB	EBS Only	$0.102 per Hour
a1.2xlarge	8	N/A	16 GiB	EBS Only	$0.204 per Hour
a1.4xlarge	16	N/A	32 GiB	EBS Only	$0.408 per Hour
a1.metal	16	N/A	32 GiB	EBS Only	$0.408 per Hour
t3.nano	2	Variable	0.5 GiB	EBS Only	$0.0052 per Hour
t3.micro	2	Variable	1 GiB	EBS Only	$0.0104 per Hour

Рис. 10.9. Цена на AWS EC2 в регионе US East

Linux | RHEL | SLES | Windows | Windows with SQL Standard | Windows with SQL Web

Windows with SQL Enterprise | Linux with SQL Standard | Linux with SQL Web | Linux with SQL Enterprise

Region: EU (Frankfurt)

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
a1.medium	1	N/A	2 GiB	EBS Only	$0.0291 per Hour
a1.large	2	N/A	4 GiB	EBS Only	$0.0582 per Hour
a1.xlarge	4	N/A	8 GiB	EBS Only	$0.1164 per Hour
a1.2xlarge	8	N/A	16 GiB	EBS Only	$0.2328 per Hour
a1.4xlarge	16	N/A	32 GiB	EBS Only	$0.4656 per Hour
a1.metal	16	N/A	32 GiB	EBS Only	$0.466 per Hour
t3.nano	2	Variable	0.5 GiB	EBS Only	$0.006 per Hour
t3.micro	2	Variable	1 GiB	EBS Only	$0.012 per Hour
t3.small	2	Variable	2 GiB	EBS Only	$0.024 per Hour

Рис. 10.10. Цена на AWS EC2 в регионе EU

Если сомневаетесь, выбирайте US East (N. Virginia); это самый старый и, вероятно, дешевый регион с самым богатым набором сервисов.

Некоторые регионы доступны не всем пользователям. Например, регионы *GovCloud* и *China* по умолчанию недоступны пользователям из США. Чтобы узнать, какие регионы доступны вам, используйте команду `aws ec2 describe-regions`:

```
$ aws ec2 describe-regions
{
    "Regions": [
        {
            "Endpoint": "ec2.eu-north-1.amazonaws.com",
            "RegionName": "eu-north-1",
            "OptInStatus": "opt-in-not-required"
        },
        {
            "Endpoint": "ec2.ap-south-1.amazonaws.com",
            "RegionName": "ap-south-1",
            "OptInStatus": "opt-in-not-required"
        },
<опущено>
```

Как утверждает компания Amazon, все регионы независимы друг от друга и большинство ресурсов не копируется между ними. Это означает, что при использовании одного и того же сервиса в нескольких регионах, таких как *US-East* и *US-West*, нам придется самим позаботиться о копировании необходимых ресурсов между ними.

Нужный регион можно выбрать в консоли AWS в раскрывающемся меню в правом верхнем углу (рис. 10.11).

В консоли видны только сервисы, доступные в текущем регионе. Например, инстансы EC2, арендованные в регионе US East, будут недоступны в регионе US West. Я сам несколько раз ошибался подобным образом и не мог понять, куда делись мои инстансы!

Число после названия региона на рис. 10.7 обозначает количество зон доступности (AZ) в этом регионе. Название зоны состоит из региона и буквы, например: `us-east-1a`, `us-east-1b` и т. д. Каждый регион имеет несколько зон (обычно три). Каждая зона имеет свою изолированную инфраструктуру с резервным источником питания, внутренней сетью и оборудованием. Все зоны в регионе соединены оптоволоконными кабелями с низкой задержкой и обычно находятся на расстоянии не более 100 километров друг от друга (рис. 10.12).

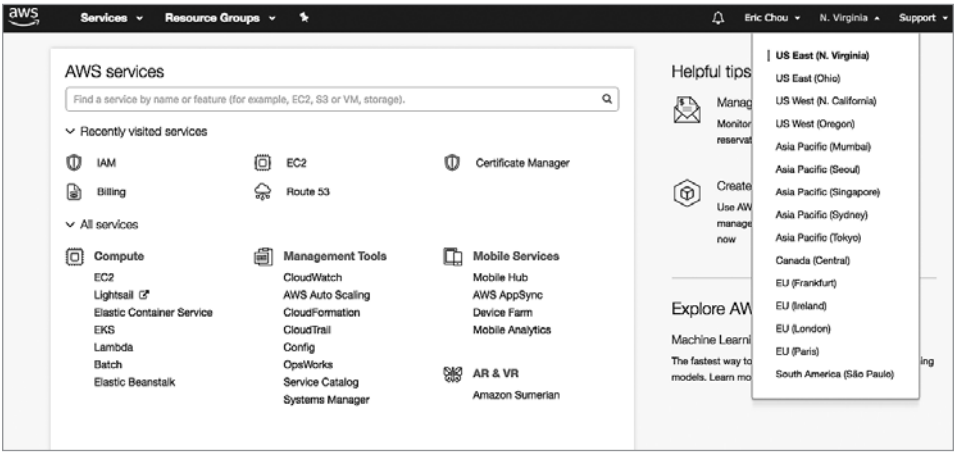


Рис. 10.11. Регионы AWS

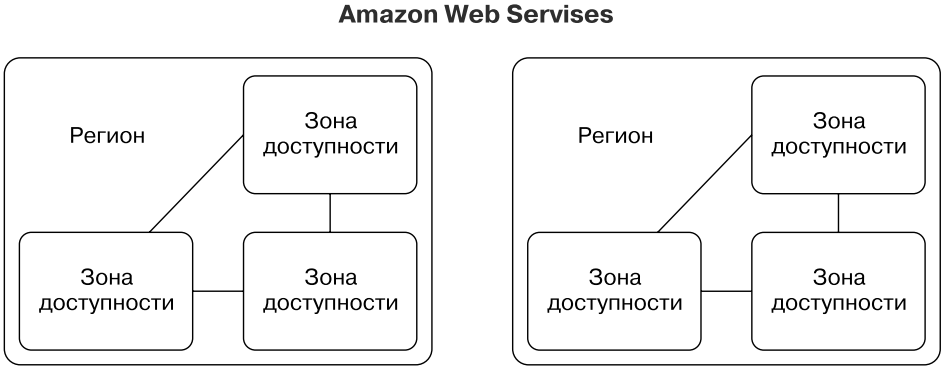


Рис. 10.12. Регионы и зоны доступности в AWS

Многие ресурсы, которые мы создаем в AWS, могут автоматически копироваться между зонами доступности (но не между регионами). Например, можно сконфигурировать управляемую реляционную базу данных (Amazon RDS) так, что она будет копироваться между зонами. Деление на зоны доступности играет важную роль в дублировании сервисов, и их ограничения необходимо учитывать в сетевых проектах.

Для каждой учетной записи используются свои идентификаторы зон доступности. Например, моя зона us-east-1a может не совпадать с зоной us-east-1a другого пользователя, даже несмотря на одинаковое название.

Получим список зон в регионе с помощью AWS CLI:

```
$ aws ec2 describe-availability-zones --region us-east-1
{
    "AvailabilityZones": [
        {
            "State": "available",
            "Messages": [],
            "RegionName": "us-east-1",
            "ZoneName": "us-east-1a",
            "ZoneId": "use1-az2"
        },
        {
            "State": "available",
            "Messages": [],
            "RegionName": "us-east-1",
            "ZoneName": "us-east-1b",
            "ZoneId": "use1-az4"
        },
<опущено>
```

Почему я уделяю столько внимания регионам и зонам доступности? Как вы сами дальше увидите, сетевые сервисы AWS обычно привязаны к какой-то географической области. *Виртуальное частное облако (Virtual Private Cloud, VPC)*, к примеру, должно целиком находиться в одном регионе, а каждая подсеть — принадлежать одной зоне доступности. С другой стороны, шлюзы NAT привязаны к зонам, поэтому, чтобы их продублировать, придется создать по одному шлюзу в каждой зоне.

Мы рассмотрим оба эти сервиса, и примеры с ними иллюстрируют, что регионы и зоны доступности — это основа сетевых сервисов AWS (рис. 10.13).

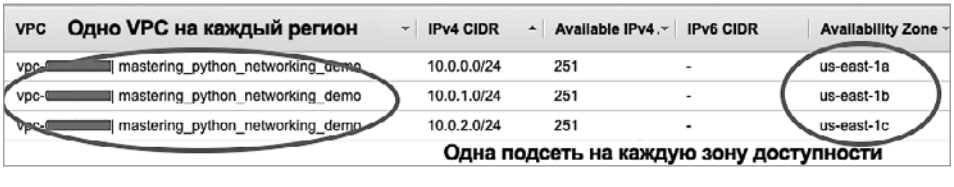


Рис. 10.13. VPC и зоны доступности в отдельных регионах

Граничные области AWS — часть сети доставки контента *AWS CloudFront* в 73 городах в 33 странах (по состоянию на октябрь 2019 года) и используются для доставки контента потребителям с низкой задержкой. Граничные узлы требуют меньше ресурсов, чем полноценные дата-центры, которые Amazon строит для регионов и зон доступности. Иногда их путают с настоящими регионами AWS. Если ресурсы предназначены только для граничной области, вам не будут

доступны такие сервисы AWS, как EC2 или S3. Мы еще вернемся к этой теме в разделе о AWS CloudFront CDN.

Транзитные центры AWS — один из наименее задокументированных аспектов сетей AWS. В выступлении на конференции AWS re:Invent 2014 (https://www.youtube.com/watch?v=JIQETrFC_SQ) Джеймс Гамильтон называет их точками агрегации для разных зон доступности и регионов. Если честно, после стольких лет сложно сказать, используются ли транзитные центры до сих пор и изменились ли их функции. Однако мы можем сделать предположение об их размещении и о том, какое отношение они имеют к сервису Direct Connect (подробнее о нем — ниже в этой главе).

Джеймс Гамильтон, вице-президент и выдающийся инженер из Amazon, — один из самых влиятельных технологов в AWS. Я считаю его авторитетным специалистом по сетевым технологиям AWS. Познакомьтесь с его идеями в его блоге, Perspectives, по адресу https://perspectives.mvdirona.com/.

В одной главе невозможно описать все сервисы AWS. Некоторые из них не имеют прямого отношения к сетям, поэтому здесь они не рассматриваются, но вы все равно должны о них знать.

- **IAM**, https://aws.amazon.com/iam/. Позволяет управлять доступом к сервисам и ресурсам AWS.
- **Amazon Resource Names (ARNs)**, https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html. Система, позволяющая однозначно идентифицировать любые ресурсы по всему облаку AWS, присваивая им уникальные имена. Эти имена используются для идентификации таких сервисов, как DynamoDB и API Gateway, которым нужен доступ к нашим VPC-ресурсам.
- **Amazon Elastic Compute Cloud (EC2)**, https://aws.amazon.com/ec2/. Сервис, позволяющий приобретать и выделять вычислительные мощности (такие как инстансы под управлением Linux и Windows) с помощью интерфейсов AWS. В наших примерах в этой главе используются инстансы EC2.

В учебных целях мы не станем рассматривать регионы AWS GovCloud (US) и China, так как они не используют глобальную инфраструктуру AWS, у каждого из них свои уникальные возможности и ограничения.

Это было относительно длинное, но важное введение в сетевые сервисы AWS. Понятия и термины отсюда будут использоваться в оставшейся части главы.

В следующем разделе мы рассмотрим самую важную, по моему мнению, технологию в AWS: VPC.

Виртуальное частное облако

Amazon VPC позволяет клиенту запускать ресурсы AWS в виртуальной сети, привязанной к его учетной записи. Это по-настоящему конфигурируемая сеть, в которой можно определять свои диапазоны IP-адресов, добавлять и удалять подсети, создавать маршруты, добавлять шлюзы VPN, назначать политики безопасности, подключать инстансы EC2 к своему дата-центру и многое другое.

Когда поддержки виртуальных частных облаков еще не существовало, все инстансы EC2 в одной зоне доступности, независимо от принадлежности, находились в общей плоской сети. Насколько комфортно чувствовали себя клиенты, размещая свою информацию в облаке? Наверное, не очень. В период между запуском EC2 и появлением VPC возможность создавать собственные частные сети была одной из самых востребованных.

Пакеты, покидающие ваш инстанс EC2 в VPC, перехватываются гипервизором. Гипервизор сопоставляет их с картой вашего VPC и назначает им исходящий и конечный IP-адреса реальных серверов AWS. Сервис, который этим занимается, обеспечивает гибкость виртуальных частных облаков, но также ограничивает их возможности (исключая сканирование сети и использование многоадресного вещания). Это, в конце концов, виртуальная сеть.

Начиная с декабря 2013 года все инстансы EC2 размещаются исключительно в VPC; вы больше не можете (и вряд ли когда-либо захотите) создать инстанс за пределами виртуального облака (например, EC2-Classic). Если инстанс EC2 создается в мастере запуска, он автоматически помещается в VPC по умолчанию с виртуальным интернет-шлюзом для публичного доступа. Но, как мне кажется, VPC по умолчанию следует использовать только в самых простых случаях. Обычно имеет смысл определить и сконфигурировать свое виртуальное частное облако.

Создадим VPC в US-East-1 с помощью консоли управления AWS (рис. 10.14).

Вы, наверное, помните, что VPC привязывается к отдельному региону, а подсети ограничены зонами доступности. Наше первое частное облако находится в `us-east-1`, а три подсети будут распределены по двум зонам доступности: `us-east-1a` и `us-east-1b`.

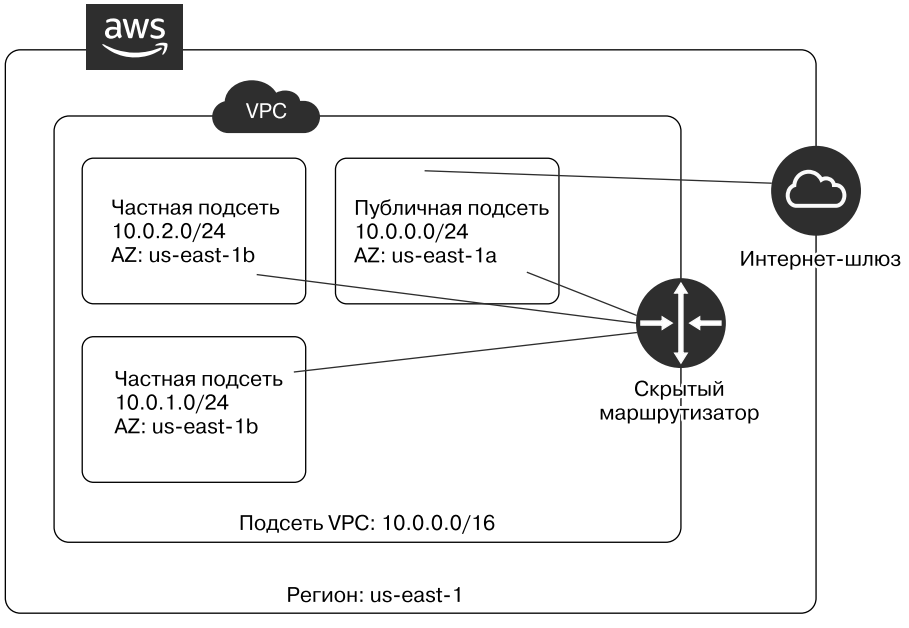


Рис. 10.14. Наше первое VPC в регионе US-East-1

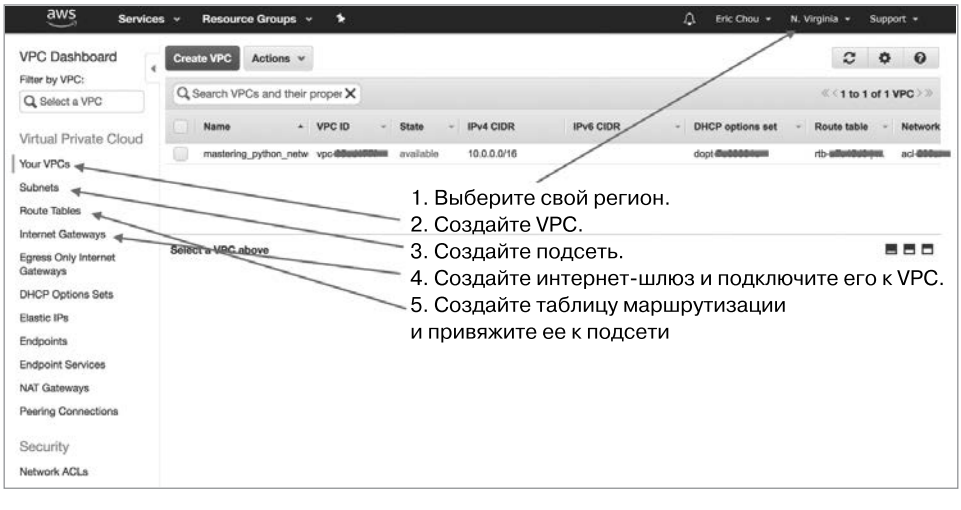


Рис. 10.15. Этапы создания VPC, подсети и других компонентов

Процесс создания VPC и подсетей в консоли AWS прост; в интернете есть хорошие практические руководства от Amazon. На рис. 10.15 я перечислил необходимые шаги и указал, в каких местах панели управления они выполняются.

Первые два шага требуют лишь нескольких щелчков кнопкой мыши; большинство сетевых инженеров легко с ними справятся, даже без какого-либо опыта. По умолчанию VPC содержит только локальный маршрут, 10.0.0.0/16. Теперь создадим интернет-шлюз и подключим его к VPC (рис. 10.16).

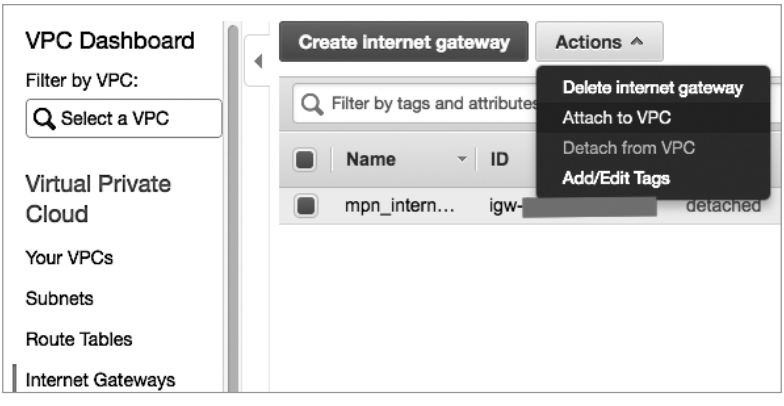


Рис. 10.16. Подключение интернет-шлюза к VPC в AWS

Теперь можно создать таблицу маршрутизации с маршрутом по умолчанию, направленным к интернет-шлюзу; это откроет доступ снаружи. Привяжем эту таблицу к нашей подсети `10.0.0.0/24` в `us-east-1a`, чтобы VPC имело доступ к интернету (рис. 10.17).

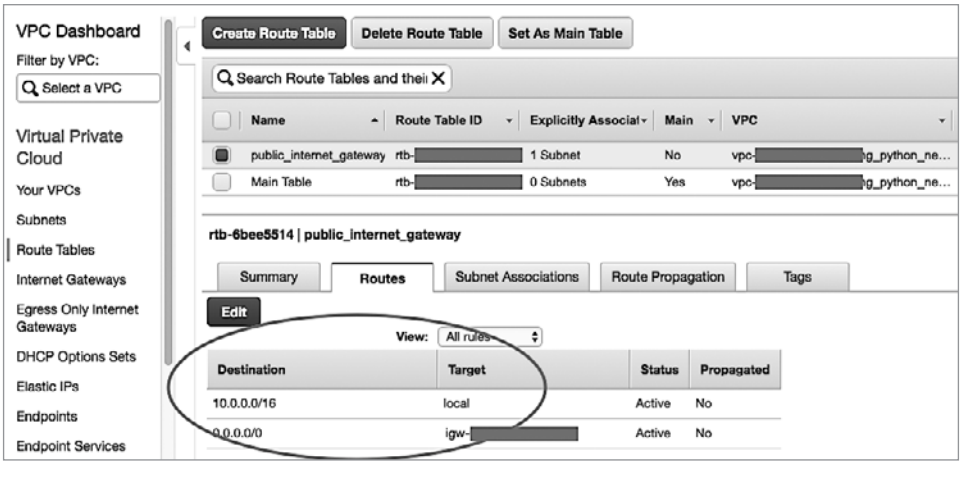


Рис. 10.17. Таблица маршрутизации

А теперь воспользуемся Boto3 Python SDK и посмотрим, что у нас получилось. Для VPC я выбрал тег `mastering_python_networking_demo`; его можно использовать как фильтр:

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

import json, boto3

region = 'us-east-1'
vpc_name = 'mastering_python_networking_demo'

ec2 = boto3.resource('ec2', region_name=region)
client = boto3.client('ec2')

filters = [{'Name':'tag:Name', 'Values':[vpc_name]}]

vpcs = list(ec2.vpcs.filter(Filters=filters))
for vpc in vpcs:
    response = client.describe_vpcs(
                VpcIds=[vpc.id,]
               )
    print(json.dumps(response, sort_keys=True, indent=4))
```

Этот сценарий запрашивает информацию о только что созданном VPC в заданном регионе:

```
(venv) $ python Chapter10_1_query_vpc.py
{
    "ResponseMetadata": {
        <опущено>
        "HTTPStatusCode": 200,
        "RequestId": "9416b03f-<опущено> ",
        "RetryAttempts": 0
    },
    "Vpcs": [
        {
            "CidrBlock": "10.0.0.0/16",
            "CidrBlockAssociationSet": [
                {
                    "AssociationId": "vpc-cidr-assoc-<опущено>",
                    "CidrBlock": "10.0.0.0/16",
                    "CidrBlockState": {
                        "State": "associated"
                    }
                }
            ],
            "DhcpOptionsId": "dopt-<опущено>",
            "InstanceTenancy": "default",
            "IsDefault": false,
            "OwnerId": "<опущено>",
            "State": "available",
            "Tags": [
```

```
                {
                    "Key": "Name",
                    "Value": "mastering_python_networking_demo"
                }
            ],
            "VpcId": "vpc-<опущено>"
        }
    ]
}
```

Документацию с описанием Boto3 VPC API смотрите на странице https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/ec2.html#vpc.

Если бы мы создали инстансы EC2 и поместили их в разные подсети в их исходном виде, они могли бы обращаться друг к другу. Вам, наверное, интересно, как подсети общаются между собой в рамках VPC, учитывая, что интернет-шлюз создан только в подсети 1a. Когда хост хочет обратиться за пределы физической локальной сети, ему нужно подключиться к маршрутизатору.

То же относится и к VPC, только в данном случае этот маршрутизатор *скрыт* и содержит локальную таблицу маршрутизации по умолчанию, которая в этом примере выглядит как `10.0.0.0/16`. Этот скрытый маршрутизатор был создан вместе с VPC. Любая подсеть, не имеющая своей таблицы маршрутизации, связана с главной таблицей.

Таблицы и цели маршрутизации

Маршрутизация — один из важнейших аспектов сетевых технологий. Рассмотрим подробнее, как она реализована в AWS VPC. Мы уже знаем, что вместе с VPC создаются скрытый маршрутизатор и главная таблица маршрутизации. В последнем примере мы создали интернет-шлюз и таблицу маршрутизации с маршрутом по умолчанию, который указывает на этот шлюз, после чего привязали эту таблицу к подсети.

До сих пор цель маршрутизации была единственной концепцией, которая выделяла VPC на фоне традиционных сетей. В контексте физических сетей ее можно считать эквивалентом следующего транзитного участка.

Давайте подытожим:

- у каждого VPC есть скрытый маршрутизатор;
- у каждого VPC есть главная таблица маршрутизации с заданным локальным маршрутом;

- вы можете создавать свои таблицы маршрутизации;
- каждая подсеть может работать в соответствии как с пользовательской, так и с главной таблицей маршрутизации;
- целью маршрутизации в таблице может выступать интернет-шлюз, NAT-шлюз, другие хосты в VPC и т. д.

Для просмотра пользовательской таблицы маршрутизации и ее связей с подсетями можно использовать Boto3:

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

import json, boto3

region = 'us-east-1'
vpc_name = 'mastering_python_networking_demo'

ec2 = boto3.resource('ec2', region_name=region)
client = boto3.client('ec2')

response = client.describe_route_tables()
print(json.dumps(response['RouteTables'][0], sort_keys=True,
indent=4))
```

Главная таблица маршрутизации является скрытой, поэтому API ее не возвращает. Поскольку у нас есть всего одна пользовательская таблица, мы увидим:

```
(venv) $ python Chapter10_2_query_route_tables.py
{
    "Associations": [
        <опущено>
    ],
    "OwnerId": "<опущено>",
    "PropagatingVgws": [],
    "RouteTableId": "rtb-<опущено>",
    "Routes": [
        {
            "DestinationCidrBlock": "10.0.0.0/16",
            "GatewayId": "local",
            "Origin": "CreateRouteTable",
            "State": "active"
        },
        {
            "DestinationCidrBlock": "0.0.0.0/0",
            "GatewayId": "igw-041f287c",
            "Origin": "CreateRoute",
            "State": "active"
        }
    ],
    "Tags": [
        {
```

```
                "Key": "Name",
                "Value": "public_internet_gateway"
            }
        ],
        "VpcId": "vpc-<опущено>"
    }
```

Мы уже создали первую публичную подсеть. Выполним те же шаги, чтобы создать еще две подсети, `us-east-1b` и `us-east-1c`, но на этот раз сделаем их частными. В итоге у нас получится три подсети: публичная `10.0.0.0/24` в `us-east-1a` и частные `10.0.1.0/24` и `10.0.2.0/24` в `us-east-1b` и `us-east-1c` соответственно.

Итак, у нас есть рабочее VPC с тремя подсетями: одной публичной и двумя частными. До сих пор для взаимодействия с AWS VPC мы использовали AWS CLI и библиотеку Boto3. Рассмотрим еще одно средство автоматизации от AWS, *CloudFormation*.

Автоматизация с использованием CloudFormation

AWS CloudFormation (https://aws.amazon.com/cloudformation/) — это инструмент, который позволяет описывать и запускать нужные нам ресурсы с помощью текстового файла. С помощью CloudFormation можно сформировать еще одно VPC в регионе `us-west-1` (рис. 10.18).

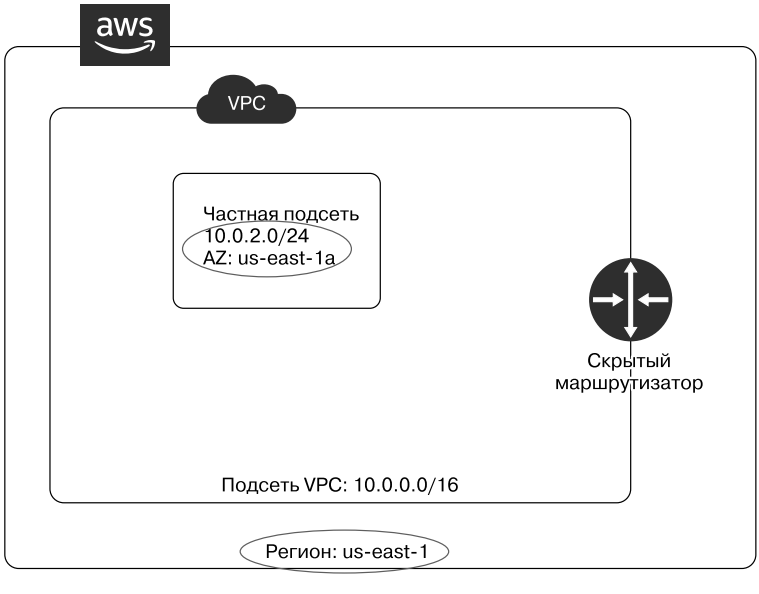


Рис. 10.18. VPC для US-West-1

Описание для CloudFormation может быть в формате YAML или JSON; в первом примере, `Chapter10_3_cloud_formation.yml`, мы используем YAML:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Create VPC in us-west-1
Resources:
  myVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: '10.1.0.0/16'
      EnableDnsSupport: 'false'
      EnableDnsHostnames: 'false'
      Tags:
        - Key: Name
        - Value: 'mastering_python_networking_demo_2'
```

Сформировать VPC на основе этого описания можно с помощью AWS CLI. Обратите внимание: здесь мы указываем регион `us-west-1`:

```
(venv) $ aws --region us-west-1 cloudformation create-stack --stack-name
'mpn- ch10-demo' --template-body file://Chapter10_3_cloud_formation.yml
{
"StackId": "arn:aws:cloudformation:us-west-1:<опущено>:stack/mpn-ch10-
demo/<опущено>"
}
```

Проверим результат с помощью AWS CLI:

```
(venv) $ aws --region us-west-1 cloudformation describe-stacks --stackname
mpn-ch10-demo
{
    "Stacks": [
        {
            "StackId": "arn:aws:cloudformation:us-west-1:<опущено>:stack/
             mpn-ch10-demo/bbf5abf0-8aba-11e8-911f-500cadc9fefe",
            "StackName": "mpn-ch10-demo",
            "Description": "Create VPC in us-west-1",
            "CreationTime": "2018-07-18T18:45:25.690Z",
            "LastUpdatedTime": "2018-07-18T19:09:59.779Z",
            "RollbackConfiguration": {},
            "StackStatus": "UPDATE_ROLLBACK_COMPLETE",
            "DisableRollback": false,
            "NotificationARNs": [],
            "Tags": [],
            "EnableTerminationProtection": false,
            "DriftInformation": {
                "StackDriftStatus": "NOT_CHECKED"
            }
        }
    ]
}
```

На основе этого описания CloudFormation создал VPC без подсети. Удалим это VPC и повторим попытку, передав описание `Chapter10_4_cloud_formation_full.yml`, чтобы создать VPC с подсетью. Обратите внимание, что идентификатор облака VPC-ID неизвестен до его создания, поэтому в описании подсети сошлемся на него с помощью специальной переменной. Этот прием подойдет и для других ресурсов — таблицы маршрутизации и интернет-шлюза:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Create subnet in us-west-1
Resources:
  myVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: '10.1.0.0/16'
      EnableDnsSupport: 'false'
      EnableDnsHostnames: 'false'
      Tags:
        - Key: Name
          Value: 'mastering_python_networking_demo_2'

  mySubnet:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref myVPC
      CidrBlock: '10.1.0.0/24'
      AvailabilityZone: 'us-west-1a'
      Tags:
        - Key: Name
          Value: 'mpn_demo_subnet_1'
```

Сформируем VPC и убедимся, что все ресурсы благополучно созданы:

```
(venv) $ aws --region us-west-1 cloudformation create-stack --stack-name
mpn-ch10-demo-2 --template-body file://Chapter10_4_cloud_formation_full.
yml
{
"StackId": "arn:aws:cloudformation:us-west-1:<опущено>:stack/mpn-ch10- demo-
2/<опущено>"
}

$ aws --region us-west-1 cloudformation describe-stacks --stack-name mpnch10-
demo-2
{
"Stacks": [
{
"StackStatus": "CREATE_COMPLETE",
...
"StackName": "mpn-ch10-demo-2", "DisableRollback": false
}
]
}
```

Информацию о VPC и подсети можно проверить также в консоли AWS. Не забудьте указать правильный регион в раскрывающемся меню в правом верхнем углу (рис. 10.19).

Рис. 10.19. VPC в US-West-1

Взглянем на подсеть (рис. 10.20).

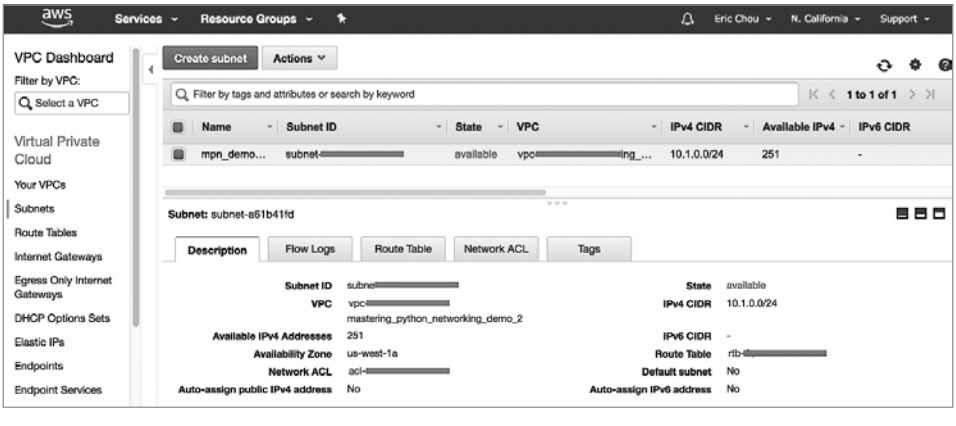


Рис. 10.20. Подсеть в US-West-1

Теперь у нас есть два VPC на двух побережьях Соединенных Штатов. Сейчас они существуют сами по себе, словно два острова. Может, вам это и нужно, а может, и нет. Если вы хотите соединить два VPC, чтобы они могли сообщать-

ся напрямую, используйте VPC-пиринг (https://docs.aws.amazon.com/AmazonVPC/latest/PeeringGuide/vpc-peering-basics.html).

У VPC-пиринга есть несколько ограничений; например, не допускается пересечение блоков адресов IPv4 или IPv6 CIDR. Дополнительные ограничения относятся к VPC-пирингу между регионами. Не забудьте свериться с документацией.

VPC-пиринг не ограничивается одной учетной записью. Можно соединять VPC, принадлежащие разным пользователям; главное, чтобы противоположная сторона приняла ваш запрос и чтобы были улажены вопросы безопасности, маршрутизации и доменных имен.

Далее мы рассмотрим группы безопасности и списки управления доступом для VPC.

Группы безопасности и списки доступа к сети

Группы безопасности и списки доступа к сети ищите в разделе Security (Безопасность) в настройках VPC (рис. 10.21).

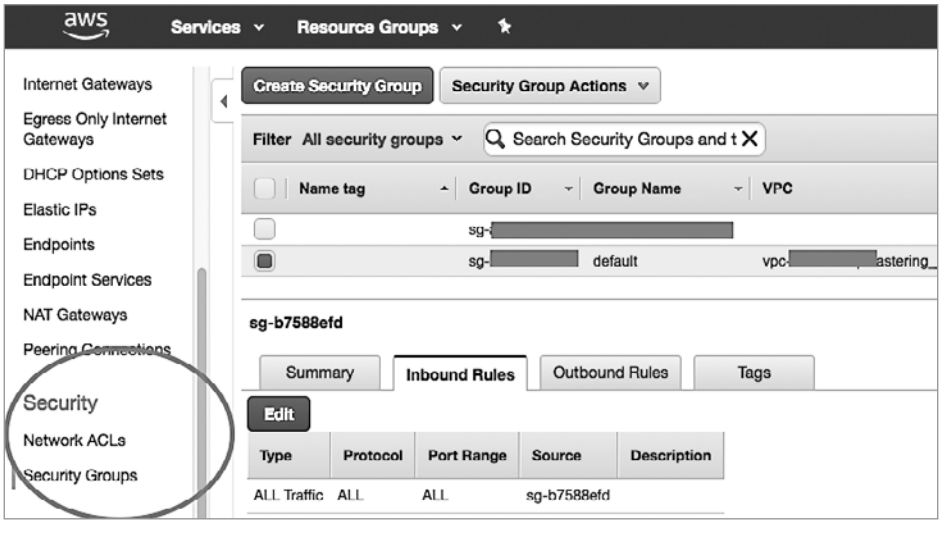


Рис. 10.21. Безопасность VPC

Группа безопасности — это виртуальный брандмауэр с постоянной конфигурацией, который управляет входящим и исходящим трафиком ресурсов. В большинстве случаев группа безопасности используется для ограничения публичного доступа к инстансам EC2. В настоящее время в каждом VPC может быть не больше 500 групп, и каждая группа может содержать до 50 входящих и 50 исходящих правил.

Следующий сценарий, `Chapter10_5_security_group.py`, создает группу безопасности с двумя простыми правилами для входящего трафика:

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

import boto3

ec2 = boto3.client('ec2')

response = ec2.describe_vpcs()
vpc_id = response.get('Vpcs', [{}])[0].get('VpcId', '')

# Запрашиваем ID группы безопасности
response = ec2.create_security_group(GroupName='mpn_security_group',
                                     Description='mpn_demo_sg',
                                     VpcId=vpc_id)
security_group_id = response['GroupId']
data = ec2.authorize_security_group_ingress(
    GroupId=security_group_id,
    IpPermissions=[
        {'IpProtocol': 'tcp',
         'FromPort': 80,
         'ToPort': 80,
         'IpRanges': [{'CidrIp': '0.0.0.0/0'}]},
        {'IpProtocol': 'tcp',
         'FromPort': 22,
         'ToPort': 22,
         'IpRanges': [{'CidrIp': '0.0.0.0/0'}]}
    ])
print('Ingress Successfully Set %s' % data)

# Описываем группу безопасности
#response = ec2.describe_security_groups(GroupIds=[security_group_id])
print(security_group_id)
```

Выполним этот сценарий, чтобы создать группу безопасности, которую затем можно связать с другими нашими ресурсами в AWS:

```
(venv) $ python Chapter10_5_security_group.py
Ingress Successfully Set {'ResponseMetadata': {'RequestId': '<опущено>',
'HTTPStatusCode': 200, 'HTTPHeaders': {'server': 'AmazonEC2', 'contenttype':
'text/xml;charset=UTF-8', 'date': 'Wed, 18 Jul 2018 20:51:55 GMT',
'content-length': '259'}, 'RetryAttempts': 0}} sg-<опущено>
```

Списки управления доступом (Access Control Lists, ACL) — это дополнительный слой безопасности без состояния. У каждой подсети в VPC есть свой ACL. Поскольку ACL не имеет состояния, необходимо указывать как входящие, так и исходящие правила.

Ниже перечислены важные различия между ACL и группами безопасности:

- группы безопасности действуют на уровне сетевого интерфейса, а ACL — на уровне подсети;
- группа безопасности поддерживает только правила типа `allow`, а для ACL можно указывать как `allow`, так и `deny`;
- группа безопасности хранит свое состояние, поэтому ответный трафик автоматически пропускается, а в ACL его нужно разрешать отдельно.

Рассмотрим самую крутую сетевую технологию в AWS: Elastic IP. Когда я только о ней узнал, меня поразила ее способность динамически назначать и переназначать IP-адреса.

Elastic IP

Elastic IP (EIP) — это механизм работы с адресами IPv4, доступными из интернета.

По состоянию на конец 2019 года EIP не поддерживал IPv6.

EIP можно динамически назначить инстансу EC2, сетевому интерфейсу или другим ресурсам. Его особенности:

- EIP привязан к конкретной учетной записи и ограничен одним регионом. Например, EIP в `us-east-1` можно привязывать только к ресурсам в `us-east-1`;
- вы можете отвязать EIP от одного ресурса и назначить его другому. Такая гибкость иногда используется для обеспечения высокой доступности. Например, вы можете мигрировать на более мощный инстанс EC2, переназначив ему IP-адрес старого инстанса;
- за каждый час использования EIP взимается небольшая плата.

EIP запрашивают в консоли AWS и затем назначают его подходящим ресурсам (рис. 10.22).

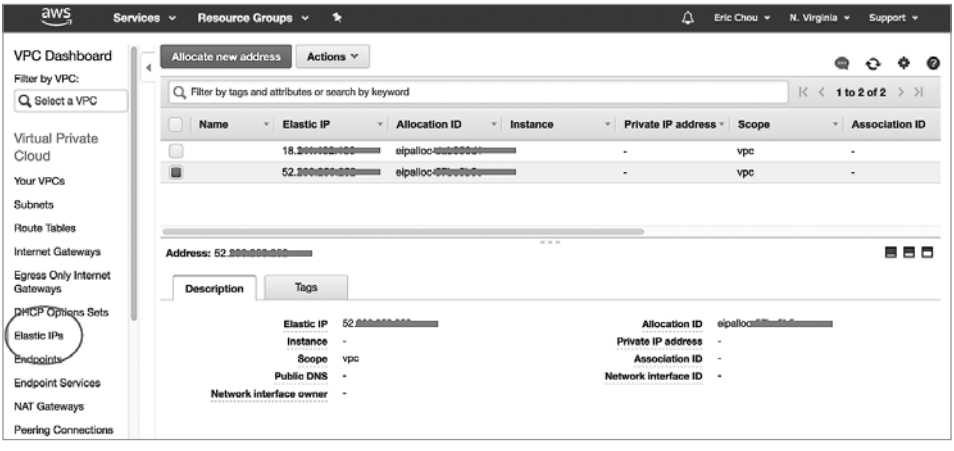


Рис. 10.22. Elastic IP

К сожалению, в целях экономии в каждом регионе по умолчанию доступно лишь пять EIP (https://docs.aws.amazon.com/vpc/latest/userguide/amazon-vpc-limits.html). Но при необходимости этот лимит можно повысить, если обратиться в службу поддержки AWS.

В следующем разделе вы увидите, как с помощью NAT-шлюзов открыть доступ из интернета к частным подсетям.

NAT-шлюзы

Чтобы открыть доступ из интернета к нашим хостам EC2 в публичной подсети, можно выделить EIP и привязать его к сетевому интерфейсу хоста. Однако на момент написания этой книги для каждого EC2-VPC можно было выделить не больше пяти EIP (https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_Appendix_Limits.html#vpc-limits-eips). Иногда удобно иметь возможность разрешить исходящий трафик для хоста в частной подсети без назначения ему постоянного EIP.

Вот где может пригодиться *NAT-шлюз*. С его помощью можно временно разрешить прохождение исходящего трафика для хоста в частной подсети, выполняя преобразование сетевых адресов. Эта операция похожа на *трансляцию*

«порт — адрес» (Port Address Translation, PAT), которую мы обычно применяем в корпоративном брандмауэре. Для использования NAT-шлюза выполните следующие шаги.

1. Создайте NAT-шлюз в подсети с доступом к интернет-шлюзу, используя AWS CLI, библиотеку Boto3 или консоль AWS. У NAT-шлюза должен быть EIP.
2. Настройте маршрут по умолчанию в частной подсети к NAT-шлюзу.
3. NAT-шлюз будет следовать маршруту по умолчанию, направляя внешний трафик к интернет-шлюзу.

Эта процедура проиллюстрирована на рис. 10.23.

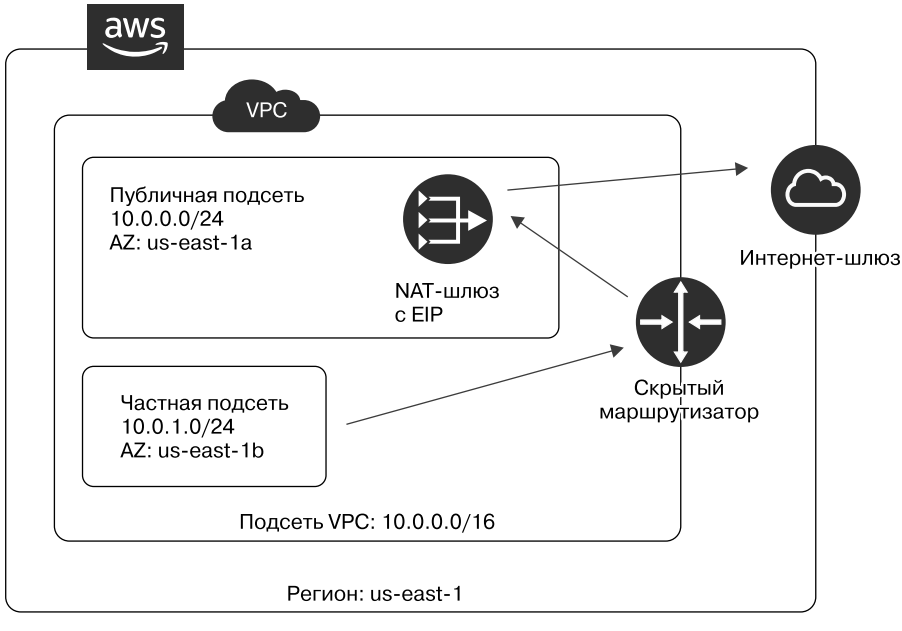


Рис. 10.23. Работа NAT-шлюза

Один из самых часто задаваемых вопросов о NAT-шлюзах касается выбора подсети, в которой этот шлюз должен находиться. Помните, что NAT-шлюзу нужен публичный доступ. Поэтому его следует создавать в подсети с публичным доступом к интернету и назначенным EIP (рис. 10.24).

Рис. 10.24. Создание NAT-шлюза

В следующем разделе мы рассмотрим подключение виртуальной сети в AWS к физической сети.

Direct Connect и VPN

До этого момента наше виртуальное частное облако было автономным и находилось в сети AWS. Оно гибкое и функциональное, но для доступа к его ресурсам приходится использовать публичные сервисы, такие как SSH или HTTPS.

В этом разделе мы рассмотрим два способа подключения к AWS VPC из частной сети: VPN-шлюз на основе IPSec и Direct Connect.

VPN-шлюзы

Первый способ подключения нашей локально размещенной сети к VPC заключается в использовании традиционных VPN-соединений поверх IPSec. Нам понадобится публично доступное устройство, способное соединяться с VPN-устройствами в AWS.

Шлюз клиента должен поддерживать VPN IPSec на основе маршрутов; при этом по VPN-соединению может проходить протокол маршрутизации и обычный пользовательский трафик. В настоящий момент для обмена маршрутами AWS рекомендует применять BGP.

На стороне VPC можно организовать аналогичную таблицу маршрутизации, в которой отдельная подсеть направлена к *виртуальному частному шлюзу (Virtual Private Gateway, VPG)* (рис. 10.25).

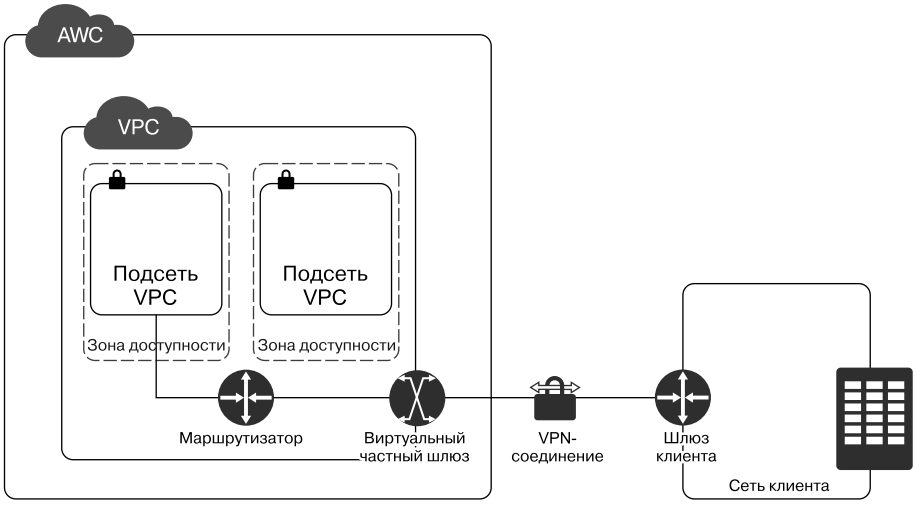


Рис. 10.25. VPN-соединение на стороне VPC (источник: https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_VPN.html)

Direct Connect

VPN-соединение поверх IPSec позволяет легко связать локально размещенное оборудование с облачными ресурсами AWS. Но ему присущи те же проблемы, что и туннелям IPSec через интернет: оно ненадежное, и с этим мало что можно поделать. Имеет ограниченные возможности мониторинга качества работы, а *соглашение об уровне обслуживания (Service-Level Agreement, SLA)* распространяется только на ту часть интернета, которую мы можем контролировать.

Как следствие — для передачи любого критически важного трафика обычно используют второй вариант, который предоставляет Amazon, а именно AWS Direct Connect. Эта технология позволяет клиентам подключать свои дата-центры и отдельные серверы к AWS VPC по выделенному виртуальному каналу.

Самое сложное в организации такого подключения — вывести сеть туда, откуда ее можно физически подключить к AWS; обычно это узел обмена трафиком.

Список местоположений AWS Direct Connect смотрите на странице https://aws.amazon.com/directconnect/details/. Канал Direct Connect — это обычное оптоволоконное соединение, которое можно заказать в определенном дата-центре; ваша сеть подключается к сетевому порту, после чего настраивается соединение dot1q trunk.

Все больше сторонних провайдеров предлагают услугу Direct Connect на основе канала MPLS и агрегированных соединений. Один из самых доступных вариантов, который мне удалось найти, — Equinix Cloud Exchange Fabric[1]. Он позволяет использовать один и тот же канал для соединения с разными облачными провайдерами, и это намного дешевле, чем выделенный канал (рис. 10.26).

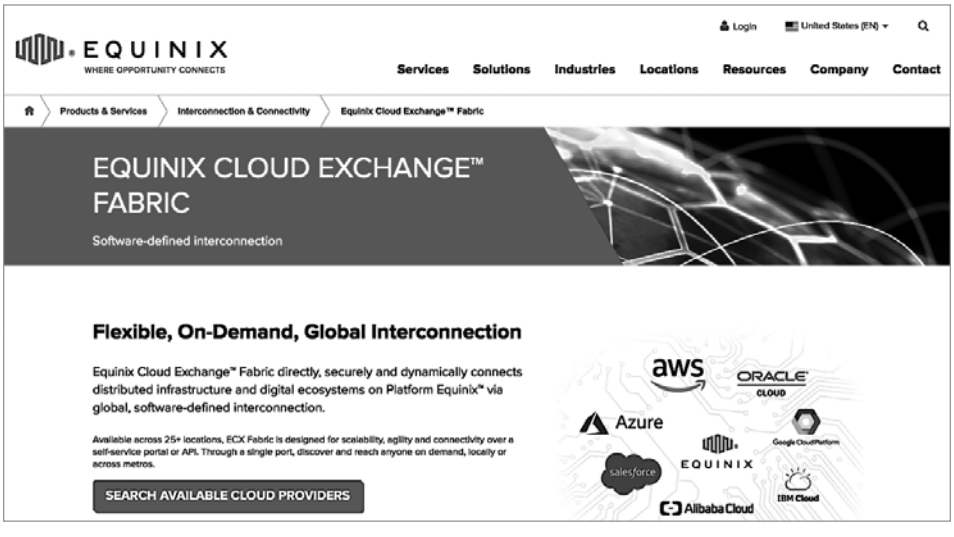


Рис. 10.26. Equinix Cloud Exchange

В следующем разделе мы рассмотрим несколько сервисов AWS для масштабирования сетей.

Сервисы для масштабирования сетей

В этом разделе мы обсудим некоторые сетевые услуги, которые предлагает AWS. Многие из них не имеют прямого отношения к сети (например, DNS и сети распространения содержимого). Нас они интересуют ввиду их тесной связи с сетевыми технологиями и влияния на производительность приложений.

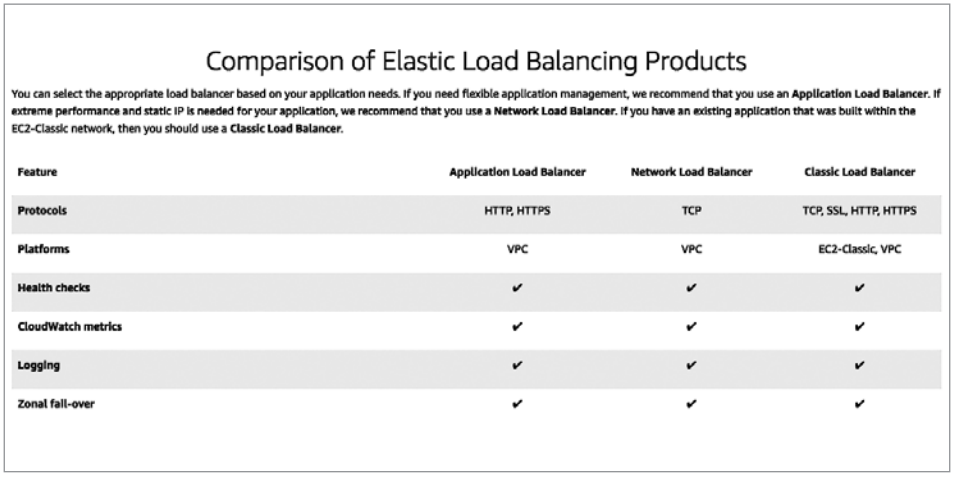
[1] В мае 2020 года данный продукт был переименован в Equinix Fabric: https://www.equinix.com/interconnection-services/equinix-fabric. — *Примеч. ред.*

Elastic Load Balancing

Elastic Load Balancing (ELB) позволяет автоматически распределять трафик, поступающий из интернета, между инстансами EC2. Как и в случае с балансировщиками нагрузки в физических сетях, это дает возможность дублировать ресурсы и улучшать отказоустойчивость, снижая при этом нагрузку на отдельные серверы. ELB поддерживает балансировку нагрузки как на уровне приложений, так и на уровне сети.

Application Load Balancer (ALB) управляет веб-трафиком по HTTP и HTTPS; *Network Load Balancer (NLB)* работает на уровне TCP. Для веб-приложений имеет смысл использовать ALB; в остальных случаях лучше выбрать NLB.

Подробное сравнение ALB и NLB — на странице https://aws.amazon.com/elasticloadbalancing/details/.



Comparison of Elastic Load Balancing Products

You can select the appropriate load balancer based on your application needs. If you need flexible application management, we recommend that you use an **Application Load Balancer**. If extreme performance and static IP is needed for your application, we recommend that you use a **Network Load Balancer**. If you have an existing application that was built within the EC2-Classic network, then you should use a **Classic Load Balancer**.

Feature	Application Load Balancer	Network Load Balancer	Classic Load Balancer
Protocols	HTTP, HTTPS	TCP	TCP, SSL, HTTP, HTTPS
Platforms	VPC	VPC	EC2-Classic, VPC
Health checks	✔	✔	✔
CloudWatch metrics	✔	✔	✔
Logging	✔	✔	✔
Zonal fail-over	✔	✔	✔

Рис. 10.27. ELB comparison
(источник: https://aws.amazon.com/elasticloadbalancing/details/)

ELB позволяет распределять трафик в момент его прохождения через определенный ресурс в нашем регионе. Если вам нужна балансировка нагрузки между регионами, используйте сервис AWS Route 53 DNS (известный также как Global Server Load Balancing).

Сервис Route 53 DNS

Все мы знаем, что такое службы доменных имен, и Route 53 — это реализация такой службы от AWS. Route 53 — полноценный доменный регистратор, который позволяет покупать и администрировать доменные имена прямо из AWS. Что касается сетевых сервисов, то DNS позволяет распределять нагрузку между географическими регионами, циклически перебирая DNS-записи.

Для балансировки трафика с помощью DNS необходимы:

- балансировщик нагрузки в каждом регионе, который вы собираетесь балансировать;
- зарегистрированное доменное имя (его не обязательно регистрировать в Route 53);
- Route 53 — DNS-сервис для доменов.

Route 53 поддерживает политику на основе величины задержки с проверкой работоспособности, которую можно использовать для маршрутизации между двумя активными экземплярами ELB. В следующем разделе мы рассмотрим сеть доставки содержимого под названием CloudFront, которая встроена в AWS.

Доставка содержимого с использованием CloudFront

CloudFront — это *сеть доставки содержимого (Content Delivery Network, CDN)* от Amazon. Она снижает задержку доступа к данным за счет их физического размещения как можно ближе к клиенту. Такими данными могут быть: статическое содержимое веб-страниц, видеоролики, приложения, API или с недавних пор Lambda-функции. Граничные узлы CloudFront могут находиться как в регионах AWS, так и во многих других местах по всему миру. В целом этот сервис работает следующим образом:

- пользователи обращаются к веб-сайту за одним или несколькими объектами;
- DNS передает запрос ближайшему граничному узлу Amazon CloudFront;
- граничный узел CloudFront либо выдает содержимое из кэша, либо запрашивает объект в месте его хранения.

Обычно с AWS CloudFront и сервисами CDN имеют дело разработчики приложений и инженеры DevOps. Но стоит понимать, как работают эти технологии.

Другие сетевые сервисы от AWS

В AWS есть множество других сетевых сервисов. Ниже перечислены самые важные.

- **AWS Transit VPC** (https://aws.amazon.com/blogs/aws/aws-solution-transit-vpc/). Это средство для подключения нескольких виртуальных частных облаков к одному, которое служит транзитным центром. Это относительно новый сервис, он может минимизировать количество соединений, которые вам необходимо настраивать и администрировать. Его также можно использовать для разделения ресурсов между учетными записями AWS.
- **Amazon GuardDuty** (https://aws.amazon.com/guardduty/). Это управляемый сервис для обнаружения угроз безопасности, который непрерывно ищет вредоносное или неразрешенное поведение, помогая защитить наши рабочие нагрузки в AWS. Среди прочего отслеживает API-вызовы и случаи несанкционированного развертывания.
- **AWS WAF** (https://aws.amazon.com/waf/). Это брандмауэр для веб-приложений, который помогает защититься от распространенных эксплойтов. Позволяет устанавливать свои правила веб-безопасности, пропускающие или блокирующие трафик.
- **AWS Shield** (https://aws.amazon.com/shield/). Это управляемый сервис для защиты приложений, выполняющихся в AWS, от *DDoS (Distributed Denial of Service — распределенная атака на отказ в обслуживании)*. Его базовые функции предоставляются бесплатно для всех клиентов; за использование расширенной версии AWS Shield нужно доплачивать.

Компания Amazon постоянно выпускает новые потрясающие сетевые сервисы. Не все они такие же фундаментальные, как VPC или NAT-шлюзы, но все приносят пользу в конкретной области.

Резюме

Эта глава была посвящена облачным сетевым сервисам в AWS. Мы обсудили такие понятия, как регион, зона доступности, граничные узлы и транзитный центр. Общее понимание устройства сети AWS позволит вам ориентироваться в ограничениях отдельных сетевых сервисов. Мы использовали AWS CLI, библиотеку Python Boto3 и CloudFormation для автоматизации некоторых задач.

Мы подробно рассмотрели виртуальные частные облака AWS, включая настройку таблицы и целей маршрутизации. В примере с группами безопасности и сетевыми ACL мы позаботились о безопасности нашего VPC. Мы также поговорили о EIP и NAT-шлюзах в контексте предоставления доступа снаружи.

Существует два способа подключения AWS VPC к локально размещенным сетям: Direct Connect и IPSec VPN. Мы кратко рассмотрели каждый из них, отметив их преимущества. Также мы обсудили сервисы для масштабирования сетей, которые предлагает AWS, включая Elastic Load Balancing, Route 53 DNS и CloudFront.

В следующей главе речь пойдет о сетевых сервисах другого облачного провайдера, Microsoft Azure.

11

Облачные сетевые технологии Azure

Как мы уже видели в главе 10, облачные сетевые технологии помогают подключить нашу организацию к облачным ресурсам. Для сегментирования и защиты виртуальных машин можно использовать *виртуальную сеть*. Таким же образом можно подключать наши локальные ресурсы к облаку. Компания Amazon, будучи первопроходцем в этой области, обычно считается лидером рынка. В этой главе мы рассмотрим сетевые продукты еще одного важного облачного провайдера, Microsoft Azure.

Проект Microsoft Azure был основан в 2008 году и изначально назывался Project Red Dog. Его публичный выпуск состоялся 1 февраля 2010 года. На тот момент он носил название Windows Azure, но в 2014 году его переименовали в Microsoft Azure. Учитывая, что первый выпуск S3 состоялся в 2006 году, на тот момент команда AWS фактически опережала Microsoft Azure на четыре года. Попытка догнать AWS была непростой задачей даже для компании с огромными ресурсами. Но у Microsoft были уникальные конкурентные преимущества, обусловленные многолетним опытом выпуска успешных продуктов и прочными отношениями с корпоративными клиентами.

Azure уделяет большое внимание использованию существующих продуктов Microsoft и сложившихся отношений с клиентами, поэтому облачные технологии этого провайдера имеют важные особенности. Например, одной из основных причин, подталкивающих клиентов установить соединение с Azure через ExpressRoute, аналог AWS Direct Connect, может быть желание повысить

удобство работы с Office 365. В качестве еще одного примера можно привести ситуацию, когда у клиента уже есть соглашение об уровне обслуживания с Microsoft, в которое можно включить Azure.

В этой главе мы обсудим сетевые сервисы, которые предлагает Azure, и возможность работы с ними из Python. Мы возьмем за основу те аспекты облачных технологий, которые уже рассмотрели в предыдущей главе, и попытаемся сравнивать продукты AWS и Azure, где это уместно.

В частности, мы обсудим:

- подготовку к работе с Azure и дадим общий обзор сетевых технологий;
- *виртуальные сети* Azure (VNets); технология Azure VNet похожа на AWS VPC и предоставляет клиентам частную сеть в облаке Azure;
- ExpressRoute и VPN-соединения;
- балансировщики нагрузки в сети Azure;
- другие сетевые сервисы Azure.

В предыдущей главе мы познакомились со многими важными концепциями облачных сетевых технологий. Используем эти знания и сравним для начала продукты, доступные в Azure и AWS.

Сравнение сетевых сервисов в Azure и AWS

Изначально проект Azure был ориентирован на модели *SaaS (Software-as-a-Service — программное обеспечение как услуга)* и *PaaS (Platform-as-a-Service — платформа как услуга)* и в меньшей степени на *IaaS (Infrastructure-as-a-Service — инфраструктура как услуга)*. В моделях SaaS и PaaS сетевые сервисы нижних уровней зачастую скрыты от пользователя. Например, SaaS-продукт Office 365 обычно предоставляется в виде удаленно размещенной конечной точки, к которой можно обращаться через интернет. PaaS-продукты для создания веб-приложений с использованием Azure App Services часто имеют вид полностью управляемых процессов на основе таких популярных фреймворков, как .NET или Node.js.

Модель IaaS, с другой стороны, требует от нас построить свою инфраструктуру в облаке Azure. Существенная часть целевой аудитории этих продуктов уже имела опыт работы с лидером в этой области, AWS. Чтобы помочь с переходом

на свое облако, Azure предоставляет на своем веб-сайте сравнение своих сервисов с аналогичными предложениями от AWS (https://docs.microsoft.com/ru-ru/azure/architecture/aws-professional/services). Я часто посещаю эту полезную страницу, когда не могу найти в Azure аналог сервиса от AWS, особенно если его название не совсем точно описывает предоставляемую им услугу. Например, можете ли вы сказать по одному лишь названию, что такое SageMaker? Вот и я не могу.

Я часто использую эту страницу и для сравнительного анализа. Например, если мне нужно сравнить стоимость выделенного соединения с AWS и Azure, я сначала убеждаюсь в том, что аналог AWS Direct Connect — Azure ExpressRoute, и затем открываю приведенную там ссылку, чтобы узнать подробности об этом сервисе.

Прокрутив страницу до раздела Сеть, можно увидеть, что многие продукты Azure похожи на продукты AWS, — например, VNet, VPN-шлюз и Load Balancer. Некоторые из них могут называться по-другому (как в случае с Route 53 и Azure DNS), но они предоставляют одни и те же услуги (рис. 11.1).

Сетевые продукты Azure и AWS имеют определенные различия с точки зрения возможностей; например, для распределения глобального трафика с помощью DNS в AWS используется тот же сервис Route 53, а в Azure для этого предусмотрен отдельный продукт, Traffic Manager. Если углубиться в эти предложения, можно найти некоторые различия в их использовании. Например, Azure Load Balancer поддерживает балансировку по сходству сеансов по умолчанию, тогда как в AWS Load Balancer эту возможность нужно настраивать дополнительно.

Но в целом высокоуровневые продукты и сервисы от Azure похожи на аналоги от AWS, с которыми мы уже познакомились. Это хорошая новость. Но есть и плохая: несмотря на похожие возможности, мы не можем точно продублировать сеть из одного провайдера в другом. Инструменты и детали реализации могут озадачить того, кто никогда не имел дела с Azure. При обсуждении продуктов в следующих разделах мы будем отмечать некоторые различия. Для начала же подготовим все необходимое для работы с Azure.

Подготовка к работе с Azure

Процесс регистрации в Azure не должен вызвать никаких затруднений. Рынок публичных облаков отличается острой конкуренцией, поэтому, чтобы привлечь

Networking

Area	AWS service	Azure service	Description
Cloud virtual networking	Virtual Private Cloud (VPC)	Virtual Network	Provides an isolated, private environment in the cloud. Users have control over their virtual networking environment, including selection of their own IP address range, creation of subnets, and configuration of route tables and network gateways.
Cross-premises connectivity	AWS VPN Gateway	Azure VPN Gateway	Connects Azure virtual networks to other Azure virtual networks, or customer on-premises networks (Site To Site). Allows end users to connect to Azure services through VPN tunneling (Point To Site).
DNS management	Route 53	Azure DNS	Manage your DNS records using the same credentials and billing and support contract as your other Azure services
	Route 53	Traffic Manager	A service that hosts domain names, plus routes users to Internet applications, connects user requests to datacenters, manages traffic to apps, and improves app availability with automatic failover.
Dedicated network	Direct Connect	ExpressRoute	Establishes a dedicated, private network connection from a location to the cloud provider (not over the Internet).
Load balancing	Network Load Balancer	Load Balancer	Azure Load Balancer load-balances traffic at layer 4 (TCP or UDP).
	Application Load Balancer	Application Gateway	Application Gateway is a layer 7 load balancer. It supports SSL termination, cookie-based session affinity, and round robin for load-balancing traffic.

Рис. 11.1. Сетевые сервисы Azure (источник: https://docs.microsoft.com/ru-ru/azure/architecture/aws-professional/services)

пользователей, Azure, как и AWS, предлагает множество услуг и акций. Последние предложения ищите на странице https://azure.microsoft.com/ru-ru/free/. На момент написания этой книги в Azure действовали выгодные акции для сервисов искусственного интеллекта и Kubernetes со множеством замечательных продуктов, доступных бесплатно на протяжении первых 12 месяцев или на постоянной основе (рис. 11.2).

После создания учетной записи можно посмотреть, какие сервисы доступны на портале Azure по адресу https://portal.azure.com (рис. 11.3).

Но прежде, чем запускать какие-либо сервисы, мы должны настроить способ оплаты. Для этого нужно добавить подписку (рис. 11.4).

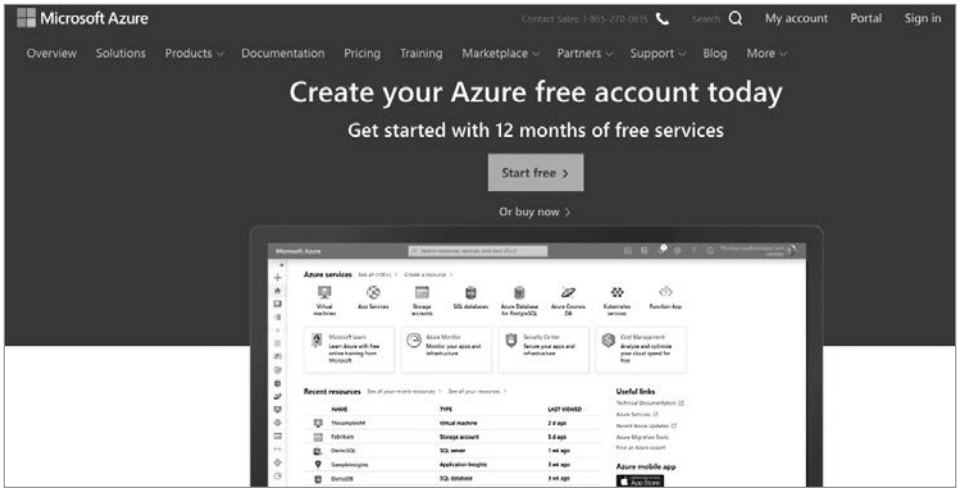


Рис.11.2. Портал Azure (источник: https://azure.microsoft.com/ru-ru/free/)

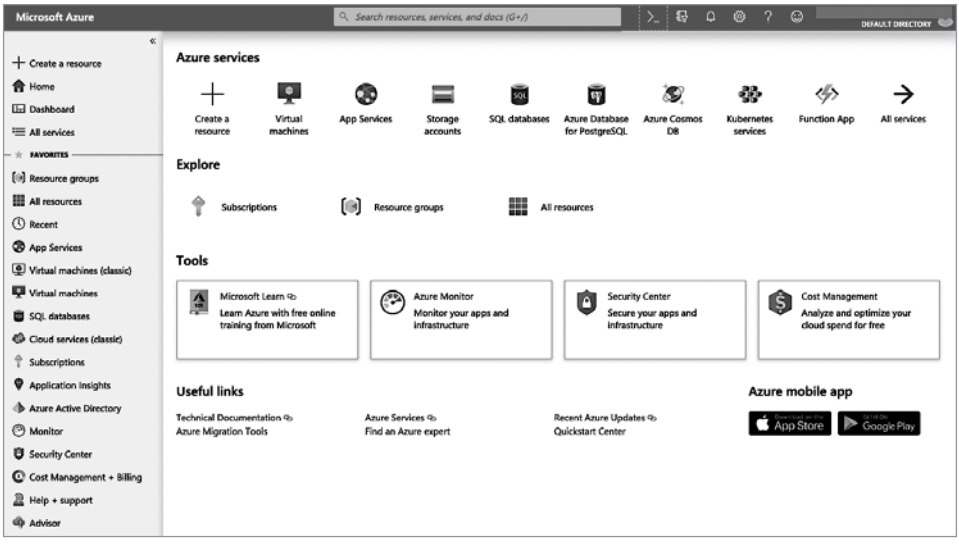


Рис. 11.3. Сервисы Azure

Советую выбрать тариф с оплатой за использованные ресурсы, который не требует предварительной оплаты и взятия на себя долгосрочных обязательств; при этом вы можете докупить разные уровни поддержки.

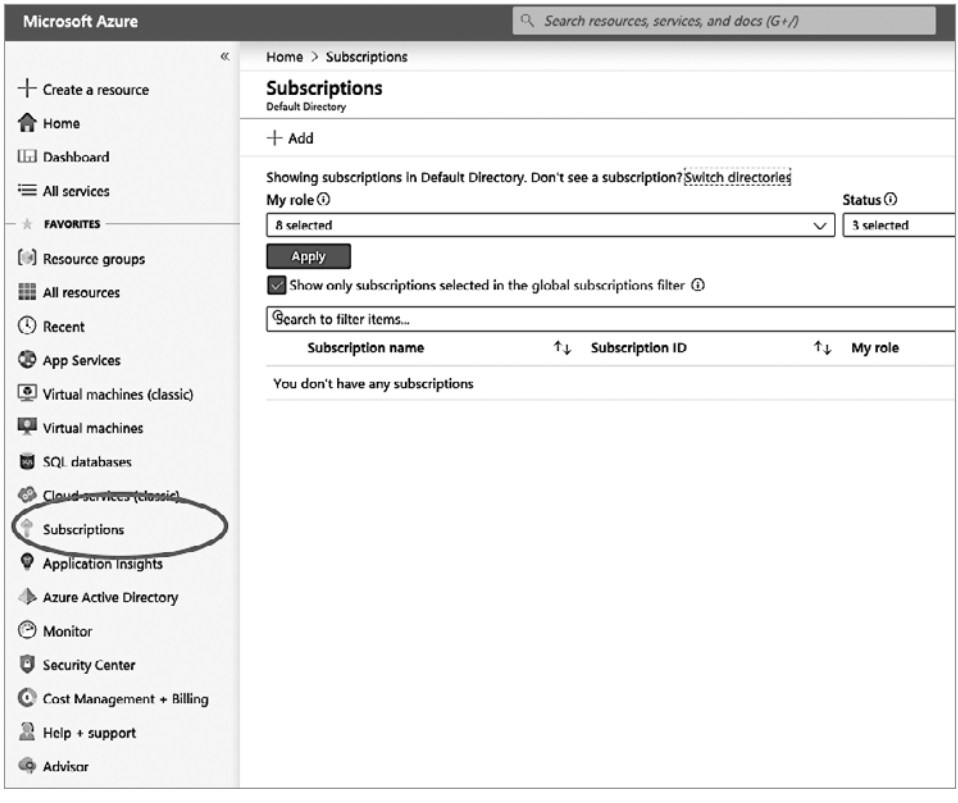


Рис. 11.4. Подписки Azure

После добавления подписки можно приступать к созданию и администрированию приложений в облаке Azure. Подробнее об этом — в следующем разделе.

Администрирование Azure и API

У Azure самый изящный и современный веб-интерфейс среди всех основных облачных провайдеров, включая AWS и Google Cloud. Его настройки, в том числе язык и регион, можно менять с помощью значка на верхней панели управления (рис. 11.5).

Управлять сервисами Azure можно разными способами: с помощью веб-интерфейса, Azure CLI, RESTful API и клиентских библиотек. Помимо графического интерфейса, в котором можно работать мышкой, Azure предоставляет

удобную командную оболочку Azure Cloud Shell. Ее можно запустить щелчком на значке в правом верхнем углу (рис. 11.6).

Рис. 11.5. Портал Azure на разных языках

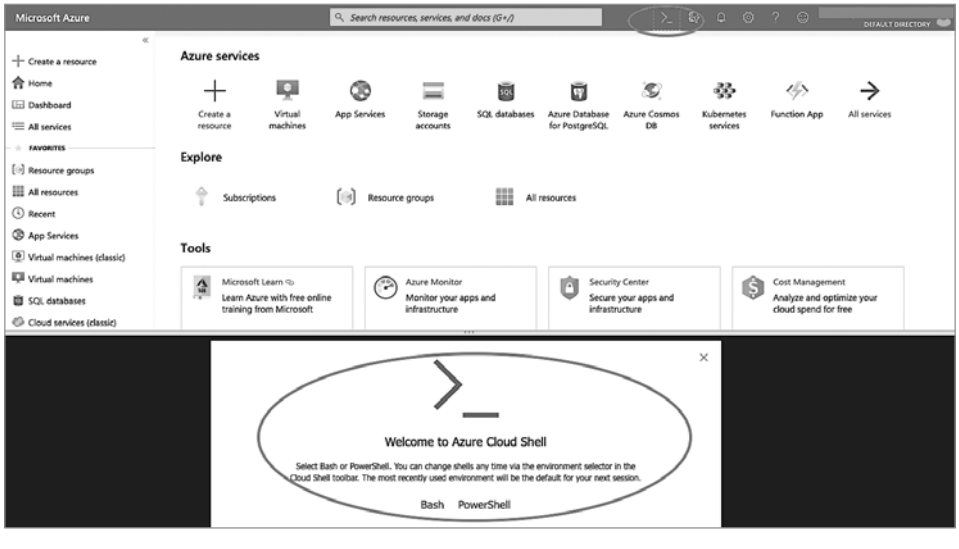


Рис. 11.6. Azure Cloud Shell

При первом запуске вам будет предложено выбрать между *Bash* и *PowerShell*. Позже вы сможете поменять свой выбор, но одновременно эти интерфейсы использовать нельзя (рис. 11.7).

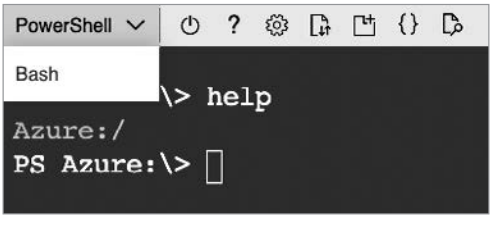


Рис. 11.7. Azure Cloud Shell с PowerShell

Лично я предпочитаю Bash, так как эта командная оболочка позволяет использовать предустановленные Azure CLI и Python SDK (рис. 11.8).

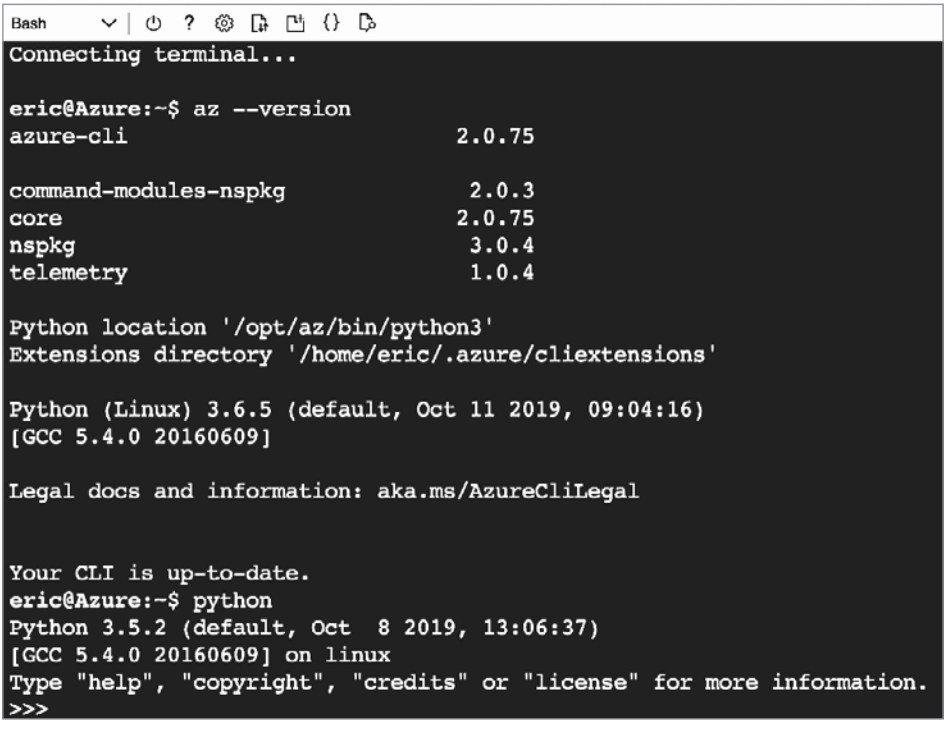


Рис. 11.8. Инструмент Azure AZ

Cloud Shell — очень удобный инструмент, так как он работает в браузере, и обращаться к нему можно практически откуда угодно. Он выдается каждой учетной записи и автоматически аутентифицируется в каждом сеансе, поэтому нам не нужно генерировать для него отдельный ключ. Но поскольку мы будем довольно часто использовать пакет Azure CLI, установим его локальную копию на управляющий хост:

```
(venv) $ curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
(venv) $ az --version
azure-cli                         2.0.75

command-modules-nspkg              2.0.3
core                              2.0.75
nspkg                              3.0.4
telemetry                          1.0.4
```

Установим туда же Azure Python SDK:

```
(venv) $ pip install azure
(venv) $ python
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import azure
>>> exit()
```

Страница «Azure для разработчиков на Python», https://docs.microsoft.com/ru-ru/azure/developer/python/, — исчерпывающий ресурс для начинающих работать с Azure на Python.

И приступим к знакомству с некоторыми субъектами-службами и запуску своих сервисов в Azure.

Субъекты-службы в Azure

В средствах автоматизации у Azure есть понятие субъектов-служб. Согласно рекомендациям по сетевой безопасности каждому человеку или инструменту выдается ровно такой уровень доступа, которого достаточно для выполнения задачи и не более того. Субъект-служба ограничивает ресурсы и уровень доступа в зависимости от ролей. В начале, чтобы проверить аутентификацию с помощью Python SDK, мы используем роль, которую для нас автоматически создал инструмент Azure CLI. Для получения токена выполните команду `az login`:

```
(venv) $ az login
To sign in, use a web browser to open the page https://microsoft.com/
devicelogin and enter the code <ваш код> to authenticate.
```

Перейдите по указанному URL и вставьте код, который выводится в консоли, чтобы войти в созданную вами учетную запись Azure (рис. 11.9).

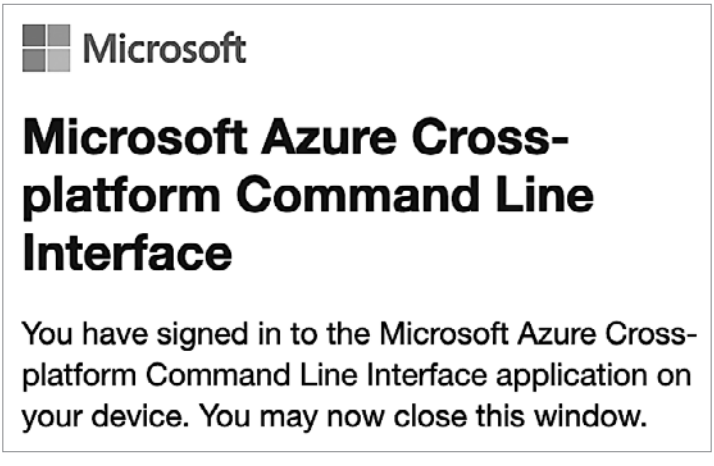


Рис. 11.9. Кросс-платформенный интерфейс командной строки Azure

Мы можем создать файл с учетными данными и поместить его в каталог Azure, который создается при установке пакета Azure CLI:

```
(venv) $ az ad sp create-for-rbac --sdk-auth > credentials.json
(venv) $ cat credentials.json
{
 "clientId": "<опущено>",
 "clientSecret": "<опущено>",
 "subscriptionId": "<опущено>",
 "tenantId": "<опущено>",
 "<опущено>"
}
(venv) echou@network-dev-2:~$ mv credentials.json ~/.azure/
```

Ограничим доступ к файлу `credentials.json` и экспортируем путь к нему в виде переменной окружения:

```
(venv) $ chmod 0600 ~/.azure/credentials.json
(venv) $ export AZURE_AUTH_LOCATION=~/.azure/credentials.json
```

Если открыть портал и перейти в раздел Access control (Управление доступом), выбрав в меню пункт Home ▸ Subscriptions ▸ Pay-As-You-Go ▸ Access control (Глав-

ная ▸ Подписки ▸ Оплата по мере использования ▸ Управление доступом), можно увидеть только что созданную роль (рис. 11.10).

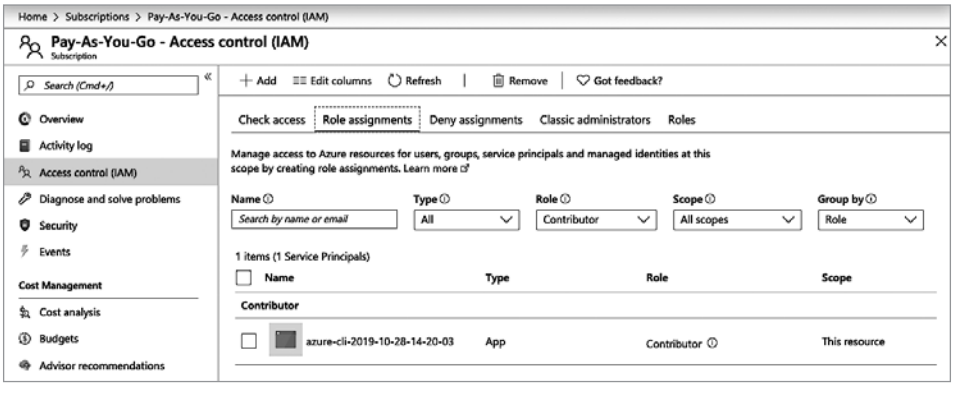


Рис. 11.10. Управление доступом для тарифа «Оплата по мере использования»

Воспользуемся простым сценарием `Chapter11_1_auth.py`, чтобы импортировать библиотеку для аутентификации клиента и сетевого администрирования:

```
from azure.common.client_factory import get_client_from_auth_file
from azure.mgmt.network import NetworkManagementClient

client = get_client_from_auth_file(NetworkManagementClient)
print("Network Management Client API Version: " + client.DEFAULT_API_
VERSION)
```

Отсутствие ошибок говорит об успешной аутентификации клиента на основе Python SDK:

```
(venv) $ python Chapter11_1_auth.py
Network Management Client API Version: 2018-12-01
```

При чтении документации Azure можно заметить, что примеры кода в основном написаны на PowerShell. В следующем разделе мы сравним PowerShell с Python.

Сравнение Python и PowerShell

Компания Microsoft разработала с нуля либо видоизменила, создав новые диалекты, множество языков программирования и фреймворков. Например, C#, .NET и PowerShell. Неудивительно, что .NET (с C#) и PowerShell имеют первоклассную поддержку в Azure. В документации Azure много ссылок на примеры

с PowerShell. На веб-форумах часто разгораются дискуссии о том, какой инструмент лучше подходит для администрирования ресурсов Azure: PowerShell или Python.

По состоянию на июль 2019 года предварительную версию PowerShell Core можно также запускать в операционных системах Linux и macOS (https://docs.microsoft.com/ru-ru/powershell/scripting/install/installing-powershell-core-on-linux?view=powershell-6). Но, учитывая то, что это не окончательный выпуск, могут иметь место программные ошибки и отсутствовать некоторые возможности.

Мы не будем спорить, какой язык лучше. Я не против использовать PowerShell, когда это необходимо (лично мне он кажется простым и понятным), и я согласен с тем, что Python SDK иногда отстает от него с точки зрения реализации последних возможностей Azure. Но поскольку Python — как минимум одна из причин, по которым вы приобрели эту книгу, наши примеры будут основаны на Python SDK и Azure CLI.

В самом начале пакет Azure CLI представлял собой набор модулей PowerShell для Windows, а на других платформах имел вид утилиты командной строки, написанной на Node.js. Но с ростом популярности он превратился в обертку вокруг Azure Python SDK; подробнее об этом читайте в статье на сайте Python.org: https://www.python.org/success-stories/building-an-open-source-and-cross-platform-azure-cli-with-python/.

В примерах дальше будем также использовать Azure CLI. Но можете не сомневаться: все, что есть в Azure CLI, доступно и в Python SDK (на случай, если вы хотите работать с этими функциями непосредственно из Python).

Итак, мы обсудили администрирование Azure и сопутствующие API. Перейдем к глобальной инфраструктуре Azure.

Глобальная инфраструктура Azure

Подобно AWS, глобальная инфраструктура Azure состоит из регионов, *зон доступности (availability zones, AZ)* и граничных узлов. На момент написания этой книги в Azure насчитывалось 54 региона и более 150 граничных узлов. Об этом сообщается на странице продукта, https://azure.microsoft.com/ru-ru/global-infrastructure/ (рис. 11.11).

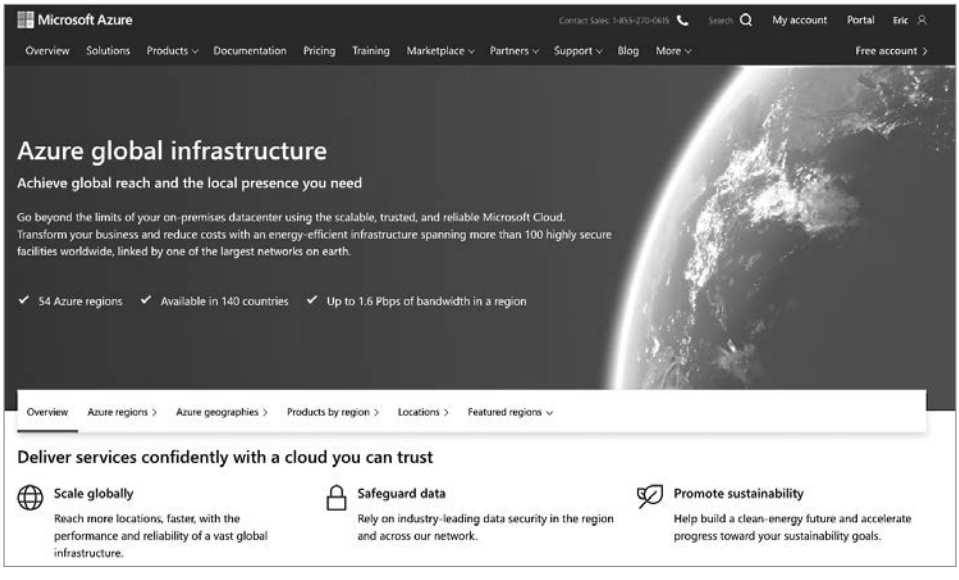


Рис. 11.11. Глобальная инфраструктура Azure
(источник: https://azure.microsoft.com/ru-ru/global-infrastructure/)

Как и в AWS, цены и набор доступных сервисов в Azure могут различаться в зависимости от региона. Мы можем продублировать один и тот же сервис в нескольких зонах доступности. Но, в отличие от AWS, не все регионы в Azure содержат зоны доступности и не все продукты их поддерживают. На самом деле зоны доступности в Azure появились только в 2018 году и поддерживаются только в отдельных регионах.

Это следует учитывать при выборе региона. Я рекомендую выбирать регионы с зонами доступности, такие как West US 2, Central US и East US 1. В отсутствие зон доступности придется копировать сервисы между разными регионами, обычно в пределах одной географической области. О географии Azure поговорим ниже.

На странице глобальной инфраструктуры Azure регионы с зонами доступности отмечены звездочками.

В отличие от AWS, регионы Azure группируются по категориям более высокого уровня — географическим областям. Географическая область — это отдельный рынок, который обычно состоит из двух или более регионов. Помимо уменьшенной задержки и лучшего сетевого соединения, дублирование

сервисов и данных в разных регионах в одной географической области обеспечивает соблюдение правовых норм. Возьмем, к примеру, регионы Германии. Если нам нужно запустить сервисы на немецком рынке, то по закону мы обязаны обеспечить хранение данных строго в государственных границах, однако ни один из немецких регионов не поддерживает зоны доступности. Нам пришлось бы дублировать данные между регионами внутри одной географической области, такими как Germany North, Germany Northeast, Germany West Central и т. д.

Я, как правило, отдаю предпочтение регионам с зонами доступности, чтобы сохранять некоторую согласованность между облачными провайдерами. Выбрав регион, который лучше подходит для наших задач, мы можем приступать к созданию VNet.

Виртуальные сети Azure

Сетевой инженер, имеющий дело с облаком Azure, большую часть времени проводит за работой с виртуальными сетями (VNet). По аналогии с традиционными сетями, которые мы создаем в наших дата-центрах, это фундаментальные компоненты наших частных сетей в Azure. VNet позволяет нашим виртуальным машинам сообщаться друг с другом, с интернетом и с другими локально размещенными сетями с помощью VPN или ExpressRoute.

Создадим на портале нашу первую виртуальную сеть. Сначала перейдем на страницу Create a Resource ▸ Networking ▸ Virtual network (Создать ресурс ▸ Сети ▸ Виртуальная сеть) (рис. 11.12).

Каждая виртуальная сеть ограничена одним регионом, и в ней можно создать несколько подсетей. Позже вы увидите, как соединить сети VNet из разных регионов, используя VNet-пиринг.

Создадим нашу первую виртуальную сеть со следующими характеристиками:

```
Name: WEST-US-2_VNet_1
Address space: 192.168.0.0/23
Subscription: <pick your subscription>
Resource group: <click on new> -> 'Mastering-Python-Networking'
Location: West US 2
Subnet name: WEST-US-2_VNet_1_Subnet_1
Address range: 192.168.1.0/24
DDoS protection: Basic
Service endpoints: Disabled
Firewall: Disabled
```

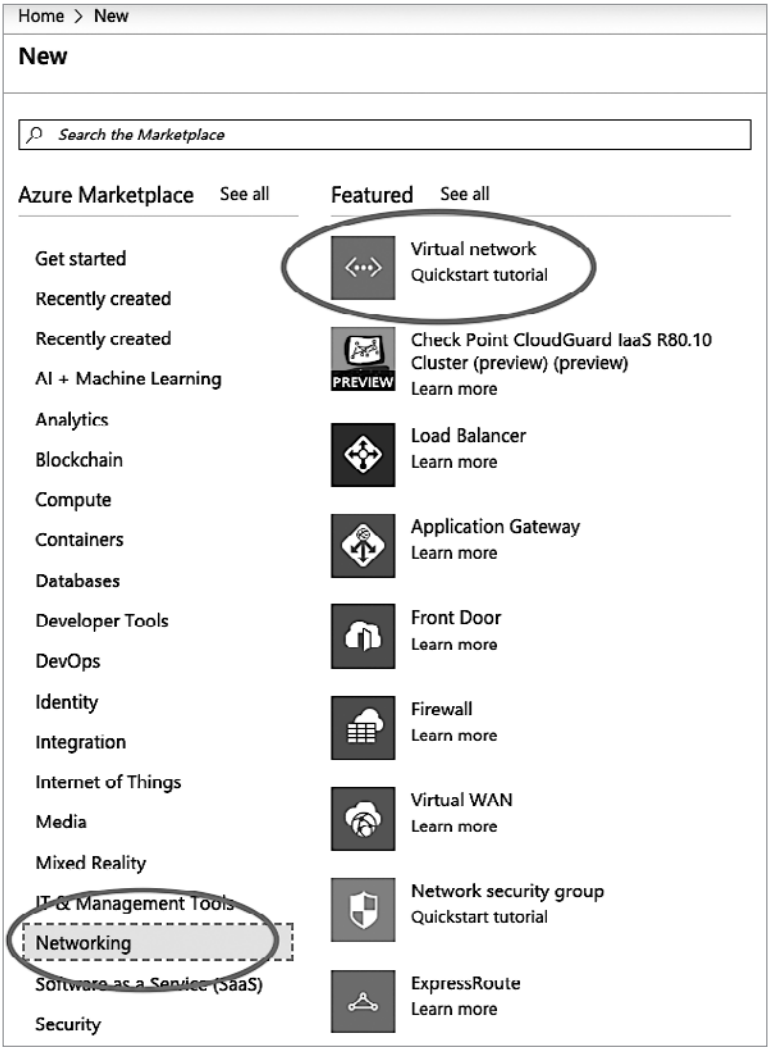


Рис. 11.12. Azure VNet

Вот снимок экрана с заполненными полями. Если вы пропустите какое-то обязательное поле, оно будет выделено красным цветом. Когда закончите, нажмите кнопку Create (Создать) (рис. 11.13).

Перейдем к созданному ресурсу, выбрав в меню пункт Home ▸ Resource groups ▸ Mastering-Python-Networking (Главная ▸ Группы ресурсов ▸ Mastering-Python-Networking) (рис. 11.14).

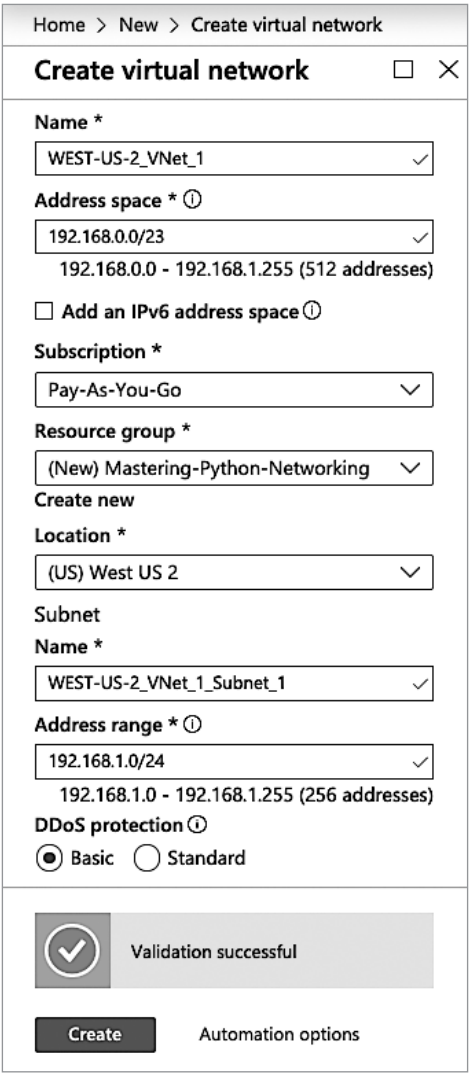


Рис. 11.13. Создание Azure VNet

Поздравляю, вы только что создали свою первую виртуальную сеть в облаке Azure! Конечно, чтобы приносить какую-то пользу, она должна взаимодействовать с окружающим миром. Как этого добиться, поговорим в следующем подразделе.

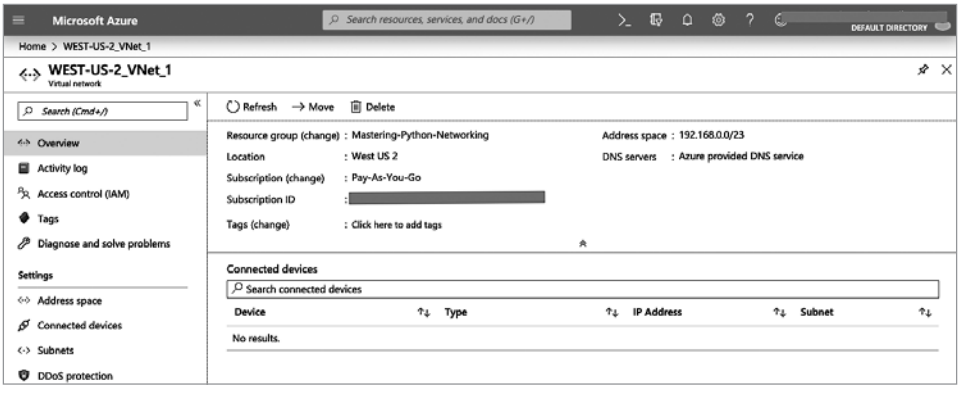


Рис. 11.14. Обзор Azure VNet

Доступ к интернету

Все ресурсы внутри VNet по умолчанию могут направлять исходящий трафик в интернет; нам не нужно добавлять NAT-шлюз, как мы это делали в AWS. Для входящего трафика требуется публичный IP-адрес; его можно назначить либо непосредственно виртуальной машине, либо балансировщику нагрузки. Чтобы посмотреть, как это работает, создадим в нашей сети виртуальные машины.

Первую виртуальную машину создадим с помощью меню Home ▸ Resource groups ▸ Mastering-Python-Networking ▸ New ▸ Create a virtual machine (Главная ▸ Группы ресурсов ▸ Mastering-Python-Networking ▸ Новая ▸ Создать виртуальную машину) (рис. 11.15).

В качестве виртуальной машины я выберу Ubuntu Server 16.04 LTS и назову ее `myMPN-VM1`. Размещу ее в регионе `West US 2`. Аутентификация будет осуществляться по паролю, но при этом я разрешу входящие SSH-соединения.

Остальные параметры можно не менять. Поместим нашу ВМ в подсеть, созданную выше, и назначим ей новый публичный IP-адрес (рис. 11.16).

Создав, мы можем зайти на нее по SSH, используя публичный IP-адрес и свою учетную запись. У ВМ всего один сетевой интерфейс, подключенный к нашей частной подсети; он также отображается в публичный IP-адрес, который Azure назначает автоматически. Преобразование между внешним и внутренним IP-адресами происходит в Azure автоматически.

```
echou@myMPN-VM1:~$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0d:3a:6e:14:f3
          inet addr:192.168.1.4  Bcast:192.168.1.255  Mask:255.255.255.0
<опущено>
echou@myMPN-VM1:~$ ping -c 1 www.google.com
PING www.google.com (172.217.14.228) 56(84) bytes of data.
64 bytes from sea30s02-in-f4.1e100.net (172.217.14.228): icmp_seq=1
ttl=51 time=4.88 ms

--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 4.888/4.888/4.888/0.000 ms
```

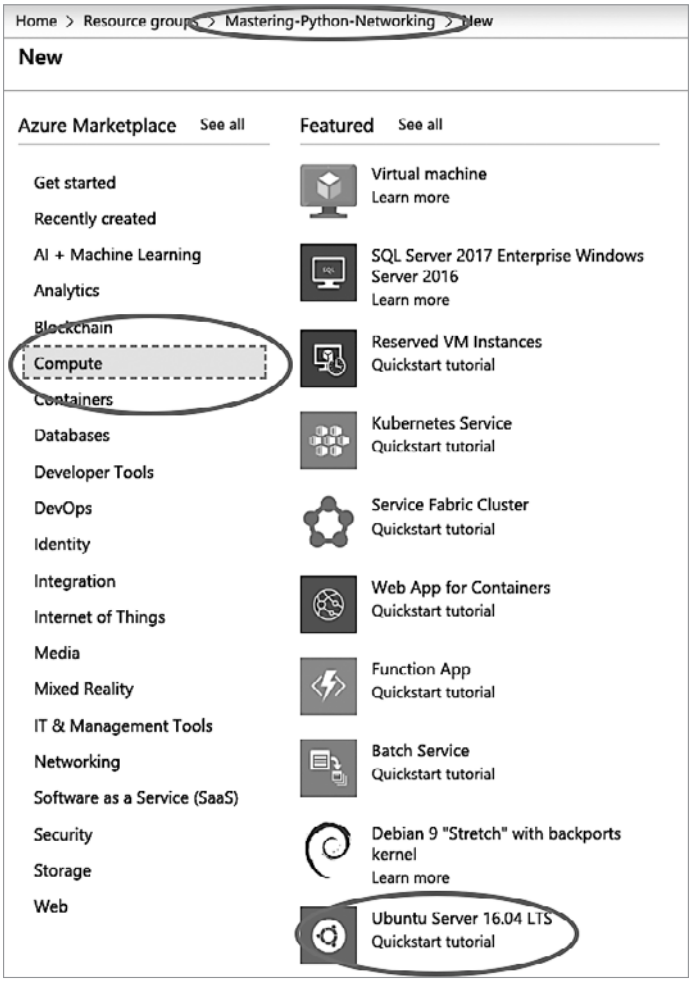


Рис. 11.15. Создание ВМ в Azure

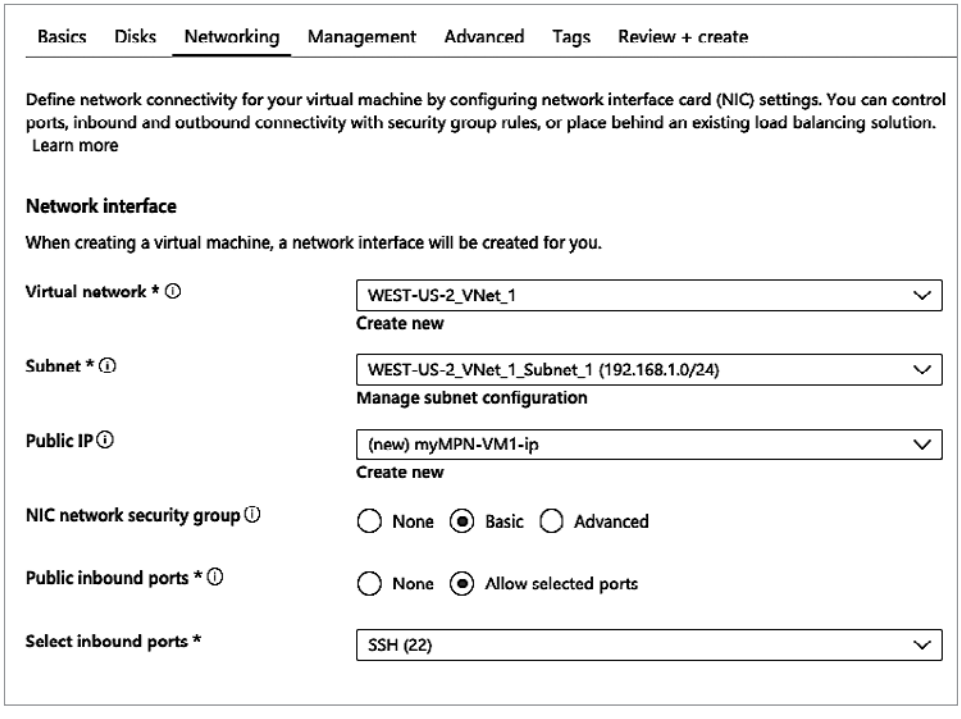


Рис. 11.16. Сетевой интерфейс Azure

Повторим тот же процесс и создадим вторую ВМ с именем `myMPN-VM2`. Откроем для нее входящий трафик по SSH, но не будем выделять ей публичный IP-адрес (рис. 11.17).

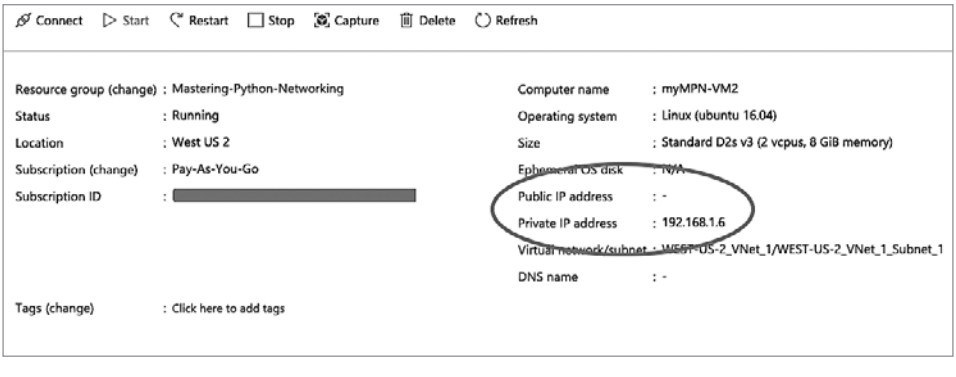


Рис. 11.17. IP-адреса для ВМ в Azure

После создания мы можем зайти в myMPN-VM2 по SSH из myMPN-VM1, используя внутренний IP-адрес:

```
echou@myMPN-VM1:~$ ssh echou@192.168.1.6
echou@192.168.1.6's password:
<опущено>
0 updates are security updates.Last login: Tue Oct 29 01:05:44 2019 from
192.168.1.4
echou@myMPN-VM2:~$
```

Проверим интернет-соединение, попробовав обновить репозитории пакетов с помощью apt:

```
echou@myMPN-VM2:~$ sudo apt update
Hit:1 http://azure.archive.ubuntu.com/ubuntu xenial InRelease
Get:2 http://azure.archive.ubuntu.com/ubuntu xenial-updates InRelease
[109 kB]
Get:3 http://azure.archive.ubuntu.com/ubuntu xenial-backports InRelease
[107 kB]
Hit:4 http://security.ubuntu.com/ubuntu xenial-security InRelease
Fetched 216 kB in 0s (720 kB/s)
```

Итак, наша ВМ находится внутри VNet и имеет доступ к интернету. Теперь в нашей сети можно создать дополнительные ресурсы.

Создание сетевых ресурсов

Рассмотрим пример создания новых сетевых ресурсов с помощью Python SDK. В следующем сценарии, Chapter11_2_network_resources.py, используется класс NetworkManagementClient (https://docs.microsoft.com/en-us/python/api/azure-mgmt-network/azure.mgmt.network.networkmanagementclient?view=azure-python), который мы определили в предыдущем примере, и API-вызов subnet.create_or_update для создания подсети 192.168.0.128/25 внутри VNet:

```
GROUP_NAME = 'Mastering-Python-Networking'
LOCATION = 'westus2'

def create_subnet(network_client):
    subnet_params = {
        'address_prefix': '192.168.0.128/25'
    }
    creation_result = network_client.subnets.create_or_update(
        GROUP_NAME,
        'WEST-US-2_VNet_1',
        'WEST-US-2_VNet_1_Subnet_2',
```

```
            subnet_params
        )

        return creation_result.result()

    creation_result = create_subnet(network_client)
```

Запустив сценарий, мы получим следующее сообщение с результатами создания ресурса:

```
(venv) $ python3 Chapter11_2_subnet.py
{'additional_properties': {'type': 'Microsoft.Network/virtualNetworks/
subnets'}, 'id': '/subscriptions/<опущено>/resourceGroups/Mastering-
Python-Networking/providers/Microsoft.Network/virtualNetworks/WESTUS-
2_VNet_1/subnets/WEST-US-2_VNet_1_Subnet_2', 'address_prefix':
'192.168.0.128/25', 'address_prefixes': None, 'network_security_group':
None, 'route_table': None, 'service_endpoints': None, 'service_endpoint_
policies': None, 'interface_endpoints': None, 'ip_configurations': None,
'ip_configuration_profiles': None, 'resource_navigation_links': None,
'service_association_links': None, 'delegations': [], 'purpose': None,
'provisioning_state': 'Succeeded', 'name': 'WEST-US-2_VNet_1_Subnet_2',
'etag': 'W/"<опущено>"'}
```

Новую подсеть можно также увидеть на портале (рис. 11.18).

Рис. 11.18. Подсети внутри Azure VNet

Больше примеров использования Python SDK ищите на страницах https://docs.microsoft.com/en-us/azure/virtual-machines/windows/python (для Windows) и https://github.com/Azure-Samples/virtual-machines-python-manage/blob/master/example.py (для Linux Ubuntu).

Если мы создадим ВМ в новой подсети, она сможет взаимодействовать с хостами в той же VNet, даже если они находятся в других подсетях, благодаря скрытому маршрутизатору, который мы уже видели при обсуждении AWS.

Для взаимодействия с другими сервисами Azure в VNet есть дополнительные инструменты. Рассмотрим их.

Конечные точки сервисов для VNet

Конечные точки сервисов позволяют подключать виртуальную сеть напрямую к другим сервисам Azure. Благодаря этому трафик между VNet и отдельным сервисом остается в пределах сети Azure. Конечной точке нужно назначить определенный сервис, который находится в том же регионе, что и VNet.

Их можно настраивать в веб-интерфейсе, выбирая сервис и подсеть (рис. 11.19).

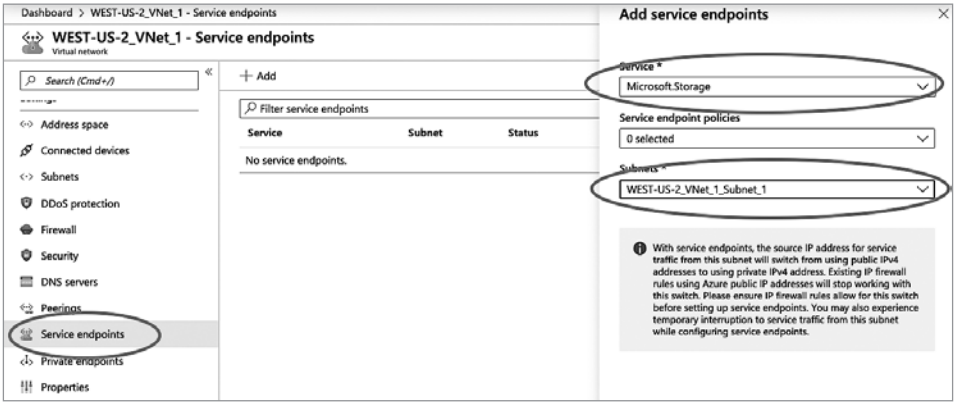


Рис. 11.19. Конечные точки сервисов Azure

Строго говоря, для организации взаимодействия между виртуальными машинами внутри VNet и сервисом не обязательно создавать конечные точки. Каждая ВМ может обращаться к сервису через привязанный к нему публичный IP-адрес, и мы можем, используя наши сетевые правила, разрешить только нужные нам IP-адреса. Однако с помощью этого инструмента можно получить доступ к ресурсам по их внутренним IP-адресам, не пропуская трафик через интернет.

VNet-пиринг

Как уже упоминалось в начале раздела, каждая виртуальная сеть ограничена одним регионом. Для соединения разных виртуальных сетей можно использовать VNet-пиринг. Используем следующие две функции в файле `Chapter11_3_vnet.py`, чтобы создать VNet в регионе `US-East`:

```
<опущено>
def create_vnet(network_client):
    vnet_params = {
        'location': LOCATION,
        'address_space': {
            'address_prefixes': ['10.0.0.0/16']
        }
    }
    creation_result = network_client.virtual_networks.create_or_update(
        GROUP_NAME,
        'EAST-US_VNet_1',
        vnet_params
    )
    return creation_result.result()
<опущено>
def create_subnet(network_client):
    subnet_params = {
        'address_prefix': '10.0.1.0/24'
    }
    creation_result = network_client.subnets.create_or_update(
        GROUP_NAME,
        'EAST-US_VNet_1',
        'EAST-US_VNet_1_Subnet_1',
        subnet_params
    )

    return creation_result.result()
```

Разрешим VNet-пиринг через двунаправленное соединение между двумя VNet. До сих пор мы использовали Python SDK, но сейчас для расширения кругозора рассмотрим пример с Azure CLI.

Получим название и идентификатор VNet командой `az network vnet list`:

```
(venv) $ az network vnet list
<опущено>
"id": "/subscriptions/<опущено>/resourceGroups/Mastering-Python-Networking/
providers/Microsoft.Network/virtualNetworks/EAST-US_VNet_1",
    "location": "eastus",
    "name": "EAST-US_VNet_1"
```

```
<опущено>
"id": "/subscriptions/<опущено>/resourceGroups/Mastering-Python-Networking/
providers/Microsoft.Network/virtualNetworks/WEST-US-2_VNet_1",
    "location": "westus2",
    "name": "WEST-US-2_VNet_1"
<опущено>
```

Проверим наличие VNet-пиринга для нашей виртуальной сети в `West US 2`:

```
(venv) $ az network vnet peering list -g "Mastering-Python-Networking"
--vnet-name WEST-US-2_VNet_1
[]
```

Запустим пиринг из `West US` в `East US` и повторим тот же процесс в обратном направлении:

```
(venv) $ az network vnet peering create -g "Mastering-Python-
Networking" -n WestUSToEastUS --vnet-name WEST-US-2_VNet_1 --remotevnet
"/subscriptions/<опущено>/resourceGroups/Mastering-Python-Networking/
providers/Microsoft.Network/virtualNetworks/EAST-US_VNet_1"

(venv) $ az network vnet peering create -g "Mastering-Python-
Networking" -n EastUSToWestUS --vnet-name EAST-US_VNet_1 --remote-vnet
"/subscriptions/b7257c5b-97c1-45ea-86a7-872ce8495a2a/resourceGroups/
Mastering-Python-Networking/providers/Microsoft.Network/virtualNetworks/
WEST-US-2_VNet_1"
```

Если теперь повторить проверку, то мы увидим, что связь с виртуальными сетями успешно установлена:

```
(venv) $ az network vnet peering list -g "Mastering-Python-Networking"
--vnet-name "WEST-US-2_VNet_1"
[
  {
    "allowForwardedTraffic": false,
    "allowGatewayTransit": false,
    "allowVirtualNetworkAccess": false,
    "etag": "W/\"<опущено>\"",
    "id": "/subscriptions/<опущено>/resourceGroups/Mastering-Python-
    Networking/providers/Microsoft.Network/virtualNetworks/WEST-US-2_VNet_1/
    virtualNetworkPeerings/WestUSToEastUS",
    "name": "WestUSToEastUS",
    "peeringState": "Connected",
    "provisioningState": "Succeeded",
    "remoteAddressSpace": {
      "addressPrefixes": [
        "10.0.0.0/16"
      ]
    },
<опущено>
```

Пиринг можно также проверить на портале Azure (рис. 11.20).

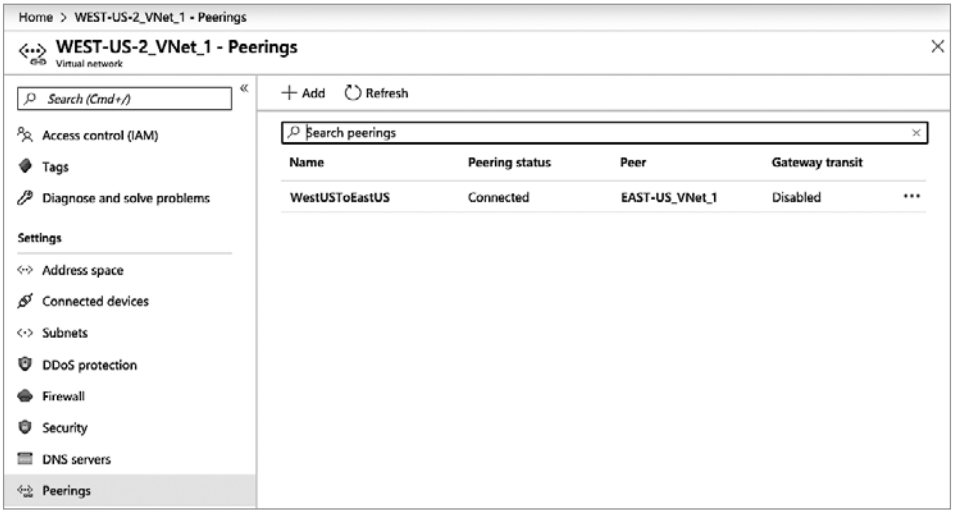


Рис. 11.20. VNet-пиринг в Azure

Итак, у нас есть несколько хостов, виртуальных сетей и настроенный пиринг между ними. Теперь посмотрим, как реализуется маршрутизация в Azure.

Маршрутизация в виртуальных сетях

Мне как сетевому инженеру всегда было немного некомфортно от скрытых маршрутов облачных провайдеров. В традиционных сетях нам приходится подключать сетевые кабели. Назначать IP-адреса, настраивать маршрутизацию, обеспечивать безопасность и следить за тем, чтобы все работало. Это иногда хлопотно, но зато мы контролируем каждый пакет и маршрут. В виртуальных облачных сетях все это, естественно, уже реализовано самим провайдером, и настройка некоторой оверлейной сети должна выполняться автоматически, чтобы хосты могли работать сразу после загрузки. Мы это уже видели.

Azure выполняет маршрутизацию в виртуальных сетях немного иначе, чем AWS. В предыдущей главе мы имели дело с таблицей маршрутизации, реализованной на уровне VPC. Но, открыв портал Azure и перейдя к настройкам VNet, мы не найдем там аналогичной таблицы.

Если еще углубиться в настройки подсети, можно заметить раскрывающийся список Route table (Таблица маршрутизации), но в ней выбрано значение None (Нет) (рис. 11.21).

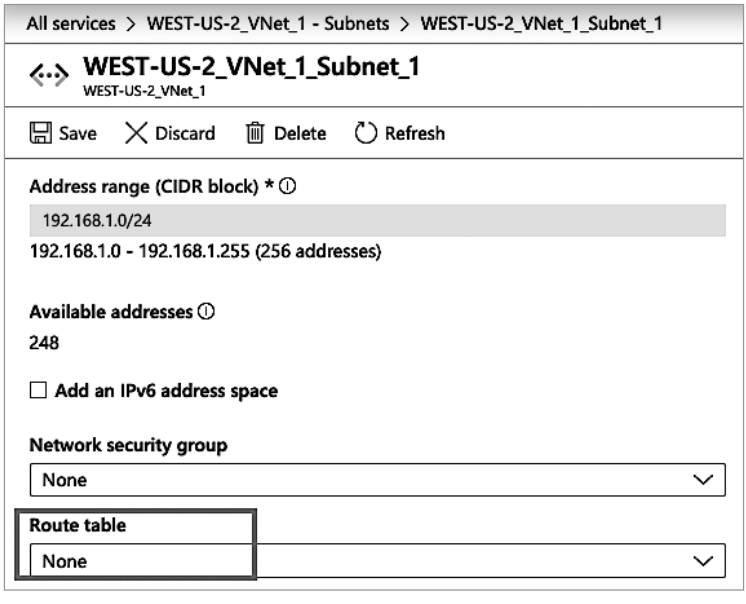


Рис. 11.21. Таблица маршрутизации для подсети в Azure

Как же таблица маршрутизации может быть пустой, если хосты в этой подсети имеют выход в интернет? Где просмотреть маршруты, сконфигурированные виртуальной сетью Azure? Оказывается, маршрутизация реализована на самом хосте на уровне NIC. Ее можно посмотреть на странице All Services ▸ Virtual Machines ▸ myNPM-VM1 ▸ Networking ▸ Topology (Все сервисы ▸ Виртуальные машины ▸ myNPM-VM1 ▸ Сеть ▸ Топология) (рис. 11.22).

Сеть показана на уровне NIC, где каждый контроллер соединяется с VNet с северной стороны, а с другими ресурсами, такими как ВМ, *сетевая группа безопасности (Network Security Group, NSG)* и IP-адрес — с южной. Ресурсы назначаются динамически; в момент, когда я делал этот снимок экрана, у меня была запущена только одна ВМ, `myMPN-VM1`, поэтому IP-адрес подключен только к ней, а к другим ВМ подключены только NSG.

Сетевые группы безопасности будут рассмотрены в следующем разделе.

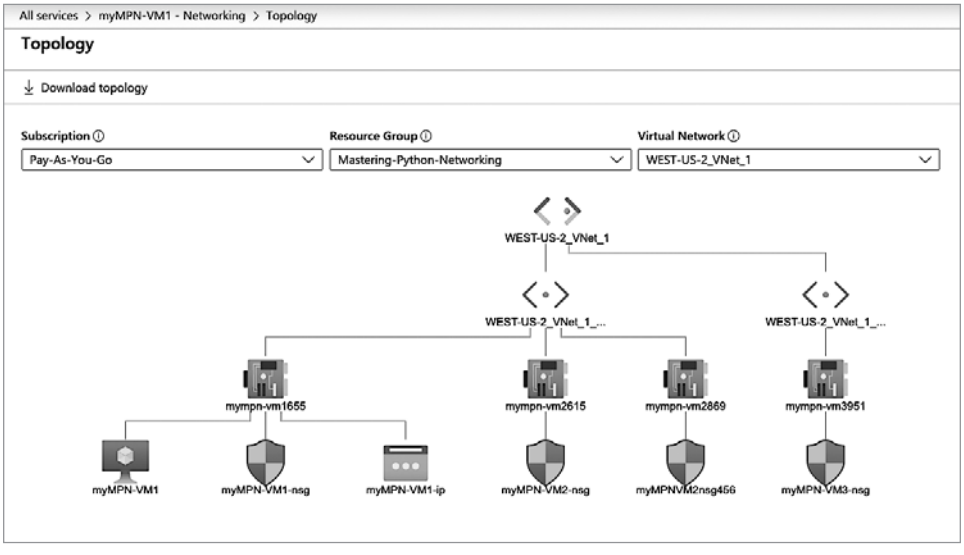


Рис. 11.22. Топология сети в Azure

Если щелкнуть на NIC (в нашей топологии это mympn-vm1655), мы увидим его настройки. В разделе Support + troubleshooting (Поддержка + решение проблем) есть ссылка Effective routes (Действующие маршруты), перейдя по ней, можно ознакомиться с текущими маршрутами, настроенными в контроллере (рис. 11.23).

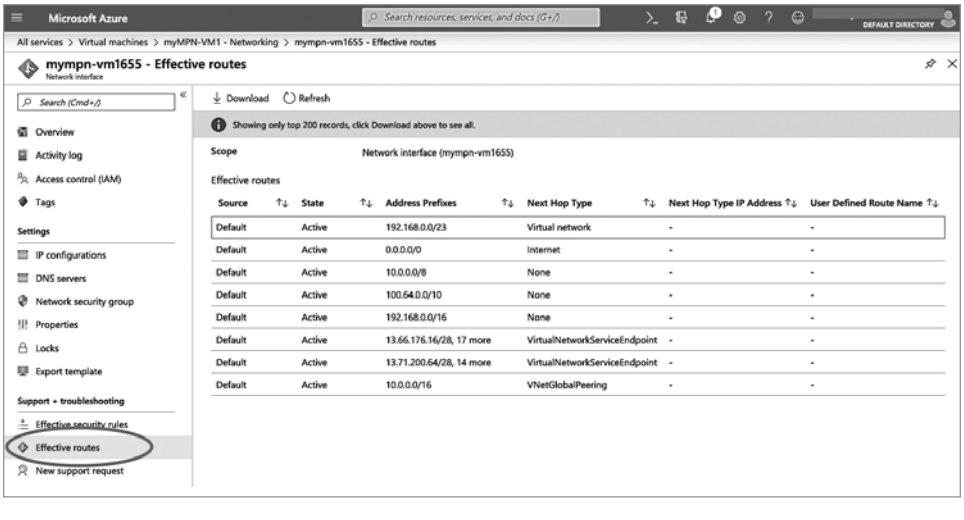


Рис. 11.23. Действующие маршруты VNet в Azure

Чтобы автоматизировать этот процесс, можно найти имя NIC с помощью Azure CLI и затем показать таблицу маршрутизации:

```
(venv) $ az vm show --name myMPN-VM1 --resource-group 'Mastering-Python-
Networking'
<опущено>
"networkProfile": {
    "networkInterfaces": [
      {
        "id": "/subscriptions/<опущено>/resourceGroups/Mastering-Python-
    Networking/providers/Microsoft.Network/networkInterfaces/mympn-vm1655",
        "primary": null,
        "resourceGroup": "Mastering-Python-Networking"
      }
    ]
  }
<опущено>
(venv) $ az network nic show-effective-route-table --name mympn-vm1655
--resource-group "Mastering-Python-Networking"
{
  "nextLink": null,
  "value": [
    {
      "addressPrefix": [
        "192.168.0.0/23"
      ],
<опущено>
```

Отлично! Одна загадка разгадана. Но что это за соседние транзитные участки в таблице маршрутизации? С этим вопросом можно обратиться к статье о маршрутизации трафика в VNet: https://docs.microsoft.com/ru-ru/azure/virtual-network/virtual-networks-udr-overview. Несколько важных замечаний.

- Маршруты, отмеченные в источнике как Default (по умолчанию), — системные, и их нельзя убрать. Но зато их можно переопределять с помощью своих маршрутов.
- Соседние транзитные участки — это маршруты внутри пользовательской VNet. В нашем случае это не просто подсеть, а сеть `192.168.0.0/23`.
- Трафик, направленный на соседний транзитный участок типа None (Нет), теряется, как и в случае с маршрутами, ведущими к интерфейсу Null.
- Тип соседнего транзитного участка VNetGlobalPeering — это то, что создается при настройке пиринга между виртуальными сетями.
- Тип соседнего транзитного участка VirtualNetworkServiceEndpoint создается при включении конечных точек сервисов в нашей VNet. Публич-

ный IP-адрес управляется самим провайдером и время от времени меняется.

Как переопределить маршруты по умолчанию? Можно создать таблицу маршрутизации и назначить ее нашим подсетям. Azure выбирает маршруты в следующем порядке:

- маршруты, определенные пользователем;
- BGP-маршруты (из межсайтового VPN-соединения или ExpressRoute);
- системные маршруты.

Таблицу маршрутизации можно создать в разделе Networking (Сети) (рис. 11.24).

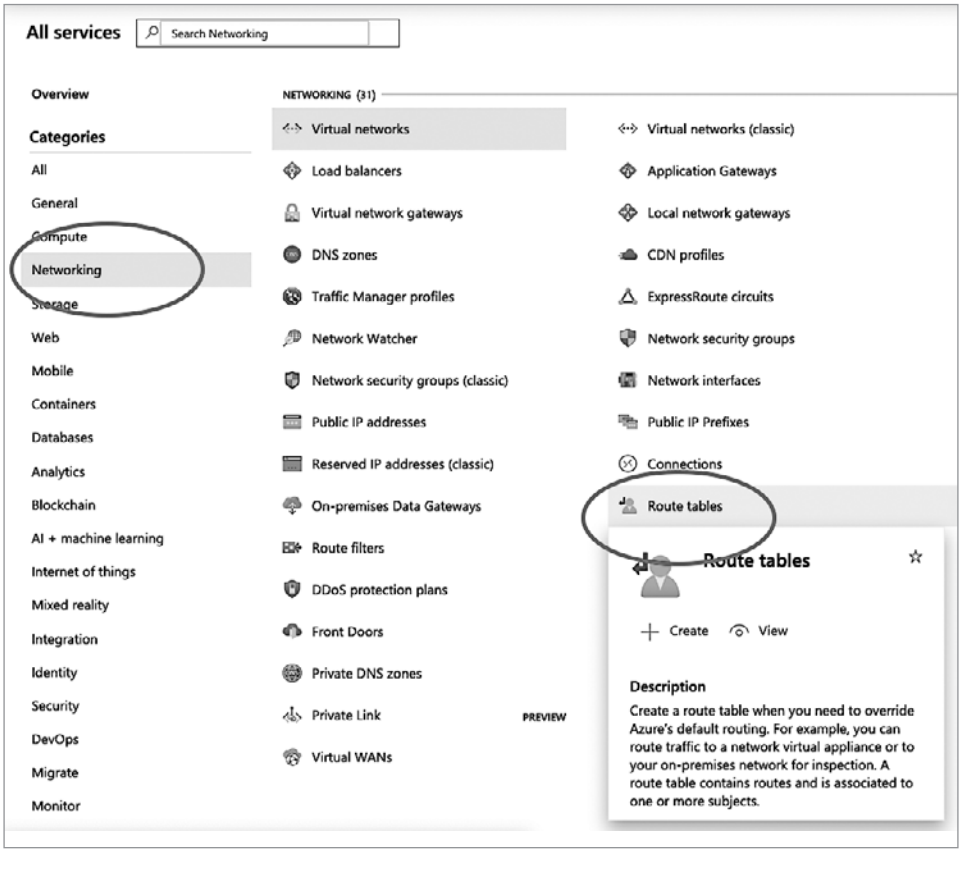


Рис. 11.24. Таблицы маршрутизации для VNet в Azure

Мы также можем создать таблицу маршрутизации с маршрутом и назначить ее подсети с помощью Azure CLI:

```
(venv) $ az network route-table create --name TempRouteTable --resource
"Mastering-Python-Networking"
(venv) $ az network route-table route create -g "Mastering-Python-
Networking" --route-table-name TempRouteTable -n TempRoute --next-hop-type
VirtualAppliance --address-prefix 172.31.0.0/16 --next-hop-ip-
address 10.0.100.4
(venv) $ az network vnet subnet update -g "Mastering-Python-Networking"
-n WEST-US-2_Vnet_1_Subnet_1 --vnet-name WEST-US-2_VNet_1 --route-table
TempRouteTable
```

Рассмотрим основной механизм безопасности в виртуальной сети — NSG.

Сетевые группы безопасности

Безопасность в VNet в основном обеспечивается за счет NSG. Так же как в традиционных списках доступа или правилах брандмауэра, каждое правило безопасности относится только к одному направлению трафика. Например, чтобы хост `A` в подсети `1` мог свободно взаимодействовать с хостом `B` в подсети `2`, нам нужно реализовать необходимые правила на обоих хостах и для входящего/исходящего соединения.

Как мы видели в предыдущих примерах, NSG можно привязать к NIC или к подсети, поэтому мы должны понимать, на каком уровне обеспечивается безопасность. В целом на уровне хоста следует реализовывать более ограничивающие правила, а на уровне подсети — более свободные. Похожий подход применяется и в традиционных сетях.

При создании наших ВМ мы разрешили входящие SSH-соединения с TCP-портом 22. Рассмотрим группу безопасности, которая была создана для нашей первой ВМ, `myMPN-VM1-nsg` (рис. 11.25).

Отмечу несколько моментов:

- правила, реализованные системой, имеют высокий уровень приоритета, от 65 000 и выше;
- виртуальные сети по умолчанию могут свободно сообщаться друг с другом в обоих направлениях;
- для внутренних хостов по умолчанию открыт доступ к интернету.

Откроем портал и реализуем входящее правило в существующей группе NSG (рис. 11.26).

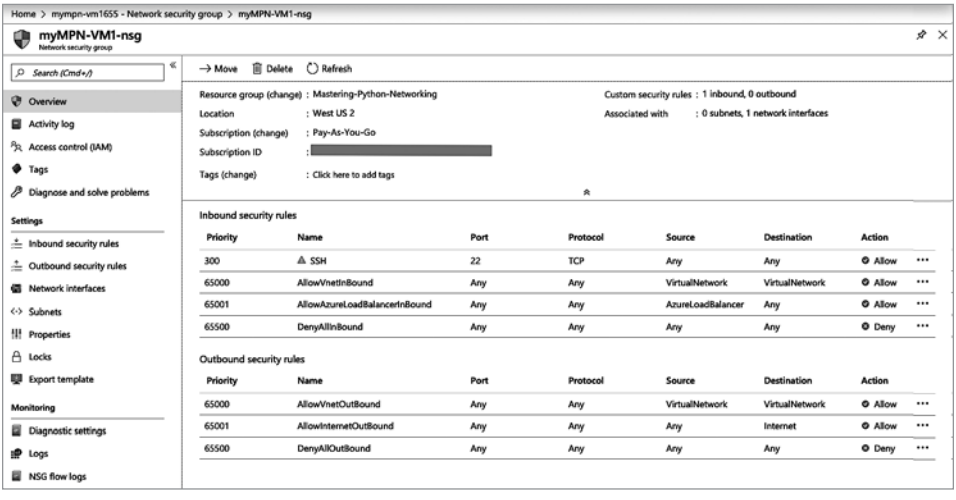


Рис. 11.25. NSG для виртуальной сети в Azure

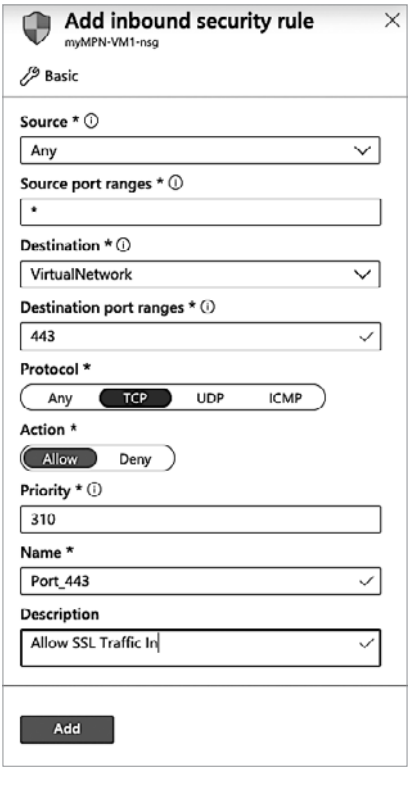


Рис. 11.26. Правило безопасности в Azure

Группу безопасности и правила для нее можно создать и с помощью Azure CLI:

```
(venv) $ az network nsg create -g "Mastering-Python-Networking" -n
TestNSG
(venv) $ az network nsg rule create -g "Mastering-Python-Networking"
--nsg-name TestNSG -n Allow_SSH --priority 150 --direction Inbound
--source-address-prefixes Internet --destination-port-ranges 22 --access
Allow --protocol Tcp --description "Permit SSH Inbound"
(venv) $ az network nsg rule create -g "Mastering-Python-Networking"
--nsg-name TestNSG -n Allow_SSL --priority 160 --direction Inbound
--source-address-prefixes Internet --destination-port-ranges 443 --access
Allow --protocol Tcp --description "Permit SSL Inbound"
```

Мы видим как новые, только что созданные правила, так и правила по умолчанию (рис. 11.27).

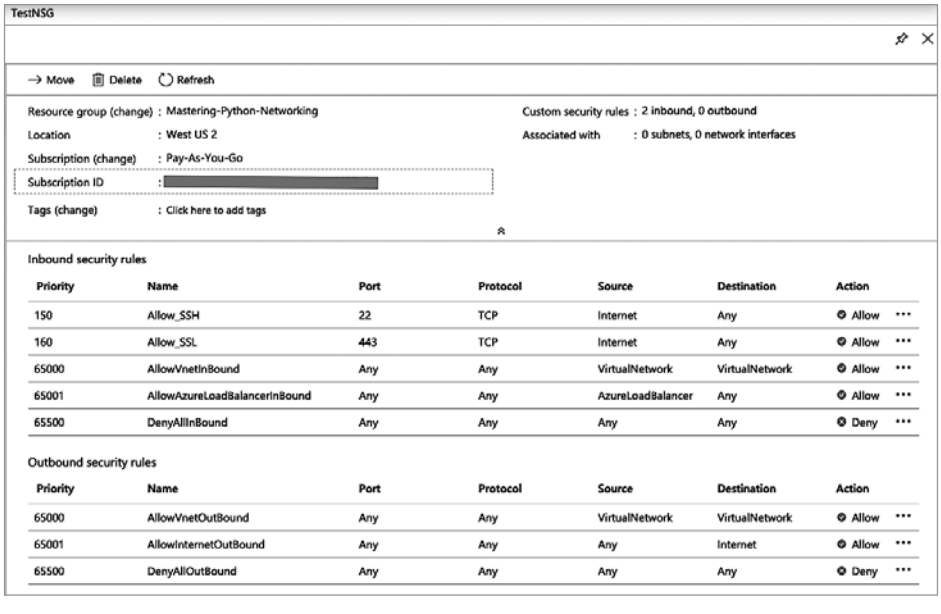


Рис. 11.27. Правила безопасности в Azure

Напоследок эту группу NSG нужно назначить подсети:

```
(venv) $ az network vnet subnet update -g "Mastering-Python-Networking"
-n WEST-US-2_VNet_1_Subnet_1 --vnet-name WEST-US-2_VNet_1 --networksecurity-
group TestNSG
```

В следующих двух разделах мы рассмотрим два основных способа подключения физических локальных сетей в дата-центре к виртуальным сетям Azure: Azure VPN и Azure ExpressRoute.

Azure VPN

При разрастании сети может наступить момент, когда Azure VNet нужно подключить к локально размещенному оборудованию. VPN-шлюз — это разновидность VNet-шлюза, шифрующая трафик между VNet, локальной физической сетью и удаленными клиентами. У каждой виртуальной сети может быть только один VPN-шлюз, но на основе этого шлюза можно создавать множественные соединения.

Больше о VNP-шлюзах в Azure читайте на странице https://docs.microsoft.com/ru-ru/azure/vpn-gateway/.

VPN-шлюзы на самом деле — это виртуальные машины с сервисами шифрования и маршрутизации, но пользователь не может конфигурировать их напрямую. Azure предоставляет список SKU на основе типа туннеля, множество параллельных соединений и общую пропускную способность (https://docs.microsoft.com/ru-ru/azure/vpn-gateway/vpn-gateway-about-vpn-gateway-settings#gwsku) (рис. 11.28).

Gateway SKUs by tunnel, connection, and throughput

SKU	S2S/VNet-to-VNet Tunnels	P2S SSTP Connections	P2S IKEv2/OpenVPN Connections	Aggregate Throughput Benchmark	BGP	Zone-redundant
Basic	Max. 10	Max. 128	Not Supported	100 Mbps	Not Supported	No
VpnGw1	Max. 30*	Max. 128	Max. 250	650 Mbps	Supported	No
VpnGw2	Max. 30*	Max. 128	Max. 500	1 Gbps	Supported	No
VpnGw3	Max. 30*	Max. 128	Max. 1000	1.25 Gbps	Supported	No
VpnGw1AZ	Max. 30*	Max. 128	Max. 250	650 Mbps	Supported	Yes
VpnGw2AZ	Max. 30*	Max. 128	Max. 500	1 Gbps	Supported	Yes
VpnGw3AZ	Max. 30*	Max. 128	Max. 1000	1.25 Gbps	Supported	Yes

Рис. 11.28. SKU VPN-шлюзов в Azure (источник: https://docs.microsoft.com/ru-ru/azure/vpn-gateway/point-to-site-about)

В предыдущей таблице сервис Azure VPN разделен на две категории: *P2S (Point-to-Site — «точка — сайт») и S2S (Site-to-Site — «сайт — сайт»)*. P2S VPN позволяет устанавливать защищенные соединения с отдельного клиентского компьютера и в основном используется для удаленной работы. В качестве метода шифрования применяются SSTP, IKEv2 или OpenVPN. При выборе SKU VPN-шлюза для P2S основное внимание следует уделить второму и третьему столбцам, в которых указано количество соединений.

Для клиентских VPN-соединений в качестве протокола туннелирования используется SSTP или IKEv2 (рис. 11.29).

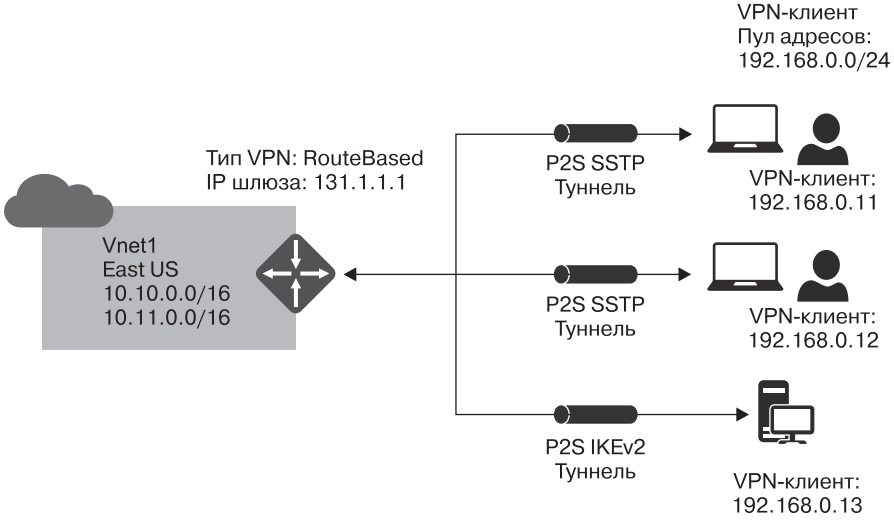


Рис. 11.29. VPN-шлюз типа «сайт — сайт» в Azure (источник: https://docs.microsoft.com/ru-ru/azure/vpn-gateway/vpn-gateway-about-vpngateways)

Помимо клиентского, существует еще один тип VPN-соединений, «сайт — сайт». Для шифрования в нем используется IPSec поверх IKE, а публичный IP-адрес требуется как для Azure, так и для физической локальной сети (рис. 11.30).

Полноценный пример создания VPN-соединения типа S2S или P2S выходит за рамки этого раздела. Azure предоставляет практические руководства по S2S (https://docs.microsoft.com/ru-ru/azure/vpn-gateway/vpn-gateway-howto-site-to-site-resource-manager-portal) и P2S (https://docs.microsoft.com/ru-ru/azure/vpn-gateway/vpn-gateway-howto-point-to-site-resource-manager-portal).

Для инженеров, которые уже настраивали VPN-сервисы, этот процесс покажется простым и понятным. Единственный момент, который может вызвать затруднения и который не объясняется в документации: устройство VPN-шлюза должно находиться в выделенной для него подсети внутри VNet с блоком IP-адресов `/27` (рис. 11.31).

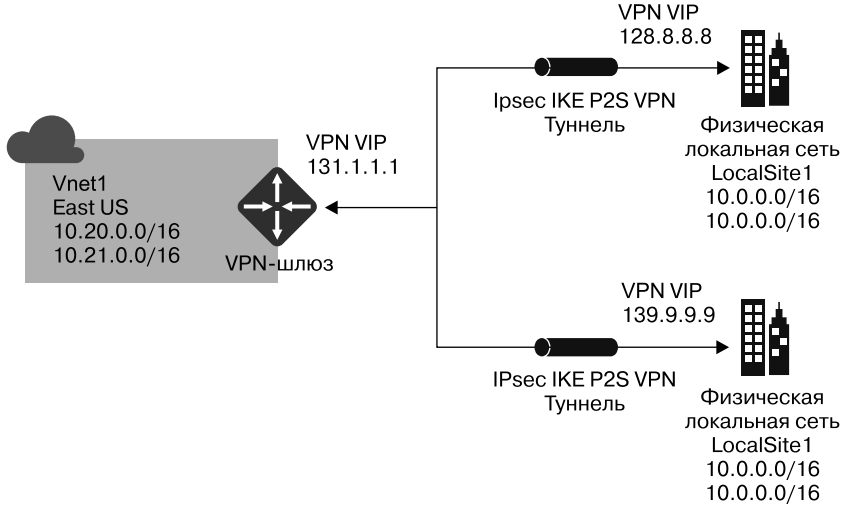


Рис. 11.30. Клиентский VPN-шлюз в Azure (источник: https://docs.microsoft.com/ru-ru/azure/vpn-gateway/vpn-gateway-about-vpngateways)

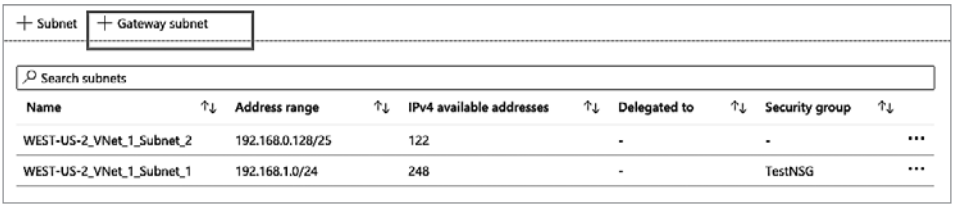


Рис. 11.31. Подсеть VPN-шлюза в Azure

Постоянно обновляемый список проверенных устройств для Azure VPN можно найти на странице https://docs.microsoft.com/ru-ru/azure/vpn-gateway/vpn-gateway-about-vpn-devices; там же приводятся ссылки на соответствующие руководства по конфигурации.

Azure ExpressRoute

Когда организации нужно подключить Azure VNet к физическим узлам, логично начинать с VPN-соединения. Но чем больше критически важного трафика проходит через это соединение, тем важнее становится его стабильность и надежность. Подобно AWS Direct Connect, Azure предоставляет ExpressRoute — частное соединение, реализованное сетевым провайдером. Как видно на рис. 11.32, наша сеть проходит через граничную партнерскую сеть и только потом соединяется с граничной сетью Azure.

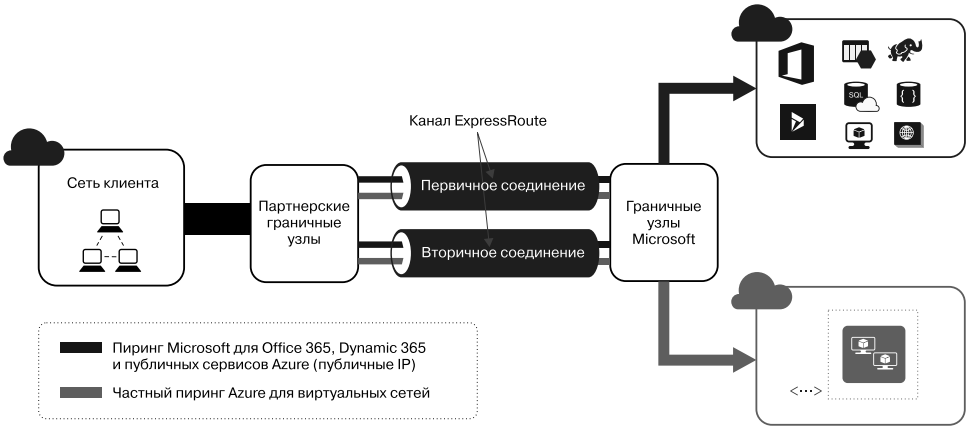


Рис. 11.32. Каналы Azure ExpressRoute (источник: https://docs.microsoft.com/ru-ru/azure/expressroute/expressroute-introduction)

Среди преимуществ ExpressRoute можно выделить:

- повышенную надежность, так как трафик не проходит по публичному интернету;
- увеличенную скорость соединения и уменьшенную задержку, так как у частного соединения между локально размещенным оборудованием и Azure, скорее всего, меньше транзитных участков;
- повышенные меры безопасности, так как это частное соединение, особенно если компания полагается на такие сервисы Microsoft, как Office 365.

ExpressRoute имеет и недостатки:

- более сложную конфигурацию с точки зрения технических и бизнес-требований;

- значительные начальные вложения, так как плата за порт и соединение зачастую фиксирована; некоторые расходы могут нивелироваться за счет уменьшения платы за интернет, если ExpressRoute заменяет VPN-соединение, однако общая стоимость владения обычно получается более высокой.

Более подробный обзор ExpressRoute можно найти на странице https://docs.microsoft.com/ru-ru/azure/expressroute/expressroute-introduction. Одно из главных отличий этого сервиса от AWS Direct Connect — поддержка соединений между регионами в пределах одной географической области. Кроме того, за дополнительную плату можно провести глобальное соединение с сервисами Microsoft и получить QoS-поддержку для Skype for Business.

Как и Direct Connect, ExpressRoute требует, чтобы клиент подключался к Azure через партнерскую сеть или в заранее оговоренной точке с использованием ExpressRoute Direct (да, это название может ввести в заблуждение). Для предприятий это обычно становится самой большой преградой, так как им приходится либо строить собственный дата-центр на одной из площадок Azure и затем соединяться с провайдером (MPLS VPN), либо сотрудничать с посредником. Эти варианты, как правило, требуют заключения бизнес-договоров, взятия на себя долгосрочных обязательств и регулярных ежемесячных выплат.

Мой совет тем, кто начинает этим заниматься, будет похож на тот, который я дал в главе 10: используйте услуги провайдера-посредника для соединения с транзитным центром, а оттуда уже либо подключайтесь напрямую к Azure, либо задействуйте промежуточную сеть, такую как Equinix CloudExchange.

В следующем разделе мы поговорим о том, как эффективно распределять входящий трафик в ситуациях, когда наш сервис выходит за рамки одного сервера.

Сетевые балансировщики нагрузки в Azure

Azure предлагает балансировщики нагрузки как в базовом, так и в стандартном SKU. В этом разделе речь идет о распределении трафика в протоколах транспортного уровня, таких как TCP и UDP, а не о балансировщиках прикладного уровня, таких как Application Gateway Load Balancer (https://azure.microsoft.com/ru-ru/services/application-gateway/).

Типичная модель развертывания обычно предусматривает одно- или двухуровневое распределение трафика, поступающего из интернета (рис. 11.33).

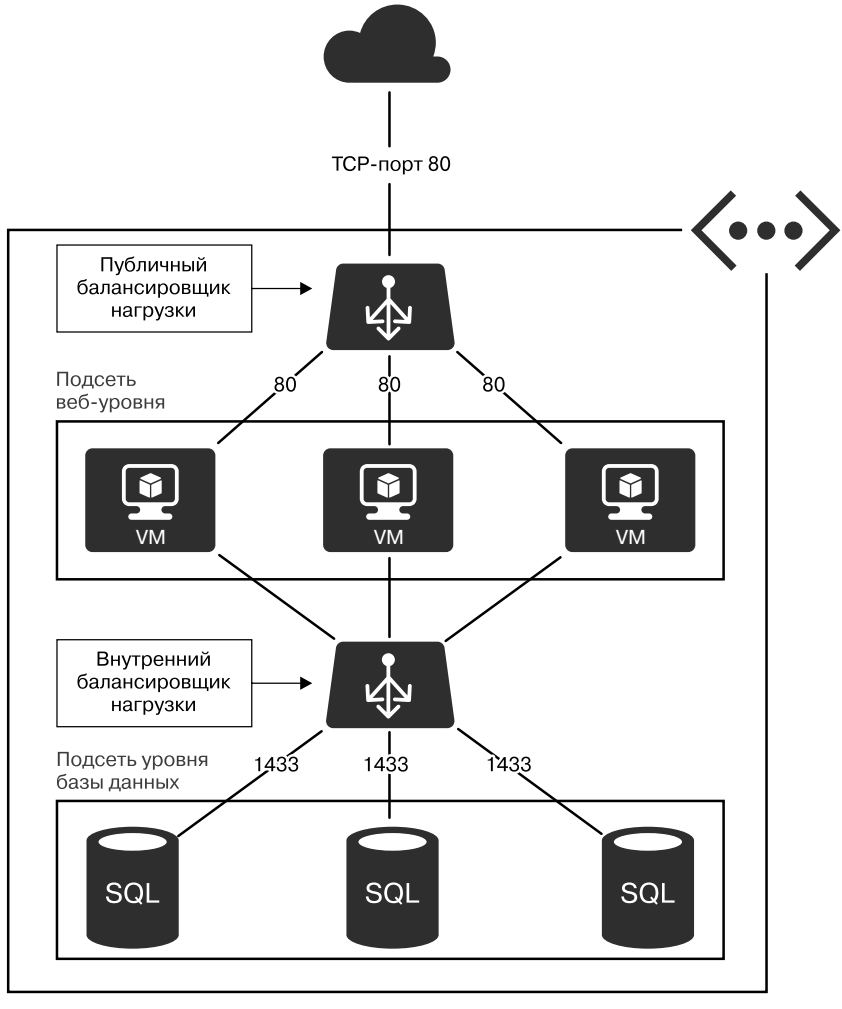


Рис. 11.33. Балансировщик нагрузки в Azure (источник: https://docs.microsoft.com/ru-ru/azure/load-balancer/load-balancer-overview)

Балансировщик нагрузки создает для входящего соединения пятиэлементный хеш (исходный и конечный IP-адреса, исходный и конечный порты и протокол) и направляет поток в одно или несколько мест назначения. SKU Standard Load Balancer представляет собой надмножество базового SKU, поэтому новые приложения должны использовать Standard Load Balancer.

Azure, как и AWS, постоянно выпускает новые сетевые сервисы. Мы уже рассмотрели те из них, которые можно назвать фундаментальными. Познакомимся с еще некоторыми сервисами.

Другие сетевые сервисы Azure

Сервисы Azure, о которых следует знать:

- **DNS-сервисы.** В Azure есть набор DNS-сервисов (https://docs.microsoft.com/ru-ru/azure/dns/dns-overview), как публичных, так и частных. Их можно использовать для географического распределения нагрузки в сетевых сервисах.
- **Контейнеры.** В последние годы Azure продвигает контейнеры. Больше о сетевых возможностях в контексте контейнеров читайте на странице https://docs.microsoft.com/ru-ru/azure/virtual-network/container-networking-overview.
- **VNet TAP.** Azure VNet TAP позволяет организовать непрерывную потоковую передачу трафика вашей виртуальной машины сборщику или анализатору сетевых пакетов (https://docs.microsoft.com/ru-ru/azure/virtual-network/virtual-network-tap-overview).
- **Защита от DDoS.** Azure предоставляет защиту от DDoS-атак (https://docs.microsoft.com/ru-ru/azure/virtual-network/ddos-protection-overview).

Сетевые сервисы — это важная часть облачных продуктов Azure, они развиваются высокими темпами. В этой главе мы рассмотрели лишь некоторые из них, но, надеюсь, теперь вы можете самостоятельно исследовать другие сервисы.

Резюме

В этой главе мы обсудили различные облачные сетевые сервисы Azure, рассмотрели глобальную сеть Azure и разные аспекты виртуальных сетей. Для создания, обновления и администрирования этих сетевых сервисов мы использовали Azure CLI и Python SDK. Если понадобится объединить сервисы Azure с локально размещенным дата-центром, то для этого можно использовать либо VPN, либо ExpressRoute. Мы также познакомились с другими сетевыми продуктами и сервисами Azure.

В следующей главе мы вернемся к процессу анализа данных и поговорим о многофункциональном пакете Elastic Stack.

12

Анализ сетевых данных с помощью Elastic Stack

В главах 7 и 8 мы обсудили различные методы мониторинга сети, включая два разных подхода к сбору сетевых данных: мы можем либо сами извлекать информацию из сетевых устройств (например, по SNMP), либо принимать те данные, которые устройство посылает по своей инициативе (используя потоковую передачу). Собранную информацию необходимо сохранить в базе данных и проанализировать, чтобы понять, что она означает. В большинстве случаев результаты анализа выводятся в графическом виде — в виде линейного графика либо столбиковой или круговой диаграммы. На каждом из этапов можно использовать отдельные инструменты, такие как PySNMP, Matplotlib и Pygal, или многофункциональные решения для мониторинга, такие как Cacti или Ntop. Инструменты, представленные в этих двух главах, позволяют организовать простой мониторинг и получить общее представление о нашей сети.

В главе 9 мы обсудили создание API-сервисов, чтобы открыть сеть для более высокоуровневых инструментов. В главах 10 и 11 мы рассмотрели возможность переноса нашей локальной сети в облако с использованием услуг AWS и Azure. В этих главах вы познакомились с большим количеством инструментов, помогающих сделать сеть программируемой.

Начиная с этой главы мы будем применять как уже знакомые вам инструменты, так и другие проекты, которые пригодились мне в моей профессиональной деятельности. Например, здесь рассмотрим проект с открытым исходным кодом

Elastic Stack (https://www.elastic.co), который расширит ваши представления о механизмах анализа и мониторинга сети.

Эта глава охватывает следующие темы:

- введение в Elastic Stack (или ELK);
- установку Elastic Stack;
- прием данных с помощью Logstash;
- прием данных с помощью Beats;
- поиск с помощью Elasticsearch;
- визуализацию данных с помощью Kibana.

Итак, начнем с краткого введения в Elastic Stack.

Что такое Elastic Stack

Elastic Stack еще называют стеком ELK (рис. 12.1). Что же это такое? Разработчики проекта говорят (https://www.elastic.co/what-is/elk-stack):

> *«ELK — это аббревиатура из названий трех открытых проектов: Elasticsearch, Logstash и Kibana. Elasticsearch — это поисковая и аналитическая система. Logstash — серверный механизм обработки данных, который берет информацию сразу из нескольких источников, преобразует ее и помещает в хранилище, например Elasticsearch. Kibana дает возможность пользователям визуализировать данные из Elasticsearch в виде диаграмм и графиков. Elastic Stack — это следующий шаг в развитии стека ELK».*

Судя по цитате, Elastic Stack фактически набор проектов, объединенных для решения целого спектра задач по сбору, хранению, извлечению, анализу и визуализации данных. Замечательная особенность этого стека: все его компоненты тесно интегрированы между собой, но при этом каждый можно использовать по отдельности. Если вам не нравится Kibana, вы можете легко подключить Grafana для создания диаграмм. Что, если мы хотим использовать другое средство сбора данных? Нет проблем, данные можно передавать в Elasticsearch с помощью RESTful API. Сердце стека Elasticsearch — открытая распределенная поисковая система. Для улучшения и поддержки функции поиска созданы и другие проекты. Поначалу это может казаться запутанным, но после знакомства с компонентами Elastic Stack картина должна проясниться.

Почему проект ELK Stack переименовали в Elastic Stack? В 2015 году компания Elastic представила семейство легковесных, узкоспециализированных средств передачи данных под названием Beats. Они мгновенно стали хитом и по-прежнему сохраняют высокую популярность, но их создатели не сумели придумать удачную аббревиатуру с буквой B и решили просто переименовать весь стек в Elastic Stack.

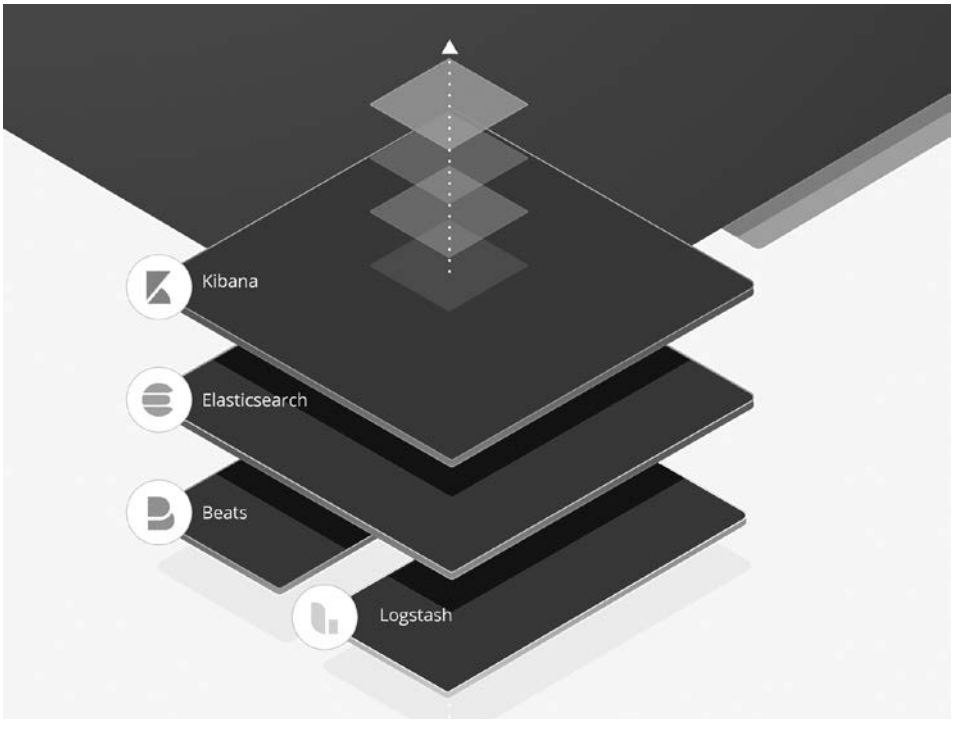


Рис. 12.1. Elastic Stack (источник: https://www.elastic.co/what-is/elk-stack)

При обсуждении Elastic Stack мы сосредоточимся на сетевом мониторинге и анализе данных, но этот стек применяется также для решения других задач, включая управление рисками, персонализацию в электронной торговле, анализ безопасности, выявление случаев мошенничества и др. Эти инструменты используются целым рядом организаций: от таких интернет-компаний, как Cisco, Box и Adobe, до правительственных учреждений вроде NASA JPL, Бюро переписи населения США и др. (https://www.elastic.co/customers/).

Разобравшись с тем, что такое стек ELK, перейдем к топологии нашей лаборатории для этой главы.

В этой книге под названием Elastic подразумевается компания, стоящая за Elastic Stack. Инструменты распространяются с открытым исходным кодом, а компания зарабатывает на предоставлении поддержки, локально размещаемых решений и консалтинге. Акции Elastic размещены публично на Нью-Йоркской фондовой бирже и обозначаются как ESTC.

Топология лаборатории

За основу возьмем топологию сети из главы 8. Управляющие интерфейсы сетевого оборудования будут находиться в управляющей сети `172.16.1.0/24` и соединяться с сетью `10.0.0.0/8` и подсетями `/3x`.

В какое место лаборатории установить стек ELK? Для этого можно выбрать управляющий хост, который мы использовали ранее. Еще один вариант: установка на отдельную *виртуальную машину (ВМ)* с двумя интерфейсами, один из которых подключен к управляющей сети, а другой — к внешней. Лично я предпочитаю второй подход с разделением серверов мониторинга и управления, потому что система мониторинга обычно имеет аппаратные и программные требования, отличные от требований других серверов, в чем вы убедитесь в следующих разделах. Еще одна причина для разделения: такая конфигурация больше похожа на то, что можно встретить в промышленных условиях; она позволяет разделить администрирование и мониторинг между двумя хостами. На рис. 12.2 приведена схема топологии нашей лаборатории.

Стек ELK будет установлен на новый сервер с Ubuntu 18.04 и двумя NIC, IP-адрес одного из которых размещен в той же управляющей сети, `172.16.1.200`. IP-адрес второго NIC этой ВМ — `192.168.2.200`, и он соединяет мою домашнюю сеть с интернетом.

В промышленных условиях обычно рекомендуется, чтобы кластер ELK состоял по меньшей мере из трех узлов: одного ведущего и двух других для хранения данных. Что касается функций, ведущие узлы ELK могут управлять кластером и индексировать данные, а остальные — эти данные извлекать. Рекомендации относительно трехузловой системы обусловлены заботой о надежности: один узел является активным ведущим, и, если он выйдет из строя, два других смогут его подменить. Но в условиях нашей лаборатории это не особенно важно, поэтому наша система будет состоять из одного узла, играющего роль ведущего, который одновременно хранит данные. Мы обойдемся без дублирования.

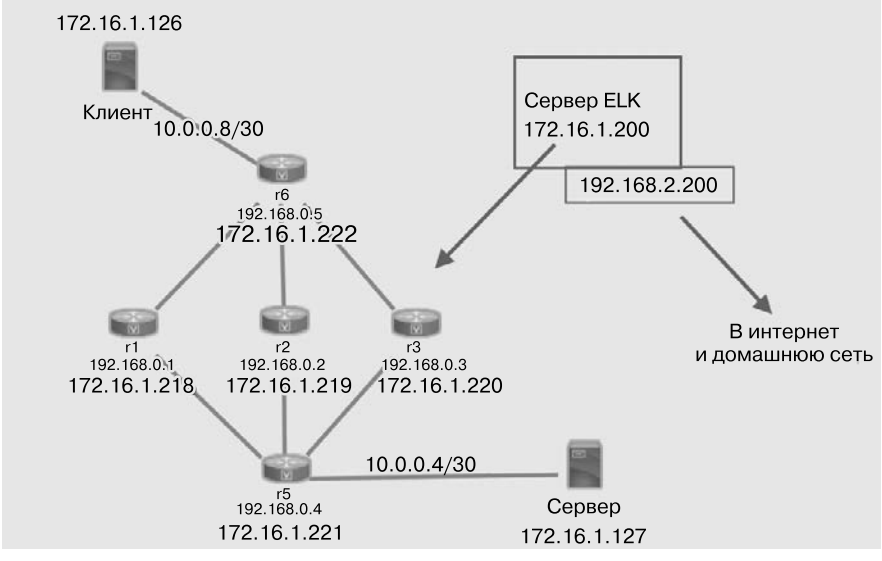


Рис. 12.2. Топология лаборатории

Аппаратные требования стека ELK во многом зависят от объема данных, которые мы хотим разместить в системе. Поскольку это всего лишь лаборатория, нам не нужно большой вычислительной мощности, так как данных у нас будет не очень много.

Подробнее о подготовительных шагах читайте в документации Elastic Stack по адресу https://www.elastic.co/guide/index.html.

В целом Elasticsearch требует много памяти, но не нуждается в мощном процессоре и большом хранилище. Создадим отдельную ВМ со следующей конфигурацией:

- процессор — 1 vCPU;
- память — 4 Гбайт (по возможности больше);
- диск — 20 Гбайт;
- сеть — 1 NIC в управляющей сети лаборатории и один дополнительный (необязательный) NIC для доступа к интернету.

Система Elasticsearch написана на Java, и каждый дистрибутив поставляется вместе с OpenJDK. Последнюю версию Elasticsearch можно загрузить на сайте Elastic.co:

```
echou@elk-stack-mpn:~$ wget https://artifacts.elastic.co/downloads/
elasticsearch/elasticsearch-7.4.2-linux-x86_64.tar.gz
echou@elk-stack-mpn:~$ tar -xvzf elasticsearch-7.4.2-linux-x86_64.tar.gz
echou@elk-stack-mpn:~$ cd elasticsearch-7.4.2/
```

Поправим параметры виртуальной памяти, установленные на узле по умолчанию (https://www.elastic.co/guide/en/elasticsearch/reference/current/vm-max-map-count.html):

```
echou@elk-stack-mpn:~$ sudo sysctl -w vm.max_map_count=262144
```

Узлы Elasticsearch по умолчанию пытаются обнаружить своих соседей и сформировать кластер. Поэтому перед запуском рекомендуется поменять имя узла и параметры, управляющие кластеризацией. Отредактируем конфигурационный файл `elasticsearch.yml`:

```
echou@elk-stack-mpn:~/elasticsearch-7.4.2$ vim config/elasticsearch.yml
# изменим следующие параметры
node.name: mpn-node-1
network.host: <change to your host IP>
http.port: 9200
discovery.seed_hosts: ["mpn-node-1"]
cluster.initial_master_nodes: ["mpn-node-1"]
```

Теперь Elasticsearch можно запустить в фоновом режиме:

```
echou@elk-stack-mpn:~/elasticsearch-7.4.2$ ./bin/elasticsearch &
```

Чтобы проверить результат, пошлем запрос HTTP GET хосту с Elasticsearch. Послать запрос можно с управляющего хоста или локально, на хосте с системой мониторинга:

```
(venv) $ curl 192.168.2.200:9200
{
"name" : "mpn-node-1",
"cluster_name" : "elasticsearch",
"cluster_uuid" : "9hTywXc-S9eg3jMi6__XSQ",
"version" : {
  "number" : "7.4.2",
  "build_flavor" : "default",
  "build_type" : "tar",
  "build_hash" : "2f90bbf7b93631e52bafb59b3b049cb44ec25e96",
  "build_date" : "2019-10-28T20:40:44.881551Z",
  "build_snapshot" : false,
  "lucene_version" : "8.2.0",
  "minimum_wire_compatibility_version" : "6.8.0",
  "minimum_index_compatibility_version" : "6.0.0-beta1"
},
"tagline" : "You Know, for Search"
}
```

Повторим эту процедуру для установки Kibana, нашего средства визуализации:

```
echou@elk-stack-mpn:~/ $ wget https://artifacts.elastic.co/downloads/
kibana/kibana-7.4.2-linux-x86_64.tar.gz
echou@elk-stack-mpn:~/ $ tar -xvzf kibana-7.4.2-linux-x86_64.tar.gz
echou@elk-stack-mpn:~/ $ cd kibana-7.4.2-linux-x86_64/
```

И снова внесем изменения в конфигурационный файл:

```
echou@elk-stack-mpn:~/ kibana-7.4.2-linux-x86_64$ vim config/kibana.yml
server.port: 5601
server.host: "192.168.2.200"
server.name: "mastering-python-networking"
elasticsearch.hosts: ["http://192.168.2.200:9200"]
```

Запустим процесс Kibana в фоновом режиме:

```
echou@elk-stack-mpn:~/kibana-7.4.2-linux-x86_64$ ./bin/kibana &
```

Когда процесс загрузится до конца, откроем в браузере страницу http://<IP-адрес>:5601 (рис. 12.3).

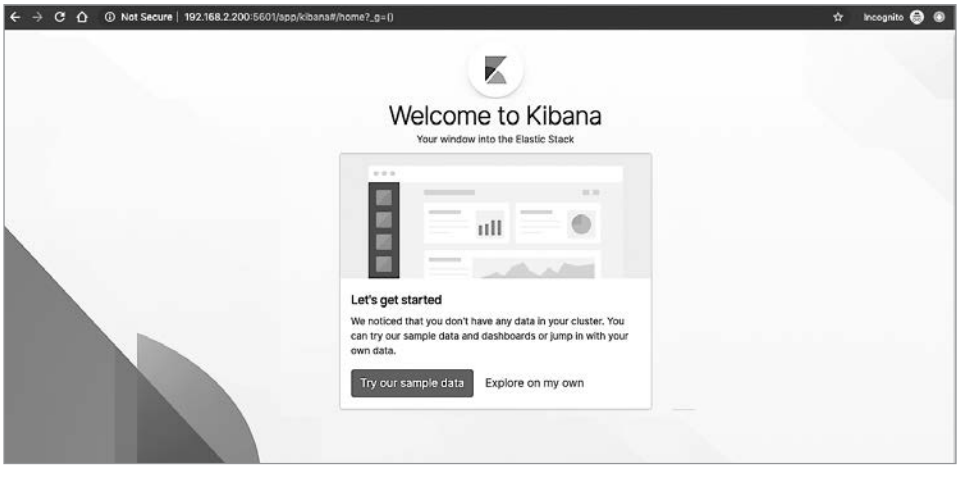


Рис. 12.3. Начальная страница Kibana

Нам предложат загрузить демонстрационные данные. Это отличный шанс опробовать новый инструмент, воспользуемся им (рис. 12.4).

Отлично! Мы почти закончили. Осталось только установить Logstash. В отличие от Elasticsearch, Logstash не поставляется вместе с Java, поэтому сначала нужно установить Java 8 или Java 11:

```
echou@elk-stack-mpn:~$ sudo apt install openjdk-11-jre-headless
echou@elk-stack-mpn:~$ java --version
openjdk 11.0.4 2019-07-16
OpenJDK Runtime Environment (build 11.0.4+11-post-Ubuntu-1ubuntu218.04.3)
OpenJDK 64-Bit Server VM (build 11.0.4+11-post-Ubuntu-1ubuntu218.04.3,
mixed mode, sharing)
```

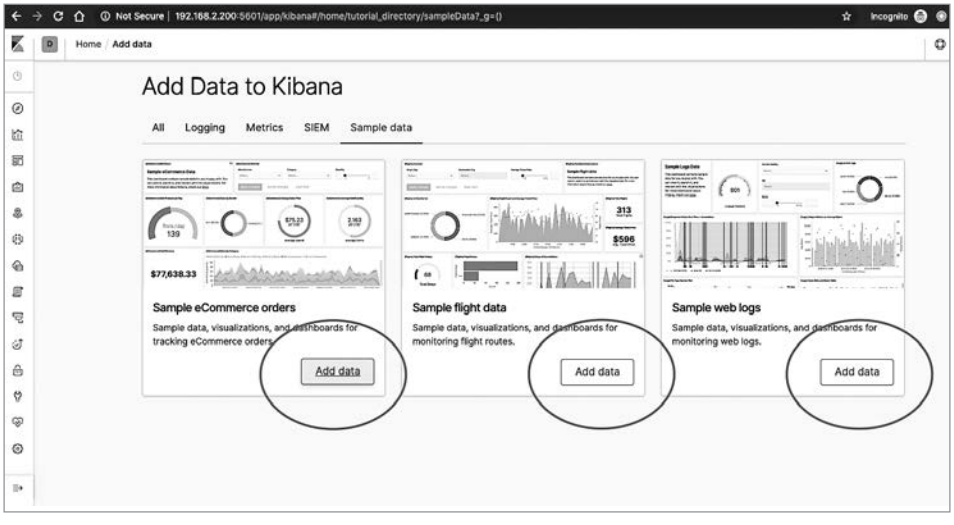


Рис. 12.4. Добавление данных в Kibana

Загрузим, распакуем и сконфигурируем Logstash, как мы это делали с Elasticsearch и Kibana:

```
echou@elk-stack-mpn:~$ wget https://artifacts.elastic.co/downloads/
logstash/logstash-7.4.2.tar.gz
echou@elk-stack-mpn:~$ tar -xvzf logstash-7.4.2.tar.gz
echou@elk-stack-mpn:~$ cd logstash-7.4.2/
echou@elk-stack-mpn:~/logstash-7.4.2$ vim config/logstash.yml
node.name: mastering-python-networking
http.host: "192.168.2.200"
http.port: 9600-9700
```

Но пока не будем запускать Logstash. Сначала нужно установить сетевые плагины и создать необходимый конфигурационный файл. Мы займемся этим чуть позже.

В следующем разделе речь пойдет о развертывании стека ELK в виде удаленного сервиса.

Elastic Stack как услуга

Elasticsearch предоставляется не только в виде пакета, но и как удаленный сервис от Elastic.co и AWS. Elastic Cloud (https://www.elastic.co/cloud/) не имеет собственной инфраструктуры, но позволяет развернуть его в AWS, Google Cloud Platform или Azure. Поскольку решение основано на других публичных облаках, его стоимость будет чуть выше, чем размещение Elasticsearch напрямую в облачном провайдере, таком как AWS (рис. 12.5).

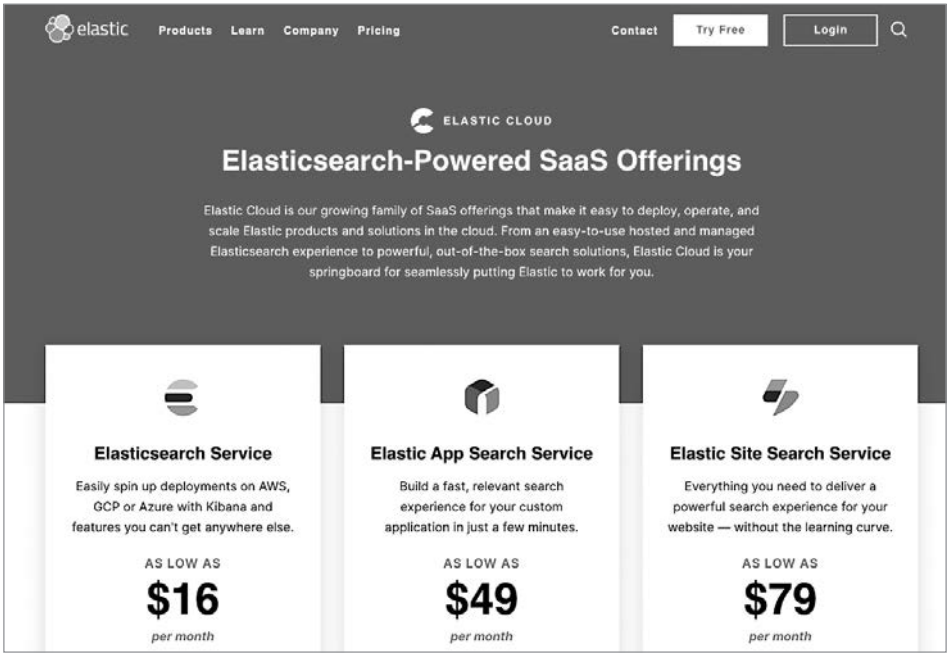


Рис. 12.5. Тарифы Elastic Cloud

AWS предлагает управляемый продукт на основе Elasticsearch (https://aws.amazon.com/ru/elasticsearch-service/), который тесно интегрирован с другими сервисами этого провайдера. Например, поток журнальных записей AWS CloudWatch можно напрямую передать инстансу AWS Elasticsearch (https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_ES_Stream.html) (рис. 12.6).

По моему опыту, Elastic Stack дает ощущение простоты в самом начале, но его дальнейшее масштабирование требует глубоких знаний. Это создает трудности для тех, кто не работает с Elasticsearch ежедневно. Если вы, как и я, хотите вос-

пользоваться преимуществами Elasticsearch, но при этом не планируете становиться специалистом по технологиям Elastic, я настоятельно рекомендую применять в промышленных условиях одно из сторонних решений.

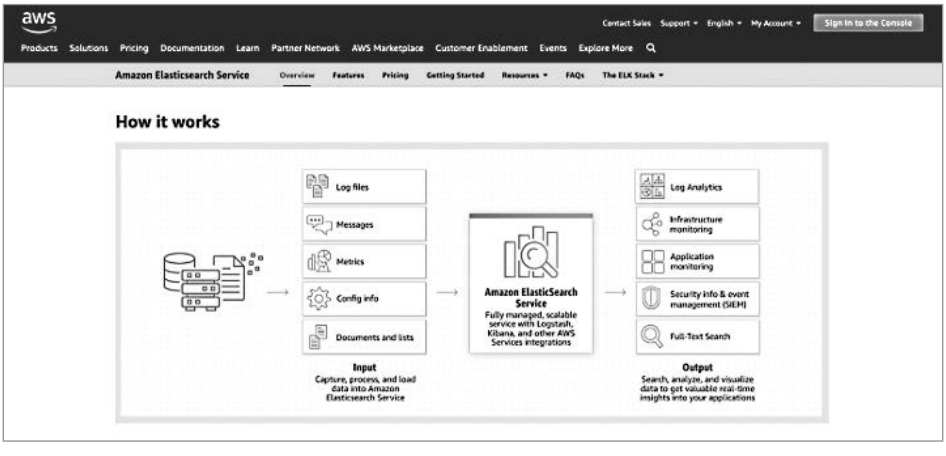


Рис. 12.6. Сервис Elasticsearch

Выбор стороннего решения зависит от облачного провайдера, услугами которого вы пользуетесь, и наличия самых последних возможностей. Поскольку сервис Elastic Cloud создан теми же людьми, которые занимаются проектом Elastic Stack, самые актуальные функции в нем обычно появляются раньше, чем в AWS. С другой стороны, если ваша инфраструктура полностью построена на облаке AWS, использование тесно интегрированного инстанса Elasticsearch сэкономит вам время и силы, необходимые для обслуживания отдельного кластера.

В следующем разделе мы рассмотрим процесс обработки данных от их приема до визуализации.

Первый полный пример

Люди, знакомящиеся с Elastic Stack, чаще всего отмечают множество деталей, которые необходимо понимать, чтобы пользоваться этой системой. Прежде чем сохранить в Elastic Stack первую полезную запись, пользователь должен создать кластер, выделить ведущий узел и узлы данных, принять данные, построить индекс и наладить управление всем этим через веб-консоль или интерфейс

командной строки. С годами процесс установки Elastic Stack упростился, улучшилась официальная документация и были подготовлены пробные наборы данных, чтобы новые пользователи могли познакомиться с инструментами, прежде чем применять этот стек в промышленных условиях.

Перед тем как углубляться в Elastic Stack, рассмотрим пример с Logstash, Elasticsearch и Kibana. Благодаря этому мы познакомимся с функциями, которые предоставляет каждый из компонентов. Позже, когда мы начнем обсуждать их по отдельности, вы уже будете представлять, какую роль он играет в системе в целом.

Для начала сохраним наши журнальные данные в Logstash. Сконфигурируем каждый маршрут для экспорта записей на сервер Logstash:

```
r[1-6]#sh run | i logging
logging host 172.16.1.200 vrf Mgmt-intf transport udp port 5144
```

На нашем хосте с Elastic Stack, где установлены все компоненты, создадим простую конфигурацию Logstash, которая прослушивает UDP-порт 5144 и посылает данные хосту Elasticsearch:

```
echou@elk-stack-mpn:~$ cd logstash-7.4.2/
echou@elk-stack-mpn:~/logstash-7.4.2$ mkdir network_configs
echou@elk-stack-mpn:~/logstash-7.4.2$ touch network_configs/simple_
config.cfg

echou@elk-stack-mpn:~/logstash-7.4.2$ cat network_configs/simple_config.
cfg
input {
  udp {
    port => 5144
    type => "syslog-ios"
  }
}

output {
  elasticsearch {
    hosts => ["http://192.168.2.200:9200"]
    index => "cisco-syslog-%{+YYYY.MM.dd}"
  }
}
```

Файл конфигурации состоит только из разделов `input` и `output` без изменения данных. Тип `syslog-ios` — это имя, которым мы обозначили индекс. В разделе `output` мы указываем это имя вместе с переменными, представляющими текущую дату. Процесс Logstash можно запустить в фоновом режиме непосредственно из каталога с выполняемым файлом:

```
echou@elk-stack-mpn:~/logstash-7.4.2$ sudo bin/logstash -f network_
configs/simple_config.cfg
[2019-11-03T09:54:37,201][INFO ][logstash.inputs.udp      ][main]
UDP listener started {:address=>"0.0.0.0:5144", :receive_buffer_
bytes=>"106496", :queue_size=>"2000"}
<опущено>
```

При получении данных Elasticsearch по умолчанию автоматически генерирует для них индекс. Мы можем создать журнальные записи на маршрутизаторе, сбросив интерфейс, перезагрузив BGP или просто путем входа в режим конфигурации и выхода из него. После получения этих новых записей мы увидим созданный индекс `cisco-syslog-<дата>`:

```
[2019-11-03T10:01:09,029][INFO ][o.e.c.m.MetaDataCreateIndexService]
[mpn-node-1] [cisco-syslog-2019.11.03] creating index, cause [auto(bulk
api)], templates [], shards [1]/[1], mappings []
[2019-11-03T10:01:09,130][INFO ][o.e.c.m.MetaDataMappingService] [mpnnode-
1] [cisco-syslog-2019.11.03/00NRNwGlRx2OTf_b-qt9SQ] create_mapping
[_doc]
```

На этом этапе уже можно воспользоваться утилитой `curl`, чтобы просмотреть созданный в Elasticsearch индекс:

```
(venv) $ curl http://192.168.2.200:9200/_cat/indices/cisco*
yellow open cisco-syslog-2019.11.03 00NRNwGlRx2OTf_b-qt9SQ 1 1 7 0 20.2kb
20.2kb
```

Теперь этот индекс можно импортировать в Kibana, выбрав в меню пункт Settings ▸ Kibana ▸ Index Patterns (Параметры ▸ Kibana ▸ Шаблоны индекса) (рис. 12.7).

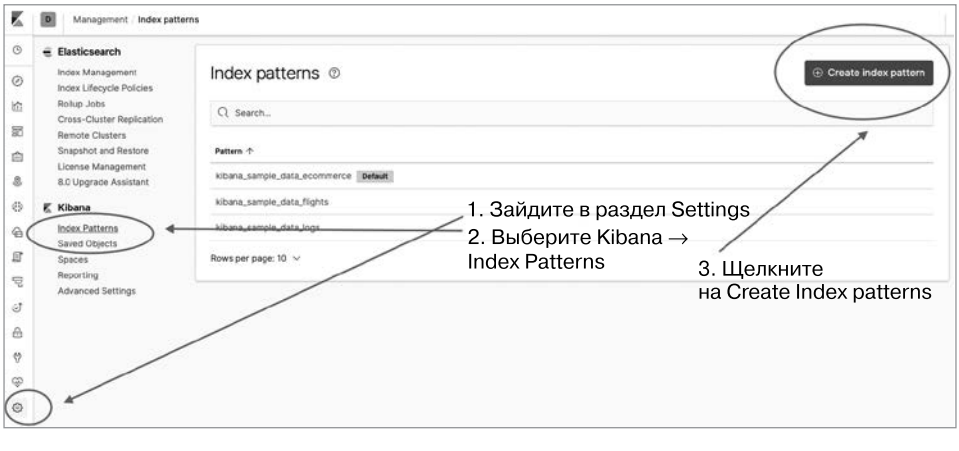


Рис. 12.7. Добавление шаблонов индекса из Elasticsearch

Поскольку индекс уже находится в Elasticsearch, нам остается лишь указать его имя. Помните, что имя нашего индекса представляет собой переменную, зависящую от времени; можно использовать символ-заполнитель (*), чтобы охватить все текущие и будущие индексы со словом `cisco` (рис. 12.8).

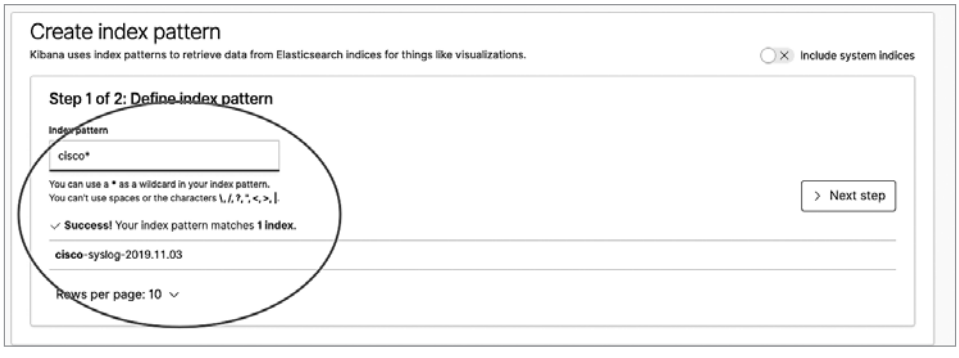


Рис. 12.8. Определение шаблона индекса в Elasticsearch

Наш индекс привязан к времени; это означает, что у нас есть поле, которое можно использовать как временную метку и выполнять поиск по дате. Это поле нужно указать отдельно. В нашем случае система Elasticsearch сама подобрала подходящее поле журнала в качестве временной метки; нам остается только выбрать его на втором шаге в раскрывающемся списке (рис. 12.9).

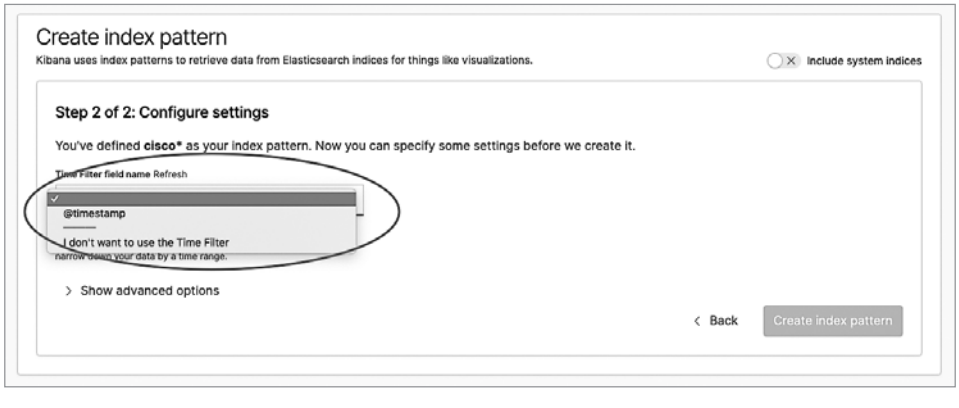


Рис. 12.9. Настройка временной метки для шаблона индекса в Elasticsearch

После создания шаблона индекса перейдем на вкладку Discover (Обнаружение), чтобы просмотреть соответствующие записи (рис. 12.10).

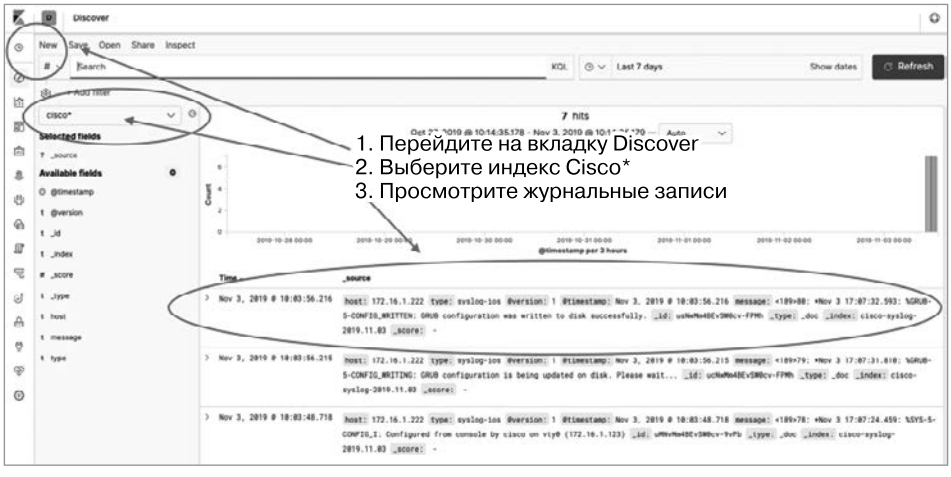


Рис. 12.10. Просмотр документа с индексом в Elasticsearch

Собрав еще немного информации, зайдем на сервер Elastic Search и нажмем Ctrl+C, чтобы остановить процесс Logstash. Этот пример показывает, как использовать Elastic Stack для приема, хранения и визуализации данных. Logstash (или Beats) принимает данные в виде непрерывного потока, который автоматически направляется в Elasticsearch. Средства визуализации Kibana дают возможность анализировать содержимое Elasticsearch, представляя его более наглядно, и затем, если мы удовлетворены результатом, создать постоянное визуальное представление. Ниже в этой главе мы рассмотрим разные методы визуализации, имеющиеся в Kibana.

Даже в этом простом примере можно увидеть, что Elasticsearch — самая важная часть рабочего процесса. Мощь этого стека и его способность адаптироваться к нашим задачам анализа сети основаны на простом REST-интерфейсе, масштабируемости хранилища данных, автоматической индексации и быстрой выдаче результатов поиска.

В следующем разделе мы посмотрим, как взаимодействовать с Elasticsearch с помощью Python.

Elasticsearch и клиент на языке Python

С Elasticsearch можно взаимодействовать через RESTful API с помощью библиотеки для Python. Например, ниже мы воспользуемся библиотекой `requests`,

чтобы выполнить запрос GET и извлечь информацию из хоста с Elasticsearch. Мы уже знаем, что HTTP-запрос типа GET к следующей конечной точке позволяет получить текущие индексы, имена которых начинаются с kibana:

```
(venv) $ curl http://192.168.2.200:9200/_cat/indices/kibana*
green open kibana_sample_data_ecommerce Pg5I-1d8SIu-LbpUtn67mA 1 0  4675
0    5mb    5mb
green open kibana_sample_data_logs      3Z2JMdk2T5OPEXnke9l5YQ 1 0 14074
0 11.2mb 11.2mb
green open kibana_sample_data_flights   sjIzh4FeQT2icLmXXhkDvA 1 0 13059
0  6.2mb  6.2mb
```

Реализуем то же самое в сценарии на языке Python, Chapter12_1.py, используя библиотеку requests:

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

def current_indices_list(es_host, index_prefix):
    current_indices = []
    http_header = {'content-type': 'application/json'}
    response = requests.get(es_host + "/_cat/indices/" + index_prefix
                            + "*", headers=http_header)
    for line in response.text.split('\n'):
        if line:
            current_indices.append(line.split()[2])
    return current_indices

if __name__ == "__main__":
    es_host = 'http://192.168.2.200:9200'
    indices_list = current_indices_list(es_host, 'kibana')
    print(indices_list)
```

Выполнив этот сценарий, мы получим список индексов, которые начинаются со слова kibana:

```
(venv) $ python Chapter12_1.py
['kibana_sample_data_ecommerce', 'kibana_sample_data_logs', 'kibana_
sample_data_flights']
```

Мы также можем использовать клиент Elasticsearch на языке Python, elasticsearch-py.readthedocs.io/en/master/. Это легковесная обертка вокруг RESTful API Elasticsearch, дающая максимальную гибкость. Установим ее и рассмотрим простой пример:

```
(venv) $ pip install elasticsearch
```

Этот сценарий, Chapter12_2, подключается к кластеру Elasticsearch и ищет любые индексы, которые начинаются со слова kibana:

```
#!/usr/bin/env python3
from elasticsearch import Elasticsearch

es_host = Elasticsearch("http://192.168.2.200/")

res = es_host.search(index="kibana*", body={"query": {"match_all": {}}})
print("Hits Total: " + str(res['hits']['total']['value']))
```

По умолчанию в результат включаются первые 10 000 записей:

```
(venv) $ python Chapter12_2.py
Hits Total: 10000
```

В этом простом примере преимущества клиентской библиотеки неочевидны. Ооднако она может очень пригодиться при выполнении более сложных операций поиска, таких как `scroll`, когда в каждом следующем запросе необходимо указывать токен, полученный из предыдущего, — и так, пока не будут возвращены все результаты. Клиент также может помочь с более сложными задачами администрирования, такими как перегенерация существующего индекса. Далее в этой главе вас ждут еще примеры с клиентской библиотекой.

В следующем разделе рассмотрим пример получения данных из журналов нашего устройства Cisco.

Прием данных с помощью Logstash

В предыдущем примере мы принимали данные из журналов сетевых устройств, используя Logstash. Возьмем его за основу и внесем изменения, как показано в `network_config/config_2.cfg`:

```
input {
  udp {
    port => 5144
    type => "syslog-core"
  }
  udp {
    port => 5145
    type => "syslog-edge"
  }
}
filter {
  if [type] == "syslog-edge" {
    grok {
      match => { "message" => ".*" }
      add_field => [ "received_at", "%{@timestamp}" ]
    }
  }
}
<опущено>
```

В разделе `input` мы настроили прослушивание двух UDP-портов, 5144 и 5145. Принимаемым журнальным записям назначается один из двух тегов: `syslog-core` или `syslog-edge`. Мы также добавили в конфигурацию раздел `filter`, чтобы отбирать именно тип `syslog-edge` и применять регулярное выражение к полю `message` в подразделе `grok`. В данном случае мы отбираем все подряд и добавляем новое поле со значением временной метки, `received_at`.

Больше о Grok читайте в официальной документации: https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html.

Мы изменим `r5` и `r6`, чтобы направлять информацию на UDP-порт 5145:

```
r[5-6]#sh run | i logging
logging host 172.16.1.200 vrf Mgmt-intf transport udp port 5145
```

При запуске сервера Logstash мы увидим оба прослушиваемых порта:

```
echou@elk-stack-mpn:~/logstash-7.4.2$ sudo bin/logstash -f network_
configs/config_2.cfg
<опущено>
[2019-11-03T15:31:35,480][INFO ][logstash.inputs.udp      ][main]
Starting UDP listener {:address=>"0.0.0.0:5145"}
[2019-11-03T15:31:35,493][INFO ][logstash.inputs.udp      ][main]
Starting UDP listener {:address=>"0.0.0.0:5144"}
<опущено>
```

Назначая записям разные типы, можем выполнять поиск по ним, используя вкладку Discover (Обнаружение) на панели управления Kibana (рис. 12.11).

Если развернуть запись типа `syslog-edge`, то мы увидим добавленное нами поле (рис. 12.12).

В конфигурационном файле Logstash доступно много параметров, разбитых по разделам `input`, `filter` и `output`. В частности, раздел `filter` дает возможность улучшать данные, отбирая подходящие записи и обрабатывая их перед сохранением в Elasticsearch. Logstash можно расширять с помощью модулей; каждый модуль — простое полноценное решение для приема и визуализации данных с отдельной панелью управления.

Подробности о модулях Logstash ищите на странице https://www.elastic.co/guide/en/logstash/7.4/logstash-modules.html.

Проект Elastic Beats похож на модули Logstash. Это набор узкоспециализированных средств экспорта данных, который обычно устанавливается в виде

агента; он собирает информацию на хосте и отправляет ее для дальнейшей обработки либо сразу в Elasticsearch, либо в Logstash.

Рис. 12.11. Индекс Syslog

Рис. 12.12. Временная метка в Syslog

Существуют буквально сотни разных модулей Beats, таких как Filebeat, Metricbeat, Packetbeat, Heartbeat и т. д. В следующем разделе вы увидите, как принимать журнальные данные в Elasticsearch с помощью Filebeat.

Прием данных с использованием Beats

Несмотря на все преимущества Logstash, процесс приема данных может оказаться сложным и плохо масштабируемым. Как показывает наш пример, трудности могут возникнуть даже с обычными сетевыми журналами, так как нам приходится анализировать журнальные записи разных форматов, принадлежащих маршрутизаторам IOS и NXOS, брандмауэрам ASA, беспроводным контроллерам Meraki и т. п. А что, если нам потребуется принимать журнальные записи от веб-сервера Apache, сведения о работоспособности хоста и информацию о безопасности? А как насчет таких форматов данных, как NetFlow, SNMP и счетчики производительности? Чем больше данных требуется агрегировать, тем сложнее может оказаться этот процесс.

Мы не можем избавиться от агрегирования и избежать всех сложностей, связанных с приемом данных, однако есть тенденция к выбору более легковесных, узкоспециализированных агентов, которые находятся как можно ближе к источнику информации. Например, на сервере Apache можно установить агент, предназначенный специально для сбора журнальных данных с веб-запросами; или же у нас может быть хост, который собирает, агрегирует и организует исключительно журнальные записи Cisco IOS. В Elastic Stack все эти средства объединены в общий проект Beats: https://www.elastic.co/products/beats.

Filebeat — это версия Elastic Beats, предназначенная для передачи и централизованного хранения журналов. Она ищет файл журнала, который мы указали в конфигурации, и приступает к его обработке; по завершении она отправляет новые журнальные записи внутреннему процессу, который агрегирует соответствующие события и передает их в Elasticsearch. В этом разделе вы увидите, как собирать данные из журналов, используя Filebeat с модулями Cisco.

Установим Filebeat и сконфигурируем на хосте Elasticsearch шаблон визуализации и индекс:

```
echou@elk-stack-mpn:~$ curl -L -O https://artifacts.elastic.co/downloads/
beats/filebeat/filebeat-7.4.2-amd64.deb
echou@elk-stack-mpn:~$ sudo dpkg -i filebeat-7.4.2-amd64.deb
```

Дерево каталогов может показаться запутанным, так как Filebeat устанавливается в разные места внутри `/usr`, `/etc/` и `/var` (рис. 12.13).

Type	Description	Location
home	Home of the Filebeat installation.	`/usr/share/filebeat`
bin	The location for the binary files.	`/usr/share/filebeat/bin`
config	The location for configuration files.	`/etc/filebeat`
data	The location for persistent data files.	`/var/lib/filebeat`
logs	The location for the logs created by Filebeat.	`/var/log/filebeat`

Рис. 12.13. Каталоги с файлами Elastic Filebeat (источник: https://www.elastic.co/guide/en/beats/filebeat/7.4/directory-layout.html)

Внесем изменения в конфигурационный файл `/etc/filebeat/filebeat.yml`, чтобы поменять местоположение Elasticsearch и Kibana:

```
setup.kibana:
  host: "192.168.2.200:5601"
output.elasticsearch:
  hosts: ["192.168.2.200:9200"]
```

Filebeat можно использовать для настройки шаблонов индексов и панелей управления Kibana:

```
echou@elk-stack-mpn:~$ sudo filebeat setup --index-management
-E output.logstash.enabled=false -E 'output.elasticsearch.
hosts=["192.168.2.200:9200"]'
echou@elk-stack-mpn:~$ sudo filebeat setup –dashboard
```

Включим модуль `cisco` в Filebeat:

```
echou@elk-stack-mpn:~$ sudo filebeat modules enable cisco
```

Настроим сначала модуль `cisco` для сбора журнальных записей. Файл находится в `/etc/filebeat/modules.d/cisco.yml`. В нашем случае следует отдельно указать путь к файлу журнала:

```
- module: cisco
  ios:
    enabled: true
    var.input: syslog
    var.syslog_host: 0.0.0.0
    var.syslog_port: 514
    var.paths: ['/home/echou/syslog/my_log.log']
```

Мы можем запускать, останавливать и проверять состояние сервиса Filebeat, используя стандартные для Ubuntu Linux команды `service Filebeat [start|stop|status]`:

```
echou@elk-stack-mpn:~$ sudo service filebeat start
```

Отредактируем или добавим на нашем устройстве UDP-порт 514 для `syslog`. Соответствующая журнальная информация должна выводиться при поиске по индексу `filebeat-*` (рис. 12.14).

Рис. 12.14. Индекс Elastic Filebeat

Если сравнить с предыдущим примером, то можно заметить, что у каждой записи появилось много новых полей и метаинформации, включая `agent.version`, `event.code` и `event.severity` (рис. 12.15).

Зачем нужны дополнительные поля? Они упрощают агрегирование результатов поиска и, следовательно, позволяют лучше их визуализировать. В следующем разделе, который посвящен Kibana, будут примеры диаграмм.

Помимо `cisco`, существуют модули для Palo Alto Networks, AWS, Google Cloud, MongoDB и многих других продуктов. Актуальный перечень модулей находится по адресу https://www.elastic.co/guide/en/beats/filebeat/7.4/filebeat-modules.html.

Что, если нам нужны данные NetFlow? Не проблема, для этого тоже есть модуль! Процесс включения модуля и подготовки панели управления такой же, как и в случае с Cisco:

```
echou@elk-stack-mpn:~$ sudo filebeat modules enable netflow
echou@elk-stack-mpn:~$ sudo filebeat setup -e
```

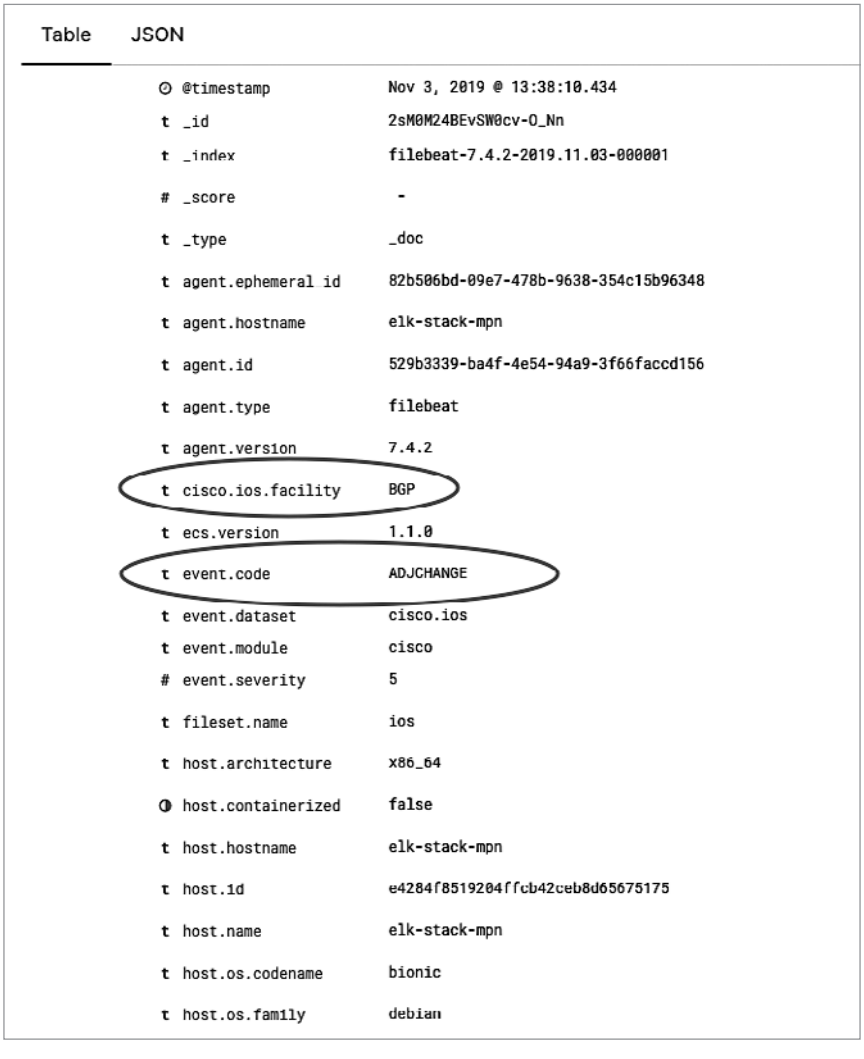


Рис. 12.15. Журнал Cisco в Elastic Filebeat

Теперь следует отредактировать конфигурационный файл модуля, `/etc/filebeat/modules.d/netflow.yml`:

```
- module: netflow
  log:
    enabled: true
    var:
      netflow_host: 0.0.0.0
      netflow_port: 2055
```

Мы сделаем так, чтобы устройства отправляли данные NetFlow в порт 2055. Если вам нужно освежить свои знания, почитайте соответствующую информацию в главе 8. В результате мы должны увидеть тип входных данных `netflow` (рис. 12.16).

Помните, что каждый модуль поставляется с готовыми шаблонами визуализации? Не будем забегать вперед, но, если перейти на вкладку Vizualization (Визуализация) на панели слева и поискать по слову *netflow*, то можно найти несколько доступных вариантов визуализации (рис. 12.17).

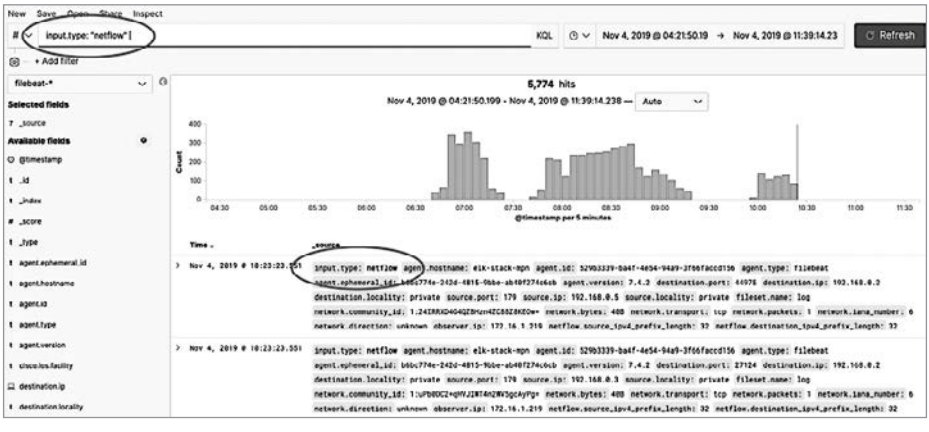


Рис. 12.16. Ввод Elastic NetFlow

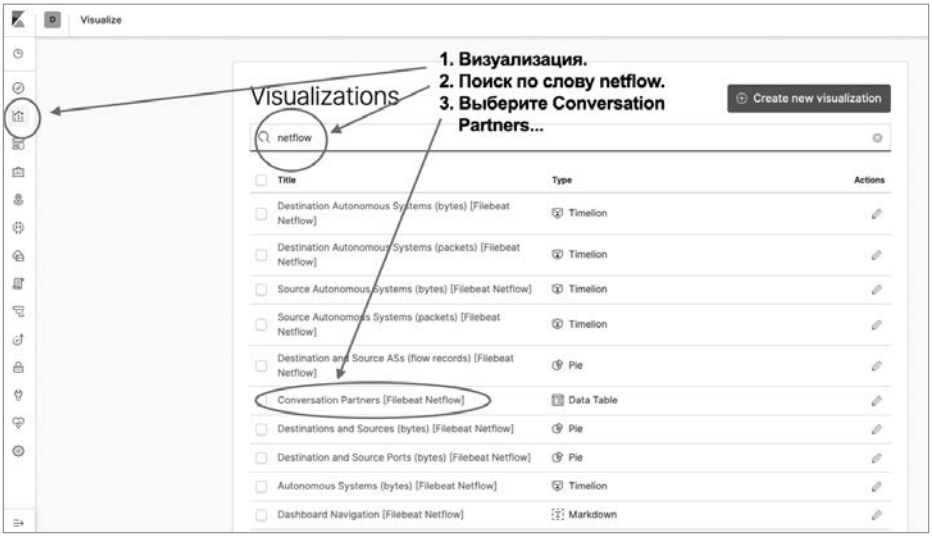


Рис. 12.17. Визуализация в Kibana

Выберите пункт Conversation Partners [Filebeat Netflow]. В результате вы получите аккуратную таблицу самых «общительных» хостов, которую можно сортировать по любому полю (рис. 12.18).

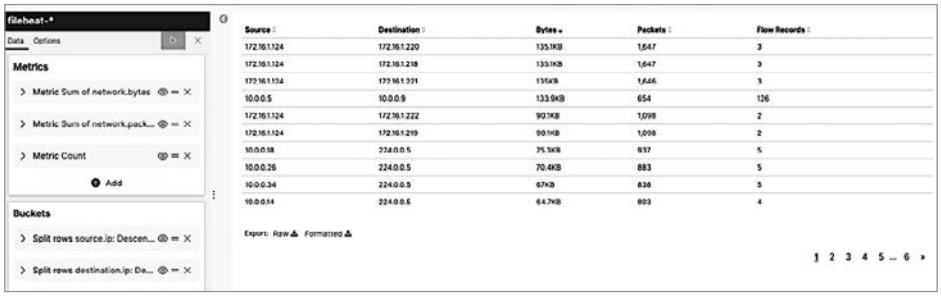


Рис. 12.18. Таблица в Kibana

Если вы хотите использовать стек ELK для мониторинга NetFlow, попробуйте проект ElastiFlow: https://github.com/robcowart/elastiflow.

В следующем разделе уделим внимание еще одному компоненту стека ELK — Elasticsearch.

Поиск с помощью Elasticsearch

Нам нужно больше данных, чтобы сделать поиск и визуализацию интереснее. Я бы посоветовал перезагрузить несколько лабораторных устройств, чтобы сгенерировать журнальные записи со сбросом интерфейсов, настройкой BGP и OSPF и с сообщениями о загрузке системы. Также можно воспользоваться данными, импортированными в начале главы.

В сценарии `Chapter12_2.py`, в котором мы выполняли поиск, в каждом запросе можно поменять два элемента: индекс и тело запроса. Я люблю оформлять такие элементы в виде входных переменных, которые можно динамически изменять при выполнении, чтобы отделить логику поиска от самого сценария. Давайте создадим файл с именем `query_body_1.json`:

```
{
  "query": {
    "match_all": {}
  }
}
```

И напишем сценарий `Chapter12_3.py`, принимающий аргументы пользователя из командной строки с помощью модуля `argparse`:

```
import argparse
parser = argparse.ArgumentParser(description='Elasticsearch Query Options')
parser.add_argument("-i", "--index", help="index to query")
parser.add_argument("-q", "--query", help="query file")

args = parser.parse_args()
```

Затем используем два входных значения для конструирования поискового запроса, как мы это делали ранее:

```
# загружаем индекс Elastic и тело запроса
query_file = args.query
with open(query_file) as f:
    query_body = json.loads(f.read())

# экземпляр Elasticsearch
es = Elasticsearch(['http://192.168.2.200:9200'])

# конструируем запрос, выполняем и помещаем результат в словарь
index = args.index
res = es.search(index=index, body=query_body)
print(res['hits']['total']['value'])
```

Чтобы узнать, какие параметры следует передать сценарию, его можно вызвать с параметром `help`. Вот результаты выполнения одного и того же запроса для двух разных индексов, которые мы создали:

```
(venv) $ python3 Chapter12_3.py --help
usage: Chapter12_3.py [-h] [-i INDEX] [-q QUERY]

Elasticsearch Query Options

optional arguments:
  -h, --help            show this help message and exit
  -i INDEX, --index INDEX
                        index to query
  -q QUERY, --query QUERY
                        query file
(venv) $ python3 Chapter12_3.py -q query_body_1.json -i "cisco*"
50
(venv) $ python3 Chapter12_3.py -q query_body_1.json -i "filebeat*"
10000
```

При разработке функции поиска на отладку обычно уходит несколько попыток. Kibana предоставляет консоль для разработки, которая позволяет экспериментировать с критериями поиска и просматривать результаты на одной странице. Например, на следующем снимке экрана (рис. 12.19) мы выполняем тот же поиск, что и выше, и сразу получаем результат в формате JSON. Это один из моих любимых инструментов в интерфейсе Kibana.

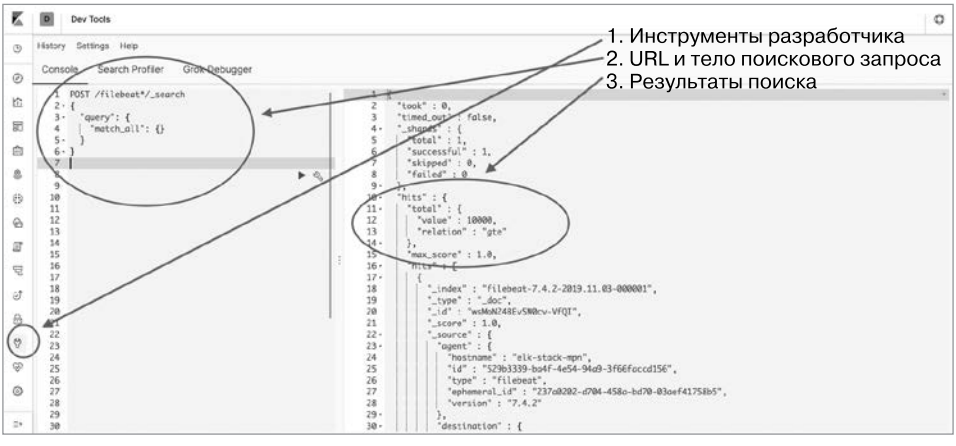


Рис. 12.19. Инструменты для разработки в Kibana

Значительная часть сетевых данных привязана ко времени; это, к примеру, касается собранных нами журнальных записей и потока NetFlow. Значения берутся в какой-то определенный момент времени, поэтому их, скорее всего, имеет смысл группировать хронологически. Например, мы можем поинтересоваться: «Какое устройство сгенерировало наибольший трафик NetFlow за последние семь дней?» или «У какого устройства больше всего сообщений о сбросе BGP за последний час?». Большинство подобных вопросов имеют отношение к агрегированию и временным интервалам. Рассмотрим запрос в `query_body_2.json`, ограничивающий временной диапазон:

```
{
  "query": {
    "bool": {
    "filter": [
      {
        "range": {
          "@timestamp": {
            "gte": "now-10m"
          }
        }
      }
    ]
    }
  }
}
```

Это булев запрос (см. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-bool-query.html), то есть он может сочетать в себе другие запросы. Мы используем фильтр для ограничения временного диапазона десятью последними

минутами. Скопируем сценарий `Chapter12_3.py` в файл `Chapter12_4.py` и отредактируем вывод так, чтобы получить количество попаданий (`hits`), а затем просмотрим содержимое полученного списка с результатами:

```
<опущено>
res = es.search(index=index, body=query_body)
print("Total hits: " + str(res['hits']['total']['value']))
for hit in res['hits']['hits']:
    pprint(hit)
```

Как показал этот сценарий, за последние 10 минут найдено всего 68 попаданий:

```
(venv) $ python3 Chapter12_4.py -i "filebeat*" -q query_body_2.json
Total hits: 68
```

Добавим в фильтр еще один параметр, чтобы указать нужный нам исходящий IP-адрес (`query_body_3.json`):

```
{
  "query": {
    "bool": {
      "must": {
        "term": {
          "source.ip": "192.168.0.1"
        }
      },
<опущено>
```

Результат будет отфильтрован как по исходящему локальному адресу интерфейса R1, так и по времени (последние 10 минут):

```
(venv) $ python3 Chapter12_4.py -i "filebeat*" -q query_body_3.json
Total hits: 18
```

Еще раз изменим тело запроса и добавим агрегирование (см. https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket.html), чтобы получить сумму байтов, переданных в предыдущей попытке поиска:

```
{
  "aggs": {
        "network_bytes_sum": {
          "sum": {
            "field": "network.bytes"
        }
    }
  },
  <опущено>
}
```

При каждом выполнении сценария `Chapter12_5.py` мы будем получать другой результат. Размер ответов, которые я получаю при последовательном выполнении, — около 1 Мбайт:

```
(venv) $ python3 Chapter12_5.py -i "filebeat*" -q query_body_4.json
1089.0
(venv) $ python3 Chapter12_5.py -i "filebeat*" -q query_body_4.json
990.0
```

Как видите, конструирование поискового запроса — это постепенный процесс, в ходе которого вы сужаете критерии, чтобы отсеять ненужные результаты. Вначале вы наверняка будете тратить много времени на чтение документации и поиск нужного синтаксиса и фильтров. Но по мере обретения опыта вы будете все лучше ориентироваться в этом процессе. Вернемся к предыдущему примеру визуализации с модулем `netflow` для поиска самых активных хостов, генерирующих наибольший трафик NetFlow, и воспользуемся инспектором для просмотра поля запроса (рис. 12.20).

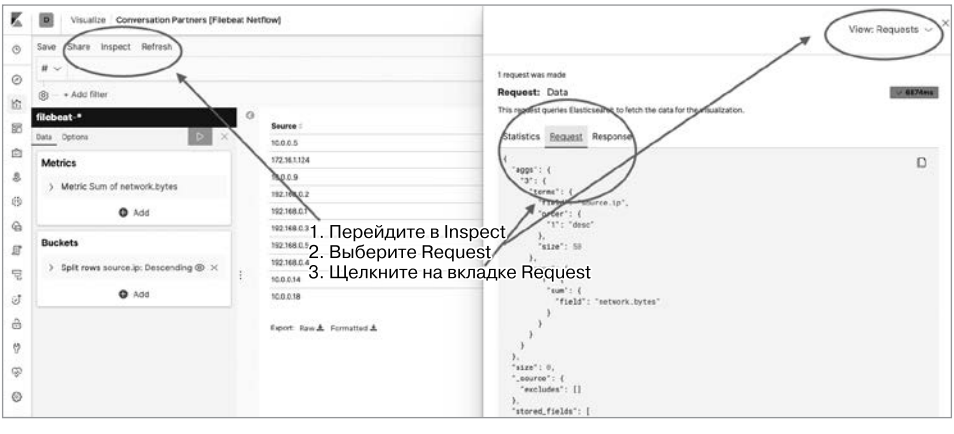


Рис. 12.20. Запрос в Kibana

Поместим этот запрос в JSON-файл `query_body_5.json` и выполним его с помощью сценария `Chapter12_6.py`. В ответ получим исходные данные для диаграммы:

```
(venv) $ python3 Chapter12_6.py -i "filebeat*" -q query_body_5.json
{'1': {'value': 8156040.0}, 'doc_count': 8256, 'key': '10.0.0.5'}
{'1': {'value': 4747596.0}, 'doc_count': 103, 'key': '172.16.1.124'}
{'1': {'value': 3290688.0}, 'doc_count': 8256, 'key': '10.0.0.9'}
{'1': {'value': 576446.0}, 'doc_count': 8302, 'key': '192.168.0.2'}
{'1': {'value': 576213.0}, 'doc_count': 8197, 'key': '192.168.0.1'}
{'1': {'value': 575332.0}, 'doc_count': 8216, 'key': '192.168.0.3'}
{'1': {'value': 433260.0}, 'doc_count': 6547, 'key': '192.168.0.5'}
{'1': {'value': 431820.0}, 'doc_count': 6436, 'key': '192.168.0.4'}
```

В следующем разделе мы подробно обсудим компонент визуализации в стеке Elastic — Kibana.

Визуализация данных с использованием Kibana

До сих пор мы использовали Kibana для обнаружения данных, управления индексами в Elasticsearch и других задач, таких как создание запросов с помощью инструментов для разработки. В готовых примерах визуализации с NetFlow показывалась самая активная пара хостов, сгенерировавшая наибольший трафик по этому протоколу. В этом разделе мы разберем создание собственных визуализаций. Начнем с круговой диаграммы.

Круговая диаграмма отлично подходит для отображения доли, занимаемой некоторым отдельным элементом. Возьмем за основу индекс Filebeat с десятью исходящими IP-адресами с самым большим количеством записей. Выберем Visualization ▸ New Visualization ▸ Pie (Визуализация ▸ Новая визуализация ▸ Круговая диаграмма) (рис. 12.21).

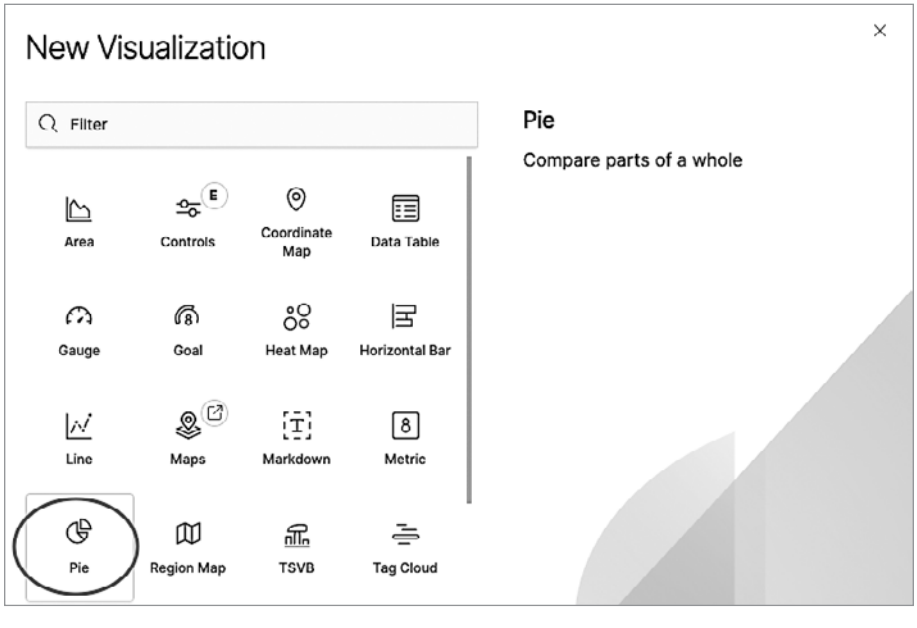


Рис. 12.21. Круговая диаграмма в Kibana

Затем введем в поле поиска слово *netflow*, чтобы получить наши индексы [Filebeat NetFlow] (рис. 12.22).

Рис. 12.22. Источник данных для круговой диаграммы

По умолчанию мы получаем количество всех записей за стандартный промежуток времени. Но его можно изменить (рис. 12.23).

Рис. 12.23. Временной интервал в Kibana

Мы можем назначить диаграмме метку (рис. 12.24).

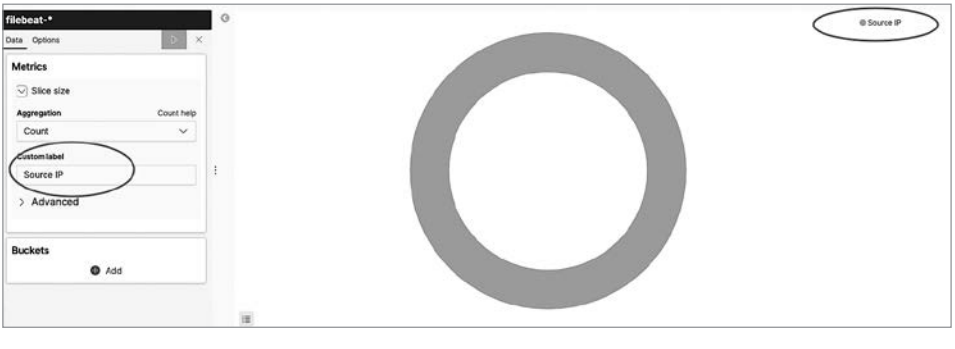


Рис. 12.24. Метка для диаграммы в Kibana

Нажмем кнопку Add (Добавить), чтобы добавить дополнительные данные. Перейдем в раздел Split slices (Разделить части) и выберем в раскрывающемся списке критерий агрегирования source.ip. Оставим в поле Order (Сортировка) значение Descending (По убыванию), но увеличим Size (Размер) до 10.

Изменения вступят в силу только после нажатия кнопки Apply (Применить) вверху. Многие, опираясь на опыт работы с современными веб-сайтами, по привычке ошибочно думают, что изменения сохраняются в реальном времени, без нажатия каких-либо кнопок (рис. 12.25).

Мы можем щелкнуть на ссылке Options (Параметры) вверху, чтобы сбросить переключатель Donut (Кольцо) и включить Show labels (Показывать метки) (рис. 12.26).

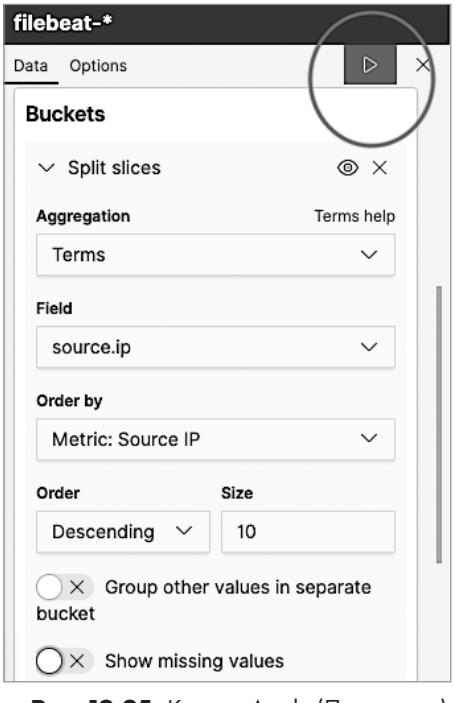


Рис. 12.25. Кнопка Apply (Применить) в Kibana

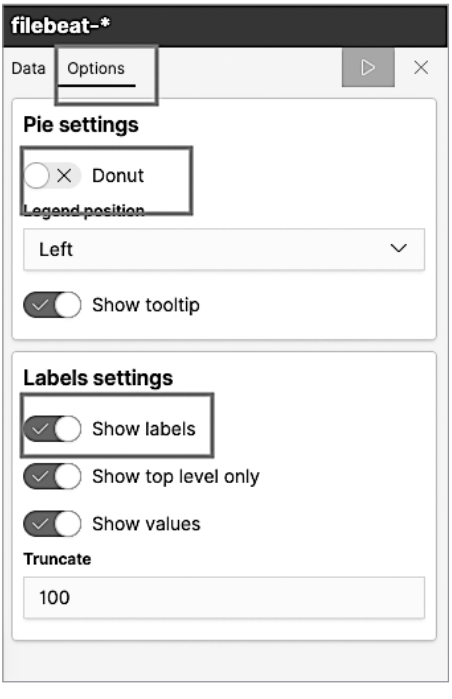


Рис. 12.26. Параметры диаграммы в Kibana

В итоге у нас получилась красивая круговая диаграмма, на которой изображены исходящие IP-адреса, отправившие наибольшее количество документов (рис. 12.27).

Как и в случае с поиском в Elasticsearch, создание диаграммы в Kibana — это постепенный процесс, который обычно требует нескольких попыток. Что, если мы выведем результаты в виде отдельных диаграмм, а не как части одной и той же диаграммы? Результат получится не очень привлекательным (рис. 12.28).

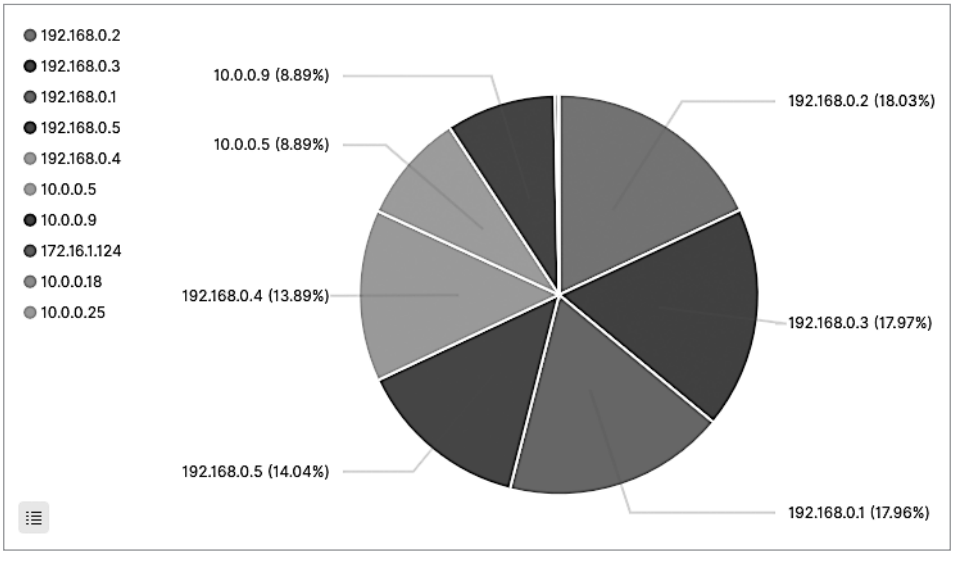


Рис. 12.27. Круговая диаграмма в Kibana

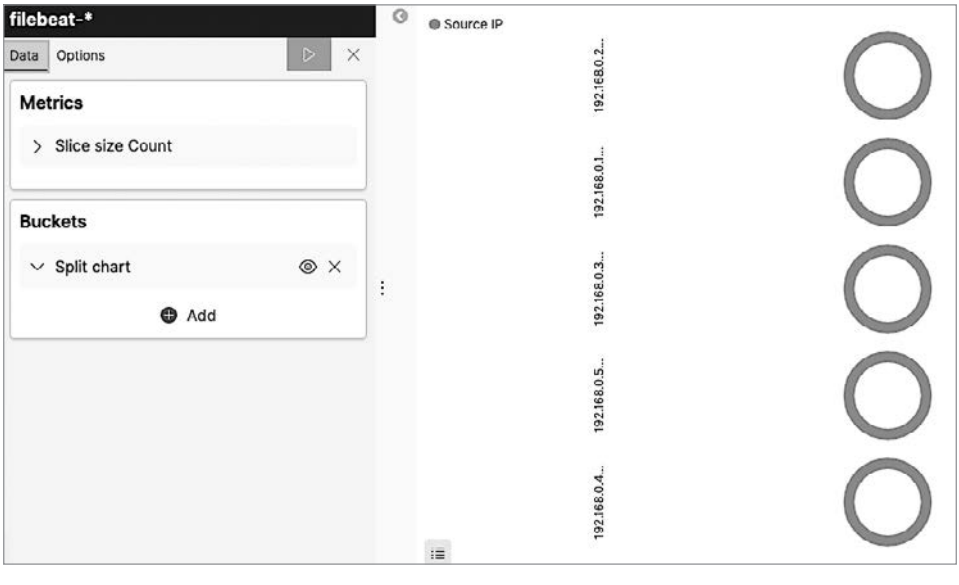


Рис. 12.28. Разделенная диаграмма в Kibana

Вернемся к объединенной диаграмме, в поле диапазона времени выберем Last 1 hour (За последний час) и сохраним диаграмму, чтобы вернуться к ней позже.

Обратите внимание: мы можем поделиться диаграммой с коллегами, передав внутренний URL (если экземпляр Kibana доступен с общего ресурса) или снимок (рис. 12.29).

Рис. 12.29. Сохранение диаграммы в Kibana

Мы также можем выполнить действия с метриками. Например, выбрать для диаграммы табличный тип и повторить суммирование данных по исходящим IP-адресам. У нас также есть возможность добавить вторую метрику с суммой переданных байтов из каждого адреса (рис. 12.30).

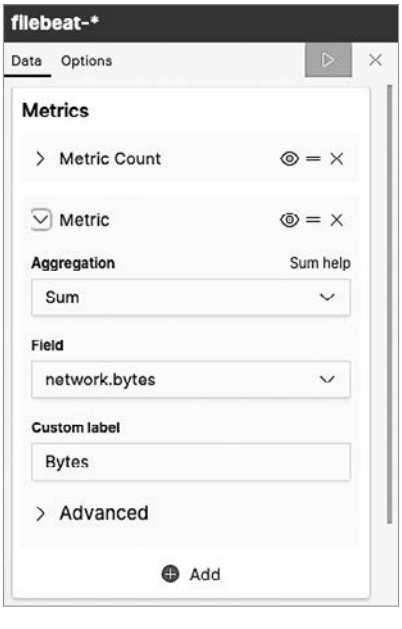


Рис. 12.30. Метрики Kibana

В результате получится таблица с количеством документов и переданных байтов. Ее можно сохранить локально в формате CVS (рис. 12.31).

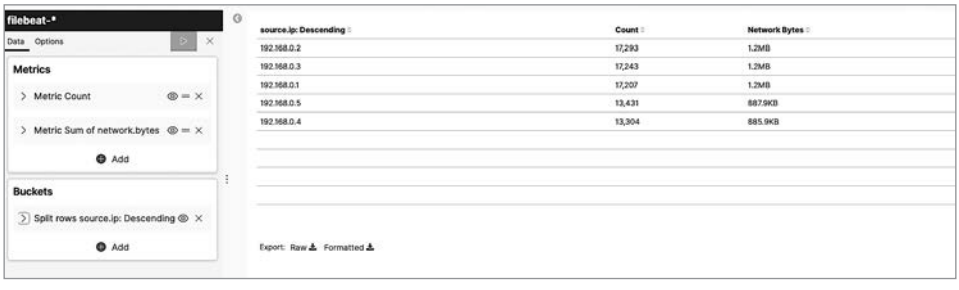


Рис. 12.31. Таблицы в Kibana

Система Kibana — очень мощное средство визуализации в составе Elastic Stack. Мы лишь вскользь прошлись по ее возможностям. Она поддерживает множество типов диаграмм для визуализации ваших данных, и на одной информационной панели можно вывести сразу несколько диаграмм. Используйте Timelion (https://www.elastic.co/guide/en/kibana/7.4/timelion.html) для группировки независимых источников данных на одной диаграмме или Canvas (https://www.elastic.co/guide/en/kibana/current/canvas.html) для представления данных из Elasticsearch.

Kibana обычно применяется на заключительном этапе рабочего процесса для осмысленного представления данных. В этой главе мы обсудили основные этапы: прием, сохранение, извлечение и отображение данных. Меня до сих пор поражает, сколько всего можно сделать за короткое время, используя интегрированный открытый стек технологий наподобие Elastic Stack.

Резюме

В этой главе мы использовали Elastic Stack для приема, анализа и визуализации сетевых данных. Увидели, как принимаются журнальные записи и потоки NetFlow с помощью Logstash и Beats. Как эта информация индексируется и делится на категории в Elasticsearch, чтобы потом ее было легче извлекать. И наконец, как она визуализируется с помощью Kibana. Мы также использовали Python для взаимодействия с этим стеком и более подробного анализа наших данных. Вместе Logstash, Beats, Elasticsearch и Kibana составляют мощный полнофункциональный проект, который помогает анализировать информацию.

В следующей главе мы посмотрим, как использовать Git в разработке средств автоматизации сети на Python.

13

Работа с Git

Мы уже имели дело с различными аспектами сетевой автоматизации с применением Python, Ansible и многих других инструментов. Для упражнений в первых 12 главах использовалось более 150 файлов, содержащих более 5300 строк кода. Это неплохо для сетевых инженеров, которые до прочтения этой книги в основном работали с интерфейсом командной строки! Вооружившись этими новыми сценариями и инструментами, мы готовы ринуться в бой и автоматизировать наши рутинные сетевые задачи, правда? Не совсем. Немного остудим пыл.

Есть моменты, которые необходимо учитывать при решении реальных задач. Пройдемся по ним и поговорим о том, чем может помочь система управления версиями Git.

Эта глава охватывает следующие темы:

- Git и различные аспекты управления контентом;
- введение в Git;
- подготовку Git к работе;
- примеры с Git;
- Git в сочетании с Python;
- автоматизацию резервного копирования конфигурационных данных;
- совместную работу с использованием Git.

Итак, посмотрим, о каких аспектах речь и чем может помочь Git.

Git и разные аспекты управления контентом

В первую очередь при создании файлов с кодом следует подумать о месте хранения, откуда мы и наши коллеги сможем их извлекать. В идеале это должно быть единое центральное хранилище файлов, которое при необходимости может предоставить их резервные копии. После первого выпуска кода нам, вероятно, будет нужно добавить новые возможности и исправить ошибки, поэтому необходимо как-то отслеживать эти изменения и сделать доступными для загрузки актуальные версии. Если новая версия кода не работает, не помешает возможность ее откатить и увидеть отличия в истории изменений файла. Это позволит нам ориентироваться в том, как обновлялся тот или иной код.

Второй аспект — совместная работа членов команды. Работая в сотрудничестве с другими сетевыми инженерами, нам, скорее всего, придется править одни и те же файлы: сценарии Python и Ansible, шаблоны Jinja2, конфигурационные файлы в формате INI и т. д. Правки, вносимые в любые текстовые файлы разными людьми, должны быть видны всем членам команды.

Третий аспект — это учет. Система, позволяющая разным людям редактировать одни и те же файлы, должна иметь историю правок с указанием автора каждого изменения и краткого описания причины, по которой изменение было сделано, чтобы человек, просматривающий историю, смог понять, зачем вносилась та или иная правка.

Это основные задачи, которые решают системы управления версиями, такие как Git. Справедливости ради стоит отметить, что процесс управления версиями может быть реализован не только в виде отдельной программной системы. Например, в Microsoft Word документ постоянно автоматически сохраняется и я могу вернуться назад во времени, чтобы просмотреть изменения, или откатиться к предыдущей версии. Это тоже разновидность управления версиями; однако документ Word обычно ограничен моим ноутбуком. Система управления версиями, которой посвящена эта глава, — это отдельный программный инструмент, основное назначение которого заключается в отслеживании изменений, вносимых в код.

В мире программирования нет недостатка в инструментах управления исходным кодом. Есть и коммерческие версии, и открытые. Самые популярные открытые системы управления версиями: CVS, SVN, Mercurial и Git. В этой главе мы сосредоточимся на Git. Многие примеры, представленные в этой книге, хранятся в одной и той же системе управления версиями, позволяющей отслеживать

изменения, совместно работать над ними и обмениваться мнениями с пользователями. Мы выбрали Git, так как это фактически стандартная система управления версиями для многих крупных проектов с открытым исходным кодом, включая Python и ядро Linux.

В феврале 2017 года разработка проекта CPython полностью переместилась в GitHub. Переход на эту систему начался в январе 2015 года. Подробнее об этом читайте в PEP 512 по адресу https://www.python.org/dev/peps/pep-0512/.

Прежде чем переходить к примерам работы с Git, поговорим об истории и преимуществах этой системы.

Введение в Git

Проект Git был создан в апреле 2005 года Линусом Торвальдсом, автором ядра Linux. В остроумной манере Линус ласково назвал его «диспетчером информации из ада». В интервью организации Linux Foundation он упомянул, что, по его мнению, управление исходным кодом было наименее интересной вещью в мире компьютеров (https://www.linuxfoundation.org/blog/2015/04/10-years-of-git-an-interview-with-git-creator-linus-torvalds/). Тем не менее, когда между сообществом разработчиков ядра Linux и проектом BitKeeper (коммерческой системой, которую они использовали в то время) возникли разногласия, Линус решил создать Git.

Что означает название Git? В британском английском сленге git — это уничижительный термин, который применяют по отношению к неприятным, надоедливым, незрелым людям. В невозмутимой манере Линус заявил, что он эгоист и все свои проекты называет в свою честь. Сначала Linux, теперь Git. Но есть мнение, что это аббревиатура от Global Information Tracker (глобальная система отслеживания информации). Выбирайте тот вариант, который вам больше по душе.

Проект был воплощен в жизнь в очень короткие сроки. Примерно через десять дней после основания (именно так, вам не показалось) Линус выработал основные идеи системы Git и начал сохранять в нее код ядра Linux. Остальное, как говорится, уже история. Спустя более десяти лет после создания Git продолжает отвечать всем ожиданиям разработчиков Linux. Эту систему управления версиями начали применять во многих других открытых проектах, хотя, как

правило, разработчики неохотно идут на такие изменения. В феврале 2017 года после многолетнего использования Mercurial (https://hg.python.org) проект Python перешел на Git и GitHub.

Итак, мы обсудили историю создания системы Git. Теперь рассмотрим ее преимущества.

Преимущества Git

Успешный хостинг таких крупных и распределенных открытых проектов, как ядро Linux и Python, свидетельствует о преимуществах Git. Иными словами, если этот инструмент достаточно хорош для разработчиков самой популярной операционной системы (по моему мнению) и самого популярного языка программирования (опять же по моему мнению) в мире, его, скорее всего, будет достаточно для моего домашнего проекта. К тому же это относительно новая система управления версиями, а люди неохотно переходят на новые инструменты, если те не предлагают существенных улучшений по сравнению со старыми. Рассмотрим некоторые преимущества Git.

- **Распределенная разработка.** Git поддерживает параллельную, независимую и одновременную разработку в частных локальных репозиториях. Многие другие системы управления версиями требуют постоянной синхронизации с центральным репозиторием. Распределенность и локальность Git дают разработчикам больше гибкости.
- **Масштабирование до поддержки тысяч разработчиков.** Над иными открытыми проектами работают тысячи разработчиков, и Git обеспечивает надежную интеграцию результатов их труда.
- **Производительность.** Линус постарался сделать систему Git быстрой и эффективной. Чтобы сэкономить место для хранения огромных обновлений кода в ядре Linux и время для их передачи, в Git применяются сжатие и проверка различий между разными версиями.
- **Учет и неизменяемость.** Git журналирует каждую фиксацию, изменяющую файлы, формируя историю обновлений с описанием их причин. Объекты данных в Git нельзя изменить после их создания и сохранения в базе данных, что делает их неизменяемыми. Это улучшает и упрощает учет.
- **Атомарные транзакции.** Чтобы обеспечить целостность репозитория, разные, но связанные между собой изменения вносятся либо все вместе, либо не вносится ни одно из них. Благодаря этому репозиторий не может оказаться в частично измененном или поврежденном состоянии.

- **Самодостаточные репозитории.** Каждый репозиторий хранит полную копию всех правок, внесенных в каждый файл.
- **Свобода.** Проект Git возник в результате разногласий между разработчиками Linux и BitKeeper VCS относительно того, должно ли программное обеспечение быть свободным и стоит ли принципиально отвергать коммерческое ПО. Поэтому логично, что у Git очень либеральная лицензия.

Прежде чем знакомиться с Git, рассмотрим термины этой системы.

Терминология Git

Термины Git, которые вы должны знать:

- **Ссылка (ref).** Имя, которое начинается с `refs` и указывает на объект.
- **Репозиторий (repository).** Это база данных со всей информацией, файлами, метаданными и историей проекта. Она содержит ссылки на все коллекции объектов.
- **Ветвь (branch).** Это активное направление разработки. Последняя фиксация называется *верхушкой* (`tip`) или *головой* (`head`) ветви. В репозитории может быть несколько ветвей, но ваше *рабочее дерево*, или *рабочий каталог*, может быть связано только с одной из них. Такую ветвь иногда называют текущей *извлеченной* (checked out).
- **Переключение (checkout).** Это операция полного или частичного обновления рабочего дерева до определенной точки.
- **Фиксация (commit).** Это определенный момент времени в репозитории Git; также может означать процесс сохранения нового снимка в репозитории.
- **Слияние (merge).** Это операция по перемещению в текущую ветвь содержимого другой ветви. Например, я произвожу слияние ветвей `development` и `master`.
- **Получение (fetch).** Получение содержимого удаленного репозитория.
- **Извлечение (pull).** Получение и слияние репозитория.
- **Тег (tag).** Метка, назначаемая важному моменту времени, сохраненному в репозитории. В главе 4 мы видели, как с помощью меток обозначались выпуски (например, `v2.5.0a1`).

Это не полный список. Больше терминов с описанием ищите в глоссарии Git по адресу https://git-scm.com/docs/gitglossary.

Напоследок, прежде чем переходить к настройке и использованию Git, поговорим о важном различии между Git и GitHub, которое люди, незнакомые с этими двумя системами, могут упустить.

Git и GitHub

Git и GitHub — это не одно и то же. Иногда это создает путаницу среди инженеров, которые только знакомятся с этой областью. Git — это система управления версиями, а GitHub (https://github.com/) — централизованный сервис для размещения Git-репозиториев. Компания GitHub была создана в 2008 году и до сих пор сохраняет свою независимость, хотя в 2018 году ее приобрела корпорация Microsoft.

Поскольку Git — децентрализованная система, GitHub хранит лишь одну из локальных копий репозитория нашего проекта. Зачастую репозиторий GitHub делают центральным, и все разработчики сохраняют в него свои изменения.

После приобретения GitHub корпорацией Microsoft (https://blogs.microsoft.com/blog/2018/10/26/microsoft-completes-github-acquisition/) многие в сообществе разработчиков волновались о независимости этого проекта. Но, как было сказано в пресс-релизе, «GitHub сохранит свои идеалы, согласно которым разработчик стоит на первом месте, продолжит работать независимо и останется открытой платформой».

GitHub развивает идею центрального репозитория в распределенной системе за счет таких механизмов, как *форк* (fork — «создание копии») и *запрос на включение внесенных изменений* (pull request). Разработчики, сопровождающие проект в GitHub, обычно поощряют создание его копии (форк) и работу с ней как с центральным репозиторием. После внесения изменений вы можете подать запрос на включение изменений в главный проект, где их проверят и зафиксируют, если они соответствуют всем критериям. GitHub также предоставляет веб-интерфейс для репозиториев, которым можно пользоваться в дополнение к командной строке; это делает Git более удобным в работе.

Итак, определившись с различиями между Git и GitHub, мы можем приступить к главной теме! Для начала подготовим Git к работе.

Подготовка Git к работе

До сих пор мы использовали Git только для загрузки файлов из GitHub. В этом разделе мы пойдем немного дальше и настроим локальный репозиторий Git,

чтобы иметь возможность сохранять в нем наши файлы. В этом примере я использую уже знакомый вам управляющий хост с Ubuntu 18.04. Если вы используете другую версию Linux или другую операционную систему, то просто поищите в интернете соответствующие инструкции по установке.

Установите Git с помощью диспетчера пакетов `apt`, если вы этого еще не сделали:

```
(venv) $ sudo apt update
(venv) $ sudo apt install -y git
(venv) $ git --version
git version 2.17.1
```

После установки `git` выполним некоторые настройки, чтобы сообщения о фиксации содержали правильные сведения:

```
$ git config --global user.name "Ваше имя"
$ git config --global user.email "email@domain.com"
$ git config --list
user.name=Ваше имя
user.email=email@domain.com
```

Можно пойти другим путем и отредактировать файл `~/.gitconfig`:

```
$ cat ~/.gitconfig
[user]
name = Ваше имя
email = email@domain.com
```

Git поддерживает много параметров, которые можно менять, но имя и адрес электронной почты позволят фиксировать изменения без получения предупреждений. Ввод сообщений, сопровождающих фиксации, по умолчанию выполняется с помощью текстового редактора Emacs, но лично я предпочитаю использовать для этого VIM:

```
(опционально)
$ git config --global core.editor "vim"
$ git config --list
user.name=Ваше имя
user.email=email@domain.com
core.editor=vim
```

Прежде чем начинать работу с Git, обсудим файл `.gitignore`.

Gitignore

Есть файлы, которые не следует сохранять в GitHub или других репозиториях; речь идет о файлах с паролями, API-ключами и другой конфиденциальной

информацией. Чтобы не допустить их сохранения, проще всего создать в корневой папке репозитория файл `.gitignore`. Git будет использовать `.gitignore`, чтобы определить, какие файлы и каталоги следует игнорировать, прежде чем производить фиксацию. Файл `.gitignore` должен отправляться и распространяться между другими пользователями как можно раньше.

Представьте себе панику, которую можно испытать, если случайно сохранить в публичный репозиторий свой групповой API-ключ. Файл .gitignore имеет смысл создавать вместе с новым репозиторием. На самом деле эта возможность предусмотрена на платформе GitHub.

В `.gitignore` можно указать файлы, которые относятся к определенному языку. Исключим файлы с байт-кодом Python:

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class
```

Также можно указать файлы, относящиеся к нашей операционной системе:

```
# OSX
# =========================

.DS_Store
.AppleDouble
.LSOverride
```

Больше о `.gitignore` можно узнать на странице справки GitHub: https://help.github.com/articles/ignoring-files/. Вот еще несколько ссылок:

- Руководство по использованию Gitignore: https://git-scm.com/docs/gitignore.
- Набор шаблонов для `.gitignore` от GitHub: https://github.com/github/gitignore.
- Пример `.gitignore` для языка Python: https://github.com/github/gitignore/blob/master/Python.gitignore.
- Файл `.gitignore` для репозитория этой книги: https://github.com/PacktPublishing/Mastering-Python-Networking-Third-Edition/blob/master/.gitignore.

По моему мнению, файл `.gitignore` следует создавать одновременно с любым новым репозиторием. Это позволит внедрить данную концепцию как можно раньше. В следующем разделе мы рассмотрим некоторые примеры использования Git.

Примеры работы с Git

По моему опыту, для работы с Git в основном используют командную строку и различные параметры. Графические инструменты могут пригодиться для отслеживания изменений, просмотра истории и сравнения разных версий, но они редко применяются для таких рутинных операций, как ветвление и фиксация. Справку по параметрам командной строки Git можно получить с помощью команды `git --help`:

```
(venv) $ git --help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

Создадим репозиторий и файл внутри него:

```
(venv) $ mkdir TestRepo-1
(venv) $ cd TestRepo-1/
(venv) $ git init
Initialized empty Git repository in /home/echou/Mastering_Python_
Networking_third_edition/Chapter13/TestRepo-1/.git/
(venv) $ echo "this is my test file" > myFile.txt
```

При инициализации репозитория с помощью Git была создана новая скрытая папка `.git`. Она содержит файлы, относящиеся к Git:

```
(venv) $ ls -a
.  ..  .git  myFile.txt
(venv) $ ls .git/
branches  config  description  HEAD  hooks  info  objects  refs
```

Git имеет иерархическую конфигурацию, которая хранится в разных местах. По умолчанию она состоит из файлов *системного* и *глобального* уровней, а также уровня *репозитория*. Чем ниже уровень, тем выше его приоритет. Например, конфигурация репозитория имеет преимущество перед глобальной конфигурацией. Чтобы увидеть агрегированную конфигурацию, используйте команду `git config -l`:

```
$ ls .git/config
.git/config

$ ls ~/.gitconfig
/home/echou/.gitconfig

$ git config -l
```

```
user.name=Eric Chou
user.email=<email>
core.editor=vim
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
```

Файл, который мы создали в репозитории, не отслеживается автоматически. Его нужно добавить в систему Git, чтобы она о нем узнала:

```
$ git status
On branch master
Initial commit

Untracked files:
     (use "git add <file>..." to include in what will be committed)

myFile.txt

nothing added to commit but untracked files present (use "git add" to track)

$ git add myFile.txt
$ git status
On branch master

Initial commit

Changes to be committed:
   (use "git rm --cached <file>..." to unstage)

new file: myFile.txt
```

Добавленный файл находится в промежуточном (staged) состоянии. Чтобы сделать изменения окончательными, их нужно зафиксировать:

```
$ git commit -m "adding myFile.txt"
[master (root-commit) 5f579ab] adding myFile.txt
 1 file changed, 1 insertion(+)
 create mode 100644 myFile.txt

$ git status
On branch master
nothing to commit, working directory clean
```

В последнем примере при выполнении команды commit мы указали сообщение, сопровождающее фиксацию, с помощью параметра -m. Если бы мы опустили этот параметр, нам все равно пришлось бы ввести это сообщение, но уже в текстовом редакторе. Мы для ввода сообщения используем редактор Vim.

Отредактируем файл и снова его зафиксируем. Обратите внимание: Git заметил изменения в файле:

```
$ vim myFile.txt
$ cat myFile.txt
this is the second iteration of my test file
$ git status
On branch master
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

modified: myFile.txt
$ git add myFile.txt
$ git commit -m "made modifications to myFile.txt"
[master a3dd3ea] made modifications to myFile.txt
1 file changed, 1 insertion(+), 1 deletion(-)
```

Номер фиксации в Git представляет собой хеш SHA-1м, это важная особенность. Если выполнить тот же шаг на другом компьютере, значение хеша будет таким же. Благодаря этому Git знает, что два репозитория идентичны, даже если с ними работают параллельно.

Если вам интересно, можно ли изменить значение SHA-1 случайно или умышленно так, чтобы оно совпало с другим хешем, прочитайте интересную статью о подобных коллизиях в блоге GitHub, https://github.blog/2017-03-20-sha-1-collision-detection-on-github-com/.

Историю фиксаций можно вывести с помощью команды `git log`. Записи отображаются в обратном хронологическом порядке; каждая фиксация содержит имя и адрес электронной почты автора, дату, сообщение и внутренний идентификационный номер:

```
(venv) $ git log
commit ff7dc1a40e5603fed552a3403be97addefddc4e9 (HEAD -> master)
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 08:49:02 2019 -0800

    made modifications to myFile.txt

commit 5d7c1c8543c8342b689c66f1ac1fa888090ffa34
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 08:46:32 2019 -0800

    adding myFile.txt
```

Просмотрим подробности об изменении, указав идентификатор фиксации:

```
(venv) $ git show ff7dc1a40e5603fed552a3403be97addefddc4e9
commit ff7dc1a40e5603fed552a3403be97addefddc4e9 (HEAD -> master)
```

```
Author: Eric Chou <echou@yahoo.com>
Date: Fri Nov 8 08:49:02 2019 -0800

    made modifications to myFile.txt

diff --git a/myFile.txt b/myFile.txt
index 6ccb42e..69e7d47 100644
--- a/myFile.txt
+++ b/myFile.txt
@@ -1 +1 @@
-this is my test file
+this is the second iteration of my test file
```

Если понадобится отменить внесенные изменения, можно использовать одну из двух команд: `revert` или `reset`. Первая приводит все файлы в состояние, в котором они находились до определенной фиксации:

```
(venv) $ git revert ff7dc1a40e5603fed552a3403be97addefddc4e9
[master 75921be] Revert "made modifications to myFile.txt"
 1 file changed, 1 insertion(+), 1 deletion(-)

(venv) $ cat myFile.txt
this is my test file
```

Команда `revert` выполняет новую фиксацию, не удаляя старую. Вы увидите все изменения, внесенные к этому моменту, включая отмену:

```
(venv) $ git log
commit 75921bedc83039ebaf70c90a3e8d97d65a2ee21d (HEAD -> master)
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 09:00:23 2019 -0800

    Revert "made modifications to myFile.txt"

    This reverts commit ff7dc1a40e5603fed552a3403be97addefddc4e9.

     On branch master
     Changes to be committed:
            modified:   myFile.txt
```

Команда `reset` откатывает состояние репозитория до более старой версии и удаляет все более новые изменения:

```
(venv) $ git reset --hard ff7dc1a40e5603fed552a3403be97addefddc4e9
HEAD is now at ff7dc1a made modifications to myFile.txt
(venv) $ git log
commit ff7dc1a40e5603fed552a3403be97addefddc4e9 (HEAD -> master)
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 08:49:02 2019 -0800

    made modifications to myFile.txt
```

```
commit 5d7c1c8543c8342b689c66f1ac1fa888090ffa34
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 08:46:32 2019 -0800

    adding myFile.txt
```

Лично я предпочитаю хранить историю всех изменений, включая откаты. И когда мне нужно отменить какое-то изменение, я обычно выбираю `revert` вместо `reset`. В этом разделе вы увидели, как работать с отдельными файлами. Теперь обсудим наборы файлов, сгруппированных в пакет, который называется ветвью (branch).

Ветви в Git

Ветвь в Git — это направление разработки внутри репозитория. Git позволяет иметь много ветвей и, следовательно, заниматься разработкой в разных направлениях. По умолчанию у нас есть главная ветвь, `master`. Для создания ветвей есть много причин, но жестких правил, когда создавать новую ветвь или когда работать только с главной ветвью `master`, не существует. В большинстве случаев ветвь создается для исправления ошибки, выпуска публичной версии ПО или начала нового этапа разработки. Создадим ветвь, представляющую процесс разработки, и назовем ее соответствующим образом — `dev`:

```
(venv) $ git branch dev
(venv) $ git branch
  dev
* master
```

Обратите внимание: после создания ветви `dev` мы должны явно в нее перейти. Для этого есть команда `checkout`:

```
(venv) $ git checkout dev
Switched to branch 'dev'
(venv) $ git branch
* dev
  master
```

Добавим в ветвь `dev` второй файл:

```
(venv) $ echo "my second file" > mySecondFile.txt
(venv) $ git add mySecondFile.txt
(venv) $ git commit -m "added mySecondFile.txt to dev branch"
[dev a537bdc] added mySecondFile.txt to dev branch
 1 file changed, 1 insertion(+)
 create mode 100644 mySecondFile.txt
```

Мы можем вернуться в ветвь `master` и убедиться, что у нас теперь есть два направления разработки. После переключения в ветвь `master` в каталоге остается один файл:

```
(venv) $ git branch
* dev
  master
(venv) $ git checkout master
Switched to branch 'master'
(venv) $ ls
myFile.txt
(venv) $ git checkout dev
Switched to branch 'dev'
(venv) $ ls
myFile.txt  mySecondFile.txt
```

Чтобы перенести содержимое ветви `dev` в ветвь `master`, нужно произвести слияние (merge):

```
(venv) $ git branch
* dev
  master
(venv) $ git checkout master
Switched to branch 'master'
(venv) $ git merge dev master
Updating ff7dc1a..a537bdc
Fast-forward
 mySecondFile.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 mySecondFile.txt
(venv) $ git branch
  dev
* master
(venv) $ ls
myFile.txt  mySecondFile.txt
```

Удалить файл можно командой `git rm`. Создадим третий файл и попробуем его удалить:

```
(venv) $ touch myThirdFile.txt
(venv) $ git add myThirdFile.txt
(venv) $ git commit -m "adding myThirdFile.txt"
[master 169a203] adding myThirdFile.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 myThirdFile.txt
(venv) $ ls
myFile.txt  mySecondFile.txt  myThirdFile.txt
(venv) $ git rm myThirdFile.txt
rm 'myThirdFile.txt'
(venv) $ git status
On branch master
```

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    myThirdFile.txt
(venv) $ git commit -m "deleted myThirdFile.txt"
[master 1b24b4e] deleted myThirdFile.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 myThirdFile.txt
```

Два последних изменения можно увидеть в журнале:

```
(venv) $ git log
commit 1b24b4e95eb0c01cc9a7124dc6ac1ea37d44d51a (HEAD -> master)
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 10:02:45 2019 -0800

    deleted myThirdFile.txt

commit 169a2034fb9844889f5130f0e42bf9c9b7c08b05
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 10:00:56 2019 -0800

    adding myThirdFile.txt
```

Мы прошлись по основным операциям с Git, теперь посмотрим, как опубликовать репозиторий в GitHub.

Пример работы с GitHub

В этом примере мы используем GitHub в качестве центра синхронизации нашего репозитория и предоставления доступа к нему другим пользователям.

Создадим репозиторий на сайте GitHub. Для публичных проектов с открытым исходным кодом эта операция всегда была бесплатной. А с января 2019-го появилась возможность создавать неограниченное количество бесплатных частных репозиториев. Мы создадим частный репозиторий и добавим в него лицензию и файл `.gitignore` (рис. 13.1).

После создания репозитория мы получим его URL (рис. 13.2).

Воспользуемся этим URL для создания удаленной цели, которая будет служить «источником истины» для нашего проекта. Назовем эту цель `gitHubRepo`:

```
(venv) $ git remote add gitHubRepo https://github.com/ericchou1/TestRepo.
git
(venv) $ git remote -v
gitHubRepo   https://github.com/ericchou1/TestRepo.git (fetch)
gitHubRepo   https://github.com/ericchou1/TestRepo.git (push)
```

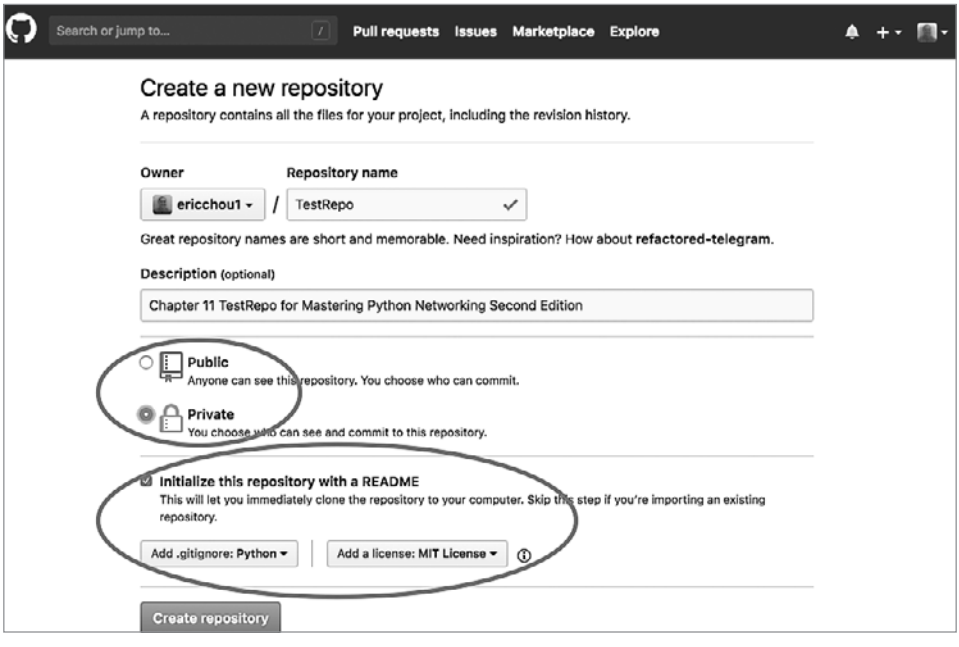


Рис. 13.1. Создание частного репозитория в GitHub

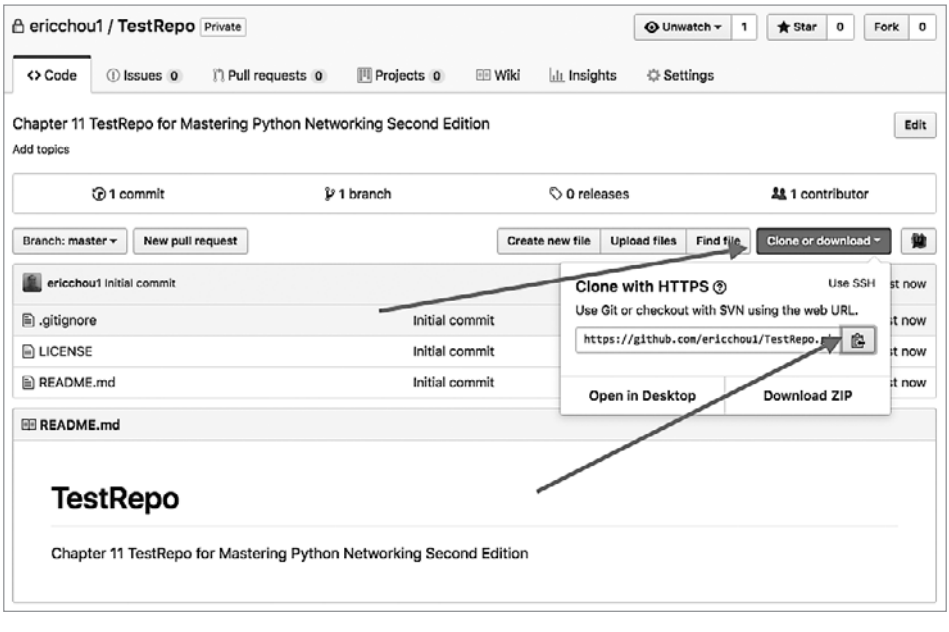


Рис. 13.2. URL репозитория в GitHub

При создании мы добавили файлы `README.md` и `LICENSE`, поэтому удаленный и локальный репозитории будут различаться.

При попытке сохранить изменения в репозиторий GitHub сообщит об ошибке:

```
(venv) $ git push gitHubRepo master
Username for 'https://github.com': <опущено>
Password for 'https://echou@yahoo.com@github.com': <опущено>
To https://github.com/ericchou1/TestRepo.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/ericchou1/
TestRepo.git'
```

Чтобы этого не случилось, загрузим новые файлы из GitHub командой `git pull`:

```
(venv) $ git pull gitHubRepo master
Username for 'https://github.com': <опущено>
Password for 'https://<username>@github.com': <опущено>
From https://github.com/ericchou1/TestRepo
* branch master -> FETCH_HEAD
Merge made by the 'recursive' strategy.
.gitignore | 104
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ LICENSE |
21 ++++++++++++++
README.md | 2 ++
3 files changed, 127 insertions(+)
create mode 100644 .gitignore
create mode 100644 LICENSE
create mode 100644 README.md
```

Теперь наше содержимое можно сохранить в GitHub командой `push`:

```
$ git push gitHubRepo master
Username for 'https://github.com': <username>
Password for 'https://<username>@github.com':
Counting objects: 15, done.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (15/15), 1.51 KiB | 0 bytes/s, done. Total 15
(delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/ericchou1/TestRepo.git a001b81..0aa362a master ->
master
```

Текущее содержимое репозитория GitHub можно просмотреть на веб-странице (рис. 13.3).

Другой пользователь может просто скопировать, или клонировать (clone), этот репозиторий:

```
[Следующие действия выполняются на другом хосте]
$ cd /tmp
```

```
$ git clone https://github.com/ericchou1/TestRepo.git
Cloning into 'TestRepo'...
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 20 (delta 2), reused 15 (delta 1), pack-reused 0
Unpacking objects: 100% (20/20), done.
$ cd TestRepo/
$ ls
LICENSE myFile.txt
README.md mySecondFile.txt
```

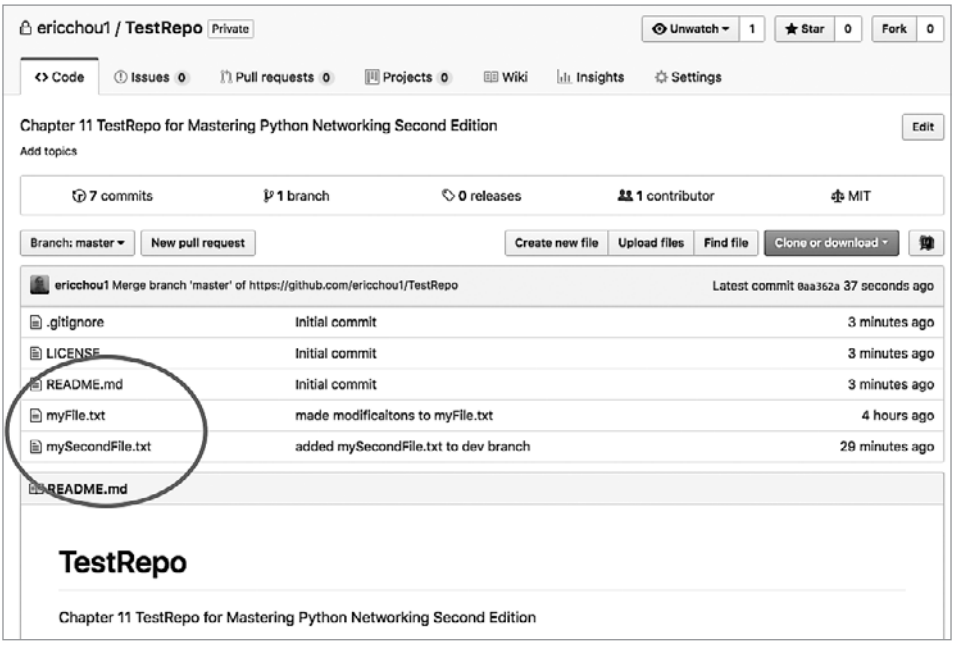


Рис. 13.3. Репозиторий GitHub

Этот скопированный репозиторий будет идентичен оригиналу, включая историю всех фиксаций:

```
$ git log
commit 0aa362a47782e7714ca946ba852f395083116ce5 (HEAD -> master,
origin/master, origin/HEAD)
Merge: bc078a9 a001b81
Author: Eric Chou <опущено>
Date: Fri Jul 20 14:18:58 2018 -0700

    Merge branch 'master' of https://github.com/ericchou1/TestRepo

commit a001b816bb75c63237cbc93067dffcc573c05aa2
Author: Eric Chou <опущено>
```

```
Date: Fri Jul 20 14:16:30 2018 -0700

    Initial commit
...
```

В настройках репозитория можно пригласить еще одного человека для совместной работы над проектом (рис. 13.4).

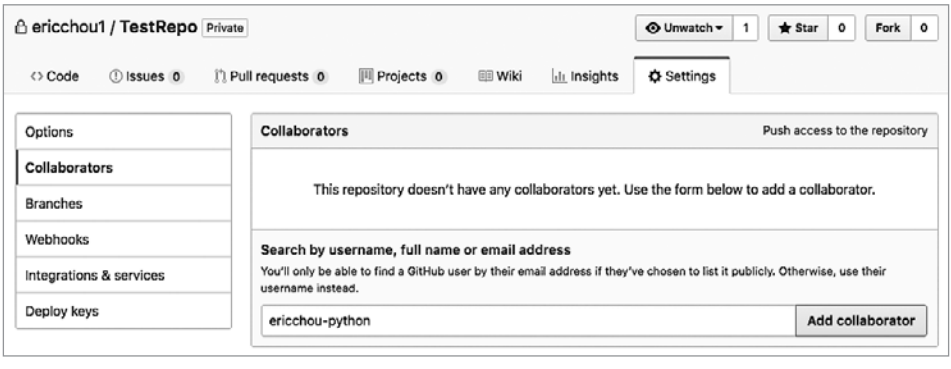


Рис. 13.4. Приглашение в репозиторий

В следующем примере вы увидите, как создать форк и выполнить запрос на внесение изменений в чужой репозиторий.

Совместная работа с использованием запросов на внесение изменений

Как уже упоминалось, Git поддерживает совместную работу нескольких разработчиков над одним проектом. Посмотрим, как это делается, когда код размещен в GitHub.

Мы будем использовать публичный GitHub-репозиторий для второго издания этой книги. Я войду от имени другого пользователя, поэтому у меня не будет административных привилегий. Я нажму кнопку Fork, чтобы скопировать репозиторий в свою учетную запись (рис. 13.5).

На создание копии уйдет несколько секунд (рис. 13.6).

По окончании в моей учетной записи появится копия репозитория (рис. 13.7).

Файлы в копии можно изменять, выполняя уже знакомые нам шаги. Я отредактирую файл `README.md`. После внесения изменений можно нажать кнопку New pull request (Новый запрос на внесение изменений) (рис. 13.8).

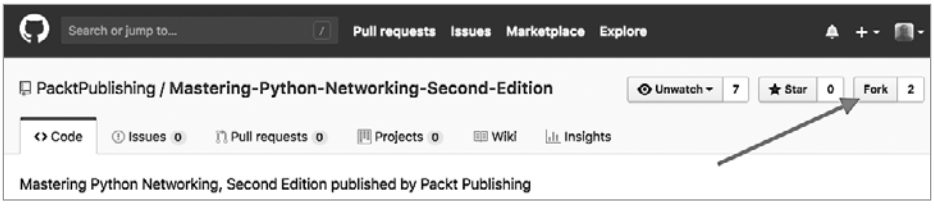


Рис. 13.5. Кнопка Fork в GitHub

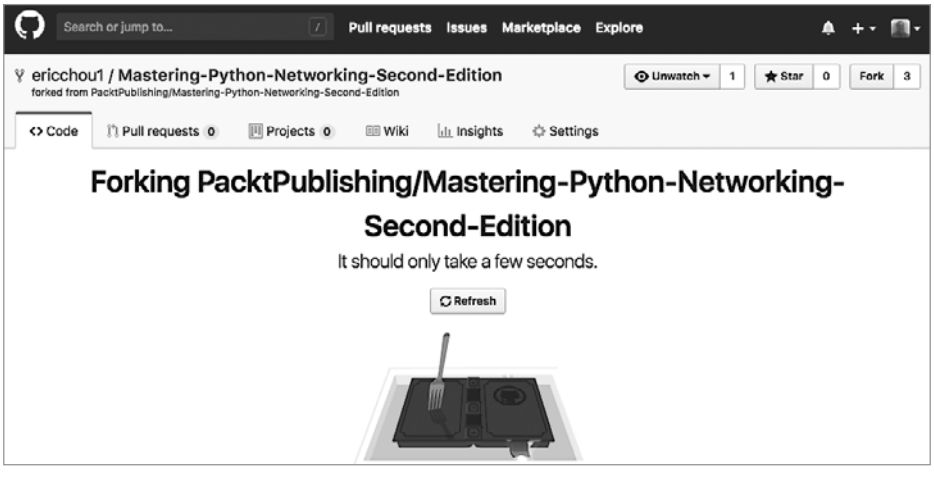


Рис. 13.6. Процесс создания копии

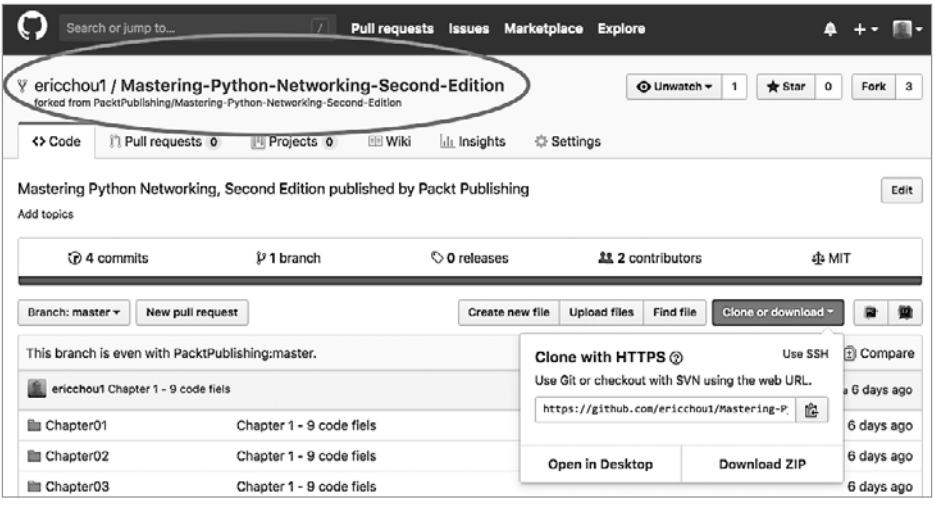


Рис. 13.7. Копия в GitHub

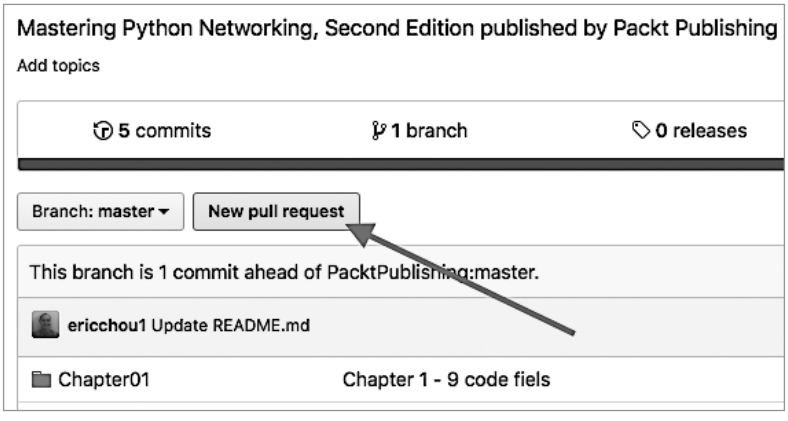


Рис. 13.8. Запрос на внесение изменений

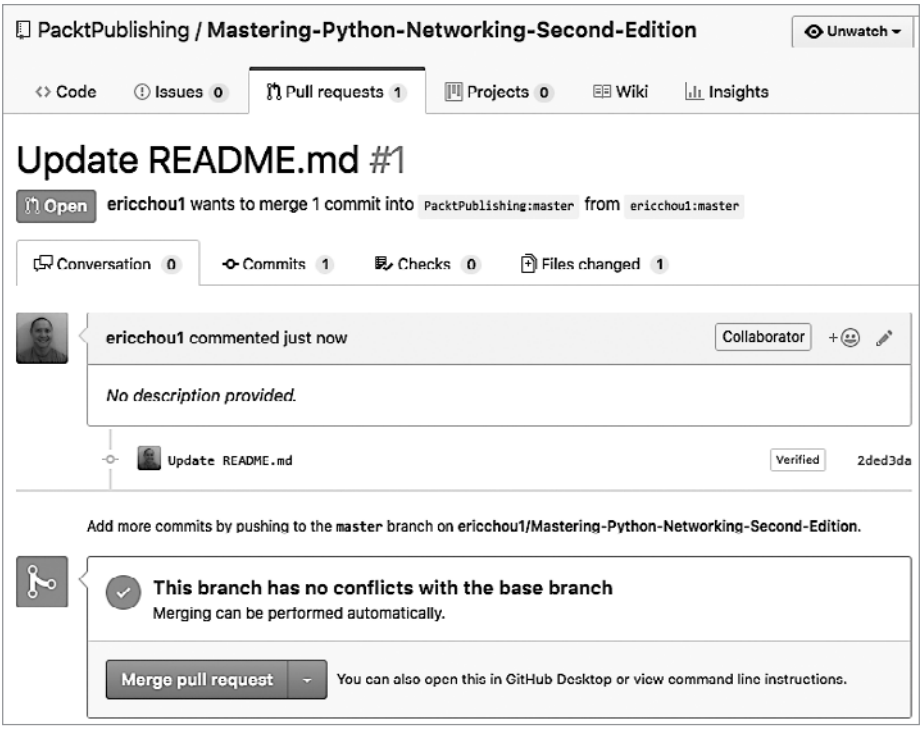


Рис. 13.9. Подробности о запросе на внесение изменений

При создании запроса на внесение изменений необходимо предоставить как можно больше информации, чтобы обосновать наши правки (рис. 13.9).

Владелец репозитория получит уведомление о запросе на внесение изменений; если он его примет, изменения попадут в оригинальный репозиторий (рис. 13.10).

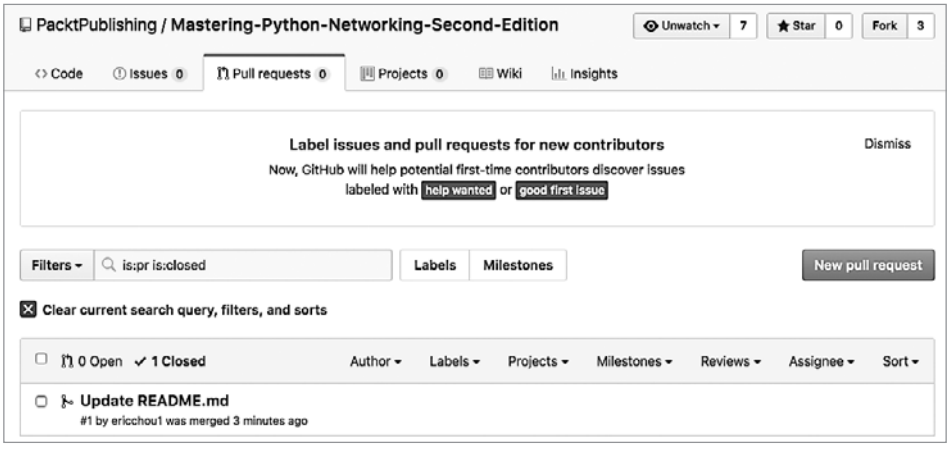


Рис. 13.10. Запись о внесенных изменениях

GitHub предоставляет прекрасную платформу для взаимодействия с другими разработчиками; этот сервис быстро становится самым популярным выбором для крупных открытых проектов. Поскольку Git и GitHub используются очень широко, следующим логичным шагом будет автоматизировать процессы, которые мы видели в этом разделе. Посмотрим, как использовать Git вместе с Python.

Git и Python

Существует несколько пакетов для Python, которые используют для работы с Git и GitHub. В этом разделе мы поговорим о библиотеках GitPython и PyGitHub.

GitPython

Мы возьмем пакет GitPython (https://gitpython.readthedocs.io/en/stable/index.html) для работы с нашим Git-репозиторием. Давайте его установим и создадим объект `Repo` в интерактивной оболочке Python. Там же мы выведем перечень всех фиксаций в репозитории:

```
(venv) $ pip install gitpython
(venv) $ python
>>> from git import Repo
>>> repo = Repo('/home/echou/Mastering_Python_Networking_
third_edition/Chapter13/TestRepo-1')
>>> for commits in list(repo.iter_commits('master')):
... print(commits)
...
1b24b4e95eb0c01cc9a7124dc6ac1ea37d44d51a
169a2034fb9844889f5130f0e42bf9c9b7c08b05
a537bdcc1648458ce88120ae607b4ddea7fa9637
ff7dc1a40e5603fed552a3403be97addefddc4e9
5d7c1c8543c8342b689c66f1ac1fa888090ffa34
```

Мы также можем просмотреть индексы в объекте `repo`:

```
>>> for (path, stage), entry in repo.index.entries.items():
... print(path, stage, entry)
...
myFile.txt 0 100644 69e7d4728965c885180315c0d4c206637b3f6bad 0 myFile.txt
mySecondFile.txt 0 100644 75d6370ae31008f683cf18ed086098d05bf0e4dc 0
mySecondFile.txt
```

GitPython предоставляет хорошую интеграцию со всеми функциями Git, хотя для новичков эта библиотека может оказаться не самой простой. Для полноценной работы с ней необходимо понимать терминологию и структуру Git. Но о ней стоит помнить: она может понадобиться в других проектах.

PyGitHub

Посмотрим, как использовать библиотеку PyGitHub (https://pygithub.readthedocs.io/en/latest/) для работы с репозиториями в GitHub. Этот пакет является оберткой вокруг GitHub APIv3, https://developer.github.com/v3/:

```
(venv) $ pip install PyGithub
```

Воспользуемся интерактивной оболочкой Python, чтобы вывести текущий репозиторий пользователя:

```
(venv) $ python
>>> from github import Github
>>> g = Github("<username>", "<password>")
>>> for repo in g.get_user().get_repos():
... print(repo.name)
...
Mastering-Python-Networking-Second-Edition
Mastering-Python-Networking-Third-Edition
```

Мы можем также гибко управлять доступом к репозиторию с помощью специального токена. GitHub позволяет назначить токену набор прав (рис. 13.11).

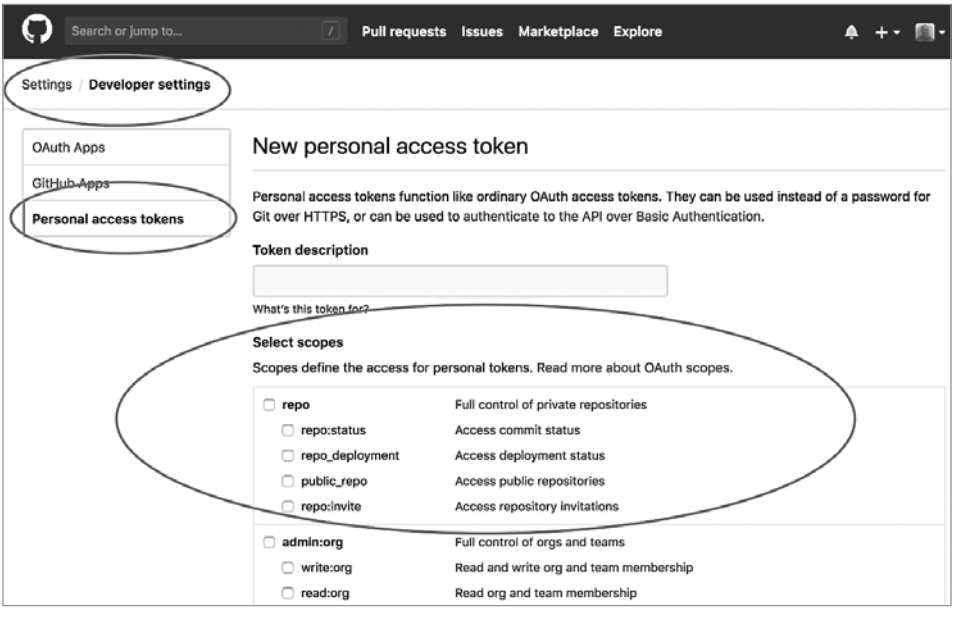


Рис. 13.11. Генерация токена в GitHub

Если использовать токен доступа в качестве механизма аутентификации, вывод будет немного другим:

```
>>> from github import Github
>>> g = Github("<token>")
>>> for repo in g.get_user().get_repos():
... print(repo)
...
Repository(full_name="oreillymedia/distributed_denial_of_service_ddos")
Repository(full_name="PacktPublishing/-Hands-on-Network- Programming-with-
Python")
Repository(full_name="PacktPublishing/Mastering-Python-Networking")
Repository(full_name="PacktPublishing/Mastering-Python-Networking-Second-
Edition")
...
```

Итак, мы познакомились с Git, GitHub и некоторыми пакетами для Python. Теперь мы можем применить их для автоматизации. В следующем разделе рассмотрим примеры.

Автоматизация резервного копирования конфигурационных файлов

В этом примере мы используем PyGithub для резервного копирования каталога, в котором находится конфигурация нашего маршрутизатора. Мы уже знаем, как из наших устройств можно извлекать информацию с помощью Python и Ansible; теперь эту информацию можно выгрузить в GitHub.

У нас есть подкаталог `configs`, в котором находятся текстовые конфигурационные файлы нашего маршрутизатора:

```
$ ls configs/
iosv-1 iosv-2

$ cat configs/iosv-1
Building configuration...

Current configuration : 4573 bytes
!
! Last configuration change at 02:50:05 UTC Sat Jun 2 2018 by cisco
!
version 15.6
service timestamps debug datetime msec
...
```

Используем сценарий `Chapter13_1.py`, чтобы извлечь последний индекс из нашего репозитория в GitHub, сформировать данные, которые необходимо зафиксировать, и автоматически сохранить конфигурацию:

```
#!/usr/bin/env python3
# reference: https://stackoverflow.com/questions/38594717/how-do-ipush-
new-files-to-github

from github import Github, InputGitTreeElement
import os

github_token = '<token>'
configs_dir = 'configs'
github_repo = 'TestRepo'

# Получаем список файлов в каталоге configs
file_list = []
for dirpath, dirname, filenames in os.walk(configs_dir):
    for f in filenames:
        file_list.append(configs_dir + "/" + f)

g = Github(github_token)
repo = g.get_user().get_repo(github_repo)
```

```
commit_message = 'add configs'
master_ref = repo.get_git_ref('heads/master')
master_sha = master_ref.object.sha
base_tree = repo.get_git_tree(master_sha)

element_list = list()

for entry in file_list:
    with open(entry, 'r') as input_file:
        data = input_file.read()
    element = InputGitTreeElement(entry, '100644', 'blob', data)
    element_list.append(element)

# Создаем дерево репозитория и фиксируем изменения
tree = repo.create_git_tree(element_list, base_tree)
parent = repo.get_git_commit(master_sha)
commit = repo.create_git_commit(commit_message, tree, [parent])
master_ref.edit(commit.sha)
```

В репозитории в GitHub появился каталог `configs` (рис. 13.12).

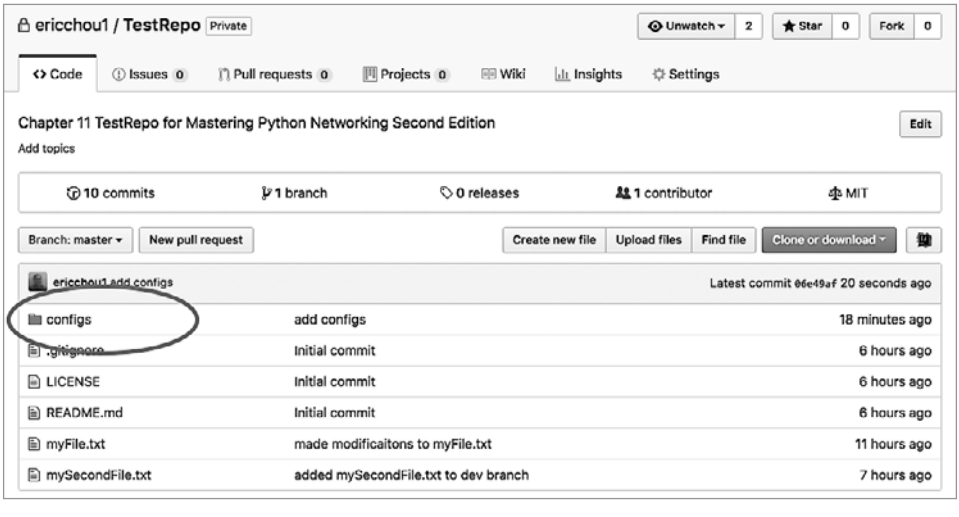


Рис. 13.12. Каталог configs

В истории изменений видно фиксацию, которую выполнил наш сценарий (рис. 13.13).

В примере с GitHub в предыдущем разделе вы видели, как можно организовать совместную работу путем создания копии (форка) репозитория и выполнения запросов на внесение изменений. Поговорим о совместной работе с использованием Git подробнее.

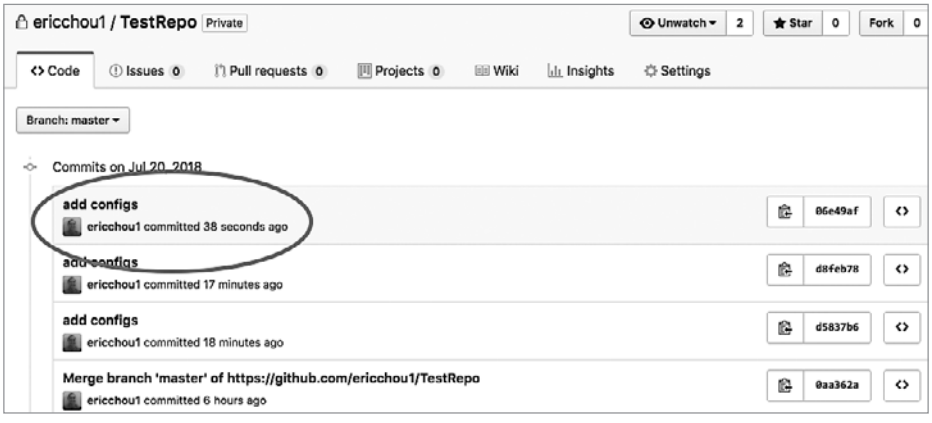


Рис. 13.13. История изменений

Совместная работа с использованием Git

Git отлично подходит для совместной работы, а GitHub позволяет разработчикам эффективно работать над общими проектами. Любой, у кого есть доступ к интернету, может поделиться своими мыслями и кодом. Мы уже знаем, как использовать Git и организовывать простое взаимодействие в GitHub, но как нам присоединиться к существующему проекту?

Нам, несомненно, хочется отплатить взаимностью открытым проектам, которые были для нас так полезны. Но с чего начать?

В этом разделе мы рассмотрим некоторые аспекты совместной разработки с использованием Git и GitHub.

- **Начинайте с малого.** Первое и самое важное — понять свою роль в команде. Вы можете быть потрясающим сетевым инженером, но посредственным программистом на Python. Существует множество дел, которые не требуют особых навыков программирования. Не бойтесь начинать с малого; документация и тестирование — вот те направления, которые позволят вам внести свой первый вклад в проект.
- **Изучите экосистему.** В любом проекте, крупном или маленьком, есть набор общепринятых норм и устоявшаяся культура. Язык Python нас привлекает своим понятным синтаксисом и дружественностью к новичкам; на его сайте есть даже руководство для разработчиков, основанное на этой идеологии (https://devguide.python.org/). У проекта Ansible тоже есть руководство для потенциальных участников сообщества (https://docs.ansible.com/ansible/latest/community/index.html). В нем описываются кодекс поведения, процесс

подачи запросов на внесение изменений, процедура создания отчетов об ошибках и то, как происходит выпуск новых версий. Ознакомьтесь с этими руководствами и изучите экосистему интересующего вас проекта.

- **Создавайте ветви.** Когда-то я по ошибке создал форк проекта и подал запрос на внесение изменений в главную ветвь. Главная ветвь предназначена для внесения изменений основными участниками проекта. Мы должны создавать отдельные ветви для наших правок, чтобы их позже можно было объединить с главной.
- **Синхронизируйте свою копию репозитория.** Нет такого правила, которое заставляло бы нас синхронизировать клонированный репозиторий с оригиналом. Но, чтобы у нас всегда была самая актуальная копия главного репозитория, мы должны регулярно выполнять `git pull` (для получения кода и слияния с локальной копией) или `git fetch` (для получения локальной копии со всеми изменениями).
- **Будьте дружелюбны.** В виртуальном мире, как и в реальном, нет места враждебности. При обсуждении проблемы будьте вежливы и дружелюбны, даже если у вас возникли разногласия.

Git и GitHub позволяют любому неравнодушному человеку внести свой вклад, облегчая совместную работу над проектами. Мы можем участвовать в разработке любых открытых и закрытых проектов, которые нас интересуют.

Резюме

В этой главе мы рассмотрели систему управления версиями под названием Git и ее близкого родственника GitHub. Проект Git был создан в 2005 году Линусом Торвальдсом для облегчения разработки ядра Linux; позже на него перешли многие другие проекты с открытым исходным кодом. Git отличается высокой скоростью, распределенностью и масштабируемостью. GitHub предоставляет централизованное место для размещения Git-репозиториев и позволяет заниматься совместной работой всем, у кого есть интернет-соединение.

Вы увидели, как Git можно использовать в командной строке, познакомились с различными операциями и тем, как они применяются в GitHub. Вы также познакомились с двумя популярными Python-библиотеками для работы с Git: GitPython и PyGitHub. В конце главы рассмотрели пример резервного копирования конфигурации и некоторые замечания о совместной работе.

В главе 14 речь пойдет о Jenkins — еще одном популярном открытом средстве для непрерывной интеграции и развертывания.

14

Непрерывная интеграция с помощью Jenkins

Сети пронизывают все аспекты технологического стека; во всех окружениях, в которых мне приходилось работать, сеть всегда имела первостепенное значение. На этой технологии основана работа других сервисов. В представлении других инженеров, бизнес-руководителей, операторов и рядовых сотрудников надежная работа сети — что-то само собой разумеющееся. Сеть всегда должна быть доступной и исправной; чем меньше о ней слышно, тем лучше.

Конечно, нам, сетевым инженерам, известно, что сеть ничуть не проще любого другого технологического стека. Из-за высокой сложности конструкция сети может получиться хрупкой. Иногда я смотрю на сеть и удивляюсь, как она вообще работает. Что еще удивительнее, она может работать месяцами и годами, не создавая проблем для бизнеса.

Одна из причин нашего интереса к автоматизации сетей: необходимость надежного и согласованного воспроизведения вносимых нами изменений. Используя сценарии на Python или фреймворк Ansible, мы делаем процесс применения изменений предсказуемым и надежным. В предыдущей главе вы уже видели, что в Git и GitHub можно надежно хранить элементы этого процесса: шаблоны, сценарии, списки зависимостей и прочие файлы. Мы можем управлять версиями кода, составляющего нашу инфраструктуру, организовывать совместную работу и отслеживать изменения. Но как собрать все это воедино? В этой главе мы обсудим популярный открытый инструмент под названием Jenkins, который может оптимизировать процесс управления сетью.

Эта глава охватывает следующие темы:

- проблемы традиционного процесса управления изменениями;
- введение в непрерывную интеграцию и Jenkins;
- установку и примеры использования Jenkins;
- Jenkins и Python;
- непрерывную интеграцию в сетевых технологиях.

Для начала разберем традиционный процесс управления изменениями. Любой бывалый сетевой инженер подтвердит, что этот процесс включает много ручной работы и субъективных решений, которые, как вы сами увидите, несогласованны и плохо поддаются систематизации.

Традиционный процесс управления изменениями

Инженеры с опытом работы в масштабном сетевом окружении знают, что некорректное изменение сетевой конфигурации может иметь серьезные последствия. Мы можем внести сотни изменений без каких-либо проблем, но достаточно одной оплошности — и наша сеть окажет негативное влияние на работу всей организации.

Полно ужасных историй о том, как перебои в работе сети приносили проблемы предприятию. Один из самых заметных и масштабных сбоев в AWS EC2 случился в 2011 году, он был вызван изменением сетевой конфигурации в ходе обычных операций масштабирования, проводившихся в регионе US-East. Это произошло в 00:47 по Тихоокеанскому стандартному времени и привело к тому, что на протяжении более чем 12 часов многие сервисы были недоступны; в результате компания Amazon потеряла миллионы долларов. Что еще важнее, это плохо сказалось на репутации относительно нового сервиса. ИТ-специалисты приводили эту поломку в качестве довода ПРОТИВ перехода в тогда еще незрелое облако AWS. На восстановление его репутации ушло много лет. Подробнее об этом читайте в отчете о происшествии по адресу https://aws.amazon.com/message/65648/.

Ввиду сложности и потенциальных последствий во многих сетевых окружениях предусмотрен *консультативный совет по изменениям* (Change-Advisory Board, CAB). Работа CAB обычно строится так.

1. Сетевой инженер проектирует изменение и подробно описывает шаги для его реализации. Описание может включать обоснование, список устройств, которые будут затронуты, команды, которые будут применены или удалены, процедуру проверки вывода и результат, ожидаемый на каждом этапе.
2. Обычно сетевой инженер должен сначала попросить своих коллег провести технический анализ. В зависимости от природы изменения он может проводиться на разных уровнях. Для анализа простого изменения хватит одного коллеги; для чего-то более сложного может потребоваться одобрение от старшего инженера.
3. Совещания CAB обычно проходят в запланированное время. Возможны и срочные внеплановые встречи.
4. Инженер подает свое изменение на рассмотрение совета. Члены совета задают вопросы, оценивают возможные последствия и одобряют или отклоняют эту заявку.
5. В запланированный промежуток времени изменение вносится либо его автором, либо другим инженером.

Этот процесс выглядит разумно и основан на мнении многих людей, но на практике влечет несколько проблем.

- **Время на оформление.** Обычно у инженера, проектирующего изменение, уходит много времени на написание документации (иногда даже больше, чем на воплощение этого изменения в жизнь). Это, как правило, вызвано тем, что любые изменения сетевой конфигурации могут иметь серьезные последствия, и этот процесс нужно задокументировать как для технических, так и для нетехнических членов CAB.
- **Опыт и знания.** Высококлассные инженеры — это ограниченный ресурс. Обычно самый большой спрос — на опытных специалистов. Они должны заниматься самыми сложными сетевыми проблемами, а не анализировать простые изменения сети.
- **Собрания занимают много времени.** Организация собрания всех членов совета требует много усилий. Что, если человек, одобрение которого нам нужно, находится в отпуске или на больничном? Что, если изменение должно быть выполнено до запланированного собрания CAB?

Это лишь некоторые проблемы работы совета по внесению изменений. Лично я терпеть не могу этот процесс. Я не отрицаю необходимости в расстановке приоритетов и анализе со стороны коллег, но, как мне кажется, мы должны

минимизировать сопутствующие издержки. В оставшейся части главы мы рассмотрим другую процедуру управления изменениями, которая может заменить вышеописанный процесс. Она позаимствована из мира разработки программного обеспечения.

Введение в непрерывную интеграцию

Непрерывная интеграция (Continuous Integration, CI) применяется в разработке программного обеспечения для быстрой публикации мелких изменений с выполнением встроенных тестов и проверок. Самое главное здесь — определить, совместимо ли изменение с CI; оно должно быть простым и небольшим, чтобы его легко было откатить. Тесты и проверки выполняются автоматически и дают минимально необходимый уровень уверенности в том, что применение изменений не нарушит работу всей системы.

До появления CI изменения в ПО зачастую вносились большими пакетами и требовали продолжительной проверки (звучит знакомо?). На развертывание изменений в промышленной среде, получение обратной связи и исправление всех ошибок могли уходить месяцы. Процесс CI, по большому счету, направлен на сокращение времени между появлением идеи и ее реализацией.

Рабочий процесс обычно состоит из следующих шагов.

1. Первый инженер берет текущую копию кодовой базы и вносит в нее изменения.
2. Первый инженер сохраняет изменения в репозитории.
3. Репозиторий может уведомить все заинтересованные стороны и дать возможность группе инженеров проанализировать изменения. Изменения могут быть либо приняты, либо отклонены.
4. Система CI может непрерывно следить за изменениями в репозитории, или же сам репозиторий может слать ей уведомления об изменениях. В любом случае система CI получает самую последнюю версию кода.
5. Система CI выполняет автоматические тесты, пытаясь выявить какие-либо неполадки.
6. Если никаких проблем не найдено, система CI может выполнить слияние изменений с основным кодом и при необходимости развернуть их в промышленной среде.

В реальности процесс зависит от конкретной организации. Например, автоматические тесты могут выполняться сразу после обнаружения изменений, до анализа кода. Некоторые организации требуют, чтобы в промежутке между этапами процесс проверялся живым человеком.

В следующем разделе даны инструкции по установке Jenkins в системе Ubuntu 18.04.

Установка Jenkins

Для опробования примеров из этой главы вы можете установить Jenkins как на управляющий хост, так и на отдельную виртуальную машину. Как я уже отмечал в предыдущих главах, мне больше импонирует второй вариант. Сетевая конфигурация виртуальной машины будет похожа на ту, которую мы до сих пор использовали на управляющем хосте: один интерфейс для интернет-соединения и еще один — для VMNet2-соединения с управляющей сетью VIRL.

Образ Jenkins и инструкции по установке для разных операционных систем можно найти по адресам https://jenkins.io/download/ и https://jenkins.io/doc/book/installing/.

Ниже приводятся команды, которые я использовал для установки Jenkins на хост с Ubuntu. Для Jenkins не нужно высокопроизводительное оборудование; в своей лаборатории я использовал один виртуальный процессор и 2 Гбайт оперативной памяти. Также необходимо установить Java версии 8 или 11. Мы выберем для нашего сервера OpenJDK-11:

```
$ sudo apt install openjdk-11-jre-headless
$ java --version
openjdk 11.0.4 2019-07-16
OpenJDK Runtime Environment (build 11.0.4+11-post-Ubuntu-1ubuntu218.04.3)
OpenJDK 64-Bit Server VM (build 11.0.4+11-post-Ubuntu-1ubuntu218.04.3,
mixed mode, sharing)

$ wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo aptkey
add -
$ sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /
etc/apt/sources.list.d/jenkins.list'
$ sudo apt-get update
$ sudo apt-get install Jenkins
$ sudo /etc/init.d/jenkins start
Correct java version found
[ ok ] Starting jenkins (via systemctl): jenkins.service.
```

Система Jenkins чувствительна к версии Java. На момент написания этой главы она поддерживает только Java 8 и 11. Java версий 9, 10 и 12 не поддерживались (https://jenkins.io/doc/administration/requirements/java/).

После установки Jenkins можно открыть в браузере соответствующий IP-адрес с портом 8080 (рис. 14.1).

Рис. 14.1. Страница для разблокирования Jenkins

Последуем инструкциям, указанным на странице: скопируем пароль администратора из `/var/lib/jenkins/secrets/initialAdminPassword` и вставим его в поле ввода:

```
$ sudo cat /var/lib/jenkins/secrets/initialAdminPassword

<одноразовый пароль администратора>
```

На следующей странице выберем способ конфигурации Jenkins. Мы остановимся на варианте Install suggested plugins (Установить рекомендуемые плагины) (рис. 14.2).

Вы будете перенаправлены на страницу создания учетной записи администратора, после чего система Jenkins будет готова к работе. Если вы увидели панель управления Jenkins, установка прошла успешно (рис. 14.3).

Все готово для того, чтобы запланировать в Jenkins первое задание.

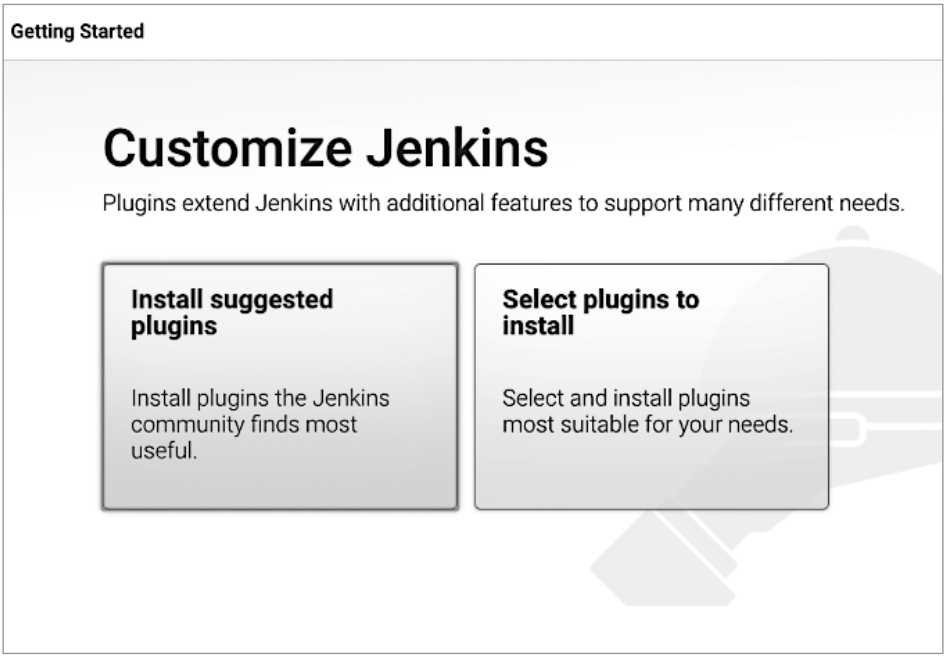


Рис. 14.2. Установка рекомендуемых плагинов

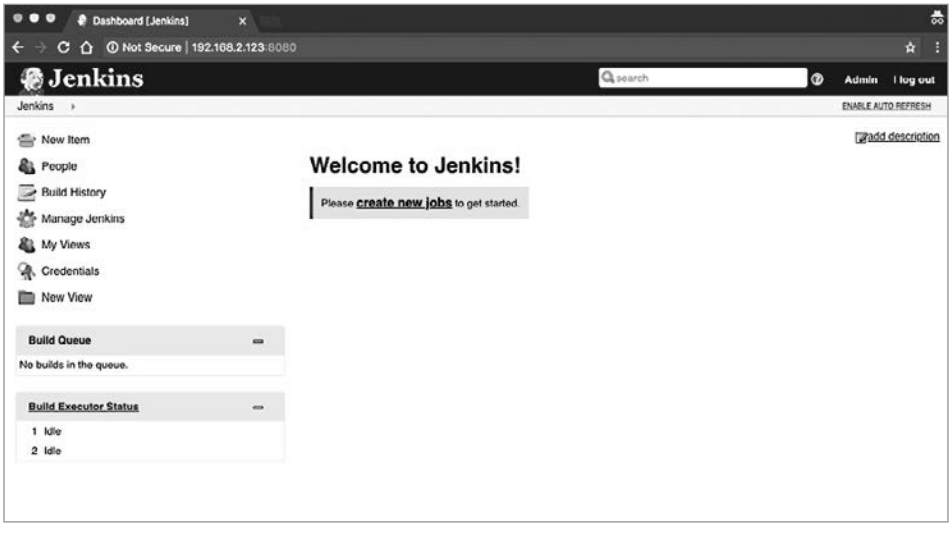


Рис. 14.3. Панель управления Jenkins

Пример с Jenkins

В этом разделе мы рассмотрим несколько примеров использования Jenkins и их связь с технологиями, с которыми вы уже познакомились. В системе Jenkins применяется много инструментов, которые обсуждались в этой книге, включая сценарии на Python, Ansible, Git и GitHub, поэтому ей посвящена одна из последних глав. При необходимости сверяйтесь с предыдущими главами.

В этих примерах для выполнения заданий будет использоваться ведущий узел Jenkins. В промышленных условиях для этого рекомендуется добавить несколько узлов-агентов.

Наша лаборатория будет иметь простую топологию, состоящую из двух узлов с устройствами IOSv (рис. 14.4).

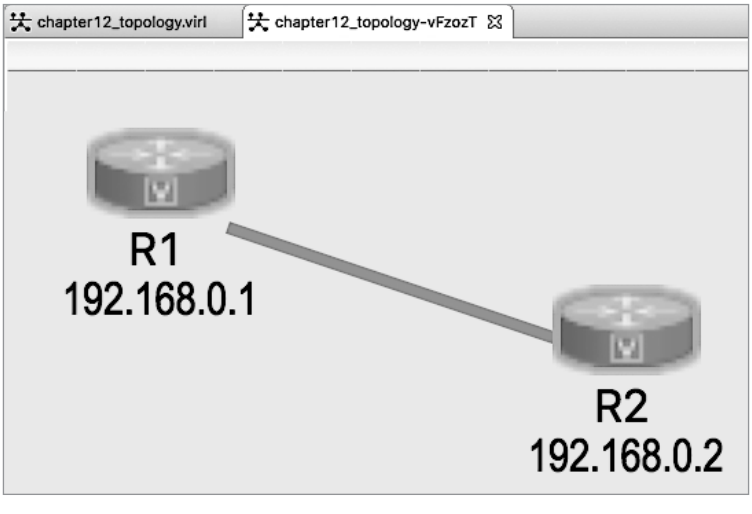


Рис. 14.4. Топология лаборатории

Создадим первое задание.

Первое задание для сценария на Python

Для первого задания возьмем сценарий `chapter2_3.py` из главы 2. Как вы, наверное, помните, этот сценарий использует библиотеку Paramiko для входа на

удаленное устройство по SSH, выполнения команды `show` и захвата ее вывода, содержащего версию устройства. Прежде чем приступать к созданию задания Jenkins, обязательно следует убедиться в том, что сценарий работает верно, для чего запустим его на нашей виртуальной машине:

```
$ ls chapter14_1.py
chapter14_1.py
$ python3 chapter14_1.py
$ ls ios*
iosv-1_output.txt  iosv-2_output.txt
```

Щелкнем на ссылке create new item (создать новый элемент), чтобы создать задание, и выберем пункт Freestyle project (Проект свободного стиля) (рис. 14.5).

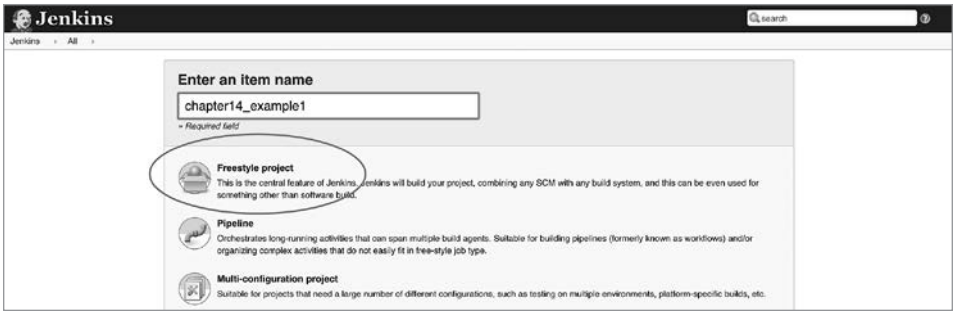


Рис. 14.5. Элемент Jenkins

Введем наше собственное описание и оставим все параметры по умолчанию. Прокрутим страницу вниз и выберем в качестве способа сборки Execute shell (Выполнить консольную команду) (рис. 14.6).

В появившемся поле ввода нужно ввести консольные команды, которые будут использоваться (рис. 14.7).

После сохранения конфигурации задания откроется панель управления проектом. Выберем пункт Build Now (Собрать сейчас), и задание появится в Build History (История сборок) (рис. 14.8).

Чтобы проверить состояние сборки, щелкните на ней и выберите пункт Console Output (Консольный вывод) на панели слева (рис. 14.9).

Как дополнительный шаг задание можно запланировать для выполнения через регулярные интервалы времени (подобно тому как это делает `cron`). Вернемся в меню job (задание) и выберем configure (настроить). Задание можно заплани-

ровать в разделе Build Triggers (Триггеры сборки). Выберите Build periodically (Собирать периодически) и введите расписание выполнения в формате cron. В этом примере сценарий будет выполняться каждый день в 02:00 и 20:00 (рис. 14.10).

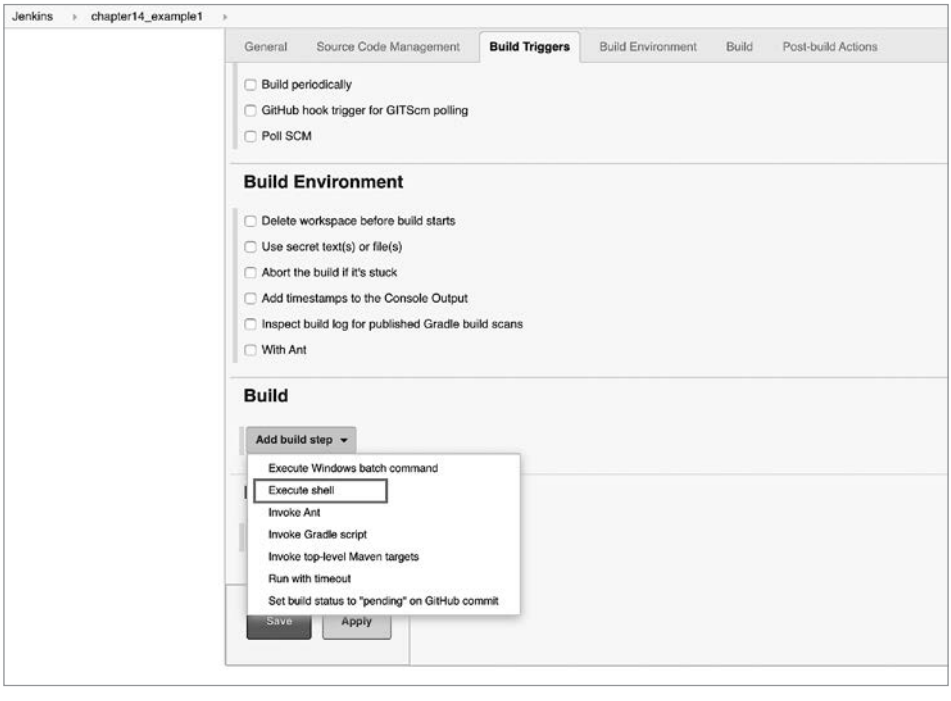


Рис. 14.6. Триггеры сборки в Jenkins

Рис. 14.7. Консольные команды в Jenkins

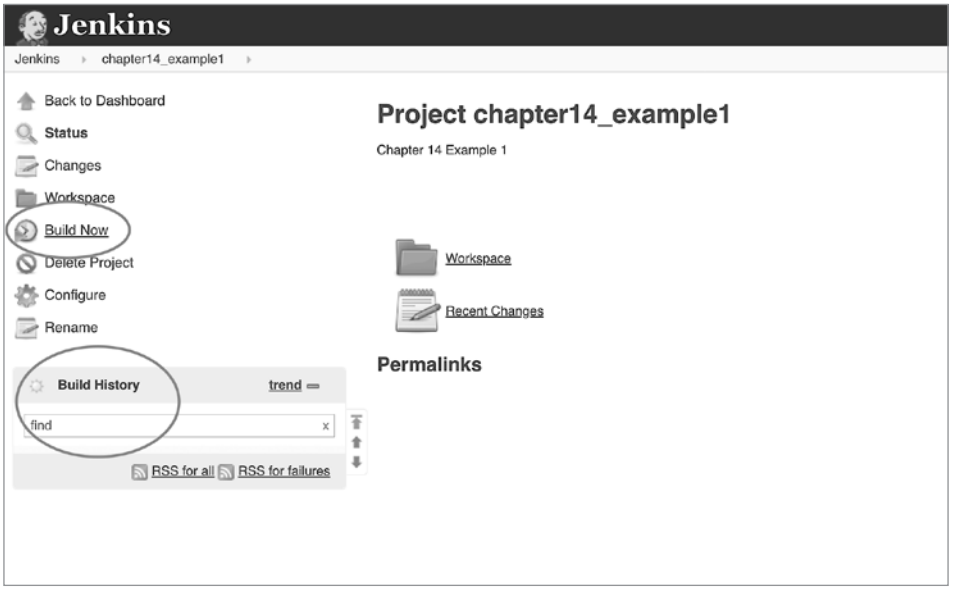


Рис. 14.8. История сборок в Jenkins

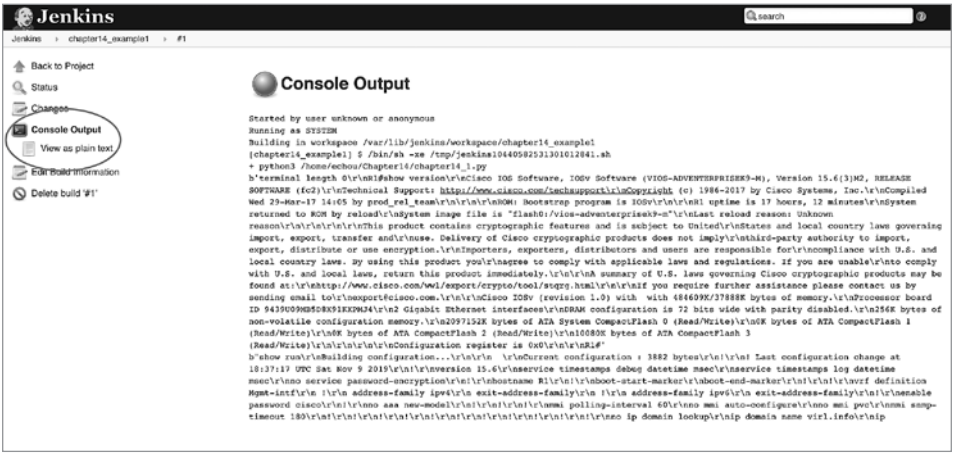


Рис. 14.9. Консольный вывод в Jenkins

В Jenkins также можно настроить SMTP-сервер, чтобы отправлять уведомления о результатах сборки. Пройдите в главное меню Manage Jenkins (Управление Jenkins) и выберите пункт Configure System (Настроить систему) (рис. 14.11).

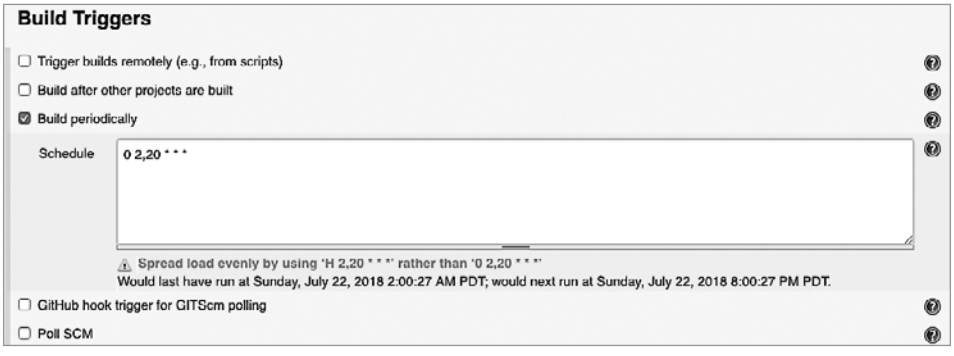


Рис. 14.10. Триггер сборки

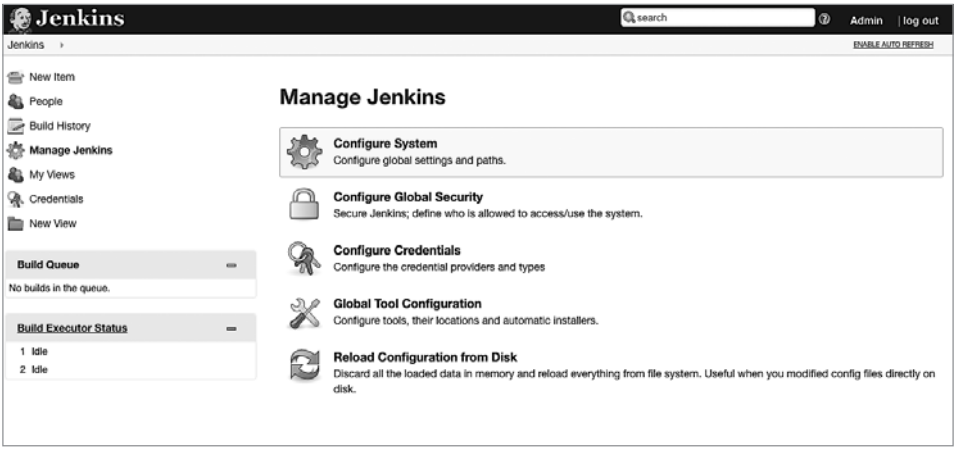


Рис. 14.11. Настройка системы

Настройки SMTP-сервера находятся в нижней части страницы. Щелкните на ссылке Advanced settings (Дополнительные настройки), укажите нужные параметры и установите флажок для отправки пробного электронного письма (рис. 14.12).

Отправку уведомлений на электронную почту можно настроить на странице Post-build Actions (Действия после сборки) (рис. 14.13).

Поздравляю! Вы только что создали свое первое задание в Jenkins. Хотя с практической точки зрения оно не делает ничего такого, чего нельзя было бы сделать вручную на нашем управляющем хосте.

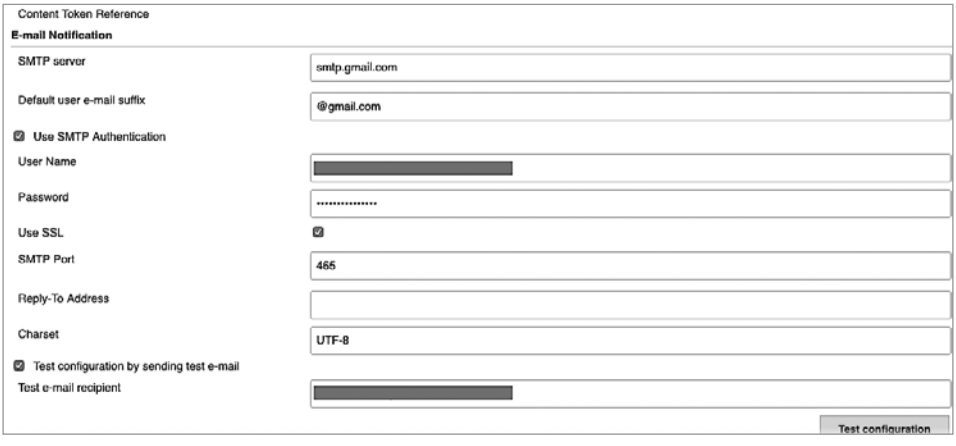


Рис. 14.12. Настройка SMTP

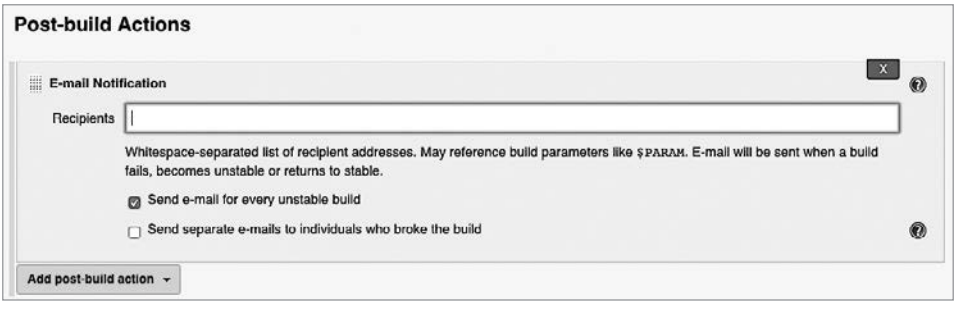


Рис. 14.13. Уведомления на электронную почту

Но у Jenkins есть несколько преимуществ.

- Возможность использовать различные механизмы аутентификации Jenkins на основе баз данных, такие как LDAP, давая возможность выполнять наш сценарий существующим пользователям.
- Возможность ограничить доступ с помощью системы авторизации Jenkins на основе ролей. Например, одним пользователям можно разрешить выполнять задания без права на запись, а другим дать полные администраторские привилегии.
- Простой доступ к сценариям через веб-интерфейс Jenkins.
- Возможность использовать сервисы электронной почты и журналирования Jenkins для централизации заданий и получения уведомлений о результатах.

Jenkins — отличный инструмент. Как и Python, он имеет свою экосистему сторонних плагинов, расширяющих его возможности. Мы рассмотрим ее в следующем разделе.

Плагины Jenkins

Чтобы проиллюстрировать процесс установки плагинов, выберем плагин планирования сборки. Управление плагинами осуществляется в разделе Manage Jenkins ▸ Manage Plugins (Управление Jenkins ▸ Управление плагинами) (рис. 14.14).

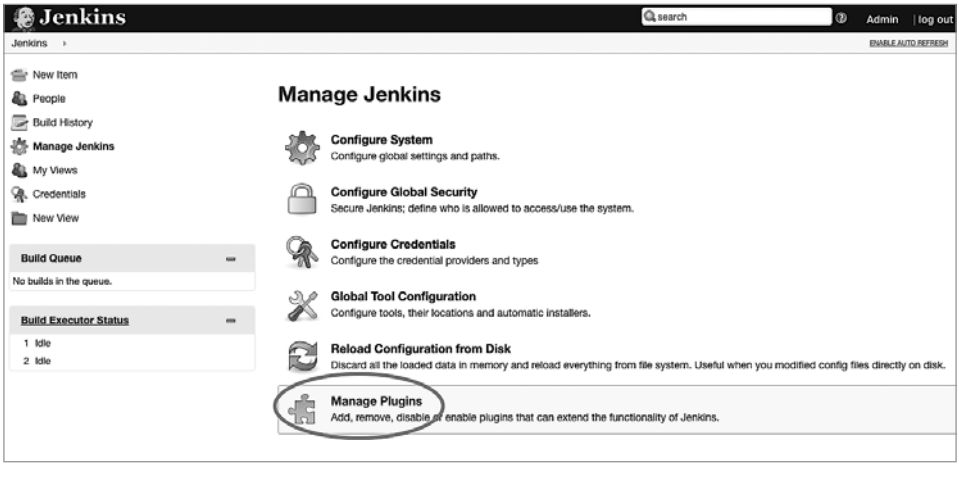


Рис. 14.14. Плагины Jenkins

Перейдем на вкладку Available (Доступные) и найдем плагин Schedule Build с помощью функции поиска (рис. 14.15).

Дальше просто нажмем кнопку Install without restart (Установить без перезагрузки), и на следующей странице можно будет наблюдать за ходом установки (рис. 14.16).

После установки появится новый значок, который поможет сделать планирование заданий проще и удобнее (рис. 14.17).

Одна из сильных сторон популярных проектов с открытым исходным кодом — возможность их расширения. Плагины позволяют адаптировать Jenkins под нужды разных клиентов. В следующем подразделе мы поговорим о том, как

интегрировать в рабочий процесс систему управления версиями и процедуру согласования.

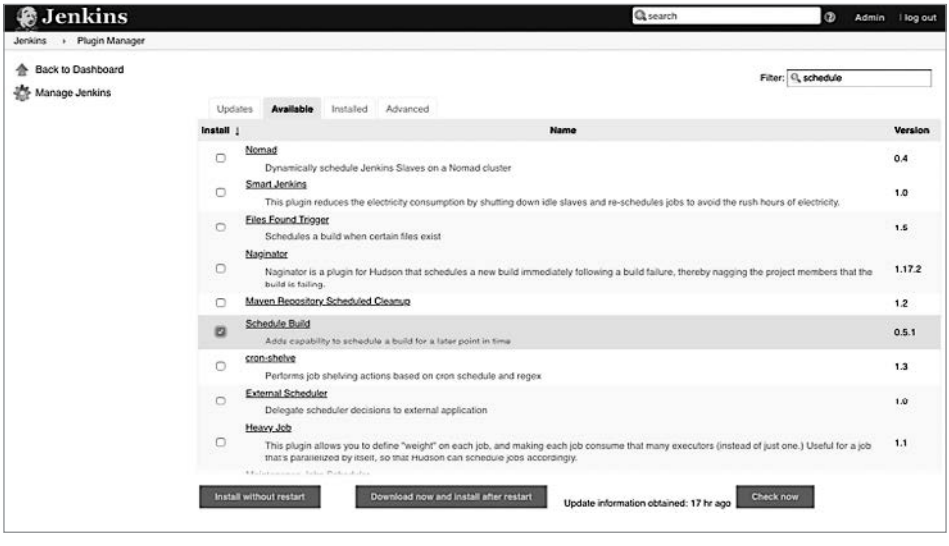


Рис. 14.15. Поиск плагинов в Jenkins

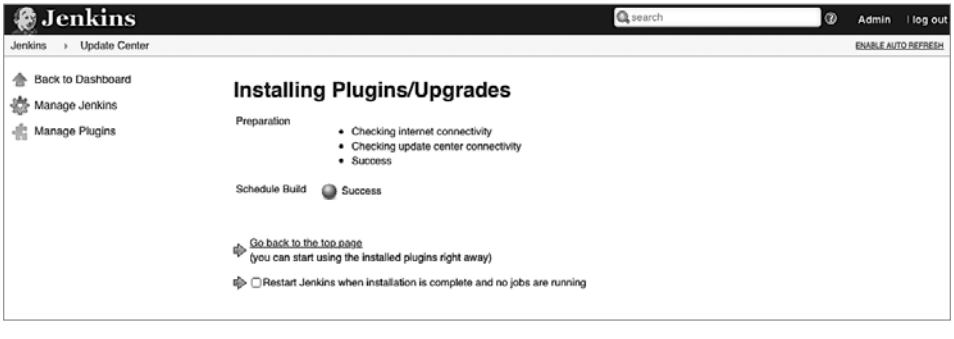


Рис. 14.16. Установка плагина в Jenkins

Рис. 14.17. Результат установки плагина в Jenkins

Пример непрерывной интеграции в контексте сетевых технологий

В этом разделе мы интегрируем наш GitHub-репозиторий с системой Jenkins. Это позволит использовать инструменты GitHub для анализа кода и совместной работы.

Создадим новый репозиторий в GitHub. Назовем его `chapter14_example2`. Мы можем клонировать его локально и добавить нужные файлы. Я добавлю сценарий Ansible, который копирует в файл вывод команды `show version`:

```
---
- name: show version
  hosts: "ios-devices"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ ansible_user }}"
      password: "{{ ansible_password }}"

  tasks:
    - name: show version
      ios_command:
        commands: show version
        provider: "{{ cli }}"

      register: output

    - name: show output
      debug:
        var: output.stdout

    - name: copy output to file
      copy: content="{{ output }}" dest=./output/{{ inventory_hostname }}.txt
```

Мы уже знаем, как выполнять сценарии Ansible. Я опущу вывод `host_vars` и файл `hosts`. Но самое важное здесь — это проверить, работает ли сценарий на локальном компьютере, прежде чем сохранить его в репозитории GitHub:

```
$ ansible-playbook -i hosts chapter14_playbook.yml

PLAY [show version]
************************************************************

TASK [show version]
************************************************************
ok: [iosv-1]
ok: [iosv-2]
```

```
...
TASK [copy output to file]
*****************************************************
changed: [iosv-1]
changed: [iosv-2]

PLAY RECAP
*********************************************************************
iosv-1 : ok=3 changed=1 unreachable=0 failed=0
iosv-2 : ok=3 changed=1 unreachable=0 failed=0
```

Теперь сценарий со всеми сопутствующими файлами можно выгрузить в репозиторий GitHub (рис. 14.18).

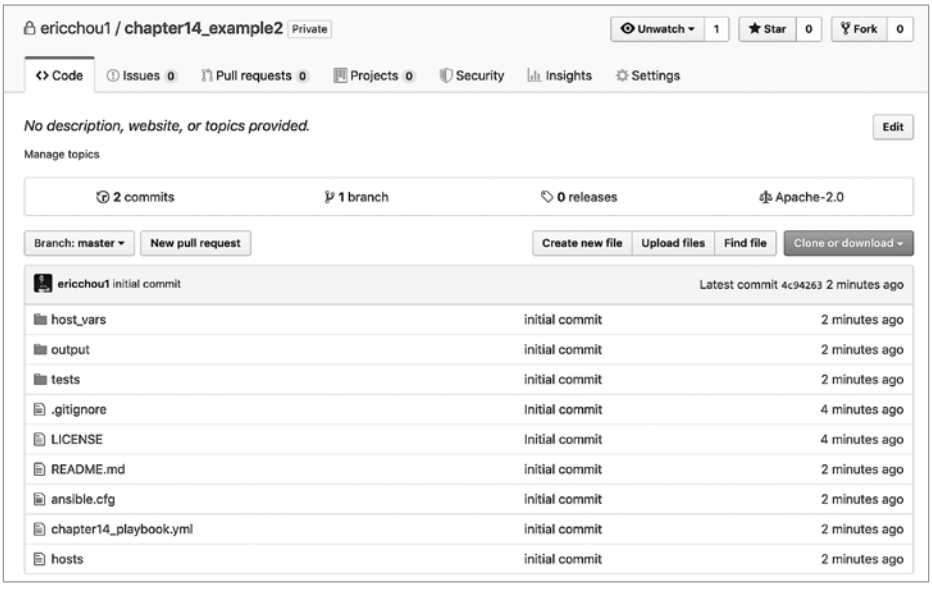


Рис. 14.18. Пример репозитория GitHub для демонстрации интеграции с Jenkins

Зайдем на хост Jenkins, чтобы установить Git и Ansible:

```
$ sudo apt-get install software-properties-common
$ sudo apt-get update
$ sudo apt-get install ansible
$ sudo apt-get install git
```

Отмечу, что некоторые инструменты можно установить на странице Global Tool Configuration (Конфигурация глобальных инструментов) и Git — один из них. Однако нам нужно установить Ansible, поэтому и Git в этом примере мы установили из командной строки (рис. 14.19).

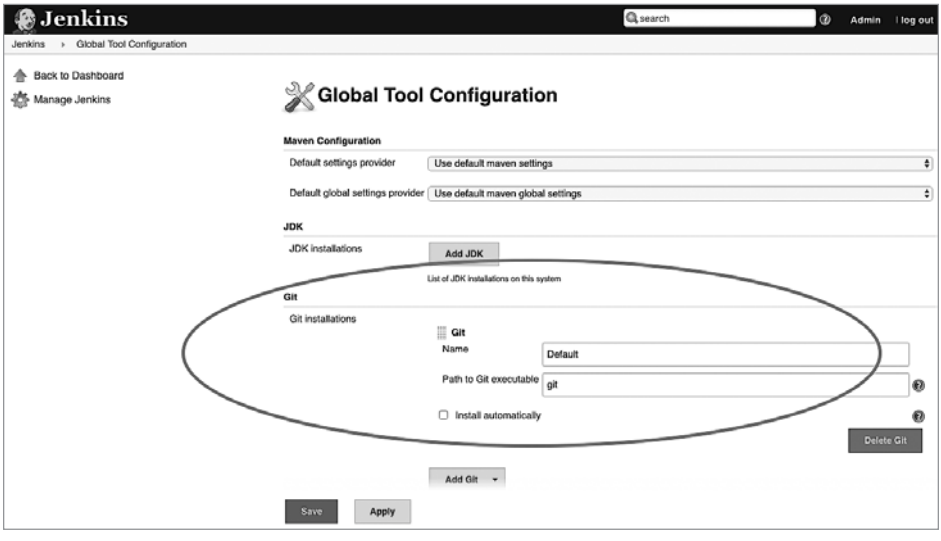


Рис. 14.19. Конфигурация глобальных инструментов

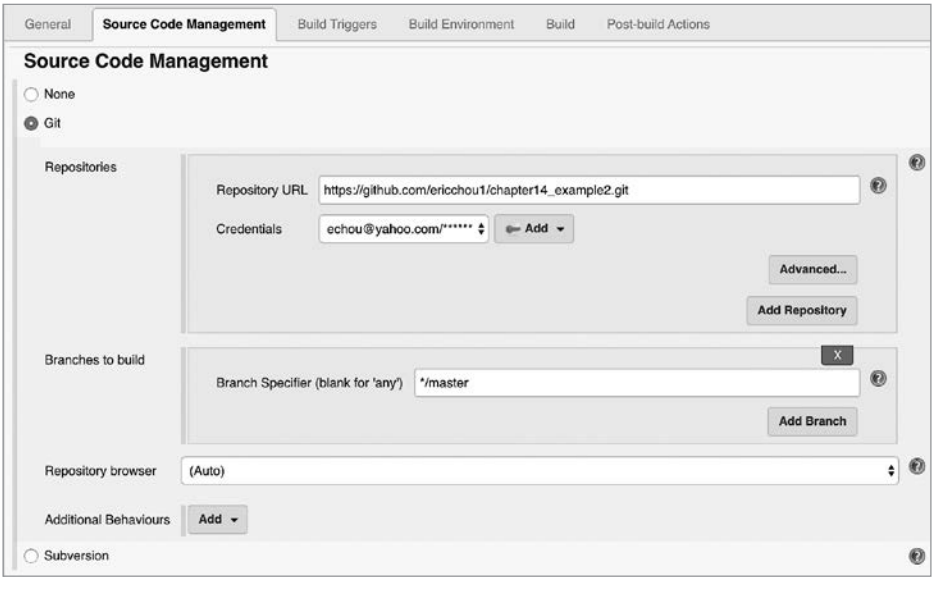


Рис. 14.20. Управление исходным кодом в Jenkins

Создадим новый проект в свободном стиле с названием `chapter14_example2`. В качестве источника на вкладке Source Code Management (Управление исходным кодом) укажем репозиторий GitHub (рис. 14.20).

Прежде чем переходить к следующему этапу, сохраним проект и выполним сборку. В разделе Console Output (Консольный вывод) нашей сборки видно, как клонируется репозиторий; значение индекса должно совпадать со значением в GitHub (рис. 14.21).

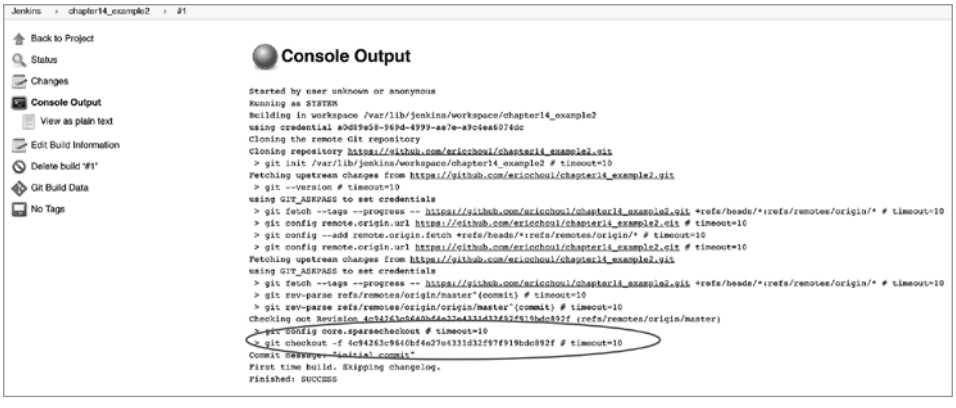


Рис. 14.21. Еще один взгляд на консольный вывод в Jenkins

Теперь можно добавить команду запуска сценария Ansible в раздел Build (Сборка) (рис. 14.22).

Рис. 14.22. Консольная команда в Jenkins

Если снова выполнить сборку, то в консольном выводе мы увидим, как Jenkins загружает код из GitHub и выполняет сценарий Ansible (рис. 14.23).

Одно из преимуществ интеграции GitHub с Jenkins: возможность просматривать всю информацию о Git-репозитории на одной странице (рис. 14.24).

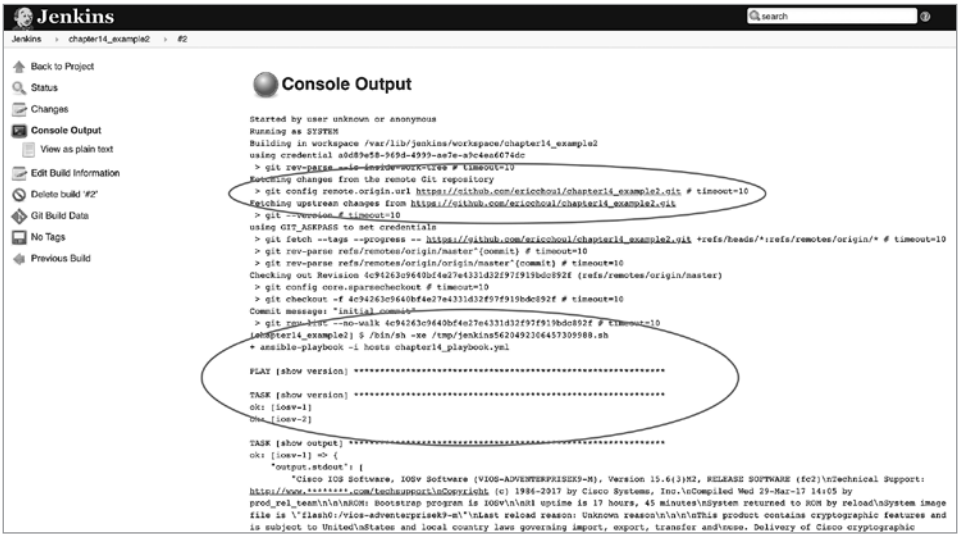


Рис. 14.23. Дальнейшее исследование консольного вывода в Jenkins

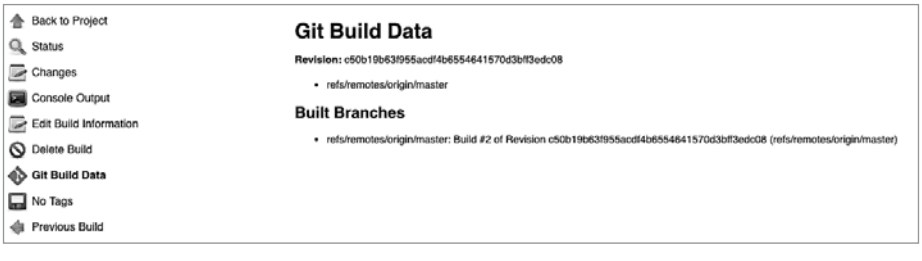


Рис. 14.24. Данные о сборке в Git

Результаты выполнения сборки, такие как вывод сценария Ansible, сохраняются в папке Workspace (рис. 14.25).

Теперь можно выполнить уже знакомый нам шаг и настроить периодическую сборку. Если хост с Jenkins доступен извне, то можно воспользоваться плагином от GitHub, чтобы инициировать сборку отправкой уведомления в Jenkins. Это двухэтапный процесс. Сначала нужно включить веб-обработчики (webhooks) в репозитории GitHub (рис. 14.26).

На втором этапе нужно установить плагины и и позволить использовать GitHub в качестве триггера сборки нашего проекта. Мы уже установили плагин Git, поэтому далее нужно установить плагин GitHub, если он еще не был установлен (рис. 14.27).

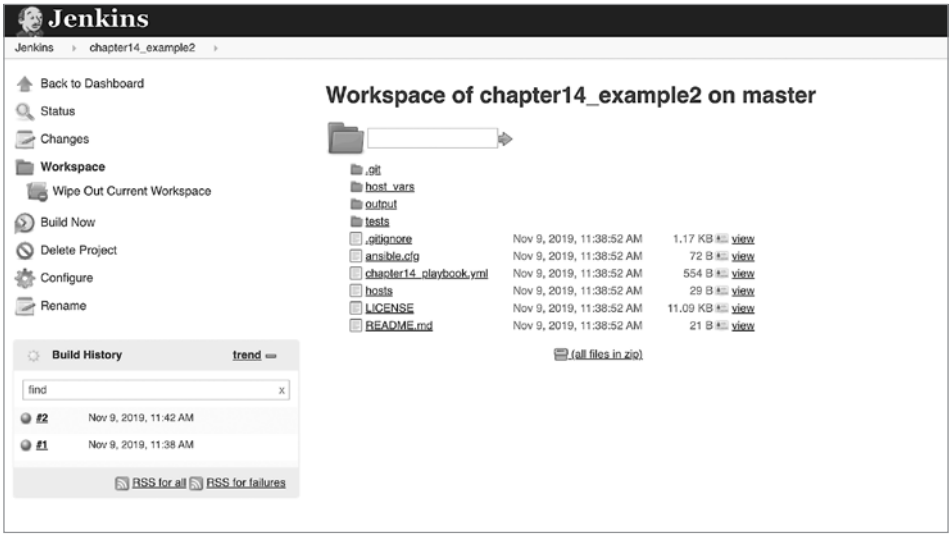


Рис. 14.25. Рабочее пространство Jenkins

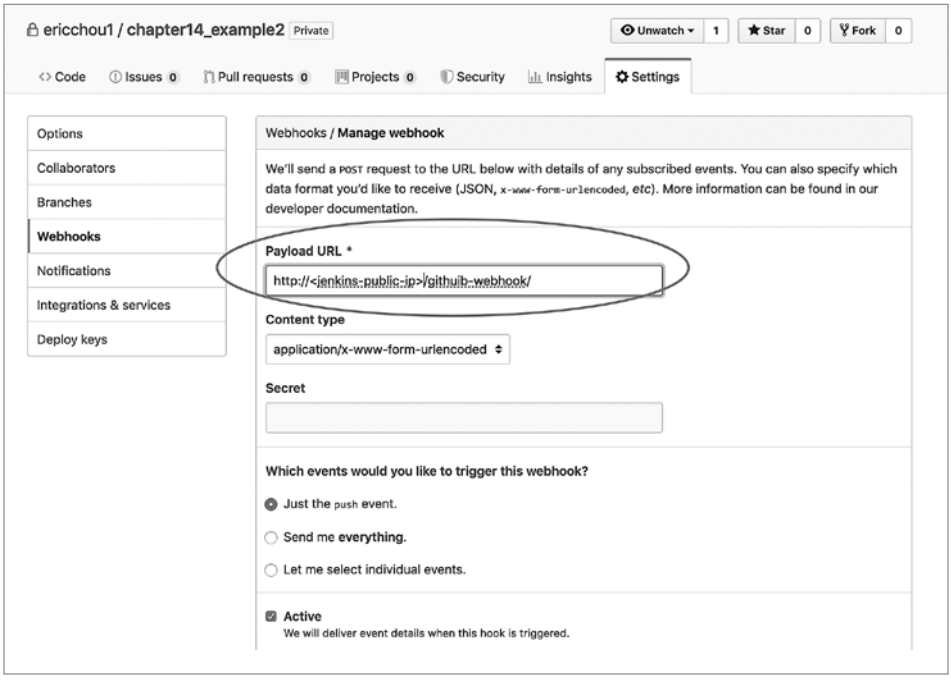


Рис. 14.26. Веб-обработчики GitHub

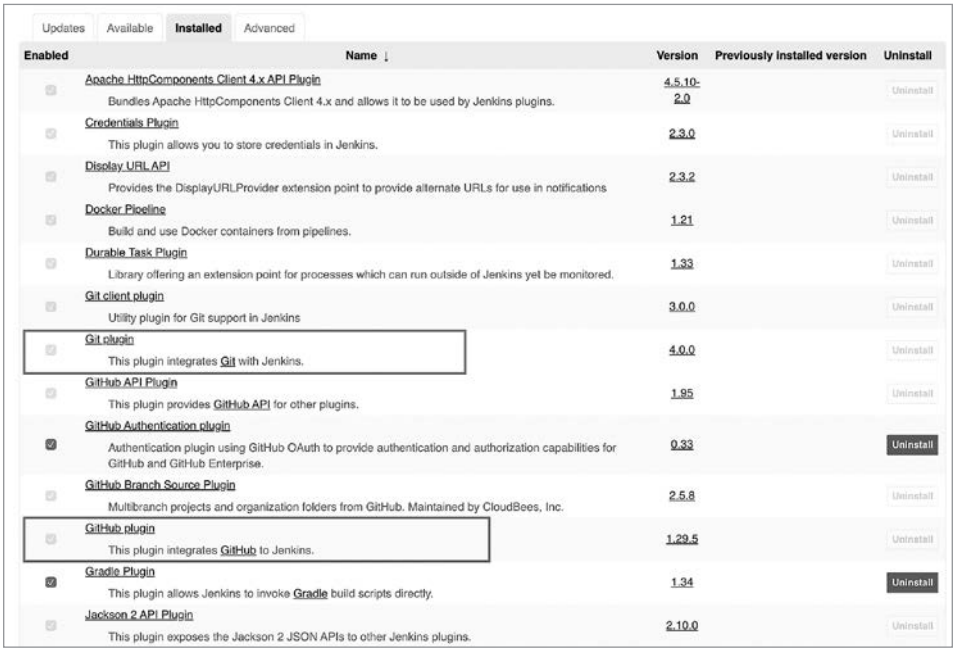


Рис. 14.27. Плагины Git и GitHub для Jenkins

Этот плагин обеспечивает двунаправленную интеграцию с GitHub и позволяет создать обработчик, который будет обращаться к экземпляру Jenkins при каждой выгрузке изменений в GitHub. Используем этот обработчик в качестве триггера сборки нашего проекта (рис. 14.28).

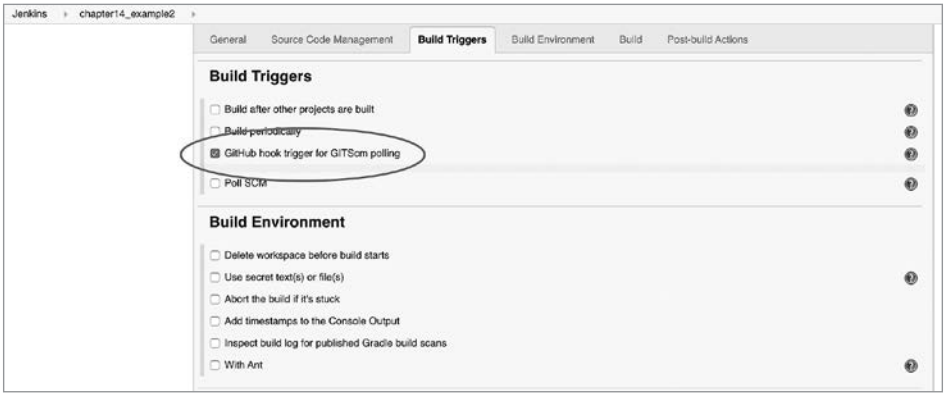


Рис. 14.28. Триггер на основе обработчика GitHub

Использование репозитория GitHub в качестве источника открывает перед нами совершенно новые возможности, позволяющие работать с инфраструктурой как с кодом. Теперь для эффективной совместной работы мы можем копировать репозитории, посылать запросы на внесение изменений, отслеживать проблемы и использовать инструменты управления проектом, которые предоставляет GitHub. Когда код будет готов, Jenkins автоматически его загрузит и выполнит от нашего имени.

Вы, наверное, заметили, что я не упоминаю автоматическое тестирование. Эта тема будет рассмотрена в главе 15.

Jenkins — это полнофункциональная и довольно сложная система. В этих двух примерах мы лишь коснулись ее возможностей. Jenkins поддерживает такие полезные функции, как конвейеры, настройка окружения, конвейеры с несколькими ветвями и т. д., которые позволяют справиться с самыми сложными проектами по автоматизации. Надеюсь, эта глава подтолкнет вас к дальнейшему изучению Jenkins.

До сих пор мы работали с Jenkins из веб-интерфейса. В следующем разделе вы увидите, как можно взаимодействовать с этой системой из сценариев на Python.

Jenkins и Python

Jenkins предоставляет полноценный RESTful API: https://wiki.jenkins.io/display/JENKINS/Remote+access+API. Есть также ряд оберток на языке Python, которые делают взаимодействие с Jenkins еще проще. Рассмотрим пакет `python-jenkins`:

```
(venv) $ pip install python-jenkins
```

Мы можем проверить работу этого пакета в интерактивной командной оболочке:

```
>>> import jenkins
>>> server = jenkins.Jenkins('http://192.168.2.124:8080',
username='<имя пользователя>', password='<пароль>')
>>> user = server.get_whoami()
>>> version = server.get_version()
>>> print('Hello %s from Jenkins %s' % (user['fullName'], version))
Hello Admin from Jenkins 2.121.2
```

Нам доступны различные аспекты управления сервером, такие как плагины:

```
>>> plugin = server.get_plugins_info()
>>> plugin
[{'active': True, 'backupVersion': None, 'bundled': False, 'deleted': False,
```

```
'dependencies': [{'optional': False, 'shortName': 'workflow-scmstep',
'version': '2.9'}, {'optional': False, 'shortName': 'workflow-step-api',
'version': '2.20'}, {'optional': False, 'shortName': 'credentials',
'version': '2.3.0'}, {'optional': False, 'shortName': 'git-client',
'version': '3.0.0'}, {'optional': False, 'shortName': 'mailer', 'version':
'1.23'}, {'optional': False, 'shortName': 'scm-api',
<опущено>
```

Также есть возможность управлять заданиями Jenkins:

```
>>> job = server.get_job_config('chapter14_example1')
>>> import pprint
>>> pprint.pprint(job)
("<?xml version='1.1' encoding='UTF-8'?>\n"
 '<project>\n'
 '  <actions/>\n'
 '  <description>Chapter 14 Example 1</description>\n'
 '  <keepDependencies>false</keepDependencies>\n'
 '  <properties/>\n'
 '  <scm class="hudson.scm.NullSCM"/>\n'
 '  <canRoam>true</canRoam>\n'
 '  <disabled>false</disabled>\n'
<опущено>
```

Пакет `python-jenkins` позволяет взаимодействовать с Jenkins из программного кода. В следующем разделе мы обсудим внедрение непрерывной интеграции и Jenkins в рабочий процесс сетевого инженера.

Непрерывная интеграция в контексте администрирования сети

Непрерывная интеграция уже давно применяется в разработке программного обеспечения, но в мире сетевых технологий это относительно новое явление. Стоит признать, что мы здесь немного отстаем. Мы все еще пытаемся понять, как избавиться от интерфейса командной строки при администрировании наших устройств, поэтому нам, очевидно, будет непросто обращаться с нашей сетью как с кодом.

Существует много примеров применения Jenkins для автоматизации сетей. Один из них был представлен на конференции AnsibleFest 2017 Network Track Тимом Фейвезером и Шиа Стюартом: https://www.ansible.com/ansible-for-networks-beyond-static-config-templates. Еще одним поделился Карлос Висенте из Dyn на конференции NANOG 63: https://archive.nanog.org/sites/default/files/monday_general_autobuild_vicente_63.28.pdf.

Непрерывная интеграция, наверное, не самая простая тема для сетевого инженера, который только начинает изучать программирование и программные инструменты, но, по моему мнению, сегодня стоит ее изучить и использовать в промышленных условиях. Даже минимальный опыт ее применения даст толчок к поиску новых методов автоматизации, которые, вне всяких сомнений, поспособствуют развитию нашей индустрии.

Резюме

В этой главе вы познакомились с традиционным процессом управления изменениями и узнали, почему он не очень хорошо подходит для современных стремительно меняющихся организаций. Сеть должна развиваться вместе с организацией и быть гибкой, чтобы быстрее и надежнее адаптироваться к изменениям.

Вы изучили концепцию непрерывной интеграции и познакомились с Jenkins. Jenkins — это полнофункциональная, расширяемая система непрерывной интеграции, которая широко используется в разработке программного обеспечения. С ее помощью мы организовали регулярное выполнение нашего сценария на Python и отправку уведомлений на электронную почту. Вы также увидели, как можно расширять возможности Jenkins с помощью плагинов.

Вы узнали, как интегрировать Jenkins с репозиторием GitHub и инициировать сборку при проверке кода. Интеграция позволяет использовать средства совместной работы, доступные в GitHub.

В главе 15 речь пойдет о разработке через тестирование с использованием Python.

15

TDD для сетей

В предыдущих главах мы применяли Python для взаимодействия с сетевыми устройствами, мониторинга и обеспечения безопасности сети, автоматизации процессов и соединения локально размещенных сетей с публичными облачными провайдерами. Мы прошли длинный путь от администрирования сети исключительно в окне терминала с использованием CLI. Созданные нами сервисы работают вместе как хорошо отлаженный механизм и образуют стройную автоматизированную программируемую сеть. Однако сети никогда не стоят на месте; они постоянно меняются, адаптируясь к бизнес-требованиям. Что, если созданные нами сервисы работают не оптимально? Как вы уже видели на примере мониторинга и систем управления исходным кодом, мы активно ищем возможные проблемы.

В этой главе мы расширим идею активного обнаружения за счет *разработки через тестирование (Test-Driven Development, TDD)*.

Эта глава охватывает следующие темы:

- обзор разработки через тестирование;
- топологию как код;
- написание тестов для сетей;
- интеграцию `pytest` с Jenkins;
- pyATS и Genie.

Начнем с обзора методов TDD, а затем обсудим их применение в контексте сетей. Мы также рассмотрим примеры использования Python в сочетании с TDD и перейдем от узкоспециализированных к общесетевым тестам.

Обзор разработки через тестирование

Идея TDD существует уже давно. Лидером этого движения (а также методологии гибкой разработки) обычно называют американского разработчика Кента Бека. Согласно методологии гибкой разработки цикл разработки «сборка — тестирование — развертывание» должен быть очень коротким; все требования к программному обеспечению оформляются в виде тестов, которые обычно создаются до написания кода. Код принимается, только если он успешно проходит все тесты.

Аналогичный принцип можно применить к сетевым технологиям. Например, приступая к проектированию современной сети, мы разбиваем процесс на шаги, начинаем с общих архитектурных требований и заканчиваем сетевыми тестами.

1. Описание общих требований к новой сети. Зачем нужно проектировать новую сеть или какой-то ее участок? Например, для нового серверного оборудования, нового хранилища данных или новой микросервисной архитектуры.
2. Разбиваем требования на более мелкие и конкретные. Это, к примеру, может быть анализ платформы нового коммутатора, тестирование потенциально более эффективного протокола маршрутизации или создание новой топологии (например, утолщенное дерево). Каждое такое требование можно отнести к одной из двух категорий: *обязательные* и *дополнительные*.
3. Мы очерчиваем план тестирования и примеряем его к потенциальным решениям.
4. План тестирования выполняется в обратном порядке; мы начинаем с проверки возможностей, а затем внедряем их в общую топологию. В завершение мы пытаемся выполнить наш тест в условиях, максимально приближенных к реальным.

Сами того не осознавая, мы уже могли применять некоторые аспекты методологии TDD в традиционном процессе проектирования сетей. Это было одним из моих открытий, к которым я пришел, изучая принципы TDD. Мы уже, по сути, применяем эти методики без формального знакомства с ними.

Постепенно реализуя части сети в виде кода, мы можем использовать TDD еще активнее. Если ваша сетевая топология описана в иерархическом формате, таком как XML или JSON, каждый ее компонент может быть корректно размещен и выражен с помощью желаемого состояния, которое иногда называют «источником истины». Таким образом, мы можем писать тесты, которые проверяют,

насколько промышленное окружение отклоняется от этого состояния. Например, если согласно желаемому состоянию у нас должна быть полная сетка соседних iBGP-узлов, мы всегда можем написать тест, который проверяет, сколько таких узлов есть у наших промышленных устройств.

В общем случае процесс TDD включает следующие шаги.

1. Написать тест, определяющий нужный результат.
2. Запустить все тесты и посмотреть, выполняется ли новый тест без ошибок.
3. Написать код.
4. Снова выполнить тест.
5. Внести необходимые изменения, если тест выполняется с ошибками.
6. Повторить.

Вы сами решаете, насколько точно следовать рекомендациям. Лично я предпочитаю относиться к этим рекомендациям как к целям, к которым нужно стремиться, и не воспринимаю их буквально. Например, процесс TDD требует сначала писать тесты, а только потом уже код (или в нашем случае компоненты сети). Но лично я предпочитаю перед написанием тестов осмотреть рабочую версию сети или кода. Это придает мне дополнительную уверенность, поэтому, если бы кто-то оценивал мой подход к TDD, я бы получил жирную единицу. Еще я люблю переключаться между разными уровнями тестирования; иногда я тестирую небольшой участок сети, а иногда провожу сквозное тестирование системного уровня, проверяя достижимость узлов или трассировку маршрута.

По моему мнению, не существует универсального подхода к тестированию. Все зависит от личных предпочтений и масштабов проекта. И этот взгляд разделяют большинство инженеров, с которыми мне приходилось работать. Всегда следует помнить о базовых принципах TDD, чтобы у нас был рабочий план действий, но никто не сможет оценить ваш стиль решения проблем лучше, чем вы сами.

Прежде чем углубляться в TDD, рассмотрим базовую терминологию.

Разные виды тестов

Рассмотрим основные понятия TDD.

- **Модульный тест.** Проверяет небольшой участок кода — отдельную функцию или класс.

- **Интеграционный тест.** Проверяет несколько компонентов кодовой базы; разные участки кода собираются вместе и проверяются как единое целое. Этот тест может проверять один или несколько модулей Python.
- **Системный тест.** Проверяет систему от начала и до конца. Этот тест работает на уровне, максимально приближенном к тому, что будет видеть конечный пользователь.
- **Функциональный тест.** Проверяет отдельную функцию.
- **Охват тестами.** Под этим термином подразумевается полнота охвата прикладного кода тестами. В роли оценки обычно выступает объем кода, затронутый тестированием.
- **Среда тестирования.** Фиксированная среда, в рамках которой выполняются тесты. Среда тестирования нужна для того, чтобы создать предсказуемое и неизменное окружение, в котором многократное выполнение теста дает одни и те же результаты.
- **Подготовка и очистка.** Все предварительные шаги выполняются в рамках подготовки, а их результаты удаляются при очистке.

Эти термины могут показаться слишком тесно связанными с разработкой ПО, и некоторые из них могут не иметь никакого отношения к сетевым технологиям. Помните, что они описывают определенную концепцию или этап. Мы будем использовать их в оставшейся части этой главы. Значение этих понятий прояснится для вас по мере их применения в контексте сетевых технологий. Итак, начнем относиться к топологии нашей сети как к коду.

Топология как код

Когда речь заходит о представлении топологии в виде кода, инженер может вмешаться в разговор и заявить: «Сеть слишком сложная, ее невозможно свести к коду!» Такое случалось на собраниях, в которых я участвовал. С одной стороны, у нас были разработчики программного обеспечения, которые хотели воплотить инфраструктуру в коде, а с другой — сетевые инженеры, которые утверждали, что это невозможно. Но прежде, чем становиться на сторону последних и выражать свое негодование, попробуйте посмотреть на вещи объективно. Возможно, вам будет легче, если я скажу, что в этой книге мы уже использовали код для описания топологии.

Если взять любой файл с топологией VIRL, с которым мы уже имели дело, он представляет собой обычный XML-документ, описывающий отношения между

узлами. Например, в этой главе наша лаборатория будет иметь такую топологию (рис. 15.1).

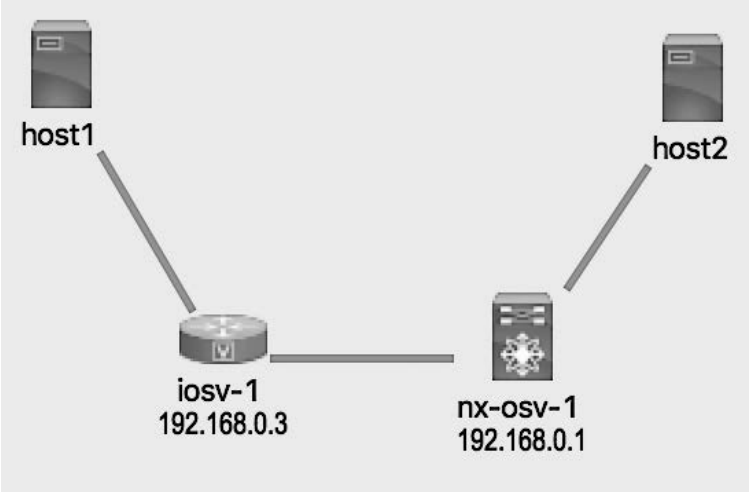


Рис. 15.1. Граф топологии нашей лаборатории

Если открыть файл топологии, `chapter15_topology.virl`, в текстовом редакторе, мы увидим, что он имеет формат XML и описывает отношения между узлами. На верхнем (корневом) уровне находится узел `<topology>` с дочерними узлами `<node>`. Каждый дочерний узел состоит из различных расширений и записей:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<topology xmlns="http://www.cisco.com/VIRL" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" schemaVersion="0.95"
xsi:schemaLocation="http://www.cisco.com/VIRL https://raw.github.com/
CiscoVIRL/schema/v0.95/virl.xsd">
<extensions>
<entry key="management_network" type="String">flat</entry>
</extensions>
```

Дочерний узел имеет такие атрибуты, как `name`, `type` и `location`. Мы также увидим конфигурацию каждого узла в текстовом значении элемента `<entry key="config">`:

```
<node name="iosv-1" type="SIMPLE" subtype="IOSv" location="182,162"
ipv4="192.168.0.3">
<extensions>
<entry key="static_ip" type="String">172.16.1.20</entry>
<entry key="config" type="string">
```

```
! IOS Config generated on 2018-07-24 00:23
! by autonetkit_0.24.0
!
hostname iosv-1
boot-start-marker
boot-end-marker
!
...
</node>
<node name="nx-osv-1" type="SIMPLE" subtype="NX-OSv"
location="281,161" ipv4="192.168.0.1">
    <extensions>
        <entry key="static_ip" type="String">172.16.1.21</entry>
        <entry key="config" type="string">! NX-OSv Config generated on
2018-07-24 00:23
! by autonetkit_0.24.0
!
version 6.2(1)
license grace-period
!
hostname nx-osv-1
```

Даже если узел является хостом, он тоже представлен в виде XML-элемента в том же файле:

```
...
<node name="host2" type="SIMPLE" subtype="server" location="347,66">
    <extensions>
        <entry key="static_ip" type="String">172.16.1.23</entry>
        <entry key="config" type="string">#cloud-config
bootcmd:
ln -s -t /etc/rc.d /etc/rc.local
hostname: host2
manage_etc_hosts: true
runcmd:
start ttyS0
systemctl start getty@ttyS0.service
systemctl start rc-local
<annotations/>
<connection dst="/virl:topology/virl:node[1]/virl:interface[1]" src="/
virl:topology/virl:node[3]/virl:interface[1]"/>
<connection dst="/virl:topology/virl:node[2]/virl:interface[1]" src="/
virl:topology/virl:node[1]/virl:interface[2]"/>
<connection dst="/virl:topology/virl:node[4]/virl:interface[1]" src="/
virl:topology/virl:node[2]/virl:interface[2]"/>
</topology>
```

Выражая сетевые компоненты в виде кода, мы можем объявить этот код источником истины для нашей сети. Мы можем написать тесты, которые будут сравнивать нашу промышленную сеть с ее описанием. В качестве основы для этого описания возьмем файл топологии.

С помощью Python можно извлечь элементы из файла топологии и преобразовать их в типы данных этого языка для дальнейшей работы с ними. В файле `chapter15_1_xml.py` используется объект `ElementTree`, который разбирает файл топологии `virl` и формирует словарь с информацией о наших устройствах:

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

import xml.etree.ElementTree as ET
import pprint

with open('chapter15_topology.virl', 'rt') as f:
    tree = ET.parse(f)

devices = {}

for node in tree.findall('./{http://www.cisco.com/VIRL}node'):
    name = node.attrib.get('name')
    devices[name] = {}
    for attr_name, attr_value in sorted(node.attrib.items()):
        devices[name][attr_name] = attr_value

# Дополнительные атрибуты
devices['iosv-1']['os'] = '15.6(3)M2'
devices['nx-osv-1']['os'] = '7.3(0)D1(1)'
devices['host1']['os'] = '16.04'
devices['host2']['os'] = '16.04'

pprint.pprint(devices)
```

В результате получится словарь языка Python с устройствами, соответствующими нашему файлу топологии.

Добавим в этот словарь элементы:

```
(venv) $ python chapter15_1_xml.py
{'host1': {'location': '117,58',
           'name': 'host1',
           'os': '16.04',
           'subtype': 'server',
           'type': 'SIMPLE'},
'host2': {'location': '347,66',
           'name': 'host2',
           'os': '16.04',
           'subtype': 'server',
           'type': 'SIMPLE'},
'iosv-1': {'ipv4': '192.168.0.3',
           'location': '182,162',
           'name': 'iosv-1',
           'os': '15.6(3)M2',
           'subtype': 'IOSv',
           'type': 'SIMPLE'},
```

```
'nx-osv-1': {'ipv4': '192.168.0.1',
             'location': '281,161',
             'name': 'nx-osv-1',
             'os': '7.3(0)D1(1)',
             'subtype': 'NX-OSv',
             'type': 'SIMPLE'}}
```

Для сравнения этого «источника истины» с устройствами, развернутыми в промышленной среде, воспользуемся сценарием из главы 3, `cisco_nxapi_2.py`, который извлекает версию ПО устройства NX-OSv. Затем сравним значение, полученное из нашего файла топологии, с информацией из устройства. Позже можно будет использовать встроенный в Python модуль `unittest`, чтобы написать тесты.

Модуль unittest рассмотрим чуть позже, но при желании вы можете еще вернуться к этому примеру.

Вот код с `unittest` в файле `chapter15_2_validation.py`, который нас интересует:

```
import unittest
<опущено>
# Тест на основе unittest
class TestNXOSVersion(unittest.TestCase):
    def test_version(self):
        self.assertEqual(nxos_version, devices['nx-osv-1']['os'])

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

Если запустить этот тест, он успешно завершится, поскольку версия ПО в промышленной среде совпадает с той, которую мы ожидали:

```
(venv) $ python chapter15_2_validation.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Если специально поменять ожидаемую версию NX-OSv, чтобы тест выполнился с ошибкой, мы увидим следующий вывод:

```
(venv) $ python chapter15_3_test_fail.py
F
======================================================================
FAIL: test_version (__main__.TestNXOSVersion)
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "chapter15_3_test_fail.py", line 50, in test_version
    self.assertEqual(nxos_version, devices['nx-osv-1']['os'])
AssertionError: '7.3(0)D1(1)' != '7.4(0)D1(1)'
- 7.3(0)D1(1)
?   ^
+ 7.4(0)D1(1)
?   ^


----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Вы видите, тест вернул отрицательный результат; причина провала — несовпадение двух версий. Этот пример показывает, что модуль `unittest` отлично подходит для проверки имеющейся кодовой базы на соответствие ожидаемому результату. Познакомимся с этим модулем поближе.

Модуль unittest

Стандартная библиотека Python включает в себя модуль `unittest`, который занимается проверкой результатов выполнения тестов, сравнивая два значения, чтобы определить успех или неудачу теста. В предыдущем примере мы видели, как метод `assertEqual()` сравнивает два значения и возвращает либо `True`, либо `False`. Аналогичное применение модуля `unittest` продемонстрировано в сценарии `chapter15_4_unittest.py`:

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

import unittest

class SimpleTest(unittest.TestCase):
    def test(self):
        one = 'a'
        two = 'a'
        self.assertEqual(one, two)
```

При использовании из командной строки модуль `unittest` способен автоматически обнаруживать тесты в сценарии:

```
(venv) $ python -m unittest chapter15_4_unittest.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Помимо сравнения двух значений, можно проверить, равно ли ожидаемое значение `True` или `False`. Можно также сгенерировать свое сообщение на случай неуспеха теста:

```
#!/usr/bin/env python3
# Примеры из https://pymotw.com/3/unittest/index.html#module-unittest

import unittest

class Output(unittest.TestCase):
    def testPass(self):
        return

    def testFail(self):
        self.assertFalse(True, 'this is a failed message')

    def testError(self):
        raise RuntimeError('Test error!')

    def testAssesrtTrue(self):
        self.assertTrue(True)

    def testAssertFalse(self):
        self.assertFalse(False)
```

Чтобы получить более подробный вывод, используйте параметр `-v`:

```
(venv) $ python -m unittest -v chapter15_5_more_unittest.py
testAssertFalse (chapter15_5_more_unittest.Output) ... ok
testAssesrtTrue (chapter15_5_more_unittest.Output) ... ok
testError (chapter15_5_more_unittest.Output) ... ERROR
testFail (chapter15_5_more_unittest.Output) ... FAIL
testPass (chapter15_5_more_unittest.Output) ... ok

======================================================================
ERROR: testError (chapter15_5_more_unittest.Output)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/echou/Mastering_Python_Networking_third_edition/Chapter15/
chapter15_5_more_unittest.py", line 14, in testError
    raise RuntimeError('Test error!')
RuntimeError: Test error!

======================================================================
FAIL: testFail (chapter15_5_more_unittest.Output)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/echou/Mastering_Python_Networking_third_edition/Chapter15/
chapter15_5_more_unittest.py", line 11, in testFail
    self.assertFalse(True, 'this is a failed message')
AssertionError: True is not false : this is a failed message

----------------------------------------------------------------------
```

```
Ran 5 tests in 0.001s

FAILED (failures=1, errors=1)
```

Начиная с версии Python 3.3, в модуль `unittest` включена библиотека `mock` для создания фиктивных объектов (https://docs.python.org/3/library/unittest.mock.html). Этот очень полезный модуль можно использовать для выполнения фиктивных API-вызовов к удаленному ресурсу. Например, мы уже видели, как с помощью NX-API можно получить номер версии NX-OS. Что, если у нас нет под рукой устройства с NX-OS, но мы все равно хотим выполнить наш тест? Для этого можно создать фиктивный объект с помощью модуля `unittest`.

В файле `chapter15_5_more_unittest_mocks.py` мы создали класс с методом для отправки API-вызовов по HTTP и ожидаем ответ формате JSON:

```
# Наш класс, выполняющий API-вызовы с помощью модуля requests
class MyClass:
    def fetch_json(self, url):
        response = requests.get(url)
        return response.json()
```

Мы также создали функцию, которая имитирует обращение к двум URL:

```
# Этот метод будет использоваться фиктивным объектом для подмены requests.get
def mocked_requests_get(*args, **kwargs):
    class MockResponse:
        def __init__(self, json_data, status_code):
            self.json_data = json_data
            self.status_code = status_code

        def json(self):
            return self.json_data

    if args[0] == 'http://url-1.com/test.json':
        return MockResponse({"key1": "value1"}, 200)
    elif args[0] == 'http://url-2.com/test.json':
        return MockResponse({"key2": "value2"}, 200)

    return MockResponse(None, 404)
```

Наконец, мы выполняем API-вызовы по двум URL, указанным в нашем тесте. Но при этом используем декоратор `mock.patch`, который перехватывает API-вызовы:

```
# Класс нашего теста
class MyClassTestCase(unittest.TestCase):
    # Мы подменяем 'requests.get' нашим собственным методом.
    # Фиктивный объект передается методу нашего теста.
    @mock.patch('requests.get', side_effect=mocked_requests_get)
```

```
    def test_fetch(self, mock_get):
        # Подменяем вызовы requests.get
        my_class = MyClass()
        # вызываем url-1
        json_data = my_class.fetch_json('http://url-1.com/test.json')
        self.assertEqual(json_data, {"key1": "value1"})
        # вызываем url-2
        json_data = my_class.fetch_json('http://url-2.com/test.json')
        self.assertEqual(json_data, {"key2": "value2"})
        # вызываем url-3, который мы не подменяли
        json_data = my_class.fetch_json('http://url-3.com/test.json')
        self.assertIsNone(json_data)

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

Этот тест будет пройден без выполнения настоящих API-вызовов к удаленной конечной точке. Здорово, правда?

```
(venv) $ python chapter15_5_more_unittest_mocks.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Дополнительную информацию о модуле `unittest` ищите в цикле статей Дага Хеллмана под названием *Python module of the week* (https://pymotw.com/3/unittest/index.html#module-unittest); там много коротких и наглядных примеров использования `unittest`. И, как всегда, хороший источник информации — документация Python: https://docs.python.org/3/library/unittest.html.

Еще о тестировании в Python

Помимо встроенной библиотеки `unittest`, существует множество фреймворков для тестирования, разрабатываемых сообществом Python. Один из самых развитых и интуитивно понятных инструментов такого рода — `pytest`. Этот фреймворк применим для всех видов и уровней тестирования ПО. Им могут пользоваться разработчики, инженеры по обеспечению качества, любители, практикующие TDD, и проекты с открытым исходным кодом.

Многие крупномасштабные открытые проекты, такие как Mozilla и Dropbox, перешли с `unittest` или `nose` (еще один фреймворк для тестирования в Python) на `pytest`. В числе привлекательных особенностей `pytest` можно выделить поддержку сторонних плагинов, простую модель окружений тестирования и переопределение утверждений.

Если вы хотите поближе познакомиться с фреймворком pytest, я настоятельно рекомендую книгу Брайана Оккена (Brian Okken) Python Testing with pytest. Еще один хороший источник информации — документация pytest: https://docs.pytest.org/en/latest/.

Работа модуля `pytest` основана на командной строке; он может автоматически запускать написанные нами тесты, отыскивая функции с именами, начинающимися с префикса `test`. Установим модуль:

```
(venv) $ pip install pytest
(venv) $ python
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pytest
>>> pytest.__version__
'5.2.2'
```

И рассмотрим несколько примеров с `pytest`.

Примеры с pytest

Первый пример, `chapter15_6_pytest_1.py`, — это простое утверждение с двумя значениями:

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

def test_passing():
    assert(1, 2, 3) == (1, 2, 3)

def test_failing():
    assert(1, 2, 3) == (3, 2, 1)
```

Если запустить `pytest` с параметром `-v`, мы получим развернутое объяснение, почему тест завершился неудачей. За такой подробный вывод мы любим `pytest`:

```
(venv) $ pytest -v chapter15_6_pytest_1.py
================================== test session starts =================
=================
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0 --
/home/echou/venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 2 items

chapter15_6_pytest_1.py::test_passing PASSED
[ 50%]
```

```
chapter15_6_pytest_1.py::test_failing FAILED
[100%]

======================================== FAILURES ========================
==================
_____________________________________ test_failing ____________________
____________________

    def test_failing():
>       assert(1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3
E         Full diff:
E         - (1, 2, 3)
E         ?  ^     ^
E         + (3, 2, 1)
E         ?  ^     ^

chapter15_6_pytest_1.py:7: AssertionError
============================== 1 failed, 1 passed in 0.03s
==============================
```

Во втором примере, `chapter15_7_pytest_2.py`, мы создадим объект `router` и инициализируем одни его атрибуты значением `None`, а другие — значениями по умолчанию. Затем с помощью `pytest` проверим два экземпляра этого объекта: один со значениями по умолчанию, а другой — без:

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

class router(object):
    def __init__(self, hostname=None, os=None, device_type='cisco_ios'):
        self.hostname = hostname
        self.os = os
        self.device_type = device_type
        self.interfaces = 24

def test_defaults():
    r1 = router()
    assert r1.hostname == None
    assert r1.os == None
    assert r1.device_type == 'cisco_ios'
    assert r1.interfaces == 24

def test_non_defaults():
    r2 = router(hostname='lax-r2', os='nxos', device_type='cisco_nxos')
    assert r2.hostname == 'lax-r2'
    assert r2.os == 'nxos'
    assert r2.device_type == 'cisco_nxos'
    assert r2.interfaces == 24
```

Запустим тест и посмотрим, корректно ли работает экземпляр объекта со значениями по умолчанию:

```
(venv) $ pytest chapter15_7_pytest_2.py
================================== test session starts =================
==================
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 2 items

chapter15_7_pytest_2.py ..
[100%]

=================================== 2 passed in 0.01s ==================
==================
```

Если в предыдущем примере заменить `unittest` на `pytest` (как в `chapter15_8_pytest_3.py`), код становится проще:

```
# тест на основе pytest
def test_version():
    assert devices['nx-osv-1']['os'] == nxos_version
```

Запустим этот тест с помощью утилиты командной строки `pytest`:

```
(venv) $ pytest chapter15_8_pytest_3.py
================================== test session starts =================
==================
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 1 item

chapter15_8_pytest_3.py .
[100%]

=================================== 1 passed in 0.09s ==================
==================
```

Мне модуль `pytest` кажется более понятным, чем `unittest`. Однако `unittest` входит в стандартную библиотеку, поэтому многие команды разработчиков используют для тестирования именно его.

Тестировать можно не только код, но и сеть целиком. В конце концов, пользователей больше заботит корректная работа их сервисов и приложений, а не отдельных компонентов. В следующем разделе мы обсудим написание тестов для сетей.

Написание тестов для сетей

До сих пор мы в основном писали тесты для кода на языке Python. Для проверки значений `True/False` или условий «равно/не равно» использовались библиотеки `unittest` и `pytest`. Мы также попробовали применить фиктивные

объекты для перехвата API-вызовов на случай, когда нет устройства с подходящим API, но нам все равно нужно выполнить наши тесты.

Несколько лет назад Мэтт Освальт объявил о выходе инструмента для проверки изменений в сетевой конфигурации под названием ToDD (Testing On Demand: Distributed). Это открытый фреймворк, предназначенный для тестирования сетевых соединений и распределенности. Больше об этом проекте читайте на странице GitHub: https://github.com/toddproject/todd. Освальт также рассказывал об этом проекте в подкасте Packet Pushers Priority Queue 81, Network Testing with ToDD: https://packetpushers.net/podcast/pq-show-81-network-testing-todd/.

Поговорим в этом разделе о том, как писать тесты, которые можно использовать в мире сетевых технологий. Для тестирования и мониторинга сетей существует множество коммерческих продуктов. За годы работы я неоднократно с ними сталкивался. Но в этом разделе я буду применять в тестах более простые открытые инструменты.

Тестирование доступности

Поиск проблем часто начинают с выполнения небольшой проверки доступности. И в этом контексте утилита `ping` — лучший друг сетевого инженера. Она позволяет проверить доступность хоста в IP-сети, отправляя ему небольшие пакеты.

Проверку на основе `ping` можно автоматизировать с помощью модуля `os` или `subprocess`:

```
>>> import os
>>> host_list = ['www.cisco.com', 'www.google.com']
>>> for host in host_list:
...     os.system('ping -c 1 ' + host)
...
PING www.cisco.com(2001:559:19:289b::b33 (2001:559:19:289b::b33)) 56 data
bytes
64 bytes from 2001:559:19:289b::b33 (2001:559:19:289b::b33): icmp_seq=1
ttl=60 time=11.3 ms

--- www.cisco.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 11.399/11.399/11.399/0.000 ms
0
PING www.google.com(sea15s11-in-x04.1e100.net (2607:f8b0:400a:808::2004))
56 data bytes
```

```
64 bytes from sea15s11-in-x04.1e100.net (2607:f8b0:400a:808::2004): icmp_
seq=1 ttl=54 time=10.8 ms

--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 10.858/10.858/10.858/0.000 ms
0
```

Еще одно преимущество модуля `subprocess` — возможность захвата вывода:

```
>>> import subprocess
>>> for host in host_list:
...     print('host: ' + host)
...     p = subprocess.Popen(['ping', '-c', '1', host],
stdout=subprocess.PIPE)
...
host: www.cisco.com
host: www.google.com
>>> print(p.communicate())
(b'PING www.google.com(sea15s11-in-x04.1e100.net
(2607:f8b0:400a:808::2004)) 56 data bytes\n64 bytes from sea15s11-in-
x04.1e100.net (2607:f8b0:400a:808::2004): icmp_seq=1 ttl=54 time=16.9
ms\n\n--- www.google.com ping statistics ---\n1 packets transmitted,
1 received, 0% packet loss, time 0ms\nrtt min/avg/max/mdev =
16.913/16.913/16.913/0.000 ms\n', None)
>>>
```

Эти два модуля оказываются крайне полезными во многих ситуациях. Они способны выполнить любую команду, доступную в окружениях Linux и Unix.

Тестирование задержек сети

Понимание задержек в сети может быть субъективным. Сетевым инженерам часто приходится слышать жалобы пользователей на медленную работу сети. Но представление о «медленности» у каждого свое.

Было бы очень полезно написать такие тесты, которые могли бы превратить субъективные подозрения в объективные значения. Проверки должны выполняться регулярно, чтобы мы могли сравнивать значения за разные периоды.

Иногда сделать это непросто, так как сеть по своей природе не хранит информации о своем состоянии. Факт успешности получения одного пакета вовсе не гарантирует, что и следующий пакет будет успешно доставлен. Лучший подход, который я видел за годы своей работы, состоит в частом использовании `ping` для большого количества хостов с записью результатов и построением графа сети. Мы можем воспользоваться инструментами из предыдущего примера,

чтобы измерить и сохранить время отклика. В файле `chapter15_10_ping.py` показано, как это реализовать:

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

import subprocess

host_list = ['www.cisco.com', 'www.google.com']

ping_time = []

for host in host_list:
    p = subprocess.Popen(['ping', '-c', '1', host], stdout=subprocess.PIPE)
    result = p.communicate()[0]
    host = result.split()[1]
    time = result.split()[13]
    ping_time.append((host, time))

print(ping_time)
```

В этом случае результат имеет вид кортежа и помещается в список:

```
(venv) $ python chapter15_10_ping.py
[(b'www.cisco.com(2001:559:19:289b::b33', b'time=16.0'),
 (b'www.google.com(sea15s11-in-x04.1e100.net', b'time=11.4')]
```

Это далеко не идеальный подход, а лишь отправная точка для реализации мониторинга и поиска неполадок. Однако в отсутствие других инструментов он дает базовые объективные значения.

Тестирование безопасности

В главе 6 мы уже рассматривали одно из лучших средств для тестирования безопасности, Scapy. В этой области существует много других открытых инструментов, но ни один из них не дает гибкости, присущей формированию собственных пакетов.

Еще один отличный инструмент для тестирования сетевой безопасности — `hping3` (http://www.hping.org/). Он позволяет легко генерировать сразу множество пакетов. Например, с помощью следующей однострочной команды можно сымитировать атаку типа SYN-flood:

```
# НЕ ИСПОЛЬЗУЙТЕ ЭТУ КОМАНДУ В ПРОМЫШЛЕННОМ ОКРУЖЕНИИ! #
echou@ubuntu:/var/log$ sudo hping3 -S -p 80 --flood 192.168.1.202
HPING 192.168.1.202 (eth0 192.168.1.202): S set, 40 headers + 0 data
bytes hping in flood mode, no replies will be shown
^C
--- 192.168.1.202 hping statistic ---
```

```
2281304 packets transmitted, 0 packets received, 100% packet loss roundtrip
min/avg/max = 0.0/0.0/0.0 ms
echou@ubuntu:/var/log$
```

Это утилита командной строки, поэтому мы можем использовать модуль `subprocess`, чтобы автоматизировать нужные нам тесты с `hping3`.

Тестирование транзакций

Сеть — ключевой, но не единственный элемент инфраструктуры. Пользователей зачастую заботят сервисы, работающие поверх сети. Если пользователю не удается посмотреть видео на YouTube или послушать подкаст, то ему кажется, что сервис вышел из строя. И даже если мы знаем, что с транспортным уровнем сети все в порядке, пользователю от этого не легче.

В связи с этим наши тесты должны как можно точнее отражать то, что испытывает пользователь. Нам вряд ли удастся продублировать весь сайт YouTube (разве что ваша компания является частью Google), но мы можем реализовать сервис прикладного уровня настолько близко к границе сети, насколько это возможно. Затем мы можем регулярно имитировать пользовательские транзакции, тем самым выполняя транзакционный тест.

Я часто использую модуль `http` из стандартной библиотеки Python, когда мне нужно быстро проверить доступность веб-сервиса на прикладном уровне. Мы уже применяли его для мониторинга сети в главе 5, но это будет полезно повторить:

```
# Python 3
(venv) $ python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 ...
127.0.0.1 - - [25/Jul/2018 10:15:23] "GET / HTTP/1.1" 200 -
```

Если можно сымитировать полный цикл взаимодействий с нужным сервисом, это замечательно. А если нет, то для ограниченного тестирования того или иного веб-сервиса прекрасно подойдет простой модуль `http` из стандартной библиотеки Python.

Тестирование сетевой конфигурации

По моему мнению, лучший тест сетевой конфигурации — сгенерировать ее из стандартизированных шаблонов и регулярно выполнять резервное копирование. Мы уже видели, как с помощью шаблонов Jinja2 можно стандартизировать нашу конфигурацию для отдельных типов устройств или ролей. Это позволяет

избежать многих ошибок, которые, к примеру, происходят во время копирования/вставки и обусловлены человеческим фактором.

Сгенерировав конфигурацию, можно написать для нее тесты для проверки достижения ожидаемых характеристик и только потом развертывать ее в промышленных устройствах. Например, в нашей сети не должно быть одинаковых локальных адресов, поэтому мы можем написать тест, который проверяет, не встречается ли локальный адрес в новой конфигурации в других наших устройствах.

Тестирование сценариев Ansible

За время использования Ansible мне, насколько я помню, не приходилось прибегать к инструментам наподобие `unittest` для тестирования своих сценариев. Сценарии, как правило, используют модули, которые уже были проверены их авторами.

Если вам нужно легковесное средство для проверки данных, взгляните на Cerberus (https://docs.python-cerberus.org/en/stable/).

Ansible предоставляет модульные тесты для своей библиотеки модулей. Это на сегодня единственный способ организации тестирования с помощью Python в рамках процесса непрерывной интеграции Ansible. Актуальные модульные тесты ищите в каталоге `/test/units` (https://github.com/ansible/ansible/tree/devel/test/units).

Описание стратегии тестирования Ansible можно найти в следующих документах:

- *тестирование Ansible:* https://docs.ansible.com/ansible/2.5/dev_guide/testing.html;
- *модульное тестирование:* https://docs.ansible.com/ansible/2.5/dev_guide/testing_units.html;
- *модульное тестирование модулей Ansible:* https://docs.ansible.com/ansible/2.5/dev_guide/testing_units_modules.html.

Существует один любопытный фреймворк для тестирования Ansible — Molecule (https://pypi.org/project/molecule/2.16.0/). Он пытается облегчить процесс разработки и тестирования ролей Ansible. Molecule поддерживает разные виды инстансов, операционных систем и дистрибутивов. Лично у меня нет опыта работы с этим инструментом, но я бы начал именно с него, если бы мне нужно было тестировать мои роли Ansible.

У вас уже должно сложиться представление о том, как тестировать различные аспекты своей сети, будь то доступность, задержки, безопасность, транзакции или конфигурация. Но можно ли интегрировать тестирование с системой управления исходным кодом, такой как Jenkins? Да, можно. О том, как это сделать, — в следующем разделе.

Интеграция pytest с Jenkins

Системы *непрерывной интеграции (Continuous Integration, CI)*, такие как Jenkins, часто используются для выполнения тестов после каждой фиксации кода. Это одно из главных преимуществ применения таких систем.

Представьте, что в вашей компании есть инженер-невидимка, который постоянно следит за любыми изменениями в сети; если что-то поменялось, он старательно выполняет массу тестов, чтобы убедиться в том, что ничего не сломалось. Кто бы отказался от такого коллеги?

Рассмотрим пример интеграции `pytest` в задание Jenkins.

Интеграция с Jenkins

Прежде чем внедрять тесты в процесс CI, установим несколько плагинов, которые помогут визуализировать эту операцию. Речь идет о плагинах *build-name-setter* и *Test Results Analyzer* (рис. 15.2).

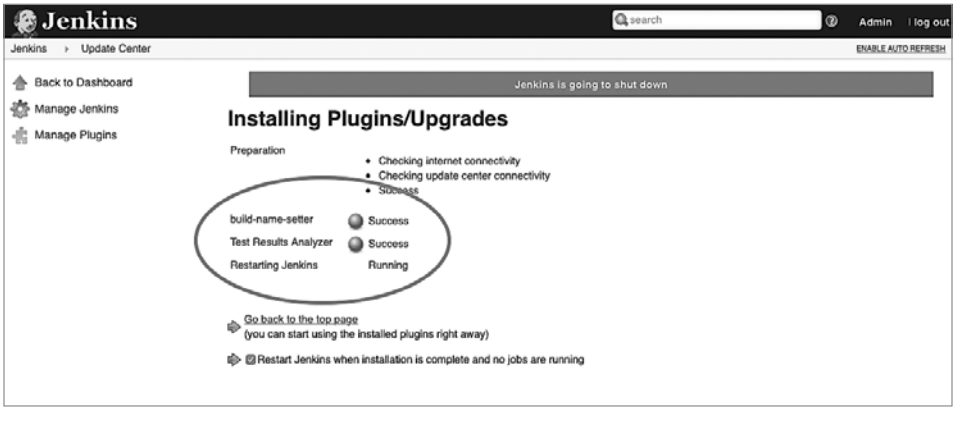


Рис. 15.2. Установка плагинов в Jenkins

Тест, который мы выполним, обратится к устройству NX-OS и получит номер версии операционной системы. Так мы убедимся, что нам доступен API устройства Nexus. Полный код сценария ищите в файле `chapter15_9_pytest_4.py`. Ниже приводится только интересующий нас фрагмент и результат выполнения:

```
    def test_transaction():
        assert nxos_version != False
```

```
(venv) $ pytest chapter15_9_pytest_4.py
================================== test session starts =================
==================
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 1 item

chapter15_9_pytest_4.py .
[100%]

================================== 1 passed in 0.10s ===================
==================
```

Воспользуемся параметром `--junit-xml=results.xml`, чтобы сгенерировать файл для Jenkins:

```
(venv) $ pytest --junit-xml=result.xml chapter15_9_pytest_4.py
================================== test session starts =================
==================
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 1 item

chapter15_9_pytest_4.py .
[100%]

- generated xml file: /home/echou/Mastering_Python_Networking_third_
edition/Chapter15/result.xml -
================================== 1 passed in 0.10s ===================
==================
```

Сохраним этот сценарий в репозитории GitHub. Я предпочитаю размещать тесты в отдельном каталоге, поэтому создал каталог `/tests` и поместил в него наш тестовый файл (рис. 15.3).

Создадим новый проект под названием `chapter15_example1` (рис. 15.4).

Мы можем скопировать предыдущее задание, чтобы не повторять все заново (рис. 15.5).

Добавим этап на основе `pystep` в раздел Execute shell (Выполнить консольную команду) (рис. 15.6).

Рис. 15.3. Репозиторий проекта

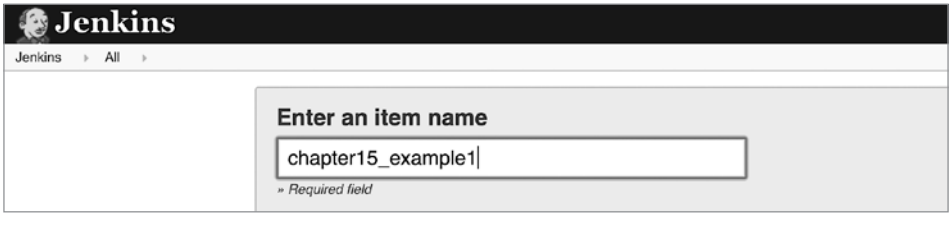


Рис. 15.4. Ввод имени проекта в Jenkins

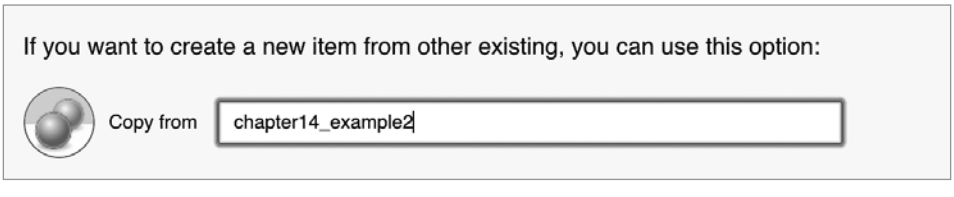


Рис. 15.5. Использование функции Copy from (Скопировать из) в Jenkins

Рис. 15.6. Консольные команды

В качестве шага, который выполняется после сборки, добавим Publish JUnit test result report (Опубликовать отчет о результатах теста JUnit) (рис. 15.7).

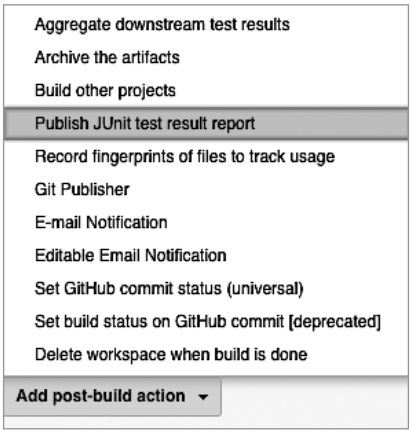


Рис. 15.7. Шаг, который выполняется после сборки

В качестве имени файла с результатами JUnit укажем `results.xml` (рис. 15.8).

Рис. 15.8. Местоположение отчета о тесте в формате XML

Выполнив сборку несколько раз, мы увидим диаграмму Test Results Analyzer (Анализатор результатов тестирования) (рис. 15.9).

Результаты тестов также доступны на домашней странице проекта. Спровоцируем неудачное выполнение теста, отключив управляющий интерфейс на устройстве Nexus. Тест с ошибкой должен немедленно появиться на диаграмме Test Result Trend (Изменение результатов тестирования) на панели управления проектом (рис. 15.10).

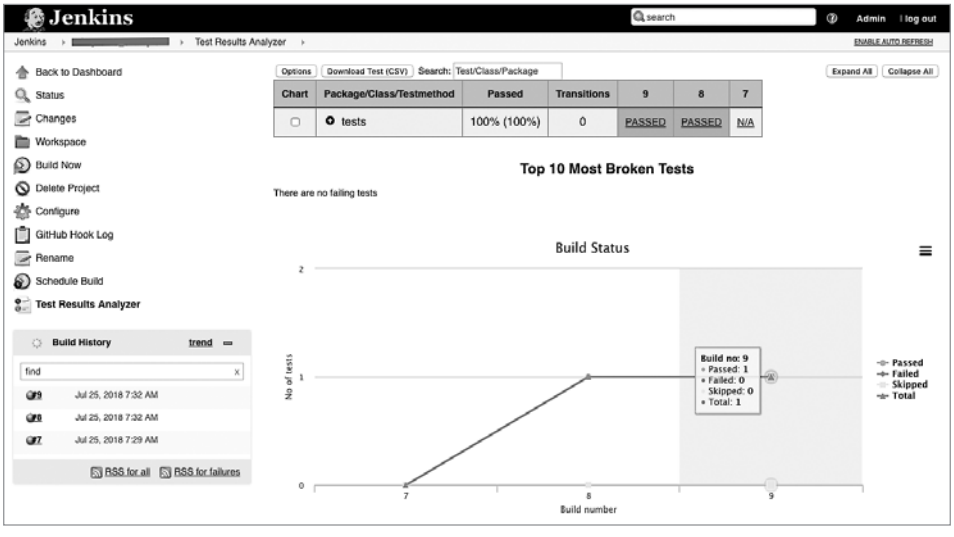


Рис. 15.9. Диаграмма в анализаторе тестов в Jenkins

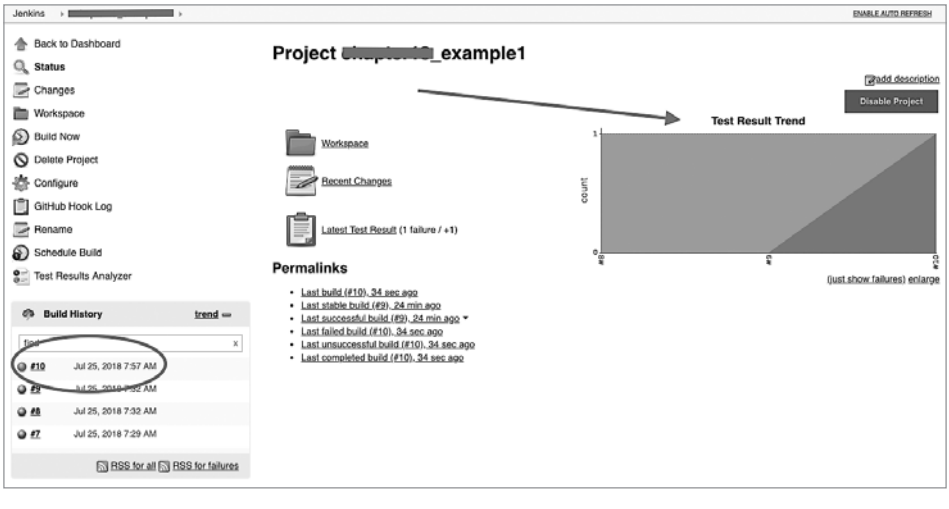


Рис. 15.10. Диаграмма изменения результатов тестирования в Jenkins

Это простой, но наглядный пример. Этот же подход применим для создания других интеграционных тестов в Jenkins.

В следующем разделе мы рассмотрим многофункциональный фреймворк тестирования с названием pyATS, разработанный компанией Cisco (и недавно

ставший открытым). Открытие исходного кода такого развитого фреймворка в пользу сообщества — широкий жест со стороны Cisco.

pyATS и Genie

pyATS (https://developer.cisco.com/pyats/) — это экосистема сквозного тестирования, изначально разработанная компанией Cisco и ставшая открытой в конце 2017 года. Ранее библиотека pyATS называлась Genie; эти названия используются в одном и том же контексте. Ввиду своего происхождения этот фреймворк целиком и полностью ориентирован на тестирование сетей.

pyATS и одноименная библиотека (также известная как Genie) удостоились награды Cisco Pioneer Award в 2018 году. Компания Cisco заслуживает аплодисментов за публикацию исходного кода этого фреймворка. Так держать, Cisco DevNet!

Этот фреймворк доступен в PyPI:

```
(venv) echou@network-dev-2:~$ pip install pyats
```

Для начала рассмотрим некоторые демонстрационные сценарии в GitHub-репозитории, https://github.com/CiscoDevNet/pyats-sample-scripts. Тесты начинаются с создания файла испытательной модели (testbed) в формате YAML. Создадим простой testbed-файл `chapter15_pyats_testbed_1.yml` для нашего устройства IOSv-1. Он похож на файл реестра `hosts`, знакомый нам по работе с Ansible:

```
testbed:
    name: Chapter_15_pyATS
    tacacs:
      username: cisco
    passwords:
      tacacs: cisco
      enable: cisco

devices:
    iosv-1:
        alias: iosv-1
        type: ios
        connections:
          defaults:
            class: unicon.Unicon
          management:
            ip: 172.16.1.20
            protocol: ssh
```

```
topology:
    iosv-1:
        interfaces:
            GigabitEthernet0/2:
                ipv4: 10.0.0.5/30
                link: link-1
                type: ethernet
            Loopback0:
                ipv4: 192.168.0.3/32
                link: iosv-1_Loopback0
                type: loopback
```

В нашем первом сценарии, `chapter15_11_pyats_1.py`, мы загрузим testbed-файл, подключимся к устройству, выполним команду `show version` и затем отключимся от устройства:

```
from pyats.topology import loader

testbed = loader.load('chapter15_pyats_testbed_1.yml')

testbed.devices
ios_1 = testbed.devices['iosv-1']

ios_1.connect()

print(ios_1.execute('show version'))

ios_1.disconnect()
```

Запустив этот сценарий, мы увидим смесь из сообщений о подготовке pyATS и вывода самого устройства. Это похоже на сценарии Paramiko, которые мы видели ранее, но в данном случае соединение устанавливает pyATS:

```
(venv) $ python chapter15_11_pyats_1.py
[2019-11-10 08:11:55,901] +++ iosv-1 logfile /tmp/iosv-1-default-
20191110T081155900.log +++
[2019-11-10 08:11:55,901] +++ Unicon plugin generic +++
<опущено>
[2019-11-10 08:11:56,249] +++ connection to spawn: ssh -l cisco
172.16.1.20, id: 140357742103464 +++
[2019-11-10 08:11:56,250] connection to iosv-1
[2019-11-10 08:11:56,314] +++ initializing handle +++
[2019-11-10 08:11:56,315] +++ iosv-1: executing command 'term length 0' +++
term length 0
iosv-1#
[2019-11-10 08:11:56,354] +++ iosv-1: executing command 'term width 0' +++
term width 0
iosv-1#
[2019-11-10 08:11:56,386] +++ iosv-1: executing command 'show version' +++
show version
<опущено>
```

Второй пример более развернутый. Он включает настройку и установку соединения, сами тесты и последующее отключение от устройства. Сначала добавим устройство `nxosv-1` в testbed-файл `chapter15_pyats_testbed_2.yml`. Оно будет участвовать в проверке связи с устройством `iosv-1`:

```
nxosv-1:
    alias: nxosv-1
    type: ios
    connections:
      defaults:
        class: unicon.Unicon
      vty:
        ip: 172.16.1.21
        protocol: ssh
```

В сценарии `chapter15_12_pyats_2.py` используются различные декораторы из модуля `aetest`, входящего в состав pyATS. Помимо методов подготовки и очистки, в классе `PingTestCase` имеется тест `ping`:

```
@aetest.loop(device = ('ios1',))
class PingTestcase(aetest.Testcase):

    @aetest.test.loop(destination = ('10.0.0.5', '10.0.0.6'))
    def ping(self, device, destination):
        try:
            result = self.parameters[device].ping(destination)
```

На практике предпочтительнее передавать testbed-файл как аргумент командной строки:

```
(venv) $ python chapter15_12_pyats_2.py --testbed chapter15_pyats_
testbed_2.yml
```

Вывод этого сценария похож на тот, что мы видели в предыдущем примере, если не считать дополнительных разделов `STEPS Report` и `Detailed Results` в каждом тесте. В выводе также указано имя журнального файла, сохраненного в каталоге `/tmp`:

```
2019-11-10T08:23:08: %AETEST-INFO: Starting common setup
<опущено>
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
--------------------+
2019-11-10T08:23:22: %AETEST-INFO: |                          STEPS Report
|
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
--------------------+
<опущено>
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
----------------------------------------+
```

```
2019-11-10T08:23:22: %AETEST-INFO: |
Detailed Results |
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
----------------------------------------+
2019-11-10T08:23:22: %AETEST-INFO:   SECTIONS/TESTCASES   RESULT
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
----------------------------------------+
2019-11-10T08:23:22: %AETEST-INFO: |                     Summary|
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
----------------------------------------+
<опущено>
2019-11-10T08:23:22: %AETEST-INFO:  Number of PASSED
3
```

Фреймворк pyATS отлично подходит для автоматизированного тестирования. Но, учитывая его происхождение, ему не хватает поддержки других производителей, помимо Cisco.

Еще один открытый инструмент для проверки сетей — Batfish от ребят из IntentionNet (https://github.com/batfish/batfish). Его назначение — проверка изменений в конфигурации перед развертыванием.

Изучение `pytest` требует определенных усилий; этот инструмент фактически предлагает свой уникальный подход к тестированию, к которому нужно привыкнуть. К тому же он в своем текущем состоянии вполне ожидаемо завязан на платформы Cisco. Но поскольку `pytest` теперь открытый проект, мы можем сами поучаствовать в поддержке других производителей и изменении синтаксиса или процесса тестирования. Мы подошли к концу главы, поэтому подведем итоги.

Резюме

В этой главе мы обсудили подход к разработке через тестирование и то, как его можно применить в сетевых технологиях. Мы начали с обзора TDD и затем рассмотрели примеры с использованием модулей `unittest` и `pytest` на языке Python. Тестирование доступности, конфигурации и безопасности сети можно проводить с помощью Python и простых инструментов командной строки Linux.

Вы также увидели, как можно использовать инструменты непрерывной интеграции, такие как Jenkins. За счет внедрения тестов в процесс непрерывной интеграции мы можем укрепить уверенность в корректности наших изменений. По крайней мере, этот подход должен помочь нам выявлять любые ошибки

раньше наших пользователей. pyATS — это инструмент с открытым исходным кодом, который недавно выпустила компания Cisco. Он представляет собой фреймворк для автоматизации тестирования сетей.

Мы не можем доверять тому, что не было протестировано. Поэтому все в нашей сети должно быть проверено с помощью программных инструментов настолько, насколько это возможно. TDD, как и многие другие концепции из мира разработки, представляет собой бесконечный цикл. Мы стремимся к максимальному охвату кода тестами, но даже при 100%-ном охвате всегда можно найти несколько новых направлений и тестов для реализации. Это особенно актуально для сетевых технологий, где роль сети зачастую играет интернет, протестировать который на все 100 % попросту невозможно.

Мы подошли к концу книги. Надеюсь, вы получили такое же удовольствие от ее чтения, какое получил я, когда писал ее. Хочу искренне поблагодарить вас за потраченное время. Желаю вам успехов и удачи при работе с сетевыми технологиями с использованием Python!