

O'REILLY®

Python и DevOps

Ключ к автоматизации Linux



Ной Гифт, Кеннеди Берман
Альфредо Деза, Григ Георгиу

Python for DevOps

Learn Ruthlessly Effective Automation

*Noah Gift, Kennedy Behrman,
Alfredo Deza, and Grig Gheorghiu*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Python и DevOps

Ключ к автоматизации Linux

Ной Гифт, Кеннеди Берман
Альфредо Деза, Григ Георгиу



Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018.1
УДК 004.43
П12

Ной Гифт, Кеннеди Берман, Альфредо Деца, Григ Георгиу

П12 Python и DevOps: Ключ к автоматизации Linux. — СПб.: Питер, 2022. — 544 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-2929-4

За последнее десятилетие технологии сильно изменились. Данные стали хитом, облака — вездесущими, и всем организациям понадобилась автоматизация. В ходе таких преобразований Python оказался одним из самых популярных языков программирования. Это практическое руководство научит вас использовать Python для повседневных задач администрирования Linux с помощью наиболее удобных утилит DevOps, включая Docker, Kubernetes и Terraform.

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

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492057697 англ.

Authorized Russian translation of the English edition of Python for DevOps
ISBN 9781492057697

© 2020 Noah Gift, Kennedy Behrman, Alfredo Deza, Grig Gheorghiu.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same

ISBN 978-5-4461-2929-4

© Перевод на русский язык ООО Издательство «Питер», 2022

© Издание на русском языке, оформление ООО Издательство «Питер», 2022

© Серия «Бестселлеры O'Reilly», 2022

Краткое содержание

Предисловие	19
От издательства.....	28
Глава 1. Основы Python для DevOps.....	29
Глава 2. Автоматизация работы с файлами и файловой системой	69
Глава 3. Работа с командной строкой.....	93
Глава 4. Полезные утилиты Linux.....	124
Глава 5. Управление пакетами	154
Глава 6. Непрерывная интеграция и непрерывное развертывание.....	189
Глава 7. Мониторинг и журналирование	207
Глава 8. Pytest для DevOps	240
Глава 9. Облачные вычисления	268
Глава 10. Инфраструктура как код.....	299
Глава 11. Контейнерные технологии: Docker и Docker Compose	328
Глава 12. Координация работы контейнеров: Kubernetes	355
Глава 13. Технологии бессерверной обработки данных	391
Глава 14. MLO и разработка ПО для машинного обучения	439
Глава 15. Инженерия данных.....	472
Глава 16. Истории из практики DevOps и интервью.....	499
Об авторах	539
Об иллюстрации на обложке	541

Оглавление

Предисловие	19
Что DevOps означает для авторов	20
Как пользоваться этой книгой	22
Условные обозначения	23
Использование примеров кода	24
Благодарности	25
От издательства	28
Глава 1. Основы Python для DevOps	29
Установка и запуск Python	30
Командная оболочка Python	30
Блокноты Jupiter	31
Процедурное программирование	32
Переменные	32
Основные математические операции	33
Комментарии	33
Встроенные функции	34
Print	34
Range	35

Контроль выполнения	35
if/elif/else	36
Циклы for	37
Циклы while	38
Обработка исключений	39
Встроенные объекты	40
Что такое объект	40
Методы и атрибуты объектов	41
Последовательности	42
Функции	55
Синтаксис функции	55
Функции как объекты	57
Анонимные функции	57
Регулярные выражения	58
Поиск	59
Наборы символов	60
Классы символов	61
Группы	61
Поименованные группы	61
Найти все	62
Поисковый итератор	62
Подстановка	63
Компиляция	63
Отложенное вычисление	64
Генераторы	64
Генераторные включения	65
Дополнительные возможности IPython	66
Выполнение инструкций командной оболочки Unix с помощью IPython	66
Упражнения	68

Глава 2. Автоматизация работы с файлами и файловой системой	69
Чтение и запись файлов	69
Поиск в тексте с помощью регулярных выражений	79
Обработка больших файлов.....	81
Шифрование текста.....	82
Хеширование с помощью пакета hashlib	82
Шифрование с помощью библиотеки cryptography	83
Модуль os	85
Управление файлами и каталогами с помощью os.path.....	86
Обход дерева каталогов с помощью os.walk	90
Пути как объекты: библиотека pathlib.....	91
 Глава 3. Работа с командной строкой	 93
Работа с командной оболочкой.....	93
Взаимодействие с интерпретатором с помощью модуля sys	93
Взаимодействие с операционной системой с помощью модуля os	94
Порождение процессов с помощью модуля subprocess.....	95
Создание утилит командной строки	97
Использование sys.argv	99
argparse	101
click.....	106
fire	110
Реализация плагинов.....	115
Ситуационный анализ: разгоняем Python с помощью утилит командной строки.....	116
Динамический компилятор Numba	117
Использование GPU с помощью CUDA Python	119
Многоядерное многопоточное выполнение кода Python с помощью Numba	120
Кластеризация методом k-средних.....	122
Упражнения.....	123

Глава 4. Полезные утилиты Linux	124
Дисковые утилиты.....	125
Измерение быстродействия	126
Разделы диска	128
Получение информации о конкретном устройстве.....	129
Сетевые утилиты	131
SSH-туннелирование	131
Оценка быстродействия HTTP с помощью Apache Benchmark (ab)	132
Нагрузочное тестирование с помощью molotov	133
Утилиты для получения информации о загрузке CPU.....	136
Просмотр процессов с помощью htop	136
Работаем с Bash и ZSH	138
Настройка командной оболочки Python под свои нужды	140
Рекурсивные подстановки	140
Поиск и замена с запросами подтверждения	141
Удаление временных файлов Python	143
Вывод списка процессов и его фильтрация	143
Метка даты/времени Unix.....	144
Комбинирование Python с Bash и ZSH.....	144
Генератор случайных чисел.....	145
Существует ли нужный мне модуль?	146
Переходим из текущего каталога по пути к модулю	146
Преобразование CSV-файла в JSON.....	147
Однострочные сценарии Python	148
Отладчики	148
Быстро ли работает конкретный фрагмент кода?	149
strace.....	150
Вопросы и упражнения	153
Задача на ситуационный анализ	153

Глава 5. Управление пакетами	154
Почему пакетная организация программ так важна	155
Случаи, когда пакетная организация программ не нужна	155
Рекомендации по пакетной организации программ.....	156
Информативный контроль версий.....	156
Журнал изменений	158
Выбор стратегии	159
Решения для создания пакетов.....	159
Нативные пакеты Python.....	160
Создание пакетов для Debian	166
Создание пакетов RPM	174
Диспетчеризация с помощью systemd.....	180
Долгоживущие процессы	181
Настройка	182
Юниты systemd	183
Установка юнита.....	185
Управление журналами.....	186
Вопросы и упражнения	188
Задача на ситуационный анализ	188
Глава 6. Непрерывная интеграция и непрерывное развертывание	189
Ситуационный анализ примера из практики: перевод плохо работавшего сайта с WordPress на Hugo.....	189
Настройка Hugo	191
Преобразование WordPress в посты Hugo.....	192
Создание поискового индекса Algolia и его обновление	194
Координация с помощью Makefile.....	196
Развертывание с помощью AWS CodePipeline	196
Ситуационный анализ примера из практики: развертывание приложения Python App Engine с помощью Google Cloud Build.....	197
Ситуационный анализ примера из практики: NFSOPS.....	205

Глава 7. Мониторинг и журналирование	207
Ключевые понятия создания надежных систем.....	207
Неизменные принципы DevOps	208
Централизованное журналирование	209
Ситуационный анализ: база данных при промышленной эксплуатации разрушает жесткие диски	209
Производить или покупать?	210
Отказоустойчивость.....	211
Мониторинг	213
Graphite	214
StatsD.....	214
Prometheus	215
Телеметрия	219
Соглашения о наименованиях	222
Журналирование.....	224
Почему это так трудно.....	224
basicconfig.....	225
Углубляемся в конфигурацию.....	226
Распространенные паттерны.....	231
Стек ELK.....	232
Logstash.....	233
Elasticsearch и Kibana.....	235
Вопросы и упражнения	239
Задача на ситуационный анализ	239
Глава 8. Pytest для DevOps.....	240
Сверхспособности тестирования фреймворка pytest	240
Начало работы с pytest	241
Тестирование с помощью pytest.....	242
Отличия от unittest.....	244

Возможности pytest.....	245
conftest.py	246
Этот замечательный оператор assert	247
Параметризация	248
Фикстуры.....	250
Приступим	250
Встроенные фикстуры.....	252
Инфраструктурное тестирование	255
Что такое проверка системы	256
Введение в Testinfra	257
Подключение к удаленным узлам	258
Фикстуры и особые фикстуры	261
Примеры.....	263
Тестирование блокнотов Jupyter с помощью pytest	266
Вопросы и упражнения	267
Задача на ситуационный анализ	267
Глава 9. Облачные вычисления	268
Основы облачных вычислений.....	269
Типы облачных вычислений.....	271
Типы облачных сервисов	272
Инфраструктура как сервис	272
«Железо» как сервис	277
Платформа как сервис.....	278
Бессерверная обработка данных.....	278
Программное обеспечение как сервис.....	282
Инфраструктура как код	283
Непрерывная поставка.....	283
Виртуализация и контейнеры	283
Аппаратная виртуализация	283
Программно определяемые сети	284

Программно определяемое хранилище	285
Контейнеры	285
Трудные задачи и потенциальные возможности распределенной обработки данных.....	286
Конкурентное выполнение на Python, быстроедействие и управление процессами в эпоху облачных вычислений	289
Управление процессами.....	289
Процессы и дочерние процессы.....	289
Решение задач с помощью библиотеки multiprocessing.....	292
Ветвление процессов с помощью Pool()	293
Функция как сервис и бессерверная обработка данных.....	295
Повышение производительности Python с помощью библиотеки Numba	295
Динамический компилятор Numba	295
Высокопроизводительные серверы.....	296
Резюме	297
Вопросы	298
Вопросы на ситуационный анализ.....	298
Глава 10. Инфраструктура как код	299
Классификация инструментов автоматизации выделения инфраструктуры	301
Выделение инфраструктуры вручную.....	302
Автоматическое выделение инфраструктуры с помощью Terraform.....	304
Выделение корзины S3	304
Предоставление SSL-сертификата с помощью ACM AWS	307
Выделение раздачи Amazon CloudFront.....	308
Создание записи DNS Route 53	311
Копирование статических файлов в корзину S3.....	312
Удаление всех ресурсов AWS, выделенных с помощью Terraform.....	313

Автоматическое выделение инфраструктуры с помощью Pulumi	313
Создание нового проекта Pulumi на Python для AWS.....	314
Создание значений параметров конфигурации для стека staging	319
Создаем SSL-сертификат ACM.....	319
Выделение зоны Route 53 и записей DNS	320
Выделение раздачи CloudFront.....	323
Создание записи DNS Route 53 для URL сайта.....	324
Создание и развертывание нового стека.....	325
Упражнения.....	327
Глава 11. Контейнерные технологии: Docker и Docker Compose.....	328
Что такое контейнер Docker	329
Создание, сборка, запуск и удаление образов и контейнеров Docker.....	330
Публикация образов Docker в реестре Docker.....	334
Запуск контейнера Docker из одного образа на другом хост-компьютере	335
Запуск нескольких контейнеров Docker с помощью Docker Compose	337
Портирование сервисов docker-compose на новый хост-компьютер и операционную систему.....	350
Упражнения.....	354
Глава 12. Координация работы контейнеров: Kubernetes.....	355
Краткий обзор основных понятий Kubernetes	356
Создание манифестов Kubernetes на основе файла docker_compose.yaml с помощью Kompose	357
Развертывание манифестов Kubernetes на локальном кластере Kubernetes, основанном на minikube.....	359
Запуск кластера GKE Kubernetes в GCP с помощью Pulumi.....	374
Развертывание примера приложения Flask в GKE	377
Установка чартов Helm для Prometheus и Grafana	383
Удаление кластера GKE.....	389
Упражнения.....	390

Глава 13. Технологии бессерверной обработки данных.....	391
Развертывание одной и той же функции Python в облака большой тройки поставщиков облачных сервисов.....	394
Установка фреймворка Serverless.....	394
Развертывание функции Python в AWS Lambda.....	395
Развертывание функции Python в Google Cloud Functions.....	397
Развертывание функции на Python в Azure.....	403
Развертывание функции на Python на самохостируемых FaaS-платформах.....	408
Развертывание функции на Python в OpenFaaS.....	408
Выделение таблиц DynamoDB, функций Lambda и методов API Gateway с помощью AWS CDK.....	416
Упражнения.....	438
Глава 14. MLO и разработка ПО для машинного обучения	439
Что такое машинное обучение.....	439
Машинное обучение с учителем.....	440
Моделирование	442
Экосистема машинного обучения языка Python	445
Глубокое обучение с помощью PyTorch	445
Платформы облачного машинного обучения.....	449
Модель зрелости машинного обучения.....	451
Основная терминология машинного обучения.....	452
Уровень 1. Очерчивание рамок задачи и области определения, а также формулировка задачи	453
Уровень 2. Непрерывная поставка данных.....	453
Уровень 3. Непрерывная поставка очищенных данных	455
Уровень 4. Непрерывная поставка разведочного анализа данных	457
Уровень 5. Непрерывная поставка обычного ML и AutoML	457
Уровень 6. Цикл обратной связи эксплуатации ML	458

Приложение sklearn Flask с использованием Docker и Kubernetes.....	459
Разведочный анализ данных.....	463
Моделирование	464
Тонкая настройка масштабированного GBM.....	465
Подгонка модели	466
Оценка работы модели	466
adhoc_predict.....	467
Технологический процесс JSON.....	468
Масштабирование входных данных	468
adhoc_predict на основе выгрузки	470
Масштабирование входных данных	470
Вопросы и упражнения	471
Задача на ситуационный анализ	471
Вопросы на проверку усвоения материала	471
Глава 15. Инженерия данных	472
Малые данные.....	473
Обработка файлов малых данных	474
Запись в файл	474
Чтение файла.....	474
Конвейер с генератором для чтения и обработки строк.....	475
YAML	476
Большие данные	477
Утилиты, компоненты и платформы для работы с большими данными.....	479
Источники данных.....	480
Файловые системы	480
Хранение данных.....	481
Ввод данных в режиме реального времени	483
Ситуационный анализ: создание доморощенного конвейера данных	484
Бессерверная инженерия данных.....	485

AWS Lambda и события CloudWatch	486
Журналирование Amazon CloudWatch для AWS Lambda.....	486
Наполнение данными Amazon Simple Queue Service с помощью AWS Lambda	487
Подключение срабатывающего по событию триггера CloudWatch.....	492
Создание событийно-управляемых функций Lambda	493
Чтение событий Amazon SQS из AWS Lambda.....	493
Резюме.....	498
Упражнения.....	498
Задача на ситуационный анализ	498
Глава 16. Истории из практики DevOps и интервью.....	499
Киностудия не может снять фильм	500
Разработчик игр не может обеспечить поставку игрового ПО.....	503
Сценарии Python, запуск которых требует 60 секунд	505
Решаем горящие проблемы с помощью кэша и интеллектуальной телеметрии	506
Доавтоматизироваться до увольнения.....	507
Антипаттерны DevOps.....	509
Антипаттерн: отсутствие автоматизированного сервера сборки	509
Работать вслепую	509
Сложности координации как постоянная проблема.....	510
Отсутствие командной работы	511
Интервью.....	517
Гленн Соломон	517
Эндрю Нгуен	518
Габриэлла Роман	520
Ригоберто Рош	521
Джонатан Лакур.....	523
Вилле Туулос	525
Джозеф Рис.....	527

Тейо Хольцер	529
Мэтт Харрисон	531
Майкл Фоорд	533
Рекомендации	536
Вопросы	537
Интересные задачи	537
Дипломный проект	538
Об авторах	539
Об иллюстрации на обложке	541

Предисловие

Однажды, когда Ной¹ купался, его накрыл волн и, не дав вдохнуть, потопил в открытый океан. Лишь только он смог вдохнуть немного воздуха, его накрыл следующий, почти отняв остатки сил. И потопил еще сильнее. Только он начал переводить дух, еще один волн рухнул сверху. Чем больше он сопротивлялся волнам, тем больше терял сил. В какой-то момент Ной засомневался, что ему удастся выжить: ему нечем было дышать, все тело болело, и он в ужасе думал, что вот-вот утонет. Близость смерти помогла ему сфокусироваться на единственном, что могло его спасти: нужно беречь силы и не бороться с волнами, использовать их силу.

Учение в старте, где не практикуют DevOps, во многом напоминает этот день на пляже. Существуют проекты, в которых разброд и шатание приходится гореть на работе месяцами: все делается вручную, оповещения будят их каждую ночь, не позволяя выспаться. Единственный выход из этой трясины — внедрить DevOps.

Нужно сделать один шаг в правильную сторону, потом еще один, и так до тех пор, пока не наступит ясность. Во-первых, не стройте сервер сборки, не читайте тестировать кода, а вместо этого выполняйте вручную задачи. Делайте хоть что-то, что угодно, главное — делайте. Выполните первое действие правильно и не забудьте про это в будущем.

Зачастую стартеры и прочие компании попадают в ловушку «поиск супергероев». «Нам нужен специалист по производительности», потому что он решит наши проблемы с производительностью. «Нам нужен директор по прибыли», потому что он решит все наши проблемы с продажами. «Нам нужны специалисты по DevOps», потому что они решат наши проблемы с разбродом и шатанием.

Ной рассказывал: «В одной компании я работал над проектом с отставанием от графика более чем на год, причем веб-приложение переписывали трижды на различных языках программирования. Для завершения очередной версии всего лишь нужен был “специалист по производительности”. Я помню, что был единственным

¹ Один из авторов этой книги. — *Примеч. пер.*

достаточно хитрым или глупым, чтобы задать вопрос: “А кто такой специалист по производительности?” Мне ответили: “Это специалист, который отвечает за работу системы в целом”. В этот момент я понял, что они искали супергероя, который их спасет. Синдром нйм супергероев — лучший способ понять, что с новым программным продуктом или стартом чего-то сильно не так. Никого новый сотрудник не спасет компанию, если сначала он не спасет себя сам».

В других компаниях Нюта также приходилось слышать похожие фразы: «Если бы нам только удалось нанять ведущего специалиста по Erlang», или «Если бы нам только удалось нанять кого-нибудь, кто обеспечил бы нам прибыль», или «Если бы нам только удалось нанять разработчика на Swift» и т. д. Подобный нам — последнее, что нужно в шеступу или продукту. А насом деле нужно выяснить, что же именно с ними не так настолько, что их может спасти только супергерой.

В компании, которую я хотел нанять специалиста по производительности, окзалось, что реальная проблема заключалась в недостаточном техническом контроле. За него отвечали не те, кто должен был (и они перекрикивали тех, кто мог исправить ситуацию). Проблему удалось решить без всякого супергероя, достаточно окзалось убрать из проекта плохого исполнителя, прислушаться к сотруднику, который сам мог не только решить проблему, убрать объявление о вакансии, совершить по одному правильному действию за раз и внедрить компетентное управление технологическим процессом.

Никто не спасет вшившуюся систему, выжившая команда должна позботиться о себе с учетом честной командной работы, честного технологического процесса и веры в себя. Вшивших проблем не решит нам новых сотрудников, выпутаться из них помогут только честность с самими собой, продуманное отношение к сложившейся ситуации, понимание того, как вы в ней окзались, и выполнение по одному правильному действию за раз. Единственный супергерой — вы сами.

В нашу компанию, как утопающего в штормовом океане, никто не спасет, кроме нас самих. Именно вы и ваши сотрудники — столь нужные компании супергерои.

Из этого можно выбраться, и эта книга научит вас, как это сделать. Приступим.

Что DevOps означает для авторов

Дать точное определение многим быстрым понятиям индустрии разработки программного обеспечения очень непросто. Яркие примеры тем со множеством различных определений в зависимости от толкователя: облачные вычисления,

Agile и большие данные. Вместо того чтобы дать строгое определение DevOps, мы просто перечислим некоторые признаки и личия DevOps.

- Двустороннее сотрудничество между командой разработчиков (Dev) и эксплуатацией (Ops).
- Полный цикл эксплуатации занимает минуты, максимум часы, не дни, то и недели.
- Активное участие разработчиков в поддержке продукта, в противном случае все возвращается к традиционному противостоянию разработчиков и специалистов по эксплуатации.
- Специалистам по эксплуатации необходимы навыки разработчиков, по крайней мере знание bash и Python.
- Разработчикам необходимы навыки эксплуатации — их обязанности не заканчиваются написанием кода, касаются и развертывания системы в промышленной эксплуатации, и мониторинг уведомлений.
- Автоматизация, в том числе и еще раз автоматизация. Без навыков разработчика превильная автоматизация невозможен, как и без навыков эксплуатации.
- Идеальный вариант — разработчики в полном сервисе, по крайней мере в смысле развертывания кода.
- Реализация с использованием конвейеров CI/CD.
- GitOps.
- Двусторонний *всеохватный* обмен информацией между Dev и Ops (инструменты, знания и т. д.).
- Непрерывная совместная работа в сферах проектирования, реализации и развертывания — и, конечно, автоматизация — никогда не будет успешной без сотрудничества.
- Нет автоматизации — значит не работает нормально.
- Культура: в более технологический процесс, вне иерархии.
- Лучше микросервисы, чем монолитная архитектура.
- Системы непрерывного развертывания — душа и сердце команды разработчиков ПО.
- Никких супергероев.
- Непрерывная поставка — не один из возможных вариантов, необходимость.

Как пользоваться этой книгой

Читая эту книгу можно в любом порядке. Можете открыть любую главу, к которой хотите, — вы найдете там что-то полезное для своей работы. Если у вас уже есть опыт программирования на языке Python, можете пропустить главу 1. Если вам интересны истории из практики и ситуационный анализ, то же интервью, можете сначала взглянуть в главу 16.

Основные темы

Содержимое книги разбито на несколько основных тем. Первая их группа — «Основы языка Python», охватывающая краткое вступление в этот язык, то же введение в структуру текста, описание утилит командной строки и введение в работу с файлами.

Далее идет условный раздел «Эксплуатация», включающий описание полезных утилит Linux, управление пакетами, системы сборки, мониторинг и автоматизированное тестирование. Все эти вопросы жизненно важны для овладения искусством применения DevOps на практике.

Следующий раздел — «Основы облачных вычислений», он включает главы, посвященные облачным вычислениям, инфраструктуре как коду, Kubernetes и бессерверной работе и другим. Сейчас в индустрии много проблем, связанных с нехваткой перспективных кадров, умеющих работать в облаке. Когда вы овладеете информацией из этого раздела, то немедленно ощутите результаты своей работы и в карьере.

Далее идет раздел «Данные», в котором мы рассмотрим задачи машинного обучения, теории инженерии данных с точки зрения DevOps. Здесь вы также найдете полноценное, от начала, описание проекта машинного обучения: создание, развертывание и ввод в эксплуатацию модели машинного обучения с помощью Flask, Sklearn, Docker и Kubernetes.

Последний раздел — глава 16 — посвящен ситуационному анализу, интервью и историям из практики DevOps. Этот раздел — прекрасное чтение и сон грядущий.

Основы языка Python

- Глава 1. Основы Python для DevOps.
- Глава 2. Автоматизация работы с файлами и файловой системой.
- Глава 3. Работа с командной строкой.

Эксплуатация

- Гл в 4. Полезные утилиты Linux.
- Гл в 5. Управление пакетами.
- Гл в 6. Непрерывная интеграция и непрерывное развертывание.
- Гл в 7. Мониторинг и журналирование.
- Гл в 8. Pytest для DevOps.

Основы облачных вычислений

- Гл в 9. Облачные вычисления.
- Гл в 10. Инфраструктура как код.
- Гл в 11. Контейнерные технологии: Docker и Docker Compose.
- Гл в 12. Координация работы контейнеров: Kubernetes.
- Гл в 13. Технологии бессерверной обработки данных.

Данные

- Гл в 14. MLO и роль ботк-ПО для машинного обучения.
- Гл в 15. Инженерия данных.

Ситуационный анализ

- Гл в 16. Истории из практики DevOps и интервью.

Условные обозначения

В книге используются следующие шрифтовые соглашения.

Курсив

Отмечает новые термины.

Рубленый шрифт

Им обозначены URL и адрес электронной почты.

Моноширинный шрифт

Используется для листингов программ.

Жирный моноширинный шрифт

Представляет собой команды или другой текст, который должен быть в точности набран пользователем.

Моноширинный курсив

Отметьте текст, который необходимо изменить пользователями или знаниями, определяемыми контекстом.

Еще один моноширинный шрифт

Используется внутри блоков для имен и расширений файлов, ссылки на элементы программ, типичные переменные или имен функций, базовых переменных окружения, операторы и ключевые слова.



Данный символ означает совет или указание.



Это общее примечание.



Такой символ указывает на предупреждение или предостережение.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания по адресу <https://pythondevops.com>. Можете также посмотреть материалы по DevOps, близкие к коду, приведенному в книге, на канале YouTube Pragmatic AI Labs (<https://oreil.ly/QIYte>).

По поводу любых технических вопросов и проблем, возникших у вас при использовании примеров кода, пожалуйста, пишите по адресу электронной почты technical@pythondevops.com.

Эта книга создана, чтобы помочь вам делиться своим опытом. В целом, если к книге прилагается какой-либо пример кода, можете применять его в своих программах и документах. Обращаясь к нему с разрешения, нет необходимости, разве что вы копируете значительную часть кода. Например, написание программы, использующей несколько фрагментов кода из этой книги, не требует отдельного разрешения. Для продажи или распространения компакт-диск с примерами из книг O'Reilly, конечно, разрешение нужно. Ответ на вопрос путем цитирования

этой книги, в том числе примеров кода, не требует разрешения. Включение значительного количества кода примеров из этой книги в документацию в своего программного продукта — требует.

Мы ценим, хотя и не требуем, ссылки на первоисточник. В ссылку входят название, автор, издательство и ISBN, например: «Ной Гифт, Кеннеди Бермэн, Альфредо Дез, Григ Георгиу. Python и DevOps: ключ к успеху в мире Linux. — СПб.: Питер, 2022, 978-5-4461-2929-4».

Если вам кажется, что использование в наших примерах кода выходит за рамки разумного применения или другого разрешения, не стесняйтесь, связываясь с нами по адресу permissions@oreilly.com.

У этой книги есть своя веб-страница со списком ошибок, примерами и прочей дополнительной информацией. Найти ее можно по адресу <https://oreil.ly/python-for-devops>.

Благодарности

Для нас было бы поблгодарить двух основных научных редакторов данной книги.

Уэс Новак (Wes Novack) — архитектор и специалист по общедоступным облачным системам, также приложениям SaaS интернет-машин. Он отвечает за проектирование, создание сложных систем с инфраструктурой высокой доступности, конвейеры непрерывной поставки и быстрым выпуском версий в рамках больших многоязычных экосистем микросервисов, размещаемых в AWS и GCP, и управление ими. Уэс использует в своей работе множество языков программирования, фреймворков и утилит для описания инфраструктуры кода, в том числе и упрощения рутинных операций. Он принимает активное участие в деятельности технологического сообщества, в частности в обучении, семинарах и конференциях, кроме того, ему принадлежит видеокурс Pluralsight. Уэс — ярый сторонник методологии CALMS для DevOps: культура (Culture), автоматизация (Automation), бережливость (Lean), измерения (Measurement) и совместное использование (пространство знаний) (Sharing). Вы можете найти его в Twitter (@WesleyTech) или посетить личный блог (<https://wesnovack.com>).

Брэд Андерсен (Brad Andersen) — разработчик и архитектор программного обеспечения, разработчик в сфере проектирования и разработчик ПО уже 30 лет, всегда выступавший в роли катализатора изменений и внедрения новшеств. Он играл роли руководителя и разработчика в самых разных компаниях, от стартапов до корпораций. В настоящее время Брэд получает степень магистра одних

в Калифорнийском университете в Беркли. Больше информации вы можете найти в профиле Брэдли на LinkedIn (<https://www.linkedin.com/in/andersen-bradley>).

Мы также хотели бы поблагодарить Джереми Яброу (Jeremy Yabrow) и Колин Б. Эрмэн (Colin B. Erdman) за их вклад в виде замечательных идей и персональных отзывов.

Ной

Я хотел бы поблагодарить моих соавторов Григ, Кеннеди и Альфредо. Было замечательно работать с такой эффективной командой.

Кеннеди

Спасибо моим соавторам, работать с которыми было одно удовольствие. И спасибо моей семье за терпение и понимание.

Альфредо

В 2010 году — за девять лет до написания этой книги — я получил свою первую роботу — старшим разработчиком программного обеспечения. Мне был 31 год, у меня не было ни высшего образования, ни опыта работы. Этот робот означал меньшие деньги и отсутствие медицинской страховки, но я очень многому научился, познакомился с замечательными людьми и набрался опыта благодаря своей целеустремленности. Я не смог бы добиться ничего из этого без тех, кто дал мне реальные возможности и уверенность в верном направлении движения.

Спасибо Крису Бенсону (Chris Benson), взглядевшему во мне тягу к знаниям и всегда ходившему возможности сотрудничать со мной.

Спасибо Аляхндро Кадвиду (Alejandro Cadavid), осознавшему, что я способен исправлять то, чем никто больше не хотел заниматься. Вы помогли мне получить работу, когда никто (включая меня самого) не верил, что я могу пригодиться.

Карлос Колл (Carlos Coll) привел меня в программирование и не позволил уйти, даже когда я хотел. Умение программировать изменило всю мою жизнь, у Карлоса хватило терпения подтолкнуть меня учиться дальше и довести мою первую программу до промышленной эксплуатации.

Джони Бентон (Joni Benton), спасибо за веру в меня и помощь в получении мной первого постоянного места работы.

Спасибо Джонатану Лакуру (Jonathan LaCour), не только другу, который до сих пор вдохновляет меня добиваться большего. Ваши советы всегда были очень ценны для меня.

Ной, спасибо за дружбу и влияние, ты всегда был для меня колоссальным источником вдохновения. Мне всегда нравилось работать с тобой, как тогда, когда мы вместе влияли на инфраструктуру с нуля. Твое терпеливое руководство тогда, когда я еще ничего не знал про Python, изменило всю мою жизнь.

Наконец, огромная благодарность моей семье. Моей жене Кэти, никогда не сомневающейся в моей способности учиться и совершенствоваться, за великодушие и понимание во время всей подготовки к написанию этой книги. Моим детям, Эфрину, Игнатию и Алле: я люблю вас всех.

Григ

Спасибо всем создателям программного обеспечения с открытым исходным кодом. Без них наши роботы были бы не только менее яркими и приносили меньше удовольствия. Спасибо также всем, кто открыто делится своими знаниями. Наконец, хотел бы поблагодарить моих соавторов. Я получил от нашей совместной работы настоящее удовольствие.

От издательства

Вшизмечения, 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.

Основы Python для DevOps

DevOps — сочетание практики прогн много обеспечения с IT-з д ч ми — в последнее десятилетие обрело большую популярность. Традиционные границы между практикой прогн много обеспечения, его развертыванием, сопровождением и контролем качества все больше размываются, приводя к тесной интеграции различных групп специалистов. Python — язык программирования, популярный как в среде традиционных IT-з д ч, так и в DevOps благодаря сочетанию гибкости, широких возможностей и простоты использования.

Язык программирования Python появился в начале 1990-х и изначально предназначен для системного администрирования. В этой сфере он приобрел большую популярность и применяется очень широко. Python представляет собой универсальный язык программирования и используется практически во всех предметных областях, даже в киноиндустрии для создания спецэффектов. А совсем недавно он стал практически основным языком науки о данных и машинного обучения. Он присутствует повсюду, от медицины до биоинформатики. Python включает обширный инструментальный, охватывающий все нужды его пользователей. Изучение стандартной библиотеки Python, предоставляемой с любым дистрибутивом Python-функциональности, потребовало бы огромных усилий. А изучить все сторонние библиотеки, оживляющие экосистему Python, — поистине необъятная задача. К счастью, этого не требуется. Можно с большим успехом практиковать DevOps, изучив лишь малую толику Python.

В этой главе мы, основываясь на многолетнем опыте применения Python для DevOps, рассмотрим только необходимые элементы этого языка. Некоторые части Python составляют инструментальный, необходимый для решения ежедневных задач DevOps. После изучения этих основ вы сможете в следующих главах перейти к более продвинутым инструментам.

Установка и запуск Python

Чтобы опробовать в работе код из этой главы, вам понадобятся Python версии 3.7 или более поздней¹ и доступ к командной оболочке. В macOS X, Windows и большинстве дистрибутивов Linux для доступа к командной оболочке достаточно открыть терминал. Чтобы узнать используемую версию Python, откройте командную оболочку и наберите команду `python --version`:

```
$ python --version
Python 3.8.0
```

Скачать установочные пакеты Python можно непосредственно с сайта Python.org (<https://www.python.org/downloads>). Можно также воспользоваться системой управления пакетами, например Apt, RPM, MacPorts, Homebrew, Chocolatey и др.

Командная оболочка Python

Простейший способ работы с Python — воспользоваться встроенным интерактивным интерпретатором. Просто набрав в командной оболочке `python`, вы сможете интерактивно выполнять операции Python. Для выхода из командной оболочки наберите `exit()`:

```
$ python
Python 3.8.0 (default, Sep 23 2018, 09:47:03)
[Clang 9.0.0 (clang-900.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 2
3
>>> exit()
```

Сценарии Python

Код Python выполняется из файла с расширением `.py`:

```
# Мой первый сценарий Python
print('Hello world!')
```

Сохраните этот код в файле `hello.py`. Для вызова этого сценария наберите в командной оболочке `python` с последующим именем файла:

```
$ python hello.py
Hello world!
```

¹ По состоянию на сентябрь 2021 г. текущая стабильная версия — 3.9.7. — *Примеч. пер.*

Большая часть кода Python в промышленной эксплуатации выполняется именно в виде сценариев Python.

IPython

Помимо встроенной интерактивной командной оболочки, код Python позволяет выполнять несколько сторонних интерактивных командных оболочек. Одним из наиболее популярных — IPython (<https://ipython.org>). Возможности IPython включают *интроспекцию* (динамическое получение информации об объектах), подсветку синтаксиса, специальные *магические* команды (которые мы обсудим далее в этой главе) и многое другое, превращая изучение Python в сплошное удовольствие. Для установки IPython можно использовать систему управления пакетами Python `pip`:

```
$ pip install ipython
```

Без запуска не логичен запуск встроенной интерактивной командной оболочки, описанному в предыдущем разделе:

```
$ ipython
Python 3.8.0 (default, Sep 23 2018, 09:47:03)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.5.0 -- An enhanced Interactive Python. Type '?' for help.
In [1]: print('Hello')
Hello

In [2]: exit()
```

Блокноты Jupyter

Отпочкованный от проекта IPython проект Jupyter позволяет работать со специальными документными файлами, включающими текст, код и визуализации. Эти документы дают возможность сочетать выполнение кода, вывод результатов работы и форматирование текста. Jupyter позволяет сопровождать код документацией. Он стал популярен повсеместно, особенно в мире науки и данных. Установить Jupyter и запустить блокноты можно вот так:

```
$ pip install jupyter
$ jupyter notebook
```

Эта команда открывает веб-браузер и отображает текущий рабочий каталог. А там уже вы сможете открывать имеющиеся блокноты проекта или создавать новые.

Процедурное программирование

Если вы хоть немного сталкивались с программированием, то наверняка слышали термины «объектно-ориентированное программирование» (ООП) и «функциональное программирование» — так называются две различные архитектурные парадигмы организации программ. В качестве стартовой точки парадигмы процедурного программирования, одна из самых простых, подходит прекрасно. *Процедурное программирование* (procedural programming) означает описание инструкций компьютеру в виде упорядоченной последовательности:

```
>>> i = 3
>>> j = i + 1
>>> i + j
7
```

Как видите, в этом примере приведены три оператора, выполняемых по порядку, от первой строки до последней. Каждый из них использует сформированное предыдущими операторами состояние. В данном случае первый оператор присваивает переменной `i` значение 3. Во втором операторе значение этой переменной применяется для присваивания значения переменной `j`, в третьем значения обеих переменных складываются. Пусть вас не волнуют детали синтаксиса этих операторов, обратите внимание только на то, что они выполняются по порядку и зависят от состояния, сформированного предыдущими операторами.

Переменные

Переменная — это имя, указывающее на какое-то значение. В предыдущем примере встречались переменные `i` и `j`. Переменным в языке Python можно присваивать новые значения:

```
>>> dog_name = 'spot'
>>> dog_name
'spot'
>>> dog_name = 'rex'
>>> dog_name
'rex'
>>> dog_name = 't-' + dog_name
>>> dog_name
't-rex'
>>>
```

Типизация переменных языка Python динамическая. На практике это означает, что им можно повторно присваивать значения, относящиеся к другим типам или классам:


```
>>> big = 'large'
>>> big
'large'
>>> big = 1000*1000
>>> big
1000000
>>> big = {}
>>> big
{}
```

В данном случае одной и той же переменной присваиваются строковое значение, числовое, ассоциативный массив. Переменным можно повторно присваивать значения любого типа.

Основные математические операции

Для основных математических операций — сложения, вычитания, умножения и деления — существуют встроенные математические операторы:

```
>>> 1 + 1
2
>>> 3 - 4
-1
>>> 2*5
10
>>> 2/3
0.6666666666666666
```

Символ `//` служит для целочисленного деления. Символ `**` означает возведение в степень, `%` — оператор взятия остатка от деления:

```
>>> 5/2
2.5
>>> 5//2
2
>>> 3**2
9
>>> 5%2
1
```

Комментарии

Комментарии — текст, который интерпретатор Python игнорирует. Они удобны для документирования кода, некоторые сервисы умеют собирать их для создания отдельной документации. Однострочные комментарии отделяются от кода знаком `#` и могут начинаться с любого символа строки, так

и в любом месте в ней. Все символы от # до символ новой строки — это часть комментария:

```
# Комментарий
1 + 1 # Комментарий, следующий за оператором
```

Многострочные комментарии заключаются в блоки, начинающиеся и заканчиваемые символами `"""` или `'''`:

```
"""
Многострочный
комментарий.
"""

...

Этот оператор также представляет собой
многострочный комментарий
...
```

Встроенные функции

Функции — это сгруппированные операторы. Вызывается функция путем ввода ее имени с последующими скобками. Внутри них указываются принимаемые функции аргументы (при их наличии). В языке Python есть множество встроенных функций. Две из числен более часто используемых встроенных функций — `print` и `range`.

Print

Функция `print` генерирует видимую пользователям программы информацию. В интерактивной среде он не очень полезен, но при написании сценариев Python это важнейший инструмент. В предыдущем примере аргумент функции `print` выводится в консоль при запуске сценария:

```
# Мой первый сценарий Python
print("Hello world!")

$ python hello.py
Hello world!
```

С помощью функции `print` можно просматривать значения переменных или предоставлять обратную связь о состоянии программы. Функция `print` обычно выводит информацию в стандартный поток вывода, отображаемый в командной оболочке.

Range

Хотя `range` — один из встроенных функций, формально это вообще не функция, тип, служащий для представления последовательности чисел. При вызове конструктора `range()` возвращается представляющий последовательность чисел объект. Функция `range` принимает до трех целочисленных аргументов. При использовании лишь одного аргумента последовательность состоит из чисел от нуля до этого аргумента, не включая его. Вторым аргументом (при его наличии) отбрасывается точка, с которой начинается последовательность. Третьим аргументом служит для указания шага последовательности, по умолчанию равен 1:

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(5, 10, 3))
[5, 8]
>>>
```

Требования к оперативной памяти функции `range` невелики даже для больших последовательностей, ведь хранить нужно только значения начала, конца и шага последовательности. Функция `range` может проходить по большим последовательностям чисел без снижения быстродействия.

Контроль выполнения

В языке Python есть множество конструкций для управления потоком выполнения операций. Операторы, которые нужно выполнять вместе, можно группировать в блоки кода, которые можно выполнять многократно с помощью циклов `for` и `while` либо только при определенных условиях с помощью операторов `if`, цикл `while` или блоков `try-except`. Применение подобных конструкций — первый шаг к подлинной реализации возможностей программирования. В различных языках программирования разграничение блоков кода определяется разными соглашениями. Во многих C-подобных языках (очень важный язык, использовавшийся при написании операционной системы Unix) блок описывается путем заключения группы операторов в круглые скобки. В Python для этой цели применяются отступы. Операторы группируются по числу отступов в блоки, выполняемые как единое целое.



Интерпретатор Python не вжно, формируете вы отступы код пробел ми или символ ми тбуляции, главное — единообразие. Впрочем, руководство PEP-8 по стилю написания код Python (<https://oreil.ly/b5yU4>) рекомендует отделять уровни отступов с помощью четырех пробелов.

if/elif/else

Операторы `if/elif/else` — простейшее средство выбора веток код. Следующий непосредственно за оператором `if` блок код выполняется, если значение условия оператора равно `True`:

```
>>> i = 45
>>> if i == 45:
...     print('i is 45')
...
...
i is 45
>>>
```

Здесь использовлся оператор `==`, возвращающий `True`, если его операнды равны между собой, и `False` в противном случае. Этим блоком может следовать необязательный оператор `elif` или `else` со своим блоком, который в случае оператора `elif` выполняется, только если условие `elif` равно `True`:

```
>>> i = 35
>>> if i == 45:
...     print('i is 45')
... elif i == 35:
...     print('i is 35')
...
...
i is 35
>>>
```

При желании можно расположить друг за другом несколько ветвлений `elif`. Это напоминает множественный выбор с помощью операторов `switch` в других языках программирования. Добавление оператора `else` в конце позволяет выполнять блок только в том случае, если ни одно из предыдущих условий не равно `True`:

```
>>> i = 0
>>> if i == 45:
...     print('i is 45')
... elif i == 35:
...     print('i is 35')
... elif i > 10:
```

```

...     print('i is greater than 10')
... elif i%3 == 0:
...     print('i is a multiple of 3')
... else:
...     print('I don't know much about i...')
...
...
i is a multiple of 3
>>>

```

Операторы `if` могут быть вложенными, с содержащими операторы `if` блоками, которые выполняются только в том случае, если условие во внешнем операторе `if` равно `True`:

```

>>> cat = 'spot'
>>> if 's' in cat:
...     print("Found an 's' in a cat")
...     if cat == 'Sheba':
...         print("I found Sheba")
...     else:
...         print("Some other cat")
... else:
...     print(" a cat without 's'")
...
...
Found an 's' in a cat
Some other cat
>>>

```

Циклы `for`

Циклы `for` позволяют повторять выполнение блок операторов (блок кода) столько раз, сколько содержится элементов в *последовательности* (упорядоченной группе элементов). При проходе в цикле по последовательности блок кода обращается к текущему элементу. Чаще всего циклы используются для прохода по объекту `range` для выполнения какой-либо операции заданное число раз:

```

>>> for i in range(10):
...     x = i*2
...     print(x)
...
...
0
2
4
6

```

```
8
10
12
14
16
18
>>>
```

В этом примере блок кода имеет следующий вид:

```
...     x = i*2
...     print(x)
```

Этот код выполняется десять раз, причем переменной `i` каждый раз присваивается следующий элемент последовательности целых чисел из диапазона 0–9. Циклы `for` подходят для обхода любых типов последовательностей Python, как вы увидите далее в этой главе.

continue

Оператор `continue` позволяет пропустить один шаг в цикле и перейти сразу к очередному элементу последовательности:

```
>>> for i in range(6):
...     if i == 3:
...         continue
...     print(i)
...
...
0
1
2
4
5
>>>
```

Циклы while

Цикл `while` повторяет выполнение блока кода до тех пор, пока условие выражения равно `True`:

```
>>> count = 0
>>> while count < 3:
...     print(f"The count is {count}")
...     count += 1
... 
```

```
...
The count is 0
The count is 1
The count is 2
>>>
```

Главное — задать условие выхода из того цикла, в противном случае он будет выполняться до тех пор, пока программа не завершится в принудительно. Для этого можно, например, задать такое условное выражение, которое в конце концов окажется равным `False`. Либо воспользоваться оператором `break` для выхода из цикла с помощью вложенного условного оператора :

```
>>> count = 0
>>> while True:
...     print(f"The count is {count}")
...     if count > 5:
...         break
...     count += 1
...
...
The count is 0
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
>>>
```

Обработка исключений

Исключения — ошибки, которые могут привести к фатальному сбою программы, если их не обработать должным образом (перехватить). Благодаря их перехвату с помощью блока `try-except` программа может продолжить работу. Для создания того блока необходимо добавить отступы к блоку, в котором может возникнуть исключение, поместить перед ним оператор `try`, после него — оператор `except`. Таким образом будет следовать блок кода, который должен выполняться при возникновении ошибки:

```
>>> thinkers = ['Plato', 'PlayDo', 'Gumby']
>>> while True:
...     try:
...         thinker = thinkers.pop()
...         print(thinker)
...     except IndexError as e:
...         print("We tried to pop too many thinkers")
```

```
...         print(e)
...         break
...
...
...
Gumby
PlayDo
Plato
We tried to pop too many thinkers
pop from empty list
>>>
```

Существует множество встроенных исключений, например `IOError`, `KeyError` и `ImportError`. Во многих сторонних пакетах также описаны собственные классы исключений, вызывающих не серьезные проблемы, так что перехватывать их имеет смысл, только если вы уверены, что соответствующая проблема не критична для вашего приложения. Можно вызвать явным образом, какое исключение перехватывается. По возможности следует перехватывать как можно меньше видов исключений (в нашем примере исключение `IndexError`).

Встроенные объекты

В этом кратком обзоре мы не будем стремиться охватить ООП в целом. Впрочем, в языке Python есть немало встроенных классов.

Что такое объект

В ООП данные (состояние) и функциональность объединены. Для работы с объектами необходимо разбираться с такими понятиями, как *создание экземпляров классов* (class instantiation) — создание объектов на основе классов и *синтаксис использования точки* (dot syntax), служащий для доступа к атрибутам и методам объектов. В классе описываются атрибуты и методы, совместно используемые всеми его объектами. Его можно считать чем-то вроде чертежа автомобиля. На основе такого «чертежа» можно затем создавать экземпляры этого класса. Экземпляр класса (объект) — это конкретный автомобиль, построенный по такому чертежу.

```
>>> # Описываем класс для воображаемого описания воображаемых автомобилей
>>> class FancyCar():
...     pass
...
>>> type(FancyCar)
<class 'type'>
```



```
>>> # Создаем экземпляр воображаемого автомобиля
>>> my_car = FancyCar()
>>> type(my_car)
<class '__main__.FancyCar'>
```

Не волнуйтесь пока не счет создания собственных классов. Просто запомните, что любой объект представляет собой экземпляр какого-либо класса.

Методы и атрибуты объектов

Данные объектов хранятся в их атрибутах, представляющих собой прикрепленные к объекту или классу объектов переменные. Функциональность объектов описывается в *методе объекта* (методы, описанные для всех объектов класса) и *методе класса* (методы, относящиеся к классу и совместно используемых всеми объектами данного класса), представляющих собой связанные с объектом функции.



В документации Python прикрепленные к объекту или классу функции называются методами.

У этих функций есть доступ к атрибутам объекта, они могут модифицировать и использовать его данные. Для вызовов методов объекта или доступа к его атрибутам используется синтаксис с применением точки:

```
>>> # Описываем класс для воображаемого описания воображаемых автомобилей
>>> class FancyCar():
...     # Добавляем переменную класса
...     wheels = 4
...     # Добавляем метод
...     def driveFast(self):
...         print("Driving so fast")
...
...
...
>>> # Создаем экземпляр воображаемого автомобиля
>>> my_car = FancyCar()
>>> # Обращаемся к атрибуту класса
>>> my_car.wheels
4
>>> # Вызываем метод
>>> my_car.driveFast()
Driving so fast
>>>
```

В данном случае в нашем классе `FancyCar` описаны метод `driveFast` и трибут `wheels`. Если создать экземпляр класса `FancyCar` под названием `my_car`, можно обратиться к его трибуту и вызвать метод посредством синтаксиса использования точки.

Последовательности

Последовательности — семейство встроенных типов данных, включающее *списки* (`list`), *кортежи* (`tuple`), *диапазоны* (`range`), *строковые значения* (`string`) и *двоичные данные* (`binary`). Последовательности служат для представления упорядоченных конечных последовательностей элементов.

Операции над последовательностями

Существует множество операций, которые можно производить над любыми видами последовательностей. Мы рассмотрим некоторые наиболее распространенные операции.

С помощью операторов `in` и `not in` можно проверить, входит ли конкретный элемент в последовательность:

```
>>> 2 in [1,2,3]
True
>>> 'a' not in 'cat'
False
>>> 10 in range(12)
True
>>> 10 not in range(2, 4)
True
```

Ссылаясь на содержимое последовательности можно по индексу. Для доступа к элементу, расположенному по какому-либо индексу, используются квадратные скобки с этим индексом в качестве аргумента. Индекс первого элемента — 0, второго — 1 и так далее вплоть до номера, на единицу меньшего, чем число элементов:

```
>>> my_sequence = 'Bill Cheatham'
>>> my_sequence[0]
'B'
>>> my_sequence[2]
'l'
>>> my_sequence[12]
'm'
```

Индексацию можно выполнять и с конц последовательности, не с ее начал, указывая в качестве индекса отрицательные числа. Индекс последнего элемента — 1, предпоследнего — 2 и т. д.:

```
>>> my_sequence = "Bill Cheatham"
>>> my_sequence[-1]
'm'
>>> my_sequence[-2]
'a'
>>> my_sequence[-13]
'b'
```

Индекс элемента можно узнать с помощью метода `index`, который по умолчанию возвращает индекс первого вхождения конкретного элемента, но с помощью необязательных аргументов можно задать поддиапазон для поиска:

```
>>> my_sequence = "Bill Cheatham"
>>> my_sequence.index('C')
5
>>> my_sequence.index('a')
8
>>> my_sequence.index('a', 9, 12)
11
>>> my_sequence[11]
'a'
>>>
```

Создать новую последовательность на основе существующей можно с помощью срезов. Получить срез заданной последовательности можно ее вызовом с указанием в квадратных скобках необязательных аргументов `start`, `stop` и `step`:

```
my_sequence[start:stop:step]
```

`start` — индекс первого элемента новой последовательности, `stop` — первый индекс ее предельми, `step` — расстояние между элементами. Все эти аргументы необязательны, если они не указаны, то значениями по умолчанию. В результате выполнения этого оператора генерируется копия исходной последовательности. Значение по умолчанию для `start` — 0, для `stop` — длина последовательности, для `step` — 1. Обратите внимание на то, что если `step` не указан, то соответствующий символ также можно опустить:

```
>>> my_sequence = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> my_sequence[2:5]
['c', 'd', 'e']
```

```
>>> my_sequence[:5]
['a', 'b', 'c', 'd', 'e']
>>> my_sequence[3:]
['d', 'e', 'f', 'g']
>>>
```

Для обратной (с конца последовательности) индексации можно использовать отрицательные числа:

```
>>> my_sequence[-6:]
['b', 'c', 'd', 'e', 'f', 'g']
>>> my_sequence[3:-1]
['d', 'e', 'f']
>>>
```

Существует множество операций для получения информации о последовательностях и их содержимом. `len` возвращает длину последовательности, `min` — наименьший элемент, `max` — наибольший элемент, `count` — номер конкретного элемента в ней. Операции `min` и `max` применимы только к последовательностям, элементы которых сравнимы между собой. Помните, что эти операции работают для любых типов последовательностей:

```
>>> my_sequence = [0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 4]
>>> len(my_sequence)
12
>>> min(my_sequence)
0
>>> max(my_sequence)
4
>>> my_sequence.count(1)
3
>>>
```

Списки

Списки — один из наиболее часто используемых структур данных Python — представляют упорядоченный набор элементов произвольного типа. Нисинтаксис списка используют квадратные скобки.

Для создания пустого списка или списка на основе другого конечного итерируемого объекта (например, другой последовательности) можно использовать функцию `list()`:

```
>>> list()
[]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list("Henry Miller")
['H', 'e', 'n', 'r', 'y', ' ', 'M', 'i', 'l', 'l', 'e', 'r']
>>>
```

Чаще всего встречаются списки, созданные непосредственно с помощью квадратных скобок. Элементы в них перечисляются явно. Напомним, что элементы списка могут относиться к различным типам данных:

```
>>> empty = []
>>> empty
[]
>>> nine = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> nine
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> mixed = [0, 'a', empty, 'WheelHoss']
>>> mixed
[0, 'a', [], 'WheelHoss']
>>>
```

Самый быстрый способ добавления отдельных элементов в список — присоединение (`append`) их в его конец. Менее быстрый способ, `insert`, позволяет вставлять элементы в любое место списка. Введем следующее рассмотрение:

```
>>> pies = ['cherry', 'apple']
>>> pies
['cherry', 'apple']
>>> pies.append('rhubarb')
>>> pies
['cherry', 'apple', 'rhubarb']
>>> pies.insert(1, 'cream')
>>> pies
['cherry', 'cream', 'apple', 'rhubarb']
>>>
```

С помощью метода `extend` можно добавить содержимое одного списка в конец другого:

```
>>> pies
['cherry', 'cream', 'apple', 'rhubarb']
>>> desserts = ['cookies', 'paste']
>>> desserts
['cookies', 'paste']
>>> desserts.extend(pies)
>>> desserts
['cookies', 'paste', 'cherry', 'cream', 'apple', 'rhubarb']
>>>
```

Ниболее эффективный и часто встречающийся способ удаления последнего элемента списка, с возвратом его значения — `pop`. В этот метод можно передать

индекс для удаления и возврат элемента, расположенного по этому индексу. Этот методик работает не так эффективно, поскольку список приходится индексировать заново:

```
>>> pies
['cherry', 'cream', 'apple', 'rhubarb']
>>> pies.pop()
'rhubarb'
>>> pies
['cherry', 'cream', 'apple']
>>> pies.pop(1)
'cream'
>>> pies
['cherry', 'apple']
```

Существует также метод `remove`, удаляющий первое вхождение указанного в списке элемента:

```
>>> pies.remove('apple')
>>> pies
['cherry']
>>>
```

Одной из самых впечатляющих практических возможностей Python — *списковые включения* (list comprehensions) — позволяет использовать функциональность цикла `for` с помощью одной строки кода. Рассмотрим простой пример — возведение чисел от 0 до 9 в квадрат в цикле `for` и добавление их в конец списка:

```
>>> squares = []
>>> for i in range(10):
...     squared = i*i
...     squares.append(squared)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

Заменить этот код списковым включением можно следующим образом:

```
>>> squares = [i*i for i in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

Обратите внимание на то, что с помощью вызывается функциональность внутреннего блока кода, за которой следует оператор `for`. В списковых включениях можно использовать условные операторы для фильтрации результатов:

```
>>> squares = [i*i for i in range(10) if i%2==0]
>>> squares
[0, 4, 16, 36, 64]
>>>
```

В числе других методик работы со списками включениями — вложение и применение нескольких переменных, но чаще всего встречается более простая их форма, пока мы не увидим ее.

Строковые значения

Строковые последовательности представляют собой упорядоченные наборы символов, заключенные в кавычки. В строках Python 3 по умолчанию используется кодировка *UTF-8*.

Создать строковые значения можно либо с помощью метода-конструктора для строк `str()`, либо просто заключить текст в кавычки:

```
>>> str()
''
>>> "some new string!"
'some new string!'
>>> 'or with single quotes'
'or with single quotes'
```

С помощью конструктора строк можно создать строковые значения на основе других объектов:

```
>>> my_list = list()
>>> str(my_list)
'[]'
```

Заключить текст в тройные кавычки, можно создать многострочные строковые значения:

```
>>> multi_line = """This is a
... multi-line string,
... which includes linebreaks.
... """
>>> print(multi_line)
This is a
multi-line string,
which includes linebreaks.
>>>
```

Помимо общих для всех последовательностей методов, у строковых значений есть и немало собственных.

В начале или в конце пользовательского текста нередко встречаются пробелы. А введенную пользователем строку "да" желательно обрезать также, как и "да". В этом случае в строках Python есть метод `strip`, который возвращает строку без пробелов в начале или в конце. Существуют также методы для удаления пробелов только с правой или левой стороны строки:

```
>>> input = " I want more "
>>> input.strip()
'I want more'
>>> input.rstrip()
' I want more'
>>> input.lstrip()
'I want more '
```

В то же время существуют методы `ljust` и `rjust` для дополнения строк символами. По умолчанию они дополняют строковые значения пробелами, но могут и принимать в виде аргумента символ для дополнения:

```
>>> output = 'Barry'
>>> output.ljust(10)
'Barry      '
>>> output.rjust(10, '*')
'*****Barry'
```

Иногда бывает нужно разбить строковое значение на список подстрок, скажем превратить предложение в список слов или строку слов, разделенных запятыми. Метод `split` разбивает строку, преобразуя ее в список строк. По умолчанию в качестве токена, на основе которого выполняется разбиение, используется пробел. Для разбиения по другому символу можно добавить необязательный аргумент:

```
>>> text = "Mary had a little lamb"
>>> text.split()
['Mary', 'had', 'a', 'little', 'lamb']
>>> url = "gt.motomomo.io/v2/api/asset/143"
>>> url.split('/')
['gt.motomomo.io', 'v2', 'api', 'asset', '143']
```

Можно легко создать новое строковое значение из последовательности строковых значений и объединить (`join`) их в единое целое. В следующем примере посередине списка других строк вставляется строка-разделитель:

```
>>> items = ['cow', 'milk', 'bread', 'butter']
>>> " and ".join(items)
'cow and milk and bread and butter'
```


Часто бывает нужно изменить регистр текста: выровнять для сравнения или поменять для последующего чтения пользователем. В Python есть несколько методов¹, упрощающих эту задачу:

```
>>> name = "bill monroe"
>>> name.capitalize()
'Bill monroe'
>>> name.upper()
'BILL MONROE'
>>> name.title()
'Bill Monroe'
>>> name.swapcase()
'BILL MONROE'
>>> name = "BILL MONROE"
>>> name.lower()
'bill monroe'
```

Python также предоставляет методы, упрощающие анализ содержимого строковых значений. Тких методов довольно много, начиная от проверки регистра текста до выяснения того, содержит ли он числовое значение. Вот лишь несколько из наиболее часто используемых методов:

```
>>> "William".startswith('W')
True
>>> "William".startswith('Bill')
False
>>> "Molly".endswith('olly')
True
>>> "abc123".isalnum()
True
>>> "abc123".isalpha()
False
>>> "abc".isalnum()
True
>>> "123".isnumeric()
True
>>> "Sandy".istitle()
True
>>> "Sandy".islower()
False
>>> "SANDY".isupper()
True
```

Можно вставить текст в строку во время выполнения, упрощая ее формат. Программист может при этом использовать в строковых значениях переменных или

¹ Работают и с кириллицей. — *Примеч. пер.*

другое вычисляемое содержимое. Подобный подход применяется к к при созд нии текст , предн зн ченного для пользо телей, т к и при з писи прог рмных журн лов.

Более ст рым в ри нтом форм тиров ния строк Python обяз н функции `printf` язык C. Для вст вки отформ тиров нных зн чений в строку можно использо в ть `%` — опер тор деления по модулю в форме строка `% значения`, где значения может быть отдельным зн чением или кортежем из нескольких. А строка должн содерж ть специфик торы преобр зов ния для всех зн чений. Специфик тор преобр зов ния в миним льной форме н чин ется с `%`, з которым следует символ, отр ж ющий тип вст вляемого зн чения:

```
>>> "%s + %s = %s" % (1, 2, "Three")
'1 + 2 = Three'
>>>
```

Дополнительные ргументы форм т включ ют т же специфик торы преобр зов ния. Н пример, можно з д в ть количество цифр после з пятой:

```
>>> "%.3f" % 1.234567
'1.235'
```

Т кой мех низм форм тиров ния строковых зн чений преобл д л в Python н протяжении многих лет, и его до сих пор можно встретить в ст ром коде. У этого подход есть определенные преимуществ , н пример единый с другими язык ми прог рммирования синт ксис. Но есть и подводные к мни. В ч стности, из-з хр нения ргументов в последов тельности нередко возник ют ошибки ото бр жения объектов `tuple` и `dict`. Мы рекомендуем взять н вооружение более современные в ри нты форм тиров ния, н пример метод `format`, ш блонизиров нные строки и f-строки, во избеж ние подобных ошибок и для упрощения код , т же повышения его удобочит емости.

В Python 3 появился новый способ форм тиров ния строк — с помощью име ющегося у них метод `format`. З тем он был бэкпортиров н в Python 2. В этой специфик ции з меняемые поля в строке з д ются с помощью фигурных скобок, не символов деления по модулю, к к при ст ром стиле форм тиров ния. Вст вляемые зн чения ук зыв ются в виде ргументов метод `format`. Порядок ргументов определяет порядок подст новок в целевой строке:

```
>>> '{} comes before {}'.format('first', 'second')
'first comes before second'
>>>
```

Для вст вки зн чений в порядке, отличном от списк ргументов, можно ук зыв ть числовые индексы в фигурных скобк х. Можно т же повторять одно зн чение несколько р з, ук зыв я одно зн чение индекс в нескольких подст новочных полях:

К и в строке `format`, в f-строке спецификация форматов указывается внутри фигурных скобок после выражения значения и начинаются с двоеточия:

```
>>> count = 43
>>> f"|{count:5d}"
'| 43'
```

Выражение значения может включать вложенные выражения, также ссылаться на переменные и выражения, содержащиеся в конструкции родительского выражения:

```
>>> padding = 10
>>> f"|{count:{padding}d}"
'|          43'
```



Мы настоятельно рекомендуем при форматировании строк использовать f-строки. Они сочетают возможности мини-языка спецификаций с простым и интуитивно понятным синтаксисом.

Цель шablонизированных строк — создание простого механизма подстановки для строк, в частности для создания подобия логотипа, где необходимы простые подстановки слов. В качестве подстановочного символа в них используется `$` с необязательными фигурными скобками вокруг него. Вставляемое значение задается символами, непосредственно следующими за `$`. При выполнении метода `substitute` шablонизированной строки они используются для присвоения значений.



В ходе работы с Python всегда доступны встроенные типы и функции, но для доступа к более широкой функциональности экосистемы Python необходимо совершить оператор `import`, который позволяет добавлять в среду функциональность из стандартной библиотеки Python или сторонних сервисов. Можно выбирать импортируемые чистые пакеты с помощью ключевого слова `from`:

```
>>> from string import Template
>>> greeting = Template("$hello Mark Anthony")
>>> greeting.substitute(hello="Bonjour")
'Bonjour Mark Anthony'
>>> greeting.substitute(hello="Zdravstvuyte")
'Zdravstvuyte Mark Anthony'
>>> greeting.substitute(hello="Nin hão")
'Nin hão Mark Anthony'
```

Ассоциативные массивы

Ассоциативные массивы — наиболее часто используемые встроенные классы Python, не считая строк и списков. Ассоциативный массив задает соответствие ключей значениям. Поиск конкретного значения по ключу производится очень быстро и эффективно. Ключи могут быть строки, числа, пользовательские объекты и любые другие неизменяемые типы.



Изменяемый (mutable) объект — тот, содержимое которого может меняться после создания. Основной пример — списки: содержимое списка может меняться без изменения его самого. Строковые значения — неизменяемые. Всякий раз, когда меняется содержимое уже существующей строки, создается новая.

Ассоциативные массивы описываются в виде заглавного в фигурные скобки набор пар «ключ/значение», разделяемых запятыми. Пар «ключ/значение» состоит из ключа, двоеточия (:) и значения.

Создать экземпляр ассоциативного массива можно с помощью конструктора `dict()`. Если никаких аргументов при его вызове не указать, он создаст пустой ассоциативный массив, но может также принимать в качестве аргументов последовательность пар «ключ/значение»:

```
>>> map = dict()
>>> type(map)
<class 'dict'>
>>> map
{}
>>> kv_list = [['key-1', 'value-1'], ['key-2', 'value-2']]
>>> dict(kv_list)
{'key-1': 'value-1', 'key-2': 'value-2'}
```

Можно также создать ассоциативный массив непосредственно, с помощью фигурных скобок:

```
>>> map = {'key-1': 'value-1', 'key-2': 'value-2'}
>>> map
{'key-1': 'value-1', 'key-2': 'value-2'}
```

Обратиться к соответствующему ключу/значению можно с помощью синтаксиса квадратных скобок:

```
>>> map['key-1']
'value-1'
>>> map['key-2']
'value-2'
```

Ан логичный синтаксис можно использовать для задания значений. Если соответствующего ключа в данном ассоциативном массиве нет, он добавляется в качестве новой записи. Если же есть, значение меняется на новое:

```
>>> map
{'key-1': 'value-1', 'key-2': 'value-2'}
>>> map['key-3'] = 'value-3'
>>> map
{'key-1': 'value-1', 'key-2': 'value-2', 'key-3': 'value-3'}
>>> map['key-1'] = 13
>>> map
{'key-1': 13, 'key-2': 'value-2', 'key-3': 'value-3'}
```

Если попытаться обратиться к не существующему в ассоциативном массиве ключу, будет сгенерировано исключение `KeyError`:

```
>>> map['key-4']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    map['key-4']
KeyError: 'key-4'
```

Проверить, есть ли в данном ассоциативном массиве конкретный ключ, можно с помощью синтаксиса `in`, который мы уже встречали, работая с последовательностями. В случае ассоциативных массивов он проверяет наличие ключей:

```
>>> if 'key-4' in map:
...     print(map['key-4'])
... else:
...     print('key-4 not there')
...
...
key-4 not there
```

Более интуитивно понятное решение — использовать метод `get()`. Если конкретный ключ в ассоциативном массиве не задан, этот метод вернет указанное значение по умолчанию, если оно не указано — вернет `None`:

```
>>> map.get('key-4', 'default-value')
'default-value'
```

Для удаления пары «ключ/значение» из ассоциативного массива служит метод `del`:

```
>>> del(map['key-1'])
>>> map
{'key-2': 'value-2', 'key-3': 'value-3'}
```

Метод `keys()` возвращает объект `dict_keys`, содержащий все ключи ассоциативного массива. Метод `values()` возвращает объект `dict_values`, содержащий все значения

соединительного массива, метод `items()` — пары «ключ/значение». Последний метод удобен для обхода содержимого соединительного массива в цикле:

```
>>> map.keys()
dict_keys(['key-1', 'key-2'])
>>> map.values()
dict_values(['value-1', 'value-2'])
>>> for key, value in map.items():
...     print(f"{key}: {value}")
...
...
key-1: value-1
key-2: value-2
```

Как и списковые включения, словесные включения представляют собой однострочные операторы, возвращающие соединительный массив путем обхода последовательности в цикле:

```
>>> letters = 'abcde'
>>> # Отображение отдельных букв в их аналоги в верхнем регистре
>>> cap_map = {x: x.upper() for x in letters}
>>> cap_map['b']
'B'
```

Функции

Вы уже встречали некоторые из встроенных функций языка Python. Займемся теперь написанием собственных функций. Напомним, что *функция* — это механизм инкапсуляции блока кода, позволяющий воспроизводить поведение этого блока в различных местах без дублирования с самого кода. Функции улучшают организацию кода, его тестируемость, повышают удобство сопровождения и удобочитаемость.

Синтаксис функции

Первая строка описания функции начинается с ключевого слова `def`, за которым следуют имя функции, заключенные в скобки ее параметры, за тем `:`. Остальная часть функции представляет собой сдвинутый вправо блок кода:

```
def <Имя функции>(<Параметры>):
    <Блок кода>
```

Если в сдвинутом посредством отступов блоке кода содержится с помощью многострочного синтаксиса какое-либо строковое значение, оно играет роль документации, с помощью которой можно описать, что делает функция, смысл параметров и возвращаемое значение. Эти `docstring` очень ценны в качестве средств

общения с будущими пользователями кода. Кроме того, многие программы и сервисы умеют генерировать на их основе документацию. Указание docstring для своих функций считается рекомендуемой практикой:

```
>>> def my_function():
...     '''Строка документации
...
...     Должна описывать, что делает функция,
...     что означают ее параметры и какое значение она возвращает
...     '''
```

Аргументы функции приводятся в скобках, следующих за ее именем, и могут быть позиционными или ключевыми. Значения позиционным аргументам присваиваются в соответствии с их порядком:

```
>>> def positioned(first, second):
...     """Значения присваиваются по порядку"""
...     print(f"first: {first}")
...     print(f"second: {second}")
...
...
>>> positioned(1, 2)
first: 1
second: 2
>>>
```

Каждому ключевому аргументу присваивается значение по умолчанию:

```
>>> def keywords(first=1, second=2):
...     '''Присваиваем значения по умолчанию'''
...     print(f"first: {first}")
...     print(f"second: {second}")
...
...
>>>
```

Значения по умолчанию используются, если при вызове функции не было передано никаких фактических значений. Во время выполнения функции ключевые параметры можно вызывать по имени, при этом их порядок не важен:

```
>>> keywords()
first: 0
second: 2
>>> keywords(3,4)
first: 3
second: 4
>>> keywords(second='one', first='two')
first: two
second: one
```

При использовании ключевых параметров все описанные после них параметры также должны быть ключевыми. Все функции возвращают какое-либо значение.

Для указания этого значения применяется ключевое слово `return`. Если оператор `return` в теле функции не описан, он возвращает `None`:

```
>>> def no_return():
...     '''Оператор return не описан'''
...     pass
...
>>> result = no_return()
>>> print(result)
None
>>> def return_one():
...     '''Возвращает 1'''
...     return 1
...
>>> result = return_one()
>>> print(result)
1
```

Функции как объекты

Функции являются объектами. Их можно передавать и хранить в структурах данных. Можно описать две функции, поместить их в список, затем пройти в цикле по этому списку для их вызовов:

```
>>> def double(input):
...     '''Удваивает input'''
...     return input*2
...
>>> double
<function double at 0x107d34ae8>
>>> type(double)
<class 'function'>
>>> def triple(input):
...     '''Утраивает input'''
...     return input*3
...
>>> functions = [double, triple]
>>> for function in functions:
...     print(function(3))
...
...
6
9
```

Анонимные функции

Если требуется очень маленькая функция, можно с помощью ключевого слова `lambda` создать безымянную (анонимную) функцию. В общем случае следует ограничить их применение ситуациями, в которых одна функция ожидает

в качестве аргумента другую функцию. В данном примере мы получили на входе список списков и сортируем его. По умолчанию механизм сортировки выполняет сортировку по первым элементам подсписков:

```
>>> items = [[0, 'a', 2], [5, 'b', 0], [2, 'c', 1]]
>>> sorted(items)
[[0, 'a', 2], [2, 'c', 1], [5, 'b', 0]]
```

Для сортировки не по первым элементам, по чему-то еще можно описать метод, возвращающий, например, второй элемент, и передать его в параметр `key` функции сортировки:

```
>>> def second(item):
...     '''Возвращает второй элемент'''
...     return item[1]
...
>>> sorted(items, key=second)
[[0, 'a', 2], [5, 'b', 0], [2, 'c', 1]]
```

Благодаря ключевому слову `lambda` можно сделать то же самое без полноценного описания функции. Синтаксис лямбда-выражений включает ключевое слово `lambda`, за которым следуют имя параметра, двоеточие и возвращаемое значение:

```
lambda <Параметр>: <Возвращаемое значение>
```

Сортировка с помощью лямбда-выражений, в которой сначала используется второй элемент списка, а потом третий:

```
>>> sorted(items, key=lambda item: item[1])
[[0, 'a', 2], [5, 'b', 0], [2, 'c', 1]]
>>> sorted(items, key=lambda item: item[2])
[[5, 'b', 0], [2, 'c', 1], [0, 'a', 2]]
```

Будьте осторожны при использовании лямбда-выражений вместо обычных функций: иногда получается плохо документированный и сложный для прочтения код.

Регулярные выражения

Потребность в сопоставлении шаблонов в строковых значениях возникает то и дело для поиска идентификаторов в файлах журналов или ключевых слов во вводимом пользователем тексте или во множестве других сценариев. Ранее вы уже видели простой вариант сопоставления шаблонов для последовательностей с помощью операции `in`, а также методов `.endswith` и `.startswith`. Для более сложных вариантов сопоставления шаблонов необходимы более

совершенные инструменты. Решение этой задачи — регулярные выражения. Регулярные выражения представляют собой строки символов, описывающие поисковые шаблоны. Пакет `re` Python предоставляет возможность проведения операций с регулярными выражениями, а также логичных существующим в языке Perl. В модуле `re` специальные символы отделяются друг от друга обратными косыми чертами. Во избежание путаницы с обычными экранированными последовательностями символов при описании шаблонов регулярных выражений рекомендуется применять неформатированные строки. Их обозначают символом `r` перед первым знаком кавычки.



В строках языка Python используется несколько экранированных последовательностей. В числе наиболее распространенных — перевод строки `\n` и табуляция `\t`.

Поиск

Пусть в виде текста дан список скрытых копий из сообщения электронной почты и нужно выяснить, кто есть кто в нем:

```
In [1]: cc_list = '''Ezra Koenig <ekoenig@vpwk.com>,
...: Rostam Batmanglij <rostam@vpwk.com>,
...: Chris Tomson <ctomson@vpwk.com>,
...: Bobbi Baio <bbaio@vpwk.com>'''
```

Чтобы просто узнать, присутствует ли какое-то имя в этом тексте, можно воспользоваться синтаксисом проверки вхождения в последовательность `in`:

```
In [2]: 'Rostam' in cc_list
Out[2]: True
```

Аналогичный результат можно получить с помощью функции `re.search`, возвращающей объект `re.Match` при обнаружении совпадения:

```
In [3]: import re

In [4]: re.search(r'Rostam', cc_list)
Out[4]: <re.Match object; span=(32, 38), match='Rostam'>
```

Ее можно использовать в качестве условия для проверки на вхождение:

```
>>> if re.search(r'Rostam', cc_list):
...     print('Found Rostam')
...
...
Found Rostam
```

Наборы символов

Пок возможности модуля `re` ничем не превыш ли возможностей опер тор `in`. Но что, если вы хотите н йти в тексте кого-либо, но не помните точно имя — *Bobbi* или *Robby*?

Бл год ря регулярным выр жениям можно ук зыв ть группы символов, к ждый из которых может встреч ться в конкретном месте. Они н зыв ются н бор ми символов (character sets). Символы, которые должны проверяться н совп де ние, з ключ ются в кв др тные скобки в опис нии регулярного выр жения. Н пример, можно произвести поиск по *B* или *R*, з которым следует *obb*, д лее идет *i* или *y*:

```
In [5]: re.search(r'[RB]obb[iy]', cc_list)
Out[5]: <re.Match object; span=(101, 106), match='Bobbi'>
```

В н бор х символов можно ук зыв ть отдельные символы, р зделенные з - пятыми, или использов ть ди п зоны. Ди п зон *A–Z* включ ет все символы нглийского лф вит в верхнем регистре, ди п зон *0–9* включ ет цифры от 0 до 9:

```
In [6]: re.search(r'Chr[a-z][a-z]', cc_list)
Out [6]: <re.Match object; span=(69, 74), match='Chris'>
```

Символ `+` после элемент в регулярном выр жении соответствует одному или нескольким экземплярам этого элемент . Число в фигурных скобк х з д ет точное количество символов:

```
In [7]: re.search(r'[A-Za-z]+', cc_list)
Out [7]: <re.Match object; span=(0, 4), match='Ezra'>
In [8]: re.search(r'[A-Za-z]{6}', cc_list)
Out [8]: <re.Match object; span=(5, 11), match='Koenig'>
```

Можно сформиров ть ш блон для сопост вления с помощью сочет ния н боров символов и прочих символов для примитивного поиск дресов электронной почты. Символ `«.»` — джокерный, он соответствует произвольному символу. Для поиск н стоящего символ `«.»` необходимо экр ниров ть его с помощью обр тной косой черты:

```
In [9]: re.search(r'[A-Za-z]+@[a-z]+\.[a-z]+', cc_list)
Out[9]: <re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
```

Этот пример просто демонстрирует возможности н боров символов, но вовсе не отр ж ет всей сложности регулярных выр жений для дресов электронной почты, используемых в промышленной эксплу т ции.

Классы символов

Помимо наборов символов, в модуле `re` Python есть еще классы символов, представляющие собой уже готовые наборы символов. В числе наиболее распространенных — `\w`, эквивалентный `[a-zA-Z0-9_]`, и `\d`, эквивалентный `[0-9]`. Для сопоставления с несколькими символами можно использовать модификатор `+`:

```
>>> re.search(r'\w+', cc_list)
<re.Match object; span=(0, 4), match='Ezra'>
```

Так что можно заменить наш примитивный поисковый шаблон для адресов электронной почты на использующий `\w`:

```
>>> re.search(r'\w+@\w+\.\w+', cc_list)
<re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
```

Группы

С помощью скобок можно описывать группы в шаблоне для сопоставления, на которые можно ссылаться через объект шаблона. Они нумеруются в порядке вхождения в шаблон, причем нулевая группа соответствует шаблону в целом:

```
>>> re.search(r'(\w+)\@(\w+)\.(\w+)', cc_list)
<re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
>>> matched = re.search(r'(\w+)\@(\w+)\.(\w+)', cc_list)
>>> matched.group(0)
'ekoenig@vpwk.com'
>>> matched.group(1)
'ekoenig'
>>> matched.group(2)
'vpwk'
>>> matched.group(3)
'com'
```

Поименованные группы

Группам можно присвоить названия путем добавления `?P<Название>` в их описание. В этом случае можно будет ссылаться на группы по названию вместо номера:

```
>>> matched = re.search(r'?P<name>\w+)\@(?P<SLD>\w+)\. (?P<TLD>\w+)', cc_list)
>>> matched.group('name')
'ekoenig'
```

```
>>> print(f'''name: {matched.group("name")}
... Secondary Level Domain: {matched.group("SLD")}
... Top Level Domain: {matched.group("TLD")}''')
name: ekoenig
Secondary Level Domain: vpwk
Top Level Domain: com
```

Найти все

До сих пор мы демонстрировали, как вернуть первое найденное вхождение. Но можно воспользоваться методом `findall`, чтобы вернуть все найденные совпадения в виде списка строковых значений:

```
>>> matched = re.findall(r'\w+@\w+\.\w+', cc_list)
>>> matched
['ekoenig@vpwk.com', 'roстам@vpwk.com', 'ctomson@vpwk.com', 'cbaio@vpwk.com']
>>> matched = re.findall(r'(\w+)\@(\w+)\.(\w+)', cc_list)
>>> matched
[('ekoenig', 'vpwk', 'com'), ('roстам', 'vpwk', 'com'),
 ('ctomson', 'vpwk', 'com'), ('cbaio', 'vpwk', 'com')]
>>> names = [x[0] for x in matched]
>>> names
['ekoenig', 'roстам', 'ctomson', 'cbaio']
```

Поисковый итератор

При работе с большими текстами, например журналами, удобно обходить текст по частям. С помощью метода `finditer` можно сгенерировать объект *итератор*, который обходит текст до момента обнаружения первого совпадения, затем прекращает работу. Передвигая его функции `next`, можно получить текущее совпадение и продолжить обходку до момента обнаружения следующего совпадения. Таким образом можно обходить любое вхождение по отдельности и не выделять ресурсы на обходку входного текста целиком:

```
>>> matched = re.finditer(r'\w+@\w+\.\w+', cc_list)
>>> matched
<callable_iterator object at 0x108e68748>
>>> next(matched)
<re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
>>> next(matched)
<re.Match object; span=(51, 66), match='roстам@vpwk.com'>
>>> next(matched)
<re.Match object; span=(83, 99), match='ctomson@vpwk.com'>
```

Объект-итер `matched` можно использовать и в цикле `for`:

```
>>> matched = re.finditer("(?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)", cc_list)
>>> for m in matched:
...     print(m.groupdict())
...
...
{'name': 'ekoenig', 'SLD': 'vpwk', 'TLD': 'com'}
{'name': 'rostan', 'SLD': 'vpwk', 'TLD': 'com'}
{'name': 'ctomson', 'SLD': 'vpwk', 'TLD': 'com'}
{'name': 'cbaio', 'SLD': 'vpwk', 'TLD': 'com'}
```

Подстановка

Помимо поиска и сопоставления с шаблоном, регулярные выражения можно использовать для замены части или всего строкового значения:

```
>>> re.sub("\d", "#", "The passcode you entered was 09876")
'The passcode you entered was #####'
>>> users = re.sub("(?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)",
...                 "\g<TLD>.\g<SLD>.\g<name>", cc_list)
>>> print(users)
Ezra Koenig <com.vpwk.ekoenig>,
Rostam Batmanglij <com.vpwk.rostan>,
Chris Tomson <com.vpwk.ctomson>,
Chris Baio <com.vpwk.cbaio>
```

Компиляция

Во всех приведенных до сих пор примерах непосредственно вызывались методы модуля `re`. Во многих случаях это допустимо, но при необходимости многократного сопоставления с одним и тем же шаблоном можно значительно повысить производительность за счет компиляции регулярного выражения в объект, который затем можно использовать для сопоставления с шаблоном без перекомпиляции:

```
>>> regex = re.compile(r'\w+\@ \w+ \. \w+')
>>> regex.search(cc_list)
<re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
```

Возможности регулярных выражений не только превосходят продемонстрированное ранее. Их использованию посвящено множество книг, но к большинству простых сценариев применения вы уже готовы.

Отложенное вычисление

Идея *отложенного вычисления* (lazy evaluation) заключается в том, что иногда, особенно в роботях с большими объемами данных, не имеет смысла обрабатывать все данные перед использованием результатов. Вы уже не забудете это в примере типа `range`, где объем заложенной памяти не менялся даже в случае миллионов, соответствующих большим группам чисел.

Генераторы

Генераторы можно использовать подобно объектам `range`. Они выполняют операции над данными по частям, по мере требования, переходя в состояние до следующего вызова. Это значит, что можно хранить данные, необходимые для вычисления результатов, обращаясь к ним при каждом вызове генератора.

При написании функции-генератора необходимо использовать ключевое слово `yield` вместо оператора `return`. При каждом вызове генератор возвращает значение в `yield` значение, после чего переходит в состояние до следующего вызова. Напишем генератор-счетчик, просто возвращающий последовательные числа:

```
>>> def count():
...     n = 0
...     while True:
...         n += 1
...         yield n
...
...
>>> counter = count()
>>> counter
<generator object count at 0x10e8509a8>
>>> next(counter)
1
>>> next(counter)
2
>>> next(counter)
3
```

Обратите внимание на то, что генератор сохраняет состояние, поэтому переменная при каждом вызове генератора возвращает обновленное значение. Реализуем генератор для чисел Фибоначчи:

```
>>> def fib():
...     first = 0
...     last = 1
...     while True:
...         first, last = last, first + last
```



```

...         yield first
...
>>> f = fib()
>>> next(f)
1
>>> next(f)
1
>>> next(f)
2
>>> next(f)
3

```

Можно также использовать генераторы в циклах `for`:

```

>>> f = fib()
>>> for x in f:
...     print(x)
...     if x > 12:
...         break
...
1
1
2
3
5
8
13

```

Генераторные включения

Для создания генераторов в одну строку код можно использовать генераторные включения. Их синтаксис логичен списковым включениям, только вместо квадратных скобок применяются круглые:

```

>>> list_o_nums = [x for x in range(100)]
>>> gen_o_nums = (x for x in range(100))
>>> list_o_nums
[0, 1, 2, 3, ... 97, 98, 99]
>>> gen_o_nums
<generator object <genexpr> at 0x10ea14408>

```

Дже н т ком м леньком примере с помощью метод `sys.getsizeof`, возвращающего размер объекта в байтах, можно заметить разницу в объемах используемой оперативной памяти:

```

>>> import sys
>>> sys.getsizeof(list_o_nums)
912
>>> sys.getsizeof(gen_o_nums)
120

```

Дополнительные возможности IPython

В начале главы вы уже видели некоторые возможности IPython. Теперь посмотрим на более продвинутые его возможности, на пример выполнение инструкций командной оболочки внутри интерпретатора IPython и использование «магических» функций.

Выполнение инструкций командной оболочки Unix с помощью IPython

IPython можно использовать для выполнения инструкций командной оболочки. Это один из самых веских доводов в пользу выполнения операций DevOps в командной оболочке IPython. Рассмотрим очень простой пример, в котором перед командой `ls` используется символ `!`, по которому IPython распознает инструкции командной оболочки:

```
In [3]: var_ls = !ls -l
In [4]: type(var_ls)
Out[4]: IPython.utils.text.SList
```

Выводимые этой командой результаты присваиваются переменной `var_ls`. Python тип `IPython.utils.text.SList`. Тип `SList` преобразует обычную инструкцию командной оболочки в объект с тремя основными методами: `fields`, `grep` и `sort`. Вот пример в действии с командой Unix `df`. Метод `sort` может распознать пробелы из этой команды Unix и отсортировать третий столбец по размеру¹:

```
In [6]: df = !df
In [7]: df.sort(3, nums = True)
```

Далее взглянем на `SList` и `.grep`. Вот пример поиска команд, установленных в каталоге `/usr/bin`, в названии которых входит `kill`:

```
In [10]: ls = !ls -l /usr/bin
In [11]: ls.grep("kill")
Out[11]:
['-rwxr-xr-x    1 root   wheel           1621 Aug 20  2018 kill.d',
```

¹ Не исключено, что вы столкнетесь с проблемой кодировок, нередкой при вызове консольных команд операционной системы из блокнотов IPython. Решить ее можно, на пример, установкой кодировки командной оболочки с помощью команды `!chcp 65001` или кодировки Python. — *Примеч. пер.*

```
'-rwxr-xr-x  1 root   wheel   23984 Mar 20 23:10 killall',
'-rwxr-xr-x  1 root   wheel   30512 Mar 20 23:10 pkill']
```

Из всего этого можно сделать вывод, что IPython — идеальный инструмент для всяких экспериментов с мультимедийными сценами командной оболочки.

«Магические» команды IPython

Если вы привыкнете работать с Python, то должны научиться использовать встроенные «магические» команды. По сути, они представляют собой сокращенные формы записи, заключающие в себе огромный потенциал. Перед «магическими» командами ставятся символы `%`. Вот пример написания в Python встроившейся команды `bash`. Конечно, это всего лишь одна команда, но точно так же можно написать и целый скрипт `bash`:

```
In [13]: %%bash
...: uname -a
...:
...:
Darwin nogibjj.local 18.5.0 Darwin Kernel Version 18.5.0: Mon Mar ...
```

Очень интересна команда `%writefile`, позволяющая писать и тестировать скрипты Python или `bash` прямо во время работы и выполнять их с помощью IPython:

```
In [16]: %%writefile print_time.py
...: #!/usr/bin/env python
...: import datetime
...: print(datetime.datetime.now().time())
...:
...:
...:
Writing print_time.py
```

```
In [17]: cat print_time.py
#!/usr/bin/env python
import datetime
print(datetime.datetime.now().time())
```

```
In [18]: !python print_time.py
19:06:00.594914
```

Еще одна удобная команда, `%who`, демонстрирует загруженные в память интерактивные переменные. Она очень полезна при длительной работе в терминале:

```
In [20]: %who
df      ls      var_ls
```

Упражнения

- Напишите функцию Python, принимающую значение в качестве аргумента и выводящую его в консоль.
- Напишите функцию Python, принимающую строку в качестве аргумента и выводящую в консоль информацию о ее регистре.
- Напишите списковое включение для получения списка букв слов *smogtether* в верхнем регистре.
- Напишите генератор, попеременно возвращающий слова «Четное» и «Нечетное».

Автоматизация работы с файлами и файловой системой

Одни из самых замечательных возможностей Python — операции с текстом и файлами. В мире DevOps постоянно приходится выполнять синтаксический разбор, поиск и изменение текста в файлах, идет ли речь о поиске в журналах приложения или прострелении файлов конфигурации. Файлы — это способ сохранения состояния данных, кода и конфигурации, именно с их помощью можно просмотреть историю журналов и управлять настройками. В Python можно создавать, читать и менять файлы и текст с помощью кода, и эти возможности вы будете использовать все время. Автоматизация этих задач — один из спектров современного DevOps, который отличается от традиционного системного администрирования. Вместо того чтобы хранить набор инструкций и следовать им самостоятельно, можно просто написать код, что снизит вероятность пропустить какие-либо шаги или перепутать их очередность. Уверенность в выполнении определенной последовательности шагов при запуске системы повысит понятность и надежность процесса.

Чтение и запись файлов

Чтобы открыть файловый объект для чтения и записи, можно воспользоваться функцией `open`. Он принимает в качестве аргумента: путь к файлу и режим (по умолчанию режим чтения). Режим позволяет указать, помимо прочего, открываемый ли файл для чтения или записи, содержит ли он текстовые или двоичные данные. Для чтения содержимого текстового файла подходит режим `r`. У файлового объекта есть метод `read`, возвращающий содержимое файла в виде строкового значения:

```
In [1]: file_path = 'bookofdreams.txt'
In [2]: open_file = open(file_path, 'r')
```

```
In [3]: text = open_file.read()
```

```
In [4]: len(text)
```

```
Out[4]: 476909
```

```
In [5]: text[56]
```

```
Out[5]: 's'
```

```
In [6]: open_file
```

```
Out[6]: <_io.TextIOWrapper name='bookofdreams.txt' mode='r' encoding='UTF-8'>
```

```
In [7]: open_file.close()
```



По завершении работы с файлом желательнее его закрыть. Python закрыет файлы при выходе из области видимости, но до тех пор они остаются ресурсами и могут оказаться недоступными для других процессов.

Можно также читать файлы с помощью метода `readlines`: он читает файл и разбивает его содержимое по символам перевода строк. Он возвращает список строковых значений, каждое из которых содержит одну строку исходного текста:

```
In [8]: open_file = open(file_path, 'r')
```

```
In [9]: text = open_file.readlines()
```

```
In [10]: len(text)
```

```
Out[10]: 8796
```

```
In [11]: text[100]
```

```
Out[11]: 'science, when it admits the possibility of occasional
hallucinations\n'
```

```
In [12]: open_file.close()
```

Файлы удобно открывать с помощью операторов `with`. При этом не требуется закрывать их явно. Python сам закроет файл и освободит выделенные под него ресурсы в конце отформатированного с помощью отступов блока:

```
In [13]: with open(file_path, 'r') as open_file:
```

```
...:     text = open_file.readlines()
```

```
...:
```

```
In [14]: text[101]
```

```
Out[14]: 'in the sane and healthy, also admits, of course, the existence of\n'
```

```
In [15]: open_file.closed
```

```
Out[15]: True
```

В различных операционных системах — разные экранированные символы для обозначения конца строки: в Unix-системах — `\n`, в Windows — `\r\n`. Python преобразует их в `\n` при открытии файла в текстовом режиме. Если открывать файл

текст двоичный файл, и пример изображения `.jpeg`, подобное преобразование, вероятно, повредит данные. Однако для чтения двоичных файлов можно добиться `b` в модификатор режим:

```
In [15]: file_path = 'bookofdreamsghost00lang.pdf'
In [16]: with open(file_path, 'rb') as open_file:
...:     btext = open_file.read()
...:

In [17]: btext[0]
Out[17]: 37

In [18]: btext[:25]
Out[18]: b'%PDF-1.5\n%\xec\xf5\xf2\xe1\xef\xe3\xf5\xed\xe5\xee\xf4\n18'
```

Добавление этого модификатора позволяет открыть файл без какого-либо преобразования символов конца строки.

Для записи в файл необходимо использовать режим записи, предусмотренный аргументом `w`. Утилита `direnv` позволяет автоматически строить среду запуска. В файле `.envrc` можно описать переменные среды и среду выполнения приложения; `direnv` настроит все это на его основе при входе в каталог с файлом. Например, в Python можно задать в подобном файле переменную среды `STAGE` равной `PROD` и `TABLE_ID` равным `token-storage-1234`, воспользовавшись функцией `open` с флагом `write`:

```
In [19]: text = '''export STAGE=PROD
...: export TABLE_ID=token-storage-1234'''

In [20]: with open('.envrc', 'w') as opened_file:
...:     opened_file.write(text)
...:

In [21]: !cat .envrc
export STAGE=PROD
export TABLE_ID=token-storage-1234
```



Учтите, что метод `write` команды `pathlib` перезаписывает уже существующие файлы.

Функция `open` создаст файл, если он еще не существует, и перезапишет в противном случае. Чтобы сохранить существующее содержимое файла и просто добавить данные в его конец, используйте флаги `a` и `a+`. Он позволяет добавить новый текст в конец файла, сохраняя исходное содержимое. Запись нетекстового содержимого, и пример содержимого файла `.jpeg`, вероятно, повредит его, если

з действий `fl` и `al`. Дело в том, что Python преобразует символы конца строки в ориентированные на конкретную платформу при записи текстовых данных. Для безопасной записи двоичных данных можно использовать `fl` и `wb` или `ab`.

В главе 3 комманд `pathlib` обсуждается подробно. В числе ее возможностей — две удобные функции для чтения и записи файлов, все операции с файловым объектом комманд `pathlib` производит «з кулис ми». Вот так можно прочитать текст из файла:

```
In [35]: import pathlib
```

```
In [36]: path = pathlib.Path(
        "/Users/kbehrman/projects/autoscaler/check_pending.py")
```

```
In [37]: path.read_text()
```

Для чтения двоичных данных предназначен метод `path.read_bytes`.

Если нужно перезаписать старый файл или записать новый, существуют методы для записи как текста, так и двоичных данных:

```
In [38]: path = pathlib.Path("/Users/kbehrman/sp.config")
```

```
In [39]: path.write_text("LOG:DEBUG")
Out[39]: 9
```

```
In [40]: path = pathlib.Path("/Users/kbehrman/sp")
Out[41]: 8
```

Для неструктурированного текста обычно вполне достаточно функций `read` и `write` файлового объекта, но что делать, если речь идет о более сложных данных? Для хранения простых структурированных данных в современных сервисах широко применяется формат нотации объектов JavaScript (JavaScript Object Notation, JSON). В нем действуют две структуры данных: ассоциативный массив `map` «ключ/значение», подобный ассоциативным массивам языка Python, и список элементов, подобный списку в Python. В нем описаны типы данных для чисел, строк, *булевых* значений (для хранения значений «истина / ложь») и неопределенных (пустых) значений (`null`). Веб-сервис AWS Identity and Access Management (IAM) позволяет управлять доступом к ресурсам AWS. Для описания стратегий доступа в нем используют JSON-файлы, например:

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
```



```

        "Action": "service-prefix:action-name",
        "Resource": "*",
        "Condition": {
            "DateGreaterThan": {"aws:CurrentTime": "2017-07-01T00:00:00Z"},
            "DateLessThan": {"aws:CurrentTime": "2017-12-31T23:59:59Z"}
        }
    }
}

```

Для извлечения данных из подобного файла можно использовать стандартные методы `read` и `readlines` файлового объекта:

```

In [8]: with open('service-policy.json', 'r') as opened_file:
...:     policy = opened_file.readlines()
...:
...:

```

Результат нельзя будет применить сразу же, поскольку он будет выглядеть как одна строка или список строк в зависимости от выбранного метода чтения:

```

In [9]: print(policy)
['{\n',
 '  "Version": "2012-10-17",\n',
 '\n',
 '  "Statement": {\n',
 '    "Effect": "Allow",\n',
 '\n',
 '    "Action": "service-prefix:action-name",\n',
 '\n',
 '    "Resource": "*,\n',
 '\n',
 '    "Condition": {\n',
 '      "DateGreaterThan": {"aws:CurrentTime": "2017-07-01T00:00:00Z"},\n',
 '\n',
 '      "DateLessThan": {"aws:CurrentTime": "2017-12-31T23:59:59Z"}\n',
 '    }\n',
 '  }\n',
 '}\n']

```

Далее нужно будет произвести синтаксический разбор этой строки (строк) в структуры данных и типы, соответствующие исходному файлу, что может потребовать немалых усилий. Так что намного удобнее будет воспользоваться модулем `json`:

```

In [10]: import json
In [11]: with open('service-policy.json', 'r') as opened_file:
...:     policy = json.load(opened_file)
...:
...:
...:

```

Этот модуль производит синтаксический разбор форм-ат JSON и возвращает данные в виде соответствующих структур данных Python:

```
In [13]: from pprint import pprint
```

```
In [14]: pprint(policy)
{'Statement': {'Action': 'service-prefix:action-name',
               'Condition': {'DateGreaterThan':
                             {'aws:CurrentTime': '2017-07-01T00:00:00Z'},
                             'DateLessThan':
                             {'aws:CurrentTime': '2017-12-31T23:59:59Z'}},
               'Effect': 'Allow',
               'Resource': '*'},
 'Version': '2012-10-17'}
```



Модуль `pprint` автоматически форматирует объекты Python для вывода в консоль. Выводимые им данные обычно намного удобнее для чтения и анализа вложенных структур данных.

Теперь можно работать с данными с исходной структурой из файла. Например, вот так можно изменить ресурс, доступом к которому управляет эта стратегия, на S3:

```
In [15]: policy['Statement']['Resource'] = 'S3'
In [16]: pprint(policy)
{'Statement': {'Action': 'service-prefix:action-name',
               'Condition': {'DateGreaterThan':
                             {'aws:CurrentTime': '2017-07-01T00:00:00Z'},
                             'DateLessThan':
                             {'aws:CurrentTime': '2017-12-31T23:59:59Z'}},
               'Effect': 'Allow',
               'Resource': 'S3'},
 'Version': '2012-10-17'}
```

С помощью метода `json.dump` можно записать ассоциативный массив Python в JSON-файл. Вот так можно обновить только что модифицированный файл стратегии:

```
In [17]: with open('service-policy.json', 'w') as opened_file:
...:     policy = json.dump(policy, opened_file)
...:
...:
...:
```

В файлах конфигурации нередко применяется и еще один язык — *YAML*, представляющий собой расширенную версию JSON, но в более сложной форме, в котором пробелы используются так же, как в Python.

Одн из утилит, предн зн ченных для втом тиз ции н стройки прогр ммно-го обеспечения, его р звертыв ния и упр вления им, — Ansible. Для опис ния втом тизируемых действий в Ansible применяют т к н зыв емые сборники сцен риев (playbooks) в форм те YAML:

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
    - name: ensure apache is at the latest version
      yum:
        name: httpd
        state: latest
  ...
```

Для синт ксического р збор YAML в Python ч ще всего используется библиотек PyYAML. Он не входит в ст нд ртную библиотеку Python, но ее можно уст новить с помощью pip:

```
$ pip install PyYAML
```

После уст новки можно импортиров ть и экспортиров ть д нные в форм те YAML с помощью PyYAML н логично тому, к к мы дел ли с JSON:

```
In [18]: import yaml
```

```
In [19]: with open('verify-apache.yml', 'r') as opened_file:
...:     verify_apache = yaml.safe_load(opened_file)
...:
```

Д нные при этом з груз ются в уже привычные для н с структуры д нных Python (список, содерж щий ссоци тивный м ссив):

```
In [20]: pprint(verify_apache)
[{'handlers': [{'name': 'restart apache',
                  'service': {'name': 'httpd', 'state': 'restarted'}}],
 'hosts': 'webservers',
 'remote_user': 'root',
 'tasks': [{'name': 'ensure apache is at the latest version',
             'yum': {'name': 'httpd', 'state': 'latest'}},
            {'name': 'write the apache config file',
             'notify': ['restart apache'],
             'template': {'dest': '/etc/httpd.conf', 'src': '/srv/httpd.j2'}},
            {'name': 'ensure apache is running',
             'service': {'name': 'httpd', 'state': 'started'}}],
 'vars': {'http_port': 80, 'max_clients': 200}]
```

Можно также сохранять данные из Python в формате YAML:

```
In [22]: with open('verify-apache.yml', 'w') as opened_file:
...:     yaml.dump(verify_apache, opened_file)
...:
...:
...:
```

Еще один язык, широко применяемый для представления структурированных данных, — XML (Extensible Markup Language, расширяемый язык разметки). В нем используются иерархические документы, состоящие из маркированных элементов. Исторически так сложилось, что многие веб-системы используют XML для передачи данных, в частности, для RSS-каналов. С помощью RSS-каналов отслеживаются обновления веб-сайтов и оповещают о них пользователей, также отслеживаются публикации статей в различных источниках. RSS-каналы применяют строки в формате XML. Python включает библиотеку `xml`, предназначенную для работы с XML-документами и умеющую отображать иерархические структуры XML-документов как древовидные структуры данных. Узлы деревьев играют роль элементов XML, иерархия моделируется с помощью взаимосвязи «родительский элемент — дочерний элемент». Смысловой верхний родительский узел называется корневым элементом. Произвести синтаксический разбор XML-документа RSS и получить его корневой элемент можно следующим образом:

```
In [1]: import xml.etree.ElementTree as ET
In [2]: tree = ET.parse('http_feeds.feedburner.com_oreilly_radar_atom.xml')

In [3]: root = tree.getroot()

In [4]: root
Out[4]: <Element '{http://www.w3.org/2005/Atom}feed' at 0x11292c958>
```

Обход деревьев можно выполнить посредством прохода в цикле дочерних узлов:

```
In [5]: for child in root:
...:     print(child.tag, child.attrib)
...:
{http://www.w3.org/2005/Atom}title {}
{http://www.w3.org/2005/Atom}id {}
{http://www.w3.org/2005/Atom}updated {}
{http://www.w3.org/2005/Atom}subtitle {}
{http://www.w3.org/2005/Atom}link {'href': 'https://www.oreilly.com'}
{http://www.w3.org/2005/Atom}link {'rel': 'hub',
                                   'href': 'http://pubsubhubbub.appspot.com/'}
{http://www.w3.org/2003/01/geo/wgs84_pos#}long {}
{http://rssnamespace.org/feedburner/ext/1.0}emailServiceId {}
...
```

XML позволяет использовать *пространства имен* (группировать данные с помощью тегов). В XML перед тегами в скобках указываются пространства имен. Если структура иерархии известна, можно искать элементы на основе их путей. Для удобства можно передать ассоциативный массив с описанием пространств имен:

```
In [108]: ns = {'default':'http://www.w3.org/2005/Atom'}
In [106]: authors = root.findall("default:entry/default:author/default:name",
ns)

In [107]: for author in authors:
...:     print(author.text)
...:
Nat Torkington
VM Brasseur
Adam Jacob
Roger Magoulas
Pete Skomoroch
Adrian Cockcroft
Ben Lorica
Nat Torkington
Alison McCauley
Tiffani Bell
Arun Gupta
```

Вы можете столкнуться также с необходимостью работы с данными в виде записей, отделенных друг от друга запятыми (CSV). Этот формат часто применяется для данных в электронных таблицах. Чтобы было удобно их читать, можно воспользоваться модулем `csv` Python:

```
In [16]: import csv
In [17]: file_path = '/Users/kbehrman/Downloads/registered_user_count_ytd.csv'

In [18]: with open(file_path, newline='') as csv_file:
...:     off_reader = csv.reader(csv_file, delimiter=',')
...:     for _ in range(5):
...:         print(next(off_reader))
...:
['Date', 'PreviousUserCount', 'UserCountTotal', 'UserCountDay']
['2014-01-02', '61', '5336', '5275']
['2014-01-03', '42', '5378', '5336']
['2014-01-04', '26', '5404', '5378']
['2014-01-05', '65', '5469', '5404']
```

Объект `csv.reader` проходит в цикле CSV-файл построчно, благодаря чему можно обработать данные по одной строке за раз. Такой способ обработки файлов особенно удобен для больших CSV-файлов, которые нежелательно полностью загружать в оперативную память. Конечно, если нужно выполнять

вычисления и столбцы из нескольких строк и файл не слишком велик, имеет смысл загрузить его в память целиком.

Пакет Pandas — чрезвычайно популярный инструмент для анализа данных. Он содержит структуру данных `pandas.DataFrame`, ведущую себя подобие таблице данных, и логичной электронной таблице с очень широкими возможностями. `DataFrame` — идеальный инструмент для статистического анализа табличных данных или каких-либо операций с их столбцами или строками. Это сторонняя библиотека, которую необходимо установить с помощью `pip`. Существует множество способов загрузки данных в объекты `DataFrame`, один из наиболее простых — загрузка из CSV-файла:

```
In [54]: import pandas as pd

In [55]: df = pd.read_csv('sample-data.csv')

In [56]: type(df)
Out[56]: pandas.core.frame.DataFrame
```

Можете просмотреть несколько первых строк объекта `DataFrame` с помощью метода `head`:

```
In [57]: df.head(3)
Out[57]:
```

	Attributes	open	high	low	close	volume
0	Symbols	F	F	F	F	F
1	date	NaN	NaN	NaN	NaN	NaN
2	2018-01-02	11.3007	11.4271	11.2827	11.4271	20773320

А основные статистические показатели `DataFrame` можно вывести с помощью метода `describe`:

```
In [58]: df.describe()
Out[58]:
```

	Attributes	open	high	low	close	volume
count	357	356	356	356	356	356
unique	357	290	288	297	288	356
top	2018-10-18	10.402	8.3363	10.2	9.8111	36298597
freq	1	5	4	3	4	1

Или можно просмотреть отдельный столбец данных, указав его название в квадратных скобках:

```
In [59]: df['close']
Out[59]:
```

0	F
1	NaN
2	11.4271
3	11.5174

```

4      11.7159
      ...
352     9.83
353     9.78
354     9.71
355     9.74
356     9.52
Name: close, Length: 357, dtype: object

```

В библиотеке Pandas есть множество других методов для анализа табличных данных или выполнения различных операций над ними, ее использование посвящено множество книг. Желательно иметь о ней представление, если вам порой приходится анализировать данные.

Поиск в тексте с помощью регулярных выражений

HTTP-сервер Apache — веб-сервер с открытым исходным кодом, широко применяемый для выдачи веб-контента. Его можно не строить и сохранить файлы журналов в различных форматах. Один из часто используемых форматов — единый формат журналов (Common Log Format, CLF), понятный множеству утилит, предназначенных для анализа журналов. Далее приведено устройство этого формата:

```
<IP-адрес> <Id клиента> <Id пользователя> <Время> <Запрос> <Состояние> <Размер>
```

А вот пример строки журнала в этом формате:

```
127.0.0.1 - swills [13/Nov/2019:14:43:30 -0800] "GET /assets/234 HTTP/1.0" 200 2326
```

В главе 1 вы познакомились с регулярными выражениями и модулем `re` языка Python, так что воспользуемся ими для извлечения информации из журналов в вышеупомянутом едином формате. Один из приемов формирования регулярных выражений — по частям, благодаря чему можно добиться правильной работы каждого из подвыражений без проблем с отладкой выражения в целом. С помощью поименованных групп можно создать регулярное выражение для извлечения из строки IP-адреса:

```
In [1]: line = '127.0.0.1 - rj [13/Nov/2019:14:43:30 -0800] "GET HTTP/1.0" 200'
```

```
In [2]: re.search(r'(?P<IP>\d+\.\d+\.\d+\.\d+)', line)
Out[2]: <re.Match object; span=(0, 9), match='127.0.0.1'>
```

```
In [3]: m = re.search(r'(?P<IP>\d+\.\d+\.\d+\.\d+)', line)
```

```
In [4]: m.group('IP')
Out[4]: '127.0.0.1'
```

Можно также создать регулярное выражение для получения времени:

```
In [5]: r = r'\[(?P<Time>\d\d\/\w{3}\/\d{4}:\d{2}:\d{2}:\d{2})\]'
```

```
In [6]: m = re.search(r, line)
```

```
In [7]: m.group('Time')
```

```
Out[7]: '13/Nov/2019:14:43:30'
```

Можно захватить сразу несколько элементов — IP-адрес, пользователя, время и запрос, как вот здесь:

```
In [8]: r = r'(?P<IP>\d+\.\d+\.\d+\.\d+)'
```

```
In [9]: r += r' - (?P<User>\w+)'
```

```
In [10]: r += r'\[(?P<Time>\d\d\/\w{3}\/\d{4}:\d{2}:\d{2}:\d{2})\]'
```

```
In [11]: r += r' (?P<Request>".+")'
```

```
In [12]: m = re.search(r, line)
```

```
In [13]: m.group('IP')
```

```
Out[13]: '127.0.0.1'
```

```
In [14]: m.group('User')
```

```
Out[14]: 'rj'
```

```
In [15]: m.group('Time')
```

```
Out[15]: '13/Nov/2019:14:43:30'
```

```
In [16]: m.group('Request')
```

```
Out[16]: '"GET HTTP/1.0"'
```

Синтаксический разбор отдельной строки журнала — задача интересная, но пользы от нее немного. Впрочем, это регулярное выражение можно применять в качестве фундамента для создания регулярного выражения, позволяющего извлекать информацию из всего журнала. Пусть нам нужно извлечь все IP-адреса для запросов типа GET, имевших место 8 ноября 2019 года. Мы можем внести в предыдущее выражение необходимые модификации в соответствии со спецификой своего запроса:

```
In [62]: r = r'(?P<IP>\d+\.\d+\.\d+\.\d+)'
```

```
In [63]: r += r' - (?P<User>\w+)'
```

```
In [64]: r += r'\[(?P<Time>08/Nov/\d{4}:\d{2}:\d{2}:\d{2}) [-+]\d{4})\]'
```

```
In [65]: r += r' (?P<Request>"GET .+")'
```

Для обработки журнала воспользуемся методом `finditer`, выводя в консоль IP-адреса из соответствующих строк:


```
In [66]: matched = re.finditer(r, access_log)
```

```
In [67]: for m in matched:
...:     print(m.group('IP'))
...:
127.0.0.1
342.3.2.33
```

С помощью регулярных выражений можно делить с текстом очень многое. Если они вас не пугают, вы обнаружите, что это один из самых мощных инструментов для работы с текстом.

Обработка больших файлов

Иногда требуется обработать очень большие файлы. Если данные в файле можно обрабатывать построчно, то при использовании Python все очень просто: вместо его загрузки в оперативную память целиком, как мы делали ранее, можно читать по одной строке за раз, обрабатывать ее, затем переходить к следующей. Средство сборки мусора Python автоматически удаляет из памяти эти строки, освобождая за ним место для следующей.



Python автоматически выделяет и освобождает память. Для этого служит системная сборка мусора. Средством сборки мусора Python можно управлять с помощью пикетов, хотя требуется это редко.

Чтение файлов, созданных различными операционными системами, затрудняется тем фактом, что в этих ОС используются разные символы окончания строк. Созданные в Windows файлы содержат символ `\r` в дополнение к `\n`. А в Linux-системах они отображаются в виде чистого текста. Если нужно в большом файле подправить символы окончания строк в соответствии с применяемой операционной системой, можно открыть этот файл и читать его построчно, сохраняя строки в новый файл. А Python возьмет на себя преобразование символов окончания строк:

```
In [23]: with open('big-data.txt', 'r') as source_file:
...:     with open('big-data-corrected.txt', 'w') as target_file:
...:         for line in source_file:
...:             target_file.write(line)
...:
```

Обратите внимание на возможность использования вложенных операторов `with` для открытия двух файлов сразу и проход в цикле построчно по исходному файловому объекту. Для этого удобно описать функцию-генератор, особенно

если нужно выполнять синхронный сбор нескольких файлов одновременно в одной строке код :

```
In [46]: def line_reader(file_path):
...:     with open(file_path, 'r') as source_file:
...:         for line in source_file:
...:             yield line
...:

In [47]: reader = line_reader('big-data.txt')

In [48]: with open('big-data-corrected.txt', 'w') as target_file:
...:     for line in reader:
...:         target_file.write(line)
...:
```

Если вы не используете для разбиения своих данных символы окончания строк, например, в больших двоичных файлах, то можете читать данные порциями, передвигая количество байтов в порции методом `read` файлового объекта. Когда данные для чтения закончатся, вызов вернет пустую строку:

```
In [27]: with open('bb141548a754113e.jpg', 'rb') as source_file:
...:     while True:
...:         chunk = source_file.read(1024)
...:         if chunk:
...:             process_data(chunk)
...:         else:
...:             break
...:
```

Шифрование текста

Для обеспечения безопасности часто приходится шифровать текст. Помимо встроенного пакета `hashlib` Python существует широко используемый сторонний пакет `cryptography`. Взглянем на них.

Хеширование с помощью пакета `hashlib`

Для безопасности роли пользователей необходимо хранить в зашифрованном виде. Обычно для этой цели роли шифруют с помощью односторонней функции в битовую строку, восстановление по которой исходный роль практически невозможно. Подобные функции называются *хеш-функциями*. Помимо маскировки ролей они позволяют гарантировать неизменность отправленных по Сети документов. Для этого вычисляется хеш-функция документа и полученное значение отправляется одновременно с документом. Получатель может вычислить хеш-значение документа и убедиться, что оно совпадает с полученным.

Библиотека `hashlib` содержит безопасные алгоритмы хеширования, включая SHA1, SHA224, SHA384, SHA512 и MD5. Вот так можно хешировать пироль с помощью алгоритма MD5:

```
In [62]: import hashlib

In [63]: secret = "This is the password or document text"

In [64]: bsecret = secret.encode()

In [65]: m = hashlib.md5()

In [66]: m.update(bsecret)

In [67]: m.digest()
Out[67]: b'\xf5\x06\xe6\xfc\x1c\xbe\x86\xddj\x96C\x10\x0f5E'
```

Учтите, что, если в пироль или документ предстает собой строковое значение, необходимо преобразовать его в двоичную строку с помощью метода `encode`.

Шифрование с помощью библиотеки `cryptography`

Библиотеку `cryptography` часто выбирают для решения криптографических задач на языке Python. Это сторонний пакет, который необходимо скачать и установить с помощью `pip`. *Шифрование с симметричным ключом* (symmetric key encryption) — группа алгоритмов шифрования, основанных на совместно используемых ключах. В число таких алгоритмов AES (Advanced Encryption Algorithm — продвинутый алгоритм шифрования), Blowfish, DES (Data Encryption Standard — стандарт шифрования данных), Serpent и Twofish. Совместно используемый ключ логичен по отношению, применяемому как для шифрования, так и для расшифровки текста. Совместное использование одного ключа как создателем, так и читателем зашифрованного файла — недостаток по сравнению с *шифрованием с симметричным ключом*, которое мы обсудим чуть позже. Однако при шифровании с симметричным ключом быстродействие выше и алгоритм проще, так что для шифрования больших файлов он подходит лучше. Одним из реализаций популярного алгоритма AES — Fernet. Скачать необходимо сгенерировать ключ:

```
In [1]: from cryptography.fernet import Fernet

In [2]: key = Fernet.generate_key()

In [3]: key
Out[3]: b'q-fEOs2JJIRINDR8toMG7zhQvVhvf5BRPx3mj5Atk5B8='
```

Этот ключ нельзя терять, ведь он необходим для расшифровки. Учтите, что любой, у кого есть доступ к этому ключу, сможет расшифровать файлы.

При шифровании текста ключ в файле необходимо использовать двоичный тип данных. Следующий шаг — шифрование данных с помощью объекта `Fernet`:

```
In [4]: f = Fernet(key)

In [5]: message = b"Secrets go here"

In [6]: encrypted = f.encrypt(message)

In [7]: encrypted
Out[7]: b'gAAAAABdPyg4 ... plhkpVkc8ez0HaOLIA=='
```

Решифрование данных можно с помощью объекта `Fernet`, созданного на основе того же ключа:

```
In [1]: f = Fernet(key)

In [2]: f.decrypt(encrypted)
Out[2]: b'Secrets go here'
```

При шифровании с симметричным ключом используется пара ключей: один открытый, второй секретный. Открытый ключ должен быть общедоступным, секретный — известен только одному пользователю. Решифрование зашифрованных с помощью открытого ключа сообщений можно, только используя соответствующий секретный ключ. Это свойство шифрования широко применяется для безопасной передачи информации в локальных сетях, так и через Интернет. Один из самых популярных алгоритмов шифрования с симметричным ключом RSA (Rivest — Shamir — Adleman, Ривест — Шамир — Адлеман) широко используется для общения в персональных сетях. Библиотека `cryptography` позволяет создать пару из открытого и секретного ключей:

```
In [1]: from cryptography.hazmat.backends import default_backend

In [2]: from cryptography.hazmat.primitives.asymmetric import rsa

In [3]: private_key = rsa.generate_private_key(public_exponent=65537,
                                              key_size=4096,
                                              backend=default_backend())

In [4]: private_key
Out[4]: <cryptography.hazmat.backends.openssl.rsa._RSAPrivateKey at 0x10d377c18>

In [5]: public_key = private_key.public_key()

In [6]: public_key = private_key.public_key()

In [7]: public_key
Out[7]: <cryptography.hazmat.backends.openssl.rsa._RSAPublicKey at 0x10da642b0>
```

Далее можно зашифровать данные с помощью открытого ключа:

```
In [8]: message = b"More secrets go here"

In [9]: from cryptography.hazmat.primitives.asymmetric import padding
In [11]: from cryptography.hazmat.primitives import hashes

In [12]: encrypted = public_key.encrypt(message,
...:     padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
...:     algorithm=hashes.SHA256(),
...:     label=None))
```

И расшифровать их с помощью секретного ключа:

```
In [13]: decrypted = private_key.decrypt(encrypted,
...:     padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
...:     algorithm=hashes.SHA256(),
...:     label=None))

In [14]: decrypted
Out[14]: b'More secrets go here'
```

Модуль os

`os` — один из чаще всего используемых модулей Python, позволяющий выполнять множество низкоуровневых системных вызовов. Кроме того, он предоставляет единообразный интерфейс для различных операционных систем, что важно для приложений, которые могут потребоваться применять как в Windows-, так и в Unix-системах. Он также обеспечивает некоторые возможности, ориентированные на конкретные операционные системы (`os.O_TEXT` для Windows и `os.O_CLOEXEC` в Linux) и недоступные на всех платформах. Использовать их следует только в том случае, если для приложения точно не требуется переносимость между различными операционными системами. В примере 2.1 показаны некоторые из самых полезных методов модуля `os`.

Пример 2.1. Методы модуля os

```
In [1]: os.listdir('.') ❶
Out[1]: ['__init__.py', 'os_path_example.py']

In [2]: os.rename('_crud_handler', 'crud_handler') ❷

In [3]: os.chmod('my_script.py', 0o777) ❸

In [4]: os.mkdir('/tmp/holding') ❹

In [5]: os.makedirs('/Users/kbehrman/tmp/scripts/devops') ❺
```

```

In [6]: os.remove('my_script.py') ❹
In [7]: os.rmdir('/tmp/holding') ❺
In [8]: os.removedirs('/Users/kbehrman/tmp/scripts/devops') ❻
In [9]: os.stat('crud_handler') ❼
Out[9]: os.stat_result(st_mode=16877,
                        st_ino=4359290300,
                        st_dev=16777220,
                        st_nlink=18,
                        st_uid=501,
                        st_gid=20,
                        st_size=576,
                        st_atime=1544115987,
                        st_mtime=1541955837,
                        st_ctime=1567266289)

```

- ❶ Вывод содержимого каталога.
- ❷ Переименование файла или каталога.
- ❸ Изменение прав доступа файла или каталога.
- ❹ Создание каталога.
- ❺ Рекурсивное создание каталога.
- ❻ Удаление файла.
- ❼ Удаление отдельного каталога.
- ❽ Удаление деревьев каталогов, начиная с конечного каталога и далее по дереву. Операция прекращается на первом же непустом каталоге.
- ❾ Получение статистики файла или каталога, включающей `st_mode`, тип файла и прав доступа, а также `st_atime` — время последнего обращения к данному элементу.

Управление файлами и каталогами с помощью `os.path`

В Python можно использовать строковые значения (двоичные или обычные) для представления путей. Модуль `os.path` предоставляет множество методов для создания путей в виде строковых значений и различных операций над

ними. Как уже упоминалось, модуль `os` стремится к кросс-платформенному поведению, и подмодуль `os.path` не исключение. Он интерпретирует пути в соответствии с действующей операционной системой с использованием прямых косых черт для разделения каталогов в Unix-подобных системах и обратных — в Windows. Благодаря этому программист может формировать подходящие для любой операционной системы пути на лету. Возможность удобного разбиения и «склеивания» путей — вероятно, чаще всего применяемая функциональность подмодуля `os.path`. Для разбиения путей используются три метода — `split`, `basename` и `dirname`:

```
In [1]: import os

In [2]: cur_dir = os.getcwd() ❶

In [3]: cur_dir
Out[3]: '/Users/kbehrman/Google-Drive/projects/python-devops/samples/chapter4'

In [4]: os.path.split(cur_dir) ❷
Out[4]: ('/Users/kbehrman/Google-Drive/projects/python-devops/samples',
        'chapter4')

In [5]: os.path.dirname(cur_dir) ❸
Out[5]: '/Users/kbehrman/Google-Drive/projects/python-devops/samples'

In [6]: os.path.basename(cur_dir) ❹
Out[6]: 'chapter4'
```

- ❶ Получаем текущий рабочий каталог.
- ❷ `os.path.split` отделяет конечный уровень пути от родительского пути.
- ❸ `os.path.dirname` возвращает родительский путь.
- ❹ `os.path.basename` возвращает название конечного каталога.

`os.path.dirname` удобно использовать для обхода деревьев каталогов:

```
In [7]: while os.path.basename(cur_dir):
...:     cur_dir = os.path.dirname(cur_dir)
...:     print(cur_dir)
...:
/Users/kbehrman/projects/python-devops/samples
/Users/kbehrman/projects/python-devops
/Users/kbehrman/projects
/Users/kbehrman
/Users
/
```

Файлы часто используют для настройки приложения во время выполнения, в Unix-системах файлы в соответствии с соглашением называются по расширению, заканчиваясь на `rc`. Два простейших примера: файл `.vimrc` текстового редактора Vim и файлы `.bashrc` командной оболочки Bash. Хотя эти файлы можно в различных местах. В программах часто описывается иерархия каталогов для их поиска. Например, утилит может сначала проверить переменную среды, в которой описано, какой файл `rc` использовать, если таковой нет, проверить текущий каталог, затем домашний каталог активного пользователя. В примере 2.2 мы попробуем найти файл `rc` в этих местах. Для этого возьмем переменную `file`, значение которой Python вводит автоматически при запуске кода Python из файла. В эту переменную заносится путь относительно действующего рабочего каталога, не абсолютный/полный путь. Python не дополняет автоматически пути до абсолютного, как принято в Unix-системах, так что придется сделать это самим, прежде чем воспользоваться этим путем для формирования пути поиска файла `rc`. Анализируя Python не дополняет автоматически пути в переменных среды, так что и это нам придется сделать явно.

Пример 2.2. Метод `find_rc`

```
def find_rc(rc_name=".example.rc"):

    # Проверяем переменную среды
    var_name = "EXAMPLERC_DIR"
    if var_name in os.environ: ❶
        var_path = os.path.join(f"${var_name}", rc_name) ❷
        config_path = os.path.expandvars(var_path) ❸
        print(f"Checking {config_path}")
        if os.path.exists(config_path): ❹
            return config_path

    # Ищем в рабочем каталоге
    config_path = os.path.join(os.getcwd(), rc_name) ❺
    print(f"Checking {config_path}")
    if os.path.exists(config_path):
        return config_path

    # Ищем в домашнем каталоге пользователя
    home_dir = os.path.expanduser("~/") ❻
    config_path = os.path.join(home_dir, rc_name)
    print(f"Checking {config_path}")
    if os.path.exists(config_path):
        return config_path

    # Ищем в каталоге выполняемого файла
    file_path = os.path.abspath(__file__) ❼
    parent_path = os.path.dirname(file_path) ❽
```



```

config_path = os.path.join(parent_path, rc_name)
print(f"Checking {config_path}")
if os.path.exists(config_path):
    return config_path

print(f"File {rc_name} has not been found")

```

- ❶ Проверяем, существует ли т к я переменн я в ктивной среде.
- ❷ Формируем путь с ук з нием н зв ния переменной среды. Он будет выглядеть примерно т к: \$EXAMPLERC_DIR/.exampleerc.
- ❸ Дополняем переменную среды для вст вки ее зн чения в формируемый путь.
- ❹ Проверяем, существует ли т кой ф йл.
- ❺ Формируем путь н основе р бочего к т лог .
- ❻ Получ ем путь к дом шнему к т лог у пользо в теля с помощью функции expanduser.
- ❼ Дополняем хр нящийся в file относительный путь до бсолютного.
- ❸ Получ ем путь к содерж щему текущий ф йл к т лог у с помощью dirname.

Подмодуль path т кже позволяет получить х р ктеристики пути. С его помощью можно выяснить, чем является путь — ф йлом, к т логом, ссылкой или точкой монтиров ния. Можно т кже узн ть т кие его х р ктеристики, к к р змер и время последнего обр щения/изменения. В примере 2.3 мы выполним с помощью path обход дерев к т логов и выведем информ цию о р змере и последнем времени доступ ко всем содерж щимся в них ф йл м.

Пример 2.3. os_path_walk.py

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

```

import fire
import os

def walk_path(parent_path):
    print(f"Checking: {parent_path}")
    childs = os.listdir(parent_path) ❶

    for child in childs:
        child_path = os.path.join(parent_path, child) ❷
        if os.path.isfile(child_path): ❸
            last_access = os.path.getatime(child_path) ❹
            size = os.path.getsize(child_path) ❺
            print(f"File: {child_path}")

```

```

        print(f"\tlast accessed: {last_access}")
        print(f"\tsize: {size}")
    elif os.path.isdir(child_path): ❸
        walk_path(child_path) ❷

if __name__ == '__main__':
    fire.Fire()

```

- ❶ `os.listdir` возвращает содержимое каталога.
- ❷ Формируем полный путь элемента, находящегося в родительском каталоге.
- ❸ Проверяем, не соответствует ли этот путь файлу.
- ❹ Получаем время последнего обращения к данному файлу.
- ❺ Получаем размер данного файла.
- ❻ Проверяем, не соответствует ли этот путь каталогу.
- ❼ Обходим дерево, начиная с этого каталога.

С помощью подобных сценариев можно не ходить по большим файлам или папкам, к которым пока что не обращались, а затем сообщать о них пользователю, перемещать или удалять их.

Обход дерева каталогов с помощью `os.walk`

Модуль `os` включает удобную функцию `os.walk` для обхода деревьев каталогов, которая возвращает генератор, который, в свою очередь, на каждой итерации возвращает кортеж, состоящий из текущего пути, списка каталогов и списка файлов. В примере 2.4 мы переписываем функцию `walk_path` из примера 2.3, применив функцию `os.walk`. Как вы видели в предыдущем примере, при использовании `os.walk` не нужно проверять, какие пути соответствуют файлам, или повторно вызывать функцию для каждого подкаталога.

Пример 2.4. Переписываем функцию `walk_path`

```

def walk_path(parent_path):
    for parent_path, directories, files in os.walk(parent_path):
        print(f"Checking: {parent_path}")
        for file_name in files:
            file_path = os.path.join(parent_path, file_name)
            last_access = os.path.getatime(file_path)
            size = os.path.getsize(file_path)
            print(f"File: {file_path}")
            print(f"\tlast accessed: {last_access}")
            print(f"\tsize: {size}")

```

Пути как объекты: библиотека pathlib

Библиотека `pathlib` позволяет представлять пути в виде объектов, а не строк. В примере 2.5 мы переписываем пример 2.2, используя `pathlib` вместо `os.path`.

Пример 2.5. Переписываем функцию `find_rc`

```
def find_rc(rc_name=".example.rc"):
```

```
    # Проверяем переменную среды
    var_name = "EXAMPLERC_DIR"
    example_dir = os.environ.get(var_name) ❶
    if example_dir:
        dir_path = pathlib.Path(example_dir) ❷
        config_path = dir_path / rc_name ❸
        print(f"Checking {config_path}")
        if config_path.exists(): ❹
            return config_path.as_posix() ❺

    # Ищем в текущем рабочем каталоге
    config_path = pathlib.Path.cwd() / rc_name ❻
    print(f"Checking {config_path}")
    if config_path.exists():
        return config_path.as_posix()

    # Ищем в домашнем каталоге пользователя
    config_path = pathlib.Path.home() / rc_name ❼
    print(f"Checking {config_path}")
    if config_path.exists():
        return config_path.as_posix()

    # Ищем в каталоге выполняемого файла
    file_path = pathlib.Path(__file__).resolve() ❽
    parent_path = file_path.parent ❾
    config_path = parent_path / rc_name
    print(f"Checking {config_path}")
    if config_path.exists():
        return config_path.as_posix()

    print(f"File {rc_name} has not been found")
```

❶ В момент написания данной книги библиотека `pathlib` не дополняет переменные среды до абсолютного пути. Поэтому мы берем значение переменной из `os.environ`.

❷ Создаем объект `pathlib.Path`, соответствующий используемой операционной системе.

❸ Новые объекты `pathlib.Path` можно создавать, указывая после родительского пути прямые косые черты и соответствующие строковые значения.

- ❹ У объекта `pathlib.Path` есть метод `exists`.
- ❺ Вызов `as_posix` возвращает путь в виде строки. В зависимости от ситуации можно вернуть и сам объект `pathlib.Path`.
- ❻ Метод класса `pathlib.Path.cmd` возвращает объект `pathlib.Path`, соответствующий текущему рабочему каталогу. Здесь мы сразу же используем этот объект для создания `config_path` путем объединения его со строковым значением `rc_name`.
- ❼ Метод класса `pathlib.Path.home` возвращает объект `pathlib.Path`, соответствующий домашнему каталогу активного пользователя.
- ❽ Создаем объект `pathlib.Path` на основе содержащегося в `file` относительного пути, после чего вызываем его метод `resolve` для получения абсолютного пути.
- ❾ Возвращаем родительский объект `pathlib.Path` непосредственно из самого объекта `file_path`.

Работа с командной строкой

Именно в командной строке происходит основная деятельность DevOps. И хотя существует множество замечательных утилит с графическим интерфейсом, командная строка по-прежнему остается наиболее естественной средой работы DevOps. Взаимодействие со средой командной оболочки из Python и создание утилит командной строки Python — необходимые составляющие использования Python для DevOps.

Работа с командной оболочкой

Язык Python предоставляет утилиты для взаимодействия с различными системами и командными оболочками. Рекомендуем вам хорошо ознакомиться с этими важными инструментами, к которым относятся модули `sys`, `os` и `subprocess`.

Взаимодействие с интерпретатором с помощью модуля `sys`

Модуль `sys` предоставляет доступ к переменным и методам, тесно связанным с интерпретатором Python.



Биты во время чтения можно интерпретировать двумя основными способами. При первом, с прямым порядком битов (*little endian*), считается, что значимость битов растет (то есть каждый последующий бит соответствует большему числу). При втором, с обратным порядком битов (*big endian*), предполагается, что значимость первого бита максимальна, а далее она убывает.

Узнать порядок байтов в вшей архитектуре можно с помощью атрибута `sys.byteorder`:

```
In [1]: import sys
```

```
In [2]: sys.byteorder
```

```
Out[2]: 'little'
```

Размер объектов Python можно узнать с помощью функции `sys.getsizeof`. При ограниченном объеме памяти он может оказаться очень полезной:

```
In [3]: sys.getsizeof(1)
```

```
Out[3]: 28
```

Если ваш код должен вести себя по-разному в зависимости от операционной системы, можно воспользоваться `sys.platform` для проверки:

```
In [5]: sys.platform
```

```
Out[5]: 'darwin'
```

Еще часто встречается ситуация, когда необходимо задействовать возможности языка или модули, доступные только в определенных версиях Python. Управлять поведением в зависимости от текущего интерпретатора Python можно с помощью `sys.version_info`. В следующем примере выводятся различные сообщения для Python 3.7, Python версий выше 3, но ниже 3.7 и Python версий ниже 3:

```
if sys.version_info.major < 3:
    print("You need to update your Python version")
elif sys.version_info.minor < 7:
    print("You are not running the latest version of Python")
else:
    print("All is good.")
```

Мы обсудим применение модуля `sys` подробнее далее в этой главе, когда будем писать утилиты командной строки.

Взаимодействие с операционной системой с помощью модуля `os`

Вы уже видели, как в главе 2 модуль `os` использовался для работы с файловой системой. У него также есть множество разнообразных атрибутов и функций для работы с операционной системой. В примере 3.1 приведены некоторые из них.

Пример 3.1. Примеры возможностей модуля `os`

```
In [1]: import os
```

```
In [2]: os.getcwd() ❶
```

```
Out[2]: '/Users/kbehrman/Google-Drive/projects/python-devops'
```

```

In [3]: os.chdir('/tmp') ❷

In [4]: os.getcwd()
Out[4]: '/private/tmp'

In [5]: os.environ.get('LOGLEVEL') ❸

In [6]: os.environ['LOGLEVEL'] = 'DEBUG' ❹

In [7]: os.environ.get('LOGLEVEL')
Out[7]: 'DEBUG'

In [8]: os.getlogin() ❺
Out[8]: 'kbehrman'

```

❶ Получаем текущий рабочий каталог.

❷ Меняем текущий рабочий каталог.

❸ В `os.environ` хранятся переменные среды, значения которых были установлены при загрузке модуля `os`.

❹ Это и параметр конфигурации, и переменная среды. Данный параметр предназначен для порожденных этим кодом подпроцессов.

❺ Имя пользователя, запустившего терминал, из которого был порожден данный процесс.

Модуль `os` чаще всего применяется для извлечения параметров конфигурации из переменных среды, например уровня журналирования, или секретных данных, например ключей API.

Порождение процессов с помощью модуля `subprocess`

Во многих ситуациях приходится запускать из кода Python приложения вне Python, например встроенные инструкции командной оболочки, сценарии Bash или любые другие приложения командной строки. Для этого порождается новый *процесс* (экземпляр приложения). Модуль `subprocess` — как раз то, что нужно для порождения процесса и выполнения внутри него команд. С помощью `subprocess` вы сможете запускать свои любимые инструкции командной оболочки и прочие программы командной строки, также получать выводимые ими результаты из Python. В большинстве случаев для порождения процессов следует использовать функцию `subprocess.run`:

```

In [0]: import subprocess
In [1]: cp = subprocess.run(['ls', '-l'], capture_output=True,
                           universal_newlines=True)

In [2]: cp.stdout
Out[2]: 'total 96

```

1

0.7 0.6 1 0 .

```
In [4]: cp.stderr
```

```
<ipython input 23: c0ac40c40fcc> in <module>
```

```

166         if check and netcode:

```

```
167         raise CalledProcessError(errno.EPIPE, process.args)
```

```
467         raise CalledProcessError(retcode, process.args,
```

--\ 468


```

469         return CompletedProcess(process.args, retcode, stdout, stderr)
470

```

CalledProcessError: Command '['ls', '/doesnotexist']' returned non-zero exit

Таким образом, не придется проверять `stderr` на предмет сбоев и можно будет обработать ошибки, возвращаемые подпроцессами, так же как и прочие исключения Python.

Создание утилит командной строки

Простейший способ вызова скрипта Python из командной строки — с помощью `python`. При создании скрипта Python все операторы верхнего уровня (не вложенные в блоки код) выполняются при каждом его вызове или импорте. Если нужно вызывать какую-либо функцию при каждой загрузке кода, можно сделать это на верхнем уровне:

```

def say_it():
    greeting = 'Hello'
    target = 'Joe'
    message = f'{greeting} {target}'
    print(message)

```

```
say_it()
```

Эта функция будет выполняться при каждом вызове данного скрипта из командной строки:

```
$ python always_say_it.py
```

```
Hello Joe
```

А также при импорте файла:

```
In [1]: import always_say_it
Hello Joe
```

Впрочем, поступать так следует только с самыми простыми скриптами. Существенный недостаток этого подхода: при импорте модуля в другие модули Python код выполняется, не ждет спокойно, пока ему не обратится вызывающий модуль. А тот, кто импортирует модуль, обычно хочет сам определять, когда будет вызываться его содержимое. Можно добиться функциональности, выполняемую только при вызове из командной строки, с помощью глобальной переменной `name`. Как вы видели, он содержит имя модуля во время импорта. Если модуль вызывается непосредственно из командной строки, он получает строковое значение `main`. По соглашению запускемые в командной строке

модули заключаются блоком, в котором это проверяется, из него и запускается выполнение ориентированного на конкретную строку кода. Для модификации текущего сценария тем образом, чтобы функция в точности выполнялась только при вызове из конкретной строки, не во время импорта, необходимо вставить ее вызов в блок кода, следующий за проверкой:

```
def say_it():
    greeting = 'Hello'
    target = 'Joe'
    message = f'{greeting} {target}'
    print(message)

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

При импорте этой функции данный блок выполняться не будет, поскольку значение переменной `__name__` соответствует значению импортируемого модуля. Но будет выполняться при запуске модуля напрямую:

```
$ python say_it.py
```

```
Hello Joe
```

ДЕЛАЕМ СЦЕНАРИИ КОМАНДНОЙ ОБОЛОЧКИ ИСПОЛНЯЕМЫМИ

Чтобы не вызывать явным образом `python` в командной строке при запуске сценария, можно добавить строку `#!/usr/bin/env python` вверху файла:

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

def say_it():
    greeting = 'Hello'
    target = 'Joe'
    message = f'{greeting} {target}'
    print(message)

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

А затем сделать файл исполняемым с помощью команды `chmod` (утилита командной строки для задания прав доступа):

```
chmod +x say_it.py`
```

После этого можно будет непосредственно вызывать этот сценарий в командной строке без упоминания `python`:

```
$ ./say_it.py
```

```
Hello Joe
```

Первый этап создания утилит командной строки — выделение кода, который должен выполняться только при вызове в командной строке. Следующий шаг — прием аргументов командной строки. И если утилита не предназначена для одной-единственной задачи, необходимо получить команды, чтобы знать, что нужно сделать. Кроме того, утилиты командной строки, за исключением выполняющих простые примитивные задачи, принимают обязательные флаги для настройки. Помните, что эти команды и флаги играют роль *интерфейса пользователя* (user interface) для всех работающих с ними утилит, так что важно обеспечить удобство их применения и понятность. Написание документации — важная часть обеспечения понятности кода.

Использование sys.argv

Простейший способ обработки переданных в командной строке аргументов — атрибут `argv` модуля `sys`. Этот атрибут представляет собой список аргументов, переданных скрипту Python во время выполнения. Если скрипт выполняется в командной строке, первым аргументом должно быть его название. Остальные элементы списка представляют собой прочие аргументы командной строки в виде строковых значений:

```
#!/usr/bin/env python
"""
Простая утилита командной строки, использующая sys.argv
"""
import sys

if __name__ == '__main__':
    print(f"The first argument: '{sys.argv[0]}'")
    print(f"The second argument: '{sys.argv[1]}'")
    print(f"The third argument: '{sys.argv[2]}'")
    print(f"The fourth argument: '{sys.argv[3]}'")
```

Выполните его в командной строке — и увидите аргументы:

```
$ ./sys_argv.py --a-flag some-value 13
```

```
The first argument: './sys_argv.py'
The second argument: '--a-flag'
The third argument: 'some-value'
The fourth argument: '13'
```

Можете использовать эти аргументы для написания собственного средства синтаксического разбора аргументов. В примере 3.2 показано, как оно может выглядеть.

Пример 3.2. Синтаксический разбор с использованием sys.argv

```
#!/usr/bin/env python
"""
Простая утилита командной строки, использующая sys.argv
"""
import sys

def say_it(greeting, target):
    message = f'{greeting} {target}'
    print(message)

if __name__ == '__main__': ❶
    greeting = 'Hello' ❷
    target = 'Joe'

    if '--help' in sys.argv: ❸
        help_message = f"Usage: {sys.argv[0]} --name <NAME> --greeting <GREETING>"
        print(help_message)
        sys.exit() ❹

    if '--name' in sys.argv:
        # Выясняем позицию значения, следующего за флагом name
        name_index = sys.argv.index('--name') + 1 ❺
        if name_index < len(sys.argv): ❻
            name = sys.argv[name_index]

    if '--greeting' in sys.argv:
        # Выясняем позицию значения, следующего за флагом greeting
        greeting_index = sys.argv.index('--greeting') + 1
        if greeting_index < len(sys.argv):
            greeting = sys.argv[greeting_index]

    say_it(greeting, name) ❼
```

- ❶ Проверяем, запущены ли скрипт из командной строки.
- ❷ В этих двух строках задаются значения по умолчанию.
- ❸ Проверяем, присутствует ли в списке аргументов строковое значение `--help`.
- ❹ Выход из программы после вывода справки.
- ❺ Нам нужно знать позицию значения, следующего за флагом, к которому оно должно относиться.
- ❻ Проверяем, достаточно ли длинный список аргументов. Если нет, значит, не было указано значение для кого-то флага.
- ❼ Вызываем функцию с указанными аргументами значениями.

Пример 3.2 выводит простое спрочное сообщение и принимает аргументы для функции:

```
$ ./sys_argv.py --help
Usage: ./sys_argv.py --name <NAME> --greeting <GREETING>

$ ./sys_argv.py --name Sally --greeting Bonjour
Bonjour Sally
```

Этот подход чреват осложнениями и ошибками. Пример 3.2 не обрабатывает многие ситуации. Если пользователь допустит опечатку в названии флага или вводит его в неправильном регистре, флаг просто игнорируется, причем об этом не сообщается. Точно так же игнорируется ошибка, если пользователь попытается применить неподдерживаемую команду или указать для одного флага несколько значений. О подходе с синтаксическим сбором `argv` в м следует знать, но не используйте его для кодировки промышленной эксплуатации, разве что сознательно хотите написать средство синтаксического сбора аргументов. К счастью, существуют модули и пакеты, предназначенные специально для создания утилит командной строки. Эти пакеты предоставляют инфраструктуру для создания пользовательского интерфейса, предназначенного для запуска в командной строке модулей. Три популярных решения того плана — `argparse`, `click` и `python-fire`. Все три дают возможность проектировать нужные аргументы, необязательные флаги и средств отображения спрочной документации. Первый из них, `argparse`, входит в стандартную библиотеку языка Python, остальные два — сторонние пакеты, установленные отдельно с помощью `pip`.

argparse

Пакет `argparse` обстраивает многие нюансы передчи аргументов. С его помощью можно подробно проектировать пользовательский интерфейс командной строки, описывая команды и флаги вместе с соответствующими спрочными сообщениями. В нем используется идея объектов — синтаксических нлизов, с которыми связываются команды и флаги. Синтаксический нлизатор производит сбор аргументов, затем вы можете применять результаты для вызова своего кода. Интерфейс формируется с помощью объектов `ArgumentParser`, выполняющих синтаксический сбор вводимых пользователем данных:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Maritime control')
```

Позиционно зависимые команды и необязательные флаги добавляются в синтаксический анализатор с помощью метода `add_argument` (пример 3.3). Первый аргумент этого метода — название нового аргумента (команды или флага). Если название не начинается с тире, то предполагается, что это обязательный флаг, в противном случае — что это позиционно зависимая команда. Синтаксический анализатор создает объект для проанализированных аргументов, в котором аргументы играют роль трибутов, которые можно применять для получения доступа к введенным пользователем данным. Пример 3.3 — простая программа, повторяющая введенный пользователем текст и демонстрирующая основы работы с пакетом `argparse`.

Пример 3.3. `simple_parse.py`

```
#!/usr/bin/env python
"""
Утилита командной строки, использующая argparse
"""
import argparse
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Echo your input') ❶
    parser.add_argument('message', ❷
                        help='Message to echo')

    parser.add_argument('--twice', '-t', ❸
                        help='Do it twice', ❹
                        action='store_true')

    args = parser.parse_args() ❺

    print(args.message) ❻
    if args.twice:
        print(args.message)
```

❶ Создает объект для синтаксического анализатора со своим справочным сообщением.

❷ Добавляет позиционно зависимую команду со своим справочным сообщением.

❸ Добавляет обязательный аргумент.

❹ Создает необязательный аргумент в виде булевого значения.

❺ Производит синтаксический разбор аргументов с помощью синтаксического анализатора.

❻ Обращается к значениям аргументов по именам. `--` из имени необязательного аргумента убрано.

Если запустить эту программу с флагом `--twice`, введенное сообщение будет повторено дважды:

```
$ ./simple_parse.py hello --twice
hello
hello
```

Пакет `argparse` автоматически выдает справочные сообщения в зависимости от указанных в них параметров и описаний:

```
$ ./simple_parse.py --help
usage: simple_parse.py [-h] [--twice] message
```

Echo your input

```
positional arguments:
  message      Message to echo
```

```
optional arguments:
  -h, --help    show this help message and exit
  --twice, -t   Do it twice
```

Во многих утилитах командной строки используется несколько уровней команд для группировки команд по контролируемым сферам. Возьмем для примера `git`. В нем есть команды верхнего уровня, например `git stash`, у которых есть отдельные подкоманды, например `git stash pop`. С помощью пакета `argparse` можно создавать подкоманды путем создания субпарсеров глобального синтаксического парсера. С их помощью можно создать иерархическую команду. В примере 3.4 мы реализуем приложение для судоводителей команд для кораблей и моряков. К глобальному синтаксическому парсеру добавили два субпарсера, каждый со своими командами.

Пример 3.4. `argparse_example.py`

```
#!/usr/bin/env python
"""
Утилита командной строки, использующая argparse
"""
import argparse
def sail():
    ship_name = 'Your ship'
    print(f"{ship_name} is setting sail")
```

```

def list_ships():
    ships = ['John B', 'Yankee Clipper', 'Pequod']
    print(f"Ships: {' '.join(ships)}")

def greet(greeting, name):
    message = f'{greeting} {name}'
    print(message)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Maritime control') ❶

    parser.add_argument('--twice', '-t', ❷
                        help='Do it twice',
                        action='store_true')

    subparsers = parser.add_subparsers(dest='func') ❸

    ship_parser = subparsers.add_parser('ships', ❹
                                       help='Ship related commands')
    ship_parser.add_argument('command', ❺
                           choices=['list', 'sail'])

    sailor_parser = subparsers.add_parser('sailors', ❻
                                       help='Talk to a sailor')
    sailor_parser.add_argument('name', ❼
                              help='Sailors name')
    sailor_parser.add_argument('--greeting', '-g',
                              help='Greeting',
                              default='Ahoy there')

    args = parser.parse_args()
    if args.func == 'sailors': ❸
        greet(args.greeting, args.name)
    elif args.command == 'list':
        list_ships()
    else:
        sail()

```

❶ Созд ем синт ксический н лиз тор верхнего уровня.

❷ Доб вляем ргумент верхнего уровня, который можно использо вать с любыми ком нд ми из иер рхии этого синт ксического н лиз тор .

❸ Созд ем объект для суб н лиз торов. Для выбор суб н лиз тор служит трибут `dest`.

❹ Доб вляем суб н лиз тор для *ships*.

❺ Доб вляем ком нду в суб н лиз тор *ships*. Список возможных в ри нтов подком нд выводит п р метр `choices`.

❻ Доб вляем суб н лиз тор для *sailors*.

❼ Доб вляем обяза тельный позиционно з висимый ргумент в суб н лиз - тор *sailors*.

❸ Проверяем, к кой суб н лиз тор используется, по зн чению `func`.

В примере 3.4 есть один необяз тельный ргумент верхнего уровня (`twice`) и дв суб н лиз тор , к ждый со своими ком нд ми и фл г ми. П кет `argparse` втом тически созд ет иер рхию спр вочных сообщений и отобр ж ет их при использов нии фл г `--help`.

Ком нды верхнего уровня, включ я суб н лиз торы и ргумент верхнего уровня `twice`, сн бжены документ цией:

```
$ ./argparse_example.py --help
usage: argparse_example.py [-h] [--twice] {ships,sailors} ...
```

Maritime control

positional arguments:

{ships,sailors}

ships	Ship related commands
sailors	Talk to a sailor

optional arguments:

-h, --help	show this help message and exit
--twice, -t	Do it twice

Можно узн ть больше о подком нд х (суб н лиз тор х), ук з в фл г `help` после соответствующей ком нды:

```
$ ./argparse_example.py ships --help
usage: argparse_example.py ships [-h] {list,sail}
```

positional arguments:

{list,sail}

optional arguments:

-h, --help	show this help message and exit
------------	---------------------------------

К к видите, п кет `argparse` предост вляет широкие возможности упр вле- ния интерфейсом утилиты ком ндной строки. При жел нии можно созд ть специ льно подогн нный под конкретную рхитектуру многоуровневый интерфейс со встроенной документ цией и множеством опций. Впрочем, это потребов ло бы нем лых усилий, т к что взглянем н некоторые более про- стые в ри нты.

click

Пакет `click` изначально предназначен для работы с веб-фреймворком `flask`. Для привязки интерфейсной командной строки непосредственно к вешившим функциям в нем применяются *функции-декораторы* (function decorators). В отличие от пакета `argparse`, `click` переплетает интерфейсные решения с остальными частями кода.

ФУНКЦИИ-ДЕКОРАТОРЫ

Функции языка Python являются объектами, так что любая из них может принимать в качестве аргументов другие функции. Синтаксис декоратора — простой и аккуратный способ сделать это. Простейший формат декоратора:

```
In [2]: def some_decorator(wrapped_function):
...:     def wrapper():
...:         print('Do something before calling wrapped function')
...:         wrapped_function()
...:         print('Do something after calling wrapped function')
...:     return wrapper
...:
```

Мы можем описать другую функцию и передать ее как аргумент этой функции:

```
In [3]: def foobat():
...:     print('foobat')
...:
In [4]: f = some_decorator(foobat)
In [5]: f()
Do something before calling wrapped function
foobat
Do something after calling wrapped function
```

Синтаксис декоратора упрощает эту задачу, позволяя указать обертываемую функцию посредством *декорирования* ее аннотацией `@название_декоратора`. Вот пример использования синтаксиса декоратора для функции `some_decorator`:

```
In [6]: @some_decorator
...: def batfoo():
...:     print('batfoo')
...:
In [7]: batfoo()
Do something before calling wrapped function
batfoo
Do something after calling wrapped function
```

Теперь можно вызывать обернутую функцию по ее имени, а не по имени декоратора. Готовые функции-декораторы включены в состав как стандартной библиотеки языка Python (`staticmethod`, `classmethod`), так и сторонних пакетов, таких как `Flask` и `Click`.

Это значит, что можно привязывать флаги и опции непосредственно к параметрам соответствующих функций. Указываем функции `command` и `option` библиотеки `click` перед своей функцией в качестве декораторов, можно сделать из нее простую утилиту командной строки:

```
#!/usr/bin/env python
"""
Простой пример использования библиотеки Click
"""
import click

@click.command()
@click.option('--greeting', default='Hiya', help='How do you want to greet?')
@click.option('--name', default='Tammy', help='Who do you want to greet?')
def greet(greeting, name):
    print(f"{greeting} {name}")

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

Декоратор `click.command` указывает, что функция должна быть доступна для вызова из командной строки. Декоратор `click.option` служит для добавления аргумента командной строки с ватической привязкой его к параметру функции с соответствующим именем (`--greeting` к `greet`, `--name` к `name`). Библиотека `click` выполняет часть работы «закулисами», благодаря чему мы можем вызывать метод `greet` в блоке `main` без указания параметров, уже охваченных декоратором `option`.

Эти декораторы производят синтаксический сбор аргументов командной строки и ватически выдают спровочные сообщения:

```
$ ./simple_click.py --greeting Privet --name Peggy
Privet Peggy

$ ./simple_click.py --help
Usage: simple_click.py [OPTIONS]

Options:
  --greeting TEXT  How do you want to greet?
  --name TEXT      Who do you want to greet?
  --help           Show this message and exit.
```

Как видите, благодаря библиотеке `click` для применения функции в командной строке требуется гораздо меньше кода, чем при использовании `argparse`. Это позволяет разработчику сосредоточиться на бизнес-логике кода вместо проектирования интерфейса.

Теперь рассмотрим более сложный пример с вложенными командами. Вложение команд производится с помощью декоратора `click.group`, служащего для создания функций, представляющих группы. Для вложения команд в примере 3.5 мы используем пакет `click` с интерфейсом, очень похожим на интерфейс из примера 3.4.

Пример 3.5. `click_example.py`

```
#!/usr/bin/env python
"""
Утилита командной строки, использующая click
"""
import click

@click.group() ❶
def cli(): ❷
    pass

@click.group(help='Ship related commands') ❸
def ships():
    pass

cli.add_command(ships) ❹

@ships.command(help='Sail a ship') ❺
def sail():
    ship_name = 'Your ship'
    print(f"{ship_name} is setting sail")

@ships.command(help='List all of the ships')
def list_ships():
    ships = ['John B', 'Yankee Clipper', 'Pequod']
    print(f"Ships: {' '.join(ships)}")

@cli.command(help='Talk to a sailor') ❻
@click.option('--greeting', default='Ahoy there', help='Greeting for sailor')
@click.argument('name')
def sailors(greeting, name):
    message = f'{greeting} {name}'
    print(message)

if __name__ == '__main__':
    cli() ❼
```

❶ Создает группу верхнего уровня для прочих групп и команд.

❷ Создает функцию, которая будет выступать в роли группы верхнего уровня. Метод `click.group` преобразует функцию в группу.

❸ Создаем группу для команд `ships`.

❹ Добавляем в группу верхнего уровня группу `ships` в качестве команды. Обратите внимание на то, что функция `cli` теперь представляет собой группу с методом `add_command`.

❺ Добавляем команду в группу `ships`. Обратите внимание на то, что вместо `click.command` мы используем `ships.command`.

❻ Добавляем команду в группу `cli`.

❼ Вызываем группу верхнего уровня.

Справочные сообщения верхнего уровня, сгенерированные `click`, выглядят так:

```
./click_example.py --help
Usage: click_example.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.
```

```
Commands:
  sailors  Talk to a sailor
  ships    Ship related commands
```

А вот так можно посмотреть справку для подгруппы:

```
$ ./click_example.py ships --help
Usage: click_example.py ships [OPTIONS] COMMAND [ARGS]...

    Ship related commands
```

```
Options:
  --help  Show this message and exit.
```

```
Commands:
  list-ships  List all of the ships
  sail        Sail a ship
```

Если сравнить примеры 3.4 и 3.5, можно заметить различия между `argparse` и `click`. Подход `click` определенно требует меньшего объема кода — почти в два раза. Код пользовательского интерфейса (UI) перенесен по всей программе, что особенно важно при создании функций, работающих исключительно в качестве групп. В сложной программе с запутанным интерфейсом следует стремиться как можно сильнее изолировать различную функциональность, упрощая тем самым тестирование и отладку отдельных ее частей. В подобном случае имеет смысл применять `argparse`, чтобы отделить код интерфейса.

ОПИСАНИЕ КЛАССОВ

Описание класса начинается с ключевого слова `class`, за которым следуют имя класса и скобки:

```
In [1]: class MyClass():
```

Далее в блоке кода, сдвинутом вправо с помощью отступов, размещаются описания атрибутов и методов. Все методы класса получают в качестве первого параметра копию объекта класса. По соглашению она называется `self`:

```
In [1]: class MyClass():
...:     def some_method(self):
...:         print(f"Say hi to {self}")
...:
In [2]: myObject = MyClass()
In [3]: myObject.some_method()
Say hi to <__main__.MyClass object at 0x1056f4160>
```

У каждого класса есть метод `init`, вызываемый при создании экземпляра этого класса. Если не описать этот метод, класс унаследует от базового класса `object` языка Python метод по умолчанию:

```
In [4]: MyClass.__init__
Out[4]: <slot wrapper '__init__' of 'object' objects>
```

Атрибуты объекта обычно описываются в методе `init`:

```
In [5]: class MyOtherClass():
...:     def __init__(self, name):
...:         self.name = name
...:
In [6]: myOtherObject = MyOtherClass('Sammy')
In [7]: myOtherObject.name
Out[7]: 'Sammy'
```

fire

Попробуем еще дльше продвинуться по пути созд ния утилиты ком ндной строки с помощью миним льного объем кода . П кет `fire` созд ет интерфейсы втом тически с помощью интроспекции кода . Если нужно сдел ть доступной в ком ндной строке простую функцию, можно просто вызв ть метод `fire.Fire`, ук з в ее в к честве аргумент :

```
#!/usr/bin/env python
"""
Простой пример использования библиотеки fire
"""
import fire
```

```
def greet(greeting='Hiya', name='Tammy'):
    print(f"{greeting} {name}")
```

```
if __name__ == '__main__':
    fire.Fire(greet)
```

Далее библиотек `fire` создан UI на основе названий и параметров методов:

```
$ ./simple_fire.py --help
```

```
NAME
    simple_fire.py
```

```
SYNOPSIS
    simple_fire.py <flags>
```

```
FLAGS
    --greeting=GREETING
    --name=NAME
```

В простых случаях можно в автоматическом режиме сделать доступными в командной строке несколько методов путем вызова `fire` без параметров:

```
#!/usr/bin/env python
"""
Простой пример использования fire
"""
import fire

def greet(greeting='Hiya', name='Tammy'):
    print(f"{greeting} {name}")

def goodbye(goodbye='Bye', name='Tammy'):
    print(f"{goodbye} {name}")

if __name__ == '__main__':
    fire.Fire()
```

`fire` автоматически берет из каждой функции команду и создает документацию для нее:

```
$ ./simple_fire.py --help
INFO: Showing help with the command 'simple_fire.py -- --help'.
```

```
NAME
    simple_fire.py
```

```
SYNOPSIS
    simple_fire.py GROUP | COMMAND
```

```
GROUPS
    GROUP is one of the following:
```

```
fire
    The Python fire module.
```

COMMANDS

COMMAND is one of the following:

```
greet
```

```
goodbye
```

(END)

Это очень удобно, если нужно р зобр ться в чужом коде или отл дить свой. Одной дополнительной строки код дост точно, чтобы получить возможность р бот ть со всеми функциями модуля из ком ндной строки. Весьм впеч тляюще. Поскольку библиотек fire определяет интерфейс н основе структуры с мой прогр ммы, то он сильнее связ н с не относящимся к интерфейсу кодом, чем `argparse` или `click`. Для моделиров ния интерфейс с вложенными ком нд ми необходимо опис ть кл ссы, отр ж ющие структуру требуемого интерфейс . Пример 3.6 иллюстрирует подобный подход.

Пример 3.6. fire_example.py

```
#!/usr/bin/env python
"""
Утилита командной строки, использующая библиотеку fire
"""
import fire

class Ships(): ❶
    def sail(self):
        ship_name = 'Your ship'
        print(f"{ship_name} is setting sail")

    def list(self):
        ships = ['John B', 'Yankee Clipper', 'Pequod']
        print(f"Ships: {' '.join(ships)}")

def sailors(greeting, name): ❷
    message = f'{greeting} {name}'
    print(message)

class Cli(): ❸
    def __init__(self):
        self.sailors = sailors
        self.ships = Ships()

if __name__ == '__main__':
    fire.Fire(Cli) ❹
```


- ❶ Описываем класс для команд `ships`.
- ❷ У `sailors` подкоманд нет, так что ее можно описать в виде функции.
- ❸ Описываем класс, который будет играть роль группы верхнего уровня. Добавляем в качестве трибутов этого класса функцию `sailors` и класс `Ships`.
- ❹ Вызываем метод `fire.Fire`, передавая ему класс, который будет играть роль группы верхнего уровня.

В итоге сгенерированной документации верхнего уровня класс `Ships` описан как группа, командой `sailors` — как командой:

```
$ ./fire_example.py

NAME
    fire_example.py

SYNOPSIS
    fire_example.py GROUP | COMMAND

GROUPS
    GROUP is one of the following:

        ships

COMMANDS
    COMMAND is one of the following:

        sailors
(END)
```

Всправке по группе `ships` показаны команды, соответствующие методам класс `Ships`:

```
$ ./fire_example.py ships --help
INFO: Showing help with the command 'fire_example.py ships -- --help'.

NAME
    fire_example.py ships

SYNOPSIS
    fire_example.py ships COMMAND

COMMANDS
    COMMAND is one of the following:

        list

        sail
(END)
```

Параметры функции `sailors` превратились в позиционные аргументы:

```
$ ./fire_example.py sailors --help
INFO: Showing help with the command 'fire_example.py sailors -- --help'.
```

```
NAME
  fire_example.py sailors

SYNOPSIS
  fire_example.py sailors GREETING NAME
```

```
POSITIONAL ARGUMENTS
  GREETING
  NAME
```

```
NOTES
  You can also use flags syntax for POSITIONAL ARGUMENTS
(END)
```

Как и ожидалось, теперь можно вызывать команды и подкоманды:

```
$ ./fire_example.py ships sail
Your ship is setting sail
$ ./fire_example.py ships list
Ships: John B,Yankee Clipper,Pequod
$ ./fire_example.py sailors Hiya Karl
Hiya Karl
```

Замечательная возможность `fire` — легкий переход в интерактивный режим. Если использовать флаг `--interactive`, `fire` открывает командную оболочку IPython, в которой доступны объект и функции из текущего скрипта:

```
$ ./fire_example.py sailors Hiya Karl -- --interactive
Hiya Karl
Fire is starting a Python REPL with the following objects:
Modules: fire
Objects: Cli, Ships, component, fire_example.py, result, sailors, self, trace
```

```
Python 3.7.0 (default, Sep 23 2018, 09:47:03)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.5.0 -- An enhanced Interactive Python. Type '?' for help.
```

```
-----
In [1]: sailors
Out[1]: <function __main__.sailors(greeting, name)>
```

```
In [2]: sailors('hello', 'fred')
hello fred
```

Здесь мы запустили команду `sailors` и шей прогнали для судоходства в интерактивном режиме. Открывается командная оболочка IPython, в которой у вас есть доступ к функции `sailors`. Этот интерактивный режим в сочетании

с простотой предоставления доступ к объектам из командной строки при использовании `fire` делает ее идеальным инструментом как для отладки, так и для знакомства с новым кодом.

Теперь вы видели весь спектр библиотек создания утилит командной строки, начиная с требующей большого объема ручного труда `argparse` до более лаконичной `click` и, наконец, минималистичной `fire`. К какой же использовать? Мы рекомендуем в большинстве случаев применять `click`, сочетая простоту и широкие возможности контроля. Для сложных интерфейсов, где желательно отделить код UI от бизнес-логики, лучше подойдет `argparse`. А для быстрого доступа к коду, у которого нет интерфейса командной строки, идеальна `fire`.

Реализация плагинов

После реализации пользовательского интерфейса командной строки для приложения имеет смысл задуматься о системе плагинов. Плагины — создаваемые пользователями дополнения к программе, расширяющие ее функциональность. Системы плагинов можно найти в самых разнообразных приложениях, от больших, например Maya от Autodesk, до минималистичных веб-фреймворков, таких как Flask. Вы можете написать утилиту для обхода файловой системы и предоставить пользователям возможность создавать плагины для робота с ее (файловой системы) содержимым. Главная составляющая любой системы плагинов — обновление последних. Программист должен знать о доступных для загрузки и выполнения плагинов. В примере 3.7 мы напишем простое приложение для обновления и запуска плагинов. В нем для поиска, загрузки и запуска плагинов применяется универсальный пользовательский префикс.

Пример 3.7. `simple_plugins.py`

```
#!/usr/bin/env python
import fire
import pkgutil
import importlib

def find_and_run_plugins(plugin_prefix):
    plugins = {}

    # Обнаружение и загрузка плагинов
    print(f"Discovering plugins with prefix: {plugin_prefix}")
    for _, name, _ in pkgutil.iter_modules(): ❶
        if name.startswith(plugin_prefix): ❷
            module = importlib.import_module(name) ❸
            plugins[name] = module

    # Запуск плагинов
    for name, module in plugins.items():
```

```

    print(f"Running plugin {name}")
    module.run() ❷

if __name__ == '__main__':
    fire.Fire()

```

❶ `pkgutil.iter_modules` возвращает все доступные для текущего `sys.path` модули.

❷ Проверяем, начинается ли название модуля с интересующего нас префикса.

❸ Используем `importlib` для загрузки модуля, сохраняя его в объекте `dict` для дальнейшего использования.

❹ Вызываем метод `run` плагина.

Подключение дополнительных плагинов для примера 3.7 требует от пользователей модулей всего лишь указать префикс в названии и доступ к их функциональности с помощью метода `run`. Если написать файл с префиксом `foo_plugin` со своими методами `run`:

```

def run():
    print("Running plugin A")

def run():
    print("Running plugin B")

```

можно будет обнружить и запустить их с помощью нашей системы рботы с плагинами:

```

$ ./simple_plugins.py find_and_run_plugins foo_plugin
Running plugin foo_plugin_a
Running plugin A
Running plugin foo_plugin_b
Running plugin B

```

Этот простой пример можно легко расширить, создавая в своих приложениях системы плагинов.

Ситуационный анализ: разгоняем Python с помощью утилит командной строки

Писать код сейчас удобно как никогда: всего несколько строк кода способны очень на многое. Одной-единственной функцией достаточно для удивительных вещей. Благодаря GPU, машинному обучению, облачным сервисам и Python можно легко создавать разгоняемые утилиты командной строки. Это все реально

что перевести код с простого движка внутреннего хранения неактивных движков. Достаточно одной функции, фрагмент кода с нужной логикой и, наконец, декоратор, чтобы использовать все это из командной строки.

Написание и сопровождение обычных приложений с GUI — веб- или традиционных — поистине сизифов труд. Начнется оно с сомнительных мерений, но быстро превращается в унылые трудоемкие мыслительные процессы, и в итоге вы не узнаете, почему вообще решили, что программист — такая хорошая профессия. Почему вы боретесь с этой утилитой устаревшей веб-фреймворк, по существу втомизирующей технологию 1970-х — реляционную базу данных — посредством наборов сценариев Python? Технологии древнего Ford Pinto со склонным к взрывам топливным баком новее, чем ваш веб-фреймворк. Наверняка есть лучший способ заставить себя не хлебнуть славом.

Ответ прост: перестаньте писать веб-приложения и создавайте вместо них «реактивные» утилиты командной строки. Обсуждаемые в следующих разделах утилиты командной строки направлены на получение быстрых результатов с помощью минимального объема кода. В числе их возможностей — обучение нейронных (машинное обучение), повышение быстродействия кода в 2000 раз и, самое замечательное, генерация цветного вывода в терминале.

Вот исходные ингредиенты приведенных далее программных решений:

- фреймворк Click;
- фреймворк CUDA язык Python;
- фреймворк Numba;
- фреймворк машинного обучения Scikit-learn.

Динамический компилятор Numba

Python печально известен своим низким быстродействием, поскольку по своей природе это язык сценариев. Один из способов обойти эту проблему — использовать динамический (Just-in-Time, JIT) компилятор Numba. Взглянем, как выглядит соответствующий код¹.

Во-первых, воспользуемся декоратором для хронометража, чтобы оценить время выполнения наших функций²:

¹ Полностью код можно найти на сайте https://github.com/noahgift/nuclear_powered_command_line_tools/blob/master/nuclearcli.py. — *Примеч. пер.*

² Не забудьте импортировать wraps из пакета functools: `from functools import wraps`. — *Примеч. пер.*

```
def timing(f):
    @wraps(f)
    def wrap(*args, **kwargs):
        ts = time()
        result = f(*args, **kwargs)
        te = time()
        print(f"fun: {f.__name__}, args: [{args}, {kwargs}] took: {te-ts} sec")
        return result
    return wrap
```

Далее добавим¹ декоратор `numba.jit` с ключевым аргументом `nopython=True`. Благодаря этому код будет выполняться JIT, а не обычным Python:

```
@timing
@numba.jit(nopython=True)
def expmean_jit(reas):
    """Вычисляет средние значения"""

    val = reas.mean() ** 2
    return val
```

При его запуске вы увидите как `jit`, так и обычную версию, запущенные посредством утилиты командной строки:

```
$ python nuclearcli.py jit-test
Running NO JIT
func:'expmean' args:[(array([[1.0000e+00, 4.2080e+05, 2350e+05, ...,
                             1.0543e+06, 1.0485e+06, 1.0444e+06],
                             [2.0000e+00, 5.4240e+05, 5.4670e+05, ...,
                             1.5158e+06, 1.5199e+06, 1.5253e+06],
                             [3.0000e+00, 7.0900e+04, 7.1200e+04, ...,
                             1.1380e+05, 1.1350e+05, 1.1330e+05],
                             ...,
                             [1.5277e+04, 9.8900e+04, 9.8100e+04, ...,
                             2.1980e+05, 2.2000e+05, 2.2040e+05],
                             [1.5280e+04, 8.6700e+04, 8.7500e+04, ...,
                             1.9070e+05, 1.9230e+05, 1.9360e+05],
                             [1.5281e+04, 2.5350e+05, 2.5400e+05, ..., 7.8360e+05, 7.7950e+05,
                             7.7420e+05]], dtype=float32),), {}] took: 0.0007 sec
$ python nuclearcli.py jit-test --jit
Running with JIT
func:'expmean_jit' args:[(array([[1.0000e+00, 4.2080e+05, 4.2350e+05, ...,
                             0543e+06, 1.0485e+06, 1.0444e+06],
                             [2.0000e+00, 5.4240e+05, 5.4670e+05, ..., 1.5158e+06, 1.5199e+06,
                             1.5253e+06],
                             [3.0000e+00, 7.0900e+04, 7.1200e+04, ..., 1.1380e+05, 1.1350e+05,
                             1.1330e+05],
                             ...,
```

¹ Не забывайте перед этим установить пакет `numba`, для этого импортируйте его в коде. — Примеч. пер.

```
[1.5277e+04, 9.8900e+04, 9.8100e+04, ..., 2.1980e+05, 2.2000e+05,
2.2040e+05],
[1.5280e+04, 8.6700e+04, 8.7500e+04, ..., 1.9070e+05, 1.9230e+05,
1.9360e+05],
[1.5281e+04, 2.5350e+05, 2.5400e+05, ..., 7.8360e+05, 7.7950e+05,
@click.option('--jit/--no-jit', default=False)
7.7420e+05]], dtype=float32),), {}] took: 0.2180 sec
```

Как этот код работает? Этот простой переключатель требует всего лишь нескольких строк кода :

```
@cli.command()
def jit_test(jit):
    rea = real_estate_array()
    if jit:
        click.echo(click.style('Running with JIT', fg='green'))
        expmean_jit(rea)
    else:
        click.echo(click.style('Running NO JIT', fg='red'))
        expmean(rea)
```

В некоторых случаях JIT-версия может ускорить выполнение кода в тысячи раз, но определять это нужно путем тестирования. Кроме того, стоит обратить внимание на следующую строку:

```
click.echo(click.style('Running with JIT', fg='green'))
```

Этот сценарий позволяет выводить в терминал текст различных цветов, что может оказаться очень удобно для сложных утилит.

Использование GPU с помощью CUDA Python

Еще один способ существенно ускорить выполнение кода — запустить его непосредственно на GPU. Для следующего примера мы получим компьютер с поддержкой CUDA. Код выглядит следующим образом:

```
@cli.command()
def cuda_operation():
    """Выполняет векторизованные операции на GPU"""

    x = real_estate_array()
    y = real_estate_array()

    print("Moving calculations to GPU memory")
    x_device = cuda.to_device(x)
    y_device = cuda.to_device(y)
    out_device = cuda.device_array(
        shape=(x_device.shape[0], x_device.shape[1]), dtype=np.float32)
    print(x_device)
```

```

print(x_device.shape)
print(x_device.dtype)

print("Calculating on GPU")
add_ufunc(x_device, y_device, out=out_device)

out_host = out_device.copy_to_host()
print(f"Calculations from GPU {out_host}")

```

Имеет смысл отметить, что если вы захотите переместить массив NumPy на GPU, то векторизованная функция выполняет вычисления на GPU. А после завершения задания данные перемещаются обратно из GPU. Использование GPU может радикально улучшить выполнение кода в зависимости от того, какие именно вычисления производятся. Вот результаты выполнения утилиты командной строки:

```

$ python nuclearcli.py cuda-operation
Moving calculations to GPU memory
<numba.cuda.cudadrv.devicearray.DeviceNDArray object at 0x7f01bf6ccac8>
(10015, 259)
float32
Calculating on GPU
Calculations from GPU [
[2.0000e+00 8.4160e+05 8.4700e+05 ... 2.1086e+06 2.0970e+06 2.0888e+06]
[4.0000e+00 1.0848e+06 1.0934e+06 ... 3.0316e+06 3.0398e+06 3.0506e+06]
[6.0000e+00 1.4180e+05 1.4240e+05 ... 2.2760e+05 2.2700e+05 2.2660e+05]
...
[3.0554e+04 1.9780e+05 1.9620e+05 ... 4.3960e+05 4.4000e+05 4.4080e+05]
[3.0560e+04 1.7340e+05 1.7500e+05 ... 3.8140e+05 3.8460e+05 3.8720e+05]
[3.0562e+04 5.0700e+05 5.0800e+05 ... 1.5672e+06 1.5590e+06 1.5484e+06]
]

```

Многоядерное многопоточное выполнение кода Python с помощью Numba

Одной из часто возникающих проблем с производительностью Python является отсутствие подлинной многопоточности. Ее тоже можно решить с помощью Numba. Вот пример некоторых простейших операций:

```

@timing
@numba.jit(parallel=True)
def add_sum_threaded(rearr):
    """Использует все ядра процессора"""

    x, _ = rearr.shape
    total = 0
    for _ in numba.prange(x):
        total += rearr.sum()
    print(total)

```



```

@timing
def add_sum(rear):
    """Обычный цикл for"""

    x,_ = rear.shape
    total = 0
    for _ in numba.prange(x):
        total += rear.sum()
    print(total)

@cli.command()
@click.option('--threads/--no-jit', default=False)
def thread_test(threads):
    rear = real_estate_array()
    if threads:
        click.echo(click.style('Running with multicore threads', fg='green'))
        add_sum_threaded(rear)
    else:
        click.echo(click.style('Running NO THREADS', fg='red'))
        add_sum(rear)

```

Обратите внимание на ключевое отличие в скрипте параллельной версии: потоки выполнения для итераций порождаются с помощью декоратора `@numba.jit(parallel=True)` и оператор `numba.prange`. Как видно из рис. 3.1, все ядра CPU на машине используются практически на 100%, когда практически тот же код выполняется без параллелизации, действует только одно ядро:

```

$ python nuclearcli.py thread-test
$ python nuclearcli.py thread-test --threads

```

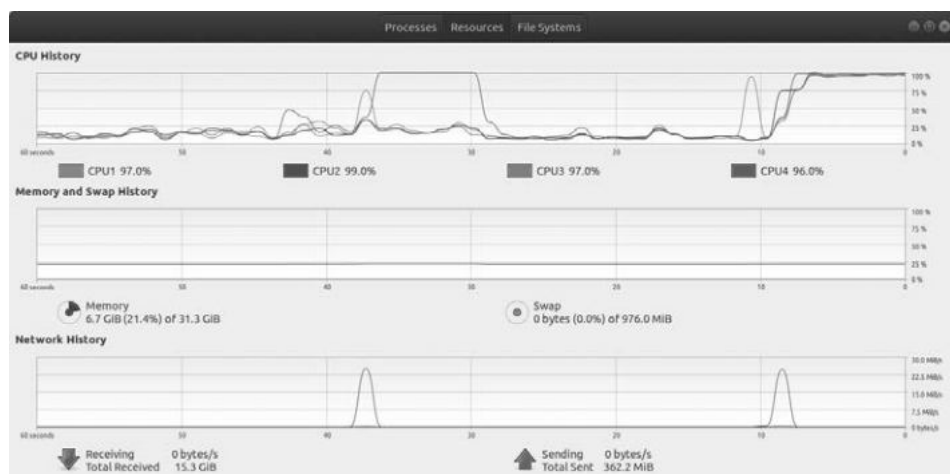


Рис. 3.1. Использование всех ядер CPU

Кластеризация методом k-средних

С помощью утилит командной строки можно реализовать и еще одну замечательную вещь — машинное обучение. В приведенном далее примере нам потребовалось всего несколько строк кода, чтобы создать функцию для кластеризации методом k-средних. В данном случае мы кластеризуем DataFrame Pandas и три кластера (по умолчанию):

```
def kmeans_cluster_housing(clusters=3):
    """Кластеризация DataFrame методом k-средних"""
    url = "https://raw.githubusercontent.com/noahgift/\
socialpower/nba/master/data/nba_2017_att_val_elo_win_housing.csv"
    val_housing_win_df = pd.read_csv(url)
    numerical_df = (
        val_housing_win_df.loc[:, ["TOTAL_ATTENDANCE_MILLIONS", "ELO",
                                   "VALUE_MILLIONS", "MEDIAN_HOME_PRICE_COUNTY_MILLIONS"]]
    )
    # Нормирование данных
    scaler = MinMaxScaler()
    scaler.fit(numerical_df)
    scaler.transform(numerical_df)
    # Кластеризация данных
    k_means = KMeans(n_clusters=clusters)
    kmeans = k_means.fit(scaler.transform(numerical_df))
    val_housing_win_df['cluster'] = kmeans.labels_
    return val_housing_win_df
```

Количество кластеров можно изменить на другое, передвигая нужный параметр с помощью click:

```
@cli.command()
@click.option("--num", default=3, help="number of clusters")
def cluster(num):
    df = kmeans_cluster_housing(clusters=num)
    click.echo("Clustered DataFrame")
    click.echo(df.head())
```

В итоге получаем кластеризованный DataFrame Pandas. Обратите внимание на наличие в нем столбца, отражающего распределение по кластерам:

```
$ python -W nuclearcli.py cluster
Clustered DataFrame
   TEAM   GMS  ...   COUNTY  cluster
0  Chicago Bulls  41  ...   Cook      0
1  Dallas Mavericks  41  ...  Dallas      0
2  Sacramento Kings  41  ... Sacramento  1
3   Miami Heat  41  ...  Miami-Dade      0
4  Toronto Raptors  41  ... York-County      0
```

```
[5 rows x 12 columns]
```

```
$ python -W nuclearcli.py cluster --num 2
```

```
Clustered DataFrame
```

	TEAM	GMS	...	COUNTY	cluster
0	Chicago Bulls	41	...	Cook	1
1	Dallas Mavericks	41	...	Dallas	1
2	Sacramento Kings	41	...	Sacramento	0
3	Miami Heat	41	...	Miami-Dade	1
4	Toronto Raptors	41	...	York-County	1

```
[5 rows x 12 columns]
```

Упражнения

- Напишите с помощью `sys` скрипт, который выводит текст «командная строка» только тогда, когда запущен из командной строки.
- Создайте с помощью библиотеки `click` утилиту командной строки, принимающую в качестве аргумента значение и выводящую его в случае, если оно не начинается с символа `p`.
- Воспользуйтесь `fire` для обертки метода в уже существующем скрипте Python из командной строки.

ГЛАВА 4

Полезные утилиты Linux

Команды и строки и ее утилиты так понравились Альфредо, что он быстро привязался к серверам под управлением Linux. Одним из первых его должностей — системный администратор в не очень крупной компании, где он отвечал за все относящееся к Linux. Небольшой IT-отдел, в котором в основном серверами и рабочими станциями под управлением Windows, всеми фибрами души не видел командную строку. В какой-то момент руководитель IT-подразделения сказал Альфредо, что понимает, как решать поставленные задачи только с помощью графических интерфейсов (GUI), утилит и общего инструментария: «Я не программист, если у программы нет GUI, я не могу ею пользоваться».

Альфредо наняли по договору подряда, чтобы он помог начать строить несколько имеющихся у компании серверов Linux. В то время в качестве системы контроля версий был особенно популярен Subversion (SVN), и все разработчики компании отправляли свою работу на один сервер SVN. Вместо централизованного сервера вторично, предоставляемого двумя *контроллер-ми доменов*, он использовал текстовую систему аутентификации, в которой пользователи ставились в соответствие хешам паролей. А значит, имена пользователей не обязательно соответствовали именам из контроллер-доменов и пароли могли быть произвольными. Значит разработчики пришивали восстановление пароля, и кому-то приходилось редактировать текстовый файл с хешами. Руководитель проекта попросил Альфредо интегрировать SVN-аутентификацию с контроллером домена (Active Directory от компании Microsoft). Первый вопрос, который Альфредо задал: «А почему сотрудники IT-подразделения до сих пор этого не сделали?» Ответ прозвучал: «Они говорят, что это невозможно, но, Альфредо, это верно, SVN можно интегрировать с Active Directory».

Он никогда раньше не использовал сервисы аутентификации и подобие Active Directory и очень плохо понимал, как работает SVN, но был настроен выполнить поставленную задачу. Альфредо начал целю прочитывать все, что только воз-

можно, про SVN и Active Directory, зпустил виртуальную машину с сервером SVN и попытлся добиться работы этой схемы аутентификации. Чтение необходимой документации и настройка заняли почти две недели. В конце концов он добился того, чего хотел, и сумел запустить эту систему в промышленную эксплуатацию. Ощущения были невероятные: он получил уникальные знания и был готов полностью ответить за систему. Руководитель ИТ, к которому весь последний отдел, были втресе. Альфредо попытлся поделиться вновь обретенными познаниями с остальными, но постоянно наткнулся на отговорки: «нет времени», «очень заняты», «другие приоритеты» и «возможно, когда-нибудь потом — может, на следующей неделе».

Ниболее удачное определение технических специалистов — *информационные роботники* (knowledge worker). Любознательность и непрерывная тяга к знаниям совершенствуют и в нас, и среду, над которой мы работаем. Никогда не позволяйте коллегам (и даже целому ИТ-подразделению, как в случае с Альфредо) препятствовать усовершенствованию систем. Хотите заставить себя использовать возможность выучить что-то новое! В худшем случае обретенные знания редко будут вам нужны, но в лучшем могут полностью изменить вашу профессиональную жизнь.

В Linux есть и графические среды, но по-настоящему скрыть его возможности можно лишь понимая и используя командную строку, в перспективе расширяя ее. Бывшие специалисты DevOps при отсутствии готовых утилит для решения какой-либо задачи создают свои собственные. Решение поставленных задач путем комбинирования базовых элементов открывает огромные перспективы, и именно это ощутил Альфредо, когда сумел решить проблему своими силами, не уступив готовое программное обеспечение.

В этой главе мы рассмотрим несколько простых и интересных проектов с командной оболочкой, включая несколько удобных команд Python, расширяющих возможности взаимодействия с машиной. Создание псевдонимов и *однострочных сценариев* — одни из самых увлекательных сторон нашей работы, иногда они оказываются настолько удобными, что становятся привычными или восточными элементами программного обеспечения.

Дисковые утилиты

Существует несколько личных утилит для получения информации о системных устройствах. Возможности многих из них пересекаются между собой, некоторые, например `fdisk` и `parted`, позволяют производить операции над диском в ходе интерактивного сеанса.

Необходимо хорошо разбираться в работе дисковых утилит не только для извлечения информации и выполнения операций с разделом диска, но и для точной оценки быстродействия. А правильный ответ на вопрос «Как измерить быстродействие устройств?» звучит так: «Зависит от обстоятельств», поскольку очень сложно найти единый подход ко всем случаям жизни.

Измерение быстродействия

В ходе работы в изолированной среде с сервером без доступа в Интернет либо на сервере, где у нас нет привычных инструментов, правильный ответ на этот вопрос можно было бы получить с помощью утилиты `dd` (доступной в большинстве основных дистрибутивов Linux). При возможности имеет смысл использовать ее совместно с `iostat`¹, чтобы отделить команду, использующую ресурсы шины, от команды, просто возвращающей отчет.

Как однажды сказал мне опытный специалист по производительности, все зависит от того, что измеряется и как. Например, утилита `dd` — однопоточная и возможности ее ограничены, в частности, она не умеет выполнять множественные операции случайного чтения и записи. Кроме того, она измеряет пропускную способность, а не количество операций ввода/вывода в секунду (IOPS). Так что же вы хотите измерить, пропускную способность или IOPS?



Осторожнее с этими примерами: они могут уничтожить вашу систему. Так что не следуйте им вслепую и используйте только те устройства, которые можно спокойно очистить².

Следующая простая однострочная команда получит с помощью команды `dd` определенную статистику по новому устройству (в данном случае `/dev/sdc`):

```
$ dd if=/dev/zero of=/dev/sdc count=10 bs=100M
10+0 records in
10+0 records out
1048576000 bytes (1.0 GB, 1000 MiB) copied, 1.01127 s, 1.0 GB/s
```

Он делает десять записей по 100 Мбайт со скоростью 1 Гбайт/с, измеряя тем самым пропускную способность. Простейший способ получить значение IOPS

¹ Возможно, в будущем придется обновить эту утилиту, например, с помощью команды `sudo apt install sysstat`. — *Примеч. пер.*

² Недобрым знаком команд `dd` иногда в шутку называют «добей диск». — *Примеч. пер.*

с помощью `dd` — воспользоваться `iostat`. В следующем примере `iostat` выполняется только на устройстве, описанном в `dd`, причем флаг `-d` означает лишь получение информации об устройстве с интервалом 1 секунд:

```
$ iostat -d /dev/sdc 1
```

Device	tps	kB_read/s	kB_wrtn/s	kB_read	kB_wrtn
sdc	6813.00	0.00	1498640.00	0	1498640

Device	tps	kB_read/s	kB_wrtn/s	kB_read	kB_wrtn
sdc	6711.00	0.00	1476420.00	0	1476420

Утилита `iostat` повторяет выводимую информацию ежесекундно, пока вы не прервете операцию, нажав `Ctrl+C`. Второй столбец в выводимой информации — `tps` (transactions per second — транзакций в секунду) — то же самое, что и IOPS. Можно более удобным способом визуализировать выводимую информацию, избавившись от всей этой мешины повторов. Для этого достаточно очистить терминал при каждом вызове:

```
$ while true; do clear && iostat -d /dev/sdc && sleep 1; done
```

Повышаем точность измерения с помощью `fio`

Если `dd` и `iostat` недостаточны, чаще всего для измерения быстродействия используют `fio`. С ее помощью можно выяснить поведение в смысле быстродействия в среде с высокой нагрузкой по чтению и/или записи (и даже выбрать процентное соотношение операций чтения/записи).

`fio` выводит довольно много информации. В примере далее мы немного сократили ее, чтобы подчеркнуть полученные значения IOPS для операций чтения и записи:

```
$ fio --name=sdc-performance --filename=/dev/sdc --ioengine=libaio \
  --iodepth=1 --rw=randrw --bs=32k --direct=0 --size=64m
sdc-performance: (g=0): rw=randwrite, bs=(R) 32.0KiB-32.0KiB,
(W) 32.0KiB-32.0KiB, (T) 32.0KiB-32.0KiB, ioengine=libaio, iodepth=1
fio-3.1
Starting 1 process
sdc-performance: (groupid=0, jobs=1): err= 0: pid=2879:
  read: IOPS=1753, BW=54.8MiB/s (57.4MB/s)(31.1MiB/567msec)
...
  iops      : min= 1718, max= 1718, avg=1718.00, stdev= 0.00, samples=1
  write: IOPS=1858, BW=58.1MiB/s (60.9MB/s)(32.9MiB/567msec)
...
  iops      : min= 1824, max= 1824, avg=1824.00, stdev= 0.00, samples=1
```

С помощью `fl` гов в этом примере мы н зыв ем з *д ние* `sd-c-performance`, ук зыв ем непосредственно н устройство `/dev/sdc` (для этого необходимы пр в доступ суперпользов теля), з действуем н тивную библиотеку Linux синхронного ввод /вывод , уст н влив ем п р метру `iodpth` зн чение `1` (количество отпр вляемых з один р з последов тельных з просов ввод /вывод) и з д ем р змер буфер `32 Кб` йт для буферизов нного ввод /вывод при опер циях чтения/з писи (для небуферизов нного ввод /вывод необходимо уст новить зн чение `1`) для `64-мег б` йтного ф йл . Очень длинн я ком нд получил сь!

Утилита `fiop` считывает огромное количество дополнительных опций, достаточных практически для любого случая, когда требуется точное измерение IOPS. Например, она способна тестировать сразу несколько устройств, также выполнять «прогревочные» операции ввода/вывода и задавать пороговые значения при тестировании ввода/вывода, если существуют определенные предельные значения, которые нельзя превышать. Наконец, множество опций командной строки можно настроить через `INI`-файлы, что позволяет описывать изящные сценарии выполнения заданий.

Разделы диска

Р-зделы диска обычно создаются с помощью утилиты **fdisk** с ее интерактивными меню, но в некоторых случаях он подходит плохо, например, если р-зделы большие (2 Тб и более). В этих случаях в качестве альтернативы можно воспользоваться **parted**.

Следующий краткий пример интерактивного сеанса демонстрирует создание с помощью утилиты `fdisk` основного раздела диска с начальным значением по умолчанию и размером 4 Гб. В конце нажимается клавиша `w` для записи изменений на диск:

```
$ sudo fdisk /dev/sds
```

```
Command (m for help): n
```

Partition type:

p primary (0 primary, 0 extended, 4 free)

e extended

Select (default p): p

Partition number (1-4, default 1):

First sector (2048-22527999, default 2048):

Using default value 2048

Last sector, +sectors or +size{K,M,G} (2048-22527999, default 22527999): +4G

Partition 1 of type Linux and of size 4 GiB is set


```
Command (m for help): w
The partition table has been altered!
```

```
Calling ioctl() to re-read partition table.
Syncing disks.
```

Утилита `parted` позволяет получить тот же результат, но ее интерфейс иной:

```
$ sudo parted /dev/sdaa
GNU Parted 3.1
Using /dev/sdaa
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) mklabel
New disk label type? gpt
(parted) mkpart
Partition name? []?
File system type? [ext2]?
Start? 0
End? 40%
```

Для выхода в конце необходимо нажать клавишу `q`. Создать разделы из командной строки без каких-либо интерактивных приглашений к вводу можно с помощью нескольких команд:

```
$ parted --script /dev/sdaa mklabel gpt
$ parted --script /dev/sdaa mkpart primary 1 40%
$ parted --script /dev/sdaa print
Disk /dev/sdaa: 11.5GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:
```

Number	Start	End	Size	File system	Name	Flags
1	1049kB	4614MB	4613MB			

Получение информации о конкретном устройстве

Для получения информации о каком-то конкретном устройстве иногда отлично подойдут утилиты `lsblk` или `blkid`. `fdisk` не захочет работать без прав суперпользователя. В следующем примере `fdisk` выводит информацию об устройстве `/dev/sda`:

```
$ fdisk -l /dev/sda
fdisk: cannot open /dev/sda: Permission denied

$ sudo fdisk -l /dev/sda
Disk /dev/sda: 42.9 GB, 42949672960 bytes, 83886080 sectors
Units = sectors of 1 * 512 = 512 bytes
```

```
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x0009d9ce
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1	*	2048	83886079	41942016	83	Linux

Утилита `blkid` схожа с `fdisk` в том, что требует привилегированного доступа суперпользователя:

```
$ blkid /dev/sda
```

```
$ sudo blkid /dev/sda
/dev/sda: PTTYPE="dos"
```

Утилита `lsblk` дает возможность получить ту же информацию, не имея повышенных полномочий:

```
$ lsblk /dev/sda
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sda    8:0    0 40G  0 disk
└─sda1  8:1    0 40G  0 part /
$ sudo lsblk /dev/sda
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sda    8:0    0 40G  0 disk
└─sda1  8:1    0 40G  0 part /
```

Утилита `blkid` с флагом `-p` для низкоуровневого исследования устройств делет рботу *очень тщательно* и предоставляет много информации об устройствах:

```
$ blkid -p /dev/sda1
UUID="8e4622c4-1066-4ea8-ab6c-9a19f626755c" TYPE="xfs" USAGE="filesystem"
PART_ENTRY_SCHEME="dos" PART_ENTRY_TYPE="0x83" PART_ENTRY_FLAGS="0x80"
PART_ENTRY_NUMBER="1" PART_ENTRY_OFFSET="2048" PART_ENTRY_SIZE="83884032"
```

Команда `lsblk` по умолчанию выводит некоторые интересные свойства устройств:

```
$ lsblk -P /dev/nvme0n1p1
NAME="nvme0n1p1" MAJ:MIN="259:1" RM="0" SIZE="512M" RO="0" TYPE="part"
```

Он позволяет и вызывать специальные флаги для запроса информации о конкретных свойствах:

```
lsblk -P -o SIZE /dev/nvme0n1p1
SIZE="512M"
```

Подобный доступ к свойствам очень упрощает написание сценариев и дальнейшее потребление данных со стороны Python.

Сетевые утилиты

Сетевые утилиты непрерывно совершенствуются по мере роста числ требующих соединения серверов. Часть этого процесса — описание и создание удобных однострочных сценариев, например, для SSH-туннелирования, часть — подробно рассмотреть способы измерения быстродействия сети, например, с помощью утилиты Apache Bench.

SSH-туннелирование

Вы пробовали когда-нибудь подключиться к HTTP-сервису, запущенному на удаленном сервере, доступном только через SSH? Подобная ситуация может возникнуть, когда включенный HTTP-сервис не должен быть общедоступным. Последний раз мы встречались с тем, когда проботли с планированием экземпляра RabbitMQ (<https://www.rabbitmq.com>), находящегося в промышленной эксплуатации, который запускался в виде HTTP-сервиса на порте 15672. Этот сервис не был открыт для всеобщего доступа, и не случайно, поскольку применялся он редко, при необходимости можно было воспользоваться SSH-туннелированием.

Для этого создается SSH-соединение с удаленным сервером, после чего удаленный порт (в данном случае 15672) переносится на локальный порт машины, запросившей соединение. На удаленной машине порт SSH был нестандартным, что несколько усложнило команду, которая выглядела вот так:

```
$ ssh -L 9998:localhost:15672 -p 2223 adeza@prod1.rabbitmq.ceph.internal -N
```

В ней можно видеть три флага, три числа и два адреса. Рассмотрим эту команду по частям, чтобы разобраться, что происходит. Флаг `-L` включает проброс портов и привязку локального порта (9998) к удаленному (по умолчанию для RabbitMQ — 15672). Флаг `-p` указывает нестандартный порт SSH удаленного сервера — 2223, далее указываются имя пользователя и адрес. В конце, флаг `-N` означает, что требуется только проброс портов, без выполнения удаленных команд.

Если выполнить эту команду правильно, она как бы висит, но при этом вы сможете, перейдя по адресу <http://localhost:9998/>, увидеть страницу входа для удаленного экземпляра RabbitMQ. Один из полезных флагов при туннелировании — `-f`: он переводит процесс в фоновый режим, что удобно для постоянного соединения, поскольку терминатор при этом готов для дальнейшей работы.

Оценка быстродействия HTTP с помощью Apache Benchmark (ab)

Мы просто *обож* ем н грузи ть серверы, с которыми р бот ем, з д ниями, чтобы убедиться в пр вильной обр ботке ими н грузки, особенно перед их переводом в промышленную эксплу т цию. Иногда мы д же пыт емся искусственно вызв ть к кое-нибудь редкое состояние гонки, возможное при высокой н грузке. Утилит Apache Benchmark (ab в ком ндной строке) — одн из тех крошечных утилит, которым требуется всего несколько фл гов, чтобы добиться нужных резуль ттов.

Следующ я ком нд выполняет по 100 з просов, всего 10 000, к лок льной м шине, н которой з пущен Nginx:

```
$ ab -c 100 -n 10000 http://localhost/
```

Подобный тест — довольно жестоко по отношению к ре льной системе, но речь идет всего лишь о лок льном сервере, з просы предст вляют собой просто HTTP GET. ab выводит очень подробный отчет, который выглядит следующим обр зом (мы немного сокр тили его):

```
Benchmarking localhost (be patient)
```

```
...
```

```
Completed 10000 requests
```

```
Finished 10000 requests
```

```
Server Software:      nginx/1.15.9
```

```
Server Hostname:      localhost
```

```
Server Port:          80
```

```
Document Path:        /
```

```
Document Length:      612 bytes
```

```
Concurrency Level:     100
```

```
Time taken for tests:   0.624 seconds
```

```
Complete requests:     10000
```

```
Failed requests:        0
```

```
Total transferred:     8540000 bytes
```

```
HTML transferred:      6120000 bytes
```

```
Requests per second:    16015.37 [#/sec] (mean)
```

```
Time per request:       6.244 [ms] (mean)
```

```
Time per request:       0.062 [ms] (mean, across all concurrent requests)
```

```
Transfer rate:          13356.57 [Kbytes/sec] received
```

```
Connection Times (ms)
```

	min	mean[+/-sd]	median	max
Connect:	0	3 0.6	3	5
Processing:	0	4 0.8	3	8
Waiting:	0	3 0.8	3	6
Total:	0	6 1.0	6	9

Подобная информация и способ ее представления умопомру чительны. С одного взгляд можно понять, откзывает ли нходящийся в промышленной эксплуатации сервер в соединениях (по полю `Failed requests`) и кковы средние покзтели. В днном случае использовались зпросы тип `GET`, но утилит `ab` позволяет здействовать и другие HTTP-«глголы», н пример `POST`, и дже выполнять `HEAD`-зпросы. Подобные утилиты следует применять осмотрительно, ведь с их помощью легко можно нгрузить сервер сильнее допустимого. Длее приведены более релистичные покзтели используемого в действительности HTTP-сервис :

```
...
Benchmarking prod1.ceph.internal (be patient)

Server Software:      nginx
Server Hostname:      prod1.ceph.internal
Server Port:          443
SSL/TLS Protocol:     TLSv1.2,ECDHE-RSA-AES256-GCM-SHA384,2048,256
Server Temp Key:      ECDH P-256 256 bits
TLS Server Name:      prod1.ceph.internal

Complete requests:    200
Failed requests:      0
Total transferred:    212600 bytes
HTML transferred:     175000 bytes
Requests per second:  83.94 [#/sec] (mean)
Time per request:     1191.324 [ms] (mean)
Time per request:     11.913 [ms] (mean, across all concurrent requests)
Transfer rate:        87.14 [Kbytes/sec] received

....
```

Теперь покзтели совсем другие, `ab` обрщется к сервису с подключенным SSL, поэтому выводит информцию о доступных протоколах. 83 зпрос в секунду предствляется не слишком хорошими хртеристиками, но речь идет о сервере API, генерирующем JSON, одномоментн янгрузк которого (н подобие только что сгенерированной) обычно не слишком высок.

Нагрузочное тестирование с помощью molotov

Проект Molotov (<https://molotov.readthedocs.io>) — интересное решение в сфере н -грузочного тестирования. Чсть его возможностей нлогичн возможностям Apache Benchmark, но это проект н языке Python, бл годря чему вы можете писать сценрии н Python и использовать модуль `asyncio`.

Вот т к выглядит простейший пример применения `molotov`:

```
import molotov

@molotov.scenario(100)
```

```
async def scenario_one(session):
    async with session.get("http://localhost:5000") as resp:
        assert resp.status == 200
```

Сохраните этот код в виде файла `load_test.py`, создайте минимальное приложение Flask для обработки запросов GET и POST по основному URL и сохраните его в виде файла `small.py`:

```
from flask import Flask, redirect, request

app = Flask('basic app')

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        return redirect('https://www.google.com/search?q=%s' % request.args['q'])
    else:
        return '<h1>GET request from Flask!</h1>'
```

Запустите приложение Flask с помощью команды `FLASK_APP=small.py flask run`, затем запустите `molotov`, передвемому созданный ранее файл `load_test.py`:

```
$ molotov -v -r 100 load_test.py
**** Molotov v1.6. Happy breaking! ****
Preparing 1 worker...
OK
SUCSESSES: 100 | FAILURES: 0 WORKERS: 0
*** Bye ***
```

В результате в одном процессе-исполнителе будет выполнено 100 запросов к локальному экземпляру Flask. Подлинными возможностями этой утилиты пользуются при большом объеме выполняемых в каждом запросе действий. В ней применяются подходы, типичные для логических модульных тестирований, в частности, созданию тестовой среды и ее использованию с последующей очисткой ресурсов, и даже код, способный регистрировать определенные события. А поскольку минимальное приложение Flask может обрабатывать запросы POST, перепроверяемые на поиск Google, добавим еще один вариант поведения в файл `load_test.py`. Пусть 100 % запросов будут типа POST:

```
@molotov.scenario(100)
async def scenario_post(session):
    resp = await session.post("http://localhost:5000", params={'q': 'devops'})
    redirect_status = resp.history[0].status
    error = "unexpected redirect status: %s" % redirect_status
    assert redirect_status == 301, error
```

Запустим этот новый вариант для выполнения одного-единственного запроса и увидим следующее:

```
$ molotov -v -r 1 --processes 1 load_test.py
**** Molotov v1.6. Happy breaking! ****
Preparing 1 worker...
OK
AssertionError('unexpected redirect status: 302',)
  File ".venv/lib/python3.6/site-packages/molotov/worker.py", line 206, in step
    **scenario['kw'])
  File "load_test.py", line 12, in scenario_two
    assert redirect_status == 301, error
SUCCESES: 0 | FAILURES: 1
*** Bye ***
```

Единственного из просов (флг -r 1) ок, а здесь достточно для того, чтобы все извершилось неудачей. Необходимо модифицировать оператор контроля для проверки состояния 302 вместо 301. После этого поменяйте соотношение изпросов POST и 80, чтобы в приложение Flask отправлялись и другие изпросы (GET). В результате файл должен выглядеть следующим образом:

```
import molotov

@molotov.scenario()
async def scenario_one(session):
    async with session.get("http://localhost:5000/") as resp:
        assert resp.status == 200

@molotov.scenario(80)
async def scenario_two(session):
    resp = await session.post("http://localhost:5000", params={'q': 'devops'})
    redirect_status = resp.history[0].status
    error = "unexpected redirect status: %s" % redirect_status
    assert redirect_status == 302, error
```

Изпустите load_test.py для выполнения десяти изпросов, два с помощью метода GET, остальные — с помощью POST:

```
127.0.0.1 - - [04/Sep/2019 12:10:54] "POST /?q=devops HTTP/1.1" 302 -
127.0.0.1 - - [04/Sep/2019 12:10:56] "POST /?q=devops HTTP/1.1" 302 -
127.0.0.1 - - [04/Sep/2019 12:10:57] "POST /?q=devops HTTP/1.1" 302 -
127.0.0.1 - - [04/Sep/2019 12:10:58] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [04/Sep/2019 12:10:58] "POST /?q=devops HTTP/1.1" 302 -
127.0.0.1 - - [04/Sep/2019 12:10:59] "POST /?q=devops HTTP/1.1" 302 -
127.0.0.1 - - [04/Sep/2019 12:11:00] "POST /?q=devops HTTP/1.1" 302 -
127.0.0.1 - - [04/Sep/2019 12:11:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [04/Sep/2019 12:11:01] "POST /?q=devops HTTP/1.1" 302 -
127.0.0.1 - - [04/Sep/2019 12:11:02] "POST /?q=devops HTTP/1.1" 302 -
```

Как видите, возможности molotov можно легко расширять с помощью чистого Python-кода и приспосабливать к прочим, более сложным потребностям. Эти примеры — лишь мала толика того, на что способен этот утилит.

Утилиты для получения информации о загрузке CPU

Две в жнейшие утилиты для получения информации о загрузке CPU — это `top` и `htop`. Утилит `top` включен в большинство современных дистрибутивов Linux, но если у вас есть возможность установить новую утилиту, настоятельно рекомендуем установить более совершенную утилиту `htop`, ее интерактивный интерфейс нравится многим гораздо больше интерфейса `top`. Существует еще несколько утилит для визуализации загрузки CPU и даже мониторинг, но ни одна из них не может похвастаться той полнотой и доступностью, как `top` и `htop`. Например, вполне можно получить информацию о загрузке CPU с помощью команды `ps`:

```
$ ps -eo pcpu,pid,user,args | sort -r | head -10
%CPU    PID  USER      COMMAND
 0.3     719  vagrant    -bash
 0.1     718  vagrant    sshd: vagrant@pts/0
 0.1     668  vagrant    /lib/systemd/systemd --user
 0.0         9  root      [rcu_bh]
 0.0     95  root      [ipv6_addrconf]
 0.0     91  root      [kworker/u4:3]
 0.0      8  root      [rcu_sched]
 0.0     89  root      [scsi_tmfs_1]
```

В этой команде `ps` мы указали несколько дополнительных полей. Первое из них — `pcpu` для вывода загрузки CPU в процент, за ним следуют идентификатор процесса, пользователь и, наконец, выполняемая команда. Далее мы добились в конвейере сортировку в обратном порядке, поскольку по умолчанию сначала выводятся процессы, наименее нагружающие CPU, нам хочется видеть сверху список команд, сильнее всего нагружающих CPU. Наконец, поскольку команда `ps` отображает эту информацию для всех процессов, мы отфильтровали в нем десять первых результатов с помощью команды `head`.

Однако данная команда довольно сложна в применении, запомнить все ее параметры непросто, кроме того, она не обновляет результаты динамически. Утилиты `top` и `htop` гораздо удобнее. Как вы увидите, обе они обладают очень широкими возможностями.

Просмотр процессов с помощью htop

Утилит `htop` напоминает `top` (интерактивную программу просмотра процессов), однако является полностью кросс-платформенной (работает на OS X, FreeBSD, OpenBSD и Linux), обладает более широкими возможностями визуализации (рис. 4.1) и просто приятна в использовании. Можете взглянуть на пример снимка

экран для запуска на сервере утилиты htop по адресу <https://hisham.hm/htop>. Один из главных недостатков htop — несовместимость со всеми сокращенными формами вызовов top, так что вам придется заново запомнить их для нее.

CPU[] Tasks: 30, 60 thr: 1 running									
Mem[] Load average: 0.09 0.17 0.09									
Swp[] Uptime: 00:03:51									
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%
3828	vagrant	20	0	22472	2357	1504	R	0.7	0.1
2683		20	0	525M	46520	12668	S	0.7	2.5
2771		20	0	425M	31488	13640	S	0.7	1.7
3099		20	0	525M	46520	12668	S	0.0	2.5
3084		20	0	525M	46520	12668	S	0.0	2.5
2796		20	0	425M	31488	13640	S	0.0	1.7
2689		20	0	213M	4436	3144	S	0.0	0.2
2686		20	0	425M	31488	13640	S	0.0	1.7
3098		20	0	525M	46520	12668	S	0.0	2.5
1086		20	0	39184	3236	2816	S	0.0	0.2
1		20	0	109M	6740	4172	S	0.0	0.4
3675	vagrant	20	0	150M	2288	976	S	0.0	0.1
2776		20	0	425M	31488	13640	S	0.0	1.7
3816		20	0	462M	19148	6628	S	0.0	1.0
3078		20	0	525M	46520	12668	S	0.0	2.5
3096		20	0	525M	46520	12668	S	0.0	2.5
1126		20	0	47688	5128	2888	S	0.0	0.3
2803		20	0	425M	31488	13640	S	0.0	1.7
3089		20	0	525M	46520	12668	S	0.0	2.5
2695		20	0	213M	4436	3144	S	0.0	0.2
2680		20	0	462M	19148	6628	S	0.0	1.0
1196		20	0	196M	4188	2024	S	0.0	0.2
2185		16	-4	62044	1084	496	S	0.0	0.1
2164		16	-4	62044	1084	496	S	0.0	0.1
2225		20	0	8576	824	676	S	0.0	0.0
2226		20	0	26376	1748	1448	S	0.0	0.1
2242		20	0	597M	12304	4812	S	0.0	0.7
2254		20	0	597M	12304	4812	S	0.0	0.7
2256		20	0	597M	12304	4812	S	0.0	0.7
2257		20	0	597M	12304	4812	S	0.0	0.7
2258		20	0	597M	12304	4812	S	0.0	0.7
2260		20	0	597M	12304	4812	S	0.0	0.7
2269		20	0	597M	12304	4812	S	0.0	0.7
2273		20	0	73648	1376	672	S	0.0	0.1

Рис. 4.1. Запущенная на сервере утилита htop

Отображенный на рис. 4.1 информация выглядит совершенно иначе и вызывает абсолютно другие ощущения. Загрузка CPU, оперативная память и объем подкачки красным отображаются слева вверху и меняются по мере изменения состояния системы. С помощью клавиш стрелок можно прокручивать отображаемую информацию вверх-вниз и даже влево-вправо, просматривая команду запуска процесса полностью.

Хотите прервать выполнение какого-либо процесса? Перейдите к нему с помощью клавиш стрелок или нажмите / для последовательного поиска (и фильтрации) процессов, после чего нажмите k. В новом меню отображаются все сигналы, которые можно отправить процессу, например SIGTERM вместо SIGKILL. Можно также выбрать более одного процесса для прерывания выполнения. Нажмите пробел, чтобы выбрать нужные процессы, при этом они будут выделены другим цветом. Ошиблись и хотите отменить выбор? Снова нажмите пробел. Интуитивно очень понятный интерфейс.

Единственная проблема `htop` — многие операции привязаны к клавишам F, они есть не во всех клавиатурных. Например, F1 — справка. Впрочем, можно использовать и другие клавиши. Например, для доступа к меню справки можно нажать клавишу H, для строк — Shift+S вместо F2.

Клавиша T (тоже интуитивно!) меняет отображение списка процессов и древовидное (tree). Вероятно, наиболее часто используемая функциональность — сортировка. Нажмите > — и слева появится меню для выбора типа сортировки: по PID, пользователю, действующей памяти, приоритету и проценту загрузки CPU — вот лишь часть доступных вариантов. Существуют также сокращенные формы вызовов для сортировки без открытия меню: по используемой памяти (Shift+I), загрузке CPU (Shift+P) и времени (Shift+T).

Наконец, еще две потрясающие возможности: можно запустить утилиты `strace` или `lsof` для выбранного процесса, если они установлены в системе и у пользователя достаточно прав для их запуска. Если какие-либо процессы требуют полномочий суперпользователя, `htop` сообщит об этом и потребует `sudo` для выполнения от имени суперпользователя. Запустить `strace` для выбранного процесса можно с помощью клавиши S, `lsof` — с помощью клавиши L.

Используя `strace` или `lsof`, можно выполнять поиск и фильтрацию с помощью символа /. Невероятно удобная утилита! Надеюсь, когда-нибудь будут добавлены и другие варианты вызова без клавиш F, хотя и сейчас можно сделать практически все, что нужно, с помощью латеральных вариантов в рядах.



Пользовательские настройки `htop`, заданные в ходе интерактивного сеанса, сохраняются в файле конфигурации, расположенном обычно по адресу `~/config/htop/htoprc`. Если описать тем или иным образом настройки, затем перепределить их в ходе сеанса работы с утилитой, новые значения перекроют все описанные ранее в файле `htoprc`.

Работаем с Bash и ZSH

Все начинается с настройки под свои нужды. Как Bash, так и ZSH обычно включают файл (его называют иначе с точки зрения содержания), содержащий настройки, но по умолчанию не отображаемый при выводе содержимого каталога. Располагается он в домашнем каталоге данного пользователя. У Bash этот файл называется `.bashrc`, у ZSH — `.zshrc`. Обе эти командные оболочки поддерживают несколько уровней местоположений, откуда настройки загружаются в заданном порядке, последний из которых — файл строк конкретного пользователя.

При установке ZSH файл `.zshrc` обычно сразу не создается. Минимальная его версия в дистрибутиве CentOS выглядит следующим образом (все комментарии убраны для краткости):

```
$ cat /etc/skel/.zshrc
autoload -U compinit
compinit

setopt COMPLETE_IN_WORD
```

Соответствующий файл Bash содержит еще пару дополнительных элементов, но ничего неожиданного. Вне всякого сомнения, рано или поздно вы столкнетесь с чрезвычайно раздражающим поведением или элементом какого-либо сервера, который вам нужно дублировать. Мы, например, просто не можем без многоцветного текста в терминале, так как какой бы командной оболочкой ни пользовались, обязательно включаем там выделение различными цветами. Не успеете оглянуться, как окочените по уши в различных настройках и захотите с ними добиться нескольких удобных псевдонимов и функций.

Далее возникает необходимость не строить текстовый редактор, и очень неудобно, когда различных машин они различаются либо при добавлении новых машин, либо они не настроены все эти удобные псевдонимы, и просто *невыносимо*, что нигде не включен поддержка выделения текста различными цветами. Все мы решили эту проблему совершенно разными, импровизированными, переносимыми способами: Альфредо в какой-то момент применял *Makefile*, его соратники либо действовали сценариями Bash, либо вообще ничего не использовали. Упорядочить файлы и строк позволяет новый проект Dotdrop (<https://deadcode.re/dotdrop>), включающий массу возможностей, в частности копирования, создания символических ссылок и поддержки отдельных *профилей* для различных машин — весьма удобно при переходе от одной машины к другой.

Dotdrop можно использовать для проектов на языке Python, и, хотя он установлен с помощью обычных `virtualenv` и `pip`, лучше включить его в качестве подмодуля в свой репозиторий файлов и строк. Если вы еще так не делаете, рекомендуем хранить все свои файлы и строк в системе контроля версий, чтобы отслеживать изменения. Альфредо открыл свободный доступ к своим файлам и строк (<https://oreil.ly/LV1AH>) и старается по возможности поддерживать их в актуальном состоянии.

Независимо от того, что вы используете, имеет смысл отслеживать изменения с помощью системы контроля версий и поддерживать все в актуальном состоянии.

Настройка командной оболочки Python под свои нужды

Вы можете не строить командную оболочку Python под свои нужды с помощью пакет `helpers` и импортировать полезные модули в файл Python, затем экспортировать его в виде переменной среды. Я храню свои файлы конфигурации в репозитории `dotfiles`, так что в файле конфигурации командной оболочки (в моем случае `$HOME/.zshrc`) я написал следующий экспорт:

```
export PYTHONSTARTUP=$HOME/dotfiles/pythonstartup.py
```

Чтобы поэкспериментировать с этим, создайте новый файл Python с названием `pythonstartup.py` (хотя название может быть любым) со следующим содержанием:

```
import types
import uuid

helpers = types.ModuleType('helpers')
helpers.uuid4 = uuid.uuid4()
```

Теперь откройте новую командную оболочку Python, указав только что созданный файл `pythonstartup.py`:

```
$ PYTHONSTARTUP=pythonstartup.py python
Python 3.7.3 (default, Apr 3 2019, 06:39:12)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> helpers
<module 'helpers'>
>>> helpers.uuid4()
UUID('966d7dbe-7835-4ac7-bbbf-06bf33db5302')
```

Сразу же становится доступен объект `helpers`. А к добавленному свойству `uuid4` можно обратиться к `helpers.uuid4()`. Конечно, верное, дождись, все импорты и определения становятся доступны в командной оболочке Python. Такой способ расширения поведения очень удобен при использовании командной оболочки по умолчанию.

Рекурсивные подстановки

Рекурсивная подстановка включена в ZSH по умолчанию, но Bash (версии 4 и выше) требует использования встроенной команды `shopt` для ее включения. Рекурсивная подстановка — удобная возможность, позволяющая обходить пути с помощью следующего синтаксиса:

```
$ ls **/*.py
```

Этот фрагмент код проходит по всем файлам и каталогам, выводя все найденные файлы, названия которых оканчиваются на .py. Вот как включить эту возможность в Bash 4:

```
$ shopt -s globstar
```

Поиск и замена с запросами подтверждения

В механизме поиска /з меню редактор Vim есть очень интересная возможность — запрос подтверждения замены (выполнять замену или пропустить это вхождение). Особенно удобно это, когда не удается подобрать регулярное выражение, точно соответствующее в шим требованиям, и некоторые близкие совпадения необходимо игнорировать. Мы неплохо разбираемся в регулярных выражениях, но старались не слишком углубляться в их изучение, чтобы избежать соблазна применять их повсюду. Чуть всего в мдост точно будет простого поиска и не захочется ломать голову над идеальным регулярным выражением.

Чтобы включить запросы подтверждения в Vim, необходимо добавить в конфигурацию:

```
:%s/исходный_терм/заменитель/gc
```

Этот комманд обозначает: искать исходный_терм во всем файле и заменять его на заменитель, во всех случаях запрашивая у пользователя, заменять его или пропустить. При обн ружении вхождения Vim выведет сообщение на подобие следующего:

```
Заменить на заменитель (y/n/a/q/l/^E/^Y)?
```

Процесс подтверждения в целом может показаться глупым, но позволяет предъявлять к регулярному выражению менее жесткие требования или вообще не принимать регулярное выражение в простых случаях поиска /з замены. Небольшой пример этого — недавние изменения API в утилите, когда трибут объект был заменен вызываемую функцию. Код возвращал True или False в зависимости от того, требовались ли полномочия суперпользователя. Сменив файл выполнялсь примерно следующим образом:

```
:%s/needs_root/needs_root()/gc
```

Дополнительная сложность заключалась в том, что строка needs_root часто встречалась в комментариях и docstring, подобрать такое регулярное выражение, чтобы не делить замены внутри блоков комментариев или docstring, было очень непросто. При использовании же флагов можно просто нажать

клаватуры Y или N и перейти к следующему вхождению. Никого регулярного выражения вообще не нужно!

При включенных рекурсивных подстановках (`shopt -s globstar` в Bash 4) следующий следующий однострочный сценарий проходит по всем подходящим файлам, выполняет поиск и в случае обнаружения в файлах указанного шлона заменяет элемент в соответствии с реакцией пользователя на запросы подтверждения:

```
vim -c "bufdo! set eventignore-=Syntax | %s/needs_root/needs_root()/gce" **/*.py
```

Здесь не помещаются развернутые пояснения. Приведенный пример обходит файловую систему рекурсивно, нходя все файлы, нзвние которых окнчивается на `.py`, згружает их в Vim и выполняет внутри них поиск и, обнаружив соответствия шлону, зменяет с подтверждением. Если в файле вхождений не найдено, он пропускается. Команда `set eventignore-=Syntax` необходим, поскольку иначе Vim не згружает файлы синтаксиса, нмнравится выделение синтаксических элементов и мы хотели бы видеть его при подобной змене. Следующая часть сценария, после символ `|`, представляет собой змену с флагом подтверждения и флагом для игнорирования ошибок, которые могли бы помешать бесперебойной работе.



Существует множество прочих флагов и вариантов усовершенствования команд змены. Узнать больше о специальных флагах при поиске/змене в Vim вы можете с помощью команды `:help substitute`, особенно в разделе `s_flags`.

Упростить зпоминание приведенного ранее однострочного сценария можно с помощью функции, принимающей два параметра (искомое и змененный путь):

```
vsed() {
  search=$1
  replace=$2
  shift
  shift
  vim -c "bufdo! set eventignore-=Syntax| %s/$search/$replace/gce" $*
}
```

Назовем ее `vsed`, как смесь Vim и утилиты `sed`, чтобы проще было зпомнить. В термине команда выглядит очень просто и дает возможность легко и уверенно вносить изменения во много файлов сразу, поскольку позволяет принимать или отклонять каждую змену:

```
$ vsed needs_root needs_root() **/*.py
```

Удаление временных файлов Python

Файлы рус Python, с недавних пор и его к т логи `pycache` порой «путаются под ногами» у р зр ботчик . Следующий простой однострочный сценарий с псевдонимом `pyclean` снимает файлы рус с помощью команды `find`, а затем ходит к т логи `pycache` и рекурсивно удаляет их с помощью встроенного флага `-delete` этой утилиты:

```
alias pyclean='find . \
  \( -type f -name "*.py[co]" -o -type d -name "__pycache__" \) -delete &&
  echo "Removed pycs and __pycache__"'
```

Вывод списка процессов и его фильтрация

Как минимум несколько раз в день вам придется просмотреть список запущенных на машине процессов и фильтровать его в поисках конкретного приложения. Совсем не удивительно, что у всех р зр ботчиков есть привычные боры или порядок флагов утилиты `ps` (мы, например, обычно используем `aux`). Вы будете выполнять эту процедуру ежедневно столько раз, что флаги и их порядок немертво отпечатываются в вашем мозгу и сделать к-то иначе будет сложно.

Для начала попробуйте следующий вариант вывода списка процессов и некоторой дополнительной информации, например их идентификаторов:

```
$ ps auxw
```

Эта команда выводит все процессы с помощью флагов *в стиле BSD* (флагов без префикса `-`) вне зависимости от того, действуют ли они термिनале (`tty`) или нет, включая информацию о пользователе — владельце процесса. Наконец, она расширяет место, выделяемое для выводимой информации (флаги `w`).

Чтобы всего вы будете фильтровать этот вывод с помощью `grep` в поисках информации о конкретном процессе. Например, чтобы проверить, запущен ли Nginx, выводимая информация передается посредством конвейера в `grep` с `nginx` в качестве аргумента:

```
$ ps auxw | grep nginx
root      29640  1536 ?        Ss   10:11   0:00 nginx: master process
www-data  29648  5440 ?        S    10:11   0:00 nginx: worker process
alfredo   30024   924 pts/14  S+   10:12   0:00 grep nginx
```

Все замечательно, но каждый раз вызывать команду `grep` довольно утомительно. Особенно это заметно, когда нужны результаты, кроме своего `grep`, не пишется:

```
$ ps auxw | grep apache
alfredo   31351  0.0  0.0  8856  912 pts/13  S+   10:15   0:00 grep apache
```

Процесс `apache` не найден, но информация отображается в том виде, что можно ошибочно подумать, что найден, каждый раз перепроверять, действительно ли там присутствует только `grep`, довольно утомительно. Один из способов решения этой проблемы — добавление к команде `grep` еще одного конвейера, чтобы отфильтровать ее с помощью результатов:

```
$ ps auxw | grep apache | grep -v grep
```

Но необходимо всегда помнить, что нужно добавит дополнительную `grep`, результат не меньше. Непомощь приходит псевдоним:

```
alias pg='ps aux | grep -v grep | grep $1'
```

Новый псевдоним фильтрует первую строку `grep`, оставляя только интересные результаты (при их наличии):

```
$ pg vim
alfredo 31585 77836 20624 pts/3 S+ 18:39 0:00 vim /home/alfredo/.zshrc
```

Метка даты/времени Unix

Получить метку даты/времени Unix в Python очень просто:

```
In [1]: import time
In [2]: int(time.time())
```

```
Out[2]: 1566168361
```

Но в командной оболочке сделать это несколько сложнее. Следующий псевдоним работает в OS X, в которую включен утилит `date` в стиле BSD:

```
alias timestamp='date -j -f "%a %b %d %T %Z %Y" "`date`" "+%s"'
```

Утилиты OS X порой довольно неуклюжи, и результатчик часто сбивается с толку, что конкретная утилит (в данном случае `date`) ведет себя совершенно по-разному. В Linux-версии утилиты `date` возможен и много более простой подход:

```
alias timestamp='date "+%s"'
```

Комбинирование Python с Bash и ZSH

Никогда не приходило в голову комбинировать Python с командной оболочкой, например, Bash или ZSH. Кажется, что это противоречит здравому смыслу, но существует несколько встречающихся буквально каждый день случаев для

этого. В общем случае мы следуем тому же эмпирическому правилу: максимум десять строк для сценария командной оболочки, все, что больше, — поле для ошибок, на которые вы впустую будете тратить время, поскольку никаких сообщений об ошибках здесь не выводится.

Генератор случайных чисел

Количество необходимых еженедельно учетных записей и паролей продолжит расти, в том числе и одноразовых учетных записей, для которых можно сгенерировать и дежурные пароли с помощью Python. Создадим удобный генератор случайных паролей, отправляющий результаты в буфер обмена, откуда их можно сразу вставить куда-нибудь:

```
In [1]: import os
```

```
In [2]: import base64
```

```
In [3]: print(base64.b64encode(os.urandom(64)).decode('utf-8'))
gNH1GXnqnbsALbAZrGaw+LmvipTeFi3tA/9uB1tNf9g2S9qTQ8hTpBYrXStp+i/o5TseeVo6wcX2A==
```

Преобразуем его в функцию командной оболочки, которая может принимать в качестве аргумента длину пароля (удобно в случаях, когда сбит ограничитель возможной длины пароля):

```
mpass() {
    if [ $1 ]; then
        length=$1
    else
        length=12
    fi
    _hash=`python3 -c "
import os,base64
exec('print(base64.b64encode(os.urandom(64))[:${length}].decode(\'utf-8\'))')
"`
    echo $_hash | xclip -selection clipboard
    echo "new password copied to the system clipboard"
}
```

По умолчанию функция `mpass` генерирует пароли длиной 12 символов путем усечения результатов и отправляет содержимое сгенерированной строки утилите `xclip` для копирования в буфер обмена.



Во многих дистрибутивах `xclip` не установлен по умолчанию, так что ее необходимо установить, чтобы эта функция работала properly. Если доступ к `xclip` нет, можно воспользоваться любой другой логичной утилитой для управления системным буфером обмена.

Существует ли нужный мне модуль?

Следующий скрипт выясняет, существует ли заданный модуль, и если это так, получает путь к нему. Его удобно повторно применять для функций, использующих эту информацию для дальнейшей обработки:

```
try() {
    python -c "
exec('''
try:
    import ${1} as _
    print(__file__)
except Exception as e:
    print(e)
''')"
```

Переходим из текущего каталога по пути к модулю

При отладке библиотек и зависимостей или просто при изучении исходного кода модулей часто возникает вопрос «Где расположен этот модуль?». Способ угадывания и просмотра модулей в Python далеко не очевиден, и в различных дистрибутивах Linux пути совершенно разные, причем применяются различные соглашения. Путь к модулю можно узнать, если импортировать его и затем воспользоваться `print`:

```
In [1]: import os
```

```
In [2]: print(os)
<module 'os' from '.virtualenvs/python-devops/lib/python3.6/os.py'>
```

Не слишком удобно, если вам требуется только путь, чтобы перейти по нему и посмотреть на модуль. Следующая функция пытается импортировать модуль, вывести его (не помним, что это командная оболочка, так что `return` ничего не делает), а затем перейти в соответствующий каталог:

```
cdp() {
    MODULE_DIRECTORY=`python -c "
exec('''
try:
    import os.path as _, ${module}
    print(_.dirname(_.realpath(${module}.__file__)))
except Exception as e:
    print(e)
''')`
    if [[ -d $MODULE_DIRECTORY ]]; then
        cd $MODULE_DIRECTORY
```

```

else
    echo "Module ${1} not found or is not importable: $MODULE_DIRECTORY"
fi
}

```

Немного повысим ее устойчивость к ошибкам и случаям, если в названии пакета присутствует тире, в модуле — подчеркивание, добавив:

```
module=$(sed 's/-/_/g' <<< $1)
```

Если во введенном названии есть тире, наш маленький скрипт решит проблему и перенесет его в нужный каталог:

```

$ cdp pkg-resources
$ pwd
/usr/lib/python2.7/dist-packages/pkg_resources

```

Преобразование CSV-файла в JSON

В Python есть несколько встроенных утилит, которые могут вас приятно удивить, если вы раньше с ними не сталкивались, в частности, для нативной обработки JSON и CSV-файлов. Для загрузки CSV-файла и дальнейшего сброса его содержимого в JSON достаточно всего нескольких строк кода. Возьмем следующий CSV-файл (`addresses.csv`) и посмотрим на содержимое после сброса в JSON в командную оболочку Python:

```

John,Doe,120 Main St.,Riverside, NJ, 08075
Jack,Jhonson,220 St. Vernardeen Av.,Phila, PA,09119
John,Howards,120 Monroe St.,Riverside, NJ,08075
Alfred, Reynolds, 271 Terrell Trace Dr., Marietta, GA, 30068
Jim, Harrison, 100 Sandy Plains Plc., Houston, TX, 77005

```

```

>>> import csv
>>> import json
>>> contents = open("addresses.csv").readlines()
>>> json.dumps(list(csv.reader(contents)))
'[["John", "Doe", "120 Main St.", "Riverside", "NJ", "08075"],
["Jack", "Jhonson", "220 St. Vernardeen Av.", "Phila", "PA", "09119"],
["John", "Howards", "120 Monroe St.", "Riverside", "NJ", "08075"],
["Alfred", "Reynolds", "271 Terrell Trace Dr.", "Marietta", "GA", "30068"],
["Jim", "Harrison", "100 Sandy Plains Plc.", "Houston", "TX", "77005"]]'

```

Преобразуем наш интерактивный сеанс в функцию, которую можно вызвать из командной строки:

```

csv2json () {
    python3 -c "
exec('
import csv,json

```

```
print(json.dumps(list(csv.reader(open(\'${1}\')))))
'''
"
```

Использовать ее в командной строке намного проще, чем запоминать все вызовы и модули:

```
$ csv2json addresses.csv
[["John", "Doe", "120 Main St.", "Riverside", " NJ", " 08075"],
["Jack", "Jhonson", "220 St. Vernardeen Av.", "Phila", " PA", "09119"],
["John", "Howards", "120 Monroe St.", "Riverside", " NJ", "08075"],
["Alfred", " Reynolds", " 271 Terrell Trace Dr.", " Marietta", " GA", " 30068"],
["Jim", " Harrison", " 100 Sandy Plains Plc.", " Houston", " TX", " 77005"]]
```

Однострочные сценарии Python

В общем случае писать длинные однострочные сценарии Python не рекомендуется. Руководство PEP 8 (https://oreil.ly/3P_qQ) неодобрительно отзывается даже о составных операторх, включающих точку с запятой (в Python можно использовать точку с запятой!). Но краткие отладочные операторы и вызовы отладчика вполне допустимы. В конце концов, они носят временный характер.

Отладчики

Некоторые программисты упорно считают оператор `print()` лучшей стратегией отладки исполняемого кода. В некоторых случаях можно использовать и его, но чаще всего мы применяем отладчик Python (с помощью модуля `pdb`) или `ipdb`, использующий IPython в качестве прикладной чистки. Благодаря созданию точек останова можно проследить значения переменных и двигаться вверх/вниз по стеку. Следующие однострочные операторы достаточножны для того, чтобы их запомнить.

Создание точек останова и переход к отладчику Python (`pdb`):

```
import pdb;pdb.set_trace()
```

Создание точек останова и переход к отладчику Python на основе IPython (`ipdb`):

```
import ipdb;ipdb.set_trace()
```

Хотя формально следующий однострочный сценарий и не является отладчиком (не позволяет двигаться вверх/вниз по стеку), но он дает возможность запустить `sys` Python, когда выполнение доходит до него:

```
import IPython; IPython.embed()
```



У всех есть свои любимые утилиты отладки. `pdb` предстает в не слишком удобным (нет автодополнения и подсветки синтаксиса), `ipdb` не вписывается больше. Не удивляйтесь, если кто-то предпочтет с другим отладчиком! В конце концов, полезно знать, как работает `pdb`, он представляет собой фундамент, который следует освоить в совершенстве вне зависимости от используемого в отладчик. В системах, которые вы не можете контролировать, вам придется применять непосредственно `pdb`, поскольку при внесении нововведений у вас нет, и хотя в этом вряд ли повторится, но позволит обойти ограничения.

Быстро ли работает конкретный фрагмент кода?

В Python есть модуль, позволяющий выполнить какой-либо фрагмент кода несколько раз подряд и вычислить для него показатели производительности. Многие пользователи любят спрашивать, как эффективнее написать цикл или обновить ассоциативный массив, так что специалисты обожают модуль `timeit`, с помощью которого можно проверить быстродействие.

Как вы уже, наверное, заметили, мы большие поклонники IPython (<https://ipython.org>), его интерактивная командная оболочка включает специальную «математическую» функцию для модуля `timeit`. «Математические» функции предназначены для выполнения какой-либо конкретной операции в командной оболочке, перед их названием указывается символ `%`. Во все времена излюбленный вопрос, связанный с быстродействием, звучит так: что выполняется быстрее — списковое включение или добавление элементов в конец списка? В приведенных далее двух примерах мы ответим на него с помощью модуля `timeit`:

```
In [1]: def f(x):
...:     return x*x
...:
In [2]: %timeit for x in range(100): f(x)
100000 loops, best of 3: 20.3 us per loop
```

В стандартной командной оболочке (или интерпретаторе) Python мы импортируем модуль `timeit` и обращаемся к нему напрямую. В этом случае вызов выглядит немного иначе:

```
>>> array = []
>>> def appending():
...     for i in range(100):
...         array.append(i)
...
>>> timeit.repeat("appending()", "from __main__ import appending")
[5.298534262983594, 5.32031941099558, 5.359099322988186]
>>> timeit.repeat("[i for i in range(100)]")
[2.2052824340062216, 2.1648171059787273, 2.1733458579983562]
```

Результаты выглядят немного странно, но дело в том, что они предназначены для обработки другим модулем или библиотекой, а не для чтения человеком. Средние показатели говорят в пользу спискового включения. Вот как выглядит соответствующий код на языке Python:

```
In [1]: def appending():
...:     array = []
...:     for i in range(100):
...:         array.append(i)
...:

In [2]: %timeit appending()
5.39 µs ± 95.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [3]: %timeit [i for i in range(100)]
2.1 µs ± 15.2 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Благодаря тому что в IPython для `timeit` есть специальный команд (что видно из префикса `%` перед ней), ее вывод удобнее для чтения людьми и понятнее им, так как не требует хитрой операции импорта, как в стандартной командной оболочке Python.

strace

В случаях, когда приложения не несут интересную и информативную информацию в журналы или вообще не выполняют журнальных действий, чрезвычайно важно знать, как программно взаимодействует с операционной системой. Результаты `strace` не слишком удобны для чтения, но при определенном понимании основ ее работы помогут понять, что происходит с проблемным приложением. Однажды Альфредо пытался выяснить причину отказа в доступе к файлу. Файл находился внутри символической ссылки, вроде бы, со всеми нужными правами доступа. В чем было дело? По одним журналам было сложно понять это, ведь в них не отображались права доступа при обращении к файлу.

В выводе `strace` можно найти следующие две строки:

```
stat("/var/lib/ceph/osd/block.db", 0x7fd) = -1 EACCES (Permission denied)
lstat("/var/lib/ceph/osd/block.db", {st_mode=S_IFLNK|0777, st_size=22}) = 0
```

Программа установила права доступа для родительского каталога — того, который соответствовал ссылке, и `block.db` — в данном случае к той же ссылке на устройство блочного ввода/вывода. У самого устройства блочного ввода/вывода были нужные права, так в чем же заключалась проблема? Оказалось,

что у ссылки не установлен *sticky bit*, предотвращающий изменение пути другими ссылками, включая устройство блочного ввода/вывода. Утилиты `chown` есть специальный флаг (`-h` или `--no-dereference`), указывающий, что смена владельца должна влиять и на ссылки.

Выполнять подобную отладку без чего-то наподобие утилиты `strace` очень сложно, а то и вовсе невозможно. Попробуйте ее с нами, для чего создайте файл `follow.py` со следующим содержанием:

```
import subprocess

subprocess.call(['ls', '-alh'])
```

Он импортирует модуль `subprocess` для выполнения системных вызовов, после чего выводит результаты системного вызова `ls`. Вместо вызова напрямую с помощью Python можете передкомандой `strace` и посмотрите, что произойдет:

```
$ strace python follow.py
```

Терминальный экран заполнится огромным количеством информации, большая часть которой, вероятно, потеряется в море информации. Стоит себя просмотреть каждую строку независимо от того, понятно ли ее содержание. Некоторые строки легче отделить от прочих. Например, экран заметно множество вызовов `read` и `fstat`, вы увидите непосредственно системные вызовы, выполняемые процессом на вашем компьютере. Из некоторыми файлами также производятся операции `open` и `close`, кроме того, в отдельном разделе можно видеть несколько вызовов `stat`:

```
stat("/home/alfredo/go/bin/python", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/local/go/bin/python", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/local/bin/python", 0x7ff) = -1 ENOENT (No such file)
stat("/home/alfredo/bin/python", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/local/sbin/python", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/local/bin/python", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/sbin/python", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/bin/python", {st_mode=S_IFREG|0755, st_size=3691008, ...}) = 0
readlink("/usr/bin/python", "python2", 4096) = 7
readlink("/usr/bin/python2", "python2.7", 4096) = 9
readlink("/usr/bin/python2.7", 0x7ff, 4096) = -1 EINVAL (Invalid argument)
stat("/usr/bin/Modules/Setup", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/bin/lib/python2.7/os.py", 0x7ffd) = -1 ENOENT (No such file)
stat("/usr/bin/lib/python2.7/os.pyc", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/lib/python2.7/os.py", {st_mode=S_IFREG|0644, ...}) = 0
stat("/usr/bin/pybuilddir.txt", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/bin/lib/python2.7/lib-dynload", 0x7ff) = -1 ENOENT (No such file)
stat("/usr/lib/python2.7/lib-dynload", {st_mode=S_IFDIR|0755, ...}) = 0
```

Моя система довольно старая, под `python` в этих результатах подразумевается `python2.7`, так что он просматривает файловую систему в поисках нужного исполняемого файла и проходит несколько каталогов, пока не доберется до `/usr/bin/python`, представляющего собой ссылку на `/usr/bin/python2`, который, в свою очередь, представляет собой еще одну ссылку, отправляющую процесс в каталог `/usr/bin/python2.7`. Далее он вызывает `stat` для каталога `/usr/bin/Modules/Setup`, о котором мы как раз ботчики Python никогда не слышали, и просто переходит далее к модулю `os`.

Затем Python переходит к `pybuilddir.txt` и `lib-dynload`. Ничего себе путешествие! Без `strace`, вероятно, мы читали бы релизующий эти действия код, пытаюсь понять последовательность переходов. Утилита `strace` несколько облегчает эту задачу, отображая все промежуточные шаги и сопровождая их полезной информацией о каждом вызове.

У этой утилиты есть множество интересных флагов, например, она умеет *подсоединяться к процессу с конкретным PID*. Если вы знаете идентификатор процесса, можете запросить у `strace` отчет обо всем происходящем с ним.

В число полезных флагов входит `-f`, при использовании которого утилита `strace` отслеживает созданные исходной программой дочерние процессы. В примере файл Python производится вызов модуля `subprocess`, который, в свою очередь, вызывает команду `ls`. Если модифицировать вызов `strace`, добавив флаг `-f`, будет выведен информация со всеми подробностями этого вызова.

При выполнении `follow.py` в домашнем каталоге с флагом `-f` можно заметить немало отличий, в частности вызовы `lstat` и `readlink` для файловых строк (некоторые из них представляют собой символические ссылки):

```
[pid 30127] lstat(".vimrc", {st_mode=S_IFLNK|0777, st_size=29, ...}) = 0
[pid 30127] lgetxattr(".vimrc", "security.selinux", 0x55c5a36f4720, 255)
[pid 30127] readlink(".vimrc", "/home/alfredo/.vimrc", 30) = 29
[pid 30127] lstat(".config", {st_mode=S_IFDIR|0700, st_size=4096, ...}) = 0
```

Здесь не только отображаются обращения к этим файлам, но и выводятся идентификаторы процессов в начале строк, благодаря чему становится проще выяснять, какой дочерний процесс что делает. Вызов `strace` без флагов `-f`, в частности, не отображает идентификаторы процессов.

Наконец, для детального анализа выведенной информации удобно было бы сохранить ее в файл. Для этого служит флаг `-o`:

```
$ strace -o output.txt python follow.py
```


Вопросы и упражнения

- Дайте определение IOPS.
- Объясните, в чем различие между пропускной способностью и IOPS.
- Назовите какое-нибудь ограничение `fdisk` в смысле создания разделов диска, отсутствующее у `parted`.
- Назовите три утилиты, которые могут предоставить информацию о диске.
- Назовите, что способны туннели SSH и когда они могут оказаться полезными?

Задача на ситуационный анализ

Создайте нагрузочный тест с помощью утилиты `molotov` для тестирования ответа сервера в формате JSON с кодом 200 состояния HTTP.

ГЛАВА 5

Управление пакетами

Зачастую небольшие сценарии оказываются не столько полезными и важными, что возникает потребность в совместном использовании и распространении их содержимого. Библиотеки Python, как и другие программные проекты, требуют пакетной организации. Без этого распространение кода сильно затрудняется и неэффективность его падает.

После прохождения проектом этап пробной версии имеет смысл отслеживать изменения, извещая пользователей о типе изменения (например, при выпуске обновления с обратной совместимостью) и предоставляя им возможность основываясь на своих программах конкретной версии. Даже в самых простых сценариях применения желательны следы нескольких рекомендаций по созданию пакетов, включающим как минимум ведение журналов изменений и контроль версий.

Существует несколько стратегий управления пакетами, знание и более распространённых из них позволит выбрать наилучший способ решения поставленной задачи. Например, библиотеки Python удобнее распространять через каталог пакетов Python (Python Package Index, PyPI), вместо того чтобы делать из них системные пакеты типа Debian и RPM. Если же сценарий Python необходимо запускать в определённые моменты или он должен выполняться в течение длительного времени, возможно, стоит создать из них системный пакет для работы с `systemd`.

И хотя утилита `systemd` не предназначена для работы с пакетами, она отлично подходит для управления процессами и последовательностью загрузки сервера. Вы сможете существенно расширить возможности своего проекта Python, если научитесь управлять процессами с помощью нескольких инструментов `systemd` и компоновки пакетов.

Универсальных утилит создания пакетов Python есть общедоступный сервис публикации (PyPI). Однако создание локального репозитория для пакетов Debian и RPM требует некоторых усилий. В этой главе мы рассмотрим несколько ути-

лит, упрощающих создание и сопровождение локального репозитория, включая локальную альтернативу PyPI.

Для стабильного и согласованного простроения программного обеспечения жизненно важны хорошее понимание различных стратегий создания пакетов и такие разумные практики, как надежный контроль версий и ведение журналов изменений.

Почему пакетная организация программ так важна

Несколько факторов делают пакетную организацию программного обеспечения неотъемлемой составляющей любого проекта (независимо от размера!). Отслеживание версий и изменений (с помощью журналов изменений) — прекрасный способ рассмотреть новые возможности и исправленные ошибки. Благодаря контролю версий пользователи лучше понимают, какие возможности предоставляет проект.

Журналы изменений с точным описанием изменений — неоценимое средство выявления возможных причин сбоев системы при поиске проблем и программных ошибок.

Контроль версий проекта, описание изменений в журнале изменений и предоставление пользователям различных способов установки и применения проекта требуют дисциплины и упорной работы. Однако польза от этого при простроении, отладке, обновлении и дальнейшем удлинении программного обеспечения весьма существенна.

Случаи, когда пакетная организация программ не нужна

Иногда вообще не нужно простроить проект по другим системам. «Сборники сценариев» Ansible обычно запускаются с одного сервера для управления прочими системами в сети. В таких случаях, как и с Ansible, достаточно контроля версий и журналов изменений.

Системы контроля версий, такие как Git, сильно упрощают решение этой задачи с помощью тегов. Тегирование в Git полезно, даже если из проекта нужно сделать пакет, поскольку большинство утилит может читать теги (если тег относится к версии проекта, конечно) и генерировать на их основе пакеты.

Недавно мы потратили немало времени на отладку, пытаясь выяснить, почему перестал работать программная установка для большого программного проекта. Внезапно все функциональные тесты мейнкой Python-утилиты, для которой было важно, чтобы программная установка завершила загрузку, стали завершаться неудачей. И хотя у программы установки были различные версии,

синхронизируемые с системой контроля версий, но не было никого, кто бы сообщил об изменениях, в котором бы упоминалось, что очередные изменения нарушают работу существующего API. Чтобы найти возможную причину проблемы, нам пришлось просмотреть все недавние коммиты.

Просмотреть несколько коммитов несложно, но попробуйте сделать это в проекте, где их более 4000! После обнаружения причин проблемы мы открыли две заявки на исправление: одну относительно недавней ошибки, вторая — с запросом на создание новых изменений.

Рекомендации по пакетной организации программ

Прежде чем приступить к созданию пакетов, стоит обдумать несколько нюансов для максимальнй бесперебойности процесса. И даже если вы не собираетесь делать из программного продукта пакеты, эти методические рекомендации помогут улучшить проект в целом.



Проекты, включенные в систему контроля версий, всегда готовы к преобразованию в пакеты.

Информативный контроль версий

Существует множество способов версионной организации программного обеспечения, но лучше всего придерживаться испытанной схемы. В руководстве разработчик Python четко определены (<https://oreil.ly/СЗУКО>) приемлемые формы контроля версий.

Схема контроля версий должна отличаться исключительной гибкостью, но с учетом согласованности, чтобы программист мог соответствующим образом расставить приоритеты (например, выбрать стабильную или бета-версию). В чистейшей форме, чаще всего встречающейся в пакетах Python, используются следующие два варианта указания версий: старшая.младшая и старшая.младшая.микро.

При этом допустимыми будут версии следующего вида :

- 0.0.1;
- 1.0;
- 2.1.1.



Хотя в з меч тельном руководстве р зр ботчик Python опис но множе-ство в ри нтов, рекомендуем применять простейшие формы, перечислен-ные р нее. Они прекр сно подходят для созд ния п кетов и соответствуют большинству рекомен дций относительно к к системных, т к и н тивных п кетов Python.

Н иболее р спростр ненный форм т ук з ния выпуск емых версий — стар-шая.младшая.микро (он используется т кже в схеме сем нтического версионни-ров ния (<https://semver.org/lang/ru/>)):

- старшая версия — для обр тно несовместимых изменений;
- младшая версия включ ет в ПО обр тно совместимую функцион льность;
- микроверсия включ ет в ПО обр тно совместимые испр вления ошибок.

Исходя из т кого опис ния версий, можно сдел ть вывод, что з висимость приложения версии **1.0.0** может перест ть р бот ть при использо вании вер-сии **2.0.0**.

После принятия решения о выпуске новой версии определить ее номер не-сложно. Если текущ я версия р зр б тыв емого проект — **1.0.0**, возможны следующие в ри нты.

- Если выпуск ем я версия включ ет обр тно несовместимые изменения, ее номер **2.0.0**.
- Если выпуск ем я версия доб вляет в ПО возможности, не н руш ющие обр тной совместимости, ее номер **1.1.0**.
- Если выпуск ем я версия испр вляет прогр ммные ошибки, т кже не н -руш ющие обр тной совместимости, ее номер **1.0.1**.

Если следов ть этой схеме, процесс выпуск версий ср зу же ст новится вполне информ тивным. И хотя было бы хорошо, если бы все прогр ммное обеспече-ние соответствов ло одному и тому же ш блону, но у некоторых проектов своя, ни н что не похож я схем версий. Н пример, в проекте Ceph (<https://ceph.com>) применяется схем версий старшая.[0|1|2].младшая:

- старшая ук зыв ет н основную версию, хотя и не обя з тельно н руш ющую обр тную совместимость;
- 0, 1 или 2 озн ч ют опытную версию, предв рительную версию и ст бильную версию соответственно;
- младшая версия используется только для испр вления ошибок, не введения новых возможностей.

При той же схеме 14.0.0 означает опытную версию, 14.2.1 — выпуск для исправления ошибок в стабильной основной версии (в данном случае 14).

Журнал изменений

Как мы уже упоминали, важно следить за выпусками версиями и включать в них изменениями в зависимости от номера версии. Вести журнал изменений не так уж сложно, если выбран конкретный способ контроля версий. Журнал изменений может храниться и в одном файле, хотя в крупных проектах он обычно разбивается на несколько небольших файлов, находящихся в одном каталоге. Рекомендуется при этом использовать простой информативный формат, удобный для сопровождения.

Следующий пример взят из настоящего журнала изменений репозитория Python:

```
1.1.3
^^^^
22-Mar-2019
* Никаких изменений кода, только добавлены файлы для создания пакета под Debian
1.1.2
^^^^
13-Mar-2019
* Перебирает несколько различных исполняемых файлов (не только ``python``)
в поисках работающего, в порядке предпочтения, начиная с ``python3``, и доходя
в конце концов до интерпретатора соединений
```

В этом журнале изменений присутствует четыре важных элемента информации.

1. Номер последней выпущенной версии.
2. Совместим ли обратно последняя выпущенная версия.
3. Датировка последней версии.
4. Включенные в этот выпуск изменения.

Этот файл не обязательно должен записываться в каком-то особом формате, главное — единообразие и информативность. Из хорошего журнала изменений сразу ясны несколько важных элементов информации. Может показаться заманчивым полностью автоматизировать процесс записи журнала изменений при каждом выпуске версии, но мы не рекомендуем так поступать: нет ничего лучше хорошо написанного и продуманного описания исправленной ошибки или добавленной возможности.

Плохо автоматизированный журнал изменений просто включает все коммиты из системы контроля версий, попавшие в данную версию. Это не лучший вариант, поскольку ту же самую информацию можно получить просто из списка коммитов.

Выбор стратегии

Для выбора типа создаваемых пакетов очень полезно понимать, к какому типу дистрибутивов требуется и какие сервисы инфраструктуры доступны. Библиотеки на чистом языке Python, расширяющие функциональность других проектов на Python, имеет смысл распространять в виде нативных пакетов Python и размещать в каталоге пакетов Python (PyPI) или локальном каталоге.

Автономные сценарии и долгоживущие процессы — хорошие кандидаты на роль системных пакетов наподобие RPM или Debian в зависимости от типов доступных систем, а также возможности в принципе разместить где-то репозиторий (и поддерживая его роботом). В случае долгоживущих процессов системная пакетка может включать привилегированную настройку юнита `systemd`, которые превращают его в доступный пользователям управляемый процесс. С помощью `systemd` можно мягко запустить, остановить или перезапустить операции, что невозможно для нативных пакетов Python.

В целом чем теснее сценарий или процесс должен взаимодействовать с системой, тем лучше он подходит в качестве кандидата на роль системного пакета или контейнера. Для сценариев на чистом Python лучше подойдут обычные пакеты Python.



Не существует жестких требований к выбору стратегии. Все зависит от обстоятельств. Выберите лучшую из доступных сред дистрибутивов (например, RPM, если серверы работают под управлением CentOS). Различные типы пакетов взаимно не исключают друг друга, один проект может существовать в виде пакетов нескольких форматов.

Решения для создания пакетов

В этом разделе мы подробно рассмотрим вопросы создания пакетов и их размещения.

Для упрощения примеров код рассмотрим маленький проект `hello-world` на языке Python со следующей структурой:

```
hello-world
├── hello_world
│   ├── __init__.py
│   └── main.py
```

1 directory, 2 files

Проект состоит из каталога верхнего уровня `hello-world` и подкаталога (`hello_world`), содержащего два файла. В зависимости от выбранного типа пакета для его создания необходимы различные файлы.

Нативные пакеты Python

С самым простым, с большим отрывом, решением будет использовать нативные утилиты создания пакетов Python и средств их размещения (PyPI). Как и при реализации прочих стратегий пакетной организации программ, в нашем проекте нужны дополнительные файлы, используемые **setuptools**.



Один из простейших способов получения виртуальной среды — псевдоним `bash` или `zsh` для перехода в нужный каталог и создания виртуальной среды с помощью команды `source`, вот так: `alias sugar="source ~/.sugar/bin/activate && cd ~/src/sugar"`.

Далее создадим новую виртуальную среду и активируем ее:

```
$ python3 -m venv /tmp/packaging
$ source /tmp/packaging/bin/activate
```



Для генерации нативного пакета Python необходим **setuptools**, представляющий собой набор утилит и вспомогательных функций, предназначенных для создания и упрощения пакетов Python.

В активной виртуальной среде существуют следующие зависимости:

- **setuptools** — набор утилит для создания пакетов;
- **twine** — утилита для регистрации пакетов и загрузки их на сервер.

Установите их с помощью команды:

```
$ pip install setuptools twine
```



Простейший способ выяснить, что установлено в системе, что — нет, — воспользоваться IPython и следующим фрагментом кода для вывода списка всех пакетов Python в виде структуры данных JSON:

```
In [1]: !pip list --format=json
[{"name": "appnope", "version": "0.1.0"},
{"name": "astroid", "version": "2.2.5"},
{"name": "atomicwrites", "version": "1.3.0"},
{"name": "attrs", "version": "19.1.0"}]
```

Файлы пакетов

Для генерации нативного пакета Python необходимо добавить в проект еще несколько файлов. Ради простоты ограничимся минимальным объемом необходи-

мых для генерации пакетов. Описание пакетов для `setuptools` приводится в файле `setup.py`, расположенном в каталоге верхнего уровня. В нашем примере проект этот файл выглядит следующим образом:

```
from setuptools import setup, find_packages

setup(
    name="hello-world",
    version="0.0.1",
    author="Example Author",
    author_email="stopspamer@ukr.net",
    url="example.com",
    description="A hello-world example package",
    packages=find_packages(),
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
)
```

Файл `setup.py` импортирует из модуля две вспомогательные функции: `setup` и `find_packages`. Именно для функции `setup` необходимо подробное описание пакетов. Функция `find_packages` представляет собой утилиту для автоматического обнаружения местоположения пакетов Python. Кроме того, этот файл импортирует `classifiers`, описывающие некоторые спектры пакетов, в частности лицензию, поддерживаемые операционные системы и версии Python. Это так называемые *trove-классификаторы* (trove classifiers), и в каталоге пакетов Python (<https://pypi.org/classifiers>) есть подробное описание всех доступных классификаторов. При загрузке в PyPI публикуются подробные описания пакетов.

Достаточно добавить всего один этот файл, чтобы сгенерировать пакет, в данном случае пакет *дистрибутив исходного кода* (source distribution package). Из-за отсутствия файла `README` при выполнении команды выводится предупреждение. Чтобы предотвратить это, добавьте пустой файл `README` в каталог верхнего уровня с помощью команды `touch README`.

Содержимое каталога проект должно выглядеть следующим образом:

```
hello-world
├── hello_world
│   ├── __init__.py
│   └── main.py
├── README
└── setup.py
```

1 directory, 2 files

Для генерации на его основе *дистрибутив исходного кода* выполните следующую команду:

```
python3 setup.py sdist
```

Результаты ее выполнения должны выглядеть примерно так:

```
$ python3 setup.py sdist
running sdist
running egg_info
writing hello_world.egg-info/PKG-INFO
writing top-level names to hello_world.egg-info/top_level.txt
writing dependency_links to hello_world.egg-info/dependency_links.txt
reading manifest file 'hello_world.egg-info/SOURCES.txt'
writing manifest file 'hello_world.egg-info/SOURCES.txt'
running check
creating hello-world-0.0.1
creating hello-world-0.0.1/hello_world
creating hello-world-0.0.1/hello_world.egg-info
copying files to hello-world-0.0.1...
copying README -> hello-world-0.0.1
copying setup.py -> hello-world-0.0.1
copying hello_world/__init__.py -> hello-world-0.0.1/hello_world
copying hello_world/main.py -> hello-world-0.0.1/hello_world
writing hello-world-0.0.1/setup.cfg
Creating tar archive
removing 'hello-world-0.0.1' (and everything under it)
```

В каталоге верхнего уровня проект появился новый каталог `dist`, он содержит *дистрибутив исходного кода* — файл `hello-world-0.0.1.tar.gz`. Если мы взглянем на структуру каталогов, то увидим, что он изменился:

```
hello-world
├── dist
│   └── hello-world-0.0.1.tar.gz
├── hello_world
│   ├── __init__.py
│   └── main.py
├── hello_world.egg-info
│   ├── dependency_links.txt
│   ├── PKG-INFO
│   ├── SOURCES.txt
│   └── top_level.txt
├── README
└── setup.py
```

3 directories, 9 files

Только что созданный файл `tar.gz` представляет собой пакет, который можно установить! Теперь его можно загрузить на PyPI, чтобы пользователи могли

сделать это. Поскольку мы придерживаемся схемы именования версий, у новички могут записать конкретную версию (в данном случае `0.0.1`) и шепотом сказать, был ли год раньше перед тем, как в функцию `setup()` дополнительных методов другие утилиты могут его обнулить и отображать информацию о нем, например создателя, описание и версию.

Установить файлы типа `tar.gz` напрямую можно с помощью утилиты Python `pip`. Попробуйте сделать это сами, указав в качестве аргумента путь к файлу:

```
$ pip install dist/hello-world-0.0.1.tar.gz
Processing ./dist/hello-world-0.0.1.tar.gz
Building wheels for collected packages: hello-world
  Building wheel for hello-world (setup.py) ... done
Successfully built hello-world
Installing collected packages: hello-world
Successfully installed hello-world-0.0.1
```

Каталог пакетов Python

Каталог пакетов Python (PyPI) — это репозиторий программного обеспечения на языке Python, в котором пользователи могут размещать пакеты Python, а также устанавливать их оттуда. Его поддержкой в качестве члена Python Software Foundation (<https://www.python.org/psf/>) занимается сообщество разработчиков Python благодаря помощи спонсоров.



Для этого придется убедиться, что вы записаны в тестовом экземпляре PyPI. Если у вас пока еще нет, зарегистрируйтесь через Интернет (<https://oreil.ly/VVx>). Имя пользователя и пароль этой учетной записи понадобятся для загрузки пакетов.

Поле для адреса электронной почты в файле `setup.py` содержит заглушку. Для публикации пакетов в каталоге необходимо изменить его на тот же адрес электронной почты, что и у владельца пакета в PyPI. Можете модифицировать остальные поля, например `author`, `url` и `description`, чтобы они лучше отражали созданный проект.

Для проверки пакета без отправки в реальную эксплуатацию его можно загрузить в тестовый экземпляр PyPI, ведущий себя точно так же, как и реальный, и позволяющий убедиться, что пакет работает должным образом.

Традиционно пакеты загружаются в PyPI с помощью `setuptools` и файла `setup.py`. Но новый подход, `twine`, упрощает эту задачу.

В начале этого раздела мы установили утилиту `twine` в нашей виртуальной среде. После этого ее можно применять для загрузки пакетов в тестовый экземпляр PyPI. Следующая команда загружает файл `tar.gz` и записывает имя пользователя и пароль:

```
$ twine upload --repository-url https://test.pypi.org/legacy/ \
  dist/hello-world-0.0.1.tar.gz
Uploading distributions to https://test.pypi.org/legacy/
Enter your username:
Enter your password:
```

Чтобы проверить, все ли прошло успешно, можем попробовать установить пакет с помощью `pip`:

```
$ python3 -m pip install --index-url https://test.pypi.org/simple/ hello-world
```

Может показаться, что в этой команде лишний пробел в URL PyPI, но URL каталогизируется не `/simple/`, `hello-world` — еще один аргумент, который указывает на имя устанавливаемого пакета Python.

Для настоящей промышленной версии необходимо создать учетную запись на основном экземпляре PyPI (<https://pypi.org/account/register>). Загрузка пакетов на *настоящий* PyPI ничем не отличается от загрузки в тестовый экземпляр, включая проверку.

Более строгие руководства по созданию пакетов могут включать следующие команды:

```
$ python setup.py register
$ python setup.py upload
```

Эти команды входят в набор утилит `setuptools` и, возможно, по-прежнему требуются для загрузки проектов в каталог пакетов. Однако в число возможностей утилиты `twine` входят безопасная аутентификация через HTTPS, также подпись пакетов с помощью `gpg`. `twine` работает даже тогда, когда не работает команда `python setup.py upload`, и, наконец, позволяет тестировать пакет перед загрузкой в каталог.

Осталось отметить, что иногда удобно создать `Makefile` и вставить в него команду `make` для автоматического развертывания проекта и сборки документации. Вот пример того, как это может работать:

```
deploy-pypi:
  pandoc --from=markdown --to=rst README.md -o README.rst
  python setup.py check --restructuredtext --strict --metadata
  rm -rf dist
  python setup.py sdist
  twine upload dist/*
  rm -f README.rst
```

Внутренний каталог пакетов

В некоторых случаях лучше поддерживать свой внутренний PyPI.

У компании, где Альфредо когда-то работал, были закрытые библиотеки, вовсе не предназначенные для общего доступа, так что необходимо было поддерживать свой экземпляр PyPI. Впрочем, у этого решения есть подводные камни. Экземпляры должны быть включены все зависимости и их версии, иначе могут возникнуть сбои при обновке. Программное обновление не может извлекать зависимости из различных источников одновременно! Несколько раз возникало, что для новой версии не удалось к кому-либо компонент, так что для завершения обновления приходилось загружать в каталог соответствующий пакет.

При локальном размещении пакетов *A*, зависящего от пакетов *B* и *C*, необходимо, чтобы все три (и все нужные их версии) существовали в одном экземпляре.

Внутренний PyPI ускоряет обновление пакетов, позволяет обеспечить конфиденциальность пакетов и, по сути, не так уж сложен в реализации.



Мы очень рекомендуем для создания внутреннего PyPI воспользоваться полнофункциональной утилитой `devpi`, обладающей возможностями зеркала копирования, предэксплуатационного тестирования, репликации и интеграции с Jenkins. В документации этого проекта (<https://doc.devpi.net>) можно найти подробную информацию и практические примеры.

Сначала создайте новый каталог `pypi`, чтобы получить нужную структуру для размещения пакетов, после чего создайте подкаталог, называющийся так же, как и пример пакет (`hello-world`). Названия подкаталогов соответствуют названиям с их пакетами:

```
$ mkdir -p pypi/hello-world
$ tree pypi
pypi
└─ hello-world
```

1 directory, 0 files

Теперь скопируйте файл `tar.gz` в каталог `hello-world`. Итоговый вариант структуры каталогов должен выглядеть так:

```
$ tree pypi
pypi
└─ hello-world
   └─ hello-world-0.0.1.tar.gz
```

1 directory, 1 file

Следующий шаг — создание веб-сервера с включенной статической индексацией. В Python есть встроенный веб-сервер, вполне подходящий для наших

экспериментов, причем даже с включенной по умолчанию в том тической индексацией! Перейдите в каталог `pypi`, в котором находится пакет `hello-world`, и запустите встроенный веб-сервер:

```
$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

В новом окне терминала создайте временную виртуальную среду для установки пакета `hello-world` из локального экземпляра PyPI. Активируйте ее и, наконец, попробуйте установить пакет, указав в команде `pip` свой локальный URL:

```
$ python3 -m venv /tmp/local-pypi
$ source /tmp/local-pypi/bin/activate
(local-pypi) $ pip install -i http://localhost:8000/ hello-world
Looking in indexes: http://localhost:8000/
Collecting hello-world
  Downloading http://localhost:8000/hello-world/hello-world-0.0.1.tar.gz
Building wheels for collected packages: hello-world
  Building wheel for hello-world (setup.py) ... done
Successfully built hello-world
Installing collected packages: hello-world
Successfully installed hello-world-0.0.1
```

В процессе, где запущен модуль `http.server`, при этом должны появиться записи журналов, отображающие все выполненные программой установки для извлечения пакета `hello-world`, и запросы:

```
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
127.0.0.1 [09:58:37] "GET / HTTP/1.1" 200 -
127.0.0.1 [09:59:39] "GET /hello-world/ HTTP/1.1" 200 -
127.0.0.1 [09:59:39] "GET /hello-world/hello-world-0.0.1.tar.gz HTTP/1.1" 200
```

Для неходящейся в промышленной эксплуатации среды не добится более мощный веб-сервер. Для простоты в этом примере мы воспользовались модулем `http.server`, но он не умеет обслуживать одновременно несколько запросов и плохо масштабируется.



При создании локального каталога пакетов без утилит, подобных `devpi`, имеет смысл задаться подробной спецификацией, включающей описание стандартных звеньев для структуры каталогов. Ее вы можете найти в PEP 503 (<https://oreil.ly/sRcAe>).

Создание пакетов для Debian

Если вы рассчитываете установить свой проект в операционной системе Debian (или основанных на Debian дистрибутивов, например Ubuntu), понадобятся дополнительные файлы. Понимание того, какие файлы нужны и как они используются инструментами создания пакетов для Debian, значительно

повышит эффективность процесс генерации подходящих для установки пакетов `.deb` и решения возникающих проблем.

В некоторых из этих текстовых файлов требуется *чрезвычайно* точное форматирование, и если формат хоть чуть-чуть не соблюден, пакет нормально установиться не будет.



В этом разделе мы предполагаем, что пакет создается в Debian или основанном на Debian дистрибутиве, чтобы проще было установить и использовать нужные утилиты создания пакетов.

Файлы пакета

Для создания пакетов программ в Debian нам понадобится каталог `debian`, содержащий несколько файлов. Чтобы уменьшить объем необходимого для генерации пакетов, опустим большинство возможных опций, в частности выполнение наборов тестов перед завершением сборки или объявление нескольких версий Python.

Создайте каталог `debian`, в котором будут располагаться все необходимые файлы. Структура проекта `hello-world` должна выглядеть следующим образом:

```
$ tree
.
├── debian
├── hello_world
│   ├── __init__.py
│   └── main.py
├── README
└── setup.py
```

2 directories, 4 files



Обратите внимание на то, что этот каталог содержит файлы `setup.py` и `README` из предыдущего раздела, посвященного созданию интерпретаторных пакетов Python. Они необходимы для утилит Debian, генерирующих пакеты `.deb`.

Файл `changelog`. Правильно сформировать этот файл вручную — задача порой непростая. Ошибки неправильного форматирования этого файлаискать очень сложно. В большинстве схем создания пакетов Debian для улучшения возможностей отладки используется утилита `dch`.

Раньше я игнорировал данный совет и пытался создать этот файл вручную. В итоге я просто тратил время впустую, поскольку сообщения об ошибках были

не слишком информативны, заметить возможные проблемы очень непросто. Далее приведен пример письма из файла `changelog`, вызвавшего проблемы:

```
--Alfredo Deza <alfredo@example.com> Sat, 11 May 2013 2:12:00 -0800
```

Этот текст вызвал следующее сообщение об ошибке:

```
parsechangelog/debian: warning: debian/changelog(17): found start of entry
where expected more change data or trailer
```

Можете сразу заметить, что было исправлено?

```
-- Alfredo Deza <alfredo@example.com> Sat, 11 May 2013 2:12:00 -0800
```

Причиной проблемы было отсутствие *пробел* между тире и моим именем. Избавьте себя от мучений и воспользуйтесь `dch`. Этот утилит входит в состав пакета `devscripts`:

```
$ sudo apt-get install devscripts
```

Число опций утилиты командной строки `dch` очень велико, так что рекомендуем просмотреть ее документацию (основной страницы вполне достаточно). Мы воспользуемся ею для первоначального создания журнала изменений (для этого нам потребуется однострочный файл `--create`). Прежде чем запустить ее, экспортируйте свои полное имя и адрес электронной почты, чтобы включить их в генерируемый файл:

```
$ export DEBEMAIL="alfredo@example.com"
$ export DEBFULLNAME="Alfredo Deza"
```

Теперь запустите `dch`, чтобы сгенерировать журнал изменений:

```
$ dch --package "hello-world" --create -v "0.0.1" \
-D stable "New upstream release"
```

Только что созданный файл должен выглядеть примерно так:

```
hello-world (0.0.1) stable; urgency=medium
```

```
* New upstream release
```

```
-- Alfredo Deza <alfredo@example.com> Thu, 11 Apr 2019 20:28:08 -0400
```



Журналы изменений Debian учитывают специфику пакетов Debian. Если формат не соответствует или кто-то другой информирует о необходимости обновления, вполне можно вести отдельный журнал изменений для проекта. Во многих проектах файл `changelog` Debian хранится в виде отдельного файла, предназначенного только для Debian.

Файл control. В этом файле задаются название пакета, его описание и все необходимые для сборки и рботы пакеты зависимости. Он также отличается жестко заданным форматом, но его редко приходится менять (в отличие от файла `changelog`). Этот файл требует использования Python 3, также следуют рекомендациям по именованиям Python для Debian.



При переходе от Python 2 к Python 3 большинство дистрибутивов остались в применении для пакетов Python 3 схемы именования `python3-{название пакета}`.

После добавления зависимостей, соглашений о именованиях и короткого описания файл должен выглядеть вот так:

```
Source: hello-world
Maintainer: Alfredo Deza <alfredo@example.com>
Section: python
Priority: optional
Build-Depends:
    debhelper (>= 11~),
    dh-python,
    python3-all
    python3-setuptools
Standards-Version: 4.3.0

Package: python3-hello-world
Architecture: all
Depends: ${misc:Depends}, ${python3:Depends}
Description: An example hello-world package built with Python 3
```

Прочие необходимые файлы. Для генерации пакета Debian необходимо еще несколько файлов. Большинство из них состоят всего из нескольких строк и меняются нечасто.

Файл `rules` представляет собой исполняемый файл, вызывающий Debian, что нужно запустить для генерации пакета, в данном случае он должен выглядеть так:

```
#!/usr/bin/make -f

export DH_VERBOSE=1

export PYBUILD_NAME=remoto

%:
    dh $@ --with python3 --buildsystem=pybuild
```

Файл `compat` задает уровень совместимости с `debhelper` (еще одна утилита для создания пакетов), рекомендуемое значение 10. Если вы получите связанное

с ним сообщение об ошибке, возможно, стоит проверить, не нужно ли большее значение:

```
$ cat compat
10
```

Без лицензии процесс сборки может не работать, так что имеет смысл указать лицензию явно. В данном примере используется лицензия MIT, и вот как должен выглядеть файл `debian/copyright`:

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0
Upstream-Name: hello-world
Source: https://example.com/hello-world
Files: *
Copyright: 2019 Alfredo Deza
License: Expat
License: Expat

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

.
The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

В итоге после добавления всех этих новых файлов в каталог `debian` проект `hello-world` выглядит вот так:

```
.
├── debian
│   ├── changelog
│   ├── compat
│   ├── control
│   ├── copyright
│   └── rules
├── hello_world
│   ├── __init__.py
│   └── main.py
├── README
└── setup.py
```

2 directories, 9 files

Генерация двоичного файла

Для генерации двоичного файла используйте утилиту командной строки `debuild`. В своем примере проект мы не stiamo подписывать пакет (для создания цифровой подписи требуется GPG-ключ), в документации `debuild` есть пример, позволяющий пропустить этап подписывания. Сценарий запускается изнутри дерева каталогов исходного кода для сборки только двоичного пакета. Вот команда, подходящая для проекта `hello-world`:

```
$ debuild -i -us -uc -b
...
dpkg-deb: building package 'python3-hello-world'
in '../python3-hello-world_0.0.1_all.deb'.
...
dpkg-genbuildinfo --build=binary
dpkg-genchanges --build=binary >../hello-world_0.0.1_amd64.changes
dpkg-genchanges: info: binary-only upload (no source code included)
dpkg-source -i --after-build hello-world-debian
dpkg-buildpackage: info: binary-only upload (no source code included)
Now running lintian hello-world_0.0.1_amd64.changes ...
E: hello-world changes: bad-distribution-in-changes-file stable
Finished running lintian.
```

Теперь в верхнем каталоге должен появиться файл `python3-hello-world_0.0.1_all.deb`. При вызове `lintian` (линтер Debian для создания пакетов) в самом конце возникнет сообщение о некорректном дистрибутиве. Об этом волноваться не стоит, ведь мы и не ориентировались на какой-то конкретный дистрибутив (например, Debian Buster), собирали пакет, который можно было бы установить в любом дистрибутиве на основе Debian, совместимом со всеми зависимостями (в данном случае только Python 3).

Репозитории Debian

Существует множество утилит для автоматизации работы с репозиториями Debian, но совсем не помешает разобраться в схеме их создания (Альфредо дже помог разобраться с одним репозиторием (<https://oreil.ly/hJMgY>), предзнаменительный как для RPM, так и для Debian). Убедитесь, что созданный нами бинарный пакет находится в нужном месте:

```
$ mkdir /opt/binaries
$ cp python3-hello-world_0.0.1_all.deb /opt/binaries/
```

Для этого требуется необходимо, чтобы был установлен утилита `reprepro`:

```
$ sudo apt-get install reprepro
```

Создайте новый каталог где-нибудь в системе для хранения пакетов. В данном примере его роль играет каталог `/opt/repo`. Для основных настроек репозитория

пон добится файл `distributions` с описанием содержимого репозитория, который выглядит вот так:

```
Codename: sid
Origin: example.com
Label: example.com
Architectures: amd64 source
DscIndices: Sources Release .gz .bz2
DebIndices: Packages Release . .gz .bz2
Components: main
Suite: stable
Description: example repo for hello-world package
Contents: .gz .bz2
```

Сохраните этот файл по адресу `/opt/repo/conf/distributions`. Создайте еще один каталог, в котором будет располагаться с mirrors репозиторий:

```
$ mkdir /opt/repo/debian/sid
```

Для создания репозитория необходимо узнать утилите `reprepro`, что нужно использовать созданный нами файл `distributions`, роль нового каталога должен играть `/opt/repo/debian/sid`. Наконец, укажите в нем созданный бинарный файл в качестве целевого для дистрибутива `sid Debian`:

```
$ reprepro --confdir /opt/repo/conf/distributions -b /opt/repo/debian/sid \
  -C main includedeb sid /opt/binaries/python3-hello-world_0.0.1_all.deb
Exporting indices...
```

Этим командой создан репозиторий для дистрибутива `sid Debian`. Ее можно приспособить для различных дистрибутивов на основе Debian, например Ubuntu Bionic. Для этого достаточно изменить `sid` на `bionic`.

Следующий шаг после создания репозитория — добиться, чтобы он работал как нужно. Для промышленной эксплуатации подойдет надежный веб-сервер, например Apache или Nginx, но для тестирования пример мы воспользуемся модулем `http.server` Python. Перейдите в каталог, содержащий репозиторий, и запустите веб-сервер:

```
$ cd /opt/repo/debian/sid
$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Необходимо узнать системе управления пакетами Debian `apt` (`Aptitude`) новое местоположение пакетов. Соответствующая конфигурация представляет собой простой файл, содержащий всего одну строку со ссылкой на URL и компоненты нового репозитория. Создайте файл `/etc/apt/sources.list.d/hello-world.list`. Его содержимое должно выглядеть так:

```
$ cat /etc/apt/sources.list.d/hello-world.list
deb [trusted=yes] http://localhost:8000/ sid main
```

Н стройк [trusted=yes] озн ч ет, что apt не должн требов ть подпис нных п - кетов. В подпис нных должным обр зом репозиториях эт н стройк не нужн .

После созд ния ук з ного ф йл обновите apt, чтобы он обн ружил новое местоположение репозитория и н шл (и уст новил) п кет hello-world:

```
$ sudo apt-get update
Ign:1 http://localhost:8000 sid InRelease
Get:2 http://localhost:8000 sid Release [2,699 B]
Ign:3 http://localhost:8000 sid Release.gpg
Get:4 http://localhost:8000 sid/main amd64 Packages [381 B]
Get:5 http://localhost:8000 sid/main amd64 Contents (deb) [265 B]
Fetched 3,345 B in 1s (6,382 B/s)
Reading package lists... Done
```

При поиске п кет python3-hello-world мы видим опис ние, внесенное в ф йл distributions в процессе н стройки reprepro:

```
$ apt-cache search python3-hello-world
python3-hello-world - An example hello-world package built with Python 3
```

Уст новк и уд ление п кет должны выполняться без проблем:

```
$ sudo apt-get install python3-hello-world
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  python3-hello-world
0 upgraded, 1 newly installed, 0 to remove and 48 not upgraded.
Need to get 2,796 B of archives.
Fetched 2,796 B in 0s (129 kB/s)
Selecting previously unselected package python3-hello-world.
(Reading database ... 242590 files and directories currently installed.)
Preparing to unpack .../python3-hello-world_0.0.1_all.deb ...
Unpacking python3-hello-world (0.0.1) ...
Setting up python3-hello-world (0.0.1) ...
$ sudo apt-get remove --purge python3-hello-world
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be REMOVED:
  python3-hello-world*
0 upgraded, 0 newly installed, 1 to remove and 48 not upgraded.
After this operation, 19.5 kB disk space will be freed.
Do you want to continue? [Y/n] Y
(Reading database ... 242599 files and directories currently installed.)
Removing python3-hello-world (0.0.1) ...
```

Создание пакетов RPM

Как и при создании пакетов Debian, в ходе работы с RPM уже должен быть подготовлен возможность генерации и тивного пакет Python с помощью файла `setup.py`. Однако, в отличие от Debian, где требовалось много файлов, создание пакетов RPM требует всего одного файла — `spec`. Систем управления пакетами RPM (известная ранее как систем управления пакетами Red Hat) прекрасно подойдет для этой цели, если роль целевого дистрибутива Linux играет CentOS или Fedora.

Файл `spec`

В простейшем варианте файл `spec` (в этом примере он называется `hello-world.spec`) ничего сложного нет, большинство разделов его говорят с вами о себе. Его можно даже сгенерировать с помощью `setuptools`:

```
$ python3 setup.py bdist_rpm --spec-only
running bdist_rpm
running egg_info
writing hello_world.egg-info/PKG-INFO
writing dependency_links to hello_world.egg-info/dependency_links.txt
writing top-level names to hello_world.egg-info/top_level.txt
reading manifest file 'hello_world.egg-info/SOURCES.txt'
writing manifest file 'hello_world.egg-info/SOURCES.txt'
writing 'dist/hello-world.spec'
```

Полученный в результате файл `dist/hello-world.spec` будет выглядеть примерно так:

```
%define name hello-world
%define version 0.0.1
%define unmangled_version 0.0.1
%define release 1

Summary: A hello-world example package
Name: %{name}
Version: %{version}
Release: %{release}
Source0: %{name}-%{unmangled_version}.tar.gz
License: MIT
Group: Development/Libraries
BuildRoot: %{_tmppath}/%{name}-%{version}-%{release}-buildroot
Prefix: %{_prefix}
BuildArch: noarch
Vendor: Example Author <author@example.com>
Url: example.com
```

```
%description
A Python3 hello-world package

%prep
%setup -n %{name}-%{unmangled_version} -n %{name}-%{unmangled_version}

%build
python3 setup.py build

%install
python3 setup.py install --single-version-externally-managed -O1 \
--root=$RPM_BUILD_ROOT --record=INSTALLED_FILES

%clean
rm -rf $RPM_BUILD_ROOT

%files -f INSTALLED_FILES
%defattr(-,root,root)
```

Хотя он выглядит простым, сразу же возникает потенциальная проблема: версию придется обновлять каждый раз. Этот процесс напоминает файл `changelog` Debian, который следует обновлять с помощью утилиты `bump` для каждой новой версии.

Определенные перспективы несет интеграция `setuptools`, позволяющая при необходимости выполнять дальнейшую модификацию этого файла и копировать его в корневой каталог проекта для постоянного хранения. В некоторых проектах используется базовый шаблон, заполняемый в процессе сборки для генерации файла `spec`. В проекте Ceph (<https://ceph.com>) выпускаемым версиям с помощью системы контроля версий (Git) присвоен тег, который сценарии выпуска версий применяют к упомянутому шаблону через `Makefile`. Стоит отметить, что существуют и дополнительные методы для дальнейшей автоматизации этого процесса.



Генерировать файл `spec` не всегда имеет смысл, поскольку некоторые его разделы приходится жестко «зашивать», следуя ким-либо правилам создания дистрибутива или из-за конкретных зависимостей, не входящих в сгенерированный файл. В подобных случаях лучше сгенерировать его один раз, затем внести нужные изменения в строки и сохранить, после чего сделать полученный файл `spec` частью проекта.

Генерирование бинарного файла

Существует несколько утилит для генерирования бинарных файлов RPM, одна из них — утилита командной строки `rpmbuild`:

```
$ sudo yum install rpm-build
```



Утилита командной строки называется `rpm-build`, пакет — `rpm-build`, так что убедитесь, что `rpm-build` (утилита командной строки) доступен для использования в терминале.

Для создания бинарного файла утилите `rpm-build` необходим определенная структура каталогов. После их создания в каталоге `SOURCES` должен присутствовать файл `source` (сгенерированный `setuptools` файлом `tar.gz`). Далее покажем, как следует создать эту структуру и как он будет выглядеть в итоге:

```
$ mkdir -p /opt/repo/centos/{SOURCES,SRPMS,SPECS,RPMS,BUILD}
$ cp dist/hello-world-0.0.1.tar.gz /opt/repo/centos/SOURCES/
$ tree /opt/repo/centos
/opt/repo/centos
├── BUILD
├── BUILDROOT
├── RPMS
├── SOURCES
│   └── hello-world-0.0.1.tar.gz
├── SPECS
└── SRPMS
```

6 directories, 1 file

Эта структура каталогов необходим всегда, причем по умолчанию утилита `rpm-build` ожидает ее наличия в домашнем каталоге. Чтобы не сваливать все в одну кучу, воспользуемся другим местом в файловой системе (`/opt/repo/centos`). Это значит, что нам нужно попросить утилиту `rpm-build` действовать этот каталог. В результате бланка файла `-ba` будут сгенерированы бинарный файл и пакет `source` (мы немного сократили выводимое командой):

```
$ rpm-build -ba --define "_topdir /opt/repo/centos" dist/hello-world.spec
...
Executing(%build): /bin/sh -e /var/tmp/rpm-tmp.CmG0dp
running build
running build_py
creating build
creating build/lib
creating build/lib/hello_world
copying hello_world/main.py -> build/lib/hello_world
copying hello_world/__init__.py -> build/lib/hello_world
Executing(%install): /bin/sh -e /var/tmp/rpm-tmp.CQgOKD
+ python3 setup.py install --single-version-externally-managed \
-01 --root=/opt/repo/centos/BUILDROOT/hello-world-0.0.1-1.x86_64
running install
writing hello_world.egg-info/PKG-INFO
writing dependency_links to hello_world.egg-info/dependency_links.txt
writing top-level names to hello_world.egg-info/top_level.txt
reading manifest file 'hello_world.egg-info/SOURCES.txt'
writing manifest file 'hello_world.egg-info/SOURCES.txt'
```



```

running install_scripts
writing list of installed files to 'INSTALLED_FILES'
Processing files: hello-world-0.0.1-1.noarch
Provides: hello-world = 0.0.1-1
Wrote: /opt/repo/centos/SRPMS/hello-world-0.0.1-1.src.rpm
Wrote: /opt/repo/centos/RPMS/noarch/hello-world-0.0.1-1.noarch.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.gcIJgT
+ umask 022
+ cd /opt/repo/centos//BUILD
+ cd hello-world-0.0.1
+ rm -rf /opt/repo/centos/BUILDROOT/hello-world-0.0.1-1.x86_64
+ exit 0

```

В структуре каталогов в `/opt/repo/centos` появится множество новых файлов, но нас интересует только один, с расширением `noarch.rpm`:

```

$ tree /opt/repo/centos/RPMS
/opt/repo/centos/RPMS
├── noarch
│   └── hello-world-0.0.1-1.noarch.rpm

```

1 directory, 1 file

Этот файл представляет собой подходящий для установки пакет RPM! Утилита генерирует и другие полезные пакеты, которые также можно опубликовать (згляните, например, в `/opt/repo/centos/SRPMS`).

Репозитории RPM

Для создания репозитория RPM мы возьмем утилиту командной строки `createrepo`. Она создает метаданные репозитория (метаданные RPM в формате XML) из бинарных файлов, найденных в указанном каталоге. В этом разделе мы создадим (и разместим в репозитории) бинарный файл типа `noarch`:

```
$ sudo yum install createrepo
```

Можете создать репозиторий там же, где мы генерировали пакет `noarch`, или воспользоваться новым (пустым) каталогом. При необходимости создайте новые бинарные файлы. А затем скопируйте пакеты:

```

$ mkdir -p /var/www/repos/centos
$ cp -r /opt/repo/centos/RPMS/noarch /var/www/repos/centos

```

Затем запустите утилиту `createrepo` для создания метаданных:

```

$ createrepo -v /var/www/repos/centos/noarch
Spawning worker 0 with 1 pkgs
Worker 0: reading hello-world-0.0.1-1.noarch.rpm
Workers Finished
Saving Primary metadata

```

```

Saving file lists metadata
Saving other metadata
Generating sqlite DBs
Starting other db creation: Thu Apr 18 09:13:35 2019
Ending other db creation: Thu Apr 18 09:13:35 2019
Starting filelists db creation: Thu Apr 18 09:13:35 2019
Ending filelists db creation: Thu Apr 18 09:13:35 2019
Starting primary db creation: Thu Apr 18 09:13:35 2019
Ending primary db creation: Thu Apr 18 09:13:35 2019
Sqlite DBs complete

```

И хотя пакет `x86_64` не существует, повторите вызов утилиты `createrepo` для нового каталога, чтобы не получить предупреждений `yum` в дальнейшем:

```

$ mkdir /var/www/repos/centos/x86_64
$ createrepo -v /var/www/repos/centos/x86_64

```

Для выдачи данных из этого каталога по HTTP давайте воспользуемся модулем `http.server`:

```

$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...

```

Для доступа к этому репозиторию `yum` понадобятся строки в файле `repo`. Создайте его — `/etc/yum.repos.d/hello-world.repo`. Он должен выглядеть вот так:

```

[hello-world]
name=hello-world example repo for noarch packages
baseurl=http://0.0.0.0:8000/$basearch
enabled=1
gpgcheck=0
type=rpm-md
priority=1

[hello-world-noarch]
name=hello-world example repo for noarch packages
baseurl=http://0.0.0.0:8000/noarch
enabled=1
gpgcheck=0
type=rpm-md
priority=1

```

Обратите внимание на то, что значение `gpgcheck` равно 0. Это значит, что мы не подписывали пакеты и утилита `yum` не должна пытаться проверить подписи, из-за чего в этом примере могли бы возникнуть ошибки. Теперь можно выполнить поиск пакетов, в ходе которого мы получим описание в виде чистых выводимых данных:

```

$ yum --enablerepo=hello-world search hello-world
Loaded plugins: fastestmirror, priorities
Loading mirror speeds from cached hostfile

```

```

* base: reflector.westga.edu
* epel: mirror.vcu.edu
* extras: mirror.steadfastnet.com
* updates: mirror.mobap.edu
base | 3.6 kB
extras | 3.4 kB
hello- world | 2.9 kB
hello-world-noarch | 2.9 kB
updates | 3.4 kB
8 packages excluded due to repository priority protections
=====
matched: hello-world
=====
hello-world.noarch : A hello-world example package

```

Функция поиска пакетов должна быть такой, чтобы пакет должен пройти успешно:

```

$ yum --enablerepo=hello-world install hello-world
Loaded plugins: fastestmirror, priorities
Loading mirror speeds from cached hostfile
* base: reflector.westga.edu
* epel: mirror.vcu.edu
* extras: mirror.steadfastnet.com
* updates: mirror.mobap.edu
8 packages excluded due to repository priority protections
Resolving Dependencies
--> Running transaction check
---> Package hello-world.noarch 0:0.0.1-1 will be installed
--> Finished Dependency Resolution

Dependencies Resolved
Installing:
  hello-world          noarch          0.0.1-1          hello-world-noarch

Transaction Summary
Install 1 Package

Total download size: 8.1 k
Installed size: 1.3 k
Downloading packages:
hello-world-0.0.1-1.noarch.rpm | 8.1 kB
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Installing : hello-world-0.0.1-1.noarch
  Verifying  : hello-world-0.0.1-1.noarch

Installed:
  hello-world.noarch 0:0.0.1-1

Complete!

```

Удаление также проходит без проблем:

```
$ yum remove hello-world
Loaded plugins: fastestmirror, priorities
Resolving Dependencies
--> Running transaction check
---> Package hello-world.noarch 0:0.0.1-1 will be erased
--> Finished Dependency Resolution

Dependencies Resolved
Removing:
  hello-world                noarch                0.0.1-1                @hello-world-noarch

Transaction Summary
Remove 1 Package

Installed size: 1.3 k
Is this ok [y/N]: y
Downloading packages:
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Erasing      : hello-world-0.0.1-1.noarch
  Verifying    : hello-world-0.0.1-1.noarch
Removed:
  hello-world.noarch 0:0.0.1-1
Complete!
```

Модуль `http.server` должен при этом вывести сообщения о каких-то действиях, демонстрирующих, что утилита `yum` обратилась для получения пакетов `hello-world`:

```
[18/Apr/2019 03:37:24] "GET /x86_64/repodata/repomd.xml HTTP/1.1"
[18/Apr/2019 03:37:24] "GET /noarch/repodata/repomd.xml HTTP/1.1"
[18/Apr/2019 03:37:25] "GET /x86_64/repodata/primary.sqlite.bz2 HTTP/1.1"
[18/Apr/2019 03:37:25] "GET /noarch/repodata/primary.sqlite.bz2 HTTP/1.1"
[18/Apr/2019 03:56:49] "GET /noarch/hello-world-0.0.1-1.noarch.rpm HTTP/1.1"
```

Диспетчеризация с помощью systemd

`systemd` — диспетчер системы и сервисов (system and service manager) для операционной системы Linux, известный также как подсистема инициализации (init system). Он играет роль подсистемы инициализации по умолчанию во многих дистрибутивах, в частности Debian и Red Hat. Вот лишь некоторые из возможностей `systemd`:

- удобное пространство параметров;
- точки подключения и триггеры для поведения по требованию;

- журнализация интеграции;
- возможность использования других модулей для координации сложных вриентов загрузки.

У **systemd** есть множество других восхитительных спектров, например возможности работы с сетью, DNS и даже монтирование устройств. Идея удобной работы с процессами в Python всегда был крепким орешком — в какой-то момент существовало сразу несколько непонимающих *подсистему инициализации* проектов в Python, каждый со своими настройками и API. **systemd** благодаря своей повсеместной доступности обеспечил переносимость и упростил взаимодействие.



supervisord (<http://supervisord.org>) и **circus** (<https://oreil.ly/adGEj>) — два широко известных средства управления процессами в Python.

Недавно Альфредо написал в Python маленький HTTP API для промышленной эксплуатации. Этот проект был перенесен с **supervisord** на **circus**, и все работало нормально. К сожалению, ограничение среды эксплуатации требовало интеграции **systemd** в операционную систему. Этот переход прошел не так гладко, поскольку утилита **systemd** была относительно новой, но по его завершении мы получили преимущество в виде единообразия при разброске и промышленной эксплуатации, благодаря чему смогли выявлять проблемы интеграции и более ретроспективно анализировать разброс. При выпуске промышленной версии API во время работы с **systemd** мы чувствовали себя уже достаточно уверенно, чтобы решить проблемы и даже выполнять тонкую настройку параметров для решения внешних проблем (случилось ли видеть сбой сценария `init` из-за отключения сети?).

В этом разделе мы создадим небольшой HTTP-сервис, доступный при загрузке системы, который можно перезапустить в любой момент. Настройка в `unit-f` обеспечит журнализацию и доступность требуемых системных ресурсов до запуска.

Долгоживущие процессы

Процессы, которые должны выполняться практически все время, — отличные кандидаты для внесения в список работающих под управлением **systemd**. Представьте себе, например, DNS или сервер электронной почты — они работают в системе непрерывно, ими нужно управлять для захвата журналов или перезапуска при изменениях конфигурации.

Мы воспользуемся м леньким сервером HTTP API, в основе которого лежит веб-фреймворк Pecan (<https://www.pecanpy.org/>).



Ничего связ нного конкретно с Pecan в этом р зделе нет, т к что примеры подходят и для других фреймворков или долгоживущих процессов.

Настройка

Выберем постоянное местор сположение по дресу `/opt/http` для созд ния к - т лог проект , после чего созд им новую вирту льную среду и уст н влив ем фреймворк Pecan:

```
$ mkdir -p /opt/http
$ cd /opt/http
$ python3 -m venv .
$ source bin/activate
(http) $ pip install "pecan==1.3.3"
```

У Pecan есть встроенные вспомог тельные функции для созд ния необходимых ф йлов и к т логов для н шего пример проект . С помощью Pecan можно созд ть простейший б зовый проект API HTTP, способный привязыв ться к `systemd`. У версии 1.3.3 есть две опции — `base` и `rest-api`:

```
$ pecan create api rest-api
Creating /opt/http/api
Recurring into +package+
  Creating /opt/http/api/api
...
Copying scaffolds/rest-api/config.py_tmpl to /opt/http/api/config.py
Copying scaffolds/rest-api/setup.cfg_tmpl to /opt/http/api/setup.cfg
Copying scaffolds/rest-api/setup.py_tmpl to /opt/http/api/setup.py
```



В жно использов ть один и тот же путь, поскольку в д льнейшем он будет применяться при н стройке сервис с помощью `systemd`.

Бл год ря включению туд ск ффолдинг проект мы без всяких усилий полу ч ем полнофункциона льный проект. Он д же содержит ф йл `setup.py` со всем необходимым, готовый к созд нию н тивного п кет Python! Уст новим проект¹, чтобы можно было его з пустить:

¹ Перед этим необходимо перейти в к т лог `api:cd api/`. — *Примеч. пер.*

```
(http) $ python setup.py install
running install
running bdist_egg
running egg_info
creating api.egg-info
...
creating dist
creating 'dist/api-0.1-py3.6.egg' and adding 'build/bdist.linux-x86_64/egg'
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
Processing api-0.1-py3.6.egg
creating /opt/http/lib/python3.6/site-packages/api-0.1-py3.6.egg
Extracting api-0.1-py3.6.egg to /opt/http/lib/python3.6/site-packages
...
Installed /opt/http/lib/python3.6/site-packages/api-0.1-py3.6.egg
Processing dependencies for api==0.1
Finished processing dependencies for api==0.1
```

Утилита командной строки `pecan` требует файл конфигурации. Он уже был создан для вас в процессе скелетонирования и располагается в каталоге верхнего уровня. Запустите сервер с `config.py` в качестве параметра:

```
(http) $ pecan serve config.py
Starting server in PID 17517
serving on 0.0.0.0:8080, view at http://127.0.0.1:8080
```

При проверке его в браузере вы получите текстовое сообщение. Вот что отображается, если воспользоваться командой `curl`:

```
(http) $ curl localhost:8080
Hello, World!
```

С помощью команды `pecan serve config.py` запускается выполнение долгоживущего процесса. Единственный способ остановить его — отправить встроенное исключение `KeyboardInterrupt` с помощью сочетания клавиш `Ctrl+C`. Для повторного его запуска требуются активация виртуальной среды и повторное выполнение той же команды `pecan serve`.

Юниты systemd

В отличие от более старых систем инициализации, работающих с исполняемыми сценариями, `systemd` работает с неформатированными текстовыми файлами. Итоговая версия юнит-файла выглядит вот так:

```
[Unit]
Description=hello world pecan service
After=network.target

[Service]
Type=simple
```

```
ExecStart=/opt/http/bin/pecan serve /opt/http/api/config.py
WorkingDirectory=/opt/http/api
StandardOutput=journal
StandardError=journal
```

```
[Install]
WantedBy=multi-user.target
```

Сохраните этот файл с названием `hello-world.service`. Далее в этом разделе мы скопируем его туда, где он должен в итоге находиться.

Важно обеспечить правильность всех названий разделов и инструкций конфигурации, поскольку они все учитывают регистр клавиш. Если названия хоть чуть-чуть расходятся, ничего работать не будет. Рассмотрим во всех подробностях каждый из разделов для HTTP-сервиса.

- **Unit.** Приводит описание и включает инструкцию `After`, указывающую `systemd`, что перед выполнением данного юнит-сервиса сетевая среда должна быть в рабочем состоянии. Требования других юнитов могут быть еще сложнее не только для запуска сервиса, но и *после* его запуска! Очень удобны, в частности, инструкции `Condition` и `Wants`.
- **Service.** Этот раздел необходим только при настройке юнит-сервиса. По умолчанию `Type=simple`. От сервисов этого типа не следует порождать новые — они должны оставаться на переднем плане, чтобы `systemd` мог управлять их работой. Строка `ExecStart` задает команду, которую необходимо выполнить для запуска сервиса. *Обязательно* используйте абсолютные пути во избежание проблем с поиском нужных файлов.

Хотя это и не обязательно, я включил инструкцию `WorkingDirectory`, чтобы гарантировать выполнение процесса в том же каталоге, где располагается приложение. В случае изменений в будущем соответствующее приложению местоположение может пригодиться.

Инструкции `StandardOutput` и `StandardError` очень удобны в работе и демонстрируют широту возможностей `systemd`. Они берут на себя все журналы, выводимые механизмами `systemd` через `stdout` и `stderr`. Мы продемонстрируем это подробнее, когда будем обсуждать взаимодействие с сервисом.

- **Install.** Инструкция `WantedBy` указывает, когда запускать юнит в случае его активации. `multi-user.target` эквивалентно `runlevel 3` (обычный уровень выполнения для сервера, которому для штатной работы достаточно текстового терминала). С помощью этой настройки систем определяет поведение после активации. После активации создается символическая ссылка в каталоге `multi-user.target.wants`.

Установка юнита

Указанный файл конфигурации необходимо поместить в определенное место, чтобы `systemd` смог его найти и загрузить. Поддерживается несколько возможных местоположений, но для созданных или управляемых администратором юнитов предпочтен каталог `/etc/systemd/system`.



Имеет смысл проверить, что инструкция `ExecStart` работает с этими путями. Использование абсолютных путей повысит риск случайной опечатки. Для проверки выполните всю строку в терминале и посмотрите, будет ли результат выполнения примерно таким:

```
$ /opt/http/bin/pecan serve /opt/http/api/config.py
Starting server in PID 20621
serving on 0.0.0.0:8080, view at http://127.0.0.1:8080
```

После проверки работы команды скопируйте unit-файл в этот каталог, дайте ему название `hello-world.service`:

```
$ cp hello-world.service /etc/systemd/system/
```

После этого необходимо перезагрузить `systemd`, чтобы дать ей знать о новом юните:

```
$ systemctl daemon-reload
```

Теперь сервис находится в полностью рабочем состоянии, его можно запустить и остановить. Для проверки состояния процесса можно воспользоваться подкомандой `status`. Вкратце рассмотрим различные команды, с помощью которых можно взаимодействовать с сервисом. Вкратце посмотрим, как познать ли его `systemd`. Вот как он должен вести себя и как должен выглядеть результат работы:

```
$ systemctl status hello-world
• hello-world.service - hello world pecan service
  Loaded: loaded (/etc/systemd/system/hello-world.service; disabled; )
  Active: inactive (dead)
```

Поскольку наш сервис не запущен, неудивительно, что он отмечен как `dead`. Запустите сервис и снова проверьте состояние (утилит `curl` должен сообщить, что на порте `8080` ничего не запущено):

```
$ curl localhost:8080
curl: (7) Failed to connect to localhost port 8080: Connection refused
$ systemctl start hello-world
$ systemctl status hello-world
```

```

• hello-world.service - hello world pecan service
  Loaded: loaded (/etc/systemd/system/hello-world.service; disabled; )
  Active: active (running) since Tue 2019-04-23 13:44:20 EDT; 5s ago
  Main PID: 23980 (pecan)
  Tasks: 1 (limit: 4915)
  Memory: 20.1M
  CGroup: /system.slice/hello-world.service
          └─23980 /opt/http/bin/python /opt/http/bin/pecan serve config.py

```

Apr 23 13:44:20 huando systemd[1]: Started hello world pecan service.

Сервис `з` пущен и он ходит в полностью рабочем состоянии. Вновь проверьте на порте `8080`, что фреймворк `з` пущен, `р` бот `ет` и `ре` гирует на `з` просы:

```

$ curl localhost:8080
Hello, World!

```

Если `ост` новить на `ш` сервис с помощью команды `systemctl stop hello-world`, команда `curl` снова нечет сообщать об ошибке соединения.

Поскольку мы создали и установили юнит, проверили, что он работает, `з` пустив и `ост` новив сервис, `т` `к`же проверили, что фреймворк `Ресан` `ре` гирует на `з` - `просы` на его порте по умолчанию. Хотелось бы, чтобы этот сервис `з` пускался и `р` бот `л` при перезагрузке сервера, и в этом нам поможет `р` `з`дел `Install`. Активируем (`enable`) сервис:

```

$ systemctl enable hello-world
Created symlink hello-world.service → /etc/systemd/system/hello-world.service.

```

В случае перезагрузки сервера наш маленький сервис `HTTP API` снова `з`пустится и будет работать.

Управление журналами

Благодаря тому что речь идет о сконфигурированном сервисе с настройками журналирования (весь вывод `stdout` и `stderr` направляется напрямую в `systemd`), управление журналами происходит без всяких усилий с нашей стороны. Не требуется настраивать файлы журналов, их циклическую замену или другие действия. `systemd` предоставляет несколько интересных и очень удобных возможностей для работы с журналами, например ограничение временного промежутка, также фильтрацию по юнитам или идентификаторам процессов.



Для взаимодействия с журналами из юнита служит утилитой команды строки `journalctl`. Этот процесс может использоваться для в случае необходимости, если вы предполагаете наличие дополнительной подкоманды из `systemd`, включающей вспомогательные функции для журналирования.

В предыдущем разделе мы запустили сервис и с помощью утилиты `curl` выполнили несколько запросов к нему, так что можно посмотреть, что показывают журналы:

```
$ journalctl -u hello-world
-- Logs begin at Mon 2019-04-15 09:05:11 EDT, end at Tue 2019-04-23
Apr 23 13:44:20 srv1 systemd[1]: Started hello world pecan service.
Apr 23 13:44:44 srv1 pecan[23980] [INFO] [pecan.commands.serve] GET / 200
Apr 23 13:44:55 srv1 systemd[1]: Stopping hello world pecan service...
Apr 23 13:44:55 srv1 systemd[1]: hello-world.service: Main process exited
Apr 23 13:44:55 srv1 systemd[1]: hello-world.service: Succeeded.
Apr 23 13:44:55 srv1 systemd[1]: Stopped hello world pecan service.
```

Флаг `-u` задает юнит, в данном случае `hello-world`, но можно также использовать шблон или даже указать несколько юнитов.

Зачастую, чтобы следить за записями в журнале, применяют команду `tail`. Ее вызов выглядит примерно так:

```
$ tail -f pecan-access.log
```

Команда, делающая то же самое с помощью `journalctl`, выглядит несколько иначе, но *работает точно так же*:

```
$ journalctl -fu hello-world
Apr 23 13:44:44 srv1 pecan[23980][INFO][pecan.commands.serve] GET / 200
Apr 23 13:44:44 srv1 pecan[23980][INFO][pecan.commands.serve] GET / 200
Apr 23 13:44:44 srv1 pecan[23980][INFO][pecan.commands.serve] GET / 200
```



Если доступен пакет `systemd` с движком `pcr2`, можно воспользоваться подкомандой `--grep` для дельнейшей фильтрации записей журнала в соответствии с шблоном.

Флаг `-f` служит для того, чтобы *следить* за журналом, он начинается с самых недавних записей и продолжает отображать записи по мере их появления — логично `tail -f`. В промышленной эксплуатации количество журналов может быть слишком велико, ошибки могли появиться, например, *сегодня*. В подобных случаях можно применить сочетание флагов `--since` и `--until`. Об флагах примим несколько видов параметров:

- `today;`
- `yesterday;`
- `"3 hours ago";`
- `-1h;`
- `-15min;`
- `-1h35min.`

В нашем ленивом примере команд `journalctl` не может выйти ничего за последние 15 минут. В случае выводимой ею информации указывается промежуток времени, далее отображаются записи, если они найдены:

```
$ journalctl -u hello-world --since "-15min"
-- Logs begin at Mon 2019-04-15 09:05:11 EDT, end at Tue 2019-04-23
-- No entries --
```

Вопросы и упражнения

- Получите записи журналов от `systemd` с помощью `journalctl`, воспользовавшись тремя различными командами.
- Поясните, для чего служит опция `WorkinDirectory` у юнитов `systemd`.
- Почему журналы изменений так важны?
- Для чего предназначен файл `setup.py`?
- Назовите три различия пакетов Debian и RPM.

Задача на ситуационный анализ

Создайте локальный экземпляр PyPI с помощью `devpi`, загрузите в него пакет Python, затем попробуйте установить его из этого локального экземпляра `devpi`.

Непрерывная интеграция и непрерывное развертывание

Автор: Ной

Практики непрерывной интеграции (CI) и непрерывного развертывания (CD) — неотъемлемая составляющая жизненного цикла разработки современного программного обеспечения. Системы CI клонируют базу кода репозитория программного обеспечения из системы контроля исходных кодов, например, GitHub, собирают программное обеспечение в артефакт — бинарный файл, архив tar или образ Docker — и, что чрезвычайно важно, выполняет модульное и/или комплексное тестирование программного обеспечения. Системы CD развертывают собранные системой CI артефакты в целевой среде. Развертывание может производиться автоматически в среде, предназначенной для разработки, но обычно включает шаг совершения ручной одобрения для промышленной эксплуатации. Подобные системы более совершенного типа — платформы непрерывного развертывания — автоматизируют шаг развертывания в промышленную эксплуатацию и способны откатывать развернутую систему в зависимости от показателей, полученных от платформ мониторинга и журналирования.

Ситуационный анализ примера из практики: перевод плохо работавшего сайта с WordPress на Hugo

Некоторое время назад один друг попросил меня об услуге: помочь с исправлением проблем веб-сайта их компании. Эта компания продвигает очень дорогое поддерживаемое и учебное оборудование, и их базовый набор отобран с учетом

под управлением WordPress, постоянно взламываемом, робот вешем уже снова или вообще не робот вешем днями. Обычно я старюсь не впутываться в подобные проекты, но, поскольку речь шла о другом, решил помочь. Нынешний код проекта преобразования этого веб-сайта Hugo вы можете в следующем репозитории Git (<https://oreil.ly/myos1>).

Этот репозиторий GitHub охватывает все шаги процесса преобразования.

1. Создание резервной копии.
2. Преобразование.
3. Обновление.
4. Развертывание.



Концовка этой истории весьма забавна. Созданный мной сверхнадежный сайт с высочайшими производительностью и безопасностью, в том числе развертыванием и невероятной поисковой оптимизацией роботом год ми без всяких уязвимостей или простоев обслуживался. Спустя долгое время, когда я уже забыл об этом проекте, я получил сообщение от этого друга, хотя уже пару лет от него не было ни слуху ни духу. Он сказал, что сайт перестал работать и ему нужна моя помощь.

Я спросил, как это могло случиться, сайт же работал на Amazon S3 с временем доступности 99,999999999 %. Он ответил, что недавно преобразовал его обратно в WordPress, чтобы проще было вносить изменения. Я засмеялся и сказал, что для его проекта не подходит. Как говорится, добрые дела остаются безвозмездными.

Вот некоторые требования, которые я учел в этом проекте.

- Необходимость непрерывного развертывания.
- Быстрота роботов, так же как и ботки!
- Он должен представлять собой статический веб-сайт, размещаемый на облачном хостинге.
- Необходим приемлемый технологический процесс преобразования из WordPress.
- Возможность создания подходящего поискового интерфейса на языке Python.

В конце концов я решил остановиться на Hugo (<https://gohugo.io/>), AWS (<https://aws.amazon.com/>) и Algolia (<https://www.algolia.com/>). Общая архитектура приведена на рис. 6.1.

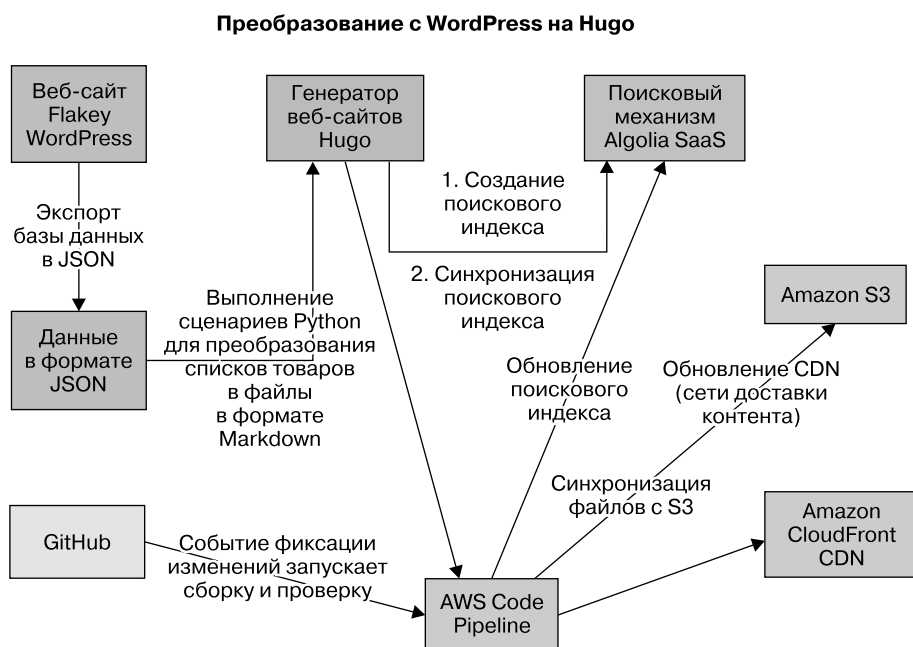


Рис. 6.1. Непрерывное развертывание с помощью Hugo

Настройка Hugo

Начать работу с Hugo очень просто (см. руководство по началу работы с Hugo (https://oreil.ly/r_Rcg)). Во-первых, необходимо его установить. На моей машине под управлением OS X я воспользовался командой

```
brew install hugo
```

Если вы уже установили Hugo, возможно, необходимо его обновить:

```
Error: hugo 0.40.3 is already installed
To upgrade to 0.57.2, run brew upgrade hugo.
```

Если вы работаете на другой платформе, можете использовать инструкции, приведенные тут: <https://oreil.ly/FfWdo>. Чтобы проверить, что все работает, выполните команду `hugo version`:

```
(.python-devops) → ~ hugo version
Hugo Static Site Generator v0.57.2/extended darwin/amd64 BuildDate: unknown
```

Осталось только инициализировать проект с приложения Hugo и уделить тему:

```
hugo new site quickstart
```

Этот командный создатель нового сайта `quickstart`. Можно произвести его повторную сборку, *очень быстро* выполнив команду `hugo`, которая компилирует файлы в формате Markdown в HTML и CSS.

Преобразование WordPress в посты Hugo

Далее я преобразовал базу данных WordPress в формат JSON через неформатированный дампы. Затем написал сценарий Python для преобразования этих данных в посты Hugo в формате Markdown. Вот его код:

```
"""Код преобразования полей старой базы данных в формат Markdown
```

```
Если вы выполнили дампы базы данных WordPress и затем преобразовали его в JSON,
можете приспособить этот код под свои нужды"""
```

```
import os
import shutil
from category import CAT
from new_picture_products import PICTURES

def check_all_category():
    ares = {}
    REC = []
    for pic in PICTURES:
        res = check_category(pic)
        if not res:
            pic["categories"] = "Other"
            REC.append(pic)
            continue

    title, key = res
    if key:
        print("FOUND MATCH: TITLE--[{}], CATEGORY--[{}]" %\
              (title, key))
        ares[title] = key
        pic["categories"] = key
        REC.append(pic)
    return ares, REC

def check_category(rec):

    title = str(rec['title'])
    for key, values in CAT.items():
```



```

    print("KEY: %s, VALUE: %s" % (key, values))
    if title in key:
        return title, key
    for val in values:
        if title in val:
            return title, key

def move_image(val):
    """Создает новую копию загруженных на сайт изображений в каталоге img"""

    source_picture = "static/uploads/%s" % val["picture"]
    destination_dir = "static/img/"
    shutil.copy(source_picture, destination_dir)

def new_image_metadata(vals):
    new_paths = []
    for val in vals:
        pic = val['picture'].split("/)[-1:].pop()
        destination_dir = "static/img/%s" % pic
        val['picture'] = destination_dir
        new_paths.append(val)
    return new_paths

CAT_LOOKUP = {'2100': 'Foo',
              'a': 'Biz',
              'b': 'Bam',
              'c': 'Bar',
              '1': 'Foobar',
              '2': 'bizbar',
              '3': 'bam'}

def write_post(val):

    tags = val["tags"]
    date = val["date"]
    title = val["title"]
    picture = val["picture"]
    categories = val["categories"]
    out = ""

+++
tags = ["%s"]
categories = ["%s"]
date = "%s"
title = "%s"
banner = "%s"
+++
[![%s](%s)](%s)
**Product Name**: %s"" %\
(tags, categories, date, title, picture.lstrip("/"),
 title, picture, picture, title)

```

```

filename = "../content/blog/%s.md" % title
if os.path.exists(filename):
    print("Removing: %s" % filename)
    os.unlink(filename)

with open(filename, 'a') as the_file:
    the_file.write(out)

if __name__ == '__main__':
    from new_pic_category import PRODUCT
    for product in PRODUCT:
        write_post(product)

```

Создание поискового индекса Algolia и его обновление

После преобразования товаров из баз данных в посты в формате Markdown необходимо написать код на языке Python для создания поискового индекса Algolia и его синхронизации. Algolia (<https://www.algolia.com/>) — замечательный инструмент, позволяющий быстро решить проблему поискового механизма, с прекрасной поддержкой Python.

Следующий скрипт синхронизирует все файлы в формате Markdown и генерирует поисковый индекс, который можно загрузить в Algolia:

```

"""
Создает очень простой и очень легко расширяемый JSON-индекс
для Hugo для импорта в Algolia
# Возможно, имеет смысл запустить следующую команду для каталога content,
# чтобы удалить пробелы
for f in *\ *; do mv "$f" "${f// /_}"; done
"""

import os
import json

CONTENT_ROOT = "../content/products"
CONFIG = "../config.toml"
INDEX_PATH = "../index.json"

def get_base_url():
    for line in open(CONFIG):
        if line.startswith("baseurl"):
            url = line.split("=")[-1].strip().strip('""')
            return url

def build_url(base_url, title):

```

```

url = "<a href='%sproducts/%s'>%s</a>" %\
    (base_url.strip(), title.lower(), title)
return url

def clean_title(title):
    title_one = title.replace("_", " ")
    title_two = title_one.replace("-", " ")
    title_three = title_two.capitalize()
    return title_three

def build_index():
    baseurl = get_base_url()
    index = []
    posts = os.listdir(CONTENT_ROOT)
    for line in posts:
        print("FILE NAME: %s" % line)
        record = {}
        title = line.strip(".md")
        record['url'] = build_url(baseurl, title)
        record['title'] = clean_title(title)
        print("INDEX RECORD: %s" % record)
        index.append(record)
    return index

def write_index():
    index = build_index()
    with open(INDEX_PATH, 'w') as outfile:
        json.dump(index, outfile)

if __name__ == '__main__':
    write_index()

```

И наконец, с помощью следующего фрагмента кода можно отправить этот индекс в Algolia:

```

import json
from algoliasearch import algoliasearch

def update_index():
    """Удаляет индекс, затем обновляет его"""
    print("Starting Updating Index")
    client = algoliasearch.Client("YOUR_KEY", "YOUR_VALUE")
    index = client.init_index("your_INDEX")
    print("Clearing index")
    index.clear_index()
    print("Loading index")
    batch = json.load(open('../index.json'))
    index.add_objects(batch)

if __name__ == '__main__':
    update_index()

```

Координация с помощью Makefile

Использование Makefile позволяет повторять шаги развертывания. Обычно я создаю Makefile для локальной координации этого процесса. Вот как выглядит весь ход сборки и развертывания:

```
build:
    rm -rf public
    hugo

watch: clean
    hugo server -w

create-index:
    cd algalia;python make_algalia_index.py;cd ..

update-index:
    cd algalia;python sync_algalia_index.py;cd ..

make-index: create-index update-index

clean:
    -rm -rf public

sync:
    aws s3 --profile <yourawsprofile> sync --acl \
        "public-read" public/ s3://example.com

build-deploy-local: build sync

all: build-deploy-local
```

Развертывание с помощью AWS CodePipeline

Веб-сервисы Amazon (Amazon Web Services, AWS) — что используем я платформу развертывания статических веб-сайтов посредством Amazon S3, Amazon Route 53 и Amazon CloudFront. AWS CodePipeline, сервис сервер сборки, прекрасно подходит для этих сайтов в качестве механизма развертывания. Можно войти в систему AWS CodePipeline, создать новый проект сборки и задать для использования им файл настроек `buildspec.yml`. Код можно не строить под свои нужды, использовать готовые шаблоны и изменять фактическими значениями.

Сразу после получения GitHub события фиксации изменения CodePipeline запускает установку в контейнере. Сначала он берет конкретную версию Hugo, затем выполняет сборку страниц Hugo. Благодаря мощи Go визуализация тысяч страниц Hugo занимает считанные доли секунды.

Наконец, производится синхронизация HTML-страниц в Amazon S3. А поскольку он выполняется внутри AWS и синхронизируется, то тоже происходит чрезвычайно быстро. Последний шаг — удаление объектов из CloudFront:

```
version: 0.1

environment_variables:
  plaintext:
    HUGO_VERSION: "0.42"

phases:
  install:
    commands:
      - cd /tmp
      - wget https://github.com/gohugoio/hugo/releases/\
download/v${HUGO_VERSION}/hugo_${HUGO_VERSION}_Linux-64bit.tar.gz
      - tar -xzf hugo_${HUGO_VERSION}_Linux-64bit.tar.gz
      - mv hugo /usr/bin/hugo
      - cd -
      - rm -rf /tmp/*
  build:
    commands:
      - rm -rf public
      - hugo
  post_build:
    commands:
      - aws s3 sync public/ s3://<yourwebsite>.com/ --region us-west-2 --delete
      - aws s3 cp s3://<yourwebsite>.com/\
s3://<yourwebsite>.com/ --metadata-directive REPLACE \
--cache-control 'max-age=604800' --recursive
      - aws cloudfront create-invalidation --distribution-id=<YOURID> --paths '/*'
      - echo Build completed on `date`
```

Ситуационный анализ примера из практики: развертывание приложения Python App Engine с помощью Google Cloud Build

В 2008 году я написал свою первую статью об использовании Google App Engine. Чтобы посмотреть ее в блоге O'Reilly, вам придется воспользоваться архивом Интернет (<https://oreil.ly/8LoIf>).

Этот пример — ее переосмысление применительно к современности. Это еще одна версия Google App Engine, но не этот раз действующая Google Cloud Build (<https://oreil.ly/MlIhM>). Далее приведен файл конфигурации, несенный

в репозиторий GitHub. Называется он `cloudbuild.yaml`. Весь исходный код данного проекта вы можете посмотреть в этом репозитории Git (<https://oreil.ly/vxsnc>):

```
steps:
- name: python:3.7
  id: INSTALL
  entrypoint: python3
  args:
  - '-m'
  - 'pip'
  - 'install'
  - '-t'
  - '.'
  - '-r'
  - 'requirements.txt'
- name: python:3.7
  entrypoint: ./pylint_runner
  id: LINT
  waitFor:
  - INSTALL
- name: "gcr.io/cloud-builders/gcloud"
  args: ["app", "deploy"]
  timeout: "1600s"
  images: ['gcr.io/$PROJECT_ID/pylint']
```

Отмечу, что файл `cloudbuild.yaml` состоит из цепочки, указанных в файле `requirements.txt`, так же выполняет команду `gcloud app deploy`, развертывающую приложение App Engine при внесении в GitHub:

```
Flask==1.0.2
gunicorn==19.9.0
pylint==2.3.1
```

Вот пошаговое описание настройки этого проекта.

1. Создайте проект.
2. Активируйте оболочку командную оболочку.
3. Сгляните в онлайн-руководство из документации Python 3 App Engine (<https://oreil.ly/zgf5J>).
4. Выполните команду `describe`:

```
verify project is working
```bash
gcloud projects describe $GOOGLE_CLOUD_PROJECT
```
output of command:
```bash
createTime: '2019-05-29T21:21:10.187Z'
```

```
lifecycleState: ACTIVE
name: helloml
projectId: helloml-xxxxx
projectNumber: '881692383648'
```

```

5. Возможно, вы захотите проверить, что работает с нужным проектом. Если нет, можете переключиться на другой с помощью следующей команды:

```
gcloud config set project $GOOGLE_CLOUD_PROJECT
```

6. Создайте приложение App Engine:

```
gcloud app create
```

При выполнении этой команды у вас будет запрошен регион. Выберите `us-central` [12]:

```
Creating App Engine application in project [helloml-xxx]
and region [us-central]....done.
Success! The app is now created.
Please use `gcloud app deploy` to deploy your first app.
```

7. Клонировать репозиторий пример приложения hello world:

```
git clone https://github.com/GoogleCloudPlatform/python-docs-samples
```

8. Перейдите в каталог репозитория с помощью команды `cd`:

```
cd python-docs-samples/appengine/standard_python37/hello_world
```

9. Обновите образ контейнера Cloudshell (отмечу, что это необязательный шаг):

```
git clone https://github.com/noahgift/gcp-hello-ml.git
# Отредактируйте .cloudshellcustomimagerepo.json, указав названия
# проекта и образа
# Совет: включите "Boost Mode" в Cloudshell
cloudshell env build-local
cloudshell env push
cloudshell env update-default-image
# Перезапускаем виртуальную машину Cloudshell
```

10. Создайте виртуальную среду с помощью команды `source`:

```
virtualenv --python $(which python) venv
source venv/bin/activate
```

Проверьте еще раз, что все работает:

```
which python
/home/noah_gift/python-docs-samples/appengine/standard_python37/hello_world/venv/bin/python
```

11. Вызовите редактор оболочки командной оболочки.
12. Установите нужные пакеты:


```
pip install -r requirements.txt
```

 В результате этого должен быть установлен Flask:


```
Flask==1.0.2
```
13. Запустите Flask локально, в командной оболочке GCP:


```
python main.py
```
14. Воспользуйтесь предварительным просмотром (web preview) (рис. 6.2).

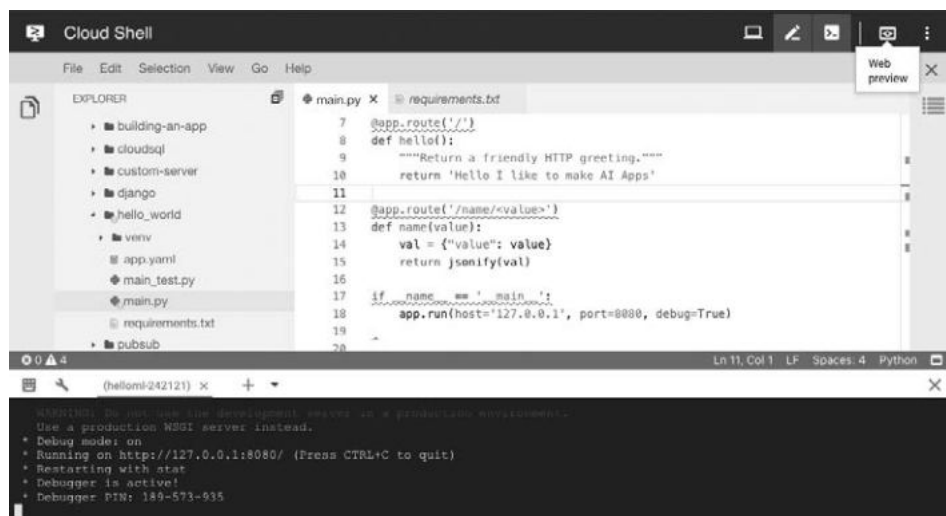


Рис. 6.2. Предварительный просмотр

15. Модифицируйте файл main.py:


```
from flask import Flask
from flask import jsonify

app = Flask(__name__)

@app.route('/')
def hello():
    """Возвращает дружеское HTTP-приветствие."""
    return 'Hello I like to make AI Apps'

@app.route('/name/<value>')
def name(value):
```



```

val = {"value": value}
return jsonify(val)

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=8080, debug=True)

```

16. Перед йте п р метры для проверки р боты этой функции:

```

@app.route('/name/<value>')
def name(value):
    val = {"value": value}
    return jsonify(val)

```

Н пример, обр щение к следующему м ршруту приведет к перед че слов *lion* в функцию `name` в Flask:

`https://8080-dot-3104625-dot-devshell.appspot.com/name/lion`

При этом возвр щ ется зн чение в веб-бр узере:

```

{
  value: "lion"
}

```

17. Р зверните приложение:

```
gcloud app deploy
```

Учтите, что в первый р з р звертыв ние з ймет около десяти минут. Кроме того, возможно, в м придется включить обл чное API сборки.

```

Do you want to continue (Y/n)? y
Beginning deployment of service [default]...

```

```

[===== Uploading 934 files to Google Cloud Storage =====]

```

18. Выполните потоковую перед чу ф йлов журн лов:

```
gcloud app logs tail -s default
```

19. Приложение для промышленной эксплу т ции р звернуто, что должно вы-
глядеть примерно т к:

```

Setting traffic split for service [default]...done.
Deployed service [default] to [https://helloml-xxx.appspot.com]
You can stream logs from the command line by running:
  $ gcloud app logs tail -s default

$ gcloud app browse
(venv) noah_gift@cloudshell:~/python-docs-samples/appengine/\
standard_python37/hello_world (helloml-242121)$ gcloud app
logs tail -s default
Waiting for new log entries...
2019-05-29 22:45:02 default[2019] [2019-05-29 22:45:02 +0000] [8]

```

```

2019-05-29 22:45:02 default[2019] [2019-05-29 22:45:02 +0000] [8]
(8)
2019-05-29 22:45:02 default[2019] [2019-05-29 22:45:02 +0000] [8]
2019-05-29 22:45:02 default[2019] [2019-05-29 22:45:02 +0000] [25]
2019-05-29 22:45:02 default[2019] [2019-05-29 22:45:02 +0000] [27]
2019-05-29 22:45:04 default[2019] "GET /favicon.ico HTTP/1.1" 404
2019-05-29 22:46:25 default[2019] "GET /name/usf HTTP/1.1" 200

```

20. Добыть новый маршрут и проверить его:

```

@app.route('/html')
def html():
    """Возвращает пользовательский HTML"""
    return """
    <title>This is a Hello World World Page</title>
    <p>Hello</p>
    <p><b>World</b></p>
    """

```

21. Установите Pandas и верните результаты в формате JSON. На данном этапе не имеет смысла создавать Makefile и выполнить следующее:

```

touch Makefile
# Следующее необходимо поместить внутрь файла Makefile
install:
    pip install -r requirements.txt

```

Имеет смысл также не строить lint:

```

pylint --disable=R,C main.py
-----
Your code has been rated at 10.00/10

```

Синтаксис веб-маршрутов выглядит так, как показано в следующем блоке кода. Добыть сверху импорт Pandas:

```

import pandas as pd

@app.route('/pandas')
def pandas_sugar():
    df = pd.read_csv(
        "https://raw.githubusercontent.com/noahgift/sugar/\
        master/data/education_sugar_cdc_2003.csv")
    return jsonify(df.to_dict())

```

При обращении к маршруту `https://<вашеприложение>.appspot.com/pandas` вы должны получить что-то наподобие изображенного на рис. 6.3.

22. Добыть следующий маршрут «Википедии»:

```

import wikipedia
@app.route('/wikipedia/<company>')
def wikipedia_route(company):
    result = wikipedia.summary(company, sentences=10)
    return result

```

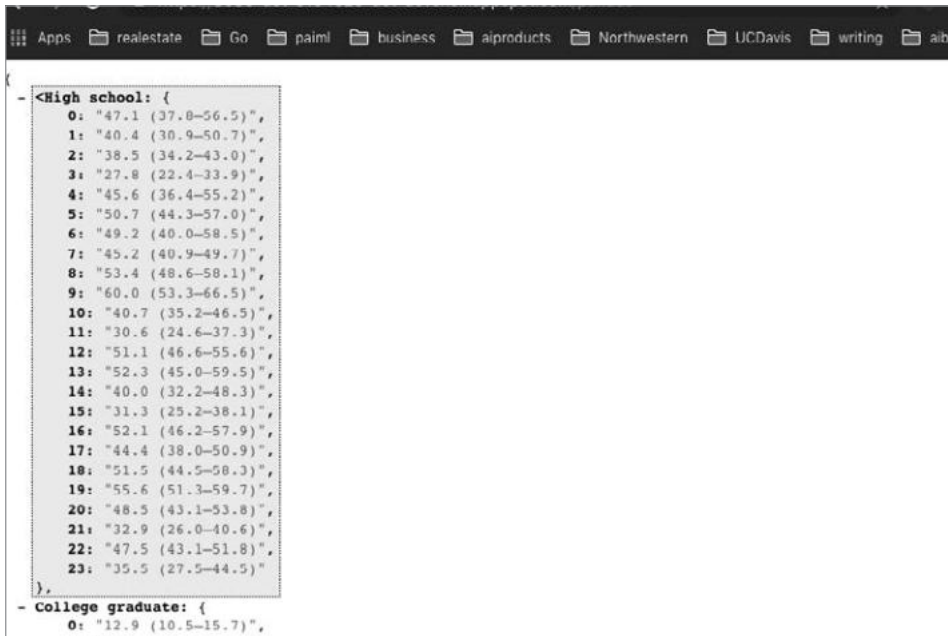


Рис. 6.3. Пример отображения результатов в формате JSON

23. Добавьте в приложение NLP:

- запустите блокнот IPython (<https://oreil.ly/c564z>);
- включите облачное API обработки естественного языка;
- выполните команду `pip install google-cloud-language`:

```

In [1]: from google.cloud import language
...: from google.cloud.language import enums
...:
...: from google.cloud.language import types
In [2]:
In [2]: text = "LeBron James plays for the Cleveland Cavaliers."
...: client = language.LanguageServiceClient()
...: document = types.Document(
...:     content=text,
...:     type=enums.Document.Type.PLAIN_TEXT)
...: entities = client.analyze_entities(document).entities
In [3]: entities

```

24. Вот пример API ИИ целиком:

```

from flask import Flask
from flask import jsonify
import pandas as pd

```

```

import wikipedia

app = Flask(__name__)
@app.route('/')
def hello():
    """Возвращает дружеское HTTP-приветствие."""
    return 'Hello I like to make AI Apps'

@app.route('/name/<value>')
def name(value):
    val = {"value": value}
    return jsonify(val)

@app.route('/html')
def html():
    """Возвращает пользовательский HTML"""
    return """
    <title>This is a Hello World World Page</title>
    <p>Hello</p>
    <p><b>World</b></p>
    """

@app.route('/pandas')
def pandas_sugar():
    df = pd.read_csv(
        "https://raw.githubusercontent.com/noahgift/sugar/\
        master/data/education_sugar_cdc_2003.csv")
    return jsonify(df.to_dict())

@app.route('/wikipedia/<company>')
def wikipedia_route(company):

    # Импортируем клиентскую библиотеку Google Cloud
    from google.cloud import language
    from google.cloud.language import enums
    from google.cloud.language import types
    result = wikipedia.summary(company, sentences=10)

    client = language.LanguageServiceClient()
    document = types.Document(
        content=result,
        type=enums.Document.Type.PLAIN_TEXT)
    entities = client.analyze_entities(document).entities
    return str(entities)

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=8080, debug=True)

```

В этом разделе мы покажем, как создать с нуля приложение App Engine в командной оболочке Google Cloud, и как осуществлять непрерывную поставку с помощью GCP Cloud Build.

Ситуационный анализ примера из практики: NFSOPS

NFSOPS — эксплуатационная методика, использующая точки монтирования NFS (Network File System, сетевая файловая система) для управления кластером компьютеров. Может показаться, что это какая-то новинка, но он применялся с первых дней существования Unix. Я еще в 2000 году использовал точки монтирования NFS на Unix-системах Калифорнийского технологического института для управления программным обеспечением и его сопровождения. Новое — это хорошо забытое старое.

Одной из задач, с которыми я столкнулся в качестве внешнего консультанта в старте, с ним ющемся вопросом виртуальной реальности, было создание инфраструктуры распределения задач для распределения ресурсов и грузки по тысячам спотовых инстансов AWS¹.

Решением, которое в итоге оказалось наиболее рботоспособным, было то, ким: воспользоваться NFSOPS (рис. 6.4) для развертывания кода Python по тысячам спотовых инстансов для машинного зрения за доли секунды.

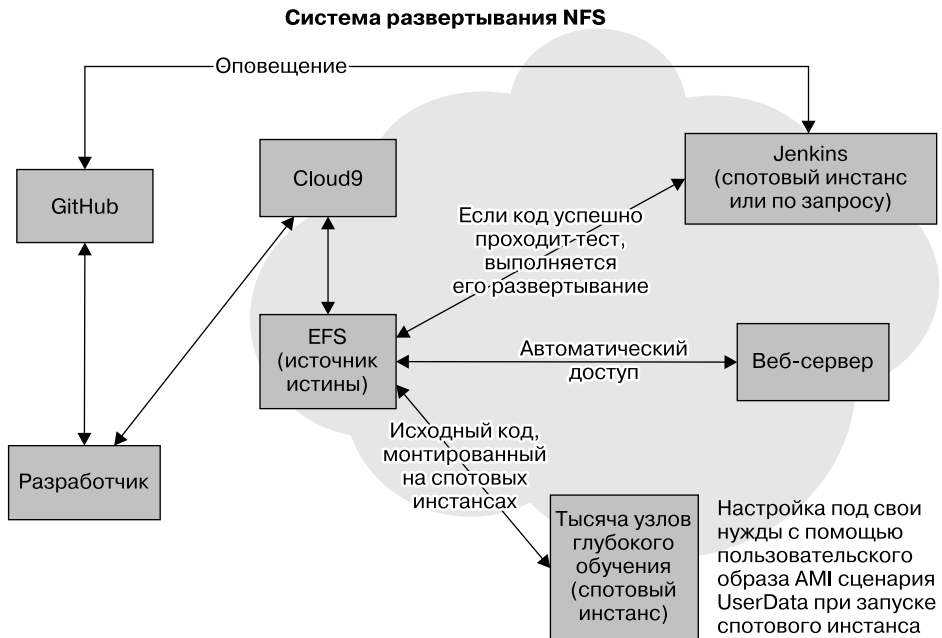


Рис. 6.4. NFSOPS

¹ Это официальная терминология Amazon: <https://aws.amazon.com/ru/ec2/spot/>. — Примеч. пер.

В основе р боты NFSOPS лежит монтирование сервером сборки, в данном случае Jenkins, нескольких точек монтирования (DEV, STAGE, PROD) д птивной файловой системы Amazon (Amazon Elastic File System, EFS). Роль последнего шаг выполнения сборки непрерывной интеграции играет синхронизация (rsync) с соответствующей точкой монтирования:

```
# Этап развертывания при выполнении Jenkins сборки
rsync -az --delete * /dev-efs/code/
```

Развертывание в точку монтирования занимает несколько долей секунды. При запуске тысяч спотовых инстансов они за несколько секунд строятся для монтирования EFS (точек монтирования NFS) и использования исходного кода. Это удобный паттерн развертывания, оптимальный по простоте и скорости. Кроме того, он хорошо сочетается с применением IAC, Amazon Machine Image (AMI) или Ansible.

Мониторинг и журналирование

Уч ствуя в ст рт п х С н-Фр ндско, Ной привык во время обеденного перерыв трениров ться: игр ть в б скетбол, бег ть до б шни Койт или з ним ться бр зильским джиу-джитсу. В большинстве ст рт пов, где р бот л Ной, обед дост вляли прямо н р бочее место.

Возвр щ ясь с обеденного перерыв , он обн ружил очень стр нную з кономерность. Ник ких вредных продуктов никто не ост влял. Среди ост тков было полно с л тов, фруктов, овощей или полезного для здоровья нежирного мяс . Пок он з ним лся спортом, орды голодных р ботников ст рт пов съед ли все вредное для здоровья, лиш я его всякого собл зн съесть что-то подобное. Из этого явно можно сдел ть вывод н тему «не дел ть то же, что все».

Ан логично проще всего игнориров ть к кие-либо опер ции при р зр ботке моделей м шинного обучения, мобильных приложений и веб-приложений. Эт пр ктик н столько р спростр нен , что н помин ет поглощение чипсов, г зировки и мороженого из привезенного в офис обед . Быть к к все не всегда хорошо. В этой гл ве описыв ется подход «едим только с л т и нежирное мясо» к р зр ботке прог рмного обеспечения.

Ключевые понятия создания надежных систем

С высоты н копенного опыт интересно посмотреть, к кие методики р зр ботки ПО док з ли свою эффективность, к кие — нет. Один из лучших нтип ттернов в этой сфере — «доверься мне». Ник кой здр вомыслящий специ лист по DevOps не ст нет верить никому н слово. Людям свойственно ошиб ться, поступ ть в соответствии с эмоциями и ни с того ни с сего ст вить под угрозу целые комп нии. Особенно если они эти комп нии основ ли.

Вместо построения иер рхии, б зирующейс н совершенной бессмыслице, лучше р зр б тив ть н дежные системы пош гово. Кроме того, при созд нии

пл тформы следует быть готовыми к сбоям в любой момент. Единственное, что может повлиять н эту збучную истину, — уч стие в созд нии рхитектуры к кого-либо влиятельного человек . В подобном случ е спр ведливость этой истины возр ст ет многокр тно.

Возможно, вы слыш ли об утилите Chaos Monkey от Netflix, но з чем морочить себе этим голову? Лучше пусть основ тели в шей комп нии, гл вный инженер или вице-президент по техническим вопрос м периодически пишут код и критикуют в шу рхитектуру и б зу код и имитируют т ким обр зом р боту Chaos Monkey. Еще лучше р зрешить им компилиров ть JAR-ф йлы посреди перебоев в обслужив нии промышленной версии и подключ ться к узл м по одному по SSH с крик ми: «Это именно то, что нужно!» Бл год ря этому будет достигнуто г рмоническое среднее с момнения и х ос .

К кой порядок действий следует выбр ть здр вомыслящему специ листу DevOps? Автом тиз ция в жнее иер рхии. Единственное решение проблемы х ос в ст рт п х — сочет ние втом тиз ции, здр вого скептицизм , с мокричности и неизменных принципов DevOps.

Неизменные принципы DevOps

Сложно предст вить себе лучшую отпр вную точку созд ния н дежной системы, чем эти неизменные принципы. Если гл вный инженер собир ет JAR-ф йлы Java н своем ноутбуке для испр вления горящих проблем при промышленной эксплу т ции — увольняйтесь. Подобную комп нию не сп сет ничего. Мы зн - ем — мы с ми были н в шем месте!

Нев жно, н сколько умен/ вторитетен/х ризм тичен/творчески р звит/ бог т человек, но если он вручную вносит критические изменения в в шу прогр ммную пл тформу во время кризис , вы обречены, просто еще об этом не зн ете. Единственн я льтерн тив этому непрерывному кошм ру — втом тиз ция.

Люди не должны уч ствов ть в р звертыв нии прогр ммного обеспечения н постоянной основе. Это гл вный нтип ттерн индустрии р зр ботки прогр ммного обеспечения и, по существу, л зейк для хулиг нов, позволяющ я им вносить х ос в в шу пл тформу. Р звертыв ние, тестиров ние и сборк прогр ммного обеспечения должны быть н 100 % втом тизиров ны.

Гл вное, что вы можете сдел ть для комп нии н н ч льном эт пе своей к рье- ры, — н строить непрерывную интегр цию и непрерывную пост вку. Все ост ль- ное блекнет в ср внении с этим.

Централизованное журналирование

Журналирование по степени важности не слишком отличается от важности. В крупном масштабе распределенных систем журналирование совершенно необходимо. Особое внимание следует уделить журналированию на уровне приложения, так и на уровне среды.

Например, исключения всегда должны отправляться в централизованную систему журналирования. В то же время при просмотре ботке прогресса многообещающего журнала отладки зачастую полезнее, чем вывод в консоль. Почему? Многие часто пытаются просмотреть ботку эвристических правил отладки исходного кода. Почему бы не использовать эти ноды для решения конкретных проблем при промышленной эксплуатации?

Хитрость заключается в уровнях журналирования. Благодаря уровням журналирования, отображаемым только в среде просмотра ботки, не промышленной эксплуатации, исходный код может включать всю логику журналирования. Аналигично можно отключить чрезмерно подробное и вводящее в заблуждение журналирование при промышленной эксплуатации.

Пример журналирования в крупном масштабе распределенных систем можно найти в Ceph (<https://ceph.com/>): у демонов может быть до 20 уровней журналирования! Все они обрываются в коде, что позволяет гибко регулировать его объем. Ceph еще более совершенствует эту стратегию за счет возможности ограничения объема журналирования для демонов. В системе может быть несколько демонов, и журналирование можно расширять для одного из них или их всех.

Ситуационный анализ: база данных при промышленной эксплуатации разрушает жесткие диски

Еще одной важнейшей стратегией журналирования связана с решением проблемы масштабируемости. По достижении приложением определенных размеров уже не имеет смысла хранить все его журналы в файле. Альфредо однажды пришлось искать причины проблемы с основной базой данных крупного веб-приложения, охватывающего сайты около сотни гигабайт, радиостанции и телевизионного канала. Эти сайты генерировали огромный объем трафика и колоссальное количество записей журналов. Журналы были так много, что журналирование PostgreSQL было ускорено на минимальный уровень, так что Альфредо не мог найти причину проблемы, так как не мог повысить уровень журналирования. При повышении уровня журналирования приложение просто перестало бы работать из-за интенсивных операций ввода/вывода. Каждый

день примерно в 05:00 и грузки и бзудных резко возрстл. И ситуация стала новилсь все хуже и хуже.

Администраторы возрзли против того, чтобы повысить уровень журналирования и посмотреть, какие запросы расходуют больше всего ресурсов (PostgreSQL может записывать информацию о запросах в журналы) целый день, так что мы нашли компромисс: повысить уровень журналирования на 15 минут около 05:00. Получив эти журналы, Альфредо сразу же смог найти медленнее всего выполняемые запросы и узнать частоту их выполнения. Среди них немедленно обнаружился лидер: запрос типа `SELECT *` выполнялся так долго, что выходил за рамки 15-минутного окна. Приложение никаких запросов, выбиравших все данные из какой-либо таблицы, не производило — что же это был за запрос?

После долгих уговоров мы получили доступ к серверу бзудных. Раз всплеск и грузки происходит около 05:00 *каждый день*, возможно, он связан с каким-либо регулярно выполняемым сценарием? Мы взглянули в `crontab` (программ, отслеживающих выполняемые в определенные моменты времени процессы) и обнаружили подозрительный сценарий `backup.sh`. В нем было несколько SQL-запросов, включавших `SELECT *`. Администраторы бзудных использовали его для создания резервных копий основной бзудных, и по мере увеличения ее размер рос и инагрузк, пока не вышел за рамки разумного. Решение? Отказаться от применения этого сценария и создать резервные копии одной из четырех вторичных бзудных (реплик).

Это решило проблему резервных копий, но ничем не помогло с невозможностью доступа к журналам. Лучше всего сразу позботиться о распределенном журналировании. Для этого преднзначены такие утилиты, как `rsyslog` (<https://www.rsyslog.com/>), которые, если их использовать с помощью `ch`, могут значительно упростить поиск причин перебоев в обслуживании при промышленной эксплуатации.

Производить или покупать?

Невероятно, насколько больше шумихи поднимется из-за *зависимости пользователей от конкретного производителя*. Впрочем, зависимость пользователей от конкретного производителя — вещь субъективная. Бросьте камень в любую сторону в деловом центре Сан-Франциско — и почти наверняка попадете в кого-нибудь вещающего об ужасах зависимости пользователей от производителей. Впрочем, если копнуть чуть глубже, становится ясно, что альтернативы-то и нет.

В экономике известен принцип сравнительного преимуществ. По существу, он заключается в том, что с экономической точки зрения лучше сосредоточить свое внимание на том, в чем вы сильны, все остальные задачи делегировать дру-

гим людям. В ч стности, в сфере обл чных сервисов н блю д ется постоянный прогресс, идущий н пользу конечным потребителям и не несущий им дополнительных з тр т, — и в большинстве случ ев ре лиз ция проще, чем прежде.

Если в ш комп ния по м шт бун еср вним с крупнейшими технологически-ми комп ниями, в м вряд ли уд стся ре лизов ть, поддержив ть и усовершен-ствов ть ч стное обл ко, чтобы одновременно экономить деньги и двиг ть бизнес вперед. Н пример, в 2017 году Amazon открыл возможность р звертыв ния мультим стерных б з д нных с втом тическим восст новлением после отк з в нескольких *зон х доступности* (Availability Zones). Н пр в х людей, пробо-в вших ее, можем ск з ть, что доб вление в систему вероятности втом тиче-ского восст новления после отк з в подобном сцен рии н ходится букв льно н гр ни возможного, это невероятно сложно. Один из гл вных вопросов, которые следует себе з д ть в пл не утсорсинг : «Профильн я ли это деятельность для в шего бизнес ?» Комп ния, поддержив ющ я собственный сервер электронной почты, притом что ее профильный вид деятельности — прод ж втомобилей, игр ет с огнем и, вероятно, уже сейч с несет убытки.

Отказоустойчивость

Отк зоустойчивость — интереснейш я для обсуждения тем , но иногда весьм неоднозн чн я. Что же т кое отк зоустойчивость и к к ее достичь? Прекр с-ный способ больше узн ть про отк зоустойчивость — изучить офици льные технические опис ния от AWS (<https://oreil.ly/zYuls>). Рекомендуем прочит ть их столько, сколько сможете.

При проектиров нии отк зоустойчивых систем полезно н ч ть с ответ н следу-ющий вопрос: что можно ре лизов ть, чтобы исключить (или хотя бы сокр тить объем) необходимость ручного вмеша тельств в случ е сбоя сервис ? Никому не понр вится оповещение о сбое критически в жной системы, особенно если ее восст новление состоит из множеств эт пов, для того, чтобы убедиться в нор-м льной р боте, необходимо обменив ться информ цией с другими сервис ми. Обр тите вним ние н то, что этот вопрос не говорит о к ком-то м ловероятном событии, в нем явно утвержд ется, что сбой сервис р но или поздно произойдет и для восст новления его р ботоспособности пон добятся определенные усилия.

Одн жды, довольно д вно, пл ниров лось полное перепроектиров ние сложной системы сборки. Он отвеч л з несколько действий, в основном связ нных с п кетной орг низ цией прогр мм и выпуском прогр ммного обеспечения: контролиров л з висимости, генериров л исполняемые ф йлы, п кеты RPM и Debian, созд в емые *make* и прочими утилит ми, кроме того, созд в л и р з-мещ л н сервере репозитории для р зличных дистрибутивов Linux (н пример,

CentOS, Debian и Ubuntu). Основным требованием к системе сборки было ее быстрое действие.

И хотя одной из главных среди поставленных задач была скорость работы, при проектировании системы, включающей несколько различных компонентов, имеет смысл при этом учесть известные проблемные места и попытаться предотвратить появление новых. В больших системах всегда возникают новые проблемы, поэтому жизненно важны правильные стратегии журналирования (и агрегирования журналов), мониторинг и восстановление.

Вернемся к системе сборки. Одной из проблем заключалась в сложности организации микширования для создания репозитория: HTTP API получал пакеты для конкретной версии конкретного проекта, репозитории генерировались втоматически. В этом процессе участвовали базовые, сервис RabbitMQ для синхронной обработки данных и большой объем пространства в хранилище под управлением Nginx для хранения репозитория. Наконец, информация о состоянии отправлялась на центральный инструментальный пульт, чтобы в процессе сборки рабочие боты могли отслеживать свою ветвь проекта. *Совершенно необходимо* было проектировать все с учетом вероятности отката этого сервиса.

На доску приклеили большую записку: «ОШИБКА: откат сервиса репозитория вследствие переполнения диска». Задача заключалась вовсе не в предотвращении переполнения диска, в создании системы, которую мог бы продолжить работать и при переполненном диске, так что после решения проблемы с диском не предстояло бы трудновосподключить его к системе. Ошибка переполненного диска — выдуманный, ее место могло занять любой другой, например не запущенный сервис RabbitMQ или проблема с DNS, но он прекрсно отработает суть поставленной задачи.

Нетрудно просто осознать важность постоянного мониторинга, журналирования и грамотной архитектуры до тех пор, пока часть этого плана не перестанет работать и не окажется невозможно понять, *почему* и *как*. Важно знать, почему произошел сбой, чтобы ретировать шаги по его предотвращению (оповещения, мониторинг и самовосстановление), гарантируя, что эта проблема не возникнет в будущем.

Чтобы система могла продолжить работу в случае отката, мы разобьем нагрузку на пять одинаковых машин, выполняющих одну работу — создание и размещение репозитория. Узлы, создающие бинарные файлы, записывают API исправную машину репозитория, которая, в свою очередь, отправляет HTTP-запрос к конечной точке /health/ следующего сервера сборки в списке. Если сервер отвечает, что исправен, туда отправляются исполняемые файлы, в противном случае API выбирает следующий сервер в списке. Если проверка

состояния узла пройден неудачно три раза подряд, он изымается из оборота. Чтобы вернуть его в оборот, системному администратору достаточно всего лишь перезапустить сервис репозитория. (У сервиса репозитория есть процедура оценки своего состояния, в случае успешного выполнения которой API оповещается о готовности.)

И хотя реализация не железобетонная (все равно требуется определенная работа для запуска сервера, и оповещения не всегда точны), он очень сильно влияет на сопровождение сервиса в случае необходимости восстановления и обеспечения работоспособности всего, даже когда система еще не восстановилась после системного сбоя. Именно в этом и заключается отстойчивость!

Мониторинг

Мониторинг — одна из тех вещей, где можно практически ничего не делать и все равно заявлять, что система мониторинга работает корректно (например, в каталоге Альфредо однажды использовал стандартное средство `curl` для проверки состояния находящегося в промышленной эксплуатации сайта), но он может и выйти за пределы степени зрелости, что средняя промышленная эксплуатация тускнеет по сравнению с ним. Практически все реализованные мониторинги и отправка отчетов об ошибках в общем случае могут помочь найти ответ на самые сложные вопросы жизненного цикла промышленной эксплуатации. Они остро необходимы, но реализовать их практически очень трудно. Именно поэтому существует множество компаний, специализирующихся на мониторинге, оповещении и визуализации метрик.

Практически большинство сервисов относятся к двум парадигмам: извлечение (pull) и помещение (push). В этой главе мы рассмотрим относящийся к первой категории Prometheus и относящиеся ко второй Graphite с StatsD. При добывании возможностей мониторинга в среду очень полезно знать, какой из вариантов лучше подходит для конкретного случая и к каким его подводные камни. А главное, вжно знать об вариантах и иметь возможность развернуть те сервисы, которые лучше подходят для конкретного сценария.

Недежное ПО, работающее в последовательном режиме, должно справляться с чрезвычайно высокими темпами поступления информации о транзакциях, хранить ее, осуществлять привязку к времени, поддерживая выполнение к ней запросов и предоставлять интерактивный графический интерфейс с фильтрами и запросами. По существу, оно должно напоминать высокопроизводительную базу данных, но со специализацией на привязке к времени, операциях над данными и визуализации.

Graphite

Graphite — хранилище числовых хронологических данных: оно умеет хранить значительную числовую информацию с привязкой к времени в соответствии с интервалами привалов. Оно предоставляет *обладатель очень широкими возможностями* API, к которому можно делать запросы относительно данных с учетом промежутков времени, также применять *функции* для преобразования данных или выполнения вычислений над ними.

Важный спектр хранилищ Graphite (<https://oreil.ly/-0YEs>) — оно *не собирает данные*, сосредоточивает внимание на API и возможностях обработки колоссальных объемов данных для значительных промежутков времени. В результате пользователям приходится выбирать, какое программное обеспечение для сбора данных применять вместе с Graphite. Существует немало вариантов программного обеспечения, способного передавать метрики в Graphite, в этой главе мы рассмотрим один из них — StatsD.

Еще один интересный спектр Graphite — хотя в его состав входит веб-приложение, способное рисовать графики по требованию, обычно оно рзвертывается вместе с другим сервисом, которым может использоваться Graphite как прикладную часть для отрисовки графиков. Прекрасный пример этого — великолепный проект Grafana (<https://grafana.com>), предоставляющий полнофункциональное веб-приложение для визуализации метрик.

StatsD

В Graphite можно помещать метрики через TCP или UDP, но StatsD намного удобнее, поскольку существуют варианты создания средств на Python для организации технологических процессов и подобие грегирования метрик по UDP с последующей передачей их в Graphite. Подобная архитектура логична для приложений Python, в которых нежелательно блокирование при отправке данных (TCP-соединения блокируются вплоть до получения ответа, UDP — нет). В ходе работы с ним очень много времени Python-цикл для захвата метрик не имеет смысла расходовать лишнее время на взаимодействие с захватывающим их сервисом.

Если кратко, отправка метрик в сервис StatsD практически не занимает времени (как и должно быть!). Благодаря возможностям Python измерение метрик не составляет сложности. Когда сервис StatsD собирает данные метрик для отправки в Graphite, он запускает процесс отправки. Все это происходит полностью синхронно, благодаря чему приложение может продолжать работу. Метрики, мониторинг и журналирование никоим образом не должны влиять на работу приложения, находящегося в промышленной эксплуатации!

Помещаемые в StatsD данные агрегируются и собираются в гибко настроившуюся прикладную часть (например, Graphite) через заданные промежутки времени (по умолчанию 10 секунд). По нашему опыту развертывания сочетания Graphite и StatsD в нескольких средах промышленной эксплуатации, проще использовать по одному экземпляру StatsD на каждом сервере приложения вместо одного экземпляра для всех приложений. Подобное развертывание упрощает настройку и повышает безопасность: конфигурация всех серверов приложения укажет на сервис StatsD на localhost, так что не требуется открывать никакие внешние порты. В конце StatsD передет метрики в Graphite через исходящее UDP-соединение. Определенное преимущество заключается также в повышении масштабируемости благодаря распределению нагрузки более по конвейеру в Graphite.



StatsD представляет собой демон Node.js, так что его установка означает и подтягивание зависимости Node.js. Это отнюдь не проект Python!

Prometheus

Во многом Prometheus (<https://prometheus.io>) похож на Graphite (широкие возможности выполнения запросов и визуализации). Основное различие — он *извлекает* информацию из источников, причем по HTTP. Так что сервисы должны открывать доступ к конечным точкам HTTP, чтобы Prometheus мог собирать метрики. Еще одно существенное отличие его от Graphite — встроенные возможности оповещения с настроиваемыми правилами генерации оповещений либо возможностью использовать Alertmanager — компонент, предназначенный для управления оповещениями, их подвешивания, агрегирования и переноса в различные системы, например электронную почту, чат-и онлайн-платформы.

В некоторых проектах, например Ceph (<https://ceph.com>), есть настроиваемые опции, активирующие сбор Prometheus информации через заданные промежутки времени. Если есть готовая интеграция подобного вида — замечательно, в противном случае придется запустить где-то экземпляр HTTP-сервера, предоставляющий сервисы данные. Например, в случае применения базы данных PostgreSQL (<https://www.postgresql.org>) экспорт в Prometheus выполняется контейнером, в котором запущен HTTP-сервис, предоставляющий доступ к данным. Такой вариант вполне подходит во многих случаях, но если уже интегрированы какие-либо системы для сбора информации, например collectd (<https://collectd.org>), то запуск HTTP-сервисов может оказаться нежелательным.

Prometheus отлично подходит для краткосрочных и часто меняющихся временных данных, в то время как Graphite лучше подходит для информации, собранной за большой промежуток времени. Оба предоставляют очень развитый язык запросов, но возможности Prometheus шире.

Отличная утилита для Python для отправки метрик в Prometheus — `prometheus_client` (<https://oreil.ly/t9NtW>), если речь идет о веб-приложении, то этот клиент интегрирован с множеством веб-серверов Python, в частности Twisted, WSGI, Flask и даже Gunicorn. Помимо этого, он может экспортировать все данные для вычисления в заданной конечной точке (вместо использования для этого отдельного экземпляра HTTP-сервера). Чтобы в своем веб-приложении выдать их в конечной точке `/metrics/`, добавьте обертку, вызывающий метод `prometheus_client.generate_latest()`, который вернет данные в формате, понятном синтаксису клиенту Prometheus.

Создайте маленькое приложение Flask (сохраните его в виде файла `web.py`), чтобы с ним познакомиться, насколько прост в применении `generate_latest()`, не забывая перед этим установить пакет `prometheus_client`:

```
from flask import Response, Flask
import prometheus_client

app = Flask('prometheus-app')

@app.route('/metrics/')
def metrics():
    return Response(
        prometheus_client.generate_latest(),
        mimetype='text/plain; version=0.0.4; charset=utf-8'
    )
```

Запустите приложение с помощью предзнаменного для запуска веб-сервера Flask:

```
$ FLASK_APP=web.py flask run
* Serving Flask app "web.py"
* Environment: production
  WARNING: This is a development server.
  Use a production WSGI server instead.
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [07/Jul/2019 10:16:20] "GET /metrics HTTP/1.1" 308 -
127.0.0.1 - - [07/Jul/2019 10:16:20] "GET /metrics/ HTTP/1.1" 200 -
```

После запуска робота, откройте веб-браузер и введите URL `http://localhost:5000/metrics`. При этом будет генерироваться информация, подходящая для сбора Prometheus, хотя ничего особо ценного в ней нет:


```
...
# HELP process_cpu_seconds_total Total user and system CPU time in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.27
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 6.0
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1024.0
```

Большинство предпосылок для промышленной эксплуатации веб-серверов, таких как Nginx и Apache, могут генерировать исчерпывающие метрики времени отклика и ожидания. Например, для добавления подобного типа данных в приложение Flask отлично подойдет промежуточное ПО, в котором фиксируются все запросы. Приложения обычно выполняют запросы и другие интересные вещи, так что добавим еще две конечные точки — одну со счетчиком, другую с таймером. Эти конечные точки будут генерировать метрики, в дальнейшем обрабатываемые библиотекой `prometheus_client` и выводимые при запросе к конечной точке `/metrics/` по HTTP.

Добавление счетчика к нашему мейнскому приложению требует нескольких небольших изменений. Создайте новую конечную точку:

```
@app.route('/')
def index():
    return '<h1>Development Prometheus-backed Flask App</h1>'
```

Теперь опишем объект `Counter`. Добавьте название счетчика (`requests`), короткое его описание (`Application Request Count`) и по крайней мере одну удобную метку (например, `endpoint`), которая поможет определить источник этого счетчика:

```
from prometheus_client import Counter

REQUESTS = Counter(
    'requests', 'Application Request Count',
    ['endpoint']
)

@app.route('/')
def index():
    REQUESTS.labels(endpoint='/').inc()
    return '<h1>Development Prometheus-backed Flask App</h1>'
```

После описания счетчика `REQUESTS` включите его в функцию `index()`, перезапустите приложение и выполните несколько запросов. При выполнении запросов

к конечной точке `/metrics/` в выводимой информации будут отражены выполненные нами действия:

```
...
# HELP requests_total Application Request Count
# TYPE requests_total counter
requests_total{endpoint="/"} 3.0
# TYPE requests_created gauge
requests_created{endpoint="/"} 1.562512871203272e+09
```

Теперь давайте объект `Histogram` для захвата более подробной информации о конечной точке, которую иногда отвечает с некоторым задержанием. Код моделирует эту ситуацию посредством приостановки выполнения в случайном выбранном промежутке времени. Как и в функции `index`, необходимо новая конечная точка, в которой бы использовался объект `Histogram`:

```
from prometheus_client import Histogram

TIMER = Histogram(
    'slow', 'Slow Requests',
    ['endpoint']
)
```

В моделируемой нами ресурсоемкой операции действует функция, отслеживающая время ее начала и окончания, затем передающая эту информацию в объект `Histogram`:

```
import time
import random

@app.route('/database/')
def database():
    with TIMER.labels('/database').time():
        # Моделируем время отклика базы данных
        sleep(random.uniform(1, 3))
    return '<h1>Completed expensive database operation</h1>'
```

Нам нужны еще два модуля, `time` и `random`, для вычисления передвешиваемого гистограмму значения времени и моделирования производимой в базе данных ресурсоемкой операции. Запустите приложение еще раз, обратитесь к конечной точке `/database/` — и увидите, как при опросе конечной точки `/metrics/` начнут генерироваться данные. При этом должны появиться несколько записей, соответствующих измерению длительностей выполнения смоделированной операции:

```
# HELP slow Slow Requests
# TYPE slow histogram
slow_bucket{endpoint="/database",le="0.005"} 0.0
slow_bucket{endpoint="/database",le="0.01"} 0.0
```

```
slow_bucket{endpoint="/database",le="0.025"} 0.0
slow_bucket{endpoint="/database",le="0.05"} 0.0
slow_bucket{endpoint="/database",le="0.075"} 0.0
slow_bucket{endpoint="/database",le="0.1"} 0.0
slow_bucket{endpoint="/database",le="0.25"} 0.0
slow_bucket{endpoint="/database",le="0.5"} 0.0
slow_bucket{endpoint="/database",le="0.75"} 0.0
slow_bucket{endpoint="/database",le="1.0"} 0.0
slow_bucket{endpoint="/database",le="2.5"} 2.0
slow_bucket{endpoint="/database",le="5.0"} 2.0
slow_bucket{endpoint="/database",le="7.5"} 2.0
slow_bucket{endpoint="/database",le="10.0"} 2.0
slow_bucket{endpoint="/database",le="+Inf"} 2.0
slow_count{endpoint="/database"} 2.0
slow_sum{endpoint="/database"} 2.0021886825561523
```

Объект `Histogram` отличается большой гибкостью, он может выступать в роли контекстного менеджера, декоратора или получить значения напрямую. Подобная гибкость открывает невероятные возможности и помогает с легкостью создавать инструменты для работы с метриками.

Телеметрия

В одной из известных компаний было приложение, используемое несколькими гигантами, — гигантское монолитное веб-приложение без мониторинга выполнения. Команда эксплуатации неплохо справлялась с отслеживанием расхода системных ресурсов, в частности оперативной памяти и CPU, но не было никакой проверки количества вызовов в секунду API сторонних поставщиков видео, также степени их ресурсоемкости. Можно возразить, что подобные измерения можно получить посредством журналирования, и это действительно так, но опять же речь идет о гигантском монолитном приложении, и без того отличающемся безумным объемом журналов.

Задумка состояла в создании дежных метрик с удобной визуализацией и организацией запросов, причем таких, чтобы их добавление в код требовало не многодневного обучения разработчиков, всего лишь добавления операторов журналирования. Для любой технологии телеметрия во время выполнения должна быть как можно ближе к описанному в предыдущем предложении варианту. Любое решение, которое ему не соответствует, вряд ли будет успешно работять. Если выполнение запросов или визуализация в нем требуют значительных усилий, мало кто им будет пользоваться или обращать на него внимание. Если реализация (и сопровождение!) сложны, от него могут отказаться. Если добавление такой телеметрии времени выполнения неудобно для разработчиков, то не имеет значения, что вся инфраструктура и сервисы готовы использовать метрики — ничего они не получат (по крайней мере, ничего осмысленного).

`python-statsd` — прекр сн я (и крошечн я) библиотек , помещ ющ я метрики в StatsD (д лее их можно перен пр вить в Graphite), с помощью которой можно легко и удобно орг низов ть мониторинг н основе метрик. Удобно выделить в приложении отдельный модуль в к честве д птер этой библиотеки, поскольку в м нужно будет доб вить свои н стройки, дел ть это во множестве мест довольно утомительно.



Существует несколько п кетов Python-клиентов для StatsD в PyPI. Для следующих примеров используйте п кет `python-statsd`. Уст новите его в виртуальной среде с помощью ком нды `pip install python-statsd`. Применение не того клиент может привести к ошибка м импорт !

Один из простейших сцен риев использов ния — счетчик, и примеры для библиотеки `python-statsd` выглядят примерно вот т к:

```
>>> import statsd
>>>
>>> counter = statsd.Counter('app')
>>> counter += 1
```

В этом примере предпол г ется, что StatsD з пущен н लोक льной м шине, зн чит, не нужно созд в ть соединение, н стройки по умолч нию прекр сно подходят. Но при обр щении к кл ссу `Counter` перед ется н зв ние (`app`), что в среде промышленной эксплу т ции не ср бот ет. К кр сск зыв ется в р зделе «Согл шения о н именов ниях» д лее, для р боты приложения жизненно в ж н хорош я схем н именов ния, помог ющ я легко идентифициров ть среду и местоположение д нных метрик, но дел ть это повсюду — избыточно. В некоторых сред х Graphite для всех метрик, отпр вляемых с целью утентифик ции, простр нству имен должно предшествов ть *секретное зн чение*. В результ те возник ет дополнительный слой, который требуется бстр гиров ть, чтобы не нужно было использов ть его при ре лиз ции метрик.

Некоторые ч сти простр нств имен, н пример секретное зн чение, должны гибко н стр ив ться, другие могут присв ив ться прогр ммно. Если у н с есть функция `get_prefix()` для необяз тельного доб вления префикс к простр нству имен, то д птер для `Counter` в отдельном модуле для беспрепятственного вз имодействия можно созд ть следующим обр зом. Для р боты примеров созд йте новый модуль, н зовите его `metrics.py` и доб вьте в него следующее:

```
import statsd
import get_prefix

def Counter(name):
    return statsd.Counter("%s.%s" % (get_prefix(), name))
```

Создать экземпляр `Counter` для небольшого Python-приложения, вызывающего API Amazon S3 с путем `web/api/aws.py`, можно вот так (см. пример из раздела «Соглашения об именах» далее):

```
from metrics import Counter

counter = Counter(__name__)

counter += 1
```

Блгодаря использованию `__name__` объект `Counter` создается с полным пространством имен модуля Python в названии, что на стороне получателя вызов будет выглядеть как `web.api.aws.Counter`. Этот схематический ботет хорошо, но окзывается недостатком гибкой в случае, если нам понадобится несколько счетчиков в циклах, протекающих в различных местах. Необходимо модифицировать дптер, чтобы можно было применить суффикс:

```
import statsd
import get_prefix

def Counter(name, suffix=None):
    if suffix:
        name_parts = name.split('.')
        name_parts.append(suffix)
        name = '.'.join(name_parts)
    return statsd.Counter("%s.%s" % (get_prefix(), name))
```

Если счетчик в файле `aws.py` требуется в двух местах, скжем в функциях чтения и записи для S3, то можно с легкостью добиться в их названии префиксы:

```
from metrics import Counter
import boto

def s3_write(bucket, filename):
    counter = Counter(__name__, 's3.write')
    conn = boto.connect_s3()
    bucket = conn.get_bucket(bucket)
    key = boto.s3.key.Key(bucket, filename)
    with open(filename) as f:
        key.send_file(f)
    counter += 1

def s3_read(bucket, filename):
    counter = Counter(__name__, 's3.read')
    conn = boto.connect_s3()
    bucket = conn.get_bucket(bucket)
    k = Key(bucket)
    k.key = filename
    counter += 1
    return k
```

Теперь в этих двух вспомогательных функциях есть свои уникальные счетчики на основе одного дптета, так что в среде промышленной эксплуатации метрики появятся в пространстве имен, например `secret.app1.web.api.aws.s3.write.Counter`. Подобный уровень детализации удобен при идентификации метрик по операциям. Даже если и существуют сценарии использования, в которых такая детализация не нужна, всегда лучше иметь данные и игнорировать их, чем не иметь. Большинство инструментальных платформ для отображения метрик предоставляют возможности настройки их группировки.

Суффикс полезен при добавлении к названию функций (или методов классов), которые плохо отражают свое назначение, так что усовершенствование схем именования с помощью суффикса — еще одно преимущество достигнутой гибкости:

```
def helper_for_expensive_operations_on_large_files():
    counter = Counter(__name__, suffix='large_file_operations')
    while slow_operation:
        ...
        counter += 1
```



Добавлять счетчики и прочие типы метрик, например датчики, так легко, что возникает соблазн включить их в цикл, но подобная телеметрия может отрицательно повлиять на производительность требующих быстрогодействия блоков кода, выполняемых тысячами раз в секунду. Лучший вариант — ограничить число отправляемых метрик или отправлять их позднее.

В этом разделе мы покажем различные способы получения метрик для локального сервиса StatsD. Этот экземпляр в конечном итоге передается в настроенную прикладную часть, например Graphite, но приведенные упрощенные примеры не относятся исключительно к StatsD. Напротив, они демонстрируют необходимость добавления вспомогательных функций и утилит в качестве дптетов для стандартных задач и то, что при наличии удобных средств регистрации счетчиков их хотят использовать повсеместно. Слишком много данных метрик — это лучше, чем их отсутствие.

Соглашения о наименованиях

В большинстве сервисов мониторинга и сбора метрик, таких как Graphite, Graphana, Prometheus и даже StatsD, есть понятие пространства имен. Пространства имен очень важны, так что желательна тщательная выборка соглашения о наименованиях, которое позволило бы с легкостью идентифицировать компоненты системы и в то же время было достаточно гибким для дальнейшего р

рения или даже изменения. Эти пространства имен — логичные пространства имен Python: имена разделяются точками, и каждая отдельная часть отражает иерархию слева направо. Первый элемент слева — родительский, каждая последующая часть — дочерний элемент.

Например, пусть быстродействующее приложение Python, выводящее изображения на веб-сайте, выполняет обращения к API AWS. Модуль Python, в одном из мест которого мы хотели бы получить метрики, расположен по следующему пути: `web/api/aws.py`. Логично было бы выбрать для этого пути пространство имен `web.api.aws`, но что, если у нас несколько серверов приложений в промышленной эксплуатации? Поскольку метрики передются с одним пространством имен, очень трудно (практически невозможно!) перейти на использование другой схемы именования. Усовершенствуем пространство имен так, чтобы удобнее было идентифицировать серверы в промышленной эксплуатации: `{название_сервера}.web.api.aws`.

Намного лучше! Но заметили ли вы другую проблему? При передаче метрик отправляется значение пространства имен. В примере со счетчиком значение будет выглядеть примерно так: `{название_сервера}.web.api.aws.counter`. Это вызовет проблемы, поскольку наше маленькое приложение выполняет несколько обращений к AWS, например S3, в будущем нам может понадобиться обратиться и к другим сервисам AWS. Исправлять именование дочерних узлов проще, чем родительских, так что в данном случае рассмотрим вариант, чтобы метрики соответствовали измеряемым величинам с максимальной возможной степенью детализации. Например, если у нас есть модуль S3 внутри файла `aws.py`, имеет смысл включить его в значение, чтобы отличить от прочих элементов. Дочерняя часть значения этой метрики будет иметь вид `aws.s3`, метрико-счетчик в итоге будет выглядеть примерно так: `aws.s3.counter`.

Такое количество переменных для пространства имен может показаться неудобным, но большинство известных сервисов метрик позволяют с легкостью комбинировать их, например, так: «Пожалуйста, сообщите мне среднее количество для всех обращений к S3 за последнюю неделю, но только от серверов, находящихся в промышленной эксплуатации на Восточном побережье США». Потрясающие возможности, правда?

Есть и еще одна потенциальная проблема. Как поступить со средними предэксплуатационного тестирования и промышленной эксплуатации? Что, если зрелые и тестирование производятся где-то на виртуальной машине? Часть `{название_сервера}` значения не слишком поможет, если все стандартные значения свои предзнаменения для зрелой машины `srv1`. При разветвлении в различных областях или в случае планов в будущем выполнить масштабирование зрелости отдельной области или строки имеет смысл добиться в пространстве

имен дополнительный элемент. Существует множество вариантов расширения пространства имен так, чтобы оно лучше подходило для конкретной среды, в частности, подойдет префикс: {область}.{экспл|предэкспл_тест|разработка}.{название_сервера}.

Журналирование

Привильная настройка тестирования в Python — немного пугающая задача. Модуль журналирования высокопроизводителен, потреблять выводимую им информацию могут несколько различных модулей. Достаточно собрать в бэбзовых его настройках, чтобы в дальнейшем с легкостью вносить изменения в его конфигурацию. Признаемся, что мы однажды релизовали логгер-тивный вариант журналирования, так как нам было лень строить модуль журналирования должным образом. Это было ошибкой: нам практически никогда не удалось охватить все то, что так хорошо умеет делать стандартный модуль, — многопоточные среды, Unicode, поддержку отличных от STDOUT выходных потоков и многое другое.

Модуль журналирования Python столь велик и существует так много различных вариантов его использования (как и практически всего программного обеспечения, рассматриваемого в этой главе), что даже целой главы недостаточно, чтобы охватить его полностью. В этом разделе приведем краткие примеры лишь простейших сценариев применения, затем постепенно перейдем к более сложным. Разобравшись в нескольких сценариях, вы без проблем сможете расширить журналирование и прочие.

Этот тем сложен, и его освоение придется потратить некоторое время, но помните: журналирование — один из *краеугольных* к *мне* DevOps, без него невозможно быть успешным специалистом в этой сфере.

Почему это так трудно

Приложения Python, как и большинство утилит командной строки и «одно-ровых» утилит, обычно имеют вертикальную архитектуру и отличаются значительной степенью процедурной организации. В начале обучения разработке нечем-то не подобие Python (или, возможно, Bash) имеет смысл прежде всего привыкнуть к подобному процессу разработки. Даже при переходе к более объектно-ориентированному программированию с более широким использованием классов и модулей все равно остается ощущение объявления того, что нужно, создания объектов для их последующего применения и т. д. Нестройки модулей

и объектов обычно не задются при импорте, так что редко можно видеть, чтобы настройки кого-то импортированного модуля задвались глобально для всего проекта еще до создания его экземпляра.

Возникнет ощущение: «Ким-то обрзом и строики модуля уже заданы, но разве это возможно, если я еще даже не вызываю его?» Журналирование в чем-то нелогично: если и строики однажды заданы, во время выполнения модуль ким-то обрзом сохраняет их вне зависимости от того, где он импортируется и используется еще до создания механизмов журналирования. Все это очень удобно, но привыкнуть к этому нелегко, когда практически ни один из составляющих стандартной библиотеки Python так себя не ведет!

basicconfig

Простейший способ выбраться из трясины настроек журналирования — воспользоваться `basicconfig`. Это простой и понятный способ настроить журналирование с множеством значений по умолчанию, требующий всего около трех строк кода:

```
>>> import logging
>>> logging.basicConfig()
>>> logger = logging.getLogger()
>>> logger.critical("this can't be that easy")
CRITICAL:root:this can't be that easy
```

Сообщения выводятся, и модуль, похоже, настроен правильно, причем для этого не нужно ничего знать о журналировании. Удобны широкие возможности и строики под свои нужды и несколько опций, хорошо подходящих для маленьких приложений, в которых не требуются гибкие и строгие интерфейсы журналирования. Форматирование записей журналирования из-за «многословности» также не требует особых усилий:

```
>>> import logging
>>> FORMAT = '%(asctime)s %(name)s %(levelname)s %(message)s'
>>> logging.basicConfig(format=FORMAT, level=logging.INFO)
>>> logger = logging.getLogger()
>>> logger.debug('this will probably not show up')
>>> logger.warning('warning is above info, should appear')
2019-07-08 08:31:08,493 root WARNING warning is above info, should appear
```

В этом примере установлен минимальный уровень журналирования — `INFO`, поэтому отладочное сообщение не выводится. В вызов `basicConfig` передются опции форматирования для времени, название (далее в этом разделе рассуждем об этом подробнее), уровень журналирования и, наконец, само сообщение. Для большинства приложений этого более чем достаточно, и приятно знать, что

с помощью достаточно простого и чистого журналирования можно добиться столь многого.

Проблем с подобными настройками состоит в том, что их может окзаться недостаточно для более сложных сценариев. В подобных настройках очень много изменений по умолчанию, возможно, неприемлемых, менять которые весьма хлопотно. Если есть хотя бы небольшая вероятность, что приложению понадобится нечто более сложное, рекомендуется задать полную конфигурацию журналирования, прозвучавшую (как ни неприятно это) во всех обсуждениях.

Углубляемся в конфигурацию

В модуле журналирования есть несколько *механизмов журналирования* (loggers), которые можно настроить независимо друг от друга, кроме того, они могут наследоваться от *родительского* механизма журналирования. Верхний механизм журналирования в иерархии — корневой (*root*), все остальные — дочерние для него (*root* — родительский). При настройке корневого механизма журналирования, по существу, задается глобальная конфигурация для всех механизмов журналирования. Подобная схема журналирования имеет смысл, если для различных приложений или частей одного приложения необходимы разные интерфейсы журналирования и настройки.

Веб-приложению не удастся отпривлять ошибки сервер WSGI по электронной почте, все остальные записываться в журнал с помощью одного настроенного механизма журналирования уровня *root*. Это логично описанному в разделе «Соглашения о именованиях» (см. ранее) в том, что значения разделены точками, причем ожидается новый уровень иерархии. Это значит, что *app.wsgi* можно настроить отпривку журналов ошибок по электронной почте, *app.requests* — отдельно журналирование в файлы.



Удобный способ работы с этим пространством имен — применить то же пространство имен, что и Python, вместо того чтобы выдумывать что-то свое. Для этого воспользуйтесь именем для создания механизмов журналирования в модулях. Задание одного и того же пространства имен для проекта и журналирования позволяет избежать путаницы.

Настройка журналирования следует издать *как можно раньше*. Если приложение представляет собой утилиту командной строки, лучше всего сделать это в главной точке входа, возможно, даже до синтаксического разбора аргументов. Настройка журналирования для веб-приложений обычно задается с помощью вспомогательных функций фреймворка. У большинства распространенных веб-фреймворков сейчас есть средства настройки журналирования: Django, Flask,

Ресан и Pyramid, все они предоставляют интерфейсы для правильной настройки журналирования. Пользуйтесь!

Следующий пример демонстрирует настройку утилиты командной строки, можно заметить некоторое сходство с `basicConfig`:

```
import logging
import os

BASE_FORMAT = "[% (name)s] [% (levelname)-6s] % (message)s"
FILE_FORMAT = "[% (asctime)s]" + BASE_FORMAT

root_logger = logging.getLogger()
root_logger.setLevel(logging.DEBUG)

try:
    file_logger = logging.FileHandler('application.log')
except (OSError, IOError):
    file_logger = logging.FileHandler('/tmp/application.log')
file_logger.setLevel(logging.INFO)
console_logger.setFormatter(logging.Formatter(BASE_FORMAT))
root_logger.addHandler(file_logger)
```

В этом примере происходит много интересного. Производится запрос к корневному механизму журналирования путем вызова `getLogger()` без каких-либо аргументов, причем уровень журналирования установлен в `DEBUG`. Для нас это неплохо, ведь другие дочерние механизмы журналирования могут менять уровень. Далее мы задаем настройку механизма журналирования в файл. В данном случае он попытается открыть файл журнала, если нет возможности записать в него, то использует временный файл в каталоге `tmp`. Далее его уровень журналирования установлен в `INFO`, формат сообщений меняется и теперь включаются метки даты/времени (удобно для файлов журналов).

Обратите внимание на то, что в конце к `root_logger` добавляется механизм журналирования в файл. Это делается странным, но в данном случае делается так из-за конфигурации корневого механизма журналирования, чтобы он отвечал за все. Благодаря добавлению к корневному механизму журналирования *потокowego оброботчик* (stream handler) приложение будет отправлять журналы в файл и в стандартный поток ошибок одновременно:

```
console_logger = logging.StreamHandler()
console_logger.setFormatter(BASE_FORMAT)
console_logger.setLevel(logging.WARNING)
root_logger.addHandler(console_logger)
```

В данном случае мы воспользовались форматом `BASE_FORMAT`, поскольку информация будет выводиться в терминал, где метки даты/времени будут только

мешать. Как видите, набор строк требуется довольно много и они сильно усложняются, если механизмов журналирования несколько. Для минимизации этого эффект лучше создать отдельный модуль со вспомогательной функцией, которая будет задавать все эти опции. В качестве альтернативы подобному варианту конфигурации модуль `logging` предлагает конфигурацию на основе ассоциативного массива, в которой параметры конфигурации задаются в интерфейсе вида «ключ/значение». Далее приведен подобная конфигурация для того же примера.

Чтобы увидеть все это в действии, добавьте в конец файла несколько операций записи в журнал, выполняемых непосредственно с помощью Python, и сохраните его в файле `log_test.py`:

```
# Корневой механизм журналирования
logger = logging.getLogger()
logger.warning('this is an info message from the root logger')

app_logger = logging.getLogger('my-app')
app_logger.warning('an info message from my-app')
```

Корневой механизм журналирования является родительским, кроме того, здесь появляется новый механизм журналирования `my-app`. Если выполнить этот файл напрямую, в терминале, как и в файле `application.log`, будет выведено следующее:

```
$ python log_test.py
[root][WARNING] this is an info message from the root logger
[my-app][WARNING] an info message from my-app
$ cat application.log
[2019-09-08 12:28:25,190][root][WARNING] this is an info message from the root
logger
[2019-09-08 12:28:25,190][my-app][WARNING] an info message from my-app
```

Выводимая информация дублируется, поскольку мы строили оба механизма журналирования, но это вовсе не обязательно. Форматирование в файловом механизме журналирования отличается, чтобы было удобнее просматривать в консоли:

```
from logging.config import dictConfig

dictConfig({
    'version': 1,
    'formatters': {
        'BASE_FORMAT': {
            'format': '[%(name)s][%(levelname)-6s] %(message)s',
        },
        'FILE_FORMAT': {
            'format': '[%(asctime)s] [%(name)s][%(levelname)-6s] %(message)s',
        },
    },
})
```

```

    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'INFO',
            'formatter': 'BASE_FORMAT'
        },
        'file': {
            'class': 'logging.FileHandler',
            'level': 'DEBUG',
            'formatter': 'FILE_FORMAT'
        }
    },
    'root': {
        'level': 'INFO',
        'handlers': ['console', 'file']
    }
}
})

```

`dictConfig` тут помог ет н гляднее отобр зить, куд что поп д ет и к к все свя-з но друг с другом, по ср внению с приведенным р нее примером. Для сложных рхитектур, где нужны несколько мех низмов журн лиров ния, предпочти-тельнее в ри нт с `dictConfig`. В большинстве веб-фреймворков используется исключительно конфигу р ция н основе ссоци тивных м ссивов.

Иногда форм т журн лиров ния игнорируется. Его ч сто р ссм трив ют к к некую обертку, предн зн ченную для удобств чит ющих журн л. Хотя в этом есть доля истины, но кв др тные скобки с обозн чением уровня журн лиров ния (н пример, `[CRITICAL]`) очень удобны, особенно когд нужно отделять среды друг от друг в соответствии с другими п р метр ми, н пример среды про-мышленной эксплу т ции, предэксплу т ционного тестиров ния и р зр ботки. Р зр ботчику может быть и т к ср зу ясно, что эти журн лы — из версии для р зр ботки, но чрезвычайн о в жно поним ть, были ли они перед ны откуд -то или собир лись центр лизов нно. Вот к к эти возможности применяются ди-н мически в `dictConfig` с помощью переменных среды и `logging.Filter`:

```

import os
from logging.config import dictConfig

import logging

class EnvironFilter(logging.Filter):
    def filter(self, record):
        record.app_environment = os.environ.get('APP_ENVIRON', 'DEVEL')
        return True

dictConfig({
    'version': 1,
    'filters': {

```

```

        'environ_filter': {
            '(): EnvironFilter
        }
    },
    'formatters': {
        'BASE_FORMAT': {
            'format':
                '[%(app_environment)s][%(name)s][%(levelname)-6s] %(message)s',
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'INFO',
            'formatter': 'BASE_FORMAT',
            'filters': ['environ_filter'],
        }
    },
    'root': {
        'level': 'INFO',
        'handlers': ['console']
    }
}
))

```

В этом примере есть много нюансов и можно легко упустить несколько модифицированных мест. Во-первых, был добавлен новый класс `EnvironFilter`, базовым для которого служит `logging.Filter`. В нем объявлен метод `filter`, принимающий в качестве аргумента `record`. Именно тот метод описания этого метода требует базовый класс. Аргумент `record` далее расширяется, включая переменную среды `APP_ENVIRON`, по умолчанию равную `DEVEL`.

Во-вторых, в `dictConfig` добавляется новый ключ (`filters`), в котором фильтры называются `environ_filter`, указывающий на класс `EnvironFilter`. И наконец, в ключе `handlers` мы добавляем ключ `filters`, принимающий список, в данном случае содержащий только один фильтр — `environ_filter`.

Описание и наименование фильтров выглядят неуклюже, но лишь потому, что наш пример тривиален. В более сложных случаях было бы гораздо труднее и расширение не требуют заполнения ассоциативного массива *шablонным кодом*, упрощая, тем самым, дальнейшую модификацию и расширение.

С помощью короткой проверки в командной строке можно посмотреть, как новый фильтр работает в среде. В этом примере используется простое приложение Pecan (<https://www.pecanpy.org>):

```

$ pecan serve config.py
Starting server in PID 25585
serving on 0.0.0.0:8080, view at http://127.0.0.1:8080
2019-08-12 07:57:28,157 [DEVEL][INFO    ] [pecan.commands.serve] GET / 200

```

Сред `DEVEL` прекрсно работает, и для смены ее среды промышленной эксплуатации достаточно всего одной переменной среды:

```
$ APP_ENVIRON='PRODUCTION' pecan serve config.py
Starting server in PID 2832
serving on 0.0.0.0:8080, view at http://127.0.0.1:8080
2019-08-12 08:15:46,552 [PRODUCTION][INFO    ] [pecan.commands.serve] GET / 200
```

Распространенные паттерны

Модуль журналирования включает несколько неплохих паттернов, и первый взгляд неочевидных, но заслуживающих к нему более чем одного применения. Один из них — использование вспомогательной функции `logging.exception`. Обычная схема выглядит вот так:

```
try:
    return expensive_operation()
except TypeError as error:
    logging.error("Running expensive_operation caused error: %s" % str(error))
```

Он неудобен по нескольким причинам: сначала «съедает» исключение и лишь потом отображает его строковое представление. Если исключение неочевидное либо генерируется в неочевидном месте, то оповестить о `TypeError` бессмысленно. Если изменить исключение на строковое значение из вершины неудачей, вы получите ошибку `ValueError`, но она мало чем поможет, если код скрывает трассу вызовов:

```
[ERROR] Running expensive_operation caused an error:
TypeError: not all arguments converted during string formatting
```

Где возникла эта ошибка? Мы знаем, что она произошла при вызове `expensive_operation()`, но где именно? В каких функции, классе или файле? Подобное журналирование не помогает, только приводит разработчика в бешенство! С помощью модуля журналирования можно занести в журнал полную трассу исключения:

```
try:
    return expensive_operation()
except TypeError:
    logging.exception("Running expensive_operation caused error")
```

Вспомогательная функция `logging.exception` как по волшебству помещает в журнал полную трассу исключения. В реализации не нужно бороться с перехватами ошибки, как раньше, или даже пытаться извлечь из исключения полезную информацию. Модуль журналирования позаботится обо всем сам.

Еще один удобный паттерн — использование встроенных возможностей модуля журналирования для интерполяции строк. Возьмем для примера следующий фрагмент кода:

```
>>> logging.error(
    "An error was produced when calling: expensive_operation, \
    with arguments: %s, %s" % (arguments))
```

В этом операторе есть две строковые переменные, он предполагает, что `arguments` состоит минимум из двух элементов. Если в `arguments` нет двух элементов, приведенный оператор приведет к сбою неходящегося в промышленной эксплуатации кода. А его сбой из-за журналирования *всегда* нежелателен. В модуле есть вспомогательная функция, позволяющая перехватить эту ошибку, сообщить о проблеме и продолжить работу программы:

```
>>> logging.error("An error was produced when calling: expensive_operation, \
    with arguments: %s, %s", arguments)
```

Этот рекомендуемый способ передачи аргументов в оператор совершенно безопасен.

Стек ELK

Подобно тому как Linux, Apache, MySQL и PHP известны под общей аббревиатурой *LAMP*, нередко можно услышать про стек *ELK*: Elasticsearch, Logstash и Kibana. Он дает возможность извлекать информацию из журналов, захватывать полезные метаданные и отправлять их в хранилище документов (Elasticsearch), а затем отображать информацию на мощной инструментальной панели (Kibana). Хорошо понимая все составляющие этого стека жизненно важно для эффективной стратегии потребления журналов. Все эти составляющие одинаково важны, и хотя для каждой из них существуют логи, в этом разделе мы рассмотрим роли, которые именно они играют в примере приложения.

Большинство неходящихся в промышленной эксплуатации систем существуют довольно давно, и у разработчика редко выпадет шанс переделать инфраструктуру с нуля. Но даже если в итоге получилось проектировать инфраструктуру с чистого листа, очень легко упустить из виду важность структуры журналов. Правильная структура журналов столь же важна, как и сэмплирование полезной информации, но в случае отсутствия инфраструктуры помощь может прийти Logstash. Выводимая по умолчанию в журнал информация при установке Nginx выглядит примерно так:

```
192.168.111.1 - - [03/Aug/2019:07:28:41 +0000] "GET / HTTP/1.1" 200 3700 "-" \
    "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0"
```


Некоторые чистоты этого оператора записи в журнал вполне очевидны, например HTTP-метод (GET) и метка даты/времени. Если у вас есть возможность управлять отображением информации, можете отбросить несущественную или включить какие-то нужные данные, главное — четко понимать смысл каждого из компонентов. Вот строки по умолчанию в конфигурации HTTP-сервера в `/etc/nginx/nginx.conf`:

```
http {
    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                   '$status $body_bytes_sent "$http_referer" '
                   '"$http_user_agent" "$http_x_forwarded_for"';
    ...
}
```

При первом взгляде на выводимую информацию может сложиться впечатление, что тире символизируют отсутствующую информацию, но это не совсем так. В нашем примере вывод записей журнала за IP-адресом следуют *два* тире: одно просто в «косметических» целях, второе обозначает отсутствие информации. Из конфигурации видно, что за IP-адресом следует одиночное тире, за тем `$remote_user`, что удобно для аутентификации (захват аутентифицированного пользователя). Если же речь идет о HTTP-сервере, на котором не включен аутентификация, `$remote_user` можно из конфигурации убрать (при наличии прав доступа к изменению файла `nginx.conf`) или проигнорировать с помощью привилегии извлечения метаданных из журналов. В следующем разделе взглянем, чем в этой ситуации может помочь Logstash с его множеством плагинов обработки входных данных.



Elasticsearch, Logstash и Kibana обычно недоступны в дистрибутивах Linux. В зависимости от вида дистрибутива необходимо импортировать соответствующие ключи подписи, кроме того, системное управление пакетом должно быть настроено на работу с нужными репозиториями. Загляните в посвященные установке разделы официальной документации (<https://oreil.ly/A-EwN>). Убедитесь также, что установлен пакет Filebeat — облегченный, но обладающий большими возможностями утилит для переноса журналов. Мы воспользуемся ею в дальнейшем для отправки журналов в Logstash.

Logstash

Первый шаг после того, как выбран стек ELK, — применить привилегии Logstash для извлечения информации из нужного источника, ее фильтрации, затем передчи в нужный сервис — в данном случае Elasticsearch. После установки Logstash в файловой системе появляется путь `/etc/logstash/` с полезным каталогом `conf.d` внутри, в который можно добавлять файлы конфигурации для различных сервисов. В нашем случае речь идет о захвате информации Nginx, ее

фильтрции, затем перед челоком льяному сервису Elasticsearch, который должен быть уже установлен и запущен.

Для потребления журналов необходимо установить утилиту `filebeat`. Он имеется в тех же репозиториях, которые вы использовали для установки Elasticsearch, Kibana и Logstash. Прежде чем настроить Logstash, необходимо убедиться, что настройки Filebeat соответствуют файлам журналов Nginx и местоположению Logstash.

После установки Filebeat добавьте пути журналов Nginx и порт Logstash по умолчанию для localhost (5044). В конфигурацию файла `/etc/filebeat/filebeat.yml` необходимо включить (или раскомментировать) следующее:

```
filebeat.inputs:

- type: log
  enabled: true

  paths:
    - /var/log/nginx/*.log

output.logstash:
  hosts: ["localhost:5044"]
```

Благодаря этому Filebeat просмотрит все до единого пути из `/var/log/nginx/`, после чего перенесет их в экземпляр localhost Logstash. Если для другого приложения Nginx требуется отдельный файл журналов, он тоже добавляется тут. В файле конфигурации могут присутствовать и другие значения по умолчанию, их можно оставить без изменений. Запустите сервис:

```
$ systemctl start filebeat
```

А теперь создайте новый файл с именем `nginx.conf` в каталоге конфигурации Logstash (`/etc/logstash/conf.d`). Первый раздел, который нужно добавить, относится к обработке входных данных:

```
input {
  beats {
    port => "5044"
  }
}
```

Раздел `input` указывает, что поток информации поступит из сервиса Filebeat по порту 5044. А поскольку все пути к файлам указываются в конфигурации Filebeat, здесь больше ничего и не требуется.

Далее необходимо извлечь информацию и связать ее с ключами (или полями). Чтобы разобраться в этих неструктурированных данных, понадобятся привил

синтаксического разбор. Для того разбор мы воспользуемся плагин `grok`. Добавьте в конец того же файла следующую конфигурацию:

```
filter {
  grok {
    match => { "message" => "%{COMBINEDAPACHELOG}" }
  }
}
```

В разделе `filter` теперь описано, как должен использоваться плагин `grok`, который получает входную строку и применяет объединяющий большими возможностями набор регулярных выражений `COMBINEDAPACHELOG`, с помощью которых можно найти и привязать все компоненты журналов веб-сервера, поступающих из Nginx.

Наконец, в разделе `output` необходимо указать, куда должны отправляться теперь уже структурированные данные:

```
output {
  elasticsearch {
    hosts => ["localhost:9200"]
  }
}
```

Это значит, что все структурированные данные пересылаются в локальный экземпляр Elasticsearch. Как видите, для этого достаточно минимальной конфигурации Logstash (и сервис `Filebeat`). Для дальнейшей более точной настройки сбора и синтаксического разбора журналов можно добавить еще несколько плагинов и опций конфигурации. Подобный подход с готовыми инструментами идеален для того, чтобы сразу приступить к работе, не задумываясь о том, какие нужны расширения или плагины. Если интересно, можете взглянуть в исходный код Logstash и найти содержащий `COMBINEDAPACHELOG` файл `grok-patterns` — этот набор регулярных выражений поистине впечатляет.

Elasticsearch и Kibana

Чтобы подготовить и запустить систему для получения структурированных данных из Logstash на локальной машине, практически ничего, кроме установки пакета `elasticsearch`, не требуется. Убедитесь, что сервис запущен и работает без проблем:

```
$ systemctl start elasticsearch
```

Аналогичным образом установите пакет `kibana` и запустите соответствующий сервис:

```
$ systemctl start kibana
```

И хотя Kibana — это инструмент на Python, эти сервисы настолько хорошо интегрированы, что демонстрируют, что такое по-настоящему хорошее архитектурное решение. После первого запуска Kibana и чистки из пушенного в машине экземпляра Elasticsearch, просмотрив вывод из логов журналов. Такое поведение по умолчанию Elasticsearch без каких-либо дополнительных настроек. Все прозрачно, из сообщений понятно, что плагин был инициализирован и смог получить доступ к Elasticsearch:

```
{ "type": "log", "@timestamp": "2019-08-09T12:34:43Z",
  "tags": [ "status", "plugin:elasticsearch@7.3.0", "info" ], "pid": 7885,
  "state": "yellow",
  "message": "Status changed from uninitialized to yellow",
  "prevState": "uninitialized", "prevMsg": "uninitialized" }

{ "type": "log", "@timestamp": "2019-08-09T12:34:45Z",
  "tags": [ "status", "plugin:elasticsearch@7.3.0", "info" ], "pid": 7885,
  "state": "green", "message": "Status changed from yellow to green - Ready",
  "prevState": "yellow", "prevMsg": "Waiting for Elasticsearch" }
```

Если поменять настройки, указать в неправильный порт, из журналов становится совершенно ясно, что ничего с собой не берет:

```
{ "type": "log", "@timestamp": "2019-08-09T12:59:27Z",
  "tags": [ "error", "elasticsearch", "data" ], "pid": 8022,
  "message": "Request error, retrying
    GET http://localhost:9199/_xpack => connect ECONNREFUSED 127.0.0.1:9199" }

{ "type": "log", "@timestamp": "2019-08-09T12:59:27Z",
  "tags": [ "warning", "elasticsearch", "data" ], "pid": 8022,
  "message": "Unable to revive connection: http://localhost:9199/" }
```

После запуска Kibana вместе с Elasticsearch (на правильном порте!), Filebeat и Logstash вы увидите полнофункциональную инструментальную и множество возможных опций (рис. 7.1).

Выполните запрос к локальному экземпляру Nginx, чтобы в журнале появились новые записи, и запустите оброботку данных. В этом примере мы используем утилиту Apache Benchmarking (ab), но вы можете использовать просто браузер или сделать запрос напрямую с помощью curl:

```
$ ab -c 8 -n 50 http://localhost/
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
```

```
Benchmarking localhost (be patient).....done
```

Не останавливая далее Kibana, перейдите на URL/порт по умолчанию, на котором он работает, — <http://localhost:5601/>. Представление по умолчанию предлагает множество дополнительных опций. В разделе *discover* вы увидите

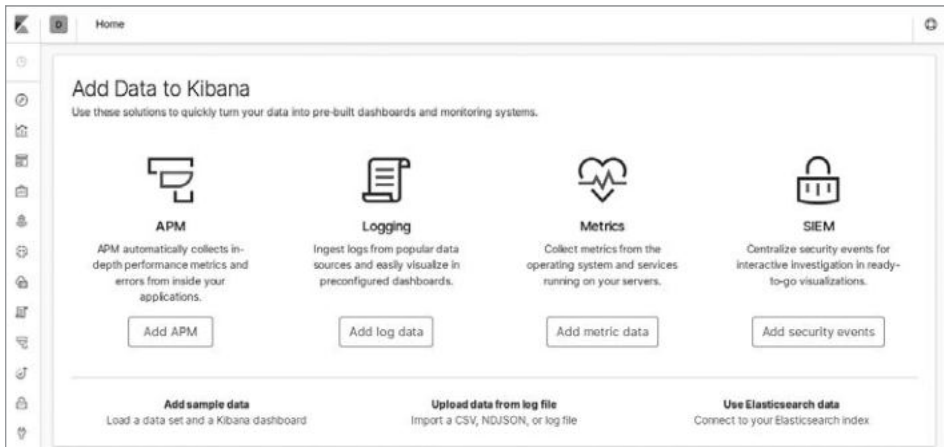


Рис. 7.1. Стартовая страница инструментальной панели Kibana

всю структурированную информацию по запросам. Вот пример доступного в Kibana (получающего данные из Elasticsearch) фрагмент в формате JSON, обработанного Logstash:

```
...
  "input": {
    "type": "log"
  },
  "auth": "-",
  "ident": "-",
  "request": "/",
  "response": "200",
  "@timestamp": "2019-08-08T21:03:46.513Z",
  "verb": "GET",
  "@version": "1",
  "referrer": "\"-\"",
  "httpversion": "1.1",
  "message": "::1 - - [08/Aug/2019:21:03:45 +0000] \"GET / HTTP/1.1\" 200",
  "clientip": "::1",
  "geoip": {},
  "ecs": {
    "version": "1.0.1"
  },
  "host": {
    "os": {
      "codename": "Core",
      "name": "CentOS Linux",
      "version": "7 (Core)",
      "platform": "centos",
      "kernel": "3.10.0-957.1.3.el7.x86_64",
      "family": "redhat"
    }
  },
}
```

```

    "id": "0a75ccb95b4644df88f159c41fdc7cfa",
    "hostname": "node2",
    "name": "node2",
    "architecture": "x86_64",
    "containerized": false
  },
  "bytes": "3700"
},
"fields": {
  "@timestamp": [
    "2019-08-08T21:03:46.513Z"
  ]
}
}
...

```

Logstash захватил в жёсткие ключи (`verb`, `timestamp`, `request` и `response`) и произвел их синтаксический разбор. Чтобы превратить эту простую архитектуру в нечто полезное на практике, придется проделать еще немало работы. Захваченные метаданные можно использовать для визуализации трафика (включая данные геолокации), причем Kibana позволяет даже задать для данных пороговые значения для оповещения в случаях, когда конкретные метрики превысят определенное значение или оканчиваются ниже его.

Инструментальной помощи все эти структурированные данные можно разобрать по составляющим и создать на их основе содержательные графики и представления (рис. 7.2).



Рис. 7.2. Структурированные данные в Kibana

Как мы видели, при использовании стека ELK не требуется практически никаких усилий и нужны лишь минимальные настройки, чтобы приступить к захвату и синтаксическому сбору журналов. Приведенные примеры тривиальны, но демонстрируют колоссальные возможности его (стека) компонентов. Нам приходилось чаще, чем хотелось бы, сталкиваясь с инфраструктурой, в которой задние cron производило журналные операции `tail` и `grep` для поиска каких-либо шаблонов, отправки сообщения электронной почты или отсылки уведомления в Nagios. Обладающие большими возможностями программные компоненты (если вы хорошо осознали их потенциал даже в простейшем варианте) жизненно важны для усовершенствования инфраструктуры, в данном случае — для лучшего понимания того, что она делает.

Вопросы и упражнения

- Что такое отказоустойчивость и чем она полезна для инфраструктурных систем?
- Что делать с системами, генерирующими огромные объемы журналов?
- Поясните, почему при помещении метрик в другие системы предпочтительнее UDP. В чем заключается проблема с TCP?
- Опишите различия между системами, ориентированными на извлечение (*pull*) и помещение (*push*) данных. В каких случаях предпочтительнее первые, а в каких — вторые?
- Предложите соглашение об именовании для хранения метрик, подходящее для среды промышленной эксплуатации, веб-серверов и серверов баз данных и различных типов приложений.

Задача на ситуационный анализ

Создайте приложение Flask, полностью реализующее журналирование на различных уровнях (`info`, `debug`, `warning` и `error`) и при возникновении исключения отправляющее метрику (например, счетчик) в удаленный экземпляр Graphite через сервис StatsD.

ГЛАВА 8

Pytest для DevOps

Непрерывная интеграция, непрерывная поставка, развертывания и любые конвейерные технологические процессы в целом даже при минимальной степени продуманности будут неполнены различными проверками. Они могут встречаться на любых этапах, также при достижении системой каких-либо важных целей.

Например, если посередине длинного списка этап развертывания вызывается командой `curl` для получения очень важного файла, как думать, нужно продолжить сборку в случае неудачного ее выполнения? Нет, верно, нет! У `curl` есть флаг (`--fail`) для возврата ненулевого кода завершения в случае ошибки HTTP. Этот простой флаг может послужить своего рода проверкой, чтобы убедиться в успешности запроса, в противном случае перевести сборку в состояние завершившейся неудачно. Ключевое здесь — *убедиться* в успешном выполнении чего-либо, именно стратегии проверки и тестирования, позволяющие создавать лучшую инфраструктуру, являются фундаментом этой главы.

А если задействовать Python с его фреймворками тестирования и подобие `pytest` для верификации систем, идея проверок не кажется столь рзоботничкой и много больше.

В этой главе мы обсудим основы тестирования на Python с помощью феноменального фреймворка `pytest`, углубимся в некоторые продвинутые его возможности и, наконец, подробно расскажем о проекте *Testinfra* — платформе к `pytest` для верификации систем.

Сверхспособности тестирования фреймворка `pytest`

У нас не хватит слов, чтобы воспеть хвалу фреймворку `pytest` тем, как он того заслуживает. Создан он был Хольгером Крекелем (Holger Krekel), сейчас с его поддержкой отвечает совсем небольшая группа людей, которым великолепно удалось добиться высокого качества этого программного продукта, используя

емого нами ежедневно. Довольно трудно кратко описать основные возможности этого полнофункционального фреймворка, не продублировав при этом его полную документацию.



В документации проекта `pytest` можно найти очень много информации, примеров и подробных описаний возможностей (<https://oreil.ly/PSAu2>), заслуживающих внимания. По мере выхода новых версий этого проекта неизменно появляются интересные новинки и различные способы усовершенствования тестирования.

Первое знакомство Альфредо с этим фреймворком состоялось, когда он мучился с написанием тестов и неудобным встроенным способом тестирования Python с использованием `unittest` (мы обсудим различия между ним и `pytest` далее в этой главе). Ему хватило всего нескольких минут, чтобы оценить волшебные возможности `pytest`. При этом не нужно было отступать от привычной схемы написания тестов и все это работало сразу, без каких-либо модификаций! Подобная гибкость характерна для всего проекта `pytest`, и даже если какие-то возможности сегодня в нем отсутствуют, его функциональность всегда можно расширить с помощью плагинов или файлов конфигурации.

Рассмотревшись, как писать более понятные тестовые сценарии, и научившись использовать возможности утилиты командной строки, механизм отчетов, расширяемости с помощью плагинов, также различных утилит фреймворка, вы захотите писать все больше и больше тестов, что, несомненно, хорошо во всех отношениях.

Начало работы с pytest

В простейшем виде `pytest` представляет собой утилиту командной строки, которая находит тесты Python и выполняет их. Пользователю не обязательно понимать ее внутреннее устройство, что сильно упрощает работу с ней. В этом разделе мы рассмотрим основные возможности фреймворка `pytest`, от написания тестов до размещения файлов (для автоматического обнаружения), и обсудим основные различия между ним и `unittest` — встроенным фреймворком тестирования Python.



Большинство интегрированных сред разработки (IDE), например PyCharm и Visual Studio Code, обладают встроенной поддержкой `pytest`. Текстовый редактор Vim поддерживается посредством плагина `pytest.vim` (<https://oreil.ly/HowKu>). Использование `pytest` из редактора экономит время и упрощает отладку в случае сбоев, но учтите, что поддерживаются не все возможности и плагины.

Тестирование с помощью pytest

Убедитесь, что `pytest` установлен и доступен для вызов в командной строке:

```
$ python3 -m venv testing
$ source testing/bin/activate
```

Создайте файл `test_basic.py`, он должен выглядеть следующим образом:

```
def test_simple():
    assert True

def test_fails():
    assert False
```

При запуске `pytest` без аргументов должно быть выведено сообщение о пройденном и не пройденном тесте:

```
$ (testing) pytest
===== test session starts =====
platform linux -- Python 3.6.8, pytest-4.4.1, py-1.8.0, pluggy-0.9.0
rootdir: /home/alfredo/python/testing
collected 2 items
test_basic.py .F [100%]
===== FAILURES =====
_____ test_fails _____
      def test_fails():
>         assert False
E         AssertionError
test_basic.py:6: AssertionError
===== 1 failed, 1 passed in 0.02 seconds =====
```

Выводимая информация немедленно начинает приносить пользу, показывая, сколько тестов было собрано, сколько пройдено успешно и какой тест не пройден, включая его номер строки.



Выводимая `pytest` по умолчанию информация полезна, но, возможно, слишком многословна. Управлять количеством выводимой информации можно с помощью опций, чтобы сократить ее, воспользуйтесь флагом `-q`.

Создавать классы, включающие тесты, не обязательно — все функции были найдены и выполнены должным образом. В набор тестов могут входить тесты обоих видов, и фреймворк прекрасно работает в подобных условиях.

Макеты и соглашения

При тестировании в Python `pytest` неявно придерживается нескольких соглашений, большинство которых относятся к именованиям и структуре. Например, попробуйте переименовать файл `test_basic.py` в `basic.py` и запустить `pytest`:

```
$ (testing) pytest -q
no tests ran in 0.00 seconds
```

Никаких тестов запущено не было в соответствии с соглашением о том, что названия файлов тестов должны начинаться с `test_`. Если переименовать файл обратно в `test_basic.py`, он будет автоматически обнаружен и тесты будут выполнены.



Макеты и соглашения удобны для автоматического обнаружения тестов. Фреймворк `pytest` можно не строить и использовать другие соглашения именованиях или для непосредственного выполнения файла с другим названием. Однако удобнее придерживаться соглашений по умолчанию во избежание путаницы, если тесты вдруг не запустятся.

Вот соглашения, которым будет следовать утилита `pytest` при обнаружении тестов.

- Каталог с тестами должен называться `tests`.
- Названия файлов тестов должны начинаться с `test`, например `test_basic.py`, либо заканчиваться на `test.py`.
- Названия функций тестов должны начинаться с `test_`, например `def test_simple():`.
- Названия классов тестов должны начинаться на `Test`, например `class TestSimple`.
- Методы тестов придерживаются тех же соглашений, что и функции, и должны начинаться на `test_`, например `def test_method(self):`.

А поскольку для автоматического обнаружения и выполнения тестов требуется, чтобы названия начинались на `test_`, можно просто создать вспомогательные функции и прочий не относящийся непосредственно к тестам код с другими названиями, и они будут исключаться автоматически.

Отличия от unittest

Python изначально включает набор утилит и вспомогательных функций для тестирования, входящих в состав модуля `unittest`. Так что не помешает поинтересоваться, чем отличается `pytest` и почему именно его рекомендуется применять.

Модуль `unittest` вызывает использование классов и их наследование. Это не проблема для опытного разработчика, хорошо разбирающегося в объектно-ориентированном программировании и наследовании классов, но для начинающих становится препятствием. Нельзя требовать применения классов и наследования для написания простейших тестов!

В частности, из-за необходимости наследовать от класса `unittest.TestCase` разработчику приходится знать (и помнить) большинство методов-операторов контроля, применяемых для верификации результатов. При использовании `pytest` все это отвечает одна единственная вспомогательная функция контроля `assert`.

Вот лишь часть методов-операторов контроля, которые можно использовать при написании тестов на основе `unittest`. Понять суть некоторых из них несложно, но разобраться с остальными весьма непросто:

- `self.assertEqual(a, b);`
- `self.assertNotEqual(a, b);`
- `self.assertTrue(x);`
- `self.assertFalse(x);`
- `self.assertIs(a, b);`
- `self.assertIsNot(a, b);`
- `self.assertIsNone(x);`
- `self.assertIsNotNone(x);`
- `self.assertIn(a, b);`
- `self.assertNotIn(a, b);`
- `self.assertIsInstance(a, b);`
- `self.assertNotIsInstance(a, b);`
- `self.assertRaises(exc, fun, *args, **kws);`
- `self.assertRaisesRegex(exc, r, fun, *args, **kws);`
- `self.assertWarns(warn, fun, *args, **kws);`

- `self.assertWarnsRegex(warn, r, fun, *args, **kwargs);`
- `self.assertLogs(logger, level);`
- `self.assertMultiLineEqual(a, b);`
- `self.assertSequenceEqual(a, b);`
- `self.assertListEqual(a, b);`
- `self.assertTupleEqual(a, b);`
- `self.assertSetEqual(a, b);`
- `self.assertDictEqual(a, b);`
- `self.assertAlmostEqual(a, b);`
- `self.assertNotAlmostEqual(a, b);`
- `self.assertGreater(a, b);`
- `self.assertGreaterEqual(a, b);`
- `self.assertLess(a, b);`
- `self.assertLessEqual(a, b);`
- `self.assertRegex(s, r);`
- `self.assertNotRegex(s, r);`
- `self.assertCountEqual(a, b).`

pytest дает возможность пользоваться исключительно `assert` и не требует применения чего-либо из перечисленного. Более того, он *позволяет* писать тесты с помощью `unittest` и даже выполняет их. Мы настоятельно рекомендуем этого не делать и предлагать вместо действий простые операторы `assert`.

Использовать простые операторы `assert` не только удобнее — pytest также предоставляет обдуемый большими возможностями механизм встраивания для случая непрохождения тестов (больше об этом расскажем в следующем разделе).

Возможности pytest

Помимо упрощения написания и выполнения тестов, фреймворк `pytest` предоставляет множество расширяемых опций, в частности точек подключения. Они позволяют взаимодействовать с внутренними механизмами фреймворка на различных этапах выполнения. Например, можно добывать точку подключения для механизма сбора тестов, чтобы внести изменения в процесс сборки. Еще один

полезный пример — релизация более информативного отчета о том, что тест не пройден.

При просмотре API HTTP мы порой обнаруживали отсутствие критической пользы от непрохождения тестов, применяющих HTTP-запросы к приложению: систем просто сообщалось об отсутствии операторов контроля, поскольку вместо ожидаемого ответа (HTTP 200) был получен ответ HTTP 500. Мы хотели больше узнать о записях: к конечной точке с каким URL он производился? Если это был запрос POST, включали ли он данные? И если да, то какие? Вся эта информация присутствует в объекте HTTP-ответа, так что мы написали точку подключения, чтобы взглянуть внутрь этого объекта и включить все упомянутые элементы в отчет о том, что тест не пройден.

Точки подключения — продвинутая возможность `pytest`, которая, возможно, вам и не понадобится, однако не помешает понимать, что фреймворк `pytest` достаточно гибок, чтобы удовлетворить различные требования. В следующих разделах мы рассмотрим, как расширить возможности этого фреймворка, почему оператор `assert` не столько ценен, как параметризовать тесты с целью сокращения повторов, как создавать вспомогательные функции с помощью фикстур, так же как использовать уже имеющиеся встроенные.

conftest.py

Почти все программное обеспечение позволяет расширять функциональность с помощью плагина (например, в веб-браузерах они называются *расширениями* (extensions)), и в `pytest` так же имеется обширный набор возможностей API для просмотра плагинов. Мы рассмотрим не весь API, лишь более простой подход — файл `conftest.py`. В нем можно расширять функциональность утилит *совершенно так же, как плагины*. При этом не нужно знать во всех деталях, как создать отдельный плагин, преобразовать его в пакет и установить. Фреймворк `pytest` загружает файл `conftest.py` (при его наличии) и считывает из него все инструкции. Все происходит автоматически!

Обычно файл `conftest.py` включает точки подключения, фикстуры и вспомогательные функции для последних. Если объявить эти фикстуры в качестве аргументов, можно использовать их внутри тестов (этот процесс описано далее в разделе, посвященном фикстурам).

Фикстуры и вспомогательные функции имеет смысл добавлять в этот файл, если его будут использовать несколько модулей. Если же файл тестов только один или фикстуры либо точки подключения будут применяться только в одном файле, создавать или использовать файл `conftest.py` смысла нет. Фикстуры

и вспомогательные функции можно описать в том же файле, что и тест, они будут вести себя точно так же.

Единственное условие для загрузки файла `conftest.py`: он должен находиться в каталоге `tests` и называться именно так. Кроме того, хотя название можно менять в настройках, мы не рекомендуем это делать и советуем в большинстве случаев избегать возможных проблем, придерживаясь принятых по умолчанию соглашений о наименованиях.

Этот замечательный оператор `assert`

Когда мы хотим рассказать о том, насколько замечательным инструментом является `pytest`, то начинаем с важнейших вариантов использования оператора `assert`. «Закулисами» фреймворка `pytest` просматриваются объекты и предоставляется механизм сравнения, чтобы лучше описать ошибки. Разработчики обычно не хотят закрывать эти возможности, поскольку простой оператор `assert` в Python очень плохо описывает ошибки. Рассмотрим для примера две длинные строки:

```
>>> assert "using assert for errors" == "using asert for errors"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Чем они различаются? Трудно ответить на этот вопрос, если долго в них не всматриваться. Поэтому многие и не рекомендуют его использовать. Маленький тест демонстрирует, насколько иначе сообщается об ошибках `pytest`:

```
$ (testing) pytest test_long_lines.py
===== test session starts =====
platform linux -- Python 3.6.8, pytest-4.4.1, py-1.8.0, pluggy-0.9.0
collected 1 item
test_long_lines.py F [100%]
===== FAILURES =====
_____ test_long_lines _____
    def test_long_lines():
>         assert "using assert for errors" == "using asert for errors"
E         AssertionError: assert '...rt for errors' == '...rt for errors'
E             - using assert for errors
E             ?           -
E             + using asert for errors
test_long_lines.py:2: AssertionError
===== 1 failed in 0.04 seconds =====
```

Можете ли вы теперь сказать, в чем различие? *Н много легче*. `pytest` не только сообщает, что контроль не пройден, но и указывает точно, где найдено расхождение. Этот пример — всего лишь простой оператор контроля для длинных строк,

но фреймворк `pytest` может без проблем работать и с другими структурами данных, например со списками и ассоциативными массивами. Приходилось ли вам сравнивать в тесте очень длинные списки? Вот небольшой фрагмент кода с длинными списками:

```
assert ['a', 'very', 'long', 'list', 'of', 'items'] == [
    'a', 'very', 'long', 'list', 'items']
E AssertionError: assert [...'of', 'items'] == [...'list', 'items']
E     At index 4 diff: 'of' != 'items'
E     Left contains more items, first extra item: 'items'
E     Use -v to get the full diff
```

Проинформированного пользователя о том, что тест не пройден, фреймворк точно укажет позицию элемента (индекс 4, то есть пятый элемент) и сообщит, что в одном из списков один элемент больше. Без этого уровня детектировки отладка занимала бы очень много времени. Еще одно достоинство этого отчета — по умолчанию очень длинные элементы опускаются при сравнении, так что выводится только имеющая отношение к делу порция информации. В конце концов, нам хочется знать не только что списки (или другие структуры данных) различаются, но и *в каком именно месте*.

Параметризация

Параметризация — одна из возможностей, работающая в которой непросто, поскольку в `unittest` она отсутствует и имеется только в фреймворке `pytest`. Но все становится понятно, когда вам придется писать очень похожие тесты, исследующие одно и то же, но с небольшими различиями во входных данных. Возьмем для примера класс для тестирования функции, который должен возвращать `True`, если переданное ей строковое значение описывает истину:

```
from my_module import string_to_bool

class TestStringToBool(object):

    def test_it_detects_lowercase_yes(self):
        assert string_to_bool('yes')

    def test_it_detects_odd_case_yes(self):
        assert string_to_bool('YeS')

    def test_it_detects_uppercase_yes(self):
        assert string_to_bool('YES')

    def test_it_detects_positive_str_integers(self):
        assert string_to_bool('1')

    def test_it_detects_true(self):
        assert string_to_bool('true')
```



```
def test_it_detects_true_with_trailing_spaces(self):
    assert string_to_bool('true ')

def test_it_detects_true_with_leading_spaces(self):
    assert string_to_bool(' true')
```

Видите, как все эти тесты вычисляют один и тот же результат на основе схожих входных данных? Именно в подобных случаях для группировки всех этих изменений и передчи их в тест и пригодится параметризация — она позволяет свести все эти тесты к одному:

```
import pytest
from my_module import string_to_bool

true_values = ['yes', '1', 'Yes', 'TRUE', 'TrueE', 'True', 'true']

class TestStrToBool(object):

    @pytest.mark.parametrize('value', true_values)
    def test_it_detects_truish_strings(self, value):
        assert string_to_bool(value)
```

Здесь происходит несколько вещей. Во-первых, чтобы использовать модуль `pytest.mark.parametrize`, импортируется `pytest` (фреймворк), далее описываются `true_values` в виде переменной (список) со всеми значениями, которые должны дать один результат, и, наконец, все тестовые методы заменяются одним, в котором применяется декоратор `parametrize`, объявляющий два аргумента. Первый представляет собой строку `value`, второй — значение из нее объявленного списка. Выглядит это немного странно, но, по существу, просто указывает фреймворку, что в качестве аргумента в тестовом методе необходимо использовать значение `value`. Именно отсюда и берется аргумент `value`!

И хотя при запуске выводится довольно много информации, но в ней четко покажется, как значения передются. Выглядит все практически как один тест, клонированный для каждого из переданных значений:

```
test_long_lines.py::TestLongLines::test_detects_truish_strings[yes] PASSED
test_long_lines.py::TestLongLines::test_detects_truish_strings[1] PASSED
test_long_lines.py::TestLongLines::test_detects_truish_strings[Yes] PASSED
test_long_lines.py::TestLongLines::test_detects_truish_strings[TRUE] PASSED
test_long_lines.py::TestLongLines::test_detects_truish_strings[TrueE] PASSED
test_long_lines.py::TestLongLines::test_detects_truish_strings[True] PASSED
test_long_lines.py::TestLongLines::test_detects_truish_strings[true] PASSED
```

При этом в квадратных скобках выводятся значения, используемые каждой из итераций *одного и того же теста*. Благодаря `parametrize` довольно обширный класс тестов сокращается до одного тестового метода. Когда в следующий раз вы

столкнетесь с необходимостью писать очень похожие тесты для контроля одного и того же исходного кода при различных входных данных, вы будете знать, что можно облегчить себе работу с помощью декоратора `parametrize`.

Фикстуры

Фикстуры — фреймворк `pytest` (<https://oreil.ly/gPoM5>) — своего рода маленькие вспомогательные функции, внедряемые в тест. Вне зависимости от того, пишете вы отдельную тестовую функцию или набор тестовых методов, фикстуры можно использовать одинаково. Если не планируется применять их в других тестовых файлах, вполне можно объявлять их в одном, в противном случае их можно объявить в файле `conftest.py`. Фикстуры, как и вспомогательные функции, могут быть всем необходимым для теста, от простых значений до сложных структур данных до более сложных, например баз данных для веб-приложения.

Для этих вспомогательных функций существует понятие *области видимости* (`scope`). Они могут включать специальный код очистки для каждого тестового метода, класса и модуля или, возможно, даже для всего тестового сеанса. Описывая их в тестовом методе (или тестовой функции), вы, по сути, внедряете фикстуру во время выполнения. Если это не совсем понятно, вы все поймете по мере изучения примеров в следующих разделах.

Приступим

Описывая и используя фикстуры так просто, что часто ими злоупотребляют. Мы знаем это, поскольку с нами создали немало тех, которые могли бы быть простыми вспомогательными методами! Как мы уже упоминали, сценариев применения фикстур очень много — от простых структур данных до более сложных, например строки целых баз данных для отдельного теста.

Недавно Альфредо пришлось тестировать маленькое приложение, предназначенное для синтаксического разборки конкретного файла — файла *keyring* (содержащего набор ключей). Его структура напоминает INI-файл, он включает значения, которые должны быть уникальными и соответствовать определенному формату. Воссоздать структуру этого файла для каждого теста было бы очень утомительно, так что он создал фикстуру. Вот как выглядел файл `keyring`:

```
[mon.]
key = AQBvaBFZAAAAABAA9VHgWcg3rWn8fMaX8KL01A==
caps_mon = "allow *"
```

Фикстурой служил функция, возвращающая содержимое файла с набором ключей. Создадим новый файл `test_keyring.py`, включающий фикстуру и маленькую тестовую функцию для проверки ключа по умолчанию:

```
import pytest
import random

@pytest.fixture
def mon_keyring():
    def make_keyring(default=False):
        if default:
            key = "AQBVaBFZAAAAABAA9VHgwCg3rWn8fMaX8KL01A=="
        else:
            key = "%032x==" % random.getrandbits(128)
        return ""

    [mon.]
    key = %s
    caps_mon = "allow *"
    "" % key
    return make_keyring

def test_default_key(mon_keyring):
    contents = mon_keyring(default=True)
    assert "AQBVaBFZAAAAABAA9VHgwCg3rWn8fMaX8KL01A==" in contents
```

Фикстура действует вложенную функцию, выполняющую основную работу, позволяя использовать значение ключа по умолчанию либо вернуть вложенную функцию, если вызывающей стороне нужен случайный ключ. Внутри теста мы получим фикстуру путем объявления ее аргументом тестовой функции (в данном случае `mon_keyring`) и вызовем фикстуру с параметром `default=True`, используя ключ по умолчанию, после чего проверяем, было ли сгенерировано то, что ожидалось.



На практике сгенерированное содержимое передвместит синтаксического разбор, чтобы гарантировать ожидаемое поведение после разбора и отсутствие ошибок в его ходе.

Предназначенный для промышленной эксплуатации код, использующий эту фикстуру, постепенно начал выполнять и другие виды тестирования, и в какой-то момент тест должен был проверять, может ли средство синтаксического разбора обрабатывать файлы при различных условиях. Фикстура возвращала строку, так что необходимо было расширить ее функциональность. Существующие тесты уже использовали фикстуру `mon_keyring`, поэтому, чтобы расширить функциональность, не затрагивая существующую фикстуру, был создан новый, использующий возможности фреймворка. Фикстуры могут *обращаться*

к другим фикстур м! Необходимо опис ь требуемую фикстуру к к ргумент (н логично тестовой функции или тестовому методу), чтобы фреймворк внедрял ее при выполнении.

Вот к к выглядит нов я фикстур , созд ющ я (и возвр щ ющ я) ф йл:

```
@pytest.fixture
def keyring_file(mon_keyring, tmpdir):
    def generate_file(default=False):
        keyring = tmpdir.join('keyring')
        keyring.write_text(mon_keyring(default=default))
        return keyring.strpath
    return generate_file
```

Пройдем по этому коду строк з строкой. Декор тор `pytest.fixture` сообщ - ет фреймворку, что д нн я функция является фикстурой, д лее описыв ется собственно фикстур с *двумя фикстур ми*, `mon_keyring` и `tmpdir`, в к честве ргументов. Первую из них мы созд ли р нее в ф йле `test_keyring.py`, вто - р я — встроенн я фикстур из фреймворк `pytest` (больше о встроенных фикстур х вы узн ете в следующем р зделе). Фикстур `tmpdir` позволяет использо в ь временный к т лог,уд ляемый по з вершении тест ,д лее созд ется ф йл `keyring` и з писыв ется текст, сгенериров нный фикстурой `mon_keyring` с ргументом `default`. Н конец, он возвр щ ет бсолютный путь созд нного ф йл для использов ния тестом.

Вот к к тестов я функция могл бы использов ь эту фикстуру:

```
def test_keyring_file_contents(keyring_file):
    keyring_path = keyring_file(default=True)
    with open(keyring_path) as fp:
        contents = fp.read()
    assert "AQBVaBFZAAAAABAA9VHgwCg3rWn8fMaX8KL01A==" in contents
```

Теперь вы уже хорошо предст вляете себе, что т кое фикстуры, где их мож - но опис ь и к к использо в ь в тест х. В следующем р зделе р ссмотрим некоторые из н иболее удобных встроенных фикстур, входящих в сост в фреймворк `pytest`.

Встроенные фикстуры

В предыдущем р зделе мы вкр тце упомянули одну из множеств встроенных фикстур, доступных в `pytest`, — фикстуру `tmpdir`. В этом фреймворке есть еще много фикстур. Полный список доступных фикстур можно просмотреть с по - мощью следующей ком нды:

```
$ (testing) pytest -q --fixtures
```

Особенно часто мы используем две фикстуры, упомянутые в списке, выводимом предыдущей командой, — `monkeypatch` и `capsys`. Вот краткое их описание, которое вы увидите в терминале:

```
capsys
    enables capturing of writes to sys.stdout/sys.stderr and makes
    captured output available via ``capsys.readouterr()`` method calls
    which return a ``(out, err)`` tuple.

monkeypatch
    The returned ``monkeypatch`` funcarg provides these
    helper methods to modify objects, dictionaries or os.environ::

    monkeypatch.setattr(obj, name, value, raising=True)
    monkeypatch.delattr(obj, name, raising=True)
    monkeypatch.setitem(mapping, name, value)
    monkeypatch.delitem(obj, name, raising=True)
    monkeypatch.setenv(name, value, prepend=False)
    monkeypatch.delenv(name, value, raising=True)
    monkeypatch.syspath_prepend(path)
    monkeypatch.chdir(path)

    All modifications will be undone after the requesting
    test function has finished. The ``raising``
    parameter determines if a KeyError or AttributeError
    will be raised if the set/deletion operation has no target.
```

Фикстуры `capsys` захватывают всю информацию, выводимую в ходе теста в потоки `stdout` и `stderr`. Попробуйте ли вы когда-нибудь проверить, что выводит какой-либо компонент в консоль или журнал при модульном тестировании? Реализовать это правильно совсем не просто и иногда требует отдельного планинги или библиотеки для внесения временных изменений (патчинг) во внутреннее содержание Python и его последующего изучения.

Вот две тестовые функции для проверки информации, выведенной в потоки `stdout` и `stderr` соответственно:

```
import sys

def stderr_logging():
    sys.stderr.write('stderr output being produced')

def stdout_logging():
    sys.stdout.write('stdout output being produced')

def test_verify_stderr(capsys):
    stderr_logging()
    out, err = capsys.readouterr()
    assert out == ''
    assert err == 'stderr output being produced'
```

```
def test_verify_stdout(capsys):
    stdout_logging()
    out, err = capsys.readouterr()
    assert out == 'stdout output being produced'
    assert err == ''
```

Фикстура `capsys` выполняет весь планинг, настройки и вспомогательные операции по извлечению информации, выведенной в потоки `stdout` и `stderr` в ходе тест. Содержимое очищается для каждого нового теста, гарантируя тем самым выполнение переменных привильными значениями.

Но чаще всего, и верное, мы используем фикстуру `monkeypatch`. При тестировании встречаются ситуации, когда мы не можем похвастаться контролем над тестируемым кодом, и, чтобы добиться конкретного поведения модуля или функции, необходим планинг. В экосистеме Python существует немало библиотек для планинга и имитационного моделирования (*имитационные объекты* (mocks) — это вспомогательные функции, предназначенные для задания поведения пропущенных объектов), но `monkeypatch` достаточно хорош для того, чтобы вам не нужно было устанавливать отдельную библиотеку.

Следующая функция выполняет системную команду для захвата информации с устройств, после чего производит синтаксический разбор полученного и возвращает свойство (сообщаемое командой `blkid ID_PART_ENTRY_TYPE`):

```
import subprocess

def get_part_entry_type(device):
    """
    Производит синтаксический разбор ``ID_PART_ENTRY_TYPE`` из "низкоуровневых"
    выходных данных (игнорирует кэш) с типом выходных данных ``udev``.
    """
    stdout = subprocess.check_output(['blkid', '-p', '-o', 'udev', device])
    for line in stdout.split('\n'):
        if 'ID_PART_ENTRY_TYPE=' in line:
            return line.split('=')[-1].strip()
    return ''
```

Чтобы протестировать ее, давайте желаемое поведение в трибуне `check_output` модуля `subprocess`. Вот как выглядит тестовая функция, использующая фикстуру `monkeypatch`:

```
def test_parses_id_entry_type(monkeypatch):
    monkeypatch.setattr(
        'subprocess.check_output',
        lambda cmd: '\nID_PART_ENTRY_TYPE=aaaaa')
    assert get_part_entry_type('/dev/sda') == 'aaaa'
```

Вызов `setattr` уст н *вливет* *трибут* проп тченного вызыв емого объект (в д нном случ е `check_output`). П тч состоит из лямбд -функции, возвр щ - ющей одну интересную строку. Поскольку функция `subprocess.check_output` н м н прямую неподвл стн , функция `get_part_entry_type` не позволяет внедрять зн чения к ким-то другим способом, единственный в ри нт — п тчинг.

Мы предпочит ем пробов ть другие методики, н пример внедрение зн чений (известное под н зв нием «внедрение з висимостей» (*dependency injection*)), прежде чем использов ть п тч, но иногда другого выход нет. Удобство `pytest` во многом и состоит в том, что он может выполнять п тчинг при тестиров нии, потом освобожд ть все ресурсы.

Инфраструктурное тестирование

В этом р зделе мы р сск жем о тестиров нии и проверке инфр структуры с помощью проект `Testinfra` (<https://oreil.ly/e7Afx>) — пл гин `pytest` для инфр структуры, ктивно использующего фикстуры, с помощью которого можно пис ть тесты Python н логично тестиров нию код .

В предыдущих р здел х приводились некоторые подробности использов ния и примеры `pytest`, этот р здел н чнем с понятия верифик ции н системном уровне. Мы р сск жем о тестиров нии инфр структуры в контексте ответ н вопрос: «*К к выяснить, было ли р звертыв ние успешным?*» В большинстве случ ев для ответ н него требуются производимые вручную проверки, н пример з грузк веб-с йт или просмотр список процессов, но этого недост точно и это чрев то ошибк ми, к тому же может ок з ться весьма утомительным при зн чительных р змер х системы.

И хотя с `pytest` обычно зн комятся к к со средством для н пис ния и выполнения модульных тестов Python, имеет смысл перепрофилиров ть его для тестиров ния инфр структуры. Несколько лет н з д Альфредо д ли з д ние созд ть прогр мму уст новки, возможности которой были бы доступны через HTTP API. Он был предн зн чен для созд ния кл стер Ceph (<https://ceph.com>), включ ющего довольно большое количество м шин. В ходе контроля к честв введения этого API в эксплу тацию Альфредо нередко получ л отчеты о том, что кл стер р бот ет не т к, к к нужно, т к что ему приходилось использов ть учетные д нные для вход н эти м шины и их тщ тельного изучения. При отл дке р спределенной системы из нескольких м шин возник л мультиплик тивный эффект: несколько ф йлов конфигурации, р зличные жесткие диски, сетевые н стройки, причем р злич ться могло все что угодно, д же если н первый взгляд все было идентично.

При каждой отладке этих систем список того, что нужно проверить, у Альфредо все рос и рос. Один ковали конфигурация всех серверов? Проводит ли тестирование, как ожидалось? Существует ли какой-то конкретный пользователь? В конце концов он забывал что-нибудь и тратил время на выяснение того, что он забыл. Совершенно нерациональный процесс. *Можно ли написать простые тестовые сценарии для тестирования кластера?* Для проверки отдельных пунктов список Альфредо написал несколько простых тестов, которые выполнял на состояниях кластеров машин. Не успел он опомниться, как собрал неплохой набор тестов для выявления всех возможных проблем, выполнение которого занимало всего несколько секунд.

Это было настоящее откровение в смысле усовершенствования процесса поставки. Альфредо уже мог выполнять эти функциональные тесты во время разработки программы установки и не ходить потенциальные проблемы. Если же какие-то проблемы обнаруживались группой контроля качества, он мог запустить эти же тесты в их системе. Иногда тесты выявляли проблемы со средой: диск был испорчен, в результате чего развертывание завершилось неудачей, или проблемы вызывал неудачный файл конфигурации из другого кластера. Автоматизация, мелкомодульные тесты и возможность их выполнять зачастую повышают качество работы и снижают нагрузку командой контроля качества.

В проекте TestInfra есть самые разнообразные фикстуры для эффективного тестирования систем, он включает полный набор прикладных чистей для подключения к серверам вне зависимости от типа развертывания: Ansible, Docker, SSH и Kubernetes — все они входят в число поддерживаемых подключений. Благодаря поддержке множеств различных прикладных чистей для подключения можно выполнять один и тот же набор тестов вне зависимости от изменений инфраструктуры.

В следующих разделах мы рассмотрим различные прикладные чисты и примеры реальных проектов.

Что такое проверка системы

Проверка системы может выполняться на различных уровнях (с помощью систем мониторинга и оповещения) и на разных этапах жизненного цикла приложения, например во время развертывания, выполнения или разработки. Приложение, которое Альфредо недавно ввел в промышленную эксплуатацию, должно было корректно обслуживать клиентские подключения без каких-либо сбоев даже при перезапуске. Чтобы выдерживать требуемый объем трафика, в приложении использовались балансировка и грузки: при сильной нагрузке

н систему новые соединения перен пр влялись н другие, менее згруженные серверы.

При рзвертыв нии новой версии приложение необходимо было *перез пустить*. Перез пуск озн ч л для клиентов в лучшем случ е, что приложение будет вести себя стр нно, то и р бот ть откровенно плохо. Во избеж ние этого процесс перезапуска жд л з вершения всех клиентских соединений, систем отк зыв л клиент м в новых соединениях, чтобы можно было з вершить опер ции с имеющимися клиент ми, ост льн я ч сть системы бр л н себя р боту. А когд не ост в лось ктивных соединений, рзвертыв ние продолж лось и сервисы ост н влив лись для включения нового код .

Н к ждом эт пе этого процесс выполнял сь проверк : до того, к к средство рзвертыв ния ук зыв ет б л нсировщику прекр тить отпр вку з д ний от новых клиентов, и позднее, чтобы убедиться в отсутствии ктивных клиентов. Если превр тить этот технологический процесс в тест, то опис ние его могло быть примерно следующим: *убедиться, что в н стоящий момент ник кие клиенты не р бот ют*. После внесения нового код нужно еще убедиться, что б л нсировщик подтвердил готовность сервер к р боте. Еще один возможный тест: *б л нсировщик счит ет сервер ктивным*. Н конец, необходимо удостоверить, что сервер получ ет новые клиентские соединения, — еще один тест, который нужно н пис ть!

Из этих ш гов, для которых можно н пис ть соответствующие тесты, и состоит верифик ция подобного технологического процесс .

Мониторинг общего р бочего состояния сервер (или серверов в кл стерной среде) т кже может относиться к проверке системы либо может быть сост вной ч стью непрерывной интегр ции при р зр ботке приложения и функцион льном тестиров нии. К этим ситу циям, рвно и к любым другим, в которых может ок з ться полезн верифик ция состояния, отлично применимы б зовые идеи проверки. Их не следует использов ть исключительно для тестирования, хотя для н ч л это и неплохо!

Введение в Testinfra

Н пис ние модульных тестов для тестирования инфр структуры — з меч тельн я идея, и из н шего более чем годовичного опыт р боты с проектом Testinfra мы можем сдел ть вывод, что он позволил улучшить к чество приложений, которые мы пост вляли для промышленной эксплу т ции. В следующих р здел х обсудим р зличные подробности, в ч стности подключение к р зным узл м и выполнение проверочных тестов, т кже поговорим о доступных тип х фикстур.

Создайте новую виртуальную среду и установите **pytest**:

```
$ python3 -m venv validation
$ source testing/bin/activate
(validation) $ pip install pytest
```

Установите **testinfra** версии 2.1.0:

```
(validation) $ pip install "testinfra==2.1.0"
```



Фикстуры **pytest** предоставляют всю тестовую функциональность проекта **Testinfra**. Чтобы извлечь пользу из этого рэпозитория, вам необходимо знать, как они работают.

Подключение к удаленным узлам

Поскольку существуют различные типы соединений прикладных чистей, то, если соединение не установлено непосредственно, **Testinfra** использует определенные типы по умолчанию. Лучше указать тип соединения явным образом в командной строке.

Вот список поддерживаемых **Testinfra** соединений:

- локальное;
- **Paramiko** (SSH-реализация на Python);
- **Docker**;
- **SSH**;
- **Salt**;
- **Ansible**;
- **Kubernetes** (через утилиту командной строки **kubectl**);
- **WinRM**;
- **LXC**.

В меню справки **pytest** можно найти репозиторий **testinfra** с пояснениями по поводу имеющихся флагов. Это очень удобная возможность, которая обязана своим возникновением фреймворку **pytest** и его интеграции с **Testinfra**. Справку по обоим проектам можно получить с помощью одной и той же команды:

```
(validation) $ pytest --help
```

```
...
```

```
testinfra:
```

```
--connection=CONNECTION
                        Remote connection backend (paramiko, ssh, safe-ssh,
                        salt, docker, ansible)
--hosts=HOSTS           Hosts list (comma separated)
--ssh-config=SSH_CONFIG
                        SSH config file
--ssh-identity-file=SSH_IDENTITY_FILE
                        SSH identify file
--sudo                 Use sudo
--sudo-user=SUDO_USER
                        sudo user
--ansible-inventory=ANSIBLE_INVENTORY
                        Ansible inventory file
--nagios               Nagios plugin
```

Пусть даны два роботающих сервера. Чтобы продемонстрировать опции соединений, проверим, роботят ли на них CentOS 7, взглянув для этого в файл `/etc/os-release`. Вот как выглядит соответствующая тестовая функция (сохранен в файле `test_remote.py`):

```
def test_release_file(host):
    release_file = host.file("/etc/os-release")
    assert release_file.contains('CentOS')
    assert release_file.contains('VERSION="7 (Core)"')
```

Это отдельная тестовая функция, которая принимает в качестве аргумента фикстуру `host` и выполняется для всех указанных узлов.

Флаг `--hosts` позволяет указать список серверов со схемами соединения (например, в случае SSH — `ssh://имя_хоста`), допустимы также некоторые варианты с подстановками. Передать в одной строке указанные серверы неудобно, если тестировать три или более чем один-два. Вот как выглядит тестирование двух серверов с помощью SSH:

```
(validation) $ pytest -v --hosts='ssh://node1,ssh://node2' test_remote.py
===== test session starts =====
platform linux -- Python 3.6.8, pytest-4.4.1, py-1.8.0, pluggy-0.9.0
cachedir: .pytest_cache
rootdir: /home/alfredo/python/python-devops/samples/chapter16
plugins: testinfra-3.0.0, xdist-1.28.0, forked-1.0.2
collected 2 items

test_remote.py::test_release_file[ssh://node1] PASSED [ 50%]
test_remote.py::test_release_file[ssh://node2] PASSED [100%]
===== 2 passed in 3.82 seconds =====
```

Выводим я при повышенном уровне детализации (флаг -v) информация демонстрирует, что Testinfra выполняет одну тестовую функцию для двух узлов в спецификации вызов серверов.



При настройке серверов важно обеспечить возможность соединения без проля. Никких запросов проля быть не должно, при использовании SSH желательно применять конфигурацию на основе ключей.

При этом типичными подобными тестами (в качестве примера в системе непрерывной интеграции, например) удобно генерировать серверы, определяя тип подключения и любые специфические инструкции. Testinfra может читать информацию о том, к каким серверам подключиться, из файла конфигурации SSH. При предыдущем запуске теста для создания этих серверов со специфическими ключами и настройками соединений использовался Vagrant (<https://www.vagrantup.com>). Он может генерировать специализированные файлы конфигурации SSH для создаваемых им серверов:

```
(validation) $ vagrant ssh-config
Host node1
  HostName 127.0.0.1
  User vagrant
  Port 2200
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /home/alfredo/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL

Host node2
  HostName 127.0.0.1
  User vagrant
  Port 2222
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /home/alfredo/.vagrant.d/insecure_private_key
  IdentitiesOnly yes
  LogLevel FATAL
```

Экспорт выводимого содержимого в файл с последующей передачей его в Testinfra обеспечивет большую гибкость, если серверов несколько:

```
(validation) $ vagrant ssh-config > ssh-config
(validation) $ pytest --hosts=default --ssh-config=ssh-config test_remote.py
```

Благодаря использованию флага `--hosts=default` можно не указывать серверы непосредственно в командной строке и читать их из конфигурации SSH. Даже без Vagrant применять конфигурацию SSH удобно при подключении к большому числу серверов с конкретными инструкциями.

Еще один вариант в случае, если узлы локальные, с доступом по SSH, или контейнеры Docker, — Ansible (<https://www.ansible.com>). Для тестирования полезным окажется реестр хостов (подобный конфигурации SSH) с группировкой их по различным разделам. Можно также создать группы хостов для выбора отдельных хостов для тестирования, вместо того чтобы тестировать сразу все.

Файл того реестра (под названием `hosts`) для `node1` и `node2` из предыдущего примера выглядит так:

```
[all]
node1
node2
```

Если выполнять для всех хостов, команда меняется на следующую:

```
$ pytest --connection=ansible --ansible-inventory=hosts test_remote.py
```

Можно описать в реестре и отдельную группу, если требуется исключить какие-либо хосты из тестирования. Если обозначенные узлы представляют собой веб-серверы и входят в группу `nginx`, можно протестировать только эту группу с помощью следующей команды:

```
$ pytest --hosts='ansible://nginx' --connection=ansible \
  --ansible-inventory=hosts test_remote.py
```



Для множеств системных команд необходимы полномочия суперпользователя. Testinfra позволяет указывать флаг `--sudo` или `--sudo-user` для повышения полномочий. Флаг `--sudo` ставит движок действий `sudo` при выполнении команд, флаг `--sudo-user` позволяет выполнять команду от имени другого пользователя, с более высокими полномочиями. Можно применять и фикстуру `run` напрямую.

Фикстуры и особые фикстуры

Может показаться, что до сих пор для проверки наличия файлов и их содержания в наших примерах использовались только фикстура `host`. Однако кое-что впечатление обманчиво. Фикстура `host` включает в себя все прочие фикстуры

проект Testinfra, обладающие широкими возможностями. Это значит, что в следующем примере с действиями фикстур `host.file`, имеющая много дополнительных возможностей. Можно использовать ее и напрямую:

```
In [1]: import testinfra

In [2]: host = testinfra.get_host('local://')

In [3]: node_file = host.file('/tmp')

In [4]: node_file.is_directory
Out[4]: True

In [5]: node_file.user
Out[5]: 'root'
```

Всеобъемлющая фикстура `host` использует обширный API проекта Testinfra, загружающий все возможности для каждого хоста, к которому подключается. Идея состоит в написании одного теста, применяемого для тестирования различных узлов, причем все — на основе одной и той же фикстуры `host`.

Аргументов у фикстуры `host` несколько десятков (https://oreil.ly/2_J-o). Вот некоторые из всего используемые.

- `host.ansible` — предоставляет полный доступ ко всем свойствам Ansible во время выполнения, например к хостам, реестру и переменным.
- `host.addr` — сетевые утилиты, такие как проверки IPV4 и IPV6, проверки доступности и разрешимости хоста.
- `host.docker` — прокси для API Docker для взаимодействия с контейнерами и проверки того, запущены ли они.
- `host.interface` — вспомогательные функции для получения адресов для заданного интерфейса.
- `host.iptables` — вспомогательные функции для верификации правил брандмауэра `host.iptables`.
- `host.mount_point` — проверка точек монтирования, типов файловой системы в путях и вриентов монтирования.
- `host.package` — удобна для выяснения того, установлен ли конкретный пакет, и если да, то какая версия.
- `host.process` — проверка запущенных процессов.
- `host.sudo` — позволяет выполнять команды с модификатором `sudo` или от имени другого пользователя.

- `host.system_info` — предоставляет виды метаданных системы, например версия дистрибутива, выпуск и его кодовое наименование.
- `host.check_output` — выполняет системную команду, проверяя выводимую ею информацию в случае успешного выполнения, может применяться в сочетании с `host.sudo`.
- `host.run` — выполняет системную команду, позволяя проверять код возврата, `host.stderr` и `host.stdout`.
- `host.run_expect` — проверяет, соответствует ли код возврата ожидаемому.

Примеры

Самое естественное и целесообразное применение проверочных тестов — во время создания новой системы развертывания. Ключевые принципы методологии *разработки через тестирование* (Test Driven Development, TDD), любое усовершенствование требует нового теста. Для этого разработчик добивается установить веб-сервер и запустить его на порте 80 для выданных статической страницей. По мере достижения прогресса мы будем добавлять новые тесты. Неотъемлемой составляющей написания тестов является осознание сбоев, так что смоделируем несколько проблем, например которых будем разбираться, что нужно исправить.

На свежем сервере Ubuntu начните с установки пакета Nginx:

```
$ apt install nginx
```

Создайте новый файл теста `test_webserver.py`, в который мы будем постепенно добавлять тесты:

```
def test_nginx_is_installed(host):
    assert host.package('nginx').is_installed
```

Сделайте вывод pytest закончилось с помощью флага `-q`, чтобы сосредоточить свое внимание на сбоях. Удаленный сервер называется `node4`, для подключения к нему используется SSH. Вот команда для запуска первого теста:

```
(validate) $ pytest -q --hosts='ssh://node4' test_webserver.py
.
1 passed in 1.44 seconds
```

Явный прогресс! Необходимо, чтобы веб-сервер был запущен и работал, так что добавим новый тест для проверки этого поведения:

```
def test_nginx_is_running(host):
    assert host.service('nginx').is_running
```

Повторный запуск теста, как и раньше, должен опять пройти успешно:

```
(validate) $ pytest -q --hosts='ssh://node4' test_webserver.py
.F
===== FAILURES =====
_____ test_nginx_is_running[ssh://node4] _____

host = <testinfra.host.Host object at 0x7f629bf1d668>

    def test_nginx_is_running(host):
>     assert host.service('nginx').is_running
E     AssertionError: assert False
E     + where False = <service nginx>.is_running
E     + where <service nginx> = <class 'SystemdService'>('nginx')

test_webserver.py:7: AssertionError
1 failed, 1 passed in 2.45 seconds
```

Некоторые дистрибутивы Linux не разрешают запускать сервисы при установке. Более того, тест обнаружил, что сервис Nginx не запущен, как сообщают `systemd` (сервис по умолчанию для роботов с юнит-ми). Если запустить Nginx вручную и выполнить тест еще раз, он снова должен пройти успешно:

```
(validate) $ systemctl start nginx
(validate) $ pytest -q --hosts='ssh://node4' test_webserver.py
..
2 passed in 2.38 seconds
```

Как упоминалось в начале раздела, веб-сервер должен выдать статическую содержимую страницу на порте 80. Следующий шаг — добавление еще одного теста (в тот же файл `test_webserver.py`) для проверки порта:

```
def test_nginx_listens_on_port_80(host):
    assert host.socket("tcp://0.0.0.0:80").is_listening
```

Этот тест несколько сложнее, так что имеет смысл обратить внимание на некоторые его нюансы. По умолчанию он проверяет TCP-соединения на порте 80 для *всех IP-адресов данного сервера*. Хотя для текущего тест-кейса необходимости нет, но, если у сервера есть несколько интерфейсов и он настроен привязку к конкретному адресу, значит, нужно добавить еще один тест. Добавление еще одного теста для проверки прослушивания на порте 80 на конкретном IP-адресе может пойти так перебором, но если задуматься о выводимом тестом отчете, становится ясно, что из него будет понятнее, что происходит.

1. Тест, проверяющий, что `nginx` прослушивает на порте 80: **ПРОЙДЕН**.
2. Тест, проверяющий, что `nginx` прослушивает по адресу `192.168.0.2` на порте 80: *не пройден*.

Из этого видно, что Nginx привязывается к порту 80, просто *не к тому интерфейсу, который нужен*. Дополнительный тест в данном случае — прекрасный способ улучшить детализацию за счет увеличения объема выводимой информации.

Здесь пустим только что добавленный тест:

```
(validate) $ pytest -q --hosts='ssh://node4' test_webserver.py
..F
===== FAILURES =====
_____ test_nginx_listens_on_port_80[ssh://node4] _____

host = <testinfra.host.Host object at 0x7fbaa64f26a0>

    def test_nginx_listens_on_port_80(host):
>     assert host.socket("tcp://0.0.0.0:80").is_listening
E     AssertionError: assert False
E     + where False = <socket tcp://0.0.0.0:80>.is_listening
E     + where <socket tcp://0.0.0.0:80> = <class 'LinuxSocketSS'>

test_webserver.py:11: AssertionError
1 failed, 2 passed in 2.98 seconds
```

Ни на одном IP-адресе не производилось прослушивание на порте 80. Из конфигурации Nginx становится ясно, что он был настроен на прослушивание на порте 8080 с помощью инструкции настройки порта в файле по умолчанию:

```
(validate) $ grep "listen 8080" /etc/nginx/sites-available/default
listen 8080 default_server;
```

После изменения на порт 80 и перезапуска сервиса nginx тест снова проходит успешно:

```
(validate) $ grep "listen 80" /etc/nginx/sites-available/default
listen 80 default_server;
(validate) $ systemctl restart nginx
(validate) $ pytest -q --hosts='ssh://node4' test_webserver.py
...
3 passed in 2.92 seconds
```

А поскольку встроенной фикстуры для обрботки HTTP-запросов к конкретному адресу не существует, последний тест извлекает контент с пущенного веб-сайта с помощью утилиты `wget` и выполняет операции контроля результатов, чтобы убедиться в правильной визуализации статического сайта:

```
def test_get_content_from_site(host):
    output = host.check_output('wget -qO- 0.0.0.0:80')
    assert 'Welcome to nginx' in output
```

Выполняем `test_webserver.py` еще раз и убеждаемся, что все наши предположения верны:

```
(validate) $ pytest -q --hosts='ssh://node4' test_webserver.py
....
4 passed in 3.29 seconds
```

Поняв идеи тестирования в Python и переориентировав их для проверки системы, можно добиться очень многого. Автоматизация запуск тестов в ходе разработки приложений или даже написания и выполнения тестов для существующей инфраструктуры — два прекрасных способа упростить рутинные операции, которые могут вызвать ошибки. `pytest` и `Testinfra` — отличные проекты, не только использующие их очень просто, к тому же при необходимости они легко расширяются. Тестирование — способ по-настоящему прокчать свои навыки.

Тестирование блокнотов Jupyter с помощью pytest

Один из простейших способов создать для своей компании большие проблемы — забыть о рекомендуемых практиках проектирования ПО применительно к новым одноклассным и машинному обучению. Для исправления ситуации можно воспользоваться плагином `nbval` для `pytest`, с помощью которого тестировать блокноты Jupyter. Взгляните на следующий Makefile:

```
setup:
    python3 -m venv ~/.myrepo

install:
    pip install -r requirements.txt

test:
    python -m pytest -vv --cov=myrepolib tests/*.py
    python -m pytest --nbval notebook.ipynb

lint:
    pylint --disable=R,C myrepolib cli web

all: install lint test
```

Ключевой элемент здесь — флаг `--nbval`, благодаря которому сервер сборки может протестировать блокнот из репозитория.

Вопросы и упражнения

- Назовите по крайней мере три соглашения о наименовании, соблюдение которых необходимо, чтобы `pytest` мог обнаружить тест.
- Для чего предназначен файл `conftest.py`?
- Поясните, что такое параметризация тестов.
- Что такое фикстуры и как их можно использовать в тестах? Удобны ли они? Почему?
- Расскажите, как можно применять фикстуру `monkeypatch`?

Задача на ситуационный анализ

Создайте тестовый модуль, использующий `testinfra` для подключения к удаленному серверу. С помощью тестов определите, что `Nginx` установлен, запущен с помощью `systemd` и сервер привязан к порту 80. Когда все тесты будут пройдены успешно, попробуйте добиться непрохождения одного из них, настроив `Nginx` и прослушивание на другом порте.

ГЛАВА 9

Облачные вычисления

Облачные вычисления — термин, порой вызывающий путаницу, подобно другим современным модным словечкам, таким как «большие данные», ИИ и Agile. А когда термин становится популярным, то неизбежно для различных людей он приобретает разное значение. Вот точное определение. Облако представляет собой вид поставки вычислительных сервисов по требованию с оплатой за использованное количество ресурсов, подобной любым коммунальным платежам: за природный газ, электричество или воду.

Основные достоинства облачных вычислений: стоимость, скорость, глобальный охват, отказоустойчивость, производительность, надежность и безопасность. Рассмотрим каждое.

- **Стоимость.** Непредвиденных расходов, ресурсы можно отмерять точно, в зависимости от потребностей.
- **Скорость.** Облако предоставляет возможность с момента подписания соглашения, так что опытный пользователь может полноценно задействовать их ресурсы для быстрого создания программных решений.
- **Глобальный охват.** Все крупные поставщики облачных сервисов оперируют на глобальном уровне, значит, могут предоставлять сервисы по всему миру для удовлетворения спроса в конкретных географических областях.
- **Отказоустойчивость.** Многие из них, например размещение серверов в стойках, не строят сетевого программного обеспечения и физическая охрана центров обработки данных, становятся неактуальными. Компании могут сосредоточиться на создании продуктов интеллектуальной собственности, вместо того чтобы изобретать велосипед.
- **Производительность.** В отличие от всего программного обеспечения, программное обеспечение облаков непрерывно обновляется, значит, по требованию пользователям доступно новейшее и самое быстрое программное

обеспечение. Вдобавок его связывающая инфраструктура с высокой пропускной способностью и низким значением задержки, создавая тем самым идеальную высокопроизводительную среду.

- **Ндежность.** В основе архитектуры облаков лежит избыточность на каждом шаге, в частности множество регионов и множество центров обработки данных в каждом регионе. Нитивная облачная архитектура может строиться вокруг этих возможностей, что обеспечивая ей высокую доступность. Кроме того, многие базовые облачные сервисы сами по себе высокодоступны, например Amazon S3, отличающийся надежностью в девять девяток, то есть 99,999999999 %.
- **Безопасность.** Общая безопасность определяется самым слабым звеном. За счет перехода к централизованной безопасности ее уровень повышается. Решение проблем наподобие физического доступа к центру обработки данных или шифрования хранимых данных с первых же дней существования отрасли.

Основы облачных вычислений

В некотором смысле сложно представить себе DevOps без облаков. Amazon перечисляют следующие рекомендуемые практики DevOps: непрерывная интеграция, непрерывная поставка, микросервисы, инфраструктура как код, мониторинг и журналирование, а также связь и сотрудничество. Можно даже утверждать, что все они зависят от существования облаков. Даже практики «связь и сотрудничество», которой так сложно дать точное определение, возможны благодаря набору современных инструментов связи SaaS: Jira, Trello, Slack, GitHub и др. Где работают все эти инструменты связи SaaS? В облаке.

Чем уникальны современные облачные вычисления? Как минимум тремя отличительными чертами: теоретически неограниченными вычислительными ресурсами, доступом к вычислительным ресурсам по запросу и тем, что не нужны никакие предпринятые вложения капиталов. В этих характеристиках скрыто отразится распределение Прето-навыков DevOps.

На практике облака становятся особенно экономически выгодными, если учитывать их реальные сильные стороны. Одно из использований облака может оказаться очень затратным для неопытных организаций, поскольку они не используют в достаточной степени основные возможности облака. Можно с полным основанием сказать, что на первых порах существования облаков 80 % их валового дохода обеспечивали неопытные пользователи, которые охватывали виртуальные вычислительные узлы в режиме ожидания, выбирали

не те виртуальные узлы (слишком большие), не предусматривали при проектировании системы возможность масштабирования или не применяли естественные для облачных программных архитектур, и, например, отправляли все в реляционную базу данных. Соответственно, оставшиеся 20 % приходилось нелегально бережливые организации с исключительными навыками DevOps.

До появления облачных затрат были фиксированными как в смысле денег, так и в смысле времени затрат боттиков. Центр обработки данных должен был обслуживать группу работающих на полную ставку высокооплачиваемых специалистов. По мере эволюции облачных в центре обработки данных оставались только лучшие из лучших, причем они работали на чрезвычайно искушенные организации, и пример Google, Microsoft и Amazon. Мелкая организация чисто статистически не могла позволить себе иметь в центре обработки данных специалиста по обеспечению того уровня, по крайней мере в течение длительного времени.

Принцип сравнения преимуществ — фундаментальный закон экономики. Вместо того чтобы, глядя на стоимость облачных сервисов, думать о том, как сэкономить, рассуждая то же самое самостоятельно, задумайтесь про *упущенную выгоду*, ведь вы могли бы потратить это время на что-то другое. Большинство организаций пришло к следующим выводам.

1. Соревноваться с опытом Google, Amazon и Microsoft в сфере центров обработки данных они не могут.
2. Средства, потраченные на облачные сервисы, позволяют компаниям сосредоточиться на других сферах, где они могут использовать свои уникальные знания.

Netflix решил сконцентрироваться на потоковом онлайн-вещании и создании оригинального контента, а не на поддержке собственных центров обработки данных. И глядя на динамику цен акций Netflix за 11 лет (рис. 9.1), сложно сомневаться в успехе подобной стратегии.

Одновременно уникальность Netflix — в их стремлении к высокоэффективному использованию возможностей облачных сервисов. Нынешние и бывшие сотрудники Netflix сделали множество докладов на конференциях, работали и выпустили немало утилит на GitHub, так же написали огромное количество статей и книг по тематике DevOps и облачных вычислений. Это дополнительное доказательство того факта, что недостаточно осознать необходимость применения облачных сервисов, нужно еще подкрепить это решение профессионализмом в данной сфере. В противном случае организация рискует оказаться в роли посетителя фитнес-клуба, купившего годовой абонемент, но отходившего только три недели. Те, кто купил абонемент, но не ходит, фактически субсидируют тех, кто ходит регулярно.

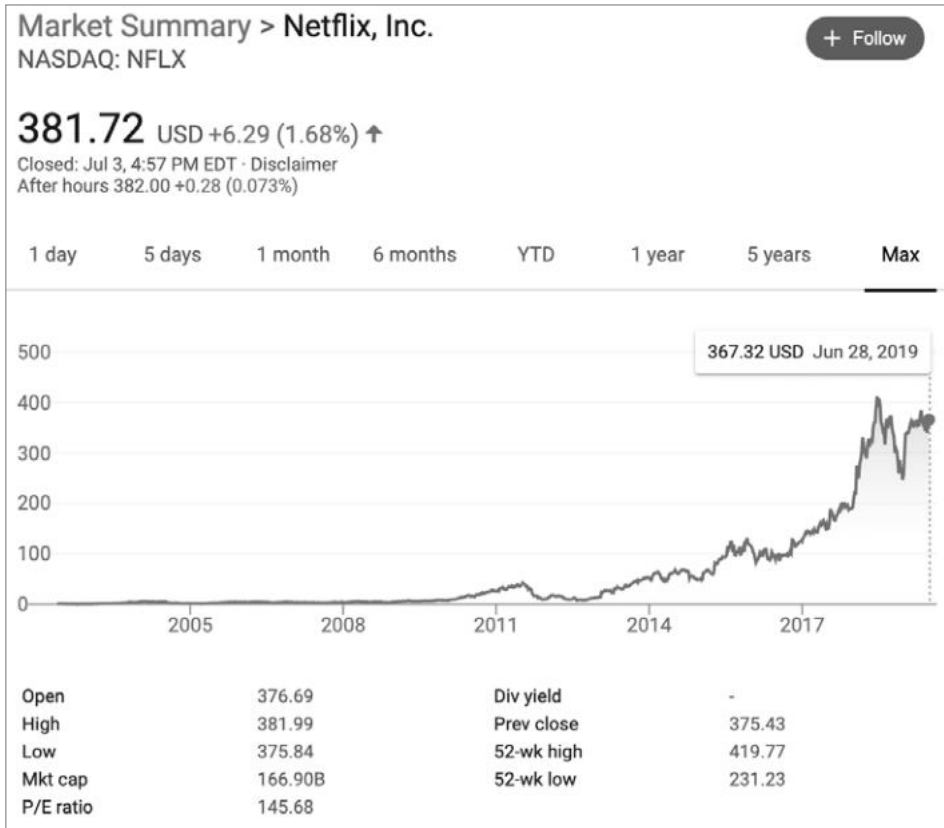


Рис. 9.1. Динамика цен акций Netflix за 11 лет

Типы облачных вычислений

Существует несколько основных типов облачных вычислений: общедоступное облако, частное облако и мультиоблако. В большинстве случаев, когда говорят об облаках, имеют в виду общедоступные облака. Однако это не единственный тип облаков. Частное облако — оно используется только одной организацией — физически располагается в центре обработки данных этой организации, или место под него предоставляется ей другой компанией. Поставщики частных облаков — это, к примеру, HPE, VMware, Dell и Oracle. Популярный вариант частного облака с открытым исходным кодом — OpenStack. Прекрасный пример его применения на практике — один из крупнейших поставщиков частных облаков OpenStack-к-сервис — компания Rackspace, занимающая относительно небольшой сегмент рынка услуг хостинга.

Более гибкий вариант — гибридное облако. Оно сочетает в себе классическое, традиционное и общедоступное облако. Пример применения подобной архитектуры: общедоступное облако используется в ситуациях, требующих масштабируемости и дополнительных вычислительных мощностей, классическое — для повседневных операций. Еще один пример подразумевает применение специализированной аппаратной архитектуры, скажем фермы GPU для глубокого обучения в классическом облаке, в роли базовой архитектуры выступает связанное с ней общедоступное облако. Даже крупнейшие поставщики облачных сервисов начинают входить в этот сегмент рынка. Хороший пример — платформа Anthos (<https://cloud.google.com/anthos>) от Google. Эта платформа берет на себя весь труд по связыванию локального центра обработки данных с GCP, благодаря чему возможны такие технологические процессы, как выполнение кластеров Kubernetes и тут и там с плавным переходом между ними.

Наконец, мультиоблако — вариант, возможный частично благодаря современным технологиям DevOps, таким как контейнеры Docker и IaC-решения, например Terraform. Стратегия мультиоблака состоит в использовании нескольких облаков одновременно. Прекрасный пример этой стратегии — выполнение заданий в контейнерах нескольких облаков одновременно. Зачем может понадобиться подобное? В частности, чтобы можно было выполнять задания на спотовых инстансах AWS, когда расценки на них достаточно низки для доходности, и на GCP, когда AWS слишком дороги. Такие утилиты, как Terraform, позволяют обобщить облачные понятия, сводя их к привычному языку конфигурации, контейнеры позволяют переносить код и среду выполнения в любое место, где можно запустить контейнер.

Типы облачных сервисов

Существует пять основных типов облачных сервисов: инфраструктура как сервис (Infrastructure as a Service, IaaS), «железо» как сервис (Metal as a Service, MaaS), платформа как сервис (Platform as a Service, PaaS), бессерверная обработка данных и программное обеспечение как сервис (Software as a Service, SaaS). Все эти облачные сервисы работают на различных уровнях абстракции и имеют свои достоинства, так и недостатки. Рассмотрим каждый из них подробнее.

Инфраструктура как сервис

IaaS представляет собой категорию более низкого уровня, включающую возможность поминутной аренды виртуальных машин, доступ к хранилищу объектов, предоставление программно определяемых сетей (SDN) и программно опре-

деляемых хранилищ (SDS) и предложения пользователям цены за доступную виртуальную машину. Подобный уровень сервиса обычно связывают с AWS, особенно в первые годы (2006) его существования, когда Amazon запустил облачное хранилище S3, SQS (Simple Queue Service) и EC2 (виртуальные машины).

Преимуществом такого сервиса для организации с большим опытом в DevOps состоит в возможности достичь высокой производительности и надежности, выделив для этого лишь небольшую группу людей. Недосток — крутая кривая обучения, к тому же при неэффективном использовании он может оканчиваться весьма затратным в смысле затрат финансов, труда и человеко-часов. В области строительства Силиконовой долины в 2009–2019 годах этот сценарий проигрывался не AWS на практике во множестве компаний.

Одна из историй, из-за которых мы вспомнили это, произошла, когда Ной занимался программной инженерией в компании, поставившей утилиту для мониторинга и поиска. В первый месяц его работы там возникли две критические проблемы, касавшиеся облаков. Первая из них, возникшая в первую же неделю, заключалась в том, что системный биллинг SaaS неправильно строил систему хранения. Компания удляла деньги заказчикам, честно платящим деньги! Суть проблемы заключалась в том, что у них не было достаточной инфраструктуры DevOps для успешной работы с облаком: ни серверной сборки, ни тестирования, ни нестоящей изолированной среды разработки, ни нлиз кода — лишь ограниченные возможности в том, что касается развертывания программного обеспечения. Именно эти практики DevOps Ной и внедрил в качестве решения в своей работе с проблемами.



Рассказ тут не совсем фигур речи — один из разработчиков поджег офис, пытаясь пожарить бекон в гриль-тостере. Ноую покзлось, что пахнет дымом, так что он пошел на кухню и увидел, как языки пламени ползут по стенам и потолку. Он был так потрясен с моей иронией ситуации, что несколько секунд просто стоял столбом. Кстати, один из быстрее собранных сотрудников (его руководитель по выпуску программного продукта) схватил огнетушитель и погасил огонь.

Вскоре обнаружилась и вторая, более серьезная проблема с нашей облачной архитектурой. Все разработчики компании должны были быть на связи 24 часа в 7 дней в неделю (за исключением технического директора — учредителя компании, нередко писавшего код, на прямую или косвенно вызывавший перебои в обслуживании... больше об этом чуть позже). Однажды, когда Ной был на дежурстве, в 2 часа ночи его разбудил звонок на мобильный от генерального директора — учредителя компании, который сказал, что их взломали и системы SaaS больше не существуют. Нет ни веб-серверов, ни поисковых конечных точек,

ни к ких-либо других виртуальных машин, на которых работала платформа. Ной спросил, почему он не получил сообщение об этом на пейджер, и генеральный директор сказал, что системный мониторинг также уничтожен. Ной решил поехать на работу в 2 часа ночи и поработать над проблемой на месте.

По мере появления дополнительной информации проблем прояснилось. Генеральный директор-учредитель изначально строил учетную запись AWS, и все сообщения о перебоях в работе сервиса приходили на его электронную почту. В течение нескольких месяцев Amazon отправлял ему по электронной почте сообщения о планах прекратить использование виртуальных машин в этом регионе, Северной Вирджинии, и о том, что через несколько месяцев они будут удалены. Что же, в итоге этот день настал и посреди ночи все серверы компании прекратили свое существование.

Ной выяснил все это еще по дороге на работу, после чего сосредоточился на создании всего SaaS компании с нуля на основе исходного кода на GitHub. Именно в этот момент Ной начал осознавать ту же сложность AWS. Восстановление работоспособности SaaS, включая прием данных, обратную платформу и отображение инструментальных панелей, продолжилось с 2 ночи до 8 утра. Полное восстановление данных из резервных копий заняло еще 48 часов.

Одна из причин, по которым восстановление работы заняло так много времени, заключалось в том, что в середине процесса развертывания не ходил с ответвленной версией Puppet, созданной одним из бывших сотрудников, но не внесенной в систему контроля версий. К счастью, Ной смог найти копию этой версии Puppet около 6 утра на одной отключенной машине, пережившей «побоище». Если бы этой машины не было, компании, возможно, пришел бы конец. Чтобы полностью воссоздать компанию подобной сложности без основного каркаса инфраструктуры кода (IAC), понадобилось бы как минимум неделя.

Подобный болезненный, однако с относительно счастливым концом опыт не учил Ноя многому. Он осознал, как велико достоинство и недостатки облака: возможности его огромны, но кривая обучения сокрушительна даже для старшего венчурного финансиста в области Silicon-Фрэнсиско. Вернемся теперь к техническому директору, не дежурившему, однако вносившему код в промышленную версию (без использования серверной сборки или системы непрерывной интеграции). Вовсе не он был злодеем в этой истории. Вполне возможно, что если бы сам Ной был техническим директором — учредителем той компании на определенном этапе карьеры, то совершил бы те же самые ошибки.

Настоящая проблема заключалась в разрушительном воздействии власти. Иерархия не означает технически безошибочности. Очень легко упиваться властью и верить, что раз ты главный, то все твои действия всегда правильные. Когда Ной

упр влял комп нией, он соверш л н логичные ошибки. Гл вный вывод — пр в всегд технологический процесс, не отдельный человек. То, что не втом тизиров но, не р бот ет норм льно. И то, что не проходит втом тического тестирования для контроля к честв , тоже не р бот ет норм льно. Если р зверты в ние не является воспроизводимым, оно тоже не р бот ет норм льно.

Еще одн , последняя история об этой комп нии связ н с мониторингом. После двух первых кризисов симптомы были устр нены, но болезнь ост л сь. Технологические процессы в комп нии были неэффективными. Эту фунд мент льную проблему иллюстрирует следующ я история. В комп нии был доморощенн я систем мониторинг (опять же созд нн я учредителями комп нии в с мом н ч ле), в среднем генериров вш я оповещения к ждые 3–4 ч с 24 ч с в сутки.

А поскольку дежурили все инженеры, з исключением технического директор , большинство из них постоянно недосып ло, к ждую ночь получ я уведомления о сбоях системы. Для «испр вления» необходимо было перез пустить сервисы. Ной вызв лся дежурить месяц подряд, чтобы д ть возможность испр вить причину проблемы. В результ те непрерывных стр д ний из-з недосып он осозн л несколько вещей. Во-первых, оповещения системы мониторинг генериров лись ничуть не лучше, чем случ йным обр зом. Он мог с тем же успехом з менить всю систему мониторинг н следующий сцен рий:

```
from random import choices

hours = list(range(1,25))
status = ["Alert", "No Alert"]
for hour in hours:
    print(f"Hour: {hour} -- {choices(status)}")
```

```
X python random_alert.py
Hour: 1 -- ['No Alert']
Hour: 2 -- ['No Alert']
Hour: 3 -- ['Alert']
Hour: 4 -- ['No Alert']
Hour: 5 -- ['Alert']
Hour: 6 -- ['Alert']
Hour: 7 -- ['Alert']
Hour: 8 -- ['No Alert']
Hour: 9 -- ['Alert']
Hour: 10 -- ['Alert']
Hour: 11 -- ['No Alert']
Hour: 12 -- ['Alert']
Hour: 13 -- ['No Alert']
Hour: 14 -- ['No Alert']
Hour: 15 -- ['No Alert']
Hour: 16 -- ['Alert']
Hour: 17 -- ['Alert']
```

```

Hour: 18 -- ['Alert']
Hour: 19 -- ['Alert']
Hour: 20 -- ['No Alert']
Hour: 21 -- ['Alert']
Hour: 22 -- ['Alert']
Hour: 23 -- ['No Alert']
Hour: 24 -- ['Alert']

```

Когда он это понял, то углубился в данные и сформировал историческую статистику по дням для всех оповещений за последний год (отметим, что учитывались только требующие реагирования оповещения, которые будили разработчиков). Из рис. 9.2 видно, что оповещения не только были бессмысленными, но и их частота повышалась, что в ретроспективе выглядит смехотворным. Они превращали рекомендуемые практики разработки в своего рода «коро-культ» и, обрзано говоря, мешали развиваться ветвям и пыльной взлетно-посадочной полосе, заполненной соломенными самолетами.



Рис. 9.2. Оповещения в предоставляющей SaaS компании по дням

Из анализ данных стало выясняться еще более печальное картина: разработчики тратили по несколько *годов* своей жизни, подталкивая среди ночи от

сообщений пейджер . Все эти страдания и жертвы были напрасными и лишь подтвердили печальную истину: жизнь несправедлива . Несправедливость ситуации угнетала, но пришлось потратить немало усилий, чтобы убедить людей согласиться на отключение оповещений. Люди склонны продолжать поступать так же, как и всегда . Кроме того, психологически хотелось придать какой-то более глубокий смысл столь жестоким и длительным страданиям. По сути дела, это было ложное божество.

Ретроспективный анализ применения облака IaaS AWS для этой конкретной компании, по существу, описывает основные доводы в пользу DevOps.

1. Необходимы конвейер поставки и петля обратной связи: сборка, тестирование, выпуск, мониторинг и планирование.
2. Разработчики и эксплуатация не разобщены. Если технический директор пишет код, он тоже должен дежурить (страдания от многолетнего недосыпа служат неплохой обратной связью).
3. Статус в иерархии не должен быть важнее технологического процесса . Члены команды должны сотрудничать с упором на сферы обязанностей и ответственности вне зависимости от должности, звания или опыта .
4. Быстродействие — одно из базовых требований DevOps. Поэтому необходимы микросервисы и непрерывная поставка, ведь с их помощью команды разработчиков могут делить сервисы, за которые отвечают, и выпускать программное обеспечение быстрее.
5. Быстрая поставка — еще одно базовое требование DevOps, но необходимы еще и непрерывная интеграция, непрерывная поставка, так же эффективные мониторинг и журналирование, позволяющие предпринять конкретные действия.
6. Возможность упрощения инфраструктуры и процесса разработки в больших масштабах. Автоматизация и единообразие — непростые требования. Решение — организовать повторяемое и автоматизируемое упрощение среды разработки, тестирования и промышленной эксплуатации с помощью IaaS.

«Железо» как сервис

Маas дает возможность обращаться с физическими серверами как с виртуальными машинами. Управлять программным обеспечением можно столь же просто, как и кластер виртуальных машин. Название Маas было предложено компанией Canonical, владельцем которой Марк Шаттлворт (Mark Shuttleworth) описывал его как «облачную семантику» в мире чистого «железа». Термин Маas так же

может означать использование физического программного обеспечения от поставщика, который предоставляет его практически также, как виртуализированное программное обеспечение. Прекрасный пример подобной компании — SoftLayer, поставщик «чистого железа», недавно приобретенный IBM.

Полный контроль над программным обеспечением имеет определенную привлекательность для нишевых приложений. Прекрасный пример — базисных на основе GPU. Непрктически подобные сервисы может предоставлять и обычное общедоступное облако, так что для выяснения того, нужен ли в конкретной ситуации MaaS, может понадобиться полный серьезный анализ выгод и затрат.

Платформа как сервис

PaaS — комплексная среда разработки и развертывания, включающая все необходимые для создания облачных сервисов ресурсы. Среди примеров — Heroku и Google App Engine. PaaS отличается от IaaS наличием утилит разработки, утилит управления базисными и высокоуровневыми сервисами с интеграцией «затрагивание одним щелчком кнопки мыши». Примеры сервисов, которые можно консолидировать: сервис аутентификации, сервис базисных, сервис веб-приложения.

PaaS зачастую критикуют за значительно более высокую стоимость в долгосрочной перспективе по сравнению с IaaS, как уже обсуждалось, впрочем, это зависит от среды. Если организация не может реализовать поведение DevOps, то стоимость — вопрос спорный. В подобном случае лучше заплатить за более дорогой сервис, обеспечивающий больше вышеупомянутых возможностей. Упущенная выгода организации, которой придется изучать продвинутые возможности управления развернутой системой IaaS, может оказаться чрезмерной для краткого жизненного цикла стартапа. Тогда организации лучше делегировать их поставщику PaaS.

Бессерверная обработка данных

Бессерверная обработка данных — одна из новых категорий облачных вычислений, все еще находящаяся в стадии активной разработки. Основной потенциал бессерверной обработки данных заключается в возможности тратить больше времени на создание приложений и сервисов и меньше — на обдумывание вариантов их выполнения. У всех крупных облачных платформ есть бессерверные решения.

Стандартные блоки бессерверных решений — вычислительные узлы, они же функция как сервис (Function as a Service, FaaS). У AWS эту роль играет Lambda, у GCP — Cloud Functions, у Microsoft — функции Azure. Традиционно выполнение этих облачных функций отделяется от среды выполнения, то есть

Python 2.7, Python 3.6 или Python 3.7. Все эти платформы поддерживают среды выполнения Python и в некоторых случаях — возможности пользовательской настройки базовой среды выполнения посредством стривемого под свои нужды контейнер Docker. Далее приведен пример простой лямбд-функции AWS, предназначенной для получения главной страницы «Википедии».

Следует отметить несколько особенностей этой лямбд-функции. Собственно, ее логик заключается в `lambda_handler`, принимающей два аргумента. Первый аргумент, `event`, передается вызывающей стороной, которая может быть чем угодно, от таймера событий Amazon Cloud Watch до запуска ее сформированным в AWS Lambda Console содержимым. Второй аргумент, `context`, содержит методы и свойства, обеспечивающие информацию о вызове, функции и среде выполнения:

```
import json
import wikipedia

print('Loading function')

def lambda_handler(event, context):
    """Средство реферирования Википедии"""

    entity = event["entity"]
    res = wikipedia.summary(entity, sentences=1)
    print(f"Response from wikipedia API: {res}")
    response = {
        "statusCode": "200",
        "headers": { "Content-type": "application/json" },
        "body": json.dumps({"message": res})
    }
    return response
```

Для использования этой лямбд-функции мы передаем следующее JSON-содержимое:

```
{"entity": "google"}
```

Выдаваемые этой лямбд-функцией результаты также представляют собой JSON-содержимое:

```
Response
{
  "statusCode": "200",
  "headers": {
    "Content-type": "application/json"
  },
  "body": "{\"message\": \"Google LLC is an American multinational technology\"}"
}
```

Один из самых впечатляющих спектров FaaS — возможность написания кода, реагирующего на события, вместо запускающего непрерывно кода, то есть приложений Ruby on Rails.

FaaS — облачно-ориентированный функциональный вариант возможности, по-настоящему скрытая одна из самых сильных сторон облаков — способность к быстрой дилатации (elasticity). Кроме того, средние затраты для написания лямбд-функций существенно снились.

Cloud9 в AWS — браузерный средние затраты, глубоко интегрированный с AWS (рис. 9.3).

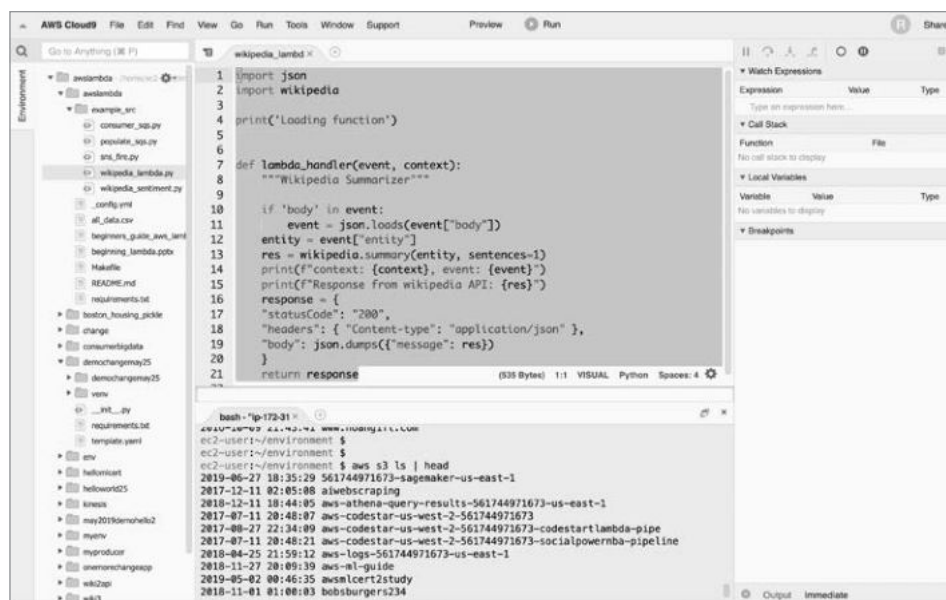


Рис. 9.3. Использование AWS Cloud9

Cloud9 ныне моя любимая среда написания лямбд-функций AWS и выполнения кода, требующего ключей API AWS. В Cloud9 встроены утилиты для создания лямбд-функций AWS, упрощающие их сборку и тестирование на локальной машине, так же как и развертывание в AWS.

На рис. 9.4 показан перед JSON-содержимого и локальное тестирование лямбд-функции в Cloud9. Тестирование функций подобным образом — одно из существенных усовершенствований платформы Cloud9.

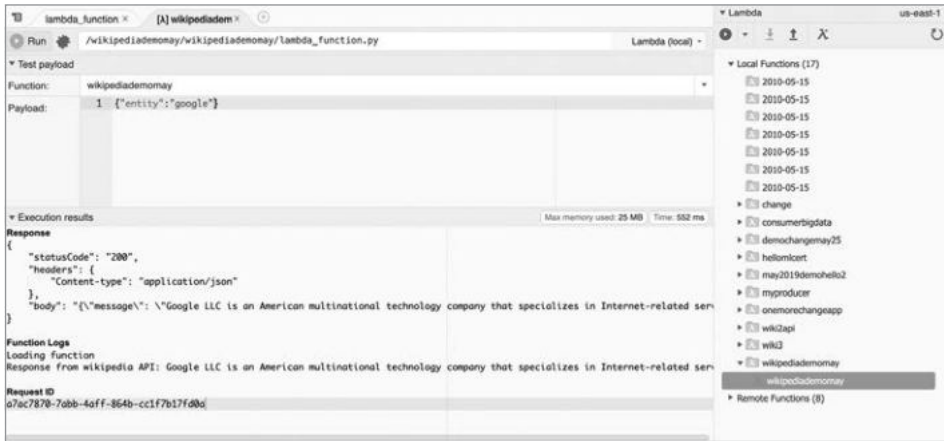


Рис. 9.4. Выполнение лямбда-функций в Cloud9

Ан логично Google Cloud предост вляет среду GCP Cloud Shell (рис. 9.5). Cloud Shell т же позволяет быстро приступить к р зр ботке, получив доступ к в ж-нейшим утилит м ком ндной строки и полной среде р зр ботки.

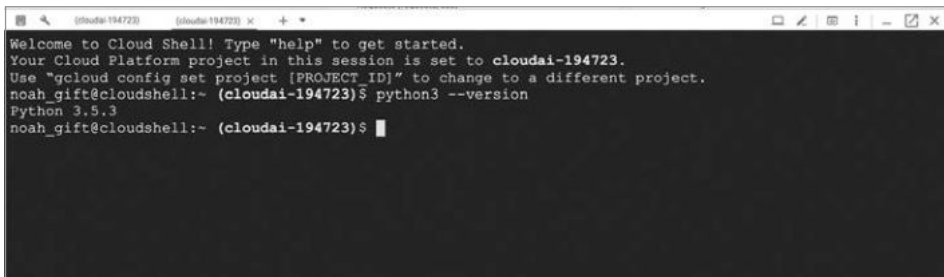


Рис. 9.5. GCP Cloud Shell

Ред ктор GCP Cloud Shell (рис. 9.6) предст вляет собой полнофункциональную IDE с подсветкой синт ксис , менеджером ф йлов и множеством других утилит, обычно присутствующих в тр диционных IDE.

Гл вный вывод: для обл ков по возможности стоит использо вать н тивные инструменты р зр ботки. Это сокр щ ет бреши в системе безоп нности, огр -ничив ет степень снижения скорости при перед че д нных с в шего ноутбук в обл ко, т же повыш ет производительность бл год ря тесной интегр ции с н тивной для них средой.

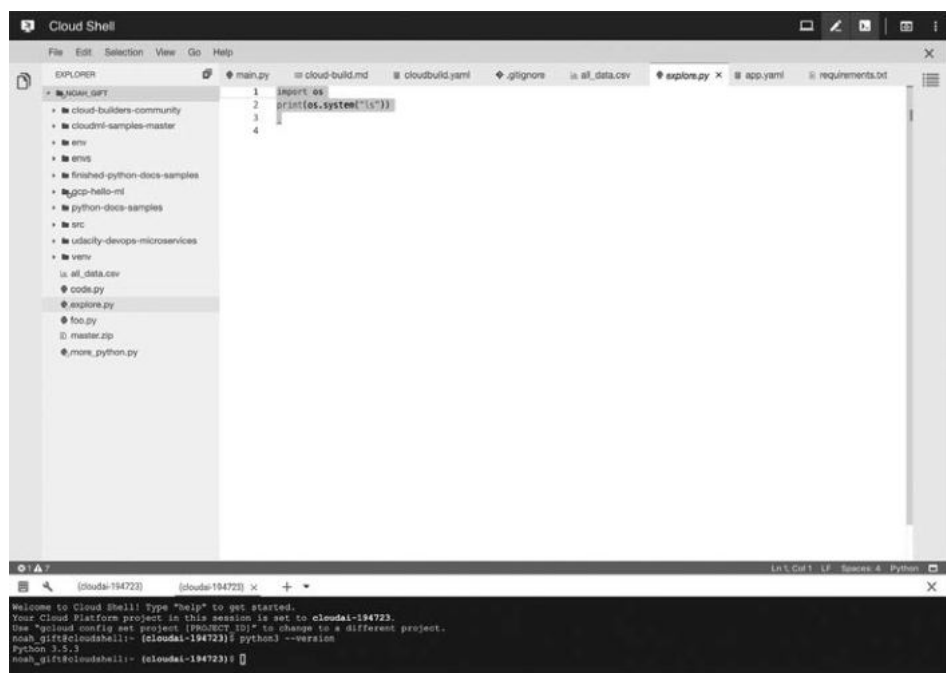


Рис. 9.6. Редактор GCP Cloud Shell

Программное обеспечение как сервис

SaaS и обл чные сервисы тесно связ ны с с мого н ч л . По мере рост функ-
цион льных возможностей обл ков продукты SaaS продолж ют н р шив ть
р зличные новшеств по верх новшеств с мих обл ков. Преимущес тв у про-
дуктов SaaS очень много, особенно в сфере DevOps. Н пример, з чем созд в ть
прогр ммное решение для мониторинг с мому, особенно н н ч льном эт пе,
если можно его просто *ренд*ов ть? Кроме того, многие б зовые принципы
DevOps, допустим непрерывн я интегр ция и непрерывн я пост вк , ср зу
доступны в приложениях SaaS от пост вщиков обл чных сервисов, н пример
AWS CodePipeline, или сторонних решениях SaaS, т ких к к CircleCI.

Во многих случ ях возможность сочет ния IaaS, PaaS и SaaS позволяет совре-
менным комп ниям р зр б тив ть прогр ммные продукты н много н дежнее
и эффективнее, чем десять лет н з д. С к ждым годом р зр ботк прогр ммного
обеспечения все больше упрощ ется бл год ря быстрой эволюции не только
обл ков, но и предост вляющих SaaS комп ний, созд ющих решения — н д-
стройки н д обл чными сервис ми.

Инфраструктура как код

Мы рассмотрим IaC намного подробнее в главе 10, где вы найдете более детальный ее анализ. Что же касается облаков и DevOps, то IaC — неотъемлемый спектр настоящих облачных вычислений. Для реализации практик DevOps в облаке IaC жизненно необходим.

Непрерывная поставка

Непрерывная поставка — более новый термин, который можно легко перепутать с непрерывной интеграцией и непрерывным развертыванием. Главное отличие — программное обеспечение *поставляется* в определенную среду, например среду предэксплуатационного тестирования, где может тестироваться как автоматически, так и вручную. И хотя оно не обязательно требует немедленного развертывания, но не ходит в подходящем для развертывания состоянии. Более подробные пояснения по поводу систем сборки можно найти в главе 15, но стоит также отметить, что для корректной работы с облачными сервисами это обязательное требование.

Виртуализация и контейнеры

Виртуализация — не менее фундаментальный компонент любого облака. Когда в 2006 году AWS был официально запущен, одним из базовых сервисов был Amazon Elastic Compute Cloud (EC2). Обсудим несколько основных сфер виртуализации.

Аппаратная виртуализация

Первой быстрой виртуализацией, выпущенной AWS, была паравиртуализация. Паравиртуализация бывает двух видов: паравиртуализация (paravirtualization) и аппаратные виртуальные машины (hardware virtual machine, HVM). HVM отличается более высокой производительностью. Основное отличие с точки зрения производительности состоит в том, что HVM может применять аппаратные расширения, использующие возможности аппаратного обеспечения хост-компьютера, что фактически делает виртуальную машину полноценной частью аппаратного обеспечения хоста, а не просто гостевой машиной, которой такое понятие не имеет, что делает основную.

Аппаратная виртуализация дает возможность запустить несколько операционных систем на одном хост-компьютере, тем же распределить между гостевыми операционными системами ресурсы CPU, ввод/вывод (как сетевого, так и дискового) и оперативной памяти. У этого подхода есть немало преимуществ, именно он лежит в основе современных облаков, но стоит отметить и недостатки перед самим Python. Во-первых, уровень детализации зачастую слишком велик для того, чтобы Python мог полноценно использовать среду. Из-за ограничений Python и потоков выполнения (они не могут работать на нескольких ядрах) виртуальная машина с двумя ядрами будет впустую расходовать ресурсы одного. В случае аппаратной виртуализации и языка Python из-за отсутствия подлинной многопоточности неопределенно количество ресурсов будет колоссальной. В конфигурации виртуальной машины для приложений Python одно или несколько ядер нередко остаются без нагрузки, что приводит к неэффективному тратам денег и энергии. К счастью, в облаках появились новые решения, позволяющие устроить эти недостатки языка Python. В частности, избавиться от данной проблемы можно с помощью контейнеров и бессерверной обработки данных, поскольку облако при этом рассматривается как операционная система и вместо потоков выполнения используются лямбда-функции или контейнеры. Вместо прослушивающих очереди потоков выполнения события регистрируют лямбда-функции из очереди облака, например SQS.

Программно определяемые сети

Программно определяемые сети (SDN) — существенная составляющая облачных вычислений. «Фишка» SDN заключается в возможности динамически программно менять поведение сети. Ранее это отвечало сетевой гуре, державший балансировщик нагрузки в железных рамках. Но однажды робот в крупной компании — оператор услуг связи, где каждый день проводил собрание под названием «Управление изменениями» один из роботов — назовем его Бобом, — контролировавший все выпускаемые программные продукты.

Необходимо иметь весьма специфические черты характера, чтобы быть таким Бобом. Споры Боба с другими сотрудниками в классической борьбе IT-специалистов по эксплуатации серверов ботаниками часто доходили до криков, и Боб наслаждался возможностью отказать в чем-то. Облака и DevOps полностью устроят такую роль, программное обеспечение и ежедневные соревнования по перекрикиванию. Соглашения сборки и жертвы программного обеспечения с использованием четко заданных настроек, программного обеспечения и данных, необходимых для среды промышленной эксплуатации, происходят в процессе непрерывной поставки. Роль Боба превратилась в нули и единицы где-то глубоко внутри матрицы, замененная кодом Terraform.

Программно определяемое хранилище

Программно определяемое хранилище (SDS) — технология, позволяющая использовать хранилище по запросу с мелко粒度 (granularity) операциями дискового и сетевого ввода/вывода. Хороший пример — том EBS Amazon, в которых можно настроить объемы предоставляемых операций дискового ввода/вывода. Обычно облачные SDS принимают объемы дискового ввода/вывода в том, что по мере роста размеров тома. Прекрасный пример того, как это все работает на практике, — Amazon Elastic File System (EFS). Он в том, что увеличивает объем дискового ввода/вывода по мере роста размеров хранилища и рассчитывает запросы от тысяч инстансов EC2 одновременно. А благодаря тесной интеграции с инстансами Amazon EC2 ожидающие выполнения операции записи могут буферизоваться и выполняться синхронно.

Свой опыт работы с EFS Ной получил в следующей ситуации. Пока еще не было AWS Batch, он спроектировал и написал систему, использующую тысячи спотовых инстансов, монтировавших том EFS, где они выполняли распределенные задания машинного зрения, полученные из Amazon SQS. Большим преимуществом для распределенных вычислений является возможность действовать всегда доступную распределенную файловую систему, это упрощает все операции, от звертывания до кластерных вычислений.

Контейнеры

Контейнеры существуют уже многие десятилетия, предоставляя виртуализацию уровня операционной системы. В пользовательском пространстве могут существовать изолированные экземпляры ядер. В начале 2000-х наблюдался стоящий бум компаний, предоставлявших услуги хостинга и применявших виртуальный хостинг веб-сайтов под управлением Apache в форме виртуализации уровня операционной системы. Нейнфреймх и в обычных операционных системах Unix, таких как AIX, HP-UX и Solaris, также давно существовали различные мыслимые контейнеры. Как раз тогда, когда Ной работал с технологией Solaris LDOM, когда он только появился в 2007 году, и был просто потрясен возможностью устаревшей полноценной операционной системы с детальной управлением CPU, операционной памятью и операциями ввода/вывода через telnet на машину, с помощью сетевой карты с двумя интерфейсами: основным и служебным.

Современные версии контейнеров непрерывно развиваются, заимствуя лучшие черты из эпохи нейнфреймов и сочетая их с новейшими идеями, такими как управление исходными кодами. В частности, одно из коренных изменений контейнеров заключалось в работе с ними как с проектами, извлекаемыми из

системы контроля версий. Стандартный формат для контейнеров сейчас — контейнеры Docker, их поддерживают все основные поставщики облачных сервисов наряду с программным обеспечением Kubernetes для управления контейнерами. Больше информации о контейнерах вы найдете в главе 12, но все основное, что связано с облаками, перечислено далее.

- **Реестр контейнеров.** У всех поставщиков облачных сервисов есть реестр, в котором хранятся пользовательские контейнеры.
- **Сервис управления Kubernetes.** У всех поставщиков облачных сервисов существует опция применения Kubernetes. В настоящее время это стандарт для управления развертываемыми контейнеризованными системами.
- **Формат Dockerfile.** Этот простой текстовый формат — стандартный для сборки контейнеров. В процессе сборки рекомендуется использовать различные линтеры, например hadolint (<https://oreil.ly/XboVE>), для исключения простейших программных ошибок.
- **Непрерывная интеграция с контейнерами.** У всех поставщиков облачных сервисов есть облачные системы сборки с возможностью интеграции с контейнерами. У Google — Cloud Build (<https://oreil.ly/xy6Ag>), у Amazon — AWS CodePipeline (<https://oreil.ly/I5bdH>), у Azure — Azure Pipelines (<https://oreil.ly/aEOx4>). Все они способны выполнять сборку контейнеров и вносить их в реестр контейнеров, так же собирать с помощью контейнеров различные проекты.
- **Тесная интеграция контейнеров во все облачные сервисы.** Работая с управляемыми сервисами в облачных платформах, можно быть уверенными по крайней мере в наличии у них одной общей черты — контейнеров! Контейнеры применяются в SageMaker — управляемой платформе машинного обучения Amazon. А в среде облачной разработки Google Cloud Shell контейнеры позволяют пользователям строить среду разработки по своему усмотрению.

Трудные задачи и потенциальные возможности распределенной обработки данных

Одним из наиболее перспективных сфер компьютерных наук — распределенная обработка данных. Нынешняя эпоха облачных вычислений характеризуется несколькими принципиальными изменениями, повлиявшими буквально на все. Одно из наиболее существенных изменений — расцвет многоядерных машин и конец закону Мура (рис. 9.7).

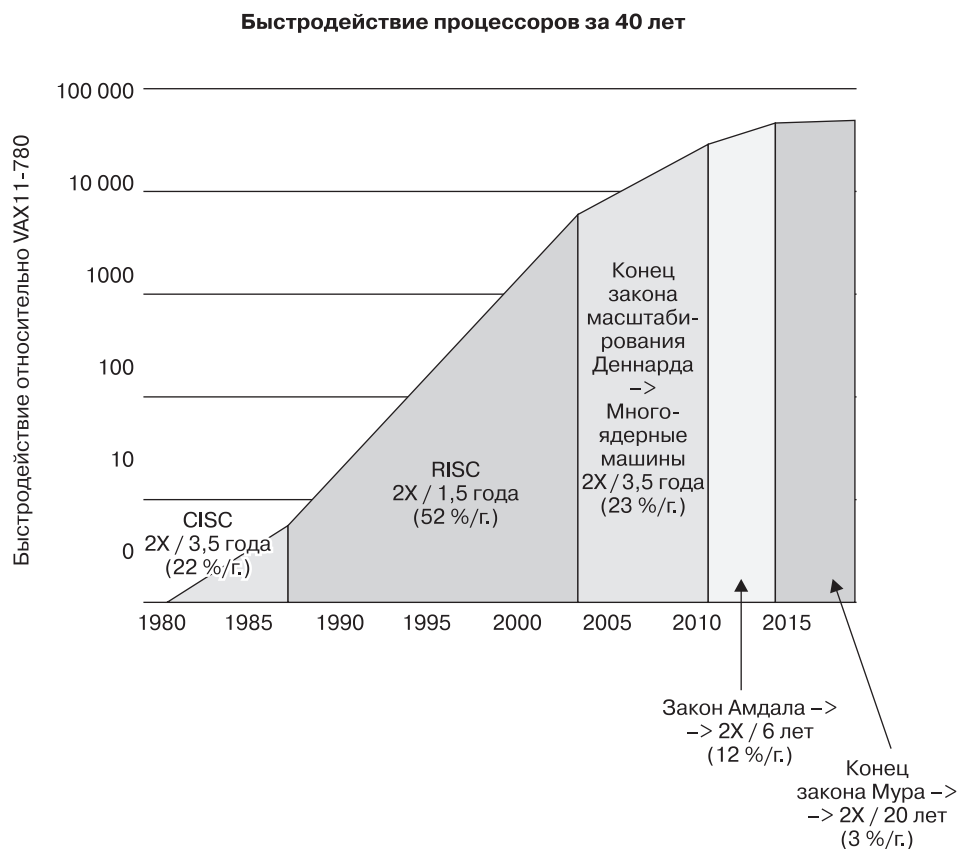


Рис. 9.7. Конец закона Мура (источник — John Hennessy and David Patterson, Computer Architecture: A Quantitative Approach, 6/e. 2018¹)

Закон Мур вскрыл две проблемы, проявившиеся в эпоху облачных вычислений. Первая: CPU предназначены для многоцелевого использования, а не для выполнения параллельных задач. Этот факт в сочетании с базовыми физическими ограничениями на рост скорости CPU приводит к снижению эффективности CPU в эпоху облачных вычислений. К 2015 году с законом Мур фактически было покончено, прирост скорости составил около 3 % в год.

Вторая: создание многоядерных машин в противовес ограничениям отдельным процессорам вызвало эффект домино в отношении языков программирования.

¹ Паттерсон Д., Хеннесси Д. Архитектура компьютеров и проектирование компьютерных систем. 4-е изд. — СПб.: Питер, 2012.

Множество языков программирования, созданных в эпоху, когда еще не было многоядерных процессоров, не говоря уже про Интернет, ранее испытывали серьезные трудности с использованием нескольких ядер процессоров. Нглядный пример — Python. Еще больше осложняет ситуацию то, что увеличение количества действующих ядер для неспресселированных по своей сути задач ничего не дает, как демонстрирует рис. 9.8.

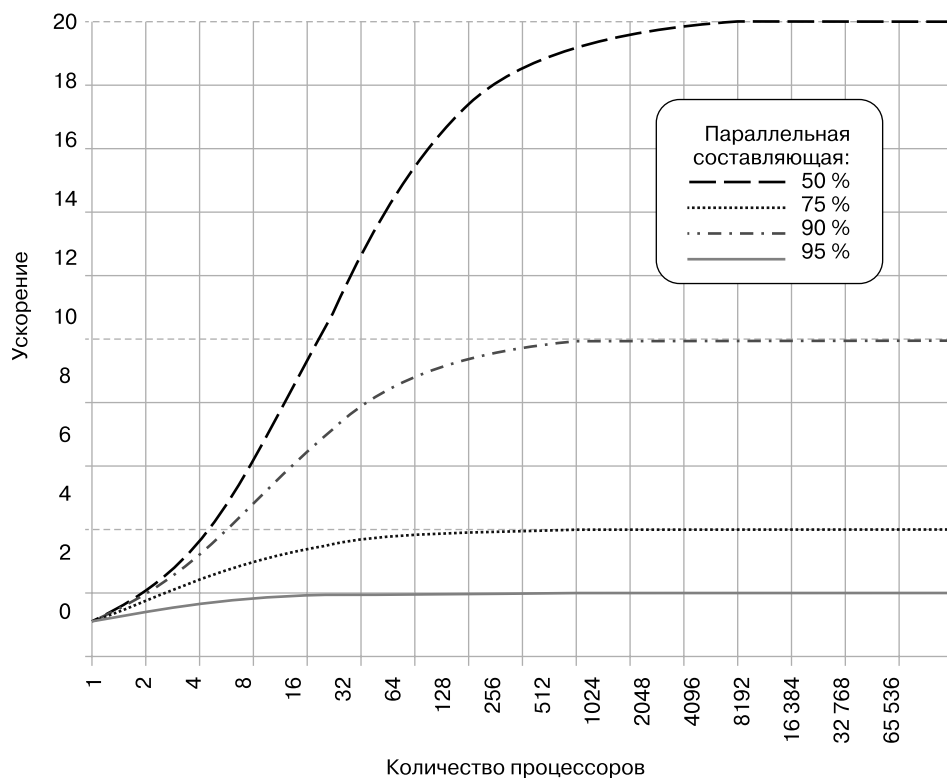


Рис. 9.8. Закон Амдала

Перспективы тут у облачных и рличных архитектур и подобие специализированных под конкретные приложения интегральных схем (application-specific integrated circuit, ASIC). В их число входят графические процессоры (GPU), программируемые пользователем вентильные матрицы (field-programmable gate array, FPGA), а также тензорные процессоры (TPU). Эти специализированные микросхемы все шире применяются для задач машинного обучения, прокладывая дорогу для облачных архитектур, сочетающих рличное параллельное обеспечение для решения сложных задач распределенных вычислений.

Конкурентное выполнение на Python, быстродействие и управление процессами в эпоху облачных вычислений

Представьте себя гуляющими поздно вечером по темной улице в опустошенном районе Сан-Франциско. Допустим, у вас черный пояс по бразильскому джиу-джитсу. Вы идете в одиночестве и мечтаете, что какой-то незнакомец, похоже, вас преследует. Он приближается, в ваше сердце нечестно бьется сильнее, вы вспоминаете свои годы тренировок по боевым искусствам. Придется ли вам драться с незнакомцем на улице? Вы проводите несколько активных спрингов в неделю в тренажерном зале и чувствуете, что готовы защитить себя, если понадобится. Вы также знаете, что бразильское джиу-джитсу — эффективное боевое искусство, прекрасно работающее в паре.

Однако драться всегда лучше избежать. Это очевидно. Незнакомец может быть вооружен. Вы можете победить, но нести противнику серьезные увечья. Или можете проиграть и сами получить серьезные увечья. Даже не стоящий матч бразильского джиу-джитсу знает, что драться на улице не лучший сценарий, несмотря на высокую вероятность победы.

Конкурентность в Python совершенно не логична. Хотя существуют удобные паттерны и подобию библиотек `multiprocessing` и `asyncio`, конкурентность не стоит использовать часто. Нередко можно вместо этого использовать возможности конкурентности платформ (бессерверную обработку данных, пакетную обработку, спотовые инстансы), не конкурентную обработку, реализуемую самостоятельно на каком-либо языке программирования.

Управление процессами

Управление процессами в Python — совершенно естественная возможность языка программирования. Python служит своеобразным связующим элементом для других платформ, языков и процессов, именно в этом он особенно силен. Кроме того, управление процессами с годами стало реализовываться совершенно по-другому и продолжает совершенствоваться.

Процессы и дочерние процессы

Простейший и наиболее эффективный способ запуска процессов с помощью стандартной библиотеки — функция `run()`. Достаточно, чтобы у вас был установлен

Python 3.7 или выше, д льше можно н ч ть с нее и упростить свой код. Простейший пример состоит всего из одной строки код :

```
out = subprocess.run(["ls", "-l"], capture_output=True)
```

Он способен выполнить практически все что угодно. Строк вызывает инструкцию командной оболочки в дочернем процессе Python и захватывает выводимые результаты. Возвращаемое значение — объект тип `CompletedProcess`. Для запуска процесса используются args: `returncode`, `stdout`, `stderr` и `check_returncode`.

Этот однострочный сценарий измещает и упрощает недостоаточно логичные и слишком сложные методы вызовов инструкций командной оболочки, что просто замечательно для разработчиков, чьи пишущих код на языке Python, смешанный с инструкциями командной оболочки. Вот еще несколько полезных советов.

Избегайте shell=True

С точки зрения безопасности лучше вызывать команды в виде элементов списка :

```
subprocess.run(["ls", "-la"])
```

Избегать использования строк следующего вида :

```
#Избегайте подобного
subprocess.run("ls -la", shell=True)
```

Причин этого очевидно. Если принимать на входе произвольную строку и выполнять ее, очень легко случайно разрушить безопасность программы. Допустим, вы написали простую программу, с помощью которой пользователь может выводить список файлов в каталоге. Он может воспользоваться вшей программой для выполнения произвольной команды. Случайно создайте брешь в защите очень опасно, надеюсь, это хорошо иллюстрирует, насколько плохой идеей будет использовать `shell=True`!

```
# Вводимые злоумышленником команды, приводящие к необратимой утере данных
user_input = 'some_dir && rm -rf /some/important/directory'
my_command = "ls -l " + user_input
subprocess.run(my_command, shell=True)
```

Можно полностью предотвратить подобное, за счет использования строк:

```
# Вводимые злоумышленником команды ни к чему плохому не приводят
user_input = 'some_dir && rm -rf /some/important/directory'
subprocess.run(["ls", "-l", user_input])
```

Задание времени ожидания и работа с ним

Принимая код, который запускает довольно долго выполняющиеся процессы, желательно уметь разумное время ожидания по умолчанию. Проще всего экспериментировать с этой возможностью с помощью команды `sleep` Unix. Вот пример команды `sleep`, выполнение которой завершится до истечения времени ожидания в командной оболочке IPython. Он возвращает объект `CompletedProcess`:

```
In [1]: subprocess.run(["sleep", "3"], timeout=4)
Out[1]: CompletedProcess(args=['sleep', '3'], returncode=0)
```

А это вторая версия, генерирующая исключение. В большинстве случаев лучше к ней будь это исключение обработать:

```
----> 1 subprocess.run(["sleep", "3"], timeout=1)
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/subprocess.py
  in run(input, capture_output, timeout, check, *popenargs, **kwargs)
    477         stdout, stderr = process.communicate()
    478         raise TimeoutExpired(process.args, timeout, output=stdout,
--> 479                               stderr=stderr)
    480     except: # Including KeyboardInterrupt, communicate handled that.
    481         process.kill()
TimeoutExpired: Command '['sleep', '3']' timed out after 1 seconds
```

Ниболее разумный подход — перехватить это исключение `TimeoutExpired`, после чего занести информацию о нем в журнал и релизовать код для освобождения памяти:

```
import logging
import subprocess

try:
    subprocess.run(["sleep", "3"], timeout=4)
except subprocess.TimeoutExpired:
    logging.exception("Sleep command timed out")
```

Журналирование исключений жизненно необходимо при создании систем профессионального уровня. Отследить ошибку при разбрасывании кода в множестве машин будет невозможно без централизованной системы журналирования с возможностью поиска. Профессионалы в сфере DevOps чрезвычайно следуют этому паттерну и призывают всем о его полезности.

Проблема с потоками выполнения Python

Возможно, в детстве у вас был приятель, с которым ваши родители не советовали водиться. Если так, видимо, они пытались помочь вам избежать плохих решений. Потоки выполнения Python во многом схожи с таким скверным другом. Если с ними связаться, ни к чему хорошему это не приведет.

Потоки выполнения в других языках программирования — вполне разумный компромисс. В таких языках, как C#, можно работать с пулами потоков выполнения, подключающихся к очередям, и рассчитывать, что каждый из порожденных процессов сможет полноценно использовать все ядра устройств. Этот неплохо себя показавший паттерн применения потоков выполнения смягчает недостатки от необходимости вручную управлять и убирать блокировки в коде.

О Python этого сказать нельзя. При порождении потоков выполнения он не задействует все ядра процессоров машины и зачастую ведет себя совершенно непредсказуемо, перескакивая с ядра на ядро и даже замедляя работу кода. Зачем использовать подобное, если есть другие варианты?

Если вы хотите больше узнать про DevOps, то верное, вы человек критичный и хотели бы получать и применять только рациональные и практичные знания. Соображения практичности — еще одна причина избегать потоков выполнения в Python. Теоретически в некоторых ситуациях можно добиться повышения быстродействия с помощью потоков выполнения, если проблем связанных с ограничениями ввод/вывод. Однако опять же какой смысл работать ненужным инструментом, если существуют нужные? Потоки выполнения Python подобны автомобилю, который все время нужно заводить «с толкача» или от чужого аккумулятора, выжимая сцепление, потому что его собственный клапаничит. Что будет, если в один прекрасный день чужого аккумулятора под рукой не окажется или нужно будет припарковаться на холме? Подобная стратегия — чистой воды безумие!

В этой главе вы не найдете примеров использования потоков выполнения. Зачем подводить плохой пример? Вместо этого стоит сосредоточиться на переносимых в этой главе альтернативных вариантах.

Решение задач с помощью библиотеки multiprocessing

Библиотека multiprocessing — единственный универсальный способ использования всех ядер машины с помощью стандартной библиотеки Python. Как видно на рис. 9.9, на уровне операционной системы есть два варианта: многопроцессная обработка и контейнеры.

Контейнеры как альтернатива мультипроцессной обработке отличаются от нее принципиально. Если библиотека multiprocessing используется для многократного вызова процесса без межпроцессного взаимодействия, то имеет смысл задействовать контейнер, виртуальную машину или какую-либо из альтернативных возможностей облака, например функцию как сервис. Один из популярных и эффективных вариантов альтернативных возможностей — AWS Lambda.

| Облако | | | |
|---------------------------|-----------|---------------------------|-----------|
| Операционная система | | Операционная система | |
| Многопроцессная обработка | Контейнер | Многопроцессная обработка | Контейнер |
| Ядра | | | |

Рис. 9.9. Параллельное выполнение кода на Python

Ан логично у контейнер т кже много преимуществ по ср внению с ветвле нием процессов вручную. Определения контейнеров предст вляют собой код. Можно выбир ть объемы используемых контейнер ми ресурсов в р зрезе опер тивной п мяти, CPU и дисковых опер ций ввод /вывод . Они предст вляют собой прямую льтерн тиву и з ч стую лучшую з мену ветвлению процессов вручную. Кроме того, н пр ктике они гор здо легче укл дыв ются в мировоз зрение DevOps.

С точки зрения DevOps, если вы осозн ли, что конкурентности в Python же лтельно избег ть при м лейшей возможности, д же сцен рии использо в ния модуля `multiprocessing` довольно огр ниченны. Возможно, лучше з действо в ть библиотеку `multiprocessing` только для р зр ботки и проведения экспе риментов, поскольку существуют зн чительно лучшие в ри нты н уровне к к контейнеров, т к и обл чных сервисов.

Можно сформулиров ть это и по-другому. Просто з дум йтесь, кому вы доверя ете созд в ть ветви процессов: н пис нному в ми н Python многопроцессному коду, Kubernetes, созд нному р зр ботчик ми Google, или AWS Lambda, созд н ному р зр ботчик ми Amazon? По н шему опыту, лучше всего стоять н плеч х гиг нтов. Р зобр вшись с этим философским вопросом, обсудим несколько способов эффективной мультипроцессной обр ботки.

Ветвление процессов с помощью `Pool()`

Простейший способ проверить возможности ветвления процессов и выполнения в них функции — р счет кл стериз ции методом k-средних с помощью библио теки м шинного обучения `sklearn`. Кл стериз ция методом k-средних требует большого объем вычислений и х р ктеризуется временной сложностью лго ритм $O(n^2)$, то есть объем вычислений р стет экспоненци льно быстрее при росте объемов д нных. Этот пример прекр сно подходит для р сп р ллевлив ния

к к н м кро-, т к и н микроуровне. В следующем примере¹ метод `make_blobs` созд ет н бор д нных, включ ющий 100 тыс. з писей и десять призн ков. При этом измеряется время выполнения к ждого лгоритм к-средних, т кже общее время:

```
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import KMeans
import time

def do_kmeans():
    """Кластеризация сгенерированных данных методом k-средних"""

    X, _ = make_blobs(n_samples=100000, centers=3, n_features=10,
                      random_state=0)
    kmeans = KMeans(n_clusters=3)
    t0 = time.time()
    kmeans.fit(X)
    print(f"KMeans cluster fit in {time.time()-t0}")

def main():
    """Выполняем все расчеты"""

    count = 10
    t0 = time.time()
    for _ in range(count):
        do_kmeans()
    print(f"Performed {count} KMeans in total time: {time.time()-t0}")
if __name__ == "__main__":
    main()
```

Время выполнения лгоритм метод к-средних демонстрирует, что это ресурсоемк я опер ция, — десять итер ций длятся 3,5 секунды:

```
(.python-devops) → python kmeans_sequential.py
KMeans cluster fit in 0.29854321479797363
KMeans cluster fit in 0.2869119644165039
KMeans cluster fit in 0.2811620235443115
KMeans cluster fit in 0.28687286376953125
KMeans cluster fit in 0.2845759391784668
KMeans cluster fit in 0.2866239547729492
KMeans cluster fit in 0.2843656539916992
KMeans cluster fit in 0.2885470390319824
KMeans cluster fit in 0.2878849506378174
KMeans cluster fit in 0.28443288803100586
Performed 10 KMeans in total time: 3.510640859603882
```

¹ В последних версиях библиотеки `sklearn` нет модуля `sklearn.datasets.samples_generator`, его з менил `sklearn.datasets`, т к что перв я строк с импортом должн выглядеть вот т к: `from sklearn.datasets import make_blobs`. — *Примеч. пер.*

Этот пример демонстрирует, почему важно профилировать код и не торопиться сразу с ним работать параллелив. При небольших размерах задачи параллельные сходки параллелив приведут лишь к замедлению работы кода, не говоря уже про усложнение его отладки.

С точки зрения DevOps всегда следует выбирать самый простой и удобный для сопровождения подход. Непривычно знать, что в мире мультипроцессного параллелив вполне подходящий, но сначала следует попробовать параллелив на микроуровне. Существуют и альтернативные микроподходы с использованием контейнеров, FaaS (AWS Lambda или другие бессерверные технологии) либо высокопроизводительный сервер для запуска процессов-исполнителей (RabbitMQ или Redis).

Функция как сервис и бессерверная обработка данных

Нынешняя эра ИИ ставит задачи, требующие новых парадигм. Рост тактовой частоты процессоров практически остановился, что, по существу, отменило закон Мур. В то же время эстафету подхватили резкий рост объемов данных, расцвет облачных вычислений и доступность специализированных интегральных схем (ASIC). Важную роль начинают играть функции как единицы выполняемой работы.

Бессерверная обработка данных и FaaS иногда используются как синонимы, описывая функции как единицы работы на облачных платформах.

Повышение производительности Python с помощью библиотеки Numba

Numba — замечательная библиотека для распределенного решения различных задач. Использование ее — все равно что совершенствоваться в вождении с помощью высокопроизводительных пистолетов. Кроме того, она позволяет полноценно использовать ASIC для решения конкретных задач.

Динамический компилятор Numba

Взглянем на пример использования динамического компилятора (Just in Time Compiler, JIT) Numba из официальной документации (<https://oreil.ly/KIW5s>), немного его модифицируем, затем поподробнее опишем, что происходит.

Этот пример представляет собой функцию на языке Python, декорированную динамическим компилятором. Благодаря аргументу `nopython=True` код

проходит через динамический компилятор и оптимизируется с помощью LLVM-компилятор. Если не использовать эту опцию, то все не оптимизированное в LLVM остается обычным кодом на языке Python:

```
import numpy as np
from numba import jit

@jit(nopython=True)
def go_fast(a):
    """На вход должен быть подан массив Numpy"""

    count = 0
    for i in range(a.shape[0]):
        count += np.tanh(a[i, i])
    return count + trace
```

Далее мы создадим массив `numpy` и измеряем время выполнения с помощью «динамической» функции `IPython`:

```
x = np.arange(100).reshape(10, 10)
%timeit go_fast(x)
```

Как видно из результатов, код выполняется 855 наносекунд:

```
The slowest run took 33.43 times longer than the fastest. This example could mean
that an intermediate result is cached. 1000000 loops, best of 3: 855 ns per loop
```

С помощью этой уловки можно запустить и обычную версию, чтобы не нужно было использовать декоратор:

```
%timeit go_fast.py_func(x)
```

Как видно из результатов, без динамического компилятора обычный код Python выполняется в 20 раз медленнее:

```
The slowest run took 4.15 times longer than the fastest. This result could mean
that an intermediate run is cached. 10000 loops, best of 3: 20.5 µs per loop
```

Динамический компилятор Numba умеет оптимизировать циклы `for`, ускоряя их выполнение, также функции и структуры данных `numpy`. Главным выводом из этого примера: возможно, имеет смысл проанализировать уже существующий код, работавший многие годы, и выяснить, не принесет ли пользу критическим компонентам инфраструктуры Python компиляция с помощью динамического компилятора Numba.

Высокопроизводительные серверы

Смелая идея — в ближайшем будущем замена человеческого развития. Простейшее определение смелости: достижение индивидуумом своего истинного потенциала. Для этого необходимо принять свою человеческую природу со

всеми ее недостатками. Согласно одной из оценок, полностью сформализуется менее 1 % людей.

Аналогичное понятие применимо и к языку программирования Python. Роботчик может полноценно использовать язык программирования, только если примет все его сильные и слабые стороны. Python — вовсе не высокопроизводительный язык. Python — отнюдь не язык, оптимизированный для написания серверов, как некоторые другие: Go, Java, C, C++, C# или Erlang. Зато Python позволяет добавлять высокоуровневую логику поверх высокопроизводительного кода, написанного на высокопроизводительном языке программирования или созданного в какой-либо форме.

Популярность Python во многом обусловлена тем, что он хорошо отражает естественный процесс мышления человека. При достаточном опыте использования Python вы сможете «думать» на нем, как думаете на родном языке. Логика можно выражать различными способами: языком, символьной нотацией, кодом, картинками, музыкой и живописью. Такие понятия теории вычислительной техники, как управление потоком, объявление типов, примитивы конкурентного выполнения и объектно-ориентированное проектирование, можно отделить от чистой логики. Они не обязательны для выражения идеи.

Мощь подобных Python языков заключается в том, что пользователь может работать на уровне логики, а не на уровне теории вычислительной техники. Всегда следует применять правильные инструменты, которыми зачастую бывает облако или сервис, написанный на другом языке программирования.

Резюме

И для DevOps, и для науки о данных характерно разделение людей по должностям и обязанностям. В числе преимуществ методологии DevOps, достигаемых ориентацией на прагматизм, — скорость, вмотивация, надежность, масштабируемость и безопасность. Эффективность операций повышается за счет применения решений микроуровня для конкурентной обработки процессов и управления ими. Не стоит использовать решения микроуровня, не исследовав сначала, какие фреймворки и решения доступны, — это основной типичный термин DevOps.

Какие же выводы вы можете сделать относительно применения Python в эпоху облачных вычислений?

- Изучите и освоите лучше всего подходящую для конкретной задачи методику конкурентной обработки.
- Изучите использование высокопроизводительных вычислительных библиотек, например Numba, для ускорения выполнения кода с помощью потоков выполнения, динамической компиляции и GPU.

- Н учитеcь применять FaaS для изящного решения нест нд ртных з д ч.
- Р ссм трив йте обл ко к к опер ционную систему, которой можно делегиров ть всю с мую трудную р боту по конкурентному выполнению з д ний.
- Примите н вооружение т кие естественные для обл ков конструкции, к к непрерывн я пост вк , контейнеры в форм те Docker и бессерверн я обр ботк д нных.

Вопросы

- Что т кое IaaS?
- Что т кое PaaS?
- Что т кое «способность к быстрой д пт ции»?
- Что озн ч ет термин «доступность»?
- К кие существуют типы обл чных вычислительных сервисов?
- Что т кое бессерверн я обр ботк д нных?
- В чем состоят гл вные р зличия между IaaS и PaaS?
- Что т кое з кон Амд л ?

Вопросы на ситуационный анализ

- Комп ния сомнев ется, переходить ли н обл чные сервисы, из-з слухов об их дороговизне. К кие существуют способы снижения рисков з тр т, связ нных с переходом н обл чные сервисы?
- Приведите пример н тивной обл чной рхитектуры. Н рисуйте структурную схему н тивной обл чной системы и перечислите ее в жнейшие возможности.
- Для чего предн зн чены спотовые и вытесняемые инст нсы? К ким обр - зом они позволяют экономить деньги? Для к ких з д ч они подходят и для к ких — нет?

Инфраструктура как код

До появления модных названий должностей и должностных обязанностей DevOps мы были просто скромными системными администраторами, или, коротко, сис админми. В те мрачные дооблачные времена нам приходилось загружать битамики своих машин серверами без ПО, ехать в центр колокации, чтобы разместить их в стойках, подключить к сети, подсоединить монитор, клавиатуру, мышью и настроить по одному. Григ до сих пор вздрагивает, вспоминая очасх, проведенных в центрах колокации под ослепительным светом лампы и ледяным потоком воздуха от кондиционеров. Сначала нам приходилось совершенствовать свое знание Bash, затем мы перешли на Perl, с мыслями везучие из нас — на Python. Как говорится, Интернет около 2004 года держался на скотче и жевательной резинке.

Примерно в 2006–2007 годах мы открыли для себя волшебный мир инстансов Amazon EC2. Выделение серверов теперь осуществлялось через простой интерфейс с помощью всего нескольких щелчков кнопкой мыши либо через утилиты командной строки. Больше никаких поездок в центры колокации, никаких укладок и подключения серверов. При необходимости можно было легко запустить десять инстансов EC2 за раз. Или даже 20! Или вообще 100! Никаких ограничений. Однако мы быстро поняли, что подход с подключением вручную к каждому инстансу EC2 по SSH и настройкой всех наших приложений на каждом инстансе по отдельности масштабируется плохо. Выделять инстансы самостоятельно окладилось не так уж трудно. Сложнее было установить все необходимые для наших приложений пакеты, добавлять нужных пользователей, задавать права доступа к файлам и, наконец, устанавливать сами приложения. Для решения этих задач и появилось ПО для автоматизации выделения инфраструктуры в виде утилит управления настройками. Первой широко известной такой утилитой стал Puppet, выпущенный в 2005 году, еще до появления Amazon EC2. Следом за ней появились и другие подобные утилиты: Chef в 2008-м, за ней SaltStack в 2011-м и Ansible в 2012-м.

К 2009 году мир был готов с рспростертыми объятиями встретить новое понятие — DevOps. До сих пор существует несколько альтернативных его определений. Самое интересное, что оно возникло в самом начале бурной эволюции ПО для автоматизации инфраструктуры. И хотя существуют значительные культурные и общечеловеческие аспекты DevOps, в этой главе мы обратим особое внимание на одно — возможность автоматизации выделения ресурсов, настройки и развертывания инфраструктуры и приложений.

К 2011 году стало не просто отслеживать все веб-сервисы, составляющие AWS. Облако Amazon далеко шагнуло от простых вычислительных мощностей (Amazon EC2) и объектного хранилища (Amazon S3). Приложения начали полагаться на многочисленные сервисы, взаимодействующие друг с другом, и для автоматизации выделения этих сервисов понадобились соответствующие утилиты. Amazon очень скоро заполнил эту нишу и в 2011-м предложил именно такую утилиту — Amazon CloudFormation. Именно тогда мы наконец действительно могли сказать, что можем описать инфраструктуру в коде. CloudFormation открыл путь к новому поколению утилит типа «инфраструктура как код» (IaC), работающих на уровне с моделью облачной инфраструктуры, ниже уровня обслуживания утилитами управления инфраструктурой первого поколения.

К 2014-му платформам AWS не считывал уже десятки утилит. В этом году в мире IaC появился еще один важный утилит — Terraform от компании HashiCorp. По сей день две чаще всего используемые утилиты IaC — это CloudFormation и Terraform.

Еще один важный шаг вперед в мире IaC и DevOps был сделан в конце 2013-го — начале 2014 года: выпуск Docker, ставшего синонимом контейнерных технологий. И хотя контейнеры существовали уже многие годы, огромным преимуществом Docker стали удобные в использовании API и интерфейс командной строки (CLI) для технологий, подобные контейнерам Linux и контрольным группам, существенно снизившие порог входа для желающих упаковать приложения в контейнеры, которые можно было бы развернуть и запустить везде, где работает Docker. Контейнерные технологии и платформы координации контейнеров обсуждятся подробнее в главах 11 и 12.

Число знающих про Docker и применяющих его пользователей резко возросло и привело к падению популярности утилит управления инфраструктурой первого поколения (Puppet, Chef, Ansible, SaltStack). Работавшие эти утилиты компании сейчас с трудом удерживаются на плаву, срочно перестраиваясь на более удобные рельсы. До появления Docker можно было выделить инфраструктуру для приложения с помощью утилит IaC, например CloudFormation или Terraform, затем развернуть само приложение (код и конфигурацию) с помощью утилиты управления инфраструктурой, например Puppet, Chef, Ansible или SaltStack. Docker

внезапно перевел эти утилиты управления инфраструктурой в разряд устаревших, поскольку предоставил средства для упаковки приложений (код и конфигурации) в контейнер Docker, выполняемый внутри инфраструктуры, выделенной с помощью утилит IaC.

Классификация инструментов автоматизации выделения инфраструктуры

В текущий момент специлист по DevOps может легко потеряться в многообразии доступных инструментов вомтизации выделения инфраструктуры.

Утилиты IaC, в частности, различаются уровнем, на котором работают. Утилиты наподобие CloudFormation и Terraform действуют на уровне облачной инфраструктуры и служат для выделения облачных ресурсов для вычислений, хранения и передачи данных по сети, а также различных дополнительных сервисов — баз данных, очередей сообщений, средств тестирования и многого другого. Утилиты управления инфраструктурой, например Puppet, Chef, Ansible и SaltStack, обычно работают на прикладном уровне и управляют установкой всех необходимых для приложения пакетов, а также правильную настройку самого приложения (хотя у многих из них есть и модули для выделения облачных ресурсов). Docker также работает на прикладном уровне.

Сравнить утилиты IaC можно также путем разбиения их на декларативные и императивные. Действия, которые должны выполнить утилиты вомтизации, можно указать декларативно, описывая желаемое будущее состояние системы. Декларативными являются Puppet, CloudFormation и Terraform. Либо можно работать с утилитой вомтизации процедурным (императивным) способом, указывая точные шаги, необходимые для достижения желаемого состояния системы. Chef и Ansible — императивные. SaltStack может работать как в декларативном, так и в императивном режиме.

Рассмотрим в качестве примера логику желаемого состояния системы предельно простой эскиз какого-либо сооружения, допустим стадион. Процедурные утилиты, такие как Chef и Ansible, строят этот стадион сектор за сектором и ряд за рядом внутри каждого из секторов. Необходимо отслеживать состояние стадиона и ход строительных работ. При использовании же декларативных утилит наподобие Puppet, CloudFormation или Terraform сначала создается эскиз стадиона. А затем утилиты обеспечивают достижение отраженного в эскизе состояния.

С учетом названного выше мы в дальнейшем сосредоточимся на утилитах IaC, которые можно разбить по нескольким измерениям.

Одно измерение — способ задания желаемого состояния системы. В CloudFormation для этого применяется синтаксис JSON или YAML, в Terraform — проприетарный синтаксис языка конфигурации HashiCorp (HCL). А Pulumi и AWS Cloud Development Kit (CDK) позволяют задать действия, а не стоящие языки программирования, включая Python, для описания желаемого состояния системы.

Еще одно измерение — поддерживаемые одной из утилит платформы облачных сервисов. Поскольку CloudFormation — сервис Amazon, логично, что он ориентирован на AWS (хотя с помощью CloudFormation можно описывать и не относящиеся к AWS ресурсы при использовании типа ресурсов `CustomResource`). То же самое справедливо и для AWS CDK. А вот Terraform поддерживает множество платформ облачных сервисов, как и Pulumi.

Поскольку эта книга посвящена Python, мы хотели бы упомянуть утилиту `troposphere` (<https://oreil.ly/Zdid->), с помощью которой можно задать в коде Python шаблоны стека CloudFormation, затем экспортировать их в JSON или YAML. Робот `troposphere` берет на себя генерацию шаблонов стека, значит, вам придется самостоятельно выделять стек с помощью CloudFormation. Еще один робот, работающий с Python и предоставляющий упоминания утилит — `stacker` (https://oreil.ly/gBF_N), который, «закрулившись», применяет `troposphere`, но предотвращает злоупотребление теми же сгенерированными шаблонами стека CloudFormation.

В дальнейшем мы покажем в действии две утилиты в том числе, Terraform и Pulumi, в ходе работы над одним сценарием — развертыванием статического веб-сайта в Amazon S3 с Amazon CloudFront CDN в качестве клиентской части, безопасность которого обеспечивается SSL-сертификатом, выделяемым сервисом AWS Certificate Manager (ACM).



Некоторые из команд в последующих примерах генерируют большие объемы выводимой информации. Мы будем опускать большую часть строк выводимого текста, с исключением случаев, когда они необходимы для понимания работы команд, чтобы сберечь деревья¹ и помочь вам лучше сосредоточиться на происходящем.

Выделение инфраструктуры вручную

Мы начинаем с того, что пройдем по сценарию в ручном режиме с помощью веб-консоли AWS. Ничто не даст полнее и следиться результату в том числе утомительной работы, чем все муки выполнения того же вручную!

¹ Из которых делается бумажка. — *Примеч. пер.*

Сначала будем следовать документации с сайта AWS по хостингу статического веб-сайта в S3 (<https://oreil.ly/kdv8T>).

Мы ранее с помощью Namecheap купили доменное имя `devops4all.dev`. Создали для этого домена зону хостинга в Amazon Route 53 и указали в серверных доменных именах Namecheap для этого домена сервер DNS AWS, отвечающие за домен хостинг.

Мы выделили две корзины S3, одну для корневого URL-сайта (`devops4all.dev`), вторую — для URL с `www` (`www.devops4all.dev`). Нашей идеей было использовать переносимость запросов с `www` на корневой URL. Мы также изучили руководство и настроили корзины и хостинг статического сайта с соответствующими правами доступа. Далее загрузили файл `index.html` и изображение в формате JPG в корневую корзину S3.

Следующий шаг — предоставление SSL-сертификата для корневого доменного имени (`devops4all.dev`), так и для всех его поддоменов (`*.devops4all.dev`). Проверку мы производили с помощью записей DNS, добавленных в зону хостинга Route 53.



ACM-сертификат необходимо предоставить в регионе AWS us-east-1, чтобы можно было использовать его в CloudFront.

Далее мы создали запись AWS CloudFront CDN, указывающую на корневую корзину S3, и воспользовались ACM-сертификатом, созданным на предыдущем шаге. Мы также указали, что HTTP-запросы должны переноситься на HTTPS. После перезагрузки записи (что заняло примерно 15 минут) мы добавили записи Route 53 типа A для корневого домена и домена `www` в качестве псевдонимов, указывающих на имя DNS конечной точки записи CloudFront.

В результате мы добились того, что при переходе по адресу `http://devops4all.dev` автоматически переносится на `https://devops4all.dev` и мы видим домашнюю страницу сайта с загруженным изображением. Мы также попробовали перейти по адресу `http://www.devops4all.dev` и были перенесены на `https://devops4all.dev`.

Создание всех упомянутых ресурсов AWS вручную заняло примерно 30 минут. Мы также потратили 15 минут в ожидании прострелки записи CloudFront, так что в итоге получилось 45 минут. Учтите, что раньше мы все это уже выполняли и знали в точности, что и к чему делать, и лишь изредка глядя в руководство AWS.



Стоит потратить минуту, чтобы оценить по достоинству то, насколько упростилось сейчас создание бесплатного SSL-сертификата. Прошли те времена, когда нужно было ждать часами или даже днями одобрения запроса на получение SSL-сертификата, причем вы должны были предоставить доказательства, что ваша компания существует. При наличии AWS ACM и Let's Encrypt в 2020 году не может быть оправдан тот факт, что не во всех страницах веб-сайта используется SSL.

Автоматическое выделение инфраструктуры с помощью Terraform

Мы решили использовать Terraform в качестве первой утилиты IaC для автоматизации упомянутых задач, хотя он и не имеет непосредственного отношения к Python. Однако у нее есть несколько преимуществ, в частности зрелость, развитая инфраструктура и средства выделения ресурсов для мультиоблаков.

При написании кода Terraform рекомендуется применять модули — переиспользуемые компоненты конфигурации Terraform. Компания HashiCorp поддерживают единый реестр (<https://registry.terraform.io/>) модулей Terraform, в котором вы можете найти готовые модули для выделения нужных ресурсов. В этом же примере мы напишем собственные модули.

Здесь использована версия 0.12.1 Terraform — последняя на момент написания книги. Установить ее на Mac можно с помощью brew:

```
$ brew install terraform
```

Выделение корзины S3

Создадим каталог `modules`, в нем — каталог `s3` с тремя файлами: `main.tf`, `variables.tf` и `outputs.tf`. Файл `main.tf` в каталоге `s3` вызовет Terraform и необходимость создания корзины S3 с заданной политикой. В нем задается объявление в файле `variables.tf` переменная `domain_name`, значение которой передается туда вызывающей модуль стороной. В результате выводится конечная точка DNS корзины S3, используемая далее другими модулями в качестве входной переменной.

Вот как выглядит содержимое названных трех файлов из каталога `modules/s3`:

```
$ cat modules/s3/main.tf
resource "aws_s3_bucket" "www" {
  bucket = "www.${var.domain_name}"
  acl    = "public-read"
  policy = <<POLICY
```



```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AddPerm",
      "Effect": "Allow",
      "Principal": "*",
      "Action": ["s3:GetObject"],
      "Resource": ["arn:aws:s3:::www.${var.domain_name}/*"]
    }
  ]
}
POLICY

website {
  index_document = "index.html"
}
}

$ cat modules/s3/variables.tf
variable "domain_name" {}

$ cat modules/s3/outputs.tf
output "s3_www_website_endpoint" {
  value = "${aws_s3_bucket.www.website_endpoint}"
}

```



Атрибут `policy` вышеупомянутого ресурса `aws_s3_bucket` представляет собой пример политики корзины S3, открывающей общий доступ к корзине. Если вы работаете с корзинами S3 в контексте IaC, вы не сможете познать политику с официальной документацией AWS по корзинам и пользовательским политикам (<https://oreil.ly/QtTYd>).

Основной сценарий Terraform, связывающий воедино все модули, — это файл `main.tf` в текущем каталоге:

```

$ cat main.tf
provider "aws" {
  region = "${var.aws_region}"
}

module "s3" {
  source = "../modules/s3"
  domain_name = "${var.domain_name}"
}

```

Он ссылается на переменные, описанные в отдельном файле `variables.tf`:

```

$ cat variables.tf
variable "aws_region" {

```

```

    default = "us-east-1"
}

variable "domain_name" {
    default = "devops4all.dev"
}

```

Вот дерево текущего к т л о г и д н ный момент:

```

|__main.tf
|__variables.tf
|__modules
| |__s3
| | |__outputs.tf
| | |__main.tf
| | |__variables.tf

```

Первый шаг — запуск Terraform — вызов команды `terraform init` для чтения содержимого всех модулей, на которые ссылается файл `main`.

Следующий шаг — выполнение команды `terraform plan`, создающей вышеупомянутый эскиз.

Для создания указанных в эскизе ресурсов выполните команду `terraform apply`:

```
$ terraform apply
```

```

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

```

Terraform will perform the following actions:

```

# module.s3.aws_s3_bucket.www will be created
+ resource "aws_s3_bucket" "www" {
  + acceleration_status = (known after apply)
  + acl                 = "public-read"
  + arn                 = (known after apply)
  + bucket              = "www.devops4all.dev"
  + bucket_domain_name = (known after apply)
  + bucket_regional_domain_name = (known after apply)
  + force_destroy       = false
  + hosted_zone_id      = (known after apply)
  + id                  = (known after apply)
  + policy              = jsonencode(
    {
      + Statement = [
        + {
          + Action = [
            + "s3:GetObject",
          ]
        }
      ]
    }
  )
}

```

```

        + Effect = "Allow"
        + Principal = "*"
        + Resource = [
            + "arn:aws:s3:::www.devops4all.dev/*",
        ]
        + Sid = "AddPerm"
    },
    ]
    + Version= "2012-10-17"
}
)
+ region = (known after apply)
+ request_payer = (known after apply)
+ website_domain= (known after apply)
+ website_endpoint = (known after apply)

+ versioning {
    + enabled = (known after apply)
    + mfa_delete = (known after apply)
}

+ website {
    + index_document = "index.html"
}
}

```

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value: yes

module.s3.aws_s3_bucket.www: Creating...

module.s3.aws_s3_bucket.www: Creation complete after 7s [www.devops4all.dev]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Теперь с помощью UI веб-консоли AWS можно убедиться, что корзин S3 был создан.

Предоставление SSL-сертификата с помощью ACM AWS

Следующий модуль предназначен для предоставления SSL-сертификата с помощью сервиса AWS Certificate Manager. Создайте каталог `modules/acm`, содержащий три файла: `main.tf`, `variables.tf` и `outputs.tf`. Файл `main.tf` в каталоге `acm` укажет Terraform создать SSL-сертификат ACM с DNS в качестве метода проверки. В нем используется объявленная в файле `variables.tf` переменная `domain_name`, значение которой передается туда вызывающей модуль стороной.

В результате выводится идентификатор ARN сертификата, служящий для входной переменной для других модулей:

```
$ cat modules/acm/main.tf
resource "aws_acm_certificate" "certificate" {
  domain_name = "${var.domain_name}"
  validation_method = "DNS"
  subject_alternative_names = ["*.${var.domain_name}"]
}

$ cat modules/acm/variables.tf
variable "domain_name" {
}

$ cat modules/acm/outputs.tf
output "certificate_arn" {
  value = "${aws_acm_certificate.certificate.arn}"
}
```

Добавляем в файл `main.tf` Terraform ссылку на новый модуль `acm`:

```
$ cat main.tf
provider "aws" {
  region = "${var.aws_region}"
}

module "s3" {
  source = "../modules/s3"
  domain_name = "${var.domain_name}"
}

module "acm" {
  source = "../modules/acm"
  domain_name = "${var.domain_name}"
}
```

Следующие три шага — те же, что и в последовательности создания корзины S3: `terraform init`, `terraform plan` и `terraform apply`.

Воспользуемся консолью AWS для добавления необходимых для процесса проверки записей Route 53. Проверка и выпуск сертификата обычно занимает несколько минут.

Выделение раздачи Amazon CloudFront

Следующий модуль предназначен для выделения раздачи Amazon CloudFront. Создайте каталог `modules/cloudfront`, содержащий три файла: `main.tf`, `variables.tf` и `outputs.tf`. Файл `main.tf` в каталоге `cloudfront` вызывает Terraform создать ресурс раздачи Amazon CloudFront. В нем используются не-

сколько переменных, объявленных в файле `variables.tf`, значения которых передются туда вызывающей модуль стороной. В результате выводятся доменное имя DNS для конечной точки CloudFront и идентификатор зоны хостинга Route 53 для рэкорда, служащие в дальнейшем входными переменными для других модулей:

```
$ cat modules/cloudfront/main.tf
resource "aws_cloudfront_distribution" "www_distribution" {
  origin {
    custom_origin_config {
      // These are all the defaults.
      http_port= "80"
      https_port = "443"
      origin_protocol_policy = "http-only"
      origin_ssl_protocols= ["TLSv1", "TLSv1.1", "TLSv1.2"]
    }

    domain_name = "${var.s3_www_website_endpoint}"
    origin_id= "www.${var.domain_name}"
  }

  enabled = true
  default_root_object = "index.html"

  default_cache_behavior {
    viewer_protocol_policy = "redirect-to-https"
    compress = true
    allowed_methods= ["GET", "HEAD"]
    cached_methods = ["GET", "HEAD"]
    target_origin_id = "www.${var.domain_name}"
    min_ttl = 0
    default_ttl = 86400
    max_ttl = 31536000

    forwarded_values {
      query_string = false
      cookies {
        forward = "none"
      }
    }
  }
}

aliases = ["www.${var.domain_name}"]

restrictions {
  geo_restriction {
    restriction_type = "none"
  }
}

viewer_certificate {
  acm_certificate_arn = "${var.acm_certificate_arn}"
}
```

```

        ssl_support_method = "sni-only"
    }
}

$ cat modules/cloudfront/variables.tf
variable "domain_name" {}
variable "acm_certificate_arn" {}
variable "s3_www_website_endpoint" {}

$ cat modules/cloudfront/outputs.tf
output "domain_name" {
    value = "${aws_cloudfront_distribution.www_distribution.domain_name}"
}

output "hosted_zone_id" {
    value = "${aws_cloudfront_distribution.www_distribution.hosted_zone_id}"
}

```

Добавляем ссылку на модуль `cloudfront` в файл `main.tf` Terraform. Перед тем как добавить `s3_www_website_endpoint` и `acm_certificate_arn` в модуль `cloudfront` в качестве входных переменных. Их значения извлекаются из выходных результатов других модулей, `s3` и `acm` соответственно.



ARN расшифровывается как Amazon Resource Name («имя ресурса Amazon») и представляет собой строковое значение, однозначно идентифицирующее данный ресурс AWS. При использовании утилит IaC, производящих действия внутри AWS, вы увидите немало генерируемых и передаваемых в виде переменных значений ARN.

```

$ cat main.tf
provider "aws" {
    region = "${var.aws_region}"
}

module "s3" {
    source = "./modules/s3"
    domain_name = "${var.domain_name}"
}

module "acm" {
    source = "./modules/acm"
    domain_name = "${var.domain_name}"
}

module "cloudfront" {
    source = "./modules/cloudfront"
    domain_name = "${var.domain_name}"
    s3_www_website_endpoint = "${module.s3.s3_www_website_endpoint}"
    acm_certificate_arn = "${module.acm.certificate_arn}"
}

```

Следующие три шага — уже привычные нам при выделении ресурсов с помощью Terraform `terraform init`, `terraform plan` и `terraform apply`.

В данном случае шаг `terraform apply` займет почти 23 минуты. Выделение ресурса Amazon CloudFront — одна из самых длительных операций в AWS, поскольку «закрулился» Amazon с задержкой почти глобально.

Создание записи DNS Route 53

Следующий модуль предназначен для создания записи DNS Route 53 для основного домена с ит `www.devops4all.dev`. Создайте каталог `modules/route53`, содержащий два файла: `main.tf` и `variables.tf`. Файл `main.tf` в каталоге `route53` вызовет Terraform создать запись DNS Route 53 типа `A` в качестве псевдонима для имени DNS конечной точки CloudFront. В нем используется несколько переменных, объявленных в файле `variables.tf`, значения которых передются туда вызывающей модуль стороной:

```
$ cat modules/route53/main.tf
resource "aws_route53_record" "www" {
  zone_id = "${var.zone_id}"
  name = "www.${var.domain_name}"
  type = "A"
  alias {
    name = "${var.cloudfront_domain_name}"
    zone_id = "${var.cloudfront_zone_id}"
    evaluate_target_health = false
  }
}

$ cat modules/route53/variables.tf
variable "domain_name" {}
variable "zone_id" {}
variable "cloudfront_domain_name" {}
variable "cloudfront_zone_id" {}
```

Добавляем ссылку на модуль `route53` в файл `main.tf` Terraform. Перед ним `zone_id`, `cloudfront_domain_name` и `cloudfront_zone_id` в модуль `route53` в качестве входных переменных. Значение `zone_id` объявлено в файле `variables.tf`, остальные значения извлекаются из выходных результатов модуля `cloudfront`:

```
$ cat main.tf
provider "aws" {
  region = "${var.aws_region}"
}

module "s3" {
  source = "../modules/s3"
```

```

    domain_name = "${var.domain_name}"
}
module "acm" {
    source = "../modules/acm"
    domain_name = "${var.domain_name}"
}
module "cloudfront" {
    source = "../modules/cloudfront"
    domain_name = "${var.domain_name}"
    s3_www_website_endpoint = "${module.s3.s3_www_website_endpoint}"
    acm_certificate_arn = "${module.acm.certificate_arn}"
}
module "route53" {
    source = "../modules/route53"
    domain_name = "${var.domain_name}"
    zone_id = "${var.zone_id}"
    cloudfront_domain_name = "${module.cloudfront.domain_name}"
    cloudfront_zone_id = "${module.cloudfront.hosted_zone_id}"
}
$ cat variables.tf
variable "aws_region" {
    default = "us-east-1"
}
variable "domain_name" {
    default = "devops4all.dev"
}
variable "zone_id" {
    default = "ZWX18ZIVHAA50"
}

```

Следующие три шага, которые мы уже привыкли, предназначены для выделения ресурсов с помощью Terraform: `terraform init`, `terraform plan` и `terraform apply`.

Копирование статических файлов в корзину S3

Для комплексного тестирования выделенного статического веб-сайта создайте простой файл `index.html`, содержащий JPEG-изображение, и скопируйте файл в выделенную ранее с помощью Terraform корзину S3. Убедитесь, что значение переменной среды `AWS_PROFILE` соответствует уже установленному в файле `~/.aws/credentials`:

```

$ echo $AWS_PROFILE
gheorghiu-net
$ aws s3 cp static_files/index.html s3://www.devops4all.dev/index.html
upload: static_files/index.html to s3://www.devops4all.dev/index.html
$ aws s3 cp static_files/devops4all.jpg s3://www.devops4all.dev/devops4all.jpg
upload: static_files/devops4all.jpg to s3://www.devops4all.dev/devops4all.jpg

```


Зайдите по адресу <https://www.devops4all.dev/> и убедитесь, что загруженное JPG-изображение отображается нормально.

Удаление всех ресурсов AWS, выделенных с помощью Terraform

Выделяя облачные ресурсы, не следует забывать об их стоимости. Если забыть об этом, в конце месяца можно получить очень неприятный сюрприз в виде счета AWS. Не забывайте удалять все выделенные ресурсы, для чего можете воспользоваться командой `terraform destroy`. Отметим также, что содержимое корзины S3 необходимо удалить до выполнения команды `terraform destroy`, поскольку Terraform не удаляет непустые корзины.



Перед выполнением команды `terraform destroy` обязательно убедитесь, что вы не удаляете необходимые для промышленной эксплуатации ресурсы!

Автоматическое выделение инфраструктуры с помощью Pulumi

Если говорить об утилитах IaC, то Pulumi — один из новичков. Так что он еще несколько сыроват, особенно если говорить о поддержке Python.

Pulumi позволяет задать желаемое состояние инфраструктуры, указывая выделяемые ресурсы с помощью настоящих языков программирования. Первым поддерживаемым Pulumi языком стал TypeScript, но сейчас поддерживаются также Go и Python.

Важно понимать разницу между написанием кода в том типизации выделения инфраструктуры на Python с помощью Pulumi и с помощью библиотек в том типизации AWS, например Boto.

При использовании Pulumi код Python описывает выделяемые ресурсы. Фактически при этом создается эскиз/состояние, которое мы обсуждали в предыдущей главе. Pulumi, тем не менее, отличается в этом отношении Terraform, но отличается тем, что открывает все возможности языков программирования, например Python, в смысле написания функций, циклов, использования переменных и т. д. Вы не ограничены при этом привычными языковыми конструкциями, как в HCL Terraform. Pulumi объединяет возможности декларативного подхода, при

котором описывается желаемое итоговое состояние, с возможностями написания этого языка программирования.

При действии же библиотек вроде `awscli` и подобие `Boto` в своем коде к описываются, так и выделяются отдельные ресурсы AWS. Никого общего эскиза/состояния нет, значит, необходимо самостоятельно отслеживать все выделяемые ресурсы и координировать их создание и удаление. Такой подход к утилитам вроде `awscli` является императивным (процедурным). Но при нем все равно сохраняется преимущество в виде возможности написания кода Python.

Для использования `Pulumi` необходимо создать бесплатную учетную запись на веб-сайте `pulumi.io`. После этого можно установить утилиту командной строки на своей локальной машине. На компьютере `Macintosh` для установки `Pulumi` можно применить `Homebrew`.

Первая команда, которую необходимо выполнить локально, — `pulumi login`:

```
$ pulumi login
Logged into pulumi.com as griggheo (https://app.pulumi.com/griggheo)
```

Создание нового проекта Pulumi на Python для AWS

Создаем каталог `proj1`, выполняем в нем команду `pulumi new` и выбираем шаблон `aws-python`. В процессе создания проекта `pulumi` предложит выбрать название стека. Назовем его `staging`:

```
$ mkdir proj1
$ cd proj1
$ pulumi new
Please choose a template: aws-python      A minimal AWS Python Pulumi program
This command will walk you through creating a new Pulumi project.
```

Enter a value or leave blank to accept the (default), and press <ENTER>.
Press ^C at any time to quit.

```
project name: (proj1)
project description: (A minimal AWS Python Pulumi program)
Created project 'proj1'
```

```
stack name: (dev) staging
Created stack 'staging'
```

```
aws:region: The AWS region to deploy into: (us-east-1)
Saved config
```

Your new project is ready to go!

To perform an initial deployment, run the following commands:

1. `virtualenv -p python3 venv`
2. `source venv/bin/activate`
3. `pip3 install -r requirements.txt`

Then, run 'pulumi up'

Важно осознать разницу между проектом Pulumi и стеком Pulumi. Проект — это код, с помощью которого задается желаемое состояние системы, то есть ресурсы, которые Pulumi должен выделить. А стек — это конкретное развернутое приложение. Стек может соответствовать конкретной среде — прототипу, прототипу, прототипу или промышленной эксплуатации. В следующих примерах мы создадим два стека Pulumi: `staging` для среды прототипа и `prod` для промышленной эксплуатации.

Вот файлы, входящие в шаблон `aws-python`, которые сгенерировал командой `pulumi new`:

```
$ ls -la
total 40
drwxr-xr-x  7 ggheo  staff  224 Jun 13 21:43 .
drwxr-xr-x 11 ggheo  staff  352 Jun 13 21:42 ..
-rw-r--r--  1 ggheo  staff   12 Jun 13 21:43 .gitignore
-rw-r--r--  1 ggheo  staff   32 Jun 13 21:43 Pulumi.staging.yaml
-rw-r--r--  1 ggheo  staff   77 Jun 13 21:43 Pulumi.yaml
-rw-r--r--  1 ggheo  staff  184 Jun 13 21:43 __main__.py
-rw-r--r--  1 ggheo  staff   34 Jun 13 21:43 requirements.txt
```

Следуем инструкциям, выведенным командой `pulumi new`, и устанавливаем `virtualenv`, после чего создаем новую среду `virtualenv` и устанавливаем указанные в файле `requirements.txt` библиотеки:

```
$ pip3 install virtualenv
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv) pip3 install -r requirements.txt
```



Прежде чем выделять с помощью команды `pulumi up` какие-либо ресурсы AWS, необходимо убедиться, что используется нужная учетная запись AWS. Один из вариантов — использовать нужную учетную запись AWS — задать переменную среды `AWS_PROFILE` в текущей командной оболочке. В противном случае в локальном файле `~/.aws/credentials` уже был настроен профиль `AWS gheorghiu-net`.

```
(venv) export AWS_PROFILE=gheorghiu-net
```

Сгенерированный Pulumi в виде чистого шаблона `aws-python` файл `__main__.py` выглядит следующим образом:

```
$ cat __main__.py
import pulumi
from pulumi_aws import s3

# Создаем ресурс AWS (корзину S3)
bucket = s3.Bucket('my-bucket')

# Экспорт названия корзины
pulumi.export('bucket_name', bucket.id)
```

Клонируем локальную м/шину репозиторий GitHub примеров Pulumi (<https://oreil.ly/SIT-v>), после чего копируем файл `__main__.py` из `pulumi-examples/aws-py-s3-folder` в текущий каталог.

Вот новый файл `__main__.py` в этом каталоге:

```
$ cat __main__.py
import json
import mimetypes
import os

from pulumi import export, FileAsset
from pulumi_aws import s3

web_bucket = s3.Bucket('s3-website-bucket', website={
    "index_document": "index.html"
})

content_dir = "www"
for file in os.listdir(content_dir):
    filepath = os.path.join(content_dir, file)
    mime_type, _ = mimetypes.guess_type(filepath)
    obj = s3.BucketObject(file,
        bucket=web_bucket.id,
        source=FileAsset(filepath),
        content_type=mime_type)

def public_read_policy_for_bucket(bucket_name):
    return json.dumps({
        "Version": "2012-10-17",
        "Statement": [{
            "Effect": "Allow",
            "Principal": "*",
            "Action": [
                "s3:GetObject"
            ],
```

```

        "Resource": [
            f"arn:aws:s3::{bucket_name}/*",
        ]
    }]
})

```

```

bucket_name = web_bucket.id
bucket_policy = s3.BucketPolicy("bucket-policy",
    bucket=bucket_name,
    policy=bucket_name.apply(public_read_policy_for_bucket))

```

```

# Экспорт названия корзины
export('bucket_name', web_bucket.id)
export('website_url', web_bucket.website_endpoint)

```

Обратите внимание на использование переменных Python для `content_dir` и `bucket_name`, использование цикла `for`, а также обычной функции Python `public_read_policy_for_bucket`. Так приятно иметь возможность применять обычные конструкции языка Python в программах IaC!

Пришло время запустить `pulumi up`, чтобы выделить указанные в файле `__main__.py` ресурсы. Этот командный отобразит все созданные ресурсы. Выберите в ответ `yes`, чтобы запустить процесс выделения ресурсов:

```

(venv) pulumi up
Previewing update (staging):

```

| | Type | Name | Plan |
|---|------------------------|-------------------|--------|
| + | pulumi:pulumi:Stack | proj1-staging | create |
| + | └─ aws:s3:Bucket | s3-website-bucket | create |
| + | └─ aws:s3:BucketObject | favicon.png | create |
| + | └─ aws:s3:BucketPolicy | bucket-policy | create |
| + | └─ aws:s3:BucketObject | python.png | create |
| + | └─ aws:s3:BucketObject | index.html | create |

```

Resources:
  + 6 to create

```

```

Do you want to perform this update? yes
Updating (staging):

```

| | Type | Name | Status |
|---|------------------------|-------------------|---------|
| + | pulumi:pulumi:Stack | proj1-staging | created |
| + | └─ aws:s3:Bucket | s3-website-bucket | created |
| + | └─ aws:s3:BucketObject | index.html | created |
| + | └─ aws:s3:BucketObject | python.png | created |
| + | └─ aws:s3:BucketObject | favicon.png | created |
| + | └─ aws:s3:BucketPolicy | bucket-policy | created |

Outputs:

```
bucket_name: "s3-website-bucket-8e08f8f"
website_url: "s3-website-bucket-8e08f8f.s3-website-us-east-1.amazonaws.com"
```

Resources:

```
+ 6 created
```

Duration: 14s

Просмотрим имеющиеся стеки Pulumi:

```
(venv) pulumi stack ls
NAME      LAST UPDATE   RESOURCE COUNT  URL
staging*  2 minutes ago  7               https://app.pulumi.com/griggheo/proj1/staging
```

```
(venv) pulumi stack
```

Current stack is staging:

```
Owner: griggheo
Last updated: 3 minutes ago (2019-06-13 22:05:38.088773 -0700 PDT)
Pulumi version: v0.17.16
```

Current stack resources (7):

| TYPE | NAME |
|----------------------------------|-------------------|
| pulumi:pulumi:Stack | proj1-staging |
| pulumi:providers:aws | default |
| aws:s3/bucket:Bucket | s3-website-bucket |
| aws:s3/bucketPolicy:BucketPolicy | bucket-policy |
| aws:s3/bucketObject:BucketObject | index.html |
| aws:s3/bucketObject:BucketObject | favicon.png |
| aws:s3/bucketObject:BucketObject | python.png |

Просмотрим вывод текущего стека :

```
(venv) pulumi stack output
```

Current stack outputs (2):

| OUTPUT | VALUE |
|-------------|--|
| bucket_name | s3-website-bucket-8e08f8f |
| website_url | s3-website-bucket-8e08f8f.s3-website-us-east-1.amazonaws.com |

Зайдите по URL, приведенный в выводе `website_url` (<http://s3-website-bucket-8e08f8f.s3-website-us-east-1.amazonaws.com/>), и убедитесь, что статический сайт доступен.

В следующих разделах мы расширим этот проект Pulumi, добавив дополнительные ресурсы AWS для выделения, чтобы выделить то же, что и с помощью Terraform: SSL-сертификат ACM, ресурс CloudFront и запись DNS Route 53 для URL нашего сайта.

Создание значений параметров конфигурации для стека staging

Наш текущий стек — `staging`. Переименуем существующий каталог `www` в `www-staging`, после чего с помощью команды `pulumi config set` зададим значения двух параметров конфигурации для текущего стека: `staging:domain_name` и `local_webdir`:

```
(venv) mv www www-staging
(venv) pulumi config set local_webdir www-staging
(venv) pulumi config set domain_name staging.devops4all.dev
```



Больше подробностей о том, как Pulumi обращается со значениями параметров конфигурации и секретными данными, вы можете найти в справочной документации Pulumi (https://oreil.ly/D_Cy5).

Для просмотра имеющихся значений параметров конфигурации для текущего стека выполняем:

```
(venv) pulumi config
KEY          VALUE
aws:region   us-east-1
domain_name  staging.devops4all.dev
local_webdir  www-staging
```

Задания параметров конфигурации, воспользуемся ими в коде Pulumi:

```
import pulumi

config = pulumi.Config('proj1') # proj1 is project name defined in Pulumi.yaml

content_dir = config.require('local_webdir')
domain_name = config.require('domain_name')
```

Теперь значения параметров конфигурации заданы, далее создадим SSL-сертификат с помощью сервиса AWS Certificate Manager.

Создаем SSL-сертификат ACM

В этом месте становится заметно, что Pulumi несколько сыроват в том, что касается SDK Python. Просто прочитайте руководство Pulumi по Python SDK для модуля `acm` (<https://oreil.ly/Niwaj>) недостоаточно, чтобы разобраться, что нужно делать в программе Pulumi.

К счастью, существует множество примеров Pulumi на TypeScript, из которых можно почерпнуть вдохновение. Один из таких примеров, прекрасно иллюстрирующий различные сценарии использования, — `aws-ts-static-website` (<https://oreil.ly/7F39c>).

Вот код TypeScript для создания нового сертификата ACM (из файла `index.ts` (<https://oreil.ly/mISr1>)):

```
const certificate = new aws.acm.Certificate("certificate", {
  domainName: config.targetDomain,
  validationMethod: "DNS",
}, { provider: eastRegion });
```

А вот неписанный нами эквивалентный код на языке Python:

```
from pulumi_aws import acm

cert = acm.Certificate('certificate', domain_name=domain_name,
  validation_method='DNS')
```



Эмпирическое правило по переносу кода Pulumi с TypeScript на Python: не зная параметров в верблюжьем регистре в языке Python необходимо преобразовать в змеиный регистр. Как вы видели в предыдущем примере, `domainName` при этом превращается в `domain_name`, `validationMethod` — в `validation_method`.

Следующий этап — выделение зоны Route 53, в ней — проверочный записи DNS для SSL-сертификата ACM.

Выделение зоны Route 53 и записей DNS

Выделение новой зоны Route 53 с помощью Pulumi не доставляет сложностей, если следовать сплоченному руководству Pulumi SDK по модулю `route53` (<https://oreil.ly/cU9Yj>):

```
from pulumi_aws import route53

domain_name = config.require('domain_name')

# Разбиваем доменное имя на имя поддомена и родительского домена,
# например "www.example.com" => "www", "example.com"
def get_domain_and_subdomain(domain):
    names = domain.split(".")
    if len(names) < 3:
        return('', domain)
    subdomain = names[0]
    parent_domain = ".".join(names[1:])
    return (subdomain, parent_domain)
```



```
(subdomain, parent_domain) = get_domain_and_subdomain(domain_name)
zone = route53.Zone("route53_zone", name=parent_domain)
```

Предыдущий фрагмент кода демонстрирует разбиение на две части прочитанного в переменную `domain_name` значения параметра конфигурации с помощью обычных функций Python. Например, `domain_name` со значением `staging.devops4all.dev` будет разбито на `subdomain(staging)` и `parent_domain(devops4all.dev)`.

Переменная `parent_domain` далее используется в качестве параметра конструктора объекта `zone`, который вызывает Pulumi необходимость выделить ресурс `route53.Zone`.



После создания зоны Route 53 необходимо сделать так, чтобы серверы имен Namecheap указывали на серверы имен, прописанные в записи DNS для нашей новой зоны, чтобы он был доступен всем.

До сих пор все было хорошо. Следующий этап — создание сертификата ACM и записи DNS для его проверки.

Сначала мы попытались перенести пример с языка TypeScript, следуя эмпирическому правилу преобразования верблюжьего регистра в змеиный:

```
TypeScript:
const certificateValidationDomain = new aws.route53.Record(
  `${config.targetDomain}-validation`, {
    name: certificate.domainValidationOptions[0].resourceRecordName,
    zoneId: hostedZoneId,
    type: certificate.domainValidationOptions[0].resourceRecordType,
    records: [certificate.domainValidationOptions[0].resourceRecordValue],
    ttl: tenMinutes,
  });
```

Первая попытка преобразования в код на Python путем изменения регистра :

```
cert = acm.Certificate('certificate',
  domain_name=domain_name, validation_method='DNS')

domain_validation_options = cert.domain_validation_options[0]

cert_validation_record = route53.Record(
  'cert-validation-record',
  name=domain_validation_options.resource_record_name,
  zone_id=zone.id,
  type=domain_validation_options.resource_record_type,
  records=[domain_validation_options.resource_record_value],
  ttl=600)
```

Безрезультатно. Команда `pulumi up` возвращает ошибку:

```
AttributeError: 'dict' object has no attribute 'resource_record_name'
```

Это неслучайно — дело в том, поскольку документация по SDK Python не содержит столько подробной информации. Мы не знали, какие атрибуты необходимо задать для объекта `domain_validation_options`.

Справиться с этой ситуацией нам удалось лишь тогда, когда мы добавили объект `domain_validation_options` в список экспортов Pulumi, выводимый Pulumi в консоль в конце операции `pulumi up`:

```
export('domain_validation_options', domain_validation_options)
```

В результате команда `pulumi up` вывел следующее:

```
+ domain_validation_options: {
+   domain_name      : "staging.devops4all.dev"
+   resourceRecordName : "_c5f82e0f032d0f4f6c7de17fc2c.staging.devops4all.dev."
+   resourceRecordType : "CNAME"
+   resourceRecordValue: "_08e3d475bf3aeda0c98.1tfvzjuylp.acm-validations.aws."
+ }
```

В точку! Оказалось, что атрибуты объекта `domain_validation_options` остались в верблюжьем регистре.

Вот вторая, уже попытка переноса в Python:

```
cert_validation_record = route53.Record(
    'cert-validation-record',
    name=domain_validation_options['resourceRecordName'],
    zone_id=zone.id,
    type=domain_validation_options['resourceRecordType'],
    records=[domain_validation_options['resourceRecordValue']],
    ttl=600)
```

Далее у нас появился новый тип выделяемого ресурса — ресурс завершения проверки сертификата. В результате операция `pulumi up` ждет, пока ACM завершит проверку сертификата, обратившись к созданной ранее записи проверки Route 53:

```
cert_validation_completion = acm.CertificateValidation(
    'cert-validation-completion',
    certificate_arn=cert.arn,
    validation_record_fqdn=[cert_validation_dns_record.fqdn])
```

```
cert_arn = cert_validation_completion.certificate_arn
```

Теперь у нас есть полностью автоматизированный способ создания SSL-сертификата ACM, также проверки его через DNS.

Следующий этап — выделение ресурсов CloudFront и корзины S3, где размещены статические файлы для нашего сайта.

Выделение раздачи CloudFront

Воспользуемся руководством SDK для модуля `cloudfront` (<https://oreil.ly/4n98->), чтобы разобраться, какие параметры конструктор необходимо передать в `cloudfront.Distribution`. Выясним, какими должны быть значения этих параметров, изучив внимательно код на TypeScript.

Вот конечный результат:

```
log_bucket = s3.Bucket('cdn-log-bucket', acl='private')

cloudfront_distro = cloudfront.Distribution ( 'cloudfront-distro',
    enabled=True,
    aliases=[ domain_name ],
    origins=[
        {
            'originId': web_bucket.arn,
            'domainName': web_bucket.website_endpoint,
            'customOriginConfig': {
                'originProtocolPolicy': "http-only",
                'httpPort': 80,
                'httpsPort': 443,
                'originSslProtocols': ["TLSv1.2"],
            },
        },
    ],
    default_root_object="index.html",
    default_cache_behavior={
        'targetOriginId': web_bucket.arn,

        'viewerProtocolPolicy': "redirect-to-https",
        'allowedMethods': ["GET", "HEAD", "OPTIONS"],
        'cachedMethods': ["GET", "HEAD", "OPTIONS"],
        'forwardedValues': {
            'cookies': { 'forward': "none" },
            'queryString': False,
        },
    },
    'minTtl': 0,
    'defaultTtl': 600,
    'maxTtl': 600,
},
    price_class="PriceClass_100",
    custom_error_responses=[
        { 'errorCode': 404, 'responseCode': 404,
```

```

        'responsePagePath': "/404.html" },
    ],
    restrictions={
        'geoRestriction': {
            'restrictionType': "none",
        },
    },
    viewer_certificate={
        'acmCertificateArn': cert_arn,
        'sslSupportMethod': "sni-only",
    },
    logging_config={
        'bucket': log_bucket.bucket_domain_name,
        'includeCookies': False,
        'prefix': domain_name,
    })

```

Осталось запустить `pulumi up` для выделения ресурсов CloudFront.

Создание записи DNS Route 53 для URL сайта

Последний шаг комплексного выделения ресурсов для стека `staging` представляет собой относительно простую задачу — запись DNS типа `A` в качестве псевдонима для домена конечной точки CloudFront:

```

site_dns_record = route53.Record(
    'site-dns-record',
    name=subdomain,
    zone_id=zone.id,
    type="A",
    aliases=[
        {
            'name': cloudfront_distro.domain_name,
            'zoneId': cloudfront_distro.hosted_zone_id,
            'evaluateTargetHealth': True
        }
    ])

```

Как обычно, выполняем команду `pulumi up`.

Теперь можно перейти по адресу `https://staging.devops4all.dev` и увидеть загруженные в S3 файлы. Перейдите в корзину журналов в консоли AWS и убедитесь в наличии трафика журналов CloudFront.

Теперь посмотрим, как развернуть тот же самый проект Pulumi в новой среде, которой соответствует новый стек Pulumi.

Создание и развертывание нового стека

Мы решили модифицировать наш прогресс Pulumi так, чтобы он не выделял новую зону Route 53, используя значение идентификатора уже существующей зоны из параметров конфигурации.

Для создания стека `prod` применим команду `pulumi stack init`, указав в качестве имени `prod`:

```
(venv) pulumi stack init
Please enter your desired stack name: prod
Created stack 'prod'
```

В списке стеков теперь два стека, `staging` и `prod` со звездочкой, означающей, что этот стек является текущим:

```
(venv) pulumi stack ls
NAME      LAST UPDATE   RESOURCE COUNT  URL
prod*     n/a           n/a             https://app.pulumi.com/griggheo/proj1/prod
staging   14 minutes ago 14             https://app.pulumi.com/griggheo/proj1/staging
```

Пришло время задать подходящие значения параметров конфигурации для стека `prod`. Воспользуемся новым параметром конфигурации `dns_zone_id` со значением, рвным идентификатору зоны, уже созданной Pulumi при выделении стека `staging`:

```
(venv) pulumi config set aws:region us-east-1
(venv) pulumi config set local_webdir www-prod
(venv) pulumi config set domain_name www.devops4all.dev
(venv) pulumi config set dns_zone_id Z2FTL2X8M0EBTW
```

Меняем код так, чтобы читать значение `zone_id` из конфигурации, не создавая объект зоны Route 53.

Выделяем ресурсы AWS с помощью команды `pulumi up`:

```
(venv) pulumi up
Previewing update (prod):
```

| | Type | Name | Plan |
|---|--------------------------------|-------------------|--------|
| | pulumi:pulumi:Stack | proj1-prod | |
| + | └─ aws:cloudfront:Distribution | cloudfront-distro | create |
| + | └─ aws:route53:Record | site-dns-record | create |

```
Resources:
  + 2 to create
  10 unchanged
```

Do you want to perform this update? yes

Updating (prod):

| | Type | Name | Status |
|---|--------------------------------|-------------------|---------|
| | pulumi:pulumi:Stack | proj1-prod | |
| + | └─ aws:cloudfront:Distribution | cloudfront-distro | created |
| + | └─ aws:route53:Record | site-dns-record | created |

Outputs:

```
+ cloudfront_domain: "d3uhgbdw67nmlc.cloudfront.net"
+ log_bucket_id      : "cdn-log-bucket-53d8ea3"
+ web_bucket_id      : "s3-website-bucket-cde"
+ website_url        : "s3-website-bucket-cde.s3-website-us-east-1.amazonaws.com"
```

Resources:

```
+ 2 created
10 unchanged
```

Duration: 18m54s

Успешно! Стек `prod` полностью развернут.

Одновременно с этим содержимое каталога `www-prod` состоит из статических файлов, минимального сжатия и в настоящий момент идентично содержимому каталога `www-staging`.

Отредактируем файл `www-prod/index.html`, поменяв «Hello, S3!» на «Hello, S3 production!», после чего снова запустим `pulumi up`, чтобы подхватить изменения и загрузить модифицированный файл в S3:

```
(venv) pulumi up
```

Previewing update (prod):

| | Type | Name | Plan | Info |
|---|------------------------|------------|--------|-----------------|
| ~ | pulumi:pulumi:Stack | proj1-prod | | |
| | └─ aws:s3:BucketObject | index.html | update | [diff: ~source] |

Resources:

```
~ 1 to update
11 unchanged
```

Do you want to perform this update? yes

Updating (prod):

| | Type | Name | Status | Info |
|---|------------------------|------------|---------|-----------------|
| ~ | pulumi:pulumi:Stack | proj1-prod | | |
| | └─ aws:s3:BucketObject | index.html | updated | [diff: ~source] |

Outputs:

```
cloudfront_domain: "d3uhgbdw67nmlc.cloudfront.net"
log_bucket_id      : "cdn-log-bucket-53d8ea3"
web_bucket_id      : "s3-website-bucket-cde"
```

```
website_url      : "s3-website-bucket-cde.s3-website-us-east-1.amazonaws.com"
Resources:
  ~ 1 updated
  11 unchanged
Duration: 4s
```

Уд ляем кэш р зд чи CloudFront, чтобы увидеть изменения.

Теперь можно з йти н <https://www.devops4all.dev> и увидеть сообщение: Hello, S3 production!.

Один из подводных к мней утилит IaC, отслежив ющих состояние системы: в некоторых ситу циях видимое утилитой состояние может отлич ться от ф ктического. В этом случ е в жно синхронизиров ть эти дв состояния, в про- тивном случ е они будут р сходиться все больше и вы ст нете бояться вносить к кие-либо изменения из стр х н рушить р боту н ходящейся в промышленной эксплу т ции системы. Не случ йно в н зв нии «*инфр структур к к код*» упомянуто слово «код». Рекомендуемые пр ктики использов ния утилит IaC гл сят, что выделять все ресурсы жел тельно в коде, не вручную. Придержи- в ться этих рекоменд ций непросто, но в долгосрочной перспективе з тр ты н это опр вдыв ют себя.

Упражнения

- Выделите тот же с мый н бор ресурсов AWS с помощью AWS Cloud Development Kit (<https://aws.amazon.com/cdk>).
- Выделите с помощью Terraform или Pulumi обл чные ресурсы у других пост вщиков обл чных сервисов, н пример Google Cloud Platform или Microsoft Azure.

Контейнерные технологии: Docker и Docker Compose

Технологии виртуализации существуют еще со времен мэйнфреймов IBM. Мне довелось побывать на мэйнфрейме, но мы уверены, что хотя бы некоторые из читателей этой книги помнят времена, когда приходилось строить или использовать физический сервер от кого-либо из тех производителей, как HP или Dell. Эти производители до сих пор существуют, и до сих пор можно побывать с теми серверами, размещенными в центре обработки данных, которые были добрыми временами доткомов.

Впрочем, большинство людей, говоря о виртуализации, подразумевают виртуальные мэйнфреймы. Чаще всего они представляют себе при этом виртуальную машину (VM), в которой работает гостевая операционная система, например Fedora или Ubuntu, под управлением гипервизора, например VMware ESX или Citrix/Xen. Огромное преимущество виртуальных машин над обычными серверами состоит в том, что при их использовании можно оптимизировать ресурсы сервера (CPU, оперативная память, диск), распределяя их на несколько виртуальных машин. Кроме того, можно задействовать несколько операционных систем, каждая — в собственной виртуальной машине на одном общем физическом сервере, не покупая по отдельному серверу для каждой операционной системы. Сервисы облачных вычислений, такие как Amazon EC2, были бы невозможны без гипервизоров и виртуальных машин. Подобную виртуализацию можно назвать виртуализацией уровня ядра, поскольку в каждой виртуальной машине работает собственное ядро операционной системы.

В вечной гонке за максимальной отдачей от каждого доллара люди осознали, что даже виртуальные машины расходуют ресурсы не совсем экономно. Следу-

ющий логичный шаг — изолировать отдельное приложение в его собственной виртуальной среде. Достичь этого можно с помощью контейнеров, работающих в одном ядре операционной системы. В данном случае они изолируются на уровне файловой системы. Контейнеры Linux (LXC) и зоны Sun Solaris — самые ранние примеры подобных технологий. Их недостатки — сложность использования и тесная связь с операционной системой, в которой они работают. Один из крупнейших прорывов в сфере контейнеров связан с Docker, предоставляющим простой способ запуска и работы с контейнерами уровня файловой системы.

Что такое контейнер Docker

Контейнер Docker инкапсулирует приложение вместе с другими пакетами ПО и библиотеками, необходимыми для работы этого приложения. Иногда термины «контейнер Docker» и «образ контейнера Docker» используются как синонимы, но между этими понятиями есть важное различие. Образ контейнера Docker называется объектом уровня файловой системы, инкапсулирующий приложение. Контейнером же Docker образ становится после запуска.

Можно выполнять много контейнеров Docker, использующих одно ядро операционной системы. Единственное требование — на хост-компьютере, где будут запускаться контейнеры, необходимо установить серверный компонент — так называемый демон (движок) Docker. Это позволяет более полно действовать и тонко разделять между контейнерами ресурсы хоста, значит, отдаленно увеличить.

Контейнеры Docker обеспечивают большую изоляцию и лучшее управление ресурсами, чем обычные процессы Linux (у полноценных виртуальных машин изоляция еще больше, управление ресурсами еще лучше). Для достижения подобного уровня изоляции и управления ресурсами движок Docker применяет те возможности ядра Linux, как пространства имен, контрольные группы и вспомогательная файловая система UnionFS.

Основное преимущество контейнеров Docker — переносимость. Созданный образ контейнера Docker можно запустить на любой операционной системе хоста, где есть серверный демон Docker. В настоящее время демон Docker может работать во всех основных операционных системах: Linux, Windows и macOS.

Несомненно, все это звучит несколько быстрее, так что пора привести конкретные примеры.

Создание, сборка, запуск и удаление образов и контейнеров Docker

Поскольку книга посвящена языку Python и DevOps, в качестве первого пример приложения, работающего в контейнере Docker, воспользуемся классическим приложением «Hello, World» Flask. В приведенных в этом разделе примерах применяется пакет Docker для Mac. В последующих разделах мы посмотрим, как установить Docker на Linux.

Вот основной файл нашего приложения Flask:

```
$ cat app.py
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World! (from a Docker container)'
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Нашему компьютеру понадобится файл `requirements.txt`, где будет указана версия пакета Flask, которую необходимо установить с помощью `pip`:

```
$ cat requirements.txt
Flask==1.0.2
```

Попытка выполнить файл `app.py` непосредственно с помощью Python на ноутбуке под управлением macOS без установки требующихся предельно пакетов приведет к ошибке:

```
$ python app.py
Traceback (most recent call last):
  File "app.py", line 1, in <module>
    from flask import Flask
ImportError: No module named flask
```

Проще всего решить эту проблему, установив нужный пакет на локальной машине с помощью системы управления пакетами `pip`. В результате возникнет нежелательная привязка к локальной операционной системе. А что, если приложение понадобится развернуть на сервере под управлением другой операционной системы? Тогда возникнет известная проблема, которую можно условно описать фразой «Работает на моей машине», при которой все прекрасно работает на нашем ноутбуке под управлением macOS, но по какой-то загадочной причине, обычно связанной с ориентированными на конкретную операционную систему

версиями библиотек Python, перестает работать на сервере предэксплуатационного тестирования или промышленной эксплуатации с другими операционными системами, например Ubuntu или Red Hat Linux.

Docker предлагает изящный выход из этой ситуации. Разработчик по-прежнему выполняется локальной машине с помощью наших любимых редакторов и наборов программных средств, но зависимость приложения укладывается во внешний переносимый контейнер Docker.

Вот Dockerfile с описанием будущего образа Docker:

```
$ cat Dockerfile
FROM python:3.7.3-alpine

ENV APP_HOME /app
WORKDIR $APP_HOME

COPY requirements.txt .

RUN pip install -r requirements.txt

ENTRYPOINT [ "python" ]
CMD [ "app.py" ]
```

Несколько примечаний относительно того, что делает этот Dockerfile.

- Использует ранее собранный образ Docker для Python 3.7.3, основанный на дистрибутиве Alpine, благодаря которому получаются меньшие по размеру образы Docker. Данный образ Docker также включает те же исполняемые файлы, что и python и pip.
- Устанавливает с помощью pip нужные пакеты.
- Задает ENTRYPOINT и CMD. Различие между ними состоит в том, что при запуске Docker собираемого из Dockerfile образа запускается программа ENTRYPOINT, за которой следуют все указанные в CMD аргументы. В данном случае выполняется команда `python app.py`.



Если не указать в Dockerfile ENTRYPOINT, будет использоваться следующее значение по умолчанию: `/bin/sh -c`.

Чтобы создать образ Docker для этого приложения, выполните команду `docker build`:

```
$ docker build -t hello-world-docker .
```

Чтобы проверить, что образ Docker был сохранен на локальной машине, выполните команду `docker images` с названием образа:

```
$ docker images hello-world-docker
REPOSITORY          TAG         IMAGE ID          CREATED           SIZE
hello-world-docker  latest     dbd84c229002     2 minutes ago    97.7MB
```

Для запуска образа Docker как контейнер Docker предначинена команда `docker run`:

```
$ docker run --rm -d -v `pwd`:/app -p 5000:5000 hello-world-docker
c879295baa26d9dff1473460bab810cbf6071c53183890232971d1b473910602
```

Несколько примечаний относительно аргументов команды `docker run`.

- Аргумент `--rm` указывает серверу Docker на необходимость удалить этот контейнер по завершении его выполнения. Удобная возможность для предотвращения хлечения локальной файловой системы.
- Аргумент `-d` указывает серверу Docker, что этот контейнер нужно запустить в фоновом режиме.
- С помощью аргумента `-v` текущий каталог (`pwd`) привязывается к каталогу `/app` внутри контейнера Docker, что важно для технологического процесса локальной разработки, поскольку позволяет редактировать файлы приложения на локальной машине с помощью переносимой переносимой их сервером для разработки Flask, запущенным внутри контейнера.
- Аргумент `-p 5000:5000` привязывает первый (локальный) из указанных портов (5000) ко второму порту (5000) внутри контейнера.

Вывести список запущенных контейнеров можно с помощью команды `docker ps`. Помните идентификатор контейнера, он будет использоваться в других наших командах `docker`:

```
$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED
c879295baa26   hello-world-docker:latest          "python app.py"         4 seconds ago
STATUS        PORTS
Up 2 seconds   0.0.0.0:5000->5000/tcp              flamboyant_germain
```

Просмотреть журналы для конкретного контейнера можно с помощью команды `docker logs`, указав название или идентификатор этого контейнера:

```
$ docker logs c879295baa26
* Serving Flask app "app" (lazy loading)
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 647-161-014
```

Выполните запрос к URL конечной точки, чтобы проверить работоспособность приложения. Поскольку порт 5000 запущенного внутри контейнера Docker приложения привязан к порту 5000 на локальной машине с помощью следующей строки `-p`, можно использовать в качестве конечной точки приложения локальный IP-адрес 127.0.0.1 и порт 5000:

```
$ curl http://127.0.0.1:5000
Hello, World! (from a Docker container)%
```

Теперь поменяйте код в `app.py`, воспользовавшись своим любимым редактором. Поменяйте текст приветствия на `Hello, World! (from a Docker container with modified code)`. Сохраните файл `app.py` и обратитесь к примеру строк в журнале контейнера Docker:

```
* Detected change in '/app/app.py', reloading
* Restarting with stat
* Debugger is active!
* Debugger PIN: 647-161-014
```

Это демонстрирует, что запущенный в контейнере сервер для запуска ботки Flask обнаружил изменение файла `app.py` и перезагрузил приложение.

Теперь при запросе к конечной точке приложения с помощью утилиты `curl` будет отображено измененное приветствие:

```
$ curl http://127.0.0.1:5000
Hello, World! (from a Docker container with modified code)%
```

Закончить выполнение контейнера можно с помощью команды `docker stop` или `docker kill`, указав в качестве аргумента идентификатор контейнера:

```
$ docker stop c879295baa26
c879295baa26
```

Удалить образ Docker с локального диска можно с помощью команды `docker rmi`:

```
$ docker rmi hello-world-docker
Untagged: hello-world-docker:latest
Deleted: sha256:dbd84c229002950550334224b4b42aba948ce450320a4d8388fa253348126402
Deleted: sha256:6a8f3db7658520a1654cc6abee8eafb463a72ddc3aa25f35ac0c5b1eccdf75cd
Deleted: sha256:aee7c3304ef6ff620956850e0b6e6b1a5a5828b58334c1b82b1a1c21afa8651f
Deleted: sha256:dca8a433d31fa06ab72af63ae23952ff27b702186de8cbea51cdea579f9221e8
Deleted: sha256:cb9d58c66b63059f39d2e70f05916fe466e5c99af919b425aa602091c943d424
Deleted: sha256:f0534bdca48bfdded3c772c67489f139d1cab72d44a19c5972ed2cd09151564c1
```

Эта выведенная информация демонстрирует различные слои файловой системы, из которых состоит образ Docker. При удалении образа удаляются и эти слои. Больше подробностей об использовании Docker слоев файловой системы для сборки образов можно найти в документации по драйверу хранения Docker (<https://oreil.ly/wqNve>).

Публикация образов Docker в реестре Docker

После сборки локальной м-шине образ Docker можно опубликовать его в публичный реестр образов Docker. Существует несколько общедоступных реестров, в этом примере применим Docker Hub. Эти реестры предназначены для того, чтобы люди и организации могли обмениваться своими образами Docker с целью переиспользования на различных машинах и в операционных системах.

Вначале создайте бесплатную учетную запись на Docker Hub (<https://hub.docker.com/>), затем — репозиторий, общедоступный или частный. Мы создадим частный репозиторий `flask-hello-world` в своей учетной записи `griggheo`. Далее в командной строке выполните команду `docker login` и укажите адрес электронной почты и пароль своей учетной записи. После этого вы сможете работать с Docker Hub через клиентское приложение `docker`.



Прежде чем попытаться опубликовать собранный локальной м-шине образ Docker в Docker Hub, мы хотели бы обратить внимание на то, что рекомендуется присваивать образам уникальные теги. Если не указать тег явно, по умолчанию образу будет присвоен `latest`. При помещении нового образа в реестр без тега `latest` будет перенесена эта последняя версия образа. Если при использовании образа Docker не указать точно нужный тег, будет использоваться версия `latest` образа, который может содержать модификации и обновления, нарушающие зависимости. Как и всегда, здесь применим принцип «наименьшего удивления»: теги желательнее действовать как при помещении образов в реестр, так и ссылаться на них в `Dockerfile`. Тем не менее вы можете пометить желаемую версию образа как `latest`, чтобы те, кого интересует самая свежая и лучшая версия, могли воспользоваться ею без указания тега.

В предыдущем разделе образ Docker автоматически получил при сборке тег `latest`, репозиторий — название образа, отмечая тот факт, что образ — локальный:

```
$ docker images hello-world-docker
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|--------------------|--------|--------------|---------------|--------|
| hello-world-docker | latest | dbd84c229002 | 2 minutes ago | 97.7MB |

Для задания тега образ Docker выполните команду `docker tag`:

```
$ docker tag hello-world-docker hello-world-docker:v1
```

Теперь вы увидите об тег для обр з hello-world-docker:

```
$ docker images hello-world-docker
REPOSITORY          TAG         IMAGE ID          CREATED           SIZE
hello-world-docker   latest      dbd84c229002     2 minutes ago    97.7MB
hello-world-docker   v1         89bd38cb198f     42 seconds ago   97.7MB
```

Прежде чем опубликов ть обр з hello-world-docker в Docker Hub, необходимо т кже м ркиров ть его н зв нием репозитория Docker Hub, содерж щего в ше имя пользо теля или н зв ние в шей орг низ ции. В н шем случ е репозито- рий н зыв ется griggheo/hello-world-docker:

```
$ docker tag hello-world-docker:latest griggheo/hello-world-docker:latest
$ docker tag hello-world-docker:v1 griggheo/hello-world-docker:v1
```

Теперь опубликуйте об тег обр з в Docker Hub с помощью ком нды docker push:

```
$ docker push griggheo/hello-world-docker:latest
$ docker push griggheo/hello-world-docker:v1
```

Если вы следили з ходом пример и выполняли все ком нды, то увидите свой обр з Docker опубликов нным с обоими тег ми в созд нном под в шей учетной з писью репозитории Docker Hub.

Запуск контейнера Docker из одного образа на другом хост-компьютере

Теперь, когд обр з Docker опубликов н в Docker Hub, мы готовы похв ст ть ся переносимостью обр зов Docker, з пустив основ нный н опубликов нном обр зе контейнер н другом хосте. Для этого мы р ссмотрим гипотетический сцен рий, в котором н м нужно р бот ть с сотрудником, у которого нет macOS и он р зр б тыв ет прогр ммы н ноутбуке под упр влением Fedora. Этот сцен - рий включ ет извлечение из хр нилищ код приложения и его модифик цию.

З пуск ем в AWS EC2-инст нс, в котором з действуется Linux 2 AMI, осно в нный н RedHat/CentOS/Fedora, после чего уст н влив ем движок Docker. Доб вьте пользо теля по умолч нию в Linux 2 AMI (ec2-user), в группу docker, чтобы он мог выполнять ком нды клиент docker:

```
$ sudo yum update -y
$ sudo amazon-linux-extras install docker
$ sudo service docker start
$ sudo usermod -a -G docker ec2-user
```

Не забудьте извлечь код приложения из удаленном EC2-инстансе. В данном случае код состоит из одного файла `app.py`.

Далее запустите контейнер Docker, основанный на опубликованном нами в Docker Hub образе. Единственное отличие — в качестве аргумента команды `docker run` используется образ `griggheo/hello-world-docker:v1`, а не просто `griggheo/hello-world-docker`.

Выполните команду `docker login`, с тем:

```
$ docker run --rm -d -v `pwd`:/app -p 5000:5000 griggheo/hello-world-docker:v1
```

```
Unable to find image 'griggheo/hello-world-docker:v1' locally
v1: Pulling from griggheo/hello-world-docker
921b31ab772b: Already exists
1a0c422ed526: Already exists
ec0818a7bbe4: Already exists
b53197ee35ff: Already exists
8b25717b4dbf: Already exists
d997915c3f9c: Pull complete
f1fd8d3cc5a4: Pull complete
10b64b1c3b21: Pull complete
Digest: sha256:af8b74f27a0506a0c4a30255f7ff563c9bf858735baa610fda2a2f638ccfe36d
Status: Downloaded newer image for griggheo/hello-world-docker:v1
9d67dc321ffb49e5e73a455bd80c55c5f09febc4f2d57112303d2b27c4c6da6a
```

Обратите внимание на то, что движок Docker на EC2-инстансе понимает, что образ Docker локальной машины нет, и скачивает его из Docker Hub, после чего запускает контейнер, основанный на только что скачанном образе.

На этом этапе был открыт доступ к порту 5000 посредством добавления правила в соответствующую этому EC2-инстансу группу безопасности. Зайдите на URL `http://54.187.189.51:5000`¹ (где `http://54.187.189.51` — внешний IP-адрес этого EC2-инстанса) — и вы увидите приветствие Hello, World! (from a Docker container with modified code).

При модификации кода приложения из удаленном EC2-инстансе запущенный внутри контейнера Docker сервер Flask автоматически перезагружает модифицированный код. Поменяйте приветствие на Hello, World! (from a Docker container on an EC2 Linux 2 AMI instance) — и вы увидите из журналов контейнера Docker, что сервер Flask перезагрузил приложение:

```
[ec2-user@ip-10-0-0-111 hello-world-docker]$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED
9d67dc321ffb   griggheo/hello-world-docker:v1     "python app.py"         3 minutes ago
```

¹ Это просто пример URL, у вас будет другой IP-адрес.


```

STATUS      PORTS      NAMES
Up 3 minutes 0.0.0.0:5000->5000/tcp  heuristic_roentgen

[ec2-user@ip-10-0-0-111 hello-world-docker]$ docker logs 9d67dc321ffb
* Serving Flask app "app" (lazy loading)
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 306-476-204
72.203.107.13 - - [19/Aug/2019 04:43:34] "GET / HTTP/1.1" 200 -
72.203.107.13 - - [19/Aug/2019 04:43:35] "GET /favicon.ico HTTP/1.1" 404 -
* Detected change in '/app/app.py', reloading
* Restarting with stat
* Debugger is active!
* Debugger PIN: 306-476-204

```

Теперь при з просе к <http://54.187.189.51:5000>¹ будет отобра ж ться новое приветствие — Hello, World! (from a Docker container on an EC2 Linux 2 AMI instance).

Отметим, что для з пуск приложения н м не пришлось уст н влив ть ничего относящегося к Python или Flask. Для использо вания возможностей переносимости Docker дост точно было просто з пустить приложение в контейнере. Не случ йно Docker выбр л н зв ние «контейнеры» для продвижения своей технологии — н это воодушевил пример грузовых тр нспортных контейнеров, которые произвели революцию в сфере грузоперевозок.



Нем ло ст тей об уп ковке приложений Python в контейнеры Docker имеется в коллекции «Созд ние подходящих для промышленной экспл у тации контейнеров Docker для р зр ботчиков н языке Python» (Production-ready Docker packaging for Python developers (<https://pythonspeed.com/docker>) Ит м р Тюрнер-Тр уринг (Itamar Turner-Trauring).

Запуск нескольких контейнеров Docker с помощью Docker Compose

В этом р зделе мы воспользуемся руководством «Flask в пример х» (*Flask By Example*, <https://oreil.ly/prNg7>), в котором описыв ется, к к созд ть приложение Flask для вычисления п р «слово — ч стотность» для текст , р сложенного по з д нному URL.

¹ И вновь в ш IP- дрес будет другим.

Начнем с клонирования репозитория «Flask в пример» с GitHub (<https://oreil.ly/M-pvc>):

```
$ git clone https://github.com/realpython/flask-by-example.git
```

Для запуска нескольких контейнеров Docker, соответствующих различным частям приложения, применим команду `compose`. При использовании `Compose` описание и настройка состоящих из приложения сервисов производится в файле `YAML`, после чего эти сервисы, работающие в контейнерах Docker, создаются, запускаются и останавливаются с помощью утилиты командной строки `docker-compose`.

Первая зависимость для нашего примера приложения — PostgreSQL, как описывается в части 2 этого руководства (<https://oreil.ly/iobKp>).

Вот как можно описать внутри файла `docker-compose.yaml` запуск PostgreSQL в контейнере Docker:

```
$ cat docker-compose.yaml
version: "3"
services:
  db:
    image: "postgres:11"
    container_name: "postgres"
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
volumes:
  dbdata:
```

Отметим несколько особенностей этого файла.

- В нем описывается сервис `db`, в основе которого лежит опубликованный в Docker Hub образ контейнера `postgres:11`.
- Задается соответствие локального порта `5432` порту `5432` контейнера.
- Задается том Docker для каталога, в котором PostgreSQL будет хранить свои данные, — `/var/lib/postgresql/data`. Благодаря этому хранящиеся в PostgreSQL данные не потеряются при перезапуске контейнера.

Утилита `docker-compose` не входит в состав движка Docker, так что ее необходимо установить отдельно. Инструкции по ее установке и различным операционным системам вы можете найти в официальной документации (<https://docs.docker.com/compose/install>).

Для загрузки описанного в файле `docker-compose.yaml` сервиса `db` выполните команду `docker-compose up -d db`, которая запустит в фоновом режиме (флаг `-d`) контейнер Docker для сервиса `db`¹:

```
$ docker-compose up -d db
Creating postgres ... done
```

Згляните в журналы сервиса `db` с помощью команды `docker-compose logs db`:

```
$ docker-compose logs db
Creating volume "flask-by-example_dbdata" with default driver
Pulling db (postgres:11)...
11: Pulling from library/postgres
Creating postgres ... done
Attaching to postgres
postgres | PostgreSQL init process complete; ready for start up.
postgres |
postgres | 2019-07-11 21:50:20.987 UTC [1]
LOG:  listening on IPv4 address "0.0.0.0", port 5432
postgres | 2019-07-11 21:50:20.987 UTC [1]
LOG:  listening on IPv6 address ":::", port 5432
postgres | 2019-07-11 21:50:20.993 UTC [1]
LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
postgres | 2019-07-11 21:50:21.009 UTC [51]
LOG:  database system was shut down at 2019-07-11 21:50:20 UTC
postgres | 2019-07-11 21:50:21.014 UTC [1]
LOG:  database system is ready to accept connections
```

При выполнении команды `docker ps` вы увидите контейнер, в котором запущен базис данных PostgreSQL:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED      STATUS      PORTS      NAMES
83b54ab10099   postgres:11 "docker-entrypoint.s..." 3 minutes ago Up 3 minutes
0.0.0.0:5432->5432/tcp    postgres
```

А выполнение команды `docker volume ls` продемонстрирует том `dbdata` Docker, смонтированный к каталогу `/var/lib/postgresql/data` PostgreSQL:

```
$ docker volume ls | grep dbdata
local          flask-by-example_dbdata
```

¹ Чтобы все работало, мне понадобилось также задать общий пароль для PostgreSQL либо не строить ее использование без пароля, что можно сделать, например, внося в вышеупомянутый файл `docker-compose.yaml` следующее:

```
environment:
  POSTGRES_HOST_AUTH_METHOD: "trust". — Примеч. пер.
```

Чтобы подключиться к базам данных PostgreSQL, запущенной в соответствующем сервису db контейнере Docker, выполните команду `docker-compose exec db, перед ней в командной строке опции psql -U postgres:`

```
$ docker-compose exec db psql -U postgres
psql (11.4 (Debian 11.4-1.pgdg90+1))
Type "help" for help.
```

```
postgres=#
```

Следуя «Flask в примерях, часть 2» (<https://oreil.ly/iobKp>), создаем базу данных wordcount:

```
$ docker-compose exec db psql -U postgres
psql (11.4 (Debian 11.4-1.pgdg90+1))
Type "help" for help.
```

```
postgres=# create database wordcount;
CREATE DATABASE
```

```
postgres=# \l
```

```

                        List of databases
  Name  | Owner   | Encoding | Collate  | Ctype    | Access privileges
-----+-----+-----+-----+-----+-----
postgres | postgres | UTF8     | en_US.utf8 | en_US.utf8 | 
template0 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres +
          |          |          |          |          | postgres=Ctc/postgres
template1 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres +
          |          |          |          |          | postgres=Ctc/postgres
wordcount | postgres | UTF8     | en_US.utf8 | en_US.utf8 | 
(4 rows)
postgres=# \q
```

Подключаемся к базам данных wordcount и создаем роль wordcount_dbadmin для использования нашим приложением Flask:

```
$ docker-compose exec db psql -U postgres wordcount
wordcount=# CREATE ROLE wordcount_dbadmin;
CREATE ROLE
wordcount=# ALTER ROLE wordcount_dbadmin LOGIN;
ALTER ROLE
wordcount=# ALTER USER wordcount_dbadmin PASSWORD 'MYPASS';
ALTER ROLE
postgres=# \q
```

Следующий этап — создание Dockerfile для уставки всего, что необходимо для нашего приложения Flask.

Внесите в файл `requirements.txt` следующие изменения:

- поменяйте версию пакет `psycopg2` с 2.6.1 на 2.7 для поддержки PostgreSQL 11;
- поменяйте версию пакет `redis` с 2.10.5 на 3.2.1 для улучшения поддержки Python 3.7;
- поменяйте версию пакет `rq` с 0.5.6 на 1.0 для улучшения поддержки Python 3.7.

Вот как выглядит теперь `Dockerfile`:

```
$ cat Dockerfile
FROM python:3.7.3-alpine

ENV APP_HOME /app
WORKDIR $APP_HOME

COPY requirements.txt .

RUN \
  apk add --no-cache postgresql-libs && \
  apk add --no-cache --virtual .build-deps gcc musl-dev postgresql-dev && \
  python3 -m pip install -r requirements.txt --no-cache-dir && \
  apk --purge del .build-deps

COPY . .

ENTRYPOINT [ "python" ]
CMD [ "app.py" ]
```



Между этим `Dockerfile` и использованной в первом примере версией `hello-world-docker` есть важное различие. Здесь содержимое текущего каталога, в котором находятся файлы приложения, копируется в образ Docker, чтобы проиллюстрировать сценарий, отличный от технологического процесса развёртки, показанного ранее. В данном случае нас больше интересует максимально переносимый вариант запуска приложения, например, в среде предэксплуатационного тестирования или промышленной эксплуатации, где не нужно модифицировать файлы приложения через смонтированные тома, как в сценарии развёртки. Для целей развёртки `docker-compose` можно использовать сомонтированными локально томами, но здесь нас интересует переносимость контейнеров Docker между различными средами, например развёртки, предэксплуатационного тестирования и промышленной эксплуатации.

Выполните команду `docker build -t flask-by-example:v1 .` для сборки локального образа Docker. Выводимые этой командой результаты приводить не стоит из-за их большого объема.

Следующий шаг из руководств «Flask в примерах» — запуск миграций Flask.

Опишите в файле `docker-compose.yaml` новый сервис `migrations`, задайте для него `image`, `command`, переменные среды (`environment`) и укажите, что для него требуется, чтобы сервис `db` был запущен и работал:

```
$ cat docker-compose.yaml
version: "3"
services:
  migrations:
    image: "flask-by-example:v1"
    command: "manage.py db upgrade"
    environment:
      APP_SETTINGS: config.ProductionConfig
      DATABASE_URL: postgresql://wordcount_dbadmin:$DBPASS@db/wordcount
    depends_on:
      - db
  db:
    image: "postgres:11"
    container_name: "postgres"
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
volumes:
  dbdata:
```

В переменной `DATABASE_URL` в качестве хоста для баз данных PostgreSQL указано не `db`, поскольку именно оно задано в качестве имени сервиса в файле `docker-compose.yaml` и утилита `docker-compose` умеет связывать одно с другим посредством создания иложенной сети, в которой все описанные в файле `docker-compose.yaml` сервисы могут взаимодействовать друг с другом по именам. См. подробности в справочном руководстве по утилите `docker-compose` (<https://oreil.ly/Io80N>).

Определение переменной `DATABASE_URL` ссылается на другую переменную — `DBPASS`, вместо того чтобы жестко «зашифровать» роль пользователя `wordcount_dbadmin`. Обычно файл `docker-compose.yaml` вносится в систему контроля версий, и рекомендуется не отправлять в GitHub секретные данные, такие как учетные данные БД. Вместо этого для работы с файлами, содержащими секретные данные, стоит использовать утилиты шифрования, например `sops` (<https://github.com/mozilla/sops>).

Вот пример создания с помощью `sops` зашифрованного посредством PGP файла.

Вначале установите `gpg` на macOS с помощью команды `brew install gpg`, после чего сгенерируйте новый ключ PGP с пустой ролью:

```
$ gpg --generate-key
pub  rsa2048 2019-07-12 [SC] [expires: 2021-07-11]
     E14104A0890994B9AC9C9F6782C1FF5E679EFF32
uid                          pydevops <my.email@gmail.com>
sub  rsa2048 2019-07-12 [E] [expires: 2021-07-11]
```

Далее скачайте с веб-страницы пакета `sops` с его выпуском (https://github.com/mozilla/sops/releases).

Для создания нового зашифрованного файла с названием, например, `environment.secrets` запустите утилиту `sops` с флагом `-pgp` и передайте ей сигнатуру сгенерированного ранее ключа:

```
$ sops --pgp BBDE7E57E00B98B3F4FBEAF21A1EEF4263996BD0 environment.secrets
```

В результате будет открыт текстовый редактор по умолчанию, в который можно будет ввести секретные данные в виде открытого текста. В этом примере файл `environment.secrets` содержит следующее:

```
export DBPASS=MYPASS
```

Сохранив файл `environment.secrets`, проинспектируйте его, чтобы убедиться, что он зашифрован и его можно спокойно вносить в систему управления исходными кодами:

```
$ cat environment.secrets
{
  "data": "ENC[AES256_GCM,data:q1Q5zc7e8KgGmu5goC9WmE7PP8gueBoSsmM=,
  iv:xG8BHCrdfdLpH9nUlTijBsYrh4TuSdvDqp5F+2Hqw4I=,
  tag:00IVAm90/UyG1jGcZerTQ==,type:str]",
  "sops": {
    "kms": null,
    "gcp_kms": null,
    "lastmodified": "2019-07-12T05:03:45Z",
    "mac": "ENC[AES256_GCM,data:wo+zPVbPbAJt9N123nYuWs55f68/DZJWj3pc0
18T2d/SbuRF6YCuOXHSHIKs1ZBpSlSjmIrPyYTqI+M4Wf7it7fnNS8b7Fnc1wmXJjptBWgLT/A1GzIKT1Vrgw9QgJ+prq+Qcrk5dPzhsOTxOoOhGRPsyN8KjkS4sGuXM=,iv:0VvSMgjF6
ypcK+1J54fonRoI7c5whmcu3iNV8xLH02k=,
    tag:YaI7DXvllvpJ3Talzl8lg==,
    type:str]",
    "pgp": [
      {
        "created_at": "2019-07-12T05:02:24Z",
        "enc": "-----BEGIN PGP MESSAGE-----\n\nhQEMA+3cyc
g5b/Hu00vU5ONr/F0htZM2MZQ5XpxoCi0\nWGB5Czc8FTS1RSwu8/c0x0Ch1FwH+IdLwL+jd
oXVe55myuu/3OKUy7H1w/W2R\nPI99Biw1m5u3ir3+9tLXmRpLWkz7+nX7FTh19QnOS25
NRUSSxS7hNaZMcYjpxW+w\nM3XeaGStgbJ9OgIp4A8YGigZQVZZF13fAG3bm2c+TNJcAb1
zDpc40fxlR+7LroJI\njuidzy0Ee49k0pq3tzqCnph5wPr3HZ1JeQmsIquf//9D509S5xH
```

```

    Sa9lkz3Y7V4KC\nefzBiS8pivm55T0s+zPBPB/GWUVlqGaxRhv1TAU=\n=WA4+
    \n-----END PGP MESSAGE-----\n",
    "fp": "E14104A0890994B9AC9C9F6782C1FF5E679EFF32"
  }
],
"unencrypted_suffix": "_unencrypted",
"version": "3.0.5"
}
}%

```

Для его р-шифровки выполните:

```

$ sops -d environment.secrets
export DBPASS=MYPASS

```



Существует одна небольшая проблема, связанная с взаимодействием sops с gpg на компьютерах Macintosh. Перед р-шифровкой файла с помощью sops необходимо выполнить следующие команды:

```

$ GPG_TTY=$(tty)
$ export GPG_TTY

```

Наша цель — запустить сервис migrations, описанный ранее в файле docker-compose.yaml. Свяжем метод управления секретными данными sops с docker-compose, р-шифровав файл environments.secrets с помощью sops -d, отправив с помощью команды source его содержимое в виртуальную среду текущей командной оболочки, после чего вызовем docker-compose up -d migrations, и все это — в однострочной команде, чтобы секретные данные не отображались в истории командной оболочки:

```

$ source <(sops -d environment.secrets); docker-compose up -d migrations
postgres is up-to-date
Recreating flask-by-example_migrations_1 ... done

```

Проверяем, что миграция выполнена успешно, взглянув в базу данных и убедившись, что были созданы две таблицы — alembic_version и results:

```

$ docker-compose exec db psql -U postgres wordcount
psql (11.4 (Debian 11.4-1.pgdg90+1))
Type "help" for help.

```

```
wordcount=# \dt
```

```

              List of relations
Schema |      Name      | Type |      Owner
-----+-----+-----+-----
public | alembic_version | table | wordcount_dbadmin
public | results         | table | wordcount_dbadmin
(2 rows)

```

```
wordcount=# \q
```


Часть 4 (<https://oreil.ly/Uy2yw>) руководств «Flask в примерах» описывает развертывание процесса-исполнителя Python на основе Python RQ, взаимодействующего с экземпляром хранилища данных Redis.

Прежде всего необходимо запустить Redis. Добавьте его описание к сервису `redis` в файл `docker-compose.yml` и убедитесь, что его внутренний порт 6379 переносится на порт 6379 локальной операционной системы:

```
redis:
  image: "redis:alpine"
  ports:
    - "6379:6379"
```

Запустите сервис `redis`, указав его в качестве аргумента команды `docker-compose up -d`:

```
$ docker-compose up -d redis
Starting flask-by-example_redis_1 ... done
```

Выполните команду `docker ps`, чтобы посмотреть на новый контейнер Docker, основанный на образе `redis:alpine`:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS          NAMES
a1555cc372d6   redis:alpine "docker-entrypoint.s..." 3 seconds ago Up 1 second   0.0.0.0:6379->6379/tcp flask-by-example_redis_1
83b54ab10099   postgres:11 "docker-entrypoint.s..." 22 hours ago  Up 16 hours   0.0.0.0:5432->5432/tcp postgres
```

Просмотрите журналы сервиса `redis` с помощью команды `docker-compose logs`:

```
$ docker-compose logs redis
Attaching to flask-by-example_redis_1
1:C 12 Jul 2019 20:17:12.966 # 000000000000 Redis запускается 000000000000
1:C 12 Jul 2019 20:17:12.966 # Redis version=5.0.5, bits=64, commit=00000000,
modified=0, pid=1, just started
1:C 12 Jul 2019 20:17:12.966 # Предупреждение: файл конфигурации не указан,
используем настройки по умолчанию. Для указания файла конфигурации примените
команду redis-server /path/to/redis.conf
1:M 12 Jul 2019 20:17:12.967 * Запускаем mode=standalone, port=6379.
1:M 12 Jul 2019 20:17:12.967 # Предупреждение: невозможно воплотить значение
511 настройки очереди соединений TCP, поскольку параметр /proc/sys/net/core/
somaxconn установлен в более низкое значение 128.
1:M 12 Jul 2019 20:17:12.967 # Сервер инициализирован
1:M 12 Jul 2019 20:17:12.967 * Готов к приему соединений
```

Следующий шаг — создание сервиса `worker` для процесса-исполнителя на основе Python RQ в файле `docker-compose.yml`:

```
worker:
  image: "flask-by-example:v1"
```

```

command: "worker.py"
environment:
  APP_SETTINGS: config.ProductionConfig
  DATABASE_URL: postgresql://wordcount_dbadmin:$DBPASS@db/wordcount
  REDISTOGO_URL: redis://redis:6379
depends_on:
  - db
  - redis

```

Зпустите сервис `worker` и логично сервису `redis` с помощью команды `docker-compose up -d`:

```

$ docker-compose up -d worker
flask-by-example_redis_1 is up-to-date
Starting flask-by-example_worker_1 ... done

```

Если выполнить команду `docker ps`, будет отображен контейнер `worker`:

```

$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED        STATUS        PORTS          NAMES
72327ab33073   flask-by-example  "python worker.py"      8 minutes ago Up 14 seconds flask-by-example_worker_1
b11b03a5bcc3   redis:alpine      "docker-entrypoint.s..." 15 minutes ago Up About a minute 0.0.0.0:6379->6379/tcp flask-by-example_redis_1
83b54ab10099   postgres:11       "docker-entrypoint.s..." 23 hours ago   Up 17 hours    0.0.0.0:5432->5432/tcp postgres

```

Просмотрите журналы сервиса `worker` с помощью команды `docker-compose logs`:

```

$ docker-compose logs worker
Attaching to flask-by-example_worker_1
20:46:34 RQ worker 'rq:worker:a66ca38275a14cac86c9b353e946a72e' started,
version 1.0
20:46:34 *** Listening on default...
20:46:34 Cleaning registries for queue: default

```

А теперь зпустите основное приложение Flask в его собственном контейнере. Создайте новый сервис `app` в файле `docker-compose.yaml`:

```

app:
  image: "flask-by-example:v1"
  command: "manage.py runserver --host=0.0.0.0"
  ports:
    - "5000:5000"
  environment:
    APP_SETTINGS: config.ProductionConfig
    DATABASE_URL: postgresql://wordcount_dbadmin:$DBPASS@db/wordcount
    REDISTOGO_URL: redis://redis:6379
  depends_on:
    - db
    - redis

```

Переносим порт 5000 контейнера приложения (порт по умолчанию для приложений Flask) на порт 5000 локальной машины. Перед тем как запускать контейнер приложения, команду `manage.py runserver --host=0.0.0.0`, чтобы гарантировать, что порт 5000 виден приложению Flask должным образом внутри контейнера.

Затем запускаем сервис `app` с помощью команды `docker-compose up -d`, выполняя в то же время `sops -d` для содержимого `DBPASS` из зашифрованного файла, затем перед вызовом `docker-compose` отпрываем с помощью команды `source` содержимое зашифрованного файла в виртуальную среду:

```
source <(sops -d environment.secrets); docker-compose up -d app
postgres is up-to-date
Recreating flask-by-example_app_1 ... done
```

В возвращаемом командой `docker ps` списке находим новый контейнер Docker, в котором работает наше приложение:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED    STATUS    PORTS          NAMES
d99168a152f1   flask-by-example  "python app.py"         3 seconds ago
Up 2 seconds   0.0.0.0:5000->5000/tcp  flask-by-example_app_1
72327ab33073   flask-by-example  "python worker.py"      16 minutes ago
Up 7 minutes   flask-by-example_worker_1
b11b03a5bcc3   redis:alpine      "docker-entrypoint.s..." 23 minutes ago
Up 9 minutes   0.0.0.0:6379->6379/tcp  flask-by-example_redis_1
83b54ab10099   postgres:11       "docker-entrypoint.s..." 23 hours ago
Up 17 hours    0.0.0.0:5432->5432/tcp  postgres
```

Просмотрим журналы контейнера приложения с помощью команды `docker-compose logs`:

```
$ docker-compose logs app
Attaching to flask-by-example_app_1
app_1      | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Выполнение команды `docker-compose logs` без аргументов позволяет просмотреть журналы всех сервисов, описанных в файле `docker-compose.yaml`:

```
$ docker-compose logs
Attaching to flask-by-example_app_1,
flask-by-example_worker_1,
flask-by-example_migrations_1,
flask-by-example_redis_1,
postgres
1:C 12 Jul 2019 20:17:12.966 # 000000000000 Redis запускается 000000000000
1:C 12 Jul 2019 20:17:12.966 # Redis version=5.0.5, bits=64, commit=00000000,
modified=0, pid=1, just started
1:C 12 Jul 2019 20:17:12.966 # Предупреждение: файл конфигурации не указан,
используем настройки по умолчанию. Для указания файла конфигурации примените
команду redis-server /path/to/redis.conf
```

```

1:M 12 Jul 2019 20:17:12.967 * Запускаем mode=standalone, port=6379.
1:M 12 Jul 2019 20:17:12.967 # Предупреждение: невозможно воплотить значение
511 настройки очереди соединений TCP, поскольку параметр /proc/sys/net/core/
somaxconn установлен в более низкое значение 128.
1:M 12 Jul 2019 20:17:12.967 # Сервер инициализирован
1:M 12 Jul 2019 20:17:12.967 * Готов к приему соединений
app_1          | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
postgres       | 2019-07-12 22:15:19.193 UTC [1]
LOG:  listening on IPv4 address "0.0.0.0", port 5432
postgres       | 2019-07-12 22:15:19.194 UTC [1]
LOG:  listening on IPv6 address ":::", port 5432
postgres       | 2019-07-12 22:15:19.199 UTC [1]
LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
postgres       | 2019-07-12 22:15:19.214 UTC [22]
LOG:  database system was shut down at 2019-07-12 22:15:09 UTC
postgres       | 2019-07-12 22:15:19.225 UTC [1]
LOG:  database system is ready to accept connections
migrations_1   | INFO [alembic.runtime.migration] Context impl PostgresqlImpl.
migrations_1   | INFO [alembic.runtime.migration] Will assume transactional DDL.
worker_1       | 22:15:20
RQ worker 'rq:worker:2edb6a54f30a4aae8a8ca2f4a9850303' started, version 1.0
worker_1       | 22:15:20 *** Listening on default...
worker_1       | 22:15:20 Cleaning registries for queue: default

```

Последний этап — тестирование приложения. Зайдите в браузер по адресу `http://127.0.0.1:5000` и введите `python.org` в поле URL. При этом приложение отправит процессу-исполнителю задание на выполнение функции `count_and_save_words` для домашней страницы с `python.org`. Приложение будет периодически приходить к заданию результатов, по завершении отобразит домашней странице количество слов.

Чтобы повысить переносимость файла `docker-compose.yaml`, помещем образ `Docker flask-by-example` в Docker Hub и ссылаемся на него в репозитории сервисов `app` и `worker` файла `docker-compose.yaml`.

Помечем локальный образ `Docker flask-by-example:v1` тегом, состоящим из имени, которому предшествует имя пользователя Docker Hub, после чего помещем образ, только что получивший тег, в Docker Hub:

```

$ docker tag flask-by-example:v1 griggheo/flask-by-example:v1
$ docker push griggheo/flask-by-example:v1

```

Вносим в файл `docker-compose.yaml` изменения, ссылаясь на новый образ Docker. Вот окончательная версия файла `docker-compose.yaml`:

```

$ cat docker-compose.yaml
version: "3"
services:
  app:

```

```

image: "griggheo/flask-by-example:v1"
command: "manage.py runserver --host=0.0.0.0"
ports:
  - "5000:5000"
environment:
  APP_SETTINGS: config.ProductionConfig
  DATABASE_URL: postgresql://wordcount_dbadmin:$DBPASS@db/wordcount
  REDISTOGO_URL: redis://redis:6379
depends_on:
  - db
  - redis
worker:
image: "griggheo/flask-by-example:v1"
command: "worker.py"
environment:
  APP_SETTINGS: config.ProductionConfig
  DATABASE_URL: postgresql://wordcount_dbadmin:$DBPASS@db/wordcount
  REDISTOGO_URL: redis://redis:6379
depends_on:
  - db
  - redis
migrations:
image: "griggheo/flask-by-example:v1"
command: "manage.py db upgrade"
environment:
  APP_SETTINGS: config.ProductionConfig
  DATABASE_URL: postgresql://wordcount_dbadmin:$DBPASS@db/wordcount
depends_on:
  - db
db:
image: "postgres:11"
container_name: "postgres"
ports:
  - "5432:5432"
volumes:
  - dbdata:/var/lib/postgresql/data
redis:
image: "redis:alpine"
ports:
  - "6379:6379"
volumes:
  dbdata:

```

Для перезапуска локальных контейнеров Docker выполните команду `docker-compose down`, затем `docker-compose up -d`:

```

$ docker-compose down
Stopping flask-by-example_worker_1 ... done
Stopping flask-by-example_app_1   ... done
Stopping flask-by-example_redis_1 ... done
Stopping postgres                  ... done

```

```

Removing flask-by-example_worker_1    ... done
Removing flask-by-example_app_1       ... done
Removing flask-by-example_migrations_1 ... done
Removing flask-by-example_redis_1     ... done
Removing postgres                     ... done
Removing network flask-by-example_default

$ source <(sops -d environment.secrets); docker-compose up -d
Creating network "flask-by-example_default" with the default driver
Creating flask-by-example_redis_1      ... done
Creating postgres                     ... done
Creating flask-by-example_migrations_1 ... done
Creating flask-by-example_worker_1    ... done
Creating flask-by-example_app_1       ... done

```

Обратите внимание на то, как просто теперь становится запустить набор контейнеров Docker с помощью `docker-compose`.



Даже если вам нужен всего один контейнер Docker, все равно имеет смысл включить его в файл `docker-compose.yml` и запустить с помощью команды `docker-compose up -d`. Это облегчит задачу, если вы захотите добавить второй контейнер, так же послужит маленьким примером инфраструктуры к коду, в котором `docker-compose.yml` будет отражать состояние локальной схемы организации Docker для вашего приложения.

Портирование сервисов `docker-compose` на новый хост-компьютер и операционную систему

А сейчас попробуем, как перенести организацию и структуру `docker-compose` из предыдущего раздела на сервер под управлением Ubuntu 18.04.

Запустите Amazon EC2-инстанс с Ubuntu 18.04 и установите там `docker-engine` и `docker-compose`:

```

$ sudo apt-get update
$ sudo apt-get remove docker docker-engine docker.io containerd runc
$ sudo apt-get install \
  apt-transport-https \
  ca-certificates \
  curl \
  gnupg-agent \
  software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo add-apt-repository \

```

```
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
$ sudo usermod -a -G docker ubuntu

# Скачиваем docker-compose
$ sudo curl -L \
"https://github.com/docker/compose/releases/download/1.24.1/docker-compose-"
$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

Скопируйте в этот удаленный EC2-инстанс файл `docker-compose.yaml` и сделайте так, чтобы сервис `db`, чтобы можно было создать базу данных для приложения:

```
$ docker-compose up -d db
Starting postgres ...
Starting postgres ... done
```

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED    STATUS    PORTS    NAMES
49fe88efdb45   postgres:11 "docker-entrypoint.s..." 29 seconds ago
Up 3 seconds   0.0.0.0:5432->5432/tcp    postgres
```

Воспользуемся `docker exec` для выполнения команды `psql -U postgres` внутри работающего контейнера Docker для базы данных PostgreSQL. После появления приглашения PostgreSQL создайте базу данных `wordcount` и роль `wordcount_dbadmin`:

```
$ docker-compose exec db psql -U postgres
psql (11.4 (Debian 11.4-1.pgdg90+1))
Type "help" for help.
```

```
postgres=# create database wordcount;
CREATE DATABASE
postgres=# \q
```

```
$ docker exec -it 49fe88efdb45 psql -U postgres wordcount
psql (11.4 (Debian 11.4-1.pgdg90+1))
Type "help" for help.
```

```
wordcount=# CREATE ROLE wordcount_dbadmin;
CREATE ROLE
wordcount=# ALTER ROLE wordcount_dbadmin LOGIN;
ALTER ROLE
wordcount=# ALTER USER wordcount_dbadmin PASSWORD 'MYPASS';
ALTER ROLE
wordcount=# \q
```

Перед запуском контейнеров для сервисов, описанных в файле `docker-compose.yaml`, мы пойдём и сделаем две вещи.

1. Выполнить команду `docker login`, чтобы извлечь из Docker Hub помещённый туда образ Docker:

```
$ docker login
```

2. Задать переменную среды `DBPASS` в текущей командной оболочке. Можно воспользоваться описанным в предыдущем разделе методом `sops`, но в этом примере мы зададим её непосредственно в командной оболочке:

```
$ export DOCKER_PASS=MYPASS
```

Теперь запустите все необходимые для приложения сервисы с помощью команды `docker-compose up -d`:

```
$ docker-compose up -d
Pulling worker (griggheo/flask-by-example:v1)...
v1: Pulling from griggheo/flask-by-example
921b31ab772b: Already exists
1a0c422ed526: Already exists
ec0818a7bbe4: Already exists
b53197ee35ff: Already exists
8b25717b4dbf: Already exists
9be5e85cacbb: Pull complete
bd62f980b08d: Pull complete
9a89f908ad0a: Pull complete
d787e00a01aa: Pull complete
Digest: sha256:4fc554da6157b394b4a012943b649ec66c999b2accb839562e89e34b7180e3e
Status: Downloaded newer image for griggheo/flask-by-example:v1
Creating fbe_redis_1      ... done
Creating postgres        ... done
Creating fbe_migrations_1 ... done
Creating fbe_app_1       ... done
Creating fbe_worker_1    ... done
```

```
$ docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|------------------------------|--------------------------|---------------|--------------|------------------------|--------------|
| f65fe9631d44 | griggheo/flask-by-example:v1 | "python3 manage.py r..." | 5 seconds ago | Up 2 seconds | 0.0.0.0:5000->5000/tcp | fbe_app_1 |
| 71fc0b24bce3 | griggheo/flask-by-example:v1 | "python3 worker.py" | 5 seconds ago | Up 2 seconds | | fbe_worker_1 |
| a66d75a20a2d | redis:alpine | "docker-entrypoint.s..." | 7 seconds ago | Up 5 seconds | 0.0.0.0:6379->6379/tcp | fbe_redis_1 |
| 56ff97067637 | postgres:11 | "docker-entrypoint.s..." | 7 seconds ago | Up 5 seconds | 0.0.0.0:5432->5432/tcp | postgres |

Теперь, предвзрительно открыв доступ к порту 5000 в группе безопасности AWS, соответствующей ншению EC2-инстнсу подуправлением Ubuntu, можете выполнить запрос к внешнему IP-адресу инстнса на порте 5000 и использовать свое приложение.

Стоит еще раз отметить, насколько Docker упрощает развертывание приложений. Переносимость контейнеров и образов Docker означает возможность выполнения приложения на любой операционной системе, где только может работать движок Docker. В приведенном здесь примере не нужно устанавливать на сервере Ubuntu никаких приложений из списка требований: ни Flask, ни PostgreSQL, ни Redis. Не нужно также копировать код приложения с локальной машины на сервер Ubuntu. Единственный файл, который должен присутствовать на сервере Ubuntu, — `docker-compose.yaml`. И весь набор состоящих приложение сервисов мы запустили с помощью всего одной команды:

```
$ docker-compose up -d
```



Будьте осторожны при скачивании и использовании образов Docker из общедоступных репозиториях Docker, поскольку многие из них содержат опасные уязвимости, и более серьезные из которых позволяют злоумышленникам проникнуть сквозь изоляцию контейнера Docker и захватить управление операционной системой хост-компьютера. Рекомендуем являться — не чуждым с заслуживающего доверия из ранее собранного образа или собрать собственный образ с нуля. Поддерживайте актуальность образа в смысле последних обновлений безопасности и обновлений ПО, пересобирайте образ при всяком появлении таких обновлений. Еще одна полезная практика — сканировать все свои образы Docker с помощью одного из множества доступных сканеров уязвимостей, и пример Clair (<https://oreil.ly/OBkxk>), Anchore (https://oreil.ly/uRI_1) или Falco (<https://oreil.ly/QXRg6>). Такое сканирование можно включить в состав конвейера непрерывной интеграции/непрерывного развертывания, в ходе которого обычно собираются образы Docker.

Хотя утилита `docker-compose` сильно упрощает запуск нескольких контейнеризованных сервисов вместо запуска одного приложения, она предназначена для выполнения на одной машине, так что польза от нее при промышленной эксплуатации ограничена. Развернутое с помощью `docker-compose` приложение можно считать готовым к промышленной эксплуатации только в том случае, если вас не волнуют возможные простои и вы готовы выполнять все на одной машине (впрочем, Григу приходилось видеть постыщиков хостинга, запускающих с помощью `docker-compose` контейнеризованные Docker приложения в промышленной эксплуатации). Для полностью готовых к промышленной эксплуатации систем необходим механизм координации, и пример Kubernetes, который мы обсудим в следующей главе.

Упражнения

- Ознакомьтесь со справочным руководством по Dockerfile (<https://oreil.ly/kA8ZF>).
- Ознакомьтесь со справочным руководством по настройке Docker Compose (<https://oreil.ly/ENMsQ>).
- Создайте KMS-ключ в AWS и используйте его с `sops` вместо локального PGP-ключа. Благодаря этому вы сможете применять к этому ключу приватный доступ AWS IAM и предоставлять доступ к нему только тем ролям, которым он действительно нужен.
- Напишите сценарий командной оболочки, использующий `docker exec` или `docker-compose exec` для выполнения команд PostgreSQL, необходимых для создания баз данных и роли.
- Поэкспериментируйте с другими технологиями контейнеров, например Podman (<https://podman.io>).

Координация работы контейнеров: Kubernetes

Для экспериментов с Docker или запуск и сбор контейнеров Docker не отдельной м/шине вполне достаточно Docker и Docker Compose. Однако при переходе от одной м/шины к нескольким в м/шине придется задуматься о координации контейнеров в сети. А при промышленной эксплуатации без этого не обойтись: для обеспечения отказоустойчивости/высокой доступности необходимы по крайней мере две м/шины.

В нашу эпоху облачных вычислений рекомендуется горизонтальное масштабирование посредством добавления новых виртуальных узлов, а не вертикальное с помощью внесения дополнительных CPU и памяти в отдельный виртуальный узел. В платформе координации Docker эти виртуальные узлы используются как источники исходных ресурсов (CPU, оперативной памяти, сетевых ресурсов), выделяемые далее отдельным контейнерам, работающим на этой платформе. Это вполне соответствует сказанному в главе 11 о преимуществе контейнеров над традиционными виртуальными м/шинами: имеющиеся исходные ресурсы при этом используются эффективнее, поскольку контейнерам можно выделять их намного избирательнее, чем виртуальным м/шинам, и отделить от каждого затратного и инфраструктуру доллится.

Существует также тенденция к переходу от выделения серверов для конкретных нужд и выполнения на каждом из виртуальных узлов конкретных пакетов ПО (например, ПО веб-сервер, ПО для кэширования и ПО базы данных) к выделению их в виде универсальных единиц ресурсов и выполнения на них контейнеров Docker, координируемых платформой координации Docker. Возможно, вы знакомы с концепцией, согласно которой серверы можно рассматривать как домашние питомцы, можно — как крупный рогатый скот. В период становления проектирования инфраструктуры каждого сервера была четко определена функция (например, сервер электронной почты) и нередко для

каждой конкретной функции предначислялся только один сервер. Для подобных серверов существовали схемы именования (Григ помнит, как во временоткомов использовал схему именования по названию планеты), и не «уходз ними» затиравлось немло времени, отсюда и сравнение с домашними питомцами. Когда рынок ворвался утилиты управления конфигурацией — подобие Puppet, Chef и Ansible, выделять по несколько серверов одного типа (например, целую ферму веб-серверов) стало гораздо проще благодаря использованию одного кодового процесса уставки на всех серверах. Это совпало с ростом популярности облачных вычислений с вышеупомянутой концепцией горизонтального масштабирования, тем же с тем, что благодаря устойчивости и высокой доступности несли относиться к конкретнейшим свойствам хорошо спроектированной инфраструктуры системы. Серверы/облачные виртуальные узлы считались уже «крупным рогатым скотом» — однородными единицами, ценными в своей совокупности.

Эпоха контейнеров и бессерверных вычислений повлекла за собой появление еще одного сравнения — с насекомыми. И действительно, жизненный цикл контейнера потенциально очень короток, как у насекомых-однодневок. Длительность существования функций как сервисов еще меньше, чем контейнеров Docker, их краткая, но насыщенная жизнь совпадает с длительностью их вызовов.

Вследствие краткости жизненного цикла контейнеров добиться их координации и взаимодействия в крупных масштабах весьма непросто. Именно эту нишу и заполнили платформы координации контейнеров. Существует несколько платформ координации Docker, включая Mesosphere и Docker Swarm, но на сегодняшний день можно с уверенностью сказать, что Kubernetes бесспорный победитель. Дальнейший текст главы посвящен краткому обзору Kubernetes, с которым следует пример запуска описанного в главе 11 приложения и портирования его с `docker-compose` на Kubernetes. Мы также продемонстрируем уставку пкетоветкнзываемых чртот (charts) Helm — системы управления пкетами Kubernetes — для утилит Prometheus и Grafana мониторинга и отображения инструментальных данных, тем же настройку этих чртот под свои нужды.

Краткий обзор основных понятий Kubernetes

Изучение множеств составляющих кластер Kubernetes лучше всего начать с официальной документации Kubernetes (<https://oreil.ly/TYpdE>).

В общих чертах кластер Kubernetes состоит из узлов, которые можно считать серверами, неважно, представляют ли они собой физические серверы или работающие в облаке виртуальные машины. В узлах работают модули — компоненты

контейнеров Docker. Модуль — единица развертывания в Kubernetes. Все контейнеры в модуле работают в единой сети и могут обращаться друг к другу так, как будто запущены на одном хосте. Во многих ситуациях выгодно запускать в модуле несколько контейнеров, а не только один. Обычно контейнер приложения играет роль основного контейнера модуля, но при необходимости можно запустить еще один или несколько дополнительных вспомогательных контейнеров (sidecar containers) для определенной функциональности, например журналирования или мониторинга. Одним из разновидностей вспомогательных контейнеров — контейнеры инициализации (init containers), которые заведомо запускаются первыми и могут применяться для различных задач обслуживания системы, например миграции баз данных. Мы рассмотрим их подробнее далее в этой главе.

Для повышения производительности и отказоустойчивости приложения обычно используют более одного модуля. Объект Kubernetes, отвечающий за запуск и поддержание работы нужного количества модулей, называется Deployment (развертывание). Взаимодействие модулей между собой реализуется другим видом объектов Kubernetes — Service (сервис). Сервисы связаны с развертываниями посредством селекторов. Сервисы также доступны для внешних клиентов либо через статические порты на всех узлах Kubernetes путем указания типа NodePort, либо посредством создания объекта LoadBalancer, соответствующего фактически балансировщику нагрузки, если таковой поддерживается поставщиком облака, в котором работает кластер Kubernetes.

Для работы с конфиденциальной информацией — паролями, ключами API и прочими учетными данными — Kubernetes предоставляет объект Secret. Мы рассмотрим пример хранения с помощью объекта Secret пароля к базе данных.

Создание манифестов Kubernetes на основе файла `docker_compose.yaml` с помощью Kompose

Еще раз взглянем на файл `docker_compose.yaml` из примера приложения Flask, которое мы обсуждали в главе 11:

```
$ cat docker-compose.yaml
version: "3"
services:
  app:
    image: "griggheo/flask-by-example:v1"
    command: "manage.py runserver --host=0.0.0.0"
    ports:
      - "5000:5000"
```

```

environment:
  APP_SETTINGS: config.ProductionConfig
  DATABASE_URL: postgresql://wordcount_dbadmin:$DBPASS@db/wordcount
  REDISTOGO_URL: redis://redis:6379
depends_on:
  - db
  - redis
worker:
  image: "griggheo/flask-by-example:v1"
  command: "worker.py"
  environment:
    APP_SETTINGS: config.ProductionConfig
    DATABASE_URL: postgresql://wordcount_dbadmin:$DBPASS@db/wordcount
    REDISTOGO_URL: redis://redis:6379
  depends_on:
    - db
    - redis
migrations:
  image: "griggheo/flask-by-example:v1"
  command: "manage.py db upgrade"
  environment:
    APP_SETTINGS: config.ProductionConfig
    DATABASE_URL: postgresql://wordcount_dbadmin:$DBPASS@db/wordcount
  depends_on:
    - db
db:
  image: "postgres:11"
  container_name: "postgres"
  ports:
    - "5432:5432"
  volumes:
    - dbdata:/var/lib/postgresql/data
redis:
  image: "redis:alpine"
  ports:
    - "6379:6379"
volumes:
  dbdata:

```

Воспользуемся утилитой `Kompose` для преобразования этого файла в формате YAML в набор манифестов Kubernetes.

Для установки свежей версии `Kompose` на машине под управлением macOS скачайте ее из репозитория Git (<https://oreil.ly/GUqaa>), переместите скаченный файл в каталог `/usr/local/bin/kompose` и сделайте его исполняемым. Обратите внимание на то, что при использовании для установки `Kompose` системы управления пакетной операционной системы (например, `apt` в Ubuntu или `yum` в Red Hat) вы рискуете получить не много более старую версию, возможно несовместимую с приведенными далее инструкциями.

Выполните команду `kompose convert` для создания файлов манифестов Kubernetes на основе имеющегося файла `docker_compose.yaml`:

```
$ kompose convert
INFO Kubernetes file "app-service.yaml" created
INFO Kubernetes file "db-service.yaml" created
INFO Kubernetes file "redis-service.yaml" created
INFO Kubernetes file "app-deployment.yaml" created
INFO Kubernetes file "db-deployment.yaml" created
INFO Kubernetes file "dbdata-persistentvolumeclaim.yaml" created
INFO Kubernetes file "migrations-deployment.yaml" created
INFO Kubernetes file "redis-deployment.yaml" created
INFO Kubernetes file "worker-deployment.yaml" created
```

Теперь можно удалить файл `docker_compose.yaml`:

```
$ rm docker-compose.yaml
```

Развертывание манифестов Kubernetes на локальном кластере Kubernetes, основанном на minikube

Наш следующий шаг — развертывание манифестов Kubernetes на локальном кластере Kubernetes, основанном на minikube.

Для запуска minikube на macOS тем должен быть предварительно установлен *VirtualBox*. Скачайте пакет VirtualBox для macOS со страницы загрузки (<https://oreil.ly/BewRq>), установите его, затем переместите в `/usr/local/bin/minikube`, чтобы сделать исполняемым. Учтите, что на момент написания данной книги пакет minikube устанавливается с кластером Kubernetes версии 1.15. Чтобы следить за ходом просмотра различных примеров, смотрите при установке minikube желаемую версию Kubernetes:

```
$ minikube start --kubernetes-version v1.15.0
🐳 minikube v1.2.0 on darwin (amd64)
🔧 Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
🐳 Configuring environment for Kubernetes v1.15.0 on Docker 18.09.6
📦 Downloading kubeadm v1.15.0
📦 Downloading kubelet v1.15.0
🔗 Pulling images ...
🚀 Launching Kubernetes ...
🕒 Verifying: apiserver proxy etcd scheduler controller dns
🏠 Done! kubectl is now configured to use "minikube"
```

Основная команда для взаимодействия с Kubernetes — `kubectl`.

Установите `kubectl` на вашей машине под управлением macOS, скачав его со страницы загрузки (<https://oreil.ly/f9Wv0>), затем переместите в `/usr/local/bin/minikube` и сделайте исполняемым.

Одно из главных понятий при работе с `kubectl` — *контекст*, то есть клиент Kubernetes, с которым вы хотите работать. В процессе установки `minikube` уже был создан контекст *minikube*. Указав `kubectl`, какой контекст использовать, можно с помощью следующей команды:

```
$ kubectl config use-context minikube
Switched to context "minikube".
```

Другой, более удобный способ — установить утилиту `kubectx` из репозитория Git (<https://oreil.ly/SIf1U>), затем выполнить:

```
$ kubectx minikube
Switched to context "minikube".
```



Еще один удобный клиентская утилита для работы с Kubernetes — `kube-ps1` (<https://oreil.ly/AcE32>). Если вы работаете под macOS, используя Zsh, добавьте в файл `~/.zshrc` следующий фрагмент кода:

```
source "/usr/local/opt/kube-ps1/share/kube-ps1.sh"
PS1='${kube_ps1}'$PS1
```

Эти команды меняют приглашение командной строки так, чтобы отображался текущий контекст и пространство имен Kubernetes. В ходе работы с несколькими клиентами Kubernetes это не стоящий плагин-выручалочка, позволяющий легко различать клиенты предэксплуатационного тестирования и промышленной эксплуатации.

Теперь можно выполнять команды `kubectl` для локального клиента `minikube`. Например, команда `kubectl get nodes` выводит список узлов, входящий в этот клиент. В данном случае узел только один, с ролью `master`:

```
$ kubectl get nodes
NAME      STATUS    ROLES    AGE    VERSION
minikube  Ready     master   2m14s  v1.15.0
```

Начнем с формирования объекта PVC (Persistent Volume Claim — заявление о постоянном хранении) из созданного `Kompose` файла `dbdata-persistentvolumeclaim.yaml`, соответствующего при запуске с помощью `docker-compose` выделенному для контейнера базы данных PostgreSQL локальному тому:


```
$ cat dbdata-persistentvolumeclaim.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: dbdata
  name: dbdata
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
status: {}
```

Для создания этого объекта в Kubernetes можно воспользоваться командой `kubectl create`, указав имя файла манифеста с помощью флага `-f`:

```
$ kubectl create -f dbdata-persistentvolumeclaim.yaml
persistentvolumeclaim/dbdata created
```

Выведем полный список всех PVC с помощью команды `kubectl get pvc`, чтобы убедиться в наличии созданного нами PVC:

```
$ kubectl get pvc
NAME          STATUS    VOLUME          CAPACITY
ACCESS MODES  STORAGECLASS  AGE
dbdata        Bound        pvc-39914723-4455-439b-a0f5-82a5f7421475  100Mi
RWX           standard      1m
```

Следующий шаг — создание объекта `Deployment` для PostgreSQL. Воспользуемся манифестом `db-deployment.yaml`, созданным ранее утилитой Kompose:

```
$ cat db-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose convert
    kompose.version: 1.16.0 (0c01309)
  creationTimestamp: null
  labels:
    io.kompose.service: db
  name: db
spec:
  replicas: 1
  strategy:
    type: Recreate
```

```

template:
  metadata:
    creationTimestamp: null
    labels:
      io.kompose.service: db
  spec:
    containers:
      - image: postgres:11
        name: postgres
        ports:
          - containerPort: 5432
        resources: {}
        volumeMounts:
          - mountPath: /var/lib/postgresql/data
            name: dbdata
        restartPolicy: Always
    volumes:
      - name: dbdata
        persistentVolumeClaim:
          claimName: dbdata
status: {}

```

Для создания этого объекта в терминале мы воспользуемся командой `kubectl create -f`, передвигая в ней имя файла манифеста:

```

$ kubectl create -f db-deployment.yaml
deployment.extensions/db created

```

Проверяем, что развертывание было создано, выводя списки всех развернутых в кластере и созданных в качестве чести развертывания модулей:

```

$ kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
db        1/1     1            1           1m

$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
db-67659d85bf-vrnw7                1/1     Running   0           1m

```

Далее создадим базис для нашего примера приложения Flask. С помощью команды, написанной в логичной `docker exec`, выполняем команду `psql` внутри запущенного контейнера Docker. В случае кластера Kubernetes эта команда называется `kubectl exec`:

```

$ kubectl exec -it db-67659d85bf-vrnw7 -- psql -U postgres
psql (11.4 (Debian 11.4-1.pgdg90+1))
Type "help" for help.

postgres=# create database wordcount;
CREATE DATABASE

```

```

postgres=# \q

$ kubectl exec -it db-67659d85bf-vrnw7 -- psql -U postgres wordcount
psql (11.4 (Debian 11.4-1.pgdg90+1))
Type "help" for help.

wordcount=# CREATE ROLE wordcount_dbadmin;
CREATE ROLE
wordcount=# ALTER ROLE wordcount_dbadmin LOGIN;
ALTER ROLE
wordcount=# ALTER USER wordcount_dbadmin PASSWORD 'MYPASS';
ALTER ROLE
wordcount=# \q

```

Следующий шаг — создание соответствующего развертыванию `db` объекта `Service`, благодаря которому развертывание становится видимым для остальных сервисов, работающих в данном кластере, и пример для сервиса-исполнителя `Redis` и основного сервиса приложения. Файл манифеста для сервиса `db` выглядит вот так:

```

$ cat db-service.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose convert
    kompose.version: 1.16.0 (0c01309)
  creationTimestamp: null
  labels:
    io.kompose.service: db
  name: db
spec:
  ports:
    - name: "5432"
      port: 5432
      targetPort: 5432
  selector:
    io.kompose.service: db
status:
  loadBalancer: {}

```

Стоит обратить внимание на следующий раздел:

```

labels:
  io.kompose.service: db

```

Он присутствует как в манифесте развертывания, так и в манифесте сервиса, собственно, связывая их между собой. Сервис связывается с любым развертыванием с той же меткой.

Создайте объект Service с помощью команды `kubectl create -f:`

```
$ kubectl create -f db-service.yaml
service/db created
```

Выводим список всех сервисов и убеждаемся, что сервис db создан:

```
$ kubectl get services
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
db            ClusterIP   10.110.108.96 <none>       5432/TCP    6s
kubernetes    ClusterIP   10.96.0.1     <none>       443/TCP    4h45m
```

Следующий сервис, который нам нужно развернуть, — Redis. Создайте объекты Deployment и Service на основе сгенерированных Kompose файлов манифестов:

```
$ cat redis-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose convert
    kompose.version: 1.16.0 (0c01309)
  creationTimestamp: null
  labels:
    io.kompose.service: redis
  name: redis
spec:
  replicas: 1
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: redis
    spec:
      containers:
        - image: redis:alpine
          name: redis
          ports:
            - containerPort: 6379
          resources: {}
          restartPolicy: Always
status: {}

$ kubectl create -f redis-deployment.yaml
deployment.extensions/redis created

$ kubectl get pods
```

```

NAME                READY   STATUS    RESTARTS   AGE
db-67659d85bf-vrnw7 1/1     Running   0           37m
redis-c6476fbff-8kpqz 1/1     Running   0           11s
$ kubectl create -f redis-service.yaml
service/redis created

```

```

$ cat redis-service.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose convert
    kompose.version: 1.16.0 (0c01309)
  creationTimestamp: null
  labels:
    io.kompose.service: redis
  name: redis
spec:
  ports:
    - name: "6379"
      port: 6379
      targetPort: 6379
  selector:
    io.kompose.service: redis
status:
  loadBalancer: {}

```

```

$ kubectl get services
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
db                  ClusterIP   10.110.108.96 <none>         5432/TCP   84s
kubernetes          ClusterIP   10.96.0.1     <none>         443/TCP    4h46m
redis              ClusterIP   10.106.44.183 <none>         6379/TCP   10s

```

Пок что р звернутые н ми дв сервис , db и redis, друг с другом ник к не связ ны. Следующ я ч сть приложения — процесс-исполнитель, который должен вз имодействов ть к к с PostgreSQL, т к и с Redis. Здесь и проявляются преимущест в сервисов Kubernetes. Р звертыв ние процесс -исполнителя может ссыл ться н конечные точки PostgreSQL и Redis по н зв ниям сервисов. Kubernetes будет зн ть, к к перен пр вить з просы от клиент (контейнеров, р бот ющих в к честве ч стей модулей в р звертыв нии процесс -исполните- ля) сервер м (контейнер м PostgreSQL и Redis, р бот ющим в к честве ч стей модулей в р звертыв ниях db и redis соответственно).

В число переменных среды в р звертыв нии процесс -исполнителя входит DATABASE_URL. В ней хр нится п роль используемой приложением б зы д нных. П роль нельзя ук зыв ть открытым текстом в ф йле м нифест р звертыв ния, поскольку этот ф йл должен вноситься в систему контроля версий. Т к что не- обходимо созд ть объект Secret Kubernetes.

Прежде всего преобразуем строку пароля в кодировку base64:

```
$ echo MYPASS | base64
MYPASSBASE64
```

Далее создадим файл манифеста, в котором описывается объект Secret Kubernetes, который мы хотим создать. Поскольку кодировка base64 нашего пароля не обеспечивает безопасности, воспользуемся sops для редактирования и сохранения зашифрованного файла манифеста secrets.yaml.enc:

```
$ sops --pgp E14104A0890994B9AC9C9F6782C1FF5E679EFF32 secrets.yaml.enc
```

Добавьте в редакторе следующие строки:

```
apiVersion: v1
kind: Secret
metadata:
  name: fbe-secret
type: Opaque
data:
  dbpass: MYPASSBASE64
```

Теперь можно вносить файл secrets.yaml.enc в систему контроля версий, поскольку в нем содержится зашифрованная версия значения пароля в кодировке base64.

Для рэншифровки зашифрованного файла можно использовать команду sops -d:

```
$ sops -d secrets.yaml.enc
apiVersion: v1
kind: Secret
metadata:
  name: fbe-secret
type: Opaque
data:
  dbpass: MYPASSBASE64
```

Наряду с помощью | вывод команды sops -d в команду kubectl create -f для создания объекта Secret Kubernetes:

```
$ sops -d secrets.yaml.enc | kubectl create -f -
secret/fbe-secret created
```

Просмотрим объекты Secret Kubernetes и получим описание созданного объекта Secret:

```
$ kubectl get secrets
```

| NAME | TYPE | DATA | AGE |
|---------------------|-------------------------------------|------|-------|
| default-token-k7652 | kubernetes.io/service-account-token | 3 | 3h19m |
| fbe-secret | Opaque | 1 | 45s |

```
$ kubectl describe secret fbe-secret
Name:          fbe-secret
Namespace:     default
Labels:        <none>
Annotations:   <none>
Type: Opaque
Data
dbpass: 12 bytes
```

Для извлечения Secret в кодировке `base64` применяем команду:

```
$ kubectl get secrets fbe-secret -ojson | jq -r ".data.dbpass"
MYPASSBASE64
```

Для получения п роля в виде открытого текст а м шине под упр влением macOS можно воспользов ться следующей командой:

```
$ kubectl get secrets fbe-secret -ojson | jq -r ".data.dbpass" | base64 -D
MYPASS
```

Н а м шине под упр влением Linux для декодировки `base64` служит флаг `-d`, т к что команда выглядит следующим образом:

```
$ kubectl get secrets fbe-secret -ojson | jq -r ".data.dbpass" | base64 -d
MYPASS
```

Теперь можно использовать объект Secret в манифесте развертывания процесса - исполнителя. Внесите изменения в сгенерированный утилитой Kompose файл `worker-deployment.yaml`, добавив две переменные среды:

- `DBPASS` — роль базы данных, который теперь будет извлекаться из объекта Secret `fbe-secret`;
- `DATABASE_URL` — полная строка соединения для PostgreSQL, включающая роль базы данных, на который ссылается в виде `${DBPASS}`.

Модифицированный файл `worker-deployment.yaml` выглядит так:

```
$ cat worker-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose convert
    kompose.version: 1.16.0 (0c01309)
  creationTimestamp: null
  labels:
    io.kompose.service: worker
  name: worker
```

```

spec:
  replicas: 1
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: worker
    spec:
      containers:
        - args:
            - worker.py
          env:
            - name: APP_SETTINGS
              value: config.ProductionConfig
            - name: DBPASS
              valueFrom:
                secretKeyRef:
                  name: fbe-secret
                  key: dbpass
            - name: DATABASE_URL
              value: postgresql://wordcount_dbadmin:${DBPASS}@db/wordcount
            - name: REDISTOGO_URL
              value: redis://redis:6379
          image: grigheo/flask-by-example:v1
          name: worker
          resources: {}
      restartPolicy: Always
status: {}

```

Создайте объект Deployment для процесс-исполнителя и логично прочим звертываниям с помощью команды `kubectl create -f:`

```

$ kubectl create -f worker-deployment.yaml
deployment.extensions/worker created

```

Выводим список модулей:

```

$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
db-67659d85bf-vrnw7                1/1     Running   1           21h
redis-c6476fbff-8kpqz              1/1     Running   1           21h
worker-7dbf5ff56c-vgs42            0/1     Init:ErrImagePull 0           7s

```

Как видите, состояние модуля процесс-исполнителя отображается как `Init:ErrImagePull`. Чтобы узнать подробности, выполняем `kubectl describe:`

```

$ kubectl describe pod worker-7dbf5ff56c-vgs42 | tail -10
node.kubernetes.io/unreachable:NoExecute for 300s

```


Events:

| Type | Reason | Age | From | Message |
|---------|-----------|---------------------|-------------------|---|
| Normal | Scheduled | 2m51s | default-scheduler | Successfully assigned default/worker-7dbf5ff56c-vgs42 to minikube |
| Normal | Pulling | 76s (x4 over 2m50s) | kubelet, minikube | Pulling image "griggheo/flask-by-example:v1" |
| Warning | Failed | 75s (x4 over 2m49s) | kubelet, minikube | Failed to pull image "griggheo/flask-by-example:v1": rpc error: code = Unknown desc = Error response from daemon: pull access denied for griggheo/flask-by-example, repository does not exist or may require 'docker login' |
| Warning | Failed | 75s (x4 over 2m49s) | kubelet, minikube | Error: ErrImagePull |
| Warning | Failed | 62s (x6 over 2m48s) | kubelet, minikube | Error: ImagePullBackOff |
| Normal | BackOff | 51s (x7 over 2m48s) | kubelet, minikube | Back-off pulling image "griggheo/flask-by-example:v1" |

Р звертыв ние попыт лось извлечь ч стный обр з Docker griggheo/flask-by-example:v1 без необходимых для доступ к ч стному реестру Docker учетных д нных. В Kubernetes есть специ льный тип объектов к к р з для т кого сце н рия — imagePullSecret.

Созд ем с помощью sops з шифров нный ф йл, содерж щий учетные д нные Docker Hub и вызов kubectl create secret:

```
$ sops --pgp E14104A0890994B9AC9C9F6782C1FF5E679EFF32 \
create_docker_credentials_secret.sh.enc
```

Содержимое ф йл выглядит вот т к:

```
DOCKER_REGISTRY_SERVER=docker.io
DOCKER_USER=Type your dockerhub username, same as when you `docker login`
DOCKER_EMAIL=Type your dockerhub email, same as when you `docker login`
DOCKER_PASSWORD=Type your dockerhub pw, same as when you `docker login`
```

```
kubectl create secret docker-registry myregistrykey \
--docker-server=$DOCKER_REGISTRY_SERVER \
--docker-username=$DOCKER_USER \
--docker-password=$DOCKER_PASSWORD \
--docker-email=$DOCKER_EMAIL
```

Решив проблему с шифрованием файла с помощью `sops` и пропуском его через `bash`:

```
$ sops -d create_docker_credentials_secret.sh.enc | bash -
secret/myregistrykey created
```

Просмотрим объект `Secret`:

```
$ kubectl get secrets myregistrykey -oyaml
apiVersion: v1
data:
  .dockerconfigjson: eyJhdXRobyI6eyJkb2NrZXIuaw8iO
kind: Secret
metadata:
  creationTimestamp: "2019-07-17T22:11:56Z"
  name: myregistrykey
  namespace: default
  resourceVersion: "16062"
  selfLink: /api/v1/namespaces/default/secrets/myregistrykey
  uid: 47d29ffc-69e4-41df-a237-1138cd9e8971
type: kubernetes.io/dockerconfigjson
```

Единственное изменение, которое необходимо внести в манифест развертывания процесса-исполнителя, — следующие строки:

```
imagePullSecrets:
- name: myregistrykey
```

Вставить их надо сразу вслед за строкой:

```
restartPolicy: Always
```

Удалите развертывание процесса-исполнителя и создайте его заново:

```
$ kubectl delete -f worker-deployment.yaml
deployment.extensions "worker" deleted

$ kubectl create -f worker-deployment.yaml
deployment.extensions/worker created
```

Теперь модуль процесса-исполнителя находится в состоянии `Running` без каких-либо ошибок:

```
$ kubectl get pods
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------|-------|---------|----------|-------|
| db-67659d85bf-vrnw7 | 1/1 | Running | 1 | 22h |
| redis-c6476fbff-8kpqz | 1/1 | Running | 1 | 21h |
| worker-7dbf5ff56c-hga37 | 1/1 | Running | 0 | 4m53s |

Просмотрим журналы модуля процесс-исполнителя с помощью команды `kubectl logs`:

```
$ kubectl logs worker-7dbf5ff56c-hga37
20:43:13 RQ worker 'rq:worker:040640781edd4055a990b798ac2eb52d'
started, version 1.0
20:43:13 *** Listening on default...
20:43:13 Cleaning registries for queue: default
```

Следующий этап — развертывание приложения. При развертывании приложения в среде с `docker-compose` в главе 11 мы использовали отдельный контейнер Docker для запуска мигрций, необходимых для обновления базы данных Flask. Подобные задачи хорошо подходят для запуска в виде вспомогательного контейнера в том же модуле, что и основной контейнер приложения. Мы опишем этот вспомогательный контейнер в манифесте развертывания нашего приложения в виде объекта Kubernetes `initContainer` (<https://oreil.ly/80L5L>). Контейнер такого типа ретировно выполняется внутри соответствующего модуля до запуска всех прочих контейнеров из него.

Добавьте следующий раздел в сгенерированный утилитой `Kompose` файл манифеста `app-deployment.yaml` и удалите файл `migrations-deployment.yaml`:

```
initContainers:
- args:
  - manage.py
  - db
  - upgrade
  env:
  - name: APP_SETTINGS
    value: config.ProductionConfig
  - name: DATABASE_URL
    value: postgresql://wordcount_dbadmin:@db/wordcount
  image: griggheo/flask-by-example:v1
  name: migrations
  resources: {}
```

```
$ rm migrations-deployment.yaml
```

Еще раз воспользуемся объектом `Secret fbe-secret`, созданным для развертывания процесс-исполнителя, в манифесте развертывания приложения:

```
$ cat app-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: kompose convert
```

```
    kompose.version: 1.16.0 (0c01309)
    creationTimestamp: null
    labels:
      io.kompose.service: app
    name: app
spec:
  replicas: 1
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: app
    spec:
      initContainers:
        - args:
            - manage.py
            - db
            - upgrade
          env:
            - name: APP_SETTINGS
              value: config.ProductionConfig
            - name: DBPASS
              valueFrom:
                secretKeyRef:
                  name: fbe-secret
                  key: dbpass
            - name: DATABASE_URL
              value: postgresql://wordcount_dbadmin:${DBPASS}@db/wordcount
          image: griggheo/flask-by-example:v1
          name: migrations
          resources: {}
      containers:
        - args:
            - manage.py
            - runserver
            - --host=0.0.0.0
          env:
            - name: APP_SETTINGS
              value: config.ProductionConfig
            - name: DBPASS
              valueFrom:
                secretKeyRef:
                  name: fbe-secret
                  key: dbpass
            - name: DATABASE_URL
              value: postgresql://wordcount_dbadmin:${DBPASS}@db/wordcount
            - name: REDISTOGO_URL
              value: redis://redis:6379
          image: griggheo/flask-by-example:v1
          name: app
```

```

    ports:
      - containerPort: 5000
    resources: {}
    restartPolicy: Always
status: {}

```

Создаем развертывание для приложения с помощью команды `kubectl create -f`, после чего выводим список модулей и информацию о модуле приложения:

```

$ kubectl create -f app-deployment.yaml
deployment.extensions/app created

```

```

$ kubectl get pods

```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------|-------|---------|----------|-------|
| app-c845d8969-18nhg | 1/1 | Running | 0 | 7s |
| db-67659d85bf-vrnw7 | 1/1 | Running | 1 | 22h |
| redis-c6476fbff-8kpqz | 1/1 | Running | 1 | 21h |
| worker-7dbf5ff56c-vgs42 | 1/1 | Running | 0 | 4m53s |

Последний элемент развертывания приложения в `minikube` — подключить к созданному сервису Kubernetes для приложения, причем объявленного с типом `LoadBalancer`, чтобы к нему можно было обратиться извне кластера:

```

$ cat app-service.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose convert
    kompose.version: 1.16.0 (0c01309)
  creationTimestamp: null
  labels:
    io.kompose.service: app
  name: app
spec:
  ports:
    - name: "5000"
      port: 5000
      targetPort: 5000
  type: LoadBalancer
  selector:
    io.kompose.service: app
status:
  loadBalancer: {}

```



К к и сервис `db`, сервис `app` связывается с развертыванием `app` через объявление метки, одинаковое в манифестах развертывания и сервиса:

```

labels:
  io.kompose.service: app

```

Создаем сервис с помощью команды `kubectl create`:

```
$ kubectl create -f app-service.yaml
service/app created
```

```
$ kubectl get services
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|------------|--------------|---------------|-------------|----------------|-----|
| app | LoadBalancer | 10.99.55.191 | <pending> | 5000:30097/TCP | 2s |
| db | ClusterIP | 10.110.108.96 | <none> | 5432/TCP | 21h |
| kubernetes | ClusterIP | 10.96.0.1 | <none> | 443/TCP | 26h |
| redis | ClusterIP | 10.106.44.183 | <none> | 6379/TCP | 21h |

Далее выполните команду:

```
$ minikube service app
```

В результате будет открыт используемый по умолчанию браузер с URL `http://192.168.99.100:30097/` и отображено домашнюю страницу с `Flask`.

В следующем разделе выполним развертывание тех же сервисов `Kubernetes` для нашего приложения в кластере `Kubernetes`, выделенном на `Google Cloud Platform (GCP)`, с помощью `Pulumi`.

Запуск кластера GKE Kubernetes в GCP с помощью Pulumi

В этом разделе воспользуемся примером `GKE Pulumi` (<https://oreil.ly/VGBfF>), так же документацией по настройке `GCP` (<https://oreil.ly/kRsFA>), так что прочитайте соответствующие документы, перейдя по указанным ссылкам.

Начнем с создания нового каталога:

```
$ mkdir pulumi_gke
$ cd pulumi_gke
```

Настройте SDK `Google Cloud` в соответствии с инструкциями для `macOS` (<https://oreil.ly/f4pPs>). Инициализируйте среду `GCP` с помощью команды `gcloud init`. Создайте новую конфигурацию и новый проект `pythonfordevops-gke-pulumi`¹:

```
$ gcloud init
Welcome! This command will take you through the configuration of gcloud.
```

¹ Изначально проект следует сделать другим, чтобы не возникло конфликта с уже существующим, созданным в той же книге. — *Примеч. пер.*

Settings from your current configuration [default] are:
core:

```
account: grig.gheorghiu@gmail.com
disable_usage_reporting: 'True'
project: pulumi-gke-testing
```

Pick configuration to use:

- [1] Re-initialize this configuration [default] with new settings
- [2] Create a new configuration

Please enter your numeric choice: 2

Enter configuration name. Names start with a lower case letter and contain only lower case letters a-z, digits 0-9, and hyphens '-':
pythonfordevops-gke-pulumi

Your current configuration has been set to: [pythonfordevops-gke-pulumi]

Pick cloud project to use:

- [1] pulumi-gke-testing
- [2] Create a new project

Please enter numeric choice or text value (must exactly match list item): 2

Enter a Project ID. pythonfordevops-gke-pulumi

Your current project has been set to: [pythonfordevops-gke-pulumi].

Войдите в учетную запись GCP:

```
$ gcloud auth login
```

Войдите в приложение по умолчанию — pythonfordevops-gke-pulumi:

```
$ gcloud auth application-default login
```

Создайте новый проект Pulumi с помощью команды `pulumi new`, указав в качестве шаблона `gcp-python`, в качестве имени проекта — `pythonfordevops-gke-pulumi`:

```
$ pulumi new
```

Please choose a template: gcp-python

A minimal Google Cloud Python Pulumi program

This command will walk you through creating a new Pulumi project.

Enter a value or leave blank to accept the (default), and press <ENTER>.
Press ^C at any time to quit.

```
project name: (pulumi_gke_py) pythonfordevops-gke-pulumi
```

```
project description: (A minimal Google Cloud Python Pulumi program)
```

```
Created project 'pythonfordevops-gke-pulumi'
```

```
stack name: (dev)
```

```
Created stack 'dev'
```

```
gcp:project: The Google Cloud project to deploy into: pythonfordevops-gke-pulumi
Saved config
```

Your new project is ready to go!

To perform an initial deployment, run the following commands:

1. `virtualenv -p python3 venv`
2. `source venv/bin/activate`
3. `pip3 install -r requirements.txt`

Then, run `'pulumi up'`.

Команда `pulumi new` создала следующие файлы:

```
$ ls -la
ls -la
total 40
drwxr-xr-x  7 ggheo  staff  224 Jul 16 15:08 .
drwxr-xr-x  6 ggheo  staff  192 Jul 16 15:06 ..
-rw-----  1 ggheo  staff   12 Jul 16 15:07 .gitignore
-rw-r--r--  1 ggheo  staff   50 Jul 16 15:08 Pulumi.dev.yaml
-rw-----  1 ggheo  staff  107 Jul 16 15:07 Pulumi.yaml
-rw-----  1 ggheo  staff  203 Jul 16 15:07 __main__.py
-rw-----  1 ggheo  staff   34 Jul 16 15:07 requirements.txt
```

Далее мы воспользуемся примером `gcp-py-gke` из репозитория GitHub примеров Pulumi (<https://oreil.ly/SIT-v>).

Скопируйте файлы `*.py` и `requirements.txt` из `examples/gcp-py-gke` в текущий каталог:

```
$ cp ~/pulumi-examples/gcp-py-gke/*.py .
$ cp ~/pulumi-examples/gcp-py-gke/requirements.txt .
```

Настройте все относящиеся к GCP переменные, необходимые для работы Pulumi в GCP:

```
$ pulumi config set gcp:project pythonfordevops-gke-pulumi
$ pulumi config set gcp:zone us-west1-a
$ pulumi config set password --secret PASS_FOR_KUBE_CLUSTER
```

Создайте и начните использовать виртуальную среду Python, установите объявленные в файле `requirements.txt` зависимости, после чего загрузите кластер GKE, описанный в файле `main.py`, с помощью команды `pulumi up`:

```
$ virtualenv -p python3 venv
$ source venv/bin/activate
$ pip3 install -r requirements.txt
$ pulumi up
```




Не забывайте активировать API Kubernetes Engine, связав его с учетной записью биллинг Google в веб-консоли GCP.

Теперь кластер GKE виден в консоли GCP (<https://oreil.ly/Su5FZ>).

Для взаимодействия с только что выделенным кластером GKE сгенерируйте подходящую конфигурацию `kubectl` и используйте ее. Удобно, что программ `Pulumi` может экспортировать конфигурацию `kubectl` в виде выходного ресурса :

```
$ pulumi stack output kubeconfig > kubeconfig.yaml
$ export KUBECONFIG=./kubeconfig.yaml
```

Выводим список узлов, из которых состоит наш кластер GKE:

```
$ kubectl get nodes
```

| NAME | STATUS | ROLES | AGE |
|--|--------|--------|-------|
| VERSION | | | |
| gke-gke-cluster-ea17e87-default-pool-fd130152-30p3-v1.13.7-gke.8 | Ready | <none> | 4m29s |
| gke-gke-cluster-ea17e87-default-pool-fd130152-kf9k-v1.13.7-gke.8 | Ready | <none> | 4m29s |
| gke-gke-cluster-ea17e87-default-pool-fd130152-x9dx-v1.13.7-gke.8 | Ready | <none> | 4m27s |

Развертывание примера приложения Flask в GKE

Воспользуемся теми же манифестами Kubernetes, что и в примере `minikube`, и развернем их в кластере Kubernetes в GKE с помощью команды `kubectl`. Начнем с создания развертывания и сервиса `redis`:

```
$ kubectl create -f redis-deployment.yaml
deployment.extensions/redis created
```

```
$ kubectl get pods
```

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------------------------|-------|---------|----------|-------|
| canary-aqw8jtf0-f54b9749-q5wqj | 1/1 | Running | 0 | 5m57s |
| redis-9946db5cc-8g6zz | 1/1 | Running | 0 | 20s |

```
$ kubectl create -f redis-service.yaml
service/redis created
```

```
$ kubectl get service redis
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|-------|-----------|---------------|-------------|----------|-----|
| redis | ClusterIP | 10.59.245.221 | <none> | 6379/TCP | 18s |

Создаем объект PersistentVolumeClaim в качестве тома данных для баз данных PostgreSQL:

```
$ kubectl create -f dbdata-persistentvolumeclaim.yaml
persistentvolumeclaim/dbdata created
```

```
$ kubectl get pvc
NAME      STATUS   VOLUME                                     CAPACITY
dbdata    Bound    pvc-00c8156c-b618-11e9-9e84-42010a8a006f  1Gi
  ACCESS MODES  STORAGECLASS  AGE
  RWO           standard      12s
```

Создаем развертывание db:

```
$ kubectl create -f db-deployment.yaml
deployment.extensions/db created
```

```
$ kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
canary-aqw8jtfo-f54b9749-q5wqj    1/1     Running            0           8m52s
db-6b4fbb57d9-cjjsx               0/1     CrashLoopBackOff   1           38s
redis-9946db5cc-8g6zz             1/1     Running            0           3m15s
```

```
$ kubectl logs db-6b4fbb57d9-cjjsx
```

```
initdb: directory "/var/lib/postgresql/data" exists but is not empty
It contains a lost+found directory, perhaps due to it being a mount point.
Using a mount point directly as the data directory is not recommended.
Create a subdirectory under the mount point.
```

При создании развертывания db мы столкнулись с проблемой. GKE выделил том постоянного хранения, смонтированный каталог `/var/lib/postgresql/data`, и, согласно приведенному сообщению об ошибке, непустой.

Удаляем неудачное развертывание db:

```
$ kubectl delete -f db-deployment.yaml
deployment.extensions "db" deleted
```

Создаем новый временный модуль для монтирования того же объекта dbdata типа PersistentVolumeClaim в виде каталога `/data` внутри модуля, чтобы можно было посмотреть содержимое его файловой системы. Запуск подобного временного модуля для целей отладки — очень полезный прием:

```
$ cat pvc-inspect.yaml
kind: Pod
apiVersion: v1
metadata:
  name: pvc-inspect
spec:
```

```
volumes:
- name: dbdata
  persistentVolumeClaim:
    claimName: dbdata
containers:
- name: debugger
  image: busybox
  command: ['sleep', '3600']
  volumeMounts:
  - mountPath: "/data"
    name: dbdata
```

```
$ kubectl create -f pvc-inspect.yaml
pod/pvc-inspect created
```

```
$ kubectl get pods
```

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------------------------|-------|---------|----------|-----|
| canary-aqw8jtfo-f54b9749-q5wqj | 1/1 | Running | 0 | 20m |
| pvc-inspect | 1/1 | Running | 0 | 35s |
| redis-9946db5cc-8g6zz | 1/1 | Running | 0 | 14m |

С помощью команды `kubectl exec` открываем командную оболочку внутри модуля, чтобы просмотреть содержимое каталога `/data`:

```
$ kubectl exec -it pvc-inspect -- sh
/ # cd /data
/data # ls -la
total 24
drwx----- 3 999      root          4096 Aug  3 17:57 .
drwxr-xr-x  1 root     root          4096 Aug  3 18:08 ..
drwx----- 2 999      root       16384 Aug  3 17:57 lost+found
/data # rm -rf lost+found/
/data # exit
```

Обратите внимание на то, что каталог `/data` содержит подкаталог `lost+found`, который необходимо удалить.

Удаляем временный модуль:

```
$ kubectl delete pod pvc-inspect
pod "pvc-inspect" deleted
```

Снова создаем развертывание `db`, и этот раз успешно:

```
$ kubectl create -f db-deployment.yaml
deployment.extensions/db created
```

```
$ kubectl get pods
```

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------------------------|-------|---------|----------|-----|
| canary-aqw8jtfo-f54b9749-q5wqj | 1/1 | Running | 0 | 23m |
| db-6b4fbb57d9-8h978 | 1/1 | Running | 0 | 19s |
| redis-9946db5cc-8g6zz | 1/1 | Running | 0 | 17m |

```
$ kubectl logs db-6b4fbb57d9-8h978
PostgreSQL init process complete; ready for start up.

2019-08-03 18:12:01.108 UTC [1]
LOG:  listening on IPv4 address "0.0.0.0", port 5432
2019-08-03 18:12:01.108 UTC [1]
LOG:  listening on IPv6 address "::", port 5432
2019-08-03 18:12:01.114 UTC [1]
LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2019-08-03 18:12:01.135 UTC [50]
LOG:  database system was shut down at 2019-08-03 18:12:01 UTC
2019-08-03 18:12:01.141 UTC [1]
LOG:  database system is ready to accept connections
```

Созд ем б з у д нных wordcount и роль wordcount_dbadmin:

```
$ kubectl exec -it db-6b4fbb57d9-8h978 -- psql -U postgres
psql (11.4 (Debian 11.4-1.pgdg90+1))
Type "help" for help.

postgres=# create database wordcount;
CREATE DATABASE
postgres=# \q

$ kubectl exec -it db-6b4fbb57d9-8h978 -- psql -U postgres wordcount
psql (11.4 (Debian 11.4-1.pgdg90+1))
Type "help" for help.

wordcount=# CREATE ROLE wordcount_dbadmin;
CREATE ROLE
wordcount=# ALTER ROLE wordcount_dbadmin LOGIN;
ALTER ROLE
wordcount=# ALTER USER wordcount_dbadmin PASSWORD 'MYNEWPASS';
ALTER ROLE
wordcount=# \q
```

Созд ем сервис db:

```
$ kubectl create -f db-service.yaml
service/db created

$ kubectl describe service db
Name:          db
Namespace:     default
Labels:        io.kompose.service=db
Annotations:   kompose.cmd: kompose convert
               kompose.version: 1.16.0 (0c01309)
Selector:      io.kompose.service=db
Type:          ClusterIP
IP:            10.59.241.181
Port:          5432 5432/TCP
```

```
TargetPort:      5432/TCP
Endpoints:       10.56.2.5:5432
Session Affinity: None
Events:          <none>
```

Создаем объект Secret на основе значения пароля бд в кодировке base64. Значение пароля в виде открытого текста сохраняется в файле, зашифрованным с помощью sops:

```
$ echo MYNEWPASS | base64
MYNEWPASSBASE64
```

```
$ sops secrets.yaml.enc
```

```
apiVersion: v1
kind: Secret
metadata:
  name: fbe-secret
type: Opaque
data:
  dbpass: MYNEWPASSBASE64
```

```
$ sops -d secrets.yaml.enc | kubectl create -f -
secret/fbe-secret created
```

```
kubectl describe secret fbe-secret
Name:          fbe-secret
Namespace:     default
Labels:        <none>
Annotations:   <none>
```

```
Type: Opaque
```

```
Data
===
dbpass: 21 bytes
```

Создайте еще один объект Secret для учетных данных Docker Hub:

```
$ sops -d create_docker_credentials_secret.sh.enc | bash -
secret/myregistrykey created
```

Поскольку мы рассмотрим сценарий развертывания приложения в GKE для целей промышленной эксплуатации, создадим три реплики в файле `worker-deployment.yaml` таким образом, чтобы в любой момент работало три модуля процесса-исполнителя:

```
$ kubectl create -f worker-deployment.yaml
deployment.extensions/worker created
```

Убеждаемся, что работают три модуля процесс-исполнителя:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
canary-aqw8jtfo-f54b9749-q5wqj    1/1     Running   0           39m
db-6b4fbb57d9-8h978               1/1     Running   0           16m
redis-9946db5cc-8g6zz             1/1     Running   0           34m
worker-8cf5dc699-98z99            1/1     Running   0           35s
worker-8cf5dc699-9s26v            1/1     Running   0           35s
worker-8cf5dc699-v6ckr            1/1     Running   0           35s

$ kubectl logs worker-8cf5dc699-98z99
18:28:08 RQ worker 'rq:worker:1355d2cad49646e4953c6b4d978571f1' started,
  version 1.0
18:28:08 *** Listening on default...
```

Ан логично задать метр replicas в файле app-deployment.yaml следующим 2:

```
$ kubectl create -f app-deployment.yaml
deployment.extensions/app created
```

Удостоверяемся, что работают два модуля приложения:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
app-7964cff98f-5bx4s              1/1     Running   0           54s
app-7964cff98f-8n8hk              1/1     Running   0           54s
canary-aqw8jtfo-f54b9749-q5wqj    1/1     Running   0           41m
db-6b4fbb57d9-8h978               1/1     Running   0           19m
redis-9946db5cc-8g6zz             1/1     Running   0           36m
worker-8cf5dc699-98z99            1/1     Running   0           2m44s
worker-8cf5dc699-9s26v            1/1     Running   0           2m44s
worker-8cf5dc699-v6ckr            1/1     Running   0           2m44s
```

Создаем сервис app:

```
$ kubectl create -f app-service.yaml
service/app created
```

Обратите внимание на то, что был создан сервис типа LoadBalancer:

```
$ kubectl describe service app
Name:                                app
Namespace:                          default
Labels:                              io.kompose.service=app
Annotations:                         kompose.cmd: kompose convert
                                    kompose.version: 1.16.0 (0c01309)
Selector:                            io.kompose.service=app
Type:                                LoadBalancer
IP:                                  10.59.255.31
LoadBalancer Ingress:                34.83.242.171
Port:                                5000 5000/TCP
```

```

TargetPort:      5000/TCP
NodePort:        5000 31305/TCP
Endpoints:       10.56.1.6:5000,10.56.2.12:5000
Session Affinity: None
External Traffic Policy: Cluster
Events:

```

| Type | Reason | Age | From | Message |
|--------|----------------------|-----|--------------------|------------------------|
| Normal | EnsuringLoadBalancer | 72s | service-controller | Ensuring load balancer |
| Normal | EnsuredLoadBalancer | 33s | service-controller | Ensured load balancer |

Проверяем работу приложения, обращаясь к URL конечной точки, в основе которого лежит соответствующий LoadBalancer Ingress IP-адрес `http://34.83.242.171:5000`.

Мы продемонстрировали создание типовых объектов Kubernetes, к Deployment, Service и Secret, из исходных манифестов Kubernetes. По мере усложнения приложения начнут проявляться ограничения этого подхода, поскольку будет все сложнее строить эти файлы для конкретной среды (например, для тестирования, интеграции или промышленной эксплуатации). У каждой среды будет свой набор знаний среды и секретных данных, которые придется отслеживать. В целом отслеживать, когда какие манифесты были обновлены, становится все сложнее. В экосистеме Kubernetes можно найти немало решений этой проблемы, одно из наиболее распространенных — систем управления пакетами Helm (<https://oreil.ly/duKVw>). Ее можно считать эквивалентом систем управления пакетами yum и apt для Kubernetes.

В следующем разделе мы покажем, как в кластере GKE с помощью Helm установить и настроить под свои нужды Prometheus и Grafana.

Установка чартов Helm для Prometheus и Grafana

Серверной части Tiller в текущей версии Helm¹ (в2 на момент написания книги) необходимо предоставить определенные права доступа внутри кластера Kubernetes.

Создаем новую учетную запись сервис-а Kubernetes для Tiller и предоставляем ей нужные права:

```

$ kubectl -n kube-system create sa tiller

$ kubectl create clusterrolebinding tiller \
  --clusterrole cluster-admin \

```

¹ Начиная с версии 3, для использования Helm в кластере больше не требуется устанавливать Tiller и команды `helm init` в этой версии уже нет. — *Примеч. пер.*

```
--serviceaccount=kube-system:tiller
```

```
$ kubectl patch deploy --namespace kube-system \
tiller-deploy -p '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'
```

Ск ч йте соответствующий в шей опер ционной системе исполняемый ф йл Helm с офици льной стр ницы Helm (<https://oreil.ly/sPwDO>) и уст новите его, после чего уст новите Tiller с помощью ком нды `helm init`:

```
$ helm init
```

Созд йте простр нство имен `monitoring`:

```
$ kubectl create namespace monitoring
namespace/monitoring created
```

Уст новите ч рт Helm Prometheus (<https://oreil.ly/CSaSo>) в простр нстве имен `monitoring`:

```
$ helm install --name prometheus --namespace monitoring stable/prometheus
NAME: prometheus
LAST DEPLOYED: Tue Aug 27 12:59:40 2019
NAMESPACE: monitoring
STATUS: DEPLOYED
```

Выводим списки всех модулей, сервисов и объектов ConfigMap в простр нстве имен `monitoring`:

```
$ kubectl get pods -nmonitoring
```

| NAME | READY | STATUS | RESTARTS | AGE |
|--|-------|---------|----------|-----|
| prometheus-alertmanager-df57f6df6-4b81v | 2/2 | Running | 0 | 3m |
| prometheus-kube-state-metrics-564564f799-t6qdm | 1/1 | Running | 0 | 3m |
| prometheus-node-exporter-b4sb9 | 1/1 | Running | 0 | 3m |
| prometheus-node-exporter-n4z2g | 1/1 | Running | 0 | 3m |
| prometheus-node-exporter-w7hn7 | 1/1 | Running | 0 | 3m |
| prometheus-pushgateway-56b65bcf5f-whx5t | 1/1 | Running | 0 | 3m |
| prometheus-server-7555945646-d86gn | 2/2 | Running | 0 | 3m |

```
$ kubectl get services -nmonitoring
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) |
|-------------------------------------|-----------|-------------|-------------|----------|
| prometheus-alertmanager-3m51s | ClusterIP | 10.0.6.98 | <none> | 80/TCP |
| prometheus-kube-state-metrics-3m51s | ClusterIP | None | <none> | 80/TCP |
| prometheus-node-exporter-3m51s | ClusterIP | None | <none> | 9100/TCP |
| prometheus-pushgateway-3m51s | ClusterIP | 10.0.13.216 | <none> | 9091/TCP |

| | | | | |
|-------------------|-----------|-----------|--------|--------|
| prometheus-server | ClusterIP | 10.0.4.74 | <none> | 80/TCP |
| 3m51s | | | | |

```
$ kubectl get configmaps -nmonitoring
```

| NAME | DATA | AGE |
|-------------------------|------|-------|
| prometheus-alertmanager | 1 | 3m58s |
| prometheus-server | 3 | 3m58s |

Подключаемся к UI Prometheus с помощью команды `kubectl port-forward`:

```
$ export PROMETHEUS_POD_NAME=$(kubectl get pods --namespace monitoring \
-l "app=prometheus,component=server" -o jsonpath="{.items[0].metadata.name}")
```

```
$ echo $PROMETHEUS_POD_NAME
prometheus-server-7555945646-d86gn
```

```
$ kubectl --namespace monitoring port-forward $PROMETHEUS_POD_NAME 9090
Forwarding from 127.0.0.1:9090 -> 9090
Forwarding from [::1]:9090 -> 9090
Handling connection for 9090
```

Переходим по адресу `localhost:9090` в браузере и видим UI Prometheus.

Устанавливаем чарт Helm для Grafana (<https://oreil.ly/--wEN>) в пространстве имен `monitoring`:

```
$ helm install --name grafana --namespace monitoring stable/grafana
NAME: grafana
LAST DEPLOYED: Tue Aug 27 13:10:02 2019
NAMESPACE: monitoring
STATUS: DEPLOYED
```

Выводим списки всех относящихся к Grafana модулей, сервисов, объектов ConfigMap и Secret в пространстве имен `monitoring`:

```
$ kubectl get pods -nmonitoring | grep grafana
grafana-84b887cf4d-wplcr          1/1      Running   0

$ kubectl get services -nmonitoring | grep grafana
grafana                           ClusterIP   10.0.5.154   <none>      80/TCP

$ kubectl get configmaps -nmonitoring | grep grafana
grafana                1      99s
grafana-test           1      99s

$ kubectl get secrets -nmonitoring | grep grafana
grafana                Opaque
grafana-test-token-85x4x  kubernetes.io/service-account-token
grafana-token-jw2qg     kubernetes.io/service-account-token
```

Получим роль для пользователя `admin` веб-интерфейс Grafana:

```
$ kubectl get secret --namespace monitoring grafana \
-o jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

КАКОЙТОСЕКРЕТНЫЙТЕКСТ

Подключемся к UI Grafana с помощью команды `kubectl port-forward`:

```
$ export GRAFANA_POD_NAME=$(kubectl get pods --namespace monitoring \
-l "app=grafana,release=grafana" -o jsonpath="{.items[0].metadata.name}")
```

```
$ kubectl --namespace monitoring port-forward $GRAFANA_POD_NAME 3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
```

Переходим по адресу `localhost:3000` в браузере и видим UI Grafana. Входим как пользователь `admin` с помощью полученного пароля.

С помощью команды `helm list` выводим список установленных чартов. Текущая установка чарта называется релизом Helm (Helm release):

```
$ helm list
NAME                REVISION  UPDATED                         STATUS  CHART
APP VERSION NAMESPACE
grafana              1         Tue Aug 27 13:10:02 2019  DEPLOYED  grafana-3.8.3
6.2.5               monitoring
prometheus.         1         Tue Aug 27 12:59:40 2019  DEPLOYED  prometheus-9.1.0
2.11.1              monitoring
```

В большинстве случаев чарты Helm придется настроить под свои нужды. Легче всего это сделать, если скачать чарт и установить его из локальной файловой системы с помощью `helm`.

С более свежими стабильными релизами чартов Helm Prometheus и Grafana можно получить с помощью команды `helm fetch`, скачивающей архивы чартов в формате TGZ:

```
$ mkdir charts
$ cd charts
$ helm fetch stable/prometheus
$ helm fetch stable/grafana
$ ls -la
total 80
drwxr-xr-x  4 ggheo  staff   128 Aug 27 13:59 .
drwxr-xr-x 15 ggheo  staff   480 Aug 27 13:55 ..
-rw-r--r--  1 ggheo  staff  16195 Aug 27 13:55 grafana-3.8.3.tgz
-rw-r--r--  1 ggheo  staff 23481 Aug 27 13:54 prometheus-9.1.0.tgz
```

Разархивируем содержимое TGZ-файлов, затем удалим их:

```
$ tar xzf prometheus-9.1.0.tgz; rm prometheus-9.1.0.tgz
$ tar xzf grafana-3.8.3.tgz; rm grafana-3.8.3.tgz
```

Шаблонизированные манифесты Kubernetes хранятся по умолчанию в подкаталогах `templates` каталогов чартов, так что в данном случае они будут располагаться в `prometheus/templates` и `grafana/templates`. Параметры конфигурации конкретного чарта объявляются в файле `values.yaml` в каталоге чарта.

В качестве примера настройки под свои нужды чарт Helm добавим в Grafana том постоянного хранения, чтобы не потерять данные при перезапуске модулей Grafana.

Отредактируйте файл `grafana/values.yaml`, задав значение `true` для подключения родительского ключа `persistence` (по умолчанию `false`) в следующем разделе:

```
## Открываем возможности постоянного хранения данных с помощью объектов
## Persistent Volume Claim
## ссылка: http://kubernetes.io/docs/user-guide/persistent-volumes/
##
persistence:
  enabled: true
  # storageClassName: default
  accessModes:
    - ReadWriteOnce
  size: 10Gi
  # annotations: {}
  finalizers:
    - kubernetes.io/pvc-protection
  # subPath: ""
  # existingClaim:
```

Обновите текущий выпуск чарта `grafana` Helm с помощью команды `helm upgrade`. Последний аргумент этой команды — название локального каталога с чартом. Выполните в родительском каталоге каталога чарта `grafana` следующую команду:

```
$ helm upgrade grafana grafana/
Release "grafana" has been upgraded. Happy Helming!
```

Убеждаемся, что для Grafana был создан PVC в пространстве имен `monitoring`:

```
kubectl describe pvc grafana -nmonitoring
Name:          grafana
Namespace:     monitoring
StorageClass:  standard
Status:        Bound
Volume:        pvc-31d47393-c910-11e9-87c5-42010a8a0021
Labels:        app=grafana
               chart=grafana-3.8.3
               heritage=Tiller
               release=grafana
```

```

Annotations: pv.kubernetes.io/bind-completed: yes
              pv.kubernetes.io/bound-by-controller: yes
              volume.beta.kubernetes.io/storage-provisioner:kubernetes.io/gce-pd
Finalizers: [kubernetes.io/pvc-protection]
Capacity: 10Gi
Access Modes:RWO
Mounted By: grafana-84f79d5c45-zlqz8
Events:
Type      Reason              Age   From              Message
----      -
Normal    ProvisioningSucceeded 88s   persistentvolume-controller Successfully
provisioned volume pvc-31d47393-c910-11e9-87c5-42010a8a0021
using kubernetes.io/gce-pd

```

Еще один пример настройки ч ртов Helm под свои нужды, и этот р з для ч рт Prometheus, — изменение используемого по умолчанию срок хр нения информации в Prometheus, р вного 15 дням.

Измените значение параметра `retention` в файле `prometheus/values.yaml` на 30 дней:

```

## Срок хранения информации в Prometheus (по умолчанию 15 дней,
## если не указано иное значение)
##
retention: "30d"

```

Обновите текущий выпуск ч рт Prometheus, з пустив команду `helm upgrade`. Выполнить эту команду необходимо в родительском к т лог е к т лог ч рт `prometheus`:

```

$ helm upgrade prometheus prometheus
Release "prometheus" has been upgraded. Happy Helming!

```

Проверим, изменился ли срок хранения информации на 30-дневный. Выполните команду `kubectl describe` для з пущенного в н стоящий момент модуля Prometheus в пространстве имен `monitoring` и обратите внимание на р здел Args вывод :

```

$ kubectl get pods -nmonitoring
NAME                                     READY   STATUS    RESTARTS   AGE
grafana-84f79d5c45-zlqz8               1/1     Running   0           9m
prometheus-alertmanager-df57f6df6-4b8lv 2/2     Running   0           87m
prometheus-kube-state-metrics-564564f799-t6qdm 1/1     Running   0           87m
prometheus-node-exporter-b4sb9          1/1     Running   0           87m
prometheus-node-exporter-n4z2g          1/1     Running   0           87m
prometheus-node-exporter-w7hn7          1/1     Running   0           87m
prometheus-pushgateway-56b65bcf5f-whx5t 1/1     Running   0           87m
prometheus-server-779ffd445f-41lqr      2/2     Running   0           3m

```

```
$ kubectl describe pod prometheus-server-779ffd445f-41lqr -nmonitoring
OUTPUT OMITTED
  Args:
    --storage.tsdb.retention.time=30d
    --config.file=/etc/config/prometheus.yml
    --storage.tsdb.path=/data
    --web.console.libraries=/etc/prometheus/console_libraries
    --web.console.templates=/etc/prometheus/consoles
    --web.enable-lifecycle
```

Удаление кластера GKE

Очистка используемых для тестирования ресурсов, которые больше не нужны, окупается буквально сторицей. В противном случае в конце месяца вы можете ждать неприятный сюрприз в виде счета от поставщиков облачных сервисов.

Удалить кластер GKE можно с помощью команды `pulumi destroy`:

```
$ pulumi destroy
```

Previewing destroy (dev):

| | Type | Name | Plan |
|---|--------------------------------|--------------------------------|--------|
| - | pulumi:pulumi:Stack | pythonfordevops-gke-pulumi-dev | delete |
| - | └─ kubernetes:core:Service | ingress | delete |
| - | └─ kubernetes:apps:Deployment | canary | delete |
| - | └─ pulumi:providers:kubernetes | gke_k8s | delete |
| - | └─ gcp:container:Cluster | gke-cluster | delete |
| - | └─ random:index:RandomString | password | delete |

Resources:

- 6 to delete

Do you want to perform this destroy? yes

Destroying (dev):

| | Type | Name | Status |
|---|--------------------------------|--------------------------------|---------|
| - | pulumi:pulumi:Stack | pythonfordevops-gke-pulumi-dev | deleted |
| - | └─ kubernetes:core:Service | ingress | deleted |
| - | └─ kubernetes:apps:Deployment | canary | deleted |
| - | └─ pulumi:providers:kubernetes | gke_k8s | deleted |
| - | └─ gcp:container:Cluster | gke-cluster | deleted |
| - | └─ random:index:RandomString | password | deleted |

Resources:

- 6 deleted

Duration: 3m18s

Упражнения

- Попробуйте использовать для PostgreSQL Google Cloud SQL вместо запуска PostgreSQL в контейнере Docker в GKE.
- Запустите кластер EKS Amazon с помощью AWS Cloud Development Kit (<https://aws.amazon.com/cdk>) и разверните в нем пример приложения.
- Попробуйте использовать Amazon RDS PostgreSQL вместо запуска PostgreSQL в контейнере Docker в EKS.
- Поэкспериментируйте с утилитой Kustomize (<https://oreil.ly/ie9n6>) для работы с YAML-файлами манифестов Kubernetes в качестве альтернативы Helm.

Технологии бессерверной обработки данных

Бессерверная обработка данных (serverless) — очень популярный сегодня термин в отрасли ИТ. Как бы то ни было, с подобными терминами, мнения относительно их точного значения существенно разнятся. На первый взгляд бессерверная обработка данных подразумевает мир, в котором больше не нужно заботиться о серверах. В некоторой степени это правда, но лишь для разработчиков, использующих предоставляемую бессерверными технологиями функциональность. Это лишь демонстрирует, что нужно сделать еще *очень много*, прежде чем мы окажемся в идеальной структуре, где нет серверов.

Многие считают термин «*бессерверная обработка данных*» синонимом функции как сервиса (FaaS). Частично это оправданно и во многом связано с запуском AWS-сервиса Lambda в 2015 году. Функции AWS Lambda могут выполняться в облаке без развертывания обычного сервера в качестве хоста для них. Отсюда и термин «*бессерверная*».

Однако FaaS не единственный сервис, который можно считать бессерверным. Сегодня все три крупнейших поставщика общедоступных облачных сервисов (Amazon, Microsoft и Google) предлагают функциональность «контейнер как сервис» (CaaS), позволяющую развертывать в их облаках полнофункциональные контейнеры Docker без выделения серверов в качестве хостов для них. Эти сервисы также можно назвать бессерверными. Примеры таких сервисов — AWS Fargate, Microsoft Azure Container Instances и Google Cloud Run.

Для каких сценариев использования подходят бессерверные технологии? Вот некоторые сценарии применения технологий FaaS, таких как AWS Lambda, особенно с учетом того, что вызов функций Lambda может инициироваться событиями от других облачных сервисов.

- Технологический процесс ETL (Extract — Transform — Load, извлечение — преобразование — загрузка) обработки данных, например, когда файл загружается

в S3, который инициирует выполнение функции Lambda, производящей ETL-обработку данных и отправляющей их в очередь или в базу данных в прикладной среде.

- ETL-обработка журналов, присылаемых в CloudWatch другими сервисами.
- Cron-подобное планирование заданий на основе событий CloudWatch, инициирующих выполнение функций Lambda.
- Уведомления в режиме реального времени на основе Amazon SNS, инициирующих выполнение функций Lambda.
- Обработка сообщений электронной почты с помощью AWS Lambda и Amazon SES.
- Бессерверный хостинг веб-сайтов, при котором статические веб-ресурсы (JavaScript, CSS и HTML) хранятся в S3 и роль клиентской среды играет сервис CloudFront CDN, API REST обслуживается API Gateway, переносимым запросы API функциям Lambda, которые взаимодействуют с прикладной средой (Amazon RDS или Amazon DynamoDB).

Многие сценарии использования бессерверных технологий описаны в онлайн-документации каждого из поставщиков облачных сервисов. Например, в бессерверной экосистеме Google Cloud служб Google AppEngine лучше всего подходит для разработки веб-приложениями, Google Functions — для разработки API, CloudRun — для запуска процессов в контейнерах Docker. В качестве конкретного примера рассмотрим сервис, предназначенный для обучения конкретного примера, например для обновления объектов с помощью фреймворка TensorFlow. Из-за ограничений FaaS в смысле вычислительных ресурсов, также оперативной и дисковой памяти наряду с ограниченной доступностью библиотек в архитектуре FaaS, вероятно, стоит запустить подобный сервис с помощью CaaS, а не FaaS.

Большая тройка поставщиков облачных сервисов сопровождает свои платформы FaaS обширными наборами программных средств DevOps. Например, при использовании AWS Lambda можно без особых усилий добиться из AWS следующие сервисы:

- AWS X-Ray для отслеживания/наблюдения;
- Amazon CloudWatch для журналирования, предупреждения о проблемах и планирования событий;
- AWS Step Functions для бессерверного согласования технологических процессов;
- AWS Cloud9 для организации среды разработки в браузере.

Как же выбрать между FaaS и CaaS? В одной плоскости ответ на этот вопрос висит от единицы развертывания. FaaS прекрасно подходит там, где требуются только функции с коротким жизненным циклом, малым числом зависимостей и небольшими объемами обработки данных. Если же, напротив, речь идет о «долгоиграющих» процессах с большим количеством зависимостей и высокими требованиями к вычислительным ресурсам, лучше воспользоваться CaaS. Большинство сервисов FaaS сильно ограничены во времени выполнения (не более 15 минут для AWS Lambda), вычислительных ресурсов, объеме оперативной памяти, дискового пространства и количестве HTTP-запросов и ответов. Положительная сторона короткого времени выполнения FaaS — платить нужно только за это краткое время выполнения функции.

Если вы помните рассуждения в главе 12, где серверы превратились в домашних питомцев, крупным рогатым скотом и насекомыми, функции provavelmente всего будет сравнить с бочками-однодневками — они не долго живут, производят какую-то обработку данных и исчезают. Вследствие краткости своего существования функции в FaaS не сохраняют состояние, что необходимо учитывать при проектировании приложений.

Еще один спектр, который следует учитывать при выборе между FaaS и CaaS, — количество и тип взаимодействий в сервисах с прочими сервисами. Например, синхронно инициировать выполнение функций AWS Lambda могут как минимум восемь других сервисов AWS, включая S3, Simple Notification Service (SNS), Simple Email Service (SES) и CloudWatch. Столь обширные возможности взаимодействия упрощают написание регагирующих событий функций, так что в этом случае FaaS явно лучше.

Как вы увидите в этой главе, в основе многих сервисов FaaS сейчас лежит Kubernetes — в настоящее время де-факто стандарт в мире контейнеризации. И хотя в своей единственной функции, «закрулившись» инструментарий FaaS создает контейнеры Docker и размещает их в кластере Kubernetes, как и в ходящемся, так и не в ходящемся под названием. Примеры основных Kubernetes технологий FaaS — OpenFaas и OpenWhisk. При самостоятельном хостинге этих платформ FaaS внезапно становится ясно, что недолго половину слов «бессерверный» составляет слово «сервер». Мы не ожидали, что придется уделять немало внимания обслуживанию кластеров Kubernetes.

Резюмируя слово DevOps и составные части — Dev (разработчики) и Ops (специалисты по эксплуатации), — можно сказать, что бессерверные технологии тяготеют скорее к части Dev. Благодаря им разработчики разбрасывают код происходит более гладко. Основными же частями специалистов по эксплуатации,

особенно при самостоятельном хостинге, — выделение вспомогательной инфраструктуры (иногда очень сложной) платформ FaaS или SaaS. Но, хотя разработчик может счесть, что специлисты по эксплуатации не особо нужны в случае бессерверных технологий (что возможно, хотя по определению это уже не DevOps), при использовании бессерверной платформы остаются немало проблем, решение которых относится к сфере Ops: безопасность, масштабируемость, ограничение ресурсов, планирование вычислительных мощностей, мониторинг, журналирование и хранение. Все это традиционно относится к сфере деятельности специалистов по эксплуатации, но в удивительном новом мире DevOps, о котором идет речь, эти задачи должны делиться сообществом разработчиков и Ops. Команды разработчиков не должны считать свою задачу завершенной по окончании написания кода, должны с гордостью отвечать за доведение сервиса до промышленной эксплуатации, в том числе за обеспечение встроенных возможностей мониторинга, журналирования и трассировки.

Начнем эту главу с примеров развертывания одной и той же функции Python, соответствующей простой конечной точке HTTP, в облачные сервисы большой тройки поставщиков с помощью предлагаемых ими решений FaaS.



Объемы информации, выводимой в результате выполнения некоторых команд из следующих примеров в консоль, довольно велики. В исключении случаев, когда это совершенно необходимо для понимания смысла, мы будем опускать большую часть строк вывода, чтобы сберечь деревья и дать читателю возможность сосредоточиться на тексте.

Развертывание одной и той же функции Python в облака большой тройки поставщиков облачных сервисов

Для AWS и Google мы воспользуемся платформой Serverless, сильно упрощающей развертывание за счет быстрого создания облачных ресурсов, необходимых для среды выполнения FaaS. Платформа Serverless не поддерживает функции Python для Microsoft Azure, так что в этом случае применимы утилиты командной строки Azure.

Установка фреймворка Serverless

Платформа Serverless (<https://serverless.com>) основывается на nodejs. Для ее установки мы воспользуемся npm:

```
$ npm install -g serverless
```

Развертывание функции Python в AWS Lambda

Начнем с клонирования репозитория GitHub с примерами платформы Serverless:

```
$ git clone https://github.com/serverless/examples.git
cd examples/aws-python-simple-http-endpoint
$ export AWS_PROFILE=gheorghiu-net
```

Конечная точка HTTP Python описана в файле `handler.py`:

```
$ cat handler.py
import json
import datetime

def endpoint(event, context):
    current_time = datetime.datetime.now().time()
    body = {
        "message": "Hello, the current time is " + str(current_time)
    }

    response = {
        "statusCode": 200,
        "body": json.dumps(body)
    }

    return response
```

Платформа Serverless применяет декларативный подход для описания создания ресурсов в YAML-файле `serverless.yml`. Вот пример этого файла, в котором объявлена функция `currentTime`, соответствующая функции Python `endpoint` из описанного ранее модуля `handler`:

```
$ cat serverless.yml
service: aws-python-simple-http-endpoint
frameworkVersion: ">=1.2.0 <2.0.0"

provider:
  name: aws
  runtime: python2.7 # or python3.7, supported as of November 2018

functions:
  currentTime:
    handler: handler.endpoint
    events:
      - http:
          path: ping
          method: get
```

Измените версию Python в файле `serverless.yml` на 3.7:

```
provider:
  name: aws
  runtime: python3.7
```

Прозвертываем эту функцию в AWS Lambda с помощью команды `serverless deploy`:

```
$ serverless deploy
Serverless: Packaging service...
Serverless: Excluding development dependencies...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless:
Uploading service aws-python-simple-http-endpoint.zip file to S3 (1.95 KB)...
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
.....
Serverless: Stack update finished...
Service Information
service: aws-python-simple-http-endpoint
stage: dev
region: us-east-1
stack: aws-python-simple-http-endpoint-dev
resources: 10
api keys:
None
endpoints:
GET - https://3a88jzlxm0.execute-api.us-east-1.amazonaws.com/dev/ping
functions:
currentTime: aws-python-simple-http-endpoint-dev-currentTime
layers:
None
Serverless:
Run the "serverless" command to setup monitoring, troubleshooting and testing.
```

Проверяем прозвернутую функцию AWS Lambda, выполняя запрос к ее конечной точке с помощью `curl`:

```
$ curl https://3a88jzlxm0.execute-api.us-east-1.amazonaws.com/dev/ping
{"message": "Hello, the current time is 23:16:30.479690"}%
```

Вызов функции Lambda напрямую с помощью команды `serverless invoke`:

```
$ serverless invoke --function currentTime
{
  "statusCode": 200,
  "body": "{\"message\": \"Hello, the current time is 23:18:38.101006\"}"
}
```

Вызов функции Lambda напрямую с одновременным выводом журналов (он отправляется в AWS CloudWatch Logs):

```
$ serverless invoke --function currentTime --log
{
  "statusCode": 200,
  "body": "{\"message\": \"Hello, the current time is 23:17:11.182463\"}"
}
```

```
-----
START RequestId: 5ac3c9c8-f8ca-4029-84fa-fcf5157b1404 Version: $LATEST
END RequestId: 5ac3c9c8-f8ca-4029-84fa-fcf5157b1404
REPORT RequestId: 5ac3c9c8-f8ca-4029-84fa-fcf5157b1404
Duration: 1.68 ms Billed Duration: 100 ms    Memory Size: 1024 MB
Max Memory Used: 56 MB
```

Обратите внимание на то, что `Billed Duration` (включенная в счет длительность использования) в предыдущем выводе составляет 100 миллисекунд, демонстрируя тем самым одно из преимуществ FaaS — счет выставляется за очень дробные промежутки времени.

Кроме того, мы хотели бы отметить также внимание на то, что весь большой объем работы по созданию ресурсов AWS, необходимых для функции Lambda, берет на себя «затюкал» платформ Serverless. Он создает стек CloudFormation под названием в данном случае `aws-python-simple-http-endpoint-dev`. Просмотреть его можно с помощью утилиты командной строки `aws`:

```
$ aws cloudformation describe-stack-resources \
  --stack-name aws-python-simple-http-endpoint-dev
  --region us-east-1 | jq '.StackResources[].ResourceType'
```

```
"AWS::ApiGateway::Deployment"
"AWS::ApiGateway::Method"
"AWS::ApiGateway::Resource"
"AWS::ApiGateway::RestApi"
"AWS::Lambda::Function"
"AWS::Lambda::Permission"
"AWS::Lambda::Version"
"AWS::Logs::LogGroup"
"AWS::IAM::Role"
"AWS::S3::Bucket"
```

Отметим, что стек CloudFormation включает не менее десяти типов ресурсов, которые в противном случае пришлось бы создавать или связывать друг с другом вручную.

Развертывание функции Python в Google Cloud Functions

В этом разделе мы воспользуемся примером кода из каталога `google-python-simple-http-endpoint` репозитория GitHub с примерами платформы Serverless:

```
$ gcloud projects list
```

| PROJECT_ID | NAME | PROJECT_NUMBER |
|----------------------------|----------------------------|----------------|
| pulumi-gke-testing | Pulumi GKE Testing | 705973980178 |
| pythonfordevops-gke-pulumi | pythonfordevops-gke-pulumi | 787934032650 |

Создаем новый проект GCP:

```
$ gcloud projects create pythonfordevops-cloudfunction
```

Инициализируем локальную среду gcloud:

```
$ gcloud init
Welcome! This command will take you through the configuration of gcloud.

Settings from your current configuration [pythonfordevops-gke-pulumi] are:
compute:
  region: us-west1
  zone: us-west1-c
core:
  account: grig.gheorghiu@gmail.com
  disable_usage_reporting: 'True'
  project: pythonfordevops-gke-pulumi

Pick configuration to use:
[1] Re-initialize this configuration with new settings
[2] Create a new configuration
[3] Switch to and re-initialize existing configuration: [default]
Please enter your numeric choice: 2

Enter configuration name. Names start with a lower case letter and
contain only lower case letters a-z, digits 0-9, and hyphens '-':
pythonfordevops-cloudfunction
Your current configuration has been set to: [pythonfordevops-cloudfunction]

Choose the account you would like to use to perform operations for
this configuration:
[1] grig.gheorghiu@gmail.com
[2] Log in with a new account
Please enter your numeric choice: 1

You are logged in as: [grig.gheorghiu@gmail.com].
Pick cloud project to use:
[1] pulumi-gke-testing
[2] pythonfordevops-cloudfunction
[3] pythonfordevops-gke-pulumi
[4] Create a new project
Please enter numeric choice or text value (must exactly match list
item): 2
Your current project has been set to: [pythonfordevops-cloudfunction].
```

Авторизация локальной командной оболочки в GCP:

```
$ gcloud auth login
```

Результат с помощью фреймворка Serverless — это же конечная точка HTTP Python, что и примере с AWS Lambda, но не этот раз к функции Google Cloud:

```
$ serverless deploy
```

```
Serverless Error -----
```

```
Serverless plugin "serverless-google-cloudfunctions"
initialization errored: Cannot find module 'serverless-google-cloudfunctions'
```

```
Require stack:
```

- /usr/local/lib/node_modules/serverless/lib/classes/PluginManager.js
- /usr/local/lib/node_modules/serverless/lib/Serverless.js
- /usr/local/lib/node_modules/serverless/lib/utils/autocomplete.js
- /usr/local/lib/node_modules/serverless/bin/serverless.js

```
Get Support -----
```

```
Docs:      docs.serverless.com
Bugs:      github.com/serverless/serverless/issues
Issues:    forum.serverless.com
```

```
Your Environment Information -----
```

```
Operating System:    darwin
Node Version:        12.9.0
Framework Version:   1.50.0
Plugin Version:      1.3.8
SDK Version:         2.1.0
```

Ошибка, которую мы наблюдаем, связана с тем, что еще не установлены зависимости в `package.json`:

```
$ cat package.json
{
  "name": "google-python-simple-http-endpoint",
  "version": "0.0.1",
  "description":
    "Example demonstrates how to setup a simple HTTP GET endpoint with python",
  "author": "Sebastian Borza <sebito91@gmail.com>",
  "license": "MIT",
  "main": "handler.py",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "serverless-google-cloudfunctions": "^2.1.0"
  }
}
```

Платформа Serverless написана на node.js, так что ее пакеты необходимо установить с помощью команды `npm install`:

```
$ npm install
```

Попробуем вернуть функцию снова :

```
$ serverless deploy
```

```
Error -----
Error: ENOENT: no such file or directory,
open '/Users/ggheo/.gcloud/keyfile.json'
```

Для генерации ключа учетных данных создайте новую учетную запись сервиса `sa` в консоли учетной записи сервиса IAM GCP. В данном случае для новой учетной записи сервиса был задан адрес электронной почты `sa-255@pythonfordevops-cloudfunction.iam.gserviceaccount.com`.

Создайте ключ учетных данных и скачайте его в файл `~/ .gcloud/pythonfordevops-cloudfunction.json`.

Укажите проект и путь к ключу в файле `serverless.yml`:

```
$ cat serverless.yml
```

```
service: python-simple-http-endpoint
```

```
frameworkVersion: ">=1.2.0 <2.0.0"
```

```
package:
  exclude:
    - node_modules/**
    - .gitignore
    - .git/**
```

```
plugins:
  - serverless-google-cloudfunctions
```

```
provider:
  name: google
  runtime: python37
  project: pythonfordevops-cloudfunction
  credentials: ~/.gcloud/pythonfordevops-cloudfunction.json
```

```
functions:
  currentTime:
    handler: endpoint
    events:
      - http: path
```

Перейдите в консоль менеджера развертывания (deployment manager) GCP и включите его API, а также биллинг для Google Cloud Storage.

Попробуем выполнить развертывание снова :

```
$ serverless deploy
Serverless: Packaging service...
Serverless: Excluding development dependencies...
Serverless: Compiling function "currentTime"...
Serverless: Uploading artifacts...

Error -----

Error: Not Found
at createError
(/Users/ggheo/code/mycode/examples/google-python-simple-http-endpoint/
node_modules/axios/lib/core/createError.js:16:15)
at settle (/Users/ggheo/code/mycode/examples/
google-python-simple-http-endpoint/node_modules/axios/lib/core/settle.js:18:12)
at IncomingMessage.handleStreamEnd
(/Users/ggheo/code/mycode/examples/google-python-simple-http-endpoint/
node_modules/axios/lib/adapters/http.js:202:11)
at IncomingMessage.emit (events.js:214:15)
at IncomingMessage.EventEmitter.emit (domain.js:476:20)
at endReadableNT (_stream_readable.js:1178:12)
at processTicksAndRejections (internal/process/task_queues.js:77:11)

For debugging logs, run again after setting the "SLS_DEBUG=*"
environment variable.
```

Затем в документацию платформы Serverless по учетным записям и ролям GCP (<https://oreil.ly/scsRg>).

Оказывается, что для использования учетной записи сервис для развертывания необходимо назначить ей следующие роли:

- Deployment Manager Editor;
- Storage Admin;
- Logging Admin;
- Cloud Functions Developer.

Затем также в документацию платформы Serverless на предмет того, какие API GCP необходимо включить (<https://oreil.ly/rKiHg>).

Выясняется, что в консоли GCP необходимо включить следующие API:

- Google Cloud Functions;
- Google Cloud Deployment Manager;
- Google Cloud Storage;
- Stackdriver Logging.

Переходим в Deployment Manager в консоли GCP и изучаем сообщения об ошибках:

```
sls-python-simple-http-endpoint-dev failed to deploy

sls-python-simple-http-endpoint-dev has resource warnings
sls-python-simple-http-endpoint-dev-1566510445295:
{"ResourceType": "storage.v1.bucket",
 "ResourceErrorCode": "403",
 "ResourceErrorMessage": {"code": 403,
 "errors": [{"domain": "global", "location": "Authorization",
 "locationType": "header",
 "message": "The project to be billed is associated
with an absent billing account.",
 "reason": "accountDisabled"}]},
 "message": "The project to be billed is associated
with an absent billing account.",
 "statusMessage": "Forbidden",
 "requestPath": "https://www.googleapis.com/storage/v1/b",
 "httpMethod": "POST"}}
```

Уд ляем р звертыв ние sls-python-simple-http-endpoint-dev в консоли GCP и снов выполняем ком нду serverless deploy:

```
$ serverless deploy

Deployed functions
first
https://us-central1-pythonfordevops-cloudfunction.cloudfunctions.net/http
```

Ком нд serverless deploy вновь з вершил сь неуд чно, поскольку изн ч льно мы не включили биллинг для Google Cloud Storage. Р звертыв ние было помечено к к з вершившееся неуд чно для ук з ного в ф йле serverless.yml сервис ,т к что последующие ком нды serverless deploy з верш лись неуд чно д же после включения биллинг Cloud Storage. После уд ления неуд чного р з-вертыв ния в консоли GCP ком нд serverless deploy н ч л р бот ть.

Вызыв ем непосредственно р звернутую функцию Google Cloud:

```
$ serverless invoke --function currentTime
Serverless: v1os7ptg9o48 {
  "statusCode": 200,
  "body": {
    "message": "Received a POST request at 03:46:39.027230"
  }
}
```

Изуч ем журн л с помощью ком нды serverless logs:

```
$ serverless logs --function currentTime
Serverless: Displaying the 4 most recent log(s):
```

```

2019-08-23T03:35:12.419846316Z: Function execution took 20 ms,
finished with status code: 200
2019-08-23T03:35:12.400499207Z: Function execution started
2019-08-23T03:34:27.133107221Z: Function execution took 11 ms,
finished with status code: 200
2019-08-23T03:34:27.122244864Z: Function execution started

```

Проверяем р боту конечной точки функции с помощью curl:

```

$ curl \
https://undefined-pythonfordevops-cloudfunction.cloudfunctions.net/endpoint
<!DOCTYPE html>
<html lang=en>
  <p><b>404.</b> <ins>That's an error.</ins>
  <p>The requested URL was not found on this server.
  <ins>That's all we know.</ins>

```

Поскольку мы не з д ли в ф йле `serverless.yml` регион, URL конечной точки н чин ется с `undefined` и возвр щ ется ошибк .

З д ем в ф йле `serverless.yml` регион `us-central1`:

```

provider:
  name: google
  runtime: python37
  region: us-central1
  project: pythonfordevops-cloudfunction
  credentials: /Users/ggheo/.gcloud/pythonfordevops-cloudfunction.json

```

Р звертыв ем новую версию с помощью ком нды `serverless deploy` и тестируем конечную точку функции, применяя `curl`:

```

$ curl \
https://us-central1-pythonfordevops-cloudfunction.cloudfunctions.net/endpoint
{
  "statusCode": 200,
  "body": {
    "message": "Received a GET request at 03:51:02.560756"
  }
}%

```

Развертывание функции на Python в Azure

Пл тформ Serverless пок что не поддержив ет Azure Functions (<https://oreil.ly/4WQKG>)¹ н Python. Поэтому мы пок жем, к к р звернуть функцию Python Azure с помощью н тивных утилит Azure.

¹ Уже поддержив ет, см. <https://www.serverless.com/blog/serverless-azure-functions-v2/>. — *Примеч. пер.*

Создайте учетную запись Microsoft Azure и установите среду выполнения Microsoft Azure для своей операционной системы в соответствии с официальной документацией Microsoft (<https://oreil.ly/GHS4c>). Если вы работаете на macOS, воспользуйтесь brew:

```
$ brew tap azure/functions
$ brew install azure-functions-core-tools
Создайте новый каталог для кода функции Python:
$ mkdir azure-functions-python
$ cd azure-functions-python
```

Установите Python 3.6, поскольку Azure Functions не поддерживают версию 3.7¹. Создайте и активируйте виртуальную среду:

```
$ brew unlink python
$ brew install \
https://raw.githubusercontent.com/Homebrew/homebrew-core/
f2a764ef944b1080be64bd88dca9a1d80130c558/Formula/python.rb \
--ignore-dependencies

$ python3 -V
Python 3.6.5

$ python3 -m venv .venv
$ source .venv/bin/activate
```

С помощью утилиты func Azure создайте локальный проект функции Azure с названием python-simple-http-endpoint:

```
$ func init python-simple-http-endpoint
Select a worker runtime:
1. dotnet
2. node
3. python
4. powershell (preview)
Choose option: 3
```

Перейдите в только что созданный каталог python-simple-http-endpoint и создайте Azure HTTP Trigger Function с помощью команды func new:

```
$ cd python-simple-http-endpoint
$ func new
Select a template:
1. Azure Blob Storage trigger
2. Azure Cosmos DB trigger
3. Azure Event Grid trigger
```

¹ В момент выпуска русскоязычного издания этой книги Azure Functions поддерживали Python вплоть до версии 3.9 (официальные дистрибутивы CPython). — *Примеч. пер.*

```

4. Azure Event Hub trigger
5. HTTP trigger
6. Azure Queue Storage trigger
7. Azure Service Bus Queue trigger
8. Azure Service Bus Topic trigger
9. Timer trigger
Choose option: 5
HTTP trigger
Function name: [HttpTrigger] currentTime
Writing python-simple-http-endpoint/currentTime/__init__.py
Writing python-simple-http-endpoint/currentTime/function.json
The function "currentTime" was created successfully
from the "HTTP trigger" template.

```

Взгляните на созданный код на языке Python:

```

$ cat currentTime/__init__.py
import logging

import azure.functions as func

def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        return func.HttpResponse(f"Hello {name}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )

```

Выполните полученную функцию на локальной машине:

```

$ func host start

[8/24/19 12:21:35 AM] Host initialized (299ms)
[8/24/19 12:21:35 AM] Host started (329ms)
[8/24/19 12:21:35 AM] Job host started
[8/24/19 12:21:35 AM] INFO: Starting Azure Functions Python Worker.
[8/24/19 12:21:35 AM] INFO: Worker ID: e49c429d-9486-4167-9165-9ecd1757a2b5,
Request ID: 2842271e-a8fe-4643-ab1a-f52381098ae6, Host Address: 127.0.0.1:53952
Hosting environment: Production

```

```
Content root path: python-simple-http-endpoint
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
[8/24/19 12:21:35 AM] INFO: Successfully opened gRPC channel to 127.0.0.1:53952
Http Functions:
    currentTime: [GET,POST] http://localhost:7071/api/currentTime
```

Проверьте ее р боту с другого терминала :

```
$ curl http://127.0.0.1:7071/api/currentTime?name=joe
Hello joe!%
```

Модифицируйте HTTP-обработчик в файле `currentTime/init.py`, включив в его ответ текущее время:

```
import datetime

def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    current_time = datetime.datetime.now().time()
    if name:
        return func.HttpResponse(f"Hello {name},
            the current time is {current_time}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )
```

Проверьте р боту обновленной функции с помощью curl:

```
$ curl http://127.0.0.1:7071/api/currentTime?name=joe
Hello joe, the current time is 17:26:54.256060!%
```

Установите CLI Azure с помощью системы управления пакетами pip:

```
$ pip install azure.cli
```

Создайте в интерактивном режиме с помощью утилиты `az` командной строки Azure Resource Group, Storage Account и Function App. Этот режим позволяет

з действий — интерактивную командную оболочку с выполнением, описаниями команд и примерами. Учтите, что для повторения этого примера вам нужно будет использовать отличное от нашего уникальное название `functionapp`. Возможно, там же вам придется указать другой регион Azure, например `eastus`, поддерживающий бесплатные пробные учетные записи:

```
$ az interactive
az>> login
az>> az group create --name myResourceGroup --location westus2
az>> az storage account create --name griggheorghiusstorage --location westus2 \
--resource-group myResourceGroup --sku Standard_LRS
az>> az functionapp create --resource-group myResourceGroup --os-type Linux \
--consumption-plan-location westus2 --runtime python \
--name pyazure-devops4all7 \
--storage-account griggheorghiusstorage
az>> exit
```

Развертывая проект `functionapp` в Azure с помощью утилиты `func`:

```
$ func azure functionapp publish pyazure-devops4all --build remote
Getting site publishing info...
Creating archive for current directory...
Perform remote build for functions project (--build remote).
Uploading 2.12 KB
```

ВЫВОД ОПУЩЕН

```
Running post deployment command(s)...
Deployment successful.
App container will begin restart within 10 seconds.
Remote build succeeded!
Syncing triggers...
Functions in pyazure-devops4all:
  currentTime - [httpTrigger]
    Invoke url:
      https://pyazure-devops4all.azurewebsites.net/api/
      currenttime?code=b0rN93004cGPcGFKyX7n9HgITTPnHZiGCmjJN/SRsPX7taM7axJbbw==
```

Проверяем работу развернутой функции в Azure посредством запроса к ее конечной точке с помощью `curl`:

```
$ curl "https://pyazure-devops4all.azurewebsites.net/api/currenttime\
?code=b0rN93004cGPcGFKyX7n9HgITTPnHZiGCmjJN/SRsPX7taM7axJbbw=\&name=joe"
Hello joe, the current time is 01:20:32.036097!%
```

Никогда не помешает очистить ненужные облачные ресурсы. В данном случае можно выполнить команду

```
$ az group delete --name myResourceGroup
```

Развертывание функции на Python на самохостируемых FaaS-платформах

Как упоминалось ранее в этой главе, многие FaaS-платформы используют кластер Kubernetes. В числе преимуществ такого подхода следующее: развертываемые функции выполняются как обычные контейнеры Docker в Kubernetes, что позволяет применять обычный инструментарий Kubernetes, особенно для наблюдения (мониторинг, журналирование и трассировки). Еще одно потенциальное преимущество — экономия денег. Благодаря выполнению бессерверных функций в виде контейнеров в уже существующем кластере Kubernetes можно использовать имеющиеся ресурсы кластера, не платя за каждый вызов функции, как было бы при развертывании функций сторонней FaaS-платформе.

В этом разделе мы рассмотрим одну из таких платформ — OpenFaaS (<https://www.openfaas.com>). В число логичных FaaS-платформ, «под капотом» которых скрыт Kubernetes, входят, в частности:

- Kubeless (<https://kubeless.io>);
- Fn Project (<https://fnproject.io>) (технология, лежащая в основе FaaS-платформы от Oracle — Oracle Functions);
- Fission (<https://fission.io>);
- Apache OpenWhisk (<https://openwhisk.apache.org>).

Развертывание функции на Python в OpenFaaS

Для этого пример воспользуемся облегченным дистрибутивом Kubernetes от компании Rancher — k3s, чтобы продемонстрировать многообразие доступных в экосистеме Kubernetes средств.

Начнем с запуска утилиты k3sup (<https://oreil.ly/qK0xJ>) для выделения кластера Kubernetes k3s на EC2-инстанции под управлением Ubuntu.

Скачайте и установите k3sup:

```
$ curl -sLS https://get.k3sup.dev | sh
$ sudo cp k3sup-darwin /usr/local/bin/k3sup
```

Проверьте возможность подключения по SSH к удаленному EC2-инстансу:

```
$ ssh ubuntu@35.167.68.86 date
Sat Aug 24 21:38:57 UTC 2019
```


Установите k3s с помощью команды `k3sup install`:

```
$ k3sup install --ip 35.167.68.86 --user ubuntu
```

ВЫВОД ОПУЩЕН

Saving file to: kubeconfig

Взглянем на содержимое файла `kubeconfig`:

```
$ cat kubeconfig
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: BASE64_FIELD
    server: https://35.167.68.86:6443
    name: default
contexts:
- context:
    cluster: default
    user: default
    name: default
current-context: default
kind: Config
preferences: {}
users:
- name: default
  user:
    password: OBFUSCATED
    username: admin
```

Сделайте так, чтобы переменная среды `KUBECONFIG` указывала на локальный файл `kubeconfig`, и попробуйте выполнить команды `kubectl` для удаленного кластера k3s:

```
$ export KUBECONFIG=./kubeconfig
```

```
$ kubectl cluster-info
```

```
Kubernetes master is running at https://35.167.68.86:6443
CoreDNS is running at
https://35.167.68.86:6443/api/v1/namespaces/kube-system/
services/kube-dns:dns/proxy
```

To further debug and diagnose cluster problems, use
'`kubectl cluster-info dump`'.

```
$ kubectl get nodes
```

| NAME | STATUS | ROLES | AGE | VERSION |
|---------------|--------|--------|-----|---------------|
| ip-10-0-0-185 | Ready | master | 10m | v1.14.6-k3s.1 |

Следующий этап — установка платформы OpenFaaS на кластере Kubernetes k3s.

Установите утилиту `faas-cli` в локальной системе под управлением macOS:

```
$ brew install faas-cli
```

Создайте привилегированный доступ RBAC для Tiller — серверной части Helm:

```
$ kubectl -n kube-system create sa tiller \
  && kubectl create clusterrolebinding tiller \
    --clusterrole cluster-admin \
    --serviceaccount=kube-system:tiller
serviceaccount/tiller created
clusterrolebinding.rbac.authorization.k8s.io/tiller created
```

Установите Tiller с помощью команды `helm init`¹:

```
$ helm init --skip-refresh --upgrade --service-account tiller
```

Скчайте, настройте и установите чарт Helm для OpenFaaS:

```
$ wget \
https://raw.githubusercontent.com/openfaas/faas-netes/master/namespaces.yml
```

```
$ cat namespaces.yml
apiVersion: v1
kind: Namespace
metadata:
  name: openfaas
  labels:
    role: openfaas-system
    access: openfaas-system
    istio-injection: enabled
```

```
apiVersion: v1
kind: Namespace
metadata:
  name: openfaas-fn
  labels:
    istio-injection: enabled
    role: openfaas-fn
```

```
$ kubectl apply -f namespaces.yml
namespace/openfaas created
namespace/openfaas-fn created
```

```
$ helm repo add openfaas https://openfaas.github.io/faas-netes/
"openfaas" has been added to your repositories
```

¹ См. сноску на с. 383 в главе 12. — *Примеч. пер.*

Генерируем случайный пароль для минимальной аутентификации шлюза OpenFaaS:

```
$ PASSWORD=$(head -c 12 /dev/urandom | shasum | cut -d' ' -f1)

$ kubectl -n openfaas create secret generic basic-auth \
--from-literal=basic-auth-user=admin \
--from-literal=basic-auth-password="$PASSWORD"
secret/basic-auth created
```

Развертываем OpenFaaS путем установки чарта Helm:

```
$ helm repo update \
&& helm upgrade openfaas --install openfaas/openfaas \
--namespace openfaas \
--set basic_auth=true \
--set serviceType=LoadBalancer \
--set functionNamespace=openfaas-fn
```

ВЫВОД ОПУЩЕН

NOTES:

To verify that openfaas has started, run:

```
kubectl --namespace=openfaas get deployments -l "release=openfaas,app=openfaas"
```



Подобную минимальную аутентификацию без TLS следует использовать только для экспериментов/изучения. Любую более или менее живую среду нужно не строить так, чтобы учетные данные передавались через защищенное TLS-соединение.

Проверяем, какие сервисы запущены в пространстве имен openfaas:

```
$ kubectl get service -n openfaas
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) |
|-------------------|--------------|---------------|-------------|----------------|
| alertmanager | ClusterIP | 10.43.193.61 | <none> | 9093/TCP |
| basic-auth-plugin | ClusterIP | 10.43.83.12 | <none> | 8080/TCP |
| gateway | ClusterIP | 10.43.7.46 | <none> | 8080/TCP |
| gateway-external | LoadBalancer | 10.43.91.91 | 10.0.0.185 | 8080:31408/TCP |
| nats | ClusterIP | 10.43.33.153 | <none> | 4222/TCP |
| prometheus | ClusterIP | 10.43.122.184 | <none> | 9090/TCP |

Выполняем перенаправление с порта 8080 удаленного инстанса локальный порт 8080:

```
$ kubectl port-forward -n openfaas svc/gateway 8080:8080 &
[1] 29183
Forwarding from 127.0.0.1:8080 -> 8080
```

З йдите в бр узере н веб-UI OpenFaaS по дресу `http://localhost:8080/` и войдите в систему с именем пользо теля `admin` и п ролем `$PASSWORD`.

Д лее мы созд ем функцию OpenFaaS н Python. Для созд ния новой функции OpenFaaS `hello-python` з действуем утилиту `faas-cli`:

```
$ faas-cli new --lang python hello-python
Folder: hello-python created.
Function created in folder: hello-python
Stack file written: hello-python.yml
```

З глянем в ф йл конфигу рции функции `hello-python`:

```
$ cat hello-python.yml
version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:8080
functions:
  hello-python:
    lang: python
    handler: ./hello-python
    image: hello-python:latest
```

Смотрим н содержимое втом тически созд нного к т лог `hello-python`:

```
$ ls -la hello-python
total 8
drwx----- 4 ggheo staff 128 Aug 24 15:16 .
drwxr-xr-x  8 ggheo staff 256 Aug 24 15:16 ..
-rw-r--r--  1 ggheo staff 123 Aug 24 15:16 handler.py
-rw-r--r--  1 ggheo staff   0 Aug 24 15:16 requirements.txt
```

```
$ cat hello-python/handler.py
def handle(req):
    """Обрабатывает запрос к функции
    Args:
        req (str): тело запроса
    """
    return req
```

Отред ктируем ф йл `handler.py`, слегк изменив код, выводящий текущее время, из `simple-http-example` пл тформы Serverless:

```
$ cat hello-python/handler.py
import json
import datetime

def handle(req):
    """Обрабатывает запрос к функции
```

```

Args:
    req (str): тело запроса
"""

current_time = datetime.datetime.now().time()
body = {
    "message": "Received a {} at {}".format(req, str(current_time))
}

response = {
    "statusCode": 200,
    "body": body
}
return json.dumps(response, indent=4)

```

Следующий шаг — сборка Python-функции OpenFaaS. Выполним сборку образа Docker на основе сгенерированного в том же каталоге Dockerfile с помощью команды сборки `faas-cli`:

```

$ faas-cli build -f ./hello-python.yml
[0] > Building hello-python.
Clearing temporary build folder: ./build/hello-python/
Preparing ./hello-python/ ./build/hello-python/function
Building: hello-python:latest with python template. Please wait..
Sending build context to Docker daemon 8.192kB
Step 1/29 : FROM openfaas/classic-watchdog:0.15.4 as watchdog

```

ВЫВОД КОМАНДЫ СБОРКИ DOCKER ОПУЩЕН

```

Successfully tagged hello-python:latest
Image: hello-python:latest built.
[0] < Building hello-python done.
[0] worker done.

```

Проверяем, появился ли образ в логах контейнерной машины образа Docker:

```

$ docker images | grep hello-python
hello-python          latest
05b2c37407e1         29 seconds ago       75.5MB

```

Помечем полученный образ Docker тегом и поместим его в реестр Docker Hub для использования в дальнейшем кластере Kubernetes:

```
$ docker tag hello-python:latest griggheo/hello-python:latest
```

Вносим изменения в файл `hello-python.yml`:

```
image: griggheo/hello-python:latest
```

Помещаем образ в Docker Hub с помощью команды `faas-cli push`:

```
$ faas-cli push -f ./hello-python.yml
[0] > Pushing hello-python [griggheo/hello-python:latest].
The push refers to repository [docker.io/griggheo/hello-python]
latest: digest:
sha256:27e1fbb7f68bb920a6ff8d3baf1fa3599ae92e0b3c607daac3f8e276aa7f3ae3
size: 4074
[0] < Pushing hello-python [griggheo/hello-python:latest] done.
[0] worker done.
```

Выполняем развертывание функции OpenFaaS на Python в удаленном кластере K3s с помощью команды `faas-cli deploy`:

```
$ faas-cli deploy -f ./hello-python.yml
Deploying: hello-python.
WARNING! Communication is not secure, please consider using HTTPS.
Letsencrypt.org offers free SSL/TLS certificates.
Handling connection for 8080
```

```
unauthorized access, run "faas-cli login"
to setup authentication for this server
```

```
Function 'hello-python' failed to deploy with status code: 401
```

Получаем учетные данные аутентификации с помощью команды `faas-cli login`:

```
$ echo -n $PASSWORD | faas-cli login -g http://localhost:8080 \
-u admin --password-stdin
Calling the OpenFaaS server to validate the credentials...
Handling connection for 8080
WARNING! Communication is not secure, please consider using HTTPS.
Letsencrypt.org offers free SSL/TLS certificates.
credentials saved for admin http://localhost:8080
```

Вносим следующие изменения в файл `hello-python.yml`:

```
gateway: http://localhost:8080
```

Поскольку наш образ ботчик возвращает JSON, добавим в файл `hello-python.yml` следующие строки кода:

```
environment:
  content_type: application/json
```

Содержимое файла `hello-python.yml` теперь выглядит так:

```
$ cat hello-python.yml
version: 1.0
provider:
  name: openfaas
  gateway: http://localhost:8080
```

```
functions:
  hello-python:
    lang: python
    handler: ./hello-python
    image: griggheo/hello-python:latest
    environment:
      content_type: application/json
```

Снова запустите команду `faas-cli deploy`:

```
$ faas-cli deploy -f ./hello-python.yml
Deploying: hello-python.
WARNING! Communication is not secure, please consider using HTTPS.
Letsencrypt.org offers free SSL/TLS certificates.
Handling connection for 8080
Handling connection for 8080

Deployed. 202 Accepted.
URL: http://localhost:8080/function/hello-python
```

При необходимости изменения кода собрать и заново развернуть функцию можно с помощью следующих команд (обратите внимание на то, что команда `faas-cli remove` удаляет текущую версию функции):

```
$ faas-cli build -f ./hello-python.yml
$ faas-cli push -f ./hello-python.yml
$ faas-cli remove -f ./hello-python.yml
$ faas-cli deploy -f ./hello-python.yml
```

Проверим работу развернутой функции с помощью `curl`:

```
$ curl localhost:8080/function/hello-python --data-binary 'hello'
Handling connection for 8080
{
  "body": {
    "message": "Received a hello at 22:55:05.225295"
  },
  "statusCode": 200
}
```

Проверяем еще и непосредственным вызовом нашей функции с помощью `faas-cli`:

```
$ echo -n "hello" | faas-cli invoke hello-python
Handling connection for 8080
{
  "body": {
    "message": "Received a hello at 22:56:23.549509"
  },
  "statusCode": 200
}
```

Следующий пример будет полноценнее. Мы продемонстрируем, как выделить с помощью AWS CDK сразу несколько функций Lambda, работающих с API Gateway, обеспечивающих (CRUD) REST-доступ для создания/чтения/обновления/удаления хранимых в таблице DynamoDB элементов `todo`. Мы также покажем, как выполнить нагрузочное тестирование нашего API REST с помощью развернутых в AWS Fargate контейнеров, а также утилиты нагрузочного тестирования Locust. Контейнеры Fargate мы также выделим с помощью AWS CDK.

Выделение таблиц DynamoDB, функций Lambda и методов API Gateway с помощью AWS CDK

Мы уже мельком упоминали AWS CDK в главе 10. AWS CDK — программный продукт, с помощью которого можно описывать желаемое состояние инфраструктуры в виде нестоящего кода (в нестоящее время поддерживаются языки программирования TypeScript и Python) вместо файлов описаний в формате YAML (как в платформе Serverless).

Установите интерфейс командной строки CDK на глобальном уровне (в зависимости от вашей операционной системы может потребоваться выполнить следующую команду с `sudo`):

```
$ npm install cdk -g
```

Создайте каталог для приложения CDK:

```
$ mkdir cdk-lambda-dynamodb-fargate
$ cd cdk-lambda-dynamodb-fargate
```

Создайте пример Python-приложения с помощью `cdk init`:

```
$ cdk init app --language=python
Applying project template app for python
Executing Creating virtualenv...

# Welcome to your CDK Python project!
This is a blank project for Python development with CDK.
The `cdk.json` file tells the CDK Toolkit how to execute your app.
```

Выводим список созданных файлов:

```
$ ls -la
total 40
drwxr-xr-x  9 ggheo  staff  288 Sep  2 10:10 .
drwxr-xr-x 12 ggheo  staff  384 Sep  2 10:10 ..
drwxr-xr-x  6 ggheo  staff  192 Sep  2 10:10 .env
```



```
-rw-r--r-- 1 ggheo staff 1651 Sep  2 10:10 README.md
-rw-r--r-- 1 ggheo staff  252 Sep  2 10:10 app.py
-rw-r--r-- 1 ggheo staff   32 Sep  2 10:10 cdk.json
drwxr-xr-x 4 ggheo staff  128 Sep  2 10:10 cdk_lambda_dynamodb_fargate
-rw-r--r-- 1 ggheo staff    5 Sep  2 10:10 requirements.txt
-rw-r--r-- 1 ggheo staff 1080 Sep  2 10:10 setup.py
```

Изучим содержимое основного файла `app.py`:

```
$ cat app.py
#!/usr/bin/env python3

from aws_cdk import core

from cdk_lambda_dynamodb_fargate.cdk_lambda_dynamodb_fargate_stack \
import CdkLambdaDynamodbFargateStack

app = core.App()
CdkLambdaDynamodbFargateStack(app, "cdk-lambda-dynamodb-fargate")

app.synth()
```

Программа CDK состоит из приложения (`app`), которое может содержать один или несколько стеков (`stacks`). Стек соответствует объекту `CloudFormation`.

Изучим модуль с описанием стека CDK:

```
$ cat cdk_lambda_dynamodb_fargate/cdk_lambda_dynamodb_fargate_stack.py
from aws_cdk import core

class CdkLambdaDynamodbFargateStack(core.Stack):

    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # Место для кода с описанием вашего стека
```

Поскольку у нас будет два стека, один для ресурсов DynamoDB/Lambda/API Gateway, второй для ресурсов Fargate, переименовав файл `cdk_lambda_dynamodb_fargate/cdk_lambda_dynamodb_fargate_stack.py` в `cdk_lambda_dynamodb_fargate/cdk_lambda_dynamodb_stack.py`, класс `CdkLambdaDynamodbFargateStack` — в `CdkLambdaDynamodbStack`.

Кроме того, необходимо, чтобы файл `app.py` ссылался на измененные названия модуля и класс:

```
from cdk_lambda_dynamodb_fargate.cdk_lambda_dynamodb_stack \
import CdkLambdaDynamodbStack

CdkLambdaDynamodbStack(app, "cdk-lambda-dynamodb")
```

Активируем виртуальную среду:

```
$ source .env/bin/activate
```

Мы возьмем средство сокращения URL из примеров CDK (<https://oreil.ly/q2dDF>) и модифицируем его, воспользовавшись кодом из примера API REST на Python для платформы Serverless для AWS (https://oreil.ly/o_gxS), и получим в результате API REST для создания и вывода списка, а также обновления и удаления элементов `todo`. Для хранения данных применим DynamoDB от Amazon.

Изучим содержимое файла `serverless.yml` из `examples/aws-python-rest-api-with-dynamodb` и просмотрим его с помощью команды `serverless`, чтобы посмотреть, какие ресурсы AWS будут созданы:

```
$ pwd
~/code/examples/aws-python-rest-api-with-dynamodb

$ serverless deploy
Serverless: Stack update finished...
Service Information
service: serverless-rest-api-with-dynamodb
stage: dev
region: us-east-1
stack: serverless-rest-api-with-dynamodb-dev
resources: 34
api keys:
  None
endpoints:
POST - https://tbst34m2b7.execute-api.us-east-1.amazonaws.com/dev/todos
GET  - https://tbst34m2b7.execute-api.us-east-1.amazonaws.com/dev/todos
GET  - https://tbst34m2b7.execute-api.us-east-1.amazonaws.com/dev/todos/{id}
PUT  - https://tbst34m2b7.execute-api.us-east-1.amazonaws.com/dev/todos/{id}
DELETE - https://tbst34m2b7.execute-api.us-east-1.amazonaws.com/dev/todos/{id}
functions:
  create: serverless-rest-api-with-dynamodb-dev-create
  list: serverless-rest-api-with-dynamodb-dev-list
  get: serverless-rest-api-with-dynamodb-dev-get
  update: serverless-rest-api-with-dynamodb-dev-update
  delete: serverless-rest-api-with-dynamodb-dev-delete
layers:
  None
Serverless: Run the "serverless" command to setup monitoring, troubleshooting
and
testing.
```

Предыдущая команда создала пять функций AWS Lambda, один API Gateway и одну таблицу DynamoDB.

В к т логе CDK доб вляем в созд в емый стек т блицу DynamoDB:

```
$ pwd
~/code/devops/serverless/cdk-lambda-dynamodb-fargate

$ cat cdk_lambda_dynamodb_fargate/cdk_lambda_dynamodb_stack.py
from aws_cdk import core
from aws_cdk import aws_dynamodb

class CdkLambdaDynamodbStack(core.Stack):

    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)
        # Описываем таблицу для хранения элементов Todo
        table = aws_dynamodb.Table(self, "Table",
                                   partition_key=aws_dynamodb.Attribute(
                                       name="id",
                                       type=aws_dynamodb.AttributeType.STRING),
                                   read_capacity=10,
                                   write_capacity=5)
```

Уст новим нужные модули Python:

```
$ cat requirements.txt
-e .
aws-cdk.core
aws-cdk.aws-dynamodb

$ pip install -r requirements.txt
```

С помощью ком нды cdk synth созд дим стек CloudFormation:

```
$ export AWS_PROFILE=gheorghiu-net
$ cdk synth
```

Перед ем конструктору CdkLambdaDynamodbStack в app.py переменную variable, содержащую значение для регион :

```
app_env = {"region": "us-east-2"}
CdkLambdaDynamodbStack(app, "cdk-lambda-dynamodb", env=app_env)
```

Снов выполняем ком нду cdk synth:

```
$ cdk synth
Resources:
  TableCD117FA1:
    Type: AWS::DynamoDB::Table
    Properties:
```

```

KeySchema:
  - AttributeName: id
    KeyType: HASH
AttributeDefinitions:
  - AttributeName: id
    AttributeType: S
ProvisionedThroughput:
  ReadCapacityUnits: 10
  WriteCapacityUnits: 5
UpdateReplacePolicy: Retain
DeletionPolicy: Retain
Metadata:
  aws:cdk:path: cdk-lambda-dynamodb-fargate/Table/Resource
CDKMetadata:
  Type: AWS::CDK::Metadata
  Properties: /
    Modules: aws-cdk=1.6.1,
      @aws-cdk/aws-applicationautoscaling=1.6.1,
      @aws-cdk/aws-autoscaling-common=1.6.1,
      @aws-cdk/aws-cloudwatch=1.6.1,
      @aws-cdk/aws-dynamodb=1.6.1,
      @aws-cdk/aws-iam=1.6.1,
      @aws-cdk/core=1.6.1,
      @aws-cdk/cx-api=1.6.1,@aws-cdk/region-info=1.6.1,
      jsii-runtime=Python/3.7.4

```

Результатом выполнения команды `cdk deploy`:

```

$ cdk deploy
cdk-lambda-dynamodb-fargate: deploying...
cdk-lambda-dynamodb-fargate: creating CloudFormation changeset...
0/3 | 11:12:25 AM | CREATE_IN_PROGRESS | AWS::DynamoDB::Table |
Table (TableCD117FA1)
0/3 | 11:12:25 AM | CREATE_IN_PROGRESS | AWS::CDK::Metadata |
CDKMetadata
0/3 | 11:12:25 AM | CREATE_IN_PROGRESS | AWS::DynamoDB::Table |
Table (TableCD117FA1) Resource creation Initiated
0/3 | 11:12:27 AM | CREATE_IN_PROGRESS | AWS::CDK::Metadata |
CDKMetadata Resource creation Initiated
1/3 | 11:12:27 AM | CREATE_COMPLETE | AWS::CDK::Metadata |
CDKMetadata
2/3 | 11:12:56 AM | CREATE_COMPLETE | AWS::DynamoDB::Table |
Table (TableCD117FA1)
3/3 | 11:12:57 AM | CREATE_COMPLETE | AWS::CloudFormation::Stack |
cdk-lambda-dynamodb-fargate

```

Stack ARN:

```

arn:aws:cloudformation:us-east-2:200562098309:stack/
cdk-lambda-dynamodb/3236a8b0-cdad-11e9-934b-0a7dfa8cb208

```

Следующий этап — добавление в стек функций Lambda и ресурс API Gateway.

В каталоге код CDK создайте каталог `lambda` и скопируйте туда модули Python из примера API REST на Python для платформы Serverless для AWS (<https://oreil.ly/mRSjn>):

```
$ pwd
~/code/devops/serverless/cdk-lambda-dynamodb-fargate

$ mkdir lambda
$ cp ~/code/examples/aws-python-rest-api-with-dynamodb/todos/* lambda
$ ls -la lambda
total 48
drwxr-xr-x  9 ggheo  staff   288 Sep  2 10:41 .
drwxr-xr-x 10 ggheo  staff  320 Sep  2 10:19 ..
-rw-r--r--  1 ggheo  staff    0 Sep  2 10:41 __init__.py
-rw-r--r--  1 ggheo  staff  822 Sep  2 10:41 create.py
-rw-r--r--  1 ggheo  staff  288 Sep  2 10:41 decimalencoder.py
-rw-r--r--  1 ggheo  staff  386 Sep  2 10:41 delete.py
-rw-r--r--  1 ggheo  staff  535 Sep  2 10:41 get.py
-rw-r--r--  1 ggheo  staff  434 Sep  2 10:41 list.py
-rw-r--r--  1 ggheo  staff 1240 Sep  2 10:41 update.py
```

Добавляем требуемые модули в файл `requirements.txt` и установим их с помощью системы управления пакетами `pip`:

```
$ cat requirements.txt
-e .
aws-cdk.core
aws-cdk.aws-dynamodb
aws-cdk.aws-lambda
aws-cdk.aws-apigateway

$ pip install -r requirements.txt
```

Создаем конструкции Lambda и API Gateway в модуле стека:

```
$ cat cdk_lambda_dynamodb_fargate/cdk_lambda_dynamodb_stack.py
from aws_cdk import core
from aws_cdk.core import App, Construct, Duration
from aws_cdk import aws_dynamodb, aws_lambda, aws_apigateway

class CdkLambdaDynamodbStack(core.Stack):

    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # Описываем таблицу для хранения элементов Todo
        table = aws_dynamodb.Table(self, "Table",
```

```

        partition_key=aws_dynamodb.Attribute(
            name="id",
            type=aws_dynamodb.AttributeType.STRING),
        read_capacity=10,
        write_capacity=5)

# Описываем функции Lambda
list_handler = aws_lambda.Function(self, "TodoListFunction",
    code=aws_lambda.Code.asset("./lambda"),
    handler="list.list",
    timeout=Duration.minutes(5),
    runtime=aws_lambda.Runtime.PYTHON_3_7)

create_handler = aws_lambda.Function(self, "TodoCreateFunction",
    code=aws_lambda.Code.asset("./lambda"),
    handler="create.create",
    timeout=Duration.minutes(5),
    runtime=aws_lambda.Runtime.PYTHON_3_7)

get_handler = aws_lambda.Function(self, "TodoGetFunction",
    code=aws_lambda.Code.asset("./lambda"),
    handler="get.get",
    timeout=Duration.minutes(5),
    runtime=aws_lambda.Runtime.PYTHON_3_7)

update_handler = aws_lambda.Function(self, "TodoUpdateFunction",
    code=aws_lambda.Code.asset("./lambda"),
    handler="update.update",
    timeout=Duration.minutes(5),
    runtime=aws_lambda.Runtime.PYTHON_3_7)

delete_handler = aws_lambda.Function(self, "TodoDeleteFunction",
    code=aws_lambda.Code.asset("./lambda"),
    handler="delete.delete",
    timeout=Duration.minutes(5),
    runtime=aws_lambda.Runtime.PYTHON_3_7)

# Передаем имя таблицы во все обработчики через переменную среды
# и предоставляем обработчикам права доступа на чтение/запись
# этой таблицы.
handler_list = [
    list_handler,
    create_handler,
    get_handler,
    update_handler,
    delete_handler
]
for handler in handler_list:
    handler.add_environment('DYNAMODB_TABLE', table.table_name)
    table.grant_read_write_data(handler)

```

```

# Описываем конечную точку API
api = aws_apigateway.LambdaRestApi(self, "TodoApi",
    handler=list_handler,
    proxy=False)

# Описываем LambdaIntegrations
list_lambda_integration = \
    aws_apigateway.LambdaIntegration(list_handler)
create_lambda_integration = \
    aws_apigateway.LambdaIntegration(create_handler)
get_lambda_integration = \
    aws_apigateway.LambdaIntegration(get_handler)
update_lambda_integration = \
    aws_apigateway.LambdaIntegration(update_handler)
delete_lambda_integration = \
    aws_apigateway.LambdaIntegration(delete_handler)

# Описываем модель API REST и связываем методы с LambdaIntegrations
api.root.add_method('ANY')
todos = api.root.add_resource('todos')
todos.add_method('GET', list_lambda_integration)
todos.add_method('POST', create_lambda_integration)
todo = todos.add_resource('{id}')
todo.add_method('GET', get_lambda_integration)
todo.add_method('PUT', update_lambda_integration)
todo.add_method('DELETE', delete_lambda_integration)

```

Стоит отметить несколько особенностей только что приведенного код .

- Мы воспользовались методом `add_environment` объектов `handler` для передачи переменной среды `DYNAMODB_TABLE`, применяемой в Python-коде для функций Lambda, а также для нее значение `table.table_name`. Однако значение таблицы DynamoDB не типизировано, поэтому формирование неизвестно, так как CDK меняет его на токен, значение которого не является приложением (см. подробности в документации по токенам (<https://oreil.ly/XfdEU>)).
- Мы использовали все возможности простого языка программирования, цикл `for`, когда проходили в цикле по всем объектам. И хотя это может показаться очевидным, но все равно заслуживает упоминания, поскольку циклы и переданные переменные реализованы в утилитарный тип «интерпретировать код», так как Terraform, очень неуклюже, если вообще реализованы.
- Мы описали HTTP-методы (GET, POST, PUT, DELETE), связанные с различными конечными точками API Gateway и соответствующей функцией Lambda для каждого из них.

Р звертыв ем стек с помощью команды `cdk deploy`:

```
$ cdk deploy
cdk-lambda-dynamodb-fargate failed: Error:
This stack uses assets, so the toolkit stack must be deployed
to the environment
(Run "cdk bootstrap aws://unknown-account/us-east-2")
```

Исправляем ошибку посредством выполнения `cdk bootstrap`:

```
$ cdk bootstrap
Bootstrapping environment aws://ACCOUNTID/us-east-2...
CDKToolkit: creating CloudFormation changeset...
Environment aws://ACCOUNTID/us-east-2 bootstrapped.
```

Снова развертываем стек CDK:

```
$ cdk deploy
ВЫВОД ОПУЩЕН
```

```
Outputs:
cdk-lambda-dynamodb.TodoApiEndpointC1E16B6C =
https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/
```

```
Stack ARN:
arn:aws:cloudformation:us-east-2:ACCOUNTID:stack/cdk-lambda-dynamodb/
15a66bb0-cdba-11e9-aef9-0ab95d3a5528
```

Следующий шаг — тестирование API REST с помощью `curl`.

Сначала создаем новый элемент `todo`:

```
$ curl -X \
POST https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/todos \
--data '{ "text": "Learn CDK" }'
{"id": "19d55d5a-cdb4-11e9-9a8f-9ed29c44196e", "text": "Learn CDK",
"checked": false,
"createdAt": "1567450902.262834",
"updatedAt": "1567450902.262834"}%
```

Создаем еще один элемент `todo`:

```
$ curl -X \
POST https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/todos \
--data '{ "text": "Learn CDK with Python" }'
{"id": "58a992c6-cdb4-11e9-9a8f-9ed29c44196e", "text": "Learn CDK with Python",
"checked": false,
"createdAt": "1567451007.680936",
"updatedAt": "1567451007.680936"}%
```


Пыт емся получить подробные сведения о только что созд нном элементе по его ID:

```
$ curl \
https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/
prod/todos/58a992c6-cdb4-11e9-9a8f-9ed29c44196e
{"message": "Internal server error"}%
```

Просм трив ем журн л CloudWatch Logs для функции Lambda `TodoGetFunction`:

```
[ERROR] Runtime.ImportModuleError:
Unable to import module 'get': No module named 'todos'
```

Чтобы испр вить эту проблему, измените в ф йле `lambda/get.py` строку:

```
from todos import decimalencoder
```

н следующую:

```
import decimalencoder
```

Снов р звертыв ем стек с помощью ком нды `cdk deploy`.

Опять пыт емся получить сведения об элементе `todo` с помощью `curl`:

```
$ curl \
https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/
prod/todos/58a992c6-cdb4-11e9-9a8f-9ed29c44196e
{"checked": false, "createdAt": "1567451007.680936",
"text": "Learn CDK with Python",
"id": "58a992c6-cdb4-11e9-9a8f-9ed29c44196e",
"updatedAt": "1567451007.680936"}
```

Производим з мену н `import decimalencoder` во всех модулях из к т лог `lambda`, в которых требуется модуль `decimalencoder`, и р звертыв ем стек з ново с помощью `cdk deploy`.

Выводим список всех `todo`, форм тируя выводимые результ ты с помощью утилиты `jq`:

```
$ curl \
https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/todos | jq
[
  {
    "checked": false,
    "createdAt": "1567450902.262834",
    "text": "Learn CDK",
```

```

    "id": "19d55d5a-cdb4-11e9-9a8f-9ed29c44196e",
    "updatedAt": "1567450902.262834"
  },
  {
    "checked": false,
    "createdAt": "1567451007.680936",
    "text": "Learn CDK with Python",
    "id": "58a992c6-cdb4-11e9-9a8f-9ed29c44196e",
    "updatedAt": "1567451007.680936"
  }
]

```

Уд ляем элемент `todo` и убежд емся, что в списке его больше нет:

```

$ curl -X DELETE \
https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/todos/
19d55d5a-cdb4-11e9-9a8f-9ed29c44196e
$ curl https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/todos | jq
[
  {
    "checked": false,
    "createdAt": "1567451007.680936",
    "text": "Learn CDK with Python",
    "id": "58a992c6-cdb4-11e9-9a8f-9ed29c44196e",
    "updatedAt": "1567451007.680936"
  }
]

```

А теперь проверяем обновление существующего элемент `todo` с помощью `curl`:

```

$ curl -X \
PUT https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/todos/
58a992c6-cdb4-11e9-9a8f-9ed29c44196e \
--data '{ "text": "Learn CDK with Python by reading the PyForDevOps book" }'
{"message": "Internal server error"}%

```

Смотрим журн л CloudWatch для связ нной с этой конечной точкой функции Lambda:

```

[ERROR] Exception: Couldn't update the todo item.
Traceback (most recent call last):
  File "/var/task/update.py", line 15, in update
    raise Exception("Couldn't update the todo item.")

```

Меняем код проверочного тест в `lambda/update.py` н следующий:

```

data = json.loads(event['body'])
if 'text' not in data:
    logging.error("Validation Failed")
    raise Exception("Couldn't update the todo item.")

```

Меняем также значение `checked` на `True`, поскольку мы уже видели сообщение, которое хотим обновить:

```
ExpressionAttributeValues={
    ':text': data['text'],
    ':checked': True,
    ':updatedAt': timestamp,
},
```

Снова развертываем стек с помощью команды `cdk deploy`.

Проверяем обновление элемента `todo` с помощью `curl`:

```
$ curl -X \
PUT https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/todos/
58a992c6-cdb4-11e9-9a8f-9ed29c44196e \
--data '{ "text": "Learn CDK with Python by reading the PyForDevOps book"}'
{"checked": true, "createdAt": "1567451007.680936",
"text": "Learn CDK with Python by reading the PyForDevOps book",
"id": "58a992c6-cdb4-11e9-9a8f-9ed29c44196e", "updatedAt": 1567453288764}%
```

Выводим список элементов `todo`, чтобы проверить, как прошло обновление:

```
$ curl https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/todos | jq
[
  {
    "checked": true,
    "createdAt": "1567451007.680936",
    "text": "Learn CDK with Python by reading the PyForDevOps book",
    "id": "58a992c6-cdb4-11e9-9a8f-9ed29c44196e",
    "updatedAt": 1567453288764
  }
]
```

Следующий шаг — выделение контейнеров AWS Fargate для негрузочного тестирования только что развернутого нами API REST. Каждый контейнер представляет собой запущенный образ Docker, использующий фреймворк в том типе тестирования Taurus (<https://oreil.ly/OGDne>) для запуска утилиты негрузочного тестирования Molotov (<https://oreil.ly/OGDne>). Мы уже рассматривали Molotov — простой и очень удобной утилите негрузочного тестирования на основе Python — в главе 5.

Начнем с создания Dockerfile для запуска Taurus и Molotov в каталоге `loadtest`:

```
$ mkdir loadtest; cd loadtest
$ cat Dockerfile
FROM blazemeter/taurus
COPY scripts /scripts
COPY taurus.yaml /bzt-configs/
```

```
WORKDIR /bzt-configs
ENTRYPOINT ["sh", "-c", "bzt -l /tmp/artifacts/bzt.log /bzt-configs/taurus.yaml"]
```

В этом Dockerfile выполняется команда строка вызов утилиты `bzt` на основе файла конфигурации `taurus.yaml`:

```
$ cat taurus.yaml
execution:
- executor: molotov
  concurrency: 10 # Количество процессов-исполнителей Molotov
  iterations: 5 # Ограничение на количество итераций для теста
  ramp-up: 30s
  hold-for: 5m
  scenario:
    script: /scripts/loadtest.py # Должен представлять собой
                                # корректный сценарий Molotov
```

В этом файле конфигурации значение параметра `concurrency` задано равным 10, то есть мы моделируем десять параллельно работающих конкурентно или виртуальных пользователей. `executor` задан как тест `molotov` на основе сценария `loadtest.py` в каталоге `scripts`. Вот этот сценарий, представляющий собой модуль Python:

```
$ cat scripts/loadtest.py
import os
import json
import random
import molotov
from molotov import global_setup, scenario

@global_setup()
def init_test(args):
    BASE_URL=os.getenv('BASE_URL', '')
    molotov.set_var('base_url', BASE_URL)

@scenario(weight=50)
async def _test_list_todos(session):
    base_url= molotov.get_var('base_url')
    async with session.get(base_url + '/todos') as resp:
        assert resp.status == 200, resp.status

@scenario(weight=30)
async def _test_create_todo(session):
    base_url= molotov.get_var('base_url')
    todo_data = json.dumps({'text':
        'Created new todo during Taurus/molotov load test'})
    async with session.post(base_url + '/todos',
```

```

        data=todo_data) as resp:
            assert resp.status == 200

@scenario(weight=10)
async def _test_update_todo(session):
    base_url= molotov.get_var('base_url')
    # Выводим список всех todo
    async with session.get(base_url + '/todos') as resp:
        res = await resp.json()
        assert resp.status == 200, resp.status
        # Выбираем случайный элемент todo и обновляем его
        # с помощью запроса PUT
        todo_id = random.choice(res)['id']
        todo_data = json.dumps({'text':
            'Updated existing todo during Taurus/molotov load test'})
        async with session.put(base_url + '/todos/' + todo_id,
            data=todo_data) as resp:
            assert resp.status == 200

@scenario(weight=10)
async def _test_delete_todo(session):
    base_url= molotov.get_var('base_url')
    # Выводим список всех todo
    async with session.get(base_url + '/todos') as resp:
        res = await resp.json()
        assert resp.status == 200, resp.status
        # Выбираем случайный элемент todo и обновляем его
        # с помощью запроса PUT
        todo_id = random.choice(res)['id']
        async with session.delete(base_url + '/todos/' + todo_id) as resp:
            assert resp.status == 200

```

Этот сценарий включает четыре функции, с помощью которых декорируются тесты сценария, для выполнения их с помощью Molotov. Они тестируют различные конечные точки API CRUD REST. Вес определяет долю общей длительности тестов, отводимую на каждый из сценариев. Например, длительность вызовов функции `_test_list_todos` составляет здесь примерно 50 % времени, `_test_create_todo` — около 30 % времени, `_test_update_todo` и `_test_delete_todo` будут выполняться примерно по 10 % времени каждый.

Собираем локальный образ Docker:

```
$ docker build -t cdk-loadtest .
```

Создаем локальный каталог артефактов:

```
$ mkdir artifacts
```

Затем запускаем контейнер Docker и монтируем локальный каталог `artifacts` внутри контейнера Docker в качестве каталога `/tmp/artifacts`:

```
$ docker run --rm -d \
--env BASE_URL=https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod \
-v `pwd`/artifacts:/tmp/artifacts cdk-loadtest
```

Производим отладку сценария `Molotov`, изучая файл `artifacts/molotov.out`.

Результаты работы `Taurus` можно просмотреть либо с помощью команды `docker logs CONTAINER_ID`, либо просто изучая содержимое файла `artifacts/bzt.log`.

Результаты изучения журналов Docker:

```
$ docker logs -f a228f8f9a2bc
19:26:26 INFO: Taurus CLI Tool v1.13.8
19:26:26 INFO: Starting with configs: ['/bzt-configs/taurus.yaml']
19:26:26 INFO: Configuring...
19:26:26 INFO: Artifacts dir: /tmp/artifacts
19:26:26 INFO: Preparing...
19:26:27 INFO: Starting...
19:26:27 INFO: Waiting for results...
19:26:32 INFO: Changed data analysis delay to 3s
19:26:32 INFO: Current: 0 vu 1 succ 0 fail 0.546 avg rt /
Cumulative: 0.546 avg rt, 0% failures
19:26:39 INFO: Current: 1 vu 1 succ 0 fail 1.357 avg rt /
Cumulative: 0.904 avg rt, 0% failures
ETC
19:41:00 WARNING: Please wait for graceful shutdown...
19:41:00 INFO: Shutting down...
19:41:00 INFO: Post-processing...
19:41:03 INFO: Test duration: 0:14:33
19:41:03 INFO: Samples count: 1857, 0.00% failures
19:41:03 INFO: Average times: total 6.465, latency 0.000, connect 0.000
19:41:03 INFO: Percentiles:
+-----+
| Percentile, % | Resp. Time, s |
+-----+
0.0	0.13
50.0	1.66
90.0	14.384
95.0	26.88
99.0	27.168
99.9	27.584
100.0	27.792
+-----+
```

Создайте инструментальные пакеты для длительности выполнения функции `Lambda` (рис. 13.1) и выделяемых и потребляемых единиц пропускной способности по чтению/записи `DynamoDB` (рис. 13.2).

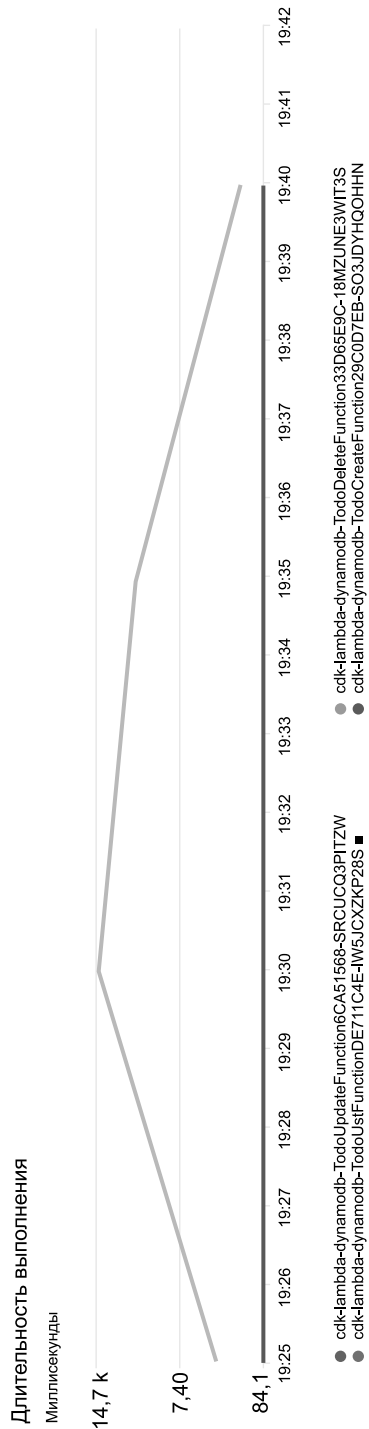


Рис. 13.1. Длительность выполнения функции Lambda

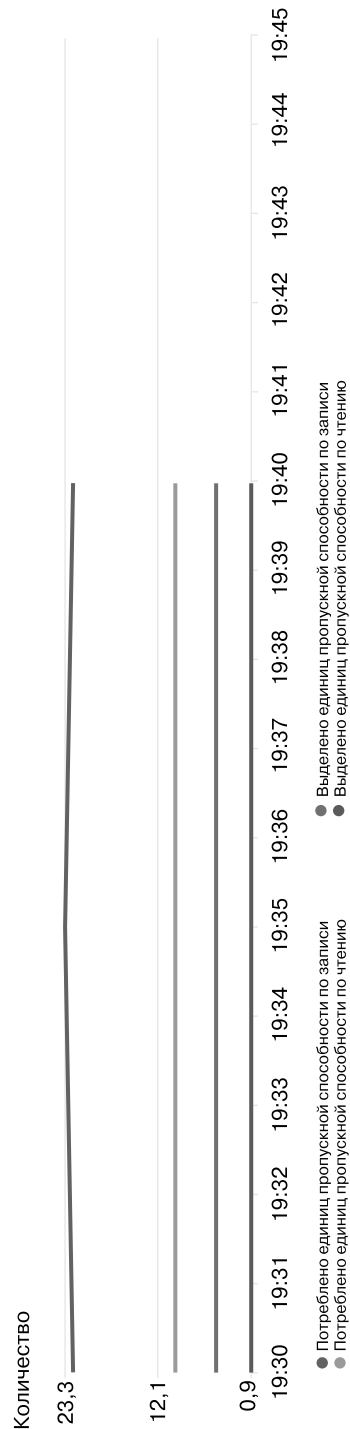


Рис. 13.2. Выделяемые и потребляемые единицы пропускной способности по чтению/записи DynamoDB

Метрики DynamoDB демонстрируют, что мы выделили слишком мало единиц пропускной способности по чтению DynamoDB. В результате возникет здержк, особенно для функции List (отображен график длительности выполнения функции Lambda красной линией, доходящей до 14,7 секунды), извлекающей все элементы `todo` из таблицы DynamoDB, что требует большого количества операций чтения. При создании таблицы DynamoDB мы запланировали выделение единиц пропускной способности по чтению равным 10, график CloudWatch демонстрирует, что он доходит до 25.

Изменим тип таблицы DynamoDB с `PROVISIONED` на `PAY_PER_REQUEST`. Внесите соответствующее изменение в файл `cdk_lambda_dynamodb_fargate/cdk_lambda_dynamodb_stack.py`:

```
table = aws_dynamodb.Table(self, "Table",
    partition_key=aws_dynamodb.Attribute(
        name="id",
        type=aws_dynamodb.AttributeType.STRING),
    billing_mode = aws_dynamodb.BillingMode.PAY_PER_REQUEST)
```

Выполните команду `cdk deploy`, после чего запустите локальный контейнер Docker для нагрузочного тестирования.

Наш результат выглядит так:

| Percentile, % | Resp. Time, s |
|---------------|---------------|
| 0.0 | 0.136 |
| 50.0 | 0.505 |
| 90.0 | 1.296 |
| 95.0 | 1.444 |
| 99.0 | 1.806 |
| 99.9 | 2.226 |
| 100.0 | 2.86 |

Лучше выглядят и графики длительности выполнения функции Lambda (рис. 13.3) и выделяемых и потребляемых единиц пропускной способности по чтению/записи DynamoDB (рис. 13.4).

Отметим, что потребляемые единицы пропускной способности по чтению автоматически выделяются DynamoDB по мере необходимости и выполняется вертикальное масштабирование, чтобы выдерживать возросшее количество запросов и чтение от функций Lambda. Наибольший вклад в смысле запросов и чтение вносит функция List, вызываемая при выводе списка, обновлении и удалении в сценарии `loadtest.py` Molotov с помощью `session.get(base_url + /todos)`.

Длительность выполнения

Различные единицы

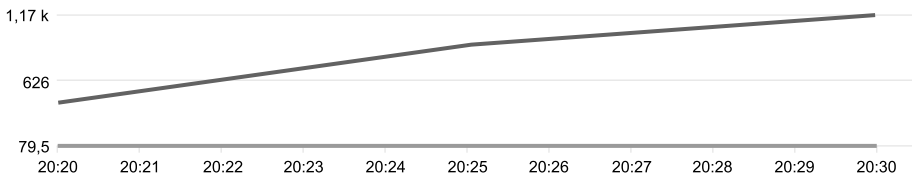


Рис. 13.3. Длительность выполнения функции Lambda

Количество

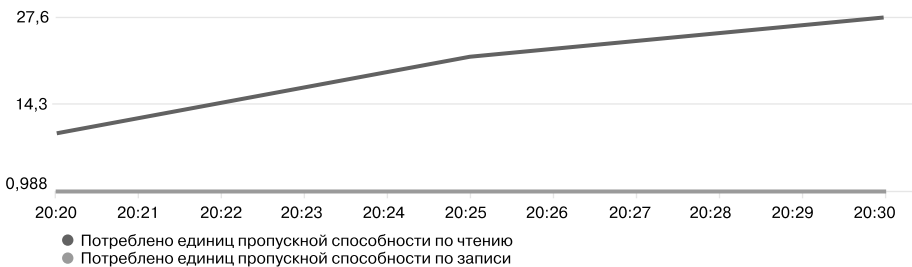


Рис. 13.4. Выделяемые и потребляемые единицы пропускной способности по чтению/записи DynamoDB

Далее мы создадим стек CDK Fargate для запуска контейнеров, основанных на сформированном ранее образе Docker:

```
$ cat cdn_lambda_dynamodb_fargate/cdn_fargate_stack.py
from aws_cdk import core
from aws_cdk import aws_ecs, aws_ec2

class FargateStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        vpc = aws_ec2.Vpc(
            self, "MyVpc",
            cidr= "10.0.0.0/16",
            max_azs=3
        )

        # Описываем кластер ECS, хостируемый в запрошенном нами VPC
        cluster = aws_ecs.Cluster(self, 'cluster', vpc=vpc)

        # Описываем нашу задачу в рамках одного контейнера
        # образ собирается и публикуется из локального каталога ресурсов
```

```

task_definition = aws_ecs.FargateTaskDefinition(self,
    'LoadTestTask')
task_definition.add_container('TaurusLoadTest',
    image=aws_ecs.ContainerImage.from_asset("loadtest"),
    environment={'BASE_URL':
        "https://k6ygy4xw24.execute-api.us-east-2.amazonaws.com/prod/"})

# Описываем сервис fargate. TPS определяет количество требуемых
# экземпляров задания (каждая задача соответствует одной TPS)
aws_ecs.FargateService(self, 'service',
    cluster=cluster,
    task_definition=task_definition,
    desired_count=1)

```

В коде кл `ss FargateStack` необходимо отметить несколько нюансов.

- С помощью конструкции `aws_ec2.Vpc` создается новое VPC.
- В этом новом VPC создается кластер ECS.
- Описание задачи Fargate создается на основе Dockerfile из каталога `loadtest`, CDK доступен интеллектуально для того, чтобы собрать образ Docker на основе этого Dockerfile и затем поместить его в реестр Docker ECR.
- Для запуска контейнеров Fargate, основанных на помещенном в реестр ECR образе, создается сервис ECS, параметр `desired_count` определяет количество запускаемых контейнеров.

Вызовем в `app.py` конструктор класса `FargateStack`:

```

$ cat app.py
#!/usr/bin/env python3

from aws_cdk import core

from cdk_lambda_dynamodb_fargate.cdk_lambda_dynamodb_stack \
import CdkLambdaDynamodbStack
from cdk_lambda_dynamodb_fargate.cdk_fargate_stack import FargateStack

app = core.App()
app_env = {
    "region": "us-east-2",
}

CdkLambdaDynamodbStack(app, "cdk-lambda-dynamodb", env=app_env)
FargateStack(app, "cdk-fargate", env=app_env)

app.synth()

```

Прозвертываем стек `cdk-fargate`:

```
$ cdk deploy cdk-fargate
```

Перейдите в консоль AWS и взгляните на кластер ECS с запущенным там контейнером Fargate (рис. 13.5).

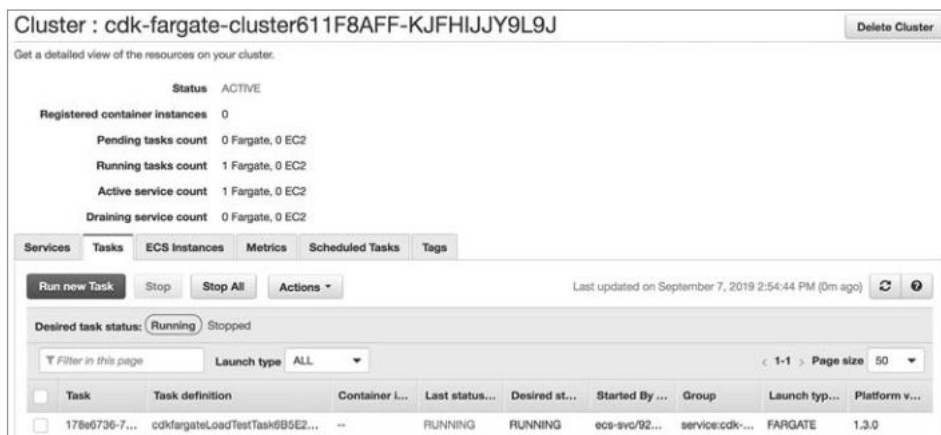


Рис. 13.5. Кластер ECS с запущенным там контейнером Fargate

Посмотрите на инструментальной панели CloudWatch длительность выполнения функции Lambda (рис. 13.6), также выделяемых и потребляемых единиц пропускной способности по чтению/записи DynamoDB (рис. 13.7). Как видите, затраты вполне удовлетворительны.

Увеличиваем количество контейнеров Fargate в `cdk_lambda_dynamodb_fargate/cdk_fargate_stack.py` до 5:

```
aws_ecs.FargateService(self, 'service',
    cluster=cluster,
    task_definition=task_definition,
    desired_count=5)
```

Снова прозвертываем стек `cdk-fargate`:

```
$ cdk deploy cdk-fargate
```

Смотрим на инструментальной панели CloudWatch длительность выполнения функции Lambda (рис. 13.8), также выделяемых и потребляемых единиц пропускной способности по чтению/записи DynamoDB (рис. 13.9).

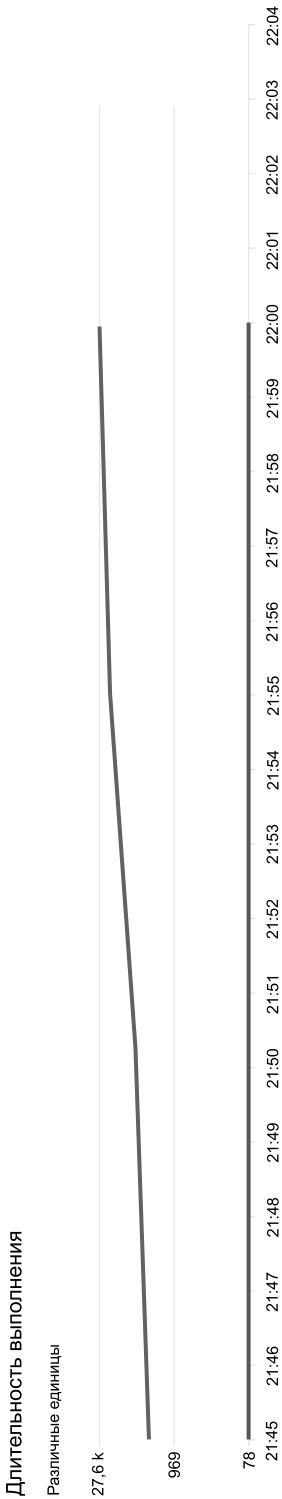


Рис. 13.6. Длительность выполнения функции Lambda

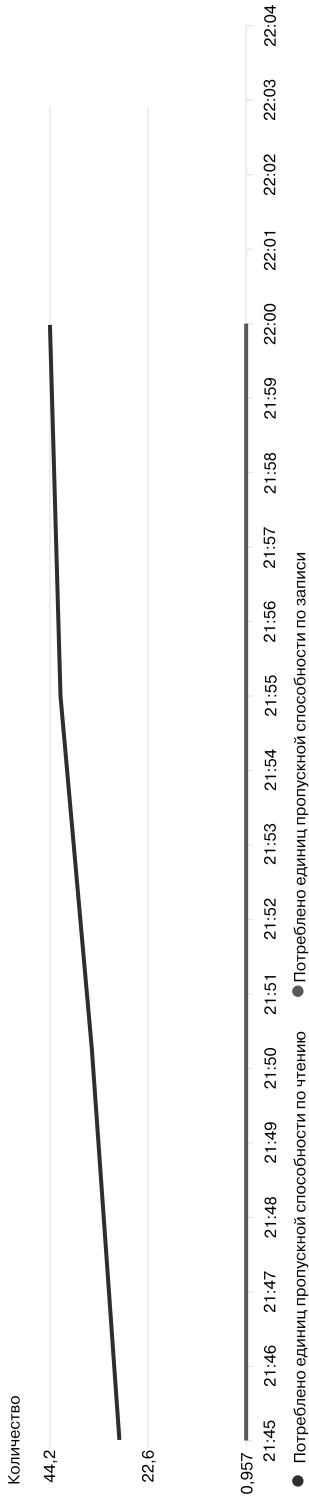
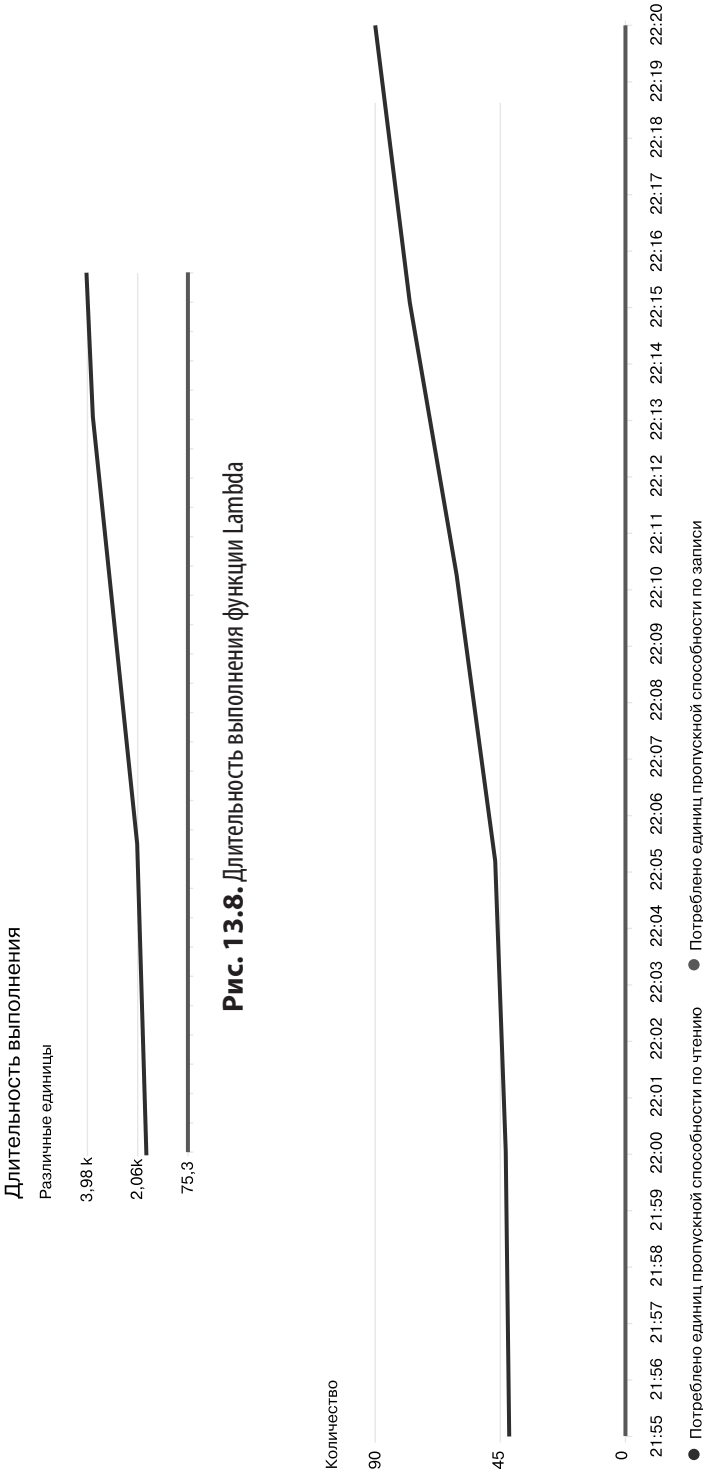


Рис. 13.7. Выделяемые и потребляемые единицы пропускной способности по чтению/записи DynamoDB



Как и ожидалось, значения обеих метрик — длительности выполнения функции Lambda и количества единиц пропускной способности по чтению DynamoDB — выросли, поскольку мы теперь моделируем $5 \cdot 10 = 50$ работающих конкурентно пользователей.

Для моделирования большего числа пользователей можно увеличить значение параметра `concurrency` в файле конфигурации `taurus.yaml`, так и значение параметра `desired_count` для контейнера Fargate. Благодаря сочетанию этих двух параметров можно легко увеличить и нагрузку конечных точек нашего API REST.

Удаляем стеки CDK:

```
$ cdk destroy cdk-fargate
$ cdk destroy cdk-lambda-dynamodb
```

Стоит отметить, что рассмотренная нами бессерверная архитектура (API Gateway + 5 функций AWS Lambda + таблица DynamoDB) неплохо подошла для приложения API CRUD REST. Мы также следовали всем рекомендуемым практикам и описали всю инфраструктуру в коде Python с помощью AWS CDK.

Упражнения

- Зпустите простую конечную точку HTTP с помощью платформы SaaS Google: Cloud Run (<https://cloud.google.com/run>).
- Зпустите простые конечные точки HTTP и других упоминавшихся платформ, основанных на Kubernetes: Kubeless (<https://kubeless.io>), Fn Project (<https://fnproject.io>) и Fission (<https://fission.io>).
- Установите и настройте Apache OpenWhisk (<https://openwhisk.apache.org>) в кластере Kubernetes, подходящем для промышленной эксплуатации, например Amazon EKS, Google GKE или Azure AKS.
- Портинруйте наш пример API REST для AWS и GCP и Azure. Для работы с несколькими API GCP предоставляет Cloud Endpoints (<https://cloud.google.com/endpoints>), Azure — API Management (<https://oreil.ly/tmDh7>).

MLO и разработка ПО для машинного обучения

Одн из с мых престижных профессий в 2020 году — специ лист по м шинному обучению. В числе прочих престижных профессий — инженер по р боте с д нными, исследов тель д нных и исследов тель в сфере м шинного обучения. Подобные профессии не меш ют в м ост в ться специ листом по DevOps: DevOps — стиль р боты, и принципы DevOps применимы к любым прогр ммным проект м, включ я м шинное обучение. Р ссмотрим основные рекомендуемые пр ктики DevOps — непрерывную интегр цию, непрерывную пост вку, микросервисы, инфр структуру к к код, мониторинг и журн лиров ние, связь и сотрудничество. Р зве к кие-то из них не подходят для м шинного обучения?

Чем сложнее проект в сфере р зр ботки ПО — м шинное обучение отлич - ется зн чительной сложностью, — тем в жнее соблюдение принципов DevOps. Существует ли лучший пример микросервисов, чем API для предск з ния н основе м шинного обучения? В этой гл ве мы обсудим, к к ре лизовыв ть м шинное обучение профессиона льно, повторяемым обр зом, н основе миро-воззрения DevOps.

Что такое машинное обучение

М шинное обучение — методик применения лгоритмов для втом тического усвоения информ ции из д нных. Существуют четыре основные р зновидно-сти м шинного обучения: м шинное обучение с учителем, ч стичное обучение с учителем, м шинное обучение без учителя и обучение с подкреплением.

Машинное обучение с учителем

При машинном обучении с учителем правильные ответы заранее известны и маркированы. Например, для задачи предсказания роста по весу можно заранее собрать примеры данных роста и весов различных людей. Рост будет при этом целевой переменной, вес — признаком.

Пройдемся по этому примеру машинного обучения с учителем. Он включает:

- исходный набор данных (<https://oreil.ly/jzWmI>);
- 25 000 искусственно созданных записей роста и веса 18-летних молодых людей.

Ввод данных

In[0]:

```
import pandas as pd
```

In[7]:

```
df = pd.read_csv(
    "https://raw.githubusercontent.com/noahgift/\
    regression-concepts/master/\
    height-weight-25k.csv")
```

```
df.head()
```

Out[7]:

| Index | Height-Inches | Weight-Pounds |
|-------|---------------|---------------|
| 0 1 | 65.78331 | 112.9925 |
| 1 2 | 71.51521 | 136.4873 |
| 2 3 | 69.39874 | 153.0269 |
| 3 4 | 68.21660 | 142.3354 |
| 4 5 | 67.78781 | 144.2971 |

Разведочный анализ данных

Взглянем на эти данные и узнаем, что из них можно извлечь.

Диаграмма рассеяния. В этом примере для визуализации набора данных мы воспользуемся популярной библиотекой `seaborn` для Python. При необходимости ее можно установить в блокноте с помощью команды `!pip install seaborn`. Все остальные библиотеки, упоминаемые в этом разделе, также можно установить с помощью команды `!pip install <название пакета>`. Если вы используете блокнот Colab, эти библиотеки уже установлены. См. график соотношения «рост/вес», построенный с помощью `lplot`, на рис. 14.1.


```
In[0]:
```

```
import seaborn as sns
import numpy as np
```

```
In[9]:
```

```
sns.lmplot("Height-Inches", "Weight-Pounds", data=df)
```

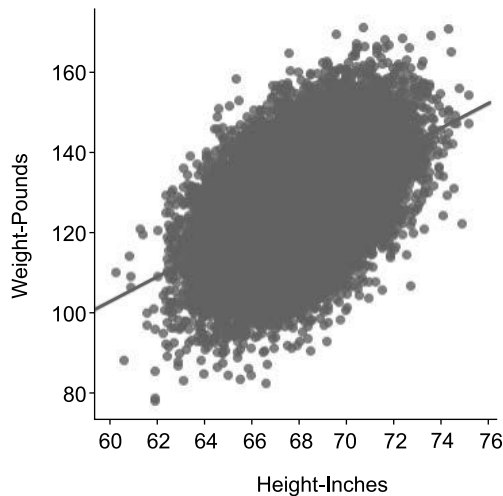


Рис. 14.1. График lmplot роста/веса

Общие статистические показатели

Далее можно сгенерировать некоторые статистические показатели.

```
In[10]:
```

```
df.describe()
```

```
Out[10]:
```

| | Index | Height-Inches | Weight-Pounds |
|-------|--------------|---------------|---------------|
| count | 25000.000000 | 25000.000000 | 25000.000000 |
| mean | 12500.500000 | 67.993114 | 127.079421 |
| std | 7217.022701 | 1.901679 | 11.660898 |
| min | 1.000000 | 60.278360 | 78.014760 |
| 25% | 6250.750000 | 66.704397 | 119.308675 |
| 50% | 12500.500000 | 67.995700 | 127.157750 |
| 75% | 18750.250000 | 69.272958 | 134.892850 |
| max | 25000.000000 | 75.152800 | 170.924000 |

Ядерное распределение плотности

Р-спределение из гр-фик-плотности (рис. 14.2) демонстрирует зависимость между двумя переменными друг от друга.

In[11]:

```
sns.jointplot("Height-Inches", "Weight-Pounds", data=df, kind="kde")
```

Out[11]:

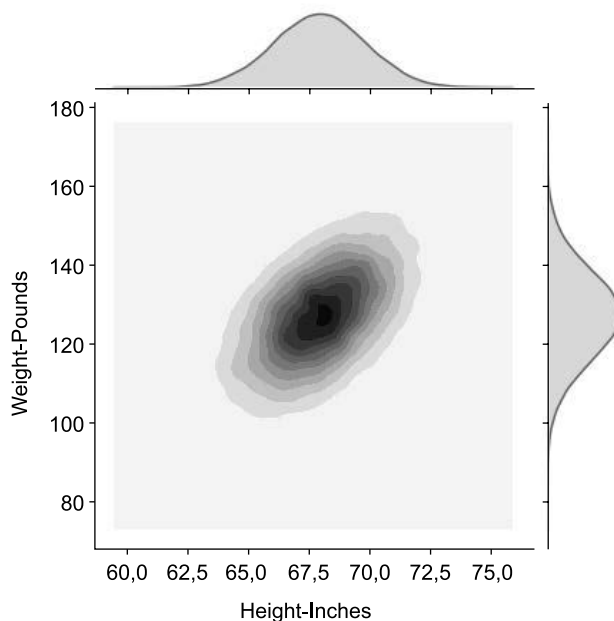


Рис. 14.2. График плотности

Моделирование

Теперь займемся моделированием. При моделировании на основе машинного обучения алгоритмы ищут закономерности из данных. Общий смысл — предсказать будущие данные на основе предыдущих.

Модель регрессии sklearn

Сначала из данных выделяются признаки и целевые переменные, затем данные разбиваются на обучающий и контрольный наборы данных. Благодаря этому

контрольный набор данных отделяется для последующей проверки безошибочности обученной модели.

In[0]:

```
from sklearn.model_selection import train_test_split
```

Выделение и изучение признаков и целевой переменной. Полезно явно извлечь целевую переменную и переменные признаки и привести их к единой форме. После этого желательно проверить форму и убедиться, что она подходит для машинного обучения с помощью sklearn.

In[0]:

```
y = df['Weight-Pounds'].values #Цель
y = y.reshape(-1, 1)
X = df['Height-Inches'].values #Признак(и)
X = X.reshape(-1, 1)
```

In[14]:

```
y.shape
```

Out[14]:
(25000, 1)

Разбиение данных. Данные разбиваются в соотношении 80/20 %.

In[15]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

Out[15]:

```
(20000, 1) (20000, 1)
(5000, 1) (5000, 1)
```

Подгонка модели. Производим подгонку модели с помощью алгоритм линейной регрессии, импортируемого из sklearn.

In[0]:

```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()
model = lm.fit(X_train, y_train)
y_predicted = lm.predict(X_test)
```

Выводим показатели безошибочности модели линейной регрессии. Посмотрим теперь, какую безошибочность демонстрирует обученная модель при

предсказаний новых данных. Для этого вычисляем RMSE (root mean squared error — среднеквадратическая ошибка) предсказаний и контрольных данных.

In[18]:

```
from sklearn.metrics import mean_squared_error
from math import sqrt

# RMSE — среднеквадратическая ошибка
rms = sqrt(mean_squared_error(y_predicted, y_test))
rms
```

Out[18]:

10.282608230082417

График соотношения предсказанного и истинного роста. Теперь построим график соотношения предсказанного и истинного роста (рис. 14.3), чтобы выяснить, насколько хороши предсказания модели.

In[19]:

```
import matplotlib.pyplot as plt
_, ax = plt.subplots()

ax.scatter(x = range(0, y_test.size), y=y_test, c = 'blue', label = 'Actual',
           alpha = 0.5)
ax.scatter(x = range(0, y_predicted.size), y=y_predicted, c = 'red',
           label = 'Predicted', alpha = 0.5)

plt.title('Actual Height vs Predicted Height')
plt.xlabel('Weight')
plt.ylabel('Height')
plt.legend()
plt.show()
```

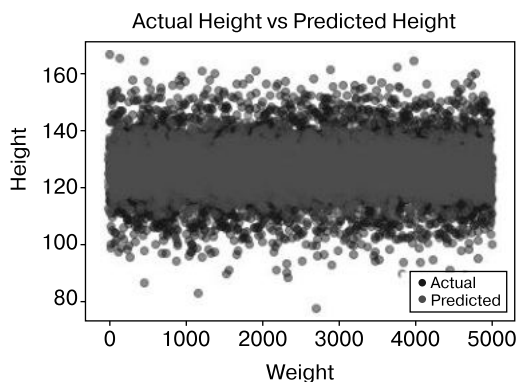


Рис. 14.3. Соотношение предсказанного и истинного роста

Это очень простой, но впечатляющий пример вполне реалистичного технологического процесса создания модели машинного обучения.

Экосистема машинного обучения языка Python

Пройдемся кратко по экосистеме машинного обучения языка Python (рис. 14.4).

Фактически она включает четыре основные сферы: глубокое обучение, sklearn, AutoML¹ и Spark. В сфере глубокого обучения наиболее популярны (в порядке убывания) фреймворки TensorFlow/Keras, PyTorch и MXNet. Компания Google поддерживает финансово TensorFlow, Facebook — PyTorch, MXNet — партнер Amazon. Вы увидите, что MXNet часто упоминается в Amazon SageMaker. Важно отметить, что эти фреймворки глубокого обучения ориентированы на использование GPU, благодаря чему работают быстрее ориентированных на CPU примерно в 50 раз.

Экосистема sklearn зачастую включает в один проект Pandas и Numpy. Sklearn также не использует GPU. Впрочем, существует проект Numba, который специально ориентирован на использование GPU (производства NVIDIA, так и AMD).

Два лидера в сфере AutoML — компания Uber с ее набором инструментов Ludwig и компания H2O с ее платформой H2O. Обе они позволяют сэкономить немало времени при разработке новых моделей машинного обучения, также оптимизируют уже существующие модели.

Наконец, есть еще экосистема Spark, основанная благодаря Hadoop. Spark может использоваться как GPU, так и CPU, что и делает ее множеством различных платформ Amazon EMR, Databricks, GCP Dataproc и др.

Глубокое обучение с помощью PyTorch

Теперь, описав экосистему машинного обучения на языке Python, мы можем обсудить портирование простого примера линейной регрессии на PyTorch и выполнение его на GPU с поддержкой CUDA. Проще всего получить доступ к GPU NVIDIA, применяя блокноты Colab. Они представляют собой хостируемые в Google совместимые с Jupyter блокноты, предоставляющие пользователю доступ к физическим (GPU), так и к тензорным процессорам (TPU). Приводимый далее код можно запустить на GPU (<https://oreil.ly/kQhKO>).

¹ Automated machine learning — автоматическое машинное обучение. — *Примеч. пер.*

Разработка систем машинного обучения: полный цикл

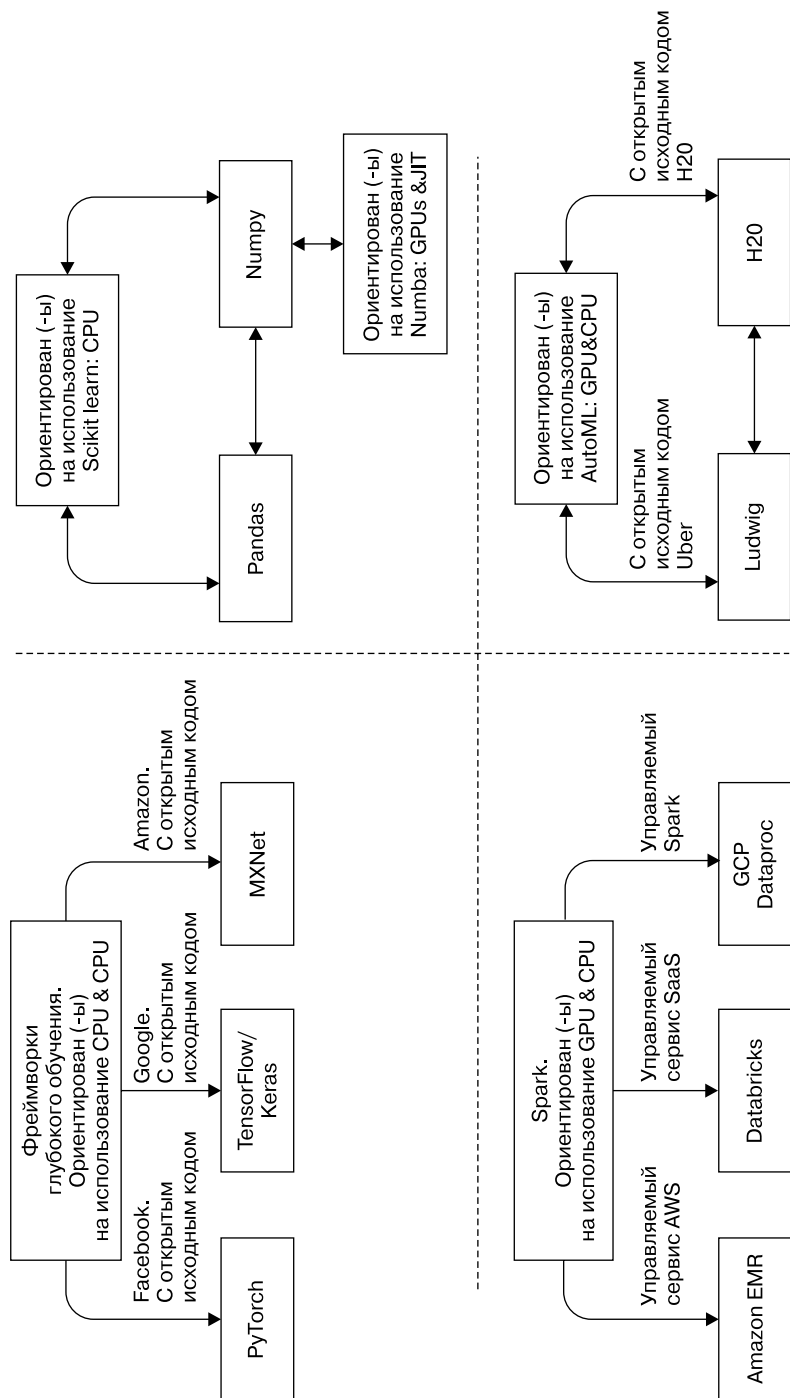


Рис. 14.4. Экосистема машинного обучения языка Python

Регрессия с помощью PyTorch

Сначала преобразуем данные в тип `float32`:

```
In[0]:

# Обучающие данные
x_train = np.array(X_train, dtype=np.float32)
x_train = x_train.reshape(-1, 1)
y_train = np.array(y_train, dtype=np.float32)
y_train = y_train.reshape(-1, 1)

# Контрольные данные
x_test = np.array(X_test, dtype=np.float32)
x_test = x_test.reshape(-1, 1)
y_test = np.array(y_test, dtype=np.float32)
y_test = y_test.reshape(-1, 1)
```

Учтите, что, если вы не используете блокноты Colab, возможно, вам придется установить PyTorch. Кроме того, если вы используете блокноты Colab, то можете бесплатно получить доступ к GPU NVIDIA и выполнить этот код на нем. В противном случае вам нужно будет выполнить этот код на платформе с GPU.

```
In[0]:

import torch
from torch.autograd import Variable

class linearRegression(torch.nn.Module):
    def __init__(self, inputSize, outputSize):
        super(linearRegression, self).__init__()
        self.linear = torch.nn.Linear(inputSize, outputSize)

    def forward(self, x):
        out = self.linear(x)
        return out
```

Теперь создайте модель с подключенным CUDA (если вы выполняете в блокноте Colab или на машине с GPU):

```
In[0]:

inputDim = 1          # для переменной 'x'
outputDim = 1         # для переменной 'y'
learningRate = 0.0001
epochs = 1000

model = linearRegression(inputDim, outputDim)
model.cuda()
```

```
Out[0]:
```

```
linearRegression(
    (linear): Linear(in_features=1, out_features=1, bias=True)
)
```

Создаем объекты для стохастического градиентного спуска и функции потерь:

```
In[0]:
```

```
criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learningRate)
```

А теперь обучаем модель:

```
In[0]:
```

```
for epoch in range(epochs):
    inputs = Variable(torch.from_numpy(x_train).cuda())
    labels = Variable(torch.from_numpy(y_train).cuda())
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    print(loss)
    # получаем градиенты относительно параметров
    loss.backward()
    # обновление параметров
    optimizer.step()
    print('epoch {}, loss {}'.format(epoch, loss.item()))
```

Вывод от 1000 итераций сокращен ради экономии места.

```
Out[0]:
```

```
tensor(29221.6543, device='cuda:0', grad_fn=<MseLossBackward>)
epoch 0, loss 29221.654296875
tensor(266.7252, device='cuda:0', grad_fn=<MseLossBackward>)
epoch 1, loss 266.72515869140625
tensor(106.6842, device='cuda:0', grad_fn=<MseLossBackward>)
epoch 2, loss 106.6842269897461
....сокращено....
epoch 998, loss 105.7930908203125
tensor(105.7931, device='cuda:0', grad_fn=<MseLossBackward>)
epoch 999, loss 105.7930908203125
```

График соотношения предсказанного и истинного роста. А теперь построим график соотношения предсказанного и истинного роста (рис. 14.5), как в предыдущей простой модели.

```
In[0]:
```

```
with torch.no_grad():
    predicted = model(Variable(torch.from_numpy(x_test).cuda())).cpu().\
        data.numpy()
    print(predicted)
```



```
plt.clf()
plt.plot(x_test, y_test, 'go', label='Actual Height', alpha=0.5)
plt.plot(x_test, predicted, '--', label='Predicted Height', alpha=0.5)
plt.legend(loc='best')
plt.show()
```

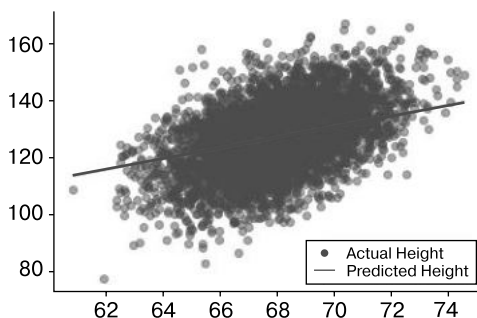


Рис. 14.5. Предсказанный и истинный рост

Выводим RMSE. В конце, выводим RMSE и сравним его:

In[0]:

```
#RMSE — средняя квадратическая ошибка
rms = sqrt(mean_squared_error(x_test, predicted))
rms
```

Out[0]:

```
59.19054613663507
```

Для глубокого обучения потребовалось немного больше кода, но суть та же — мы я, что и в модели sklearn. Одно из преимуществ — в оптимальном включении GPU в конвейеры при промышленной эксплуатации. Даже если вы с нами не знакомы с глубоким обучением, зная основы процесса создания использующих GPU моделей машинного обучения не помешает.

Платформы облачного машинного обучения

Один из повсеместно встречающихся спектров машинного обучения — облачные платформы машинного обучения. Компания Google предлагает платформу ИИ GCP (рис. 14.6).

Платформа GCP включает множество высокоуровневых компонентов, от подготовки данных до их разметки. Платформа AWS предоставляет пользователям Amazon SageMaker (рис. 14.7).

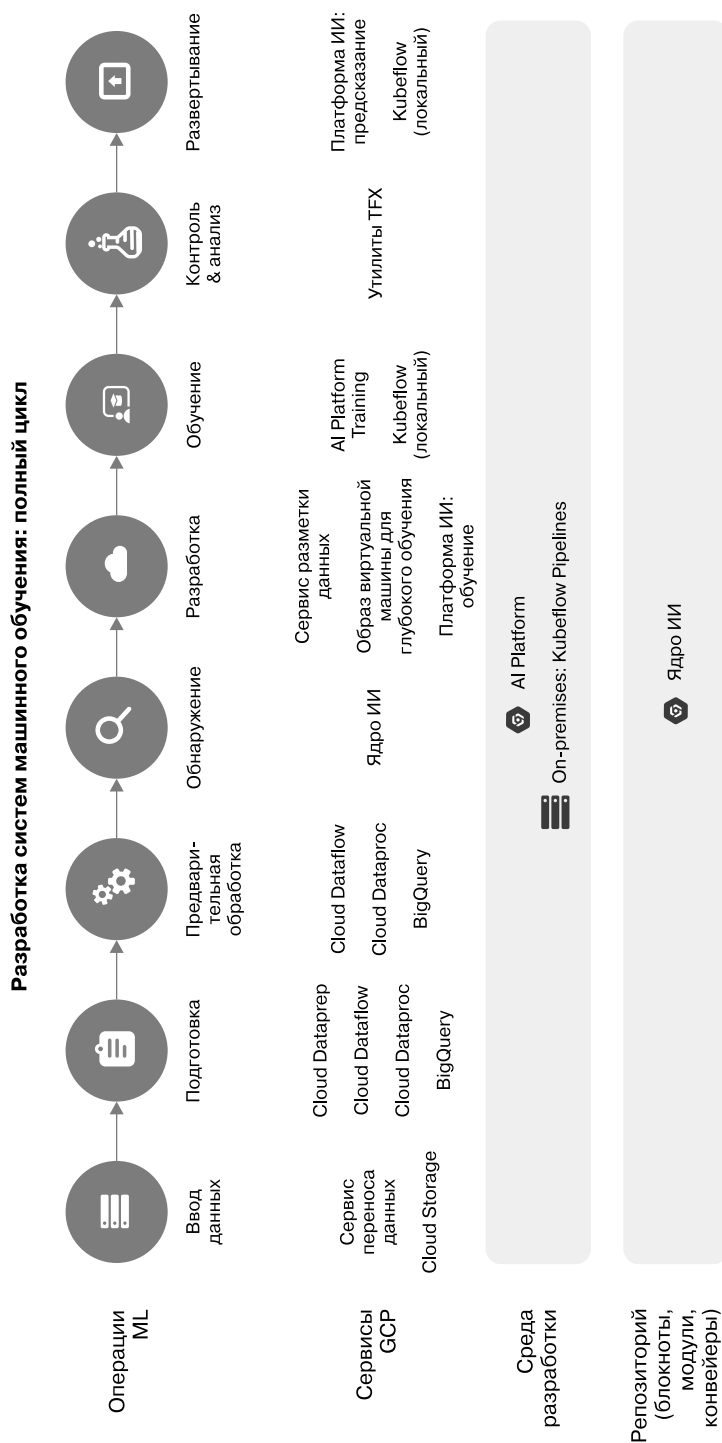


Рис. 14.6. Платформа ИИ GCP

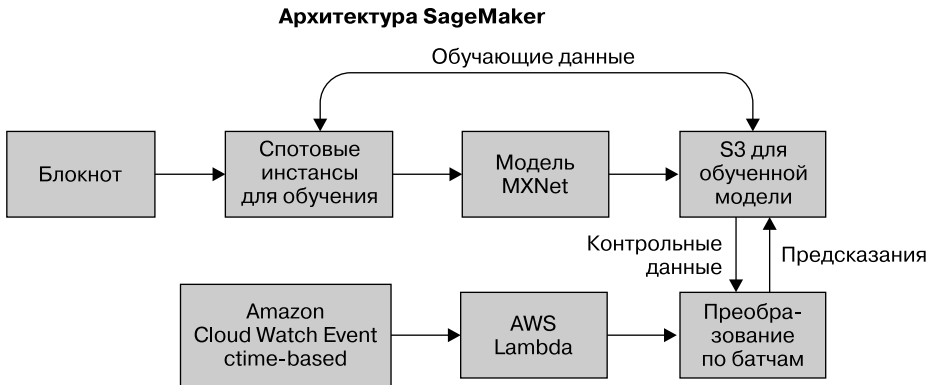


Рис. 14.7. Amazon SageMaker

Платформа Amazon SageMaker также включает множество высокоуровневых компонентов, в том числе обучение на спотовых инстансах и дистрибутивные конечные точки для предсказаний.

Модель зрелости машинного обучения

Одним из самых серьезных препятствий в настоящее время — проблемой осознания необходимости структурных изменений компаниями, желающими заниматься машинным обучением. Двигателем модели зрелости машинного обучения (рис. 14.8) выступают возникающие в связи с этим непростые задачи и возможности.

| |
|---|
| Уровень 1
Очерчивание рамок задачи и области определения, а также формулировка задачи |
| Уровень 2
Непрерывная поставка данных |
| Уровень 3
Непрерывная поставка очищенных данных |
| Уровень 4
Непрерывная поставка разведочного анализа данных |
| Уровень 5
Непрерывная поставка обычного ML и AutoML |
| Уровень 6
Цикл обратной связи эксплуатации ML |

Рис. 14.8. Модель зрелости машинного обучения

Основная терминология машинного обучения

Согласуем основную терминологию машинного обучения, что пригодится нам в остальной части главы.

- *Машинное обучение* — способ построения математических моделей на основе примеров данных или обучающих данных.
- *Модель* — конечный результат применения машинного обучения. Простейший пример — линейное уравнение, то есть прямая линия, предсказывающая связь X и Y .
- *Признак* — столбец в электронной таблице, играющий роль сигнала при создании модели машинного обучения. Хороший пример — очки, набранные в игре командой НБА.
- *Целевая величина (переменная)* — столбец в электронной таблице, который мы хотим предсказать. Хороший пример — количество игр, выигранных за сезон командой НБА.
- *Машинное обучение с учителем* — разновидность машинного обучения, связанная с предсказанием будущих значений на основе известной истории значений. Хороший пример — предсказание количества побед команды НБА за сезон по признаку количества очков за игру.
- *Машинное обучение без учителя* — разновидность машинного обучения, при которой робот учится с немониторинговыми данными. Вместо предсказания будущих значений с помощью инструментов и подобие кластеризации производится поиск скрытых закономерностей, которые далее можно использовать в качестве меток. Хороший пример — создание кластеров игроков НБА со схожим количеством очков, подборов, блок-шотов и передач. Один из кластеров мог бы называться, например, «Высокие рейтинговые игроки», другой — «Рыгивающие защитники, набирающие много очков».
- *Глубокое обучение* — разновидность машинного обучения, в которой используются искусственные нейронные сети для машинного обучения как с учителем, так и без учителя. Наиболее популярный фреймворк глубокого обучения — TensorFlow от компании Google.
- *Scikit-learn* — один из наиболее популярных фреймворков машинного обучения для языка Python.
- *Pandas* — один из наиболее популярных библиотек для речевых операций данными и их анализ. Хорошо работает в связке с scikit-learn и Numpy.
- *Numpy* — преобладающая на рынке библиотек для низкоуровневых научных расчетов. Поддерживает большие многомерные массивы и включает обширную коллекцию математических функций. Широко применяется с scikit-learn, Pandas и TensorFlow.

Уровень 1. Очерчивание рамок задачи и области определения, а также формулировка задачи

Рассмотрим первый уровень. При реализации машинного обучения в компании важно очертить круг решаемых проблем и сузить их области определения. Проекты машинного обучения чаще всего терпят неудачу именно потому, что компания не задается предвзвешенно вопросом о круге решаемых задач.

Хорошей иллюстрацией для этого может послужить создание мобильного приложения для сети ресторанов в Сан-Франциско. Неправильный подход: сразу же приступить к разработке нативных приложений для iOS и Android (двумя командами разработчиков). Стандартная команда состоит из трех разработчиков и полную ставку разработчиков для каждого мобильного приложения. Что означает необходимость нанять шесть разработчиков с заработной платой примерно 200 тыс. долларов в год. Расчетная стоимость проекта уже получается 1,2 млн долларов. Принесет ли это мобильное приложение прибыль больше 1,2 млн долларов в год? Если нет, то существует ли более простой вариант? Возможно, для разных целей лучше подойдет оптимизированное под мобильные устройства веб-приложение, которое могут создать уже разработчики в компании веб-разработчики.

А как насчет сотрудничества с компанией, специализирующейся на доставке еды, и делегирования ей этой задачи? Какие достоинства и недостатки у этого подхода? Анализ логичен: логика мышления может и должна применяться к начинаниям в сфере машинного обучения и науки о данных. Например, нужно ли компании нанять шестерых исследователей машинного обучения уровня PhD с окладом, скажем, 500 тыс. долларов в год? Какие есть альтернативные варианты? Очерчивание рамок задачи и области определения очень важно в машинном обучении и обеспечивают лучшие шансы на успех.

Уровень 2. Непрерывная поставка данных

Одним из основ современной цивилизации — водопроводы. Римские инженеры еще в 312 году до н. э. переносили воду на многие мили для обеспечения ею переполненных городов. Водопровод — неотъемлемая часть инфраструктуры, необходимой для функционирования крупного города. UNICEF оценили, что в 2018 год в мировых масштабах женщины и девушки потратили примерно 200 млн часов на хождение за водой. Упущенная выгода колоссальна: меньше времени на учебу, заботу о детях, отдых.

Рассхожее выражение гласит: «ПО превит бля». Следствие его: любой компании, занимающейся разработкой ПО, в будущем это все компании без исключения, необходим стратегический машинный обучение и ИИ. Цель этой стратегии — более серьезное отношение к непрерывной поставке данных. Как и водопровод,

непрерывно передается данными ежедневно экономит деньги. Одно из возможных решений этой задачи заключается в понятии *озеро данных* (data lake), как показано на рис. 14.9.

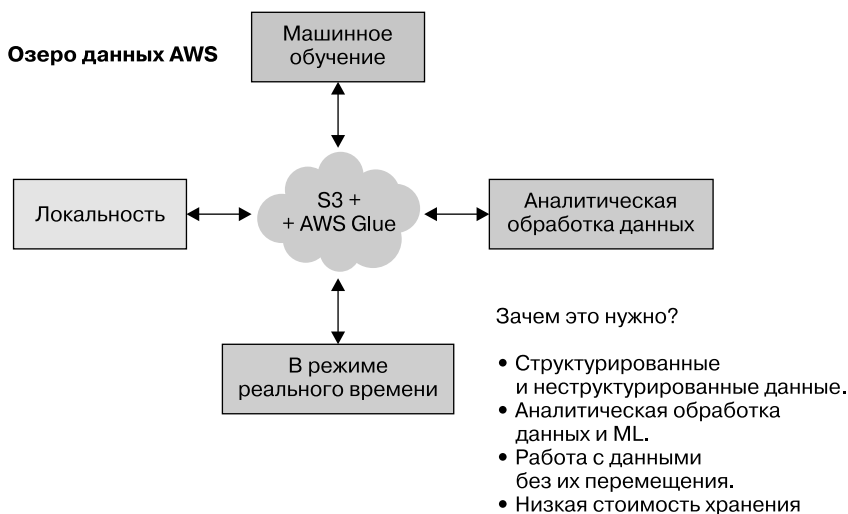


Рис. 14.9. Озеро данных AWS

На первый взгляд озеро данных способно покончиться решением, искусственно подогнанным под задачу, может, и слишком простым для того, чтобы принести реальную пользу. Однако взглянем на некоторые из решаемых им задач.

- Данные можно обрабатывать, не перемещая их.
- Хранение данных обходится очень дешево.
- Создание стратегий жизненного цикла архивирования данных не представляет трудностей.
- Создание стратегий жизненного цикла для обеспечения безопасности и аудит данных также нетрудно.
- Роботы в промышленной эксплуатации системы не сцеплены с обработкой данных.
- Объемы хранения и дисковых операций ввод/вывод могут быть практически безграничными.

Альтернативой подобной архитектуре — импровизированный мешинг, эквивалентный четырехчасовой прогулке к источнику и обратно обычной водой.

В архитектуре озер данных важнейшую роль играет безопасность, подобно безопасности в сфере водоснабжения. Благодаря централизации архитектуры хранения и обработки данных становится проще отслеживать и предотвращать утечки данных. Вот несколько идей, которые могут помочь в этом в будущем предотвратить утечки данных.

- Шифрованы ли данные при хранении? Если да, то у кого есть ключи? Носятся ли события их рэншифровки в журналы, в том числе в журналы аудита?
- Отрывается ли перемещение данных в журналы, в том числе в журналы аудита? Например, вряд ли из чего-то может получится вынести всю базу покупок телейзпределы вшей сети. Если подобное не отрывается в журналах, то почему?
- Производится ли периодический аудит безопасности? Если нет, то почему?
- Храните ли вы идентифицирующие личные данные? Почему?
- Вы производите мониторинг важнейших событий промышленной эксплуатации. Аккнсчет мониторинг событий безопасности? Почему нет?

Зачем вообще разрешать перемещение данных за пределы внутренней сети? Лучше бы сделать критически важные данные чем-то вроде пресловутой квадратной дырки, которую невозможно вытиснить за пределы основной сети без специального «кода доступа». Невозможность перемещения данных за пределы локальной среды предствляется разумным способом предотвращения утечек. Что, если внешняя сеть сама может перемещать только «круглые» пакеты данных? Это был бы замечательная возможность для облаков, которые предоставляют подобные защищенные озера данных.

Уровень 3. Непрерывная поставка очищенных данных

Надеясь, мы убедили вас в необходимости непрерывной обработки данных и ее важности для успешной реализации комплексной платформы машинного обучения. Одно из важнейших усовершенствований непрерывной обработки данных — непрерывная поставка очищенных данных. Зачем возиться с обработкой данных, которые не ходят в полном беспорядке? В связи с этим вспоминается недавняя проблема с загрязнением воды в городе Флинт, штат Мичиган. В 2014 году Флинт стал брать воду не из озера Гурон и реки Детройт, а из реки Флинт. Чиновники не побоялись о довлении в воду ингибиторов коррозии, в результате чего свинец из старых труб начал попадать в водопроводную воду. Возможно также, что именно смена источника воды привела к вспышке легионеллеза, в результате которой умерли 12 человек и заболели еще 87.

Одн из с мых первых историй успешного применения н уки о д нных связ н с з грязнением воды в 1849–1854 год х. Джон Сноу с помощью визу лиз ции д нных сумел выявить кл стеры случ ев холеры (рис. 14.10) и обн ружить первопричину эпидемии. Сточные воды слив ли прямо в водопровод, сн бж в- ший н селение питьевой водой!

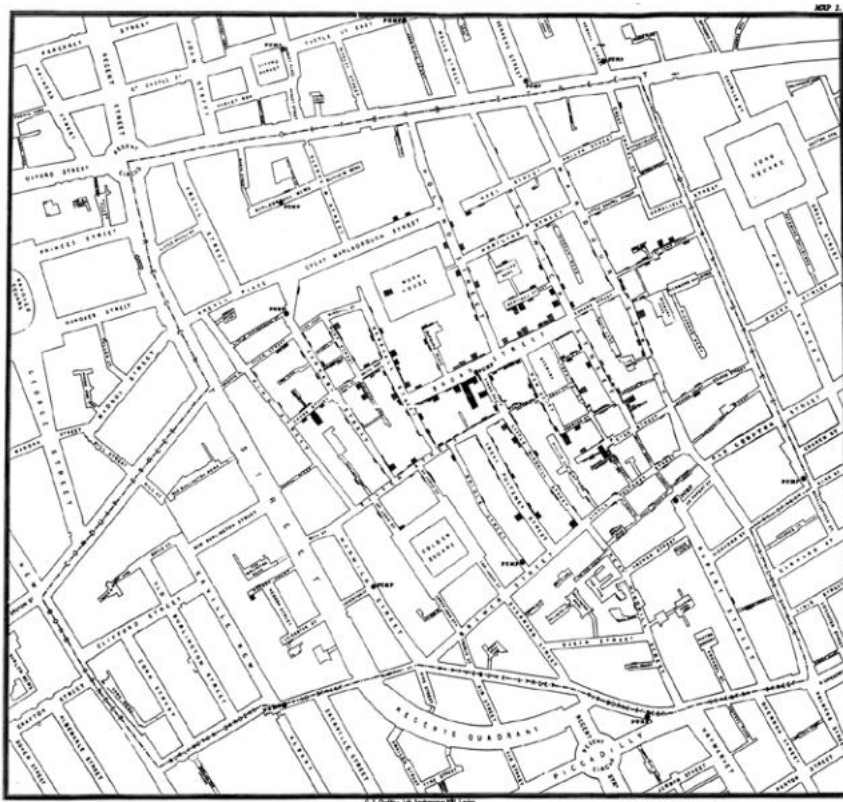


Рис. 14.10. Кластеры случаев заболевания холерой

З д йте себе следующие вопросы.

- Почему не производится втом тическ я очистк д нных?
- К к визу лизиров ть ч сти конвейер д нных, в которые могут поп д ть «сточные воды»?
- Сколько времени в ш комп ния тр тит н очистку д нных, которую можно полностью втом тизиров ть?

Уровень 4. Непрерывная поставка разведочного анализа данных

Если вы знакомы с конкурсами Kaggle, вы можете заметить, что весь ее смысл состоит в генерации как можно более точных предсказаний. Однако конкурсы машинного обучения вовсе не ограничиваются предсказаниями. Конкурсы — междисциплинарная и весьма многосторонняя область знаний. С одной стороны, как далеко можно подходить с точки зрения причинно-следственных связей. Какие глубинные движущие силы модели? Можете ли вы пояснить, как конкретная модель генерирует предсказания? Ответить на этот вопрос помогут несколько библиотек Python: ELI5, SHAP и LIME. Все они помогают разобраться, как конкретная модель функционирует модели машинного обучения.

Как далеко машинного обучения можно подходить с точки зрения предсказания, когда в конечном итоге, как конкретная модель пришла к ответу, то, насколько безошибочно предсказание. В мире нелинейных вычислений и больших данных этот подход имеет свои плюсы. Определенные задачи машинного обучения, например распознавание изображений с помощью глубокого обучения, хорошо решаются при наличии больших объемов данных. Чем больше данных и вычислительных мощностей, тем безошибочнее будет предсказание.

Доводите ли вы созданные модели до стадии промышленной эксплуатации? Почему нет? Зачем вообще создавать модели машинного обучения, если никто их не будет использовать?

Какие неизвестные величины? Что можно узнать из данных? Зачастую конкурсы данных больше интересуют процесс, чем результат. Если зациклиться только на предсказаниях, можно упустить совершенно иные аспекты данных.

Уровень 5. Непрерывная поставка обычного ML и AutoML

Люди сопротивляются изменениям испокон веков. Луддиты — секретная организация английских рабочих-текстильщиков, протестовавших против изменения путем разрушения текстильных станков с 1811 по 1816 год. В конце концов протестующие были расстреляны, восстание подавлено с помощью военных и законодательных мер, и прогресс продолжился.

На протяжении всей истории человечества непрерывно разрабатывались средства изменения выполняемых вручную задач. В ходе технологического прогресса низкоквалифицированных рабочих постепенно увольняли, высококвалифицированные получили прибавку к заработной плате. Постоянный

пример — системные администраторы и специалисты по DevOps. Да, потеряли робота часть системных администраторов, но пример с ним вписывается с меной жестких дисков в центр обработки данных, но возникли новые, более высокооплачиваемые должности и подобие архитекторов облачных сервисов.

Нередко можно увидеть в компании специалистов по машинному обучению и уже одних с годовой зарплатой от 300 тыс. до 1 млн долларов. Эти должности нередко включают многие, по существу, компоненты бизнес-регламента: тонкую настройку гиперпараметров, удаление пустых знаний и распределение знаний по кластерам. Придуманное Ноем proverbium в том тизотипе: «Если заходит речь об этом тизе чего-то, рано или поздно это что-то будет этим тизировано». Про AutoML говорят уже давно, так что в этом тизе знаний читательной чистоты машинного обучения неизбежны.

Это значит, что, как и в других примерах в этом тизе, изменится сущность должностей. Для некоторых уровень квалификации даже повысится (представьте себе человека, способного обучить тысячи моделей машинного обучения в день), некоторые будут этим тизированы, поскольку компьютер может выполнять их намного лучше (должности, связанные с тонкой настройкой знаний в JSON-структурах данных, то есть подстройкой гиперпараметров).

Уровень 6. Цикл обратной связи эксплуатации ML

Для чего разрабатываются мобильные приложения? Вероятно, чтобы пользователи применяли их на своих мобильных устройствах. А как насчет машинного обучения? Весь смысл машинного обучения, особенно по сравнению с наукой о данных или статистикой, состоит в создании модели и предсказании чего-либо. Если модель не работает в промышленной эксплуатации, то зачем он нужен?

Кроме того, ввод модели в промышленную эксплуатацию — это возможность узнать больше. Насколько точные предсказания дает модель в среде, где он получает совершенно новые для себя данные? Влияет ли модель на пользователей ожидаемым образом, то есть приводит ли к росту продаж и длительности пребывания на сайте? Эту важную информацию можно получить только при развертывании модели в среде реальной эксплуатации.

Еще один важный аспект — масштабируемость и повторяемость. Действительно зрелая в технологическом отношении организация может развернуть программное обеспечение, включая модели машинного обучения, по требованию. Здесь также необходимо применять для моделей ML рекомендуемые практики DevOps: непрерывное развертывание, микросервисы, мониторинг и телеметрию.

Один из простых способов достичь подобной технологической зрелости в в шей орг низ ции — воспользов ться той же логикой, бл год ря которой вы выбр ли обл чные вычисления, не физический центр обр ботки д нных. Арендуйте чужие зн ния и н выки и воспользуйтесь эффектом м шт б .

Приложение sklearn Flask с использованием Docker и Kubernetes

Р ссмотрим пример н стоящего р звертыв ния модели м шинного обучения н основе sklearn с помощью Docker и Kubernetes.

Д лее приведен Dockerfile, предн зн ченный для подготовки приложения Flask. Это приложение послужит для р змещения приложения sklearn. Возможно, вы з -хотите уст новить Hadolint для линтинг Dockerfile: <https://github.com/hadolint/hadolint>.

```
FROM python:3.7.3-stretch

# Рабочий каталог
WORKDIR /app

# Копируем исходный код в рабочий каталог
COPY . app.py /app/

# Установим пакеты из requirements.txt
# hadolint ignore=DL3013
RUN pip install --upgrade pip &&\
    pip install --trusted-host pypi.python.org -r requirements.txt

# Открываем порт 80
EXPOSE 80

# Выполняем app.py при запуске контейнера
CMD ["python", "app.py"]
```

Makefile, игр ющий роль отпр вной точки среды выполнения приложения:

```
setup:
    python3 -m venv ~/.python-devops

install:
    pip install --upgrade pip &&\
    pip install -r requirements.txt

test:
    #python -m pytest -vv --cov=myrepolib tests/*.py
    #python -m pytest --nbval notebook.ipynb
```

```
lint:
    hadolint Dockerfile
    pylint --disable=R,C,W1203 app.py
```

```
all: install lint test
```

Ф йл requirements.txt:

```
Flask==1.0.2
pandas==0.24.2
scikit-learn==0.20.3
```

Ф йл app.py:

```
from flask import Flask, request, jsonify
from flask.logging import create_logger
import logging

import pandas as pd
from sklearn.externals import joblib
from sklearn.preprocessing import StandardScaler

app = Flask(__name__)
LOG = create_logger(app)
LOG.setLevel(logging.INFO)

def scale(payload):
    """Масштабирование нагрузки"""

    LOG.info(f"Scaling Payload: {payload}")
    scaler = StandardScaler().fit(payload)
    scaled_adhoc_predict = scaler.transform(payload)
    return scaled_adhoc_predict

@app.route("/")
def home():
    html = "<h3>Sklearn Prediction Home</h3>"
    return html.format(format)

# СДЕЛАТЬ: вывести в журнал значение предсказания
@app.route("/predict", methods=['POST'])
def predict():
    """Предсказание с помощью sklearn

    Входные данные имеют следующий вид:
    {
    "CHAS":{
        "0":0
    },
    "RM":{
        "0":6.575
```

```

    },
    "TAX":{
        "0":296.0
    },
    "PTRATIO":{
        "0":15.3
    },
    "B":{
        "0":396.9
    },
    "LSTAT":{
        "0":4.98
    }
}

```

Результаты имеют следующий вид:

```
{ "prediction": [ 20.35373177134412 ] }
```

```

json_payload = request.json
LOG.info(f"JSON payload: {json_payload}")
inference_payload = pd.DataFrame(json_payload)
LOG.info(f"inference payload DataFrame: {inference_payload}")
scaled_payload = scale(inference_payload)
prediction = list(clf.predict(scaled_payload))
return jsonify({'prediction': prediction})

```

```

if __name__ == "__main__":
    clf = joblib.load("boston_housing_prediction.joblib")
    app.run(host='0.0.0.0', port=80, debug=True)

```

Ф ИЛ run_docker.sh:

```

#!/usr/bin/env bash

# Сборка образа [Docker]
docker build --tag=flasksklearn .

# Вывод списка образов Docker
docker image ls

# Запуск приложения Flask
docker run -p 8000:80 flasksklearn

```

Ф ИЛ run_kubernetes.sh:

```

#!/usr/bin/env bash

dockerpath="noahgift/flasksklearn"

# Запускаем в контейнере Docker Hub с помощью Kubernetes
kubectl run flasksklearn-demo\

```

```
--generator=run-pod/v1\  
--image=$dockerpath\  
--port=80 --labels app=flaskskearIndemo  
  
# Выводим список модулей Kubernetes  
kubectl get pods  
  
# Перенаправление порта контейнера на порт хоста  
kubectl port-forward flaskskearIndemo 8000:80  
  
#!/usr/bin/env bash  
# Присваиваем образу Docker тег и загружаем его в Docker Hub  
  
# Предполагаем, что все собрано  
#docker build --tag=flasksklearn .  
dockerpath="noahgift/flasksklearn"  
  
# Аутентификация и создание тега  
echo "Docker ID and Image: $dockerpath"  
docker login &&\  
    docker image tag flasksklearn $dockerpath  
  
# Помещаем образ в реестр  
docker image push $dockerpath
```

Н верное, в м интересно, к к модель созд ется, з тем выгруж ется/з грузж - ется. Вот тут (https://oreil.ly/_pHz-) вы можете н йти весь блокнот.

Во-первых, импортируем библиотеки для м шинного обучения:

```
import numpy  
from numpy import arange  
from matplotlib import pyplot  
import seaborn as sns  
import pandas as pd  
from pandas import read_csv  
from pandas import set_option  
from sklearn.preprocessing import StandardScaler  
from sklearn.model_selection import train_test_split  
from sklearn.model_selection import KFold  
from sklearn.model_selection import cross_val_score  
from sklearn.model_selection import GridSearchCV  
from sklearn.linear_model import LinearRegression  
from sklearn.linear_model import Lasso  
from sklearn.linear_model import ElasticNet  
from sklearn.tree import DecisionTreeRegressor  
from sklearn.neighbors import KNeighborsRegressor  
from sklearn.svm import SVR  
from sklearn.pipeline import Pipeline
```

```

from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.metrics import mean_squared_error

```

In[0]:

```

boston_housing = "https://raw.githubusercontent.com/
noahgift/boston_housing_pickle/master/housing.csv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS',
'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX',
'PTRATIO', 'B', 'LSTAT', 'MEDV']
df = read_csv(boston_housing, delim_whitespace=True, names=names)

```

In[0]:

```
df.head()
```

Out[0]:

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE |
|---|---------|------|-------|------|-------|-------|------|
| 0 | 0.00632 | 18.0 | 2.31 | 0 | 0.538 | 6.575 | 65.2 |
| 1 | 0.02731 | 0.0 | 7.07 | 0 | 0.469 | 6.421 | 78.9 |
| 2 | 0.02729 | 0.0 | 7.07 | 0 | 0.469 | 7.185 | 61.1 |
| 3 | 0.03237 | 0.0 | 2.18 | 0 | 0.458 | 6.998 | 45.8 |
| 4 | 0.06905 | 0.0 | 2.18 | 0 | 0.458 | 7.147 | 54.2 |

| | DIS | RAD | TAX | PTRATIO | B | LSTAT | MEDV |
|---|--------|-----|-------|---------|--------|-------|------|
| 0 | 4.0900 | 1 | 296.0 | 15.3 | 396.90 | 4.98 | 24.0 |
| 1 | 4.9671 | 2 | 242.0 | 17.8 | 396.90 | 9.14 | 21.6 |
| 2 | 4.9671 | 2 | 242.0 | 17.8 | 392.83 | 4.03 | 34.7 |
| 3 | 6.0622 | 3 | 222.0 | 18.7 | 394.63 | 2.94 | 33.4 |
| 4 | 6.0622 | 3 | 222.0 | 18.7 | 396.90 | 5.33 | 36.2 |

Разведочный анализ данных

Признаки модели:

- *CHAS* — фиктивное значение для реки Чарльз (1, если участок граничит с рекой, 0 в противном случае);
- *RM* — среднее количество комнат в доме;
- *TAX* — полный натуральный логarithм недвижимости из расчета на 10 000 долларов;
- *PTRATIO* — число учащихся на одного преподавателя;
- *Bk* — доля чернокожих по городским;

- *LSTAT* — процент р бот ющих мужчин без среднего обр зов ния;
- *MEDV* — меди нн я стоимость не рендуемых домов в тысяч х долл ров.

In[0]:

```
prices = df['MEDV']
df = df.drop(['CRIM', 'ZN', 'INDUS', 'NOX', 'AGE', 'DIS', 'RAD'], axis = 1)
features = df.drop('MEDV', axis = 1)
df.head()
```

Out[0]:

| | CHAS | RM | TAX | PTRATIO | B | LSTAT | MEDV |
|---|------|-------|-------|---------|--------|-------|------|
| 0 | 0 | 6.575 | 296.0 | 15.3 | 396.90 | 4.98 | 24.0 |
| 1 | 0 | 6.421 | 242.0 | 17.8 | 396.90 | 9.14 | 21.6 |
| 2 | 0 | 7.185 | 242.0 | 17.8 | 392.83 | 4.03 | 34.7 |
| 3 | 0 | 6.998 | 222.0 | 18.7 | 394.63 | 2.94 | 33.4 |
| 4 | 0 | 7.147 | 222.0 | 18.7 | 396.90 | 5.33 | 36.2 |

Моделирование

В этой ч сти блокнот производится собственно моделиров ние. Для удобств мы всегд делим блокнот н четыре основных р здел :

- ввод д нных;
- р звездочный н лиз д нных;
- моделиров ние;
- з ключение.

В посвященном моделиров нию р зделе д нные извлека ются из объект DataFrame и перед ются в модуль `train_test_split` sklearn, который и берет н себя весь труд по р збиению д нных н обуч ющие и проверочные.

Разбиение данных

In[0]:

```
# Отделяем проверочный набор данных
array = df.values
X = array[:,0:6]
Y = array[:,6]
validation_size = 0.20
seed = 7
X_train, X_validation, Y_train, Y_validation = train_test_split(X, Y,
    test_size=validation_size, random_state=seed)
```


In[0]:

```
for sample in list(X_validation)[0:2]:
    print(f"X_validation {sample}")
```

Out[0]:

```
X_validation [ 1.      6.395 666.    20.2  391.34  13.27 ]
X_validation [ 0.      5.895 224.    20.2  394.81  10.56 ]
```

Тонкая настройка масштабированного GBM¹

В этой модели используется несколько продвинутых методик, которые можно встретить во многих успешных проектах Kaggle. В их числе решетчатый поиск (grid search) оптимальных гиперпараметров. Обратите внимание также на масштабирующие функции. Для безошибочных предсказаний большинству алгоритмов машинного обучения требуется какое-либо масштабирующее.

In[0]:

```
# Опции проверки и метрика оценки на основе метода
# среднеквадратичной ошибки
num_folds = 10
seed = 7
RMS = 'neg_mean_squared_error'
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
param_grid = dict(n_estimators=np.array([50,100,150,200,250,300,350,400]))
model = GradientBoostingRegressor(random_state=seed)
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=RMS,
cv=kfold)
grid_result = grid.fit(rescaledX, Y_train)

print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Out[0]:

```
Best: -11.830068 using {'n_estimators': 200}
-12.479635 (6.348297) with: {'n_estimators': 50}
-12.102737 (6.441597) with: {'n_estimators': 100}
-11.843649 (6.631569) with: {'n_estimators': 150}
```

¹ Gradient boosting machines — метод градиентного бустинга. — *Примеч. пер.*

```
-11.830068 (6.559724) with: {'n_estimators': 200}
-11.879805 (6.512414) with: {'n_estimators': 250}
-11.895362 (6.487726) with: {'n_estimators': 300}
-12.008611 (6.468623) with: {'n_estimators': 350}
-12.053759 (6.453899) with: {'n_estimators': 400}
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:841:
DeprecationWarning:
DeprecationWarning)
```

Подгонка модели

Подгонку модели мы будем производить с помощью `GradientBoostingRegressor`. Последний шаг после обучения модели — только подгонка и проверка погрешности на выделенных данных. Данные масштабируются, передаются в модель, и оцениваются по качеству безошибочности по метрике «среднеквадратическая ошибка».

```
In[0]:
```

```
# Подготовка модели
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
model = GradientBoostingRegressor(random_state=seed, n_estimators=400)
model.fit(rescaledX, Y_train)
# Преобразование проверочного набора данных
rescaledValidationX = scaler.transform(X_validation)
predictions = model.predict(rescaledValidationX)
print("Mean Squared Error: \n")
print(mean_squared_error(Y_validation, predictions))
```

```
Out[0]:
```

```
Mean Squared Error:
```

```
26.326748591395717
```

Оценка работы модели

Один из самых важных спектров машинного обучения — оценка модели. Пример демонстрирует добавление в один и тот же объект `DataFrame` предсказаний и исходной цены недвижимости. Далее на основе этого `DataFrame` можно вычислить разность между ними.

```
In[0]:
```

```
predictions=predictions.astype(int)
evaluate = pd.DataFrame({
```

```

        "Org House Price": Y_validation,
        "Pred House Price": predictions
    })
evaluate["difference"] = evaluate["Org House Price"]-evaluate["Pred House
Price"]
evaluate.head()

```

Результаты приведены далее.

Out[0]:

| | Org house price | Pred house price | Difference |
|---|-----------------|------------------|------------|
| 0 | 21.7 | 21 | 0.7 |
| 1 | 18.5 | 19 | -0.5 |
| 2 | 22.2 | 20 | 2.2 |
| 3 | 20.4 | 19 | 1.4 |
| 4 | 8.8 | 9 | -0.2 |

Для просмотра распределения данных в Pandas очень удобен метод `describe`:

In[0]:

```
evaluate.describe()
```

Out[0]:

| | Org house price | Pred house price | Difference |
|-------|-----------------|------------------|------------|
| count | 102.000000 | 102.000000 | 102.000000 |
| mean | 22.573529 | 22.117647 | 0.455882 |
| std | 9.033622 | 8.758921 | 5.154438 |
| min | 6.300000 | 8.000000 | -34.100000 |
| 25% | 17.350000 | 17.000000 | -0.800000 |
| 50% | 21.800000 | 20.500000 | 0.600000 |
| 75% | 24.800000 | 25.000000 | 2.200000 |
| max | 50.000000 | 56.000000 | 22.000000 |

adhoc_predict

Проверим эту модель для предсказания и посмотрим на ее работу после загрузки. При запуске веб-API для модели машинного обучения желательны протестировать чистый код, который API будет выполнять в своем блокноте. Отладка и создание функций в блокноте намного проще, чем мучиться с созданием нужных функций в веб-приложении.

In[0]:

```
actual_sample = df.head(1)
actual_sample
```

Out[0]:

| | CHAS | RM | TAX | PTRATIO | B | LSTAT | MEDV |
|---|------|-------|-------|---------|-------|-------|------|
| 0 | 0 | 6.575 | 296.0 | 15.3 | 396.9 | 4.98 | 24.0 |

```
In[0]:
```

```
adhoc_predict = actual_sample[["CHAS", "RM", "TAX", "PTRATIO", "B", "LSTAT"]]
adhoc_predict.head()
```

```
Out[0]:
```

| | CHAS | RM | TAX | PTRATIO | B | LSTAT |
|---|------|-------|-------|---------|-------|-------|
| 0 | 0 | 6.575 | 296.0 | 15.3 | 396.9 | 4.98 |

Технологический процесс JSON

Этот раздел блокнота удобен для отладки приложений Flask. Как упомянул ранее, не много проще просмотреть код API в проекте машинного обучения, убедиться в его правильности, а затем перенести в сценарий. Альтернатива не так привлекательна: пытаться достичь нужного синтаксиса кода в программном проекте, где нет таких интерактивных инструментов, как в Jupyter.

```
In[0]:
```

```
json_payload = adhoc_predict.to_json()
json_payload
```

```
Out[0]:
```

```
{"CHAS":{"0":0},"RM":
{"0":6.575},"TAX":
{"0":296.0},"PTRATIO":
{"0":15.3},"B":{"0":396.9},"LSTAT":
{"0":4.98}}
```

Масштабирование входных данных

Для предсказания необходимо обработать масштабированные данные. Следует произвести эту операцию в блокноте, не пытаясь заставить ее работать в веб-приложении, отладка которого — не много более трудная задача. Ниже приведен код для реализации этой части конвейера машинного обучения. В дальнейшем его можно использовать для создания функции в приложении Flask.

```
In[0]:
```

```
scaler = StandardScaler().fit(adhoc_predict)
scaled_adhoc_predict = scaler.transform(adhoc_predict)
```

```
scaled_adhoc_predict
```

```
Out[0]:
```

```
array([[0., 0., 0., 0., 0., 0.]])
```

```
In[0]:
```

```
list(model.predict(scaled_adhoc_predict))
```

```
Out[0]:
```

```
[20.35373177134412]
```

Выгрузка

Теперь экспортируем нашу модель.

```
In[0]:
```

```
from sklearn.externals import joblib1
```

```
In[0]:
```

```
joblib.dump(model, 'boston_housing_prediction.joblib')
```

```
Out[0]:
```

```
['boston_housing_prediction.joblib']
```

```
In[0]:
```

```
!ls -l
```

```
Out[0]:
```

```
total 672
-rw-r--r-- 1 root root 681425 May  5 00:35 boston_housing_prediction.joblib
drwxr-xr-x 1 root root  4096 Apr 29 16:32 sample_data
```

Загрузка и предсказание

```
In[0]:
```

```
clf = joblib.load('boston_housing_prediction.joblib')
```

¹ В некоторых версиях это не работает. Проще импортировать joblib непосредственно: `import joblib`. См. <https://joblib.readthedocs.io/en/latest/>. — *Примеч. пер.*

adhoc_predict на основе выгрузки

In[0]:

```
actual_sample2 = df.head(5)
actual_sample2
```

Out[0]:

| | CHAS | RM | TAX | PTRATIO | B | LSTAT | MEDV |
|---|------|-------|-------|---------|--------|-------|------|
| 0 | 0 | 6.575 | 296.0 | 15.3 | 396.90 | 4.98 | 24.0 |
| 1 | 0 | 6.421 | 242.0 | 17.8 | 396.90 | 9.14 | 21.6 |
| 2 | 0 | 7.185 | 242.0 | 17.8 | 392.83 | 4.03 | 34.7 |
| 3 | 0 | 6.998 | 222.0 | 18.7 | 394.63 | 2.94 | 33.4 |
| 4 | 0 | 7.147 | 222.0 | 18.7 | 396.90 | 5.33 | 36.2 |

In[0]:

```
adhoc_predict2 = actual_sample[["CHAS", "RM", "TAX", "PTRATIO", "B", "LSTAT"]]
adhoc_predict2.head()
```

Out[0]:

| | CHAS | RM | TAX | PTRATIO | B | LSTAT |
|---|------|-------|-------|---------|-------|-------|
| 0 | 0 | 6.575 | 296.0 | 15.3 | 396.9 | 4.98 |

Масштабирование входных данных

In[0]:

```
scaler = StandardScaler().fit(adhoc_predict2)
scaled_adhoc_predict2 = scaler.transform(adhoc_predict2)
scaled_adhoc_predict2
```

Out[0]:

```
array([[0., 0., 0., 0., 0., 0.]])
```

In[0]:

```
# Используем загруженную модель
list(clf.predict(scaled_adhoc_predict2))
```

Out[0]:

```
[20.35373177134412]
```

Наконец, загружаем выгруженную модель обратно и проверяем на тестовых данных.

Вопросы и упражнения

- Назовите важнейшие различия между scikit-learn и PyTorch.
- Что такое AutoML и почему его имеет смысл использовать?
- Измените нашу модель scikit-learn для предсказания веса по росту.
- Выполните наш пример PyTorch в блокноте Google Colab, переключаясь между средой выполнения GPU и CPU. Поясните различия в быстродействии, если таковые наблюдаются.
- Что такое звездочный индекс данных и почему он так важен в проектах науки о данных?

Задача на ситуационный анализ

Зайдите на веб-сайт Kaggle, выберите какой-либо популярный блокнот на Python и преобразуйте его в контейнеризованное приложение Flask, выходящее за пределы, воспользовавшись приведенным в этой главе примером в качестве ориентира. Затем разверните его в облачной среде с помощью хостируемого сервиса Kubernetes, например Amazon EKS.

Вопросы на проверку усвоения материала

- Расскажите о различных типах фреймворков и экосистем машинного обучения.
- Запустите и отладьте уже существующий проект машинного обучения в scikit-learn и PyTorch.
- Контейнеризуйте модель scikit-learn Flask.
- Разберитесь с промышленной моделью зрелости машинного обучения.

ГЛАВА 15

Инженерия данных

Возможно, н у к о д ных — с м я привлек тельн я профессия XXI столетия, но конкретные специ льности быстро меняются по мере ее р звития. Исследов тель д ных — слишком общее н зв ние для множеств р зличных з д ч. По состоянию н 2020 год две примерно р вноценные специ льности — это специ лист по д нным и специ лист по м шинному обучению.

Но еще больше пор ж ет то, к к много специ листов по д нным необходимо для поддержки обычного исследов теля д ных — от трех до пяти инженеров по р боте с д нными н одного исследов теля д нных.

Почему т к? Взглянем н это с другой точки зрения: предст вьте себе, что вы придумыв ете з головки для г зетных ст тей и хотели бы придум ть что-то з - помин ющееся. Н пример, «Генер льный директор — с м я привлек тельн я должность для бог тых». Генер льных директоров немного, к к немного звезд НБА или профессион льных ктеров. Сколько людей должно р бот ть, чтобы обеспечить успех одного генер льного директора ? Последнее утверждение столь же бессодерж тельно и бессмысленно, к к и утверждение «вод мокр я».

Мы не хотим этим ск з ть, что нельзя з р бот ть себе н жизнь н укой о д нных, просто критикуем логику с мого утверждения. Спрос н н выки р боты с д нными огромен, от DevOps до м шинного обучения и телекоммуник ций. Термин «исследов тель д нных» очень р сплывч т. В к ком-то смысле он очень н помин ет этим слово DevOps. Что т кое DevOps — специ льность или стиль р боты?

Если взглянуть н опис ния в к нсий и з рпл т, ст новится очевиден спрос н специ листов по д нным и систем м м шинного обучения. А все потому, что в их должностные обяз нности входят четко определенные з д чи. В число з д ч инженер по р боте с д нными может входить, н пример, созд ние конвейер в обл ке для сбор к п кетных, т к и потоковых д нных с последующим соз-

данием API для доступ к ним, также планирование выполнения этих заданий. Это задание однозначно: конвейер либо работает, либо нет.

Аналогично проектировщик систем машинного обучения занимается созданием моделей машинного обучения и развертыванием их так, чтобы упростить сопровождение. Это также вполне однозначно задание. Впрочем, конкретный специлист может заниматься инженерией данных или машинным обучением и все равно вести себя в соответствии с практиками науки о данных и DevOps. Сегодня самое время заняться роботизацией, поскольку существует немало прекрасных возможностей создать сложные устойчивые к ошибкам конвейеры для обслуживания данными других сложных и обладающих широкими возможностями прогнозных систем. Есть высказывание: «Нельзя быть слишком богатым или слишком стройным». Точно также и с данными: навыков DevOps или роботизации данными никогда не бывает слишком много. Обсудим подробнее приправленные DevOps идеи инженерии данных.

Малые данные

Наличие инструментов — замечательная концепция. Если вызвать человека с техник, обычно он приезжает с инструментами, позволяющими ему справиться с проблемой гораздо эффективнее, чем могли бы вы. Найм в мире для каких-либо работ плотник также приходит со своим набором специальных инструментов, благодаря которому может выполнить работу много быстрее, чем вы. Инструменты жизненно важны для профессионалов, и специлисты по DevOps не исключение.

В этом разделе мы рассмотрим утилиты для инженерии данных. Они включают инструменты для небольших заданий по работе с данными, например для чтения и записи файлов, использования библиотеки `pickle`, форматов `JSON`, также записи и чтения файлов `YAML`. Свободная работа с этими форматами — необходимый навык для специалиста по автоматизации, который хочет уметь превратить любую задачу в сценарий. Далее в этой главе мы обсудим утилиты для работы с большими данными, которые принципиально отличаются от утилит для работы с малыми данными.

Что же такое большие данные, что — малые? Самый простой способ различить их — так называемый ноутбучный тест. Можно ли с ними работать на ноутбуке? Если нет — это большие данные. Хороший пример — `Pandas`. Эта библиотека требует в 5–10 раз больше оперативной памяти, чем размер наборов данных. Скорее всего, для обработки в `Pandas` файл размером 2 Гб вшего ноутбук будет недоступен.

Обработка файлов малых данных

Если и есть определяющая черта Python — это неустанное стремление к максимальной эффективности языка. Обычный программист Python старается написать ровно столько код, сколько нужно для решения задачи, не доводя его при этом до излишней сложности и неудобочитаемости. Кроме того, обычный программист Python не желает писать стереотипный код. Таким образом, язык приводит к непрерывной эволюции полезных паттернов.

Один из примеров часто используемых паттернов — использование оператор `with` для чтения и записи файлов. Оператор `with` берет на себя утомительные стереотипные действия по закрытию файла после завершения работы с ним. Оператор `with` используется и в других частях языка Python для облегчения различных утомительных задач.

Запись в файл

Этот пример иллюстрирует в том числе закрытие файла при использовании оператора `with` по завершении выполнения блока кода. Такой синтаксис помогает предотвратить программные ошибки, которые неизбежны, если программист случайно забудет закрыть дескриптор файла:

```
with open("containers.txt", "w") as file_to_write:
    file_to_write.write("Pod\n")
    file_to_write.write("Service\n")
    file_to_write.write("Volume\n")
    file_to_write.write("Namespace\n")
```

Содержимое файла при этом выглядит следующим образом:

```
cat containers.txt
```

```
Pod
Service
Volume
Namespace
```

Чтение файла

Контекст `with` рекомендуется использовать также для чтения файлов. Обратите внимание на то, что метод `readlines()` возвращает выполняемый отложенным образом итератор на основе вызовов строк:

```
with open("containers.txt") as file_to_read:
    lines = file_to_read.readlines()
    print(lines)
```

Вывод:

```
['Pod\n', 'Service\n', 'Volume\n', 'Namespace\n']
```

На практике это дает возможность обработать большие файлы журналов с помощью генераторов, не боясь израсходовать всю оперативную память машины.

Конвейер с генератором для чтения и обработки строк

Далее приведен код функции-генератора, который открывает файл и возвращает генератор:

```
def process_file_lazyily():
    """Отложенная обработка файла с помощью генератора"""

    with open("containers.txt") as file_to_read:
        for line in file_to_read.readlines():
            yield line
```

Далее мы создадим на основе этого генератора конвейер для построчного выполнения операций. В примере строка переводится в нижний регистр, но здесь можно соединить в цепочку множество других операций, и эффективность будет очень высокой, поскольку достигнута точная обработка строки:

```
# Создание объекта-генератора
pipeline = process_file_lazyily()
# Преобразование в нижний регистр
lowercase = (line.lower() for line in pipeline)
# Выводим первую обработанную строку
print(next(lowercase))
```

Результат работы конвейера:

```
pod
```

На практике это значит, что размер файлов может быть, по существу, неограниченным, если робот код выполняется при удовлетворении определенного условия. Например, представьте, что нужно найти идентификатор покупателя в тербйтном массиве данных. Конвейер с генератором может роботать следующим образом: найти этот идентификатор покупателя и выйти из цикла обработки при первом же найденном вхождении. В мире больших данных это отнюдь не теоретическая проблема.

YAML

YAML постепенно становится стандартом для связанных с DevOps файлов конфигурации. YAML — удобный для восприятия человеком формат сериализации данных, представляющий собой расширение JSON. Название его расшифровывается как *YAML Ain't Markup Language* («YAML — это не язык разметки»). YAML можно нередко встретить в таких системах сборки, как AWS CodePipeline (<https://oreil.ly/WZnII>), CircleCI (<https://oreil.ly/0r8cK>), и таких вриантах PaaS, как Google App Engine (https://oreil.ly/ny_TD).

YAML используется так широко, что совсем не случайно, вследствие потребности в языке конфигурации с возможностью быстрых итераций при взаимодействии с высоконагруженными системами. Как программисту, так и обычному пользователю интуитивно ясно, как редактировать эти файлы. Вот пример:

```
import yaml

kubernetes_components = {
    "Pod": "Basic building block of Kubernetes.",
    "Service": "An abstraction for dealing with Pods.",
    "Volume": "A directory accessible to containers in a Pod.",
    "Namespaces": "A way to divide cluster resources between users."
}

with open("kubernetes_info.yaml", "w") as yaml_to_write:
    yaml.safe_dump(kubernetes_components, yaml_to_write, default_flow_style=False)
```

Записанный на диск результат выглядит следующим образом:

```
cat kubernetes_info.yaml

Namespaces: A way to divide cluster resources between users.
Pod: Basic building block of Kubernetes.
Service: An abstraction for dealing with Pods.
Volume: A directory accessible to containers in a Pod.
```

Отсюда ясно, насколько просто сериализовать структуру данных Python в удобный для редактирования и интерактивной обработки формат. Чтение этого файла также требует всего двух строк кода:

```
import yaml

with open("kubernetes_info.yaml", "rb") as yaml_to_read:
    result = yaml.safe_load(yaml_to_read)
```

А далее можно вывести результаты во вполне привычном виде:

```
import pprint
pp = pprint.PrettyPrinter(indent=4)
```

```
pp.pprint(result)
{
  'Namespaces': 'A way to divide cluster resources between users.',
  'Pod': 'Basic building block of Kubernetes.',
  'Service': 'An abstraction for dealing with Pods.',
  'Volume': 'A directory accessible to containers in a Pod.'}
```

Большие данные

Результаты вычислений растут быстрее, чем вычислительные мощности компьютеров. Еще более интересной является ситуация то, что закон Мур, согласно которому скорость и возможности компьютеров должны удваиваться каждые два года, фактически больше не работает и начиная с примерно 2015 года согласно доктору Дэвиду Паттерсону из Калифорнийского университета в Беркли. Темпы роста CPU растут сейчас с лишь на 3% в год.

Необходимы новые методы работы с большими данными, в том числе такие ASIC, как GPU, тензорные процессоры (TPU), также ИИ и платформы данных для поставщиков облачных сервисов. На уровне микросхем это значит, что лучше всего ориентироваться на использование в сложных ИТ-процессах GPU вместо CPU. Зачастую такой GPU сочетается с системой, обеспечивающей распределенный механизм хранения, что делает возможными как распределенные вычисления, так и распределенные операции дискового ввода/вывода. Прекрасные примеры — Apache Spark, Amazon SageMaker и платформа ИИ Google. Все они могут применять ASIC (GPU, TPU и др.), также распределенное хранилище вместе с системой управления. Еще один, более низкоуровневый пример — API спотовых инстансов Amazon с точки зрения монтирования файловой системы Amazon Elastic File System (EFS).

Для специалистов по DevOps отсюда вытекают несколько вещей. Во-первых, поставщик программного обеспечения в эти системы должен производиться с дополнительной осторожностью. Например, стоит ответить на следующие вопросы: привильны ли драйверы GPU на целевой платформе? Производится ли контейнерное развертывание? Будет ли данная система использовать распределенную GPU-обработку? Данные преимущественно пакетные или потоковые? Заложив временное обдумывание этих вопросов имеет большое значение для разработки привильной архитектуры.

Одна из проблем с модными словечками вроде ИИ, «большие данные», «облачные вычисления» и «исследователи данных» — для разных людей их смысл может различаться. Рассмотрим, например, термин «исследователь данных» (data scientist). В одной компании это специалист, занимающийся созданием информационных панелей бизнес-аналитики для отдела продаж, в другой — разработчик программного обеспечения для самоуправляемых автомобилей.

У термин «большие данные» точно такие же проблемы с контекстом: в зависимости от того, с кем вы говорите, он может значить разное. Задумайтесь над одним из определений. Необходимы ли для обработки данных на ноутбуке те же пакеты программ, что и в среде промышленной эксплуатации?

Прекрасный пример утилиты для больших данных — пакет Pandas. По информации создателя этого пакета, он может использоваться в 5–10 раз больше оперативной памяти, чем размер обрабатываемого файла. Например, при 16 Гб оперативной памяти в ноутбуке и CSV-файлом размером 2 Гб из данных уже относится к большим данным, поскольку оперативной памяти ноутбук может не хватить для работы с этим файлом. Так что необходимо пересмотреть способ решения задачи. Например, можно открыть лишь фрагмент данных или изначально урезать массив данных.

Рассмотрим конкретный пример этой проблемы и ее обход. Представьте себе, что вы исследователи данных, который снова и снова получает ошибки нехватки памяти, поскольку берет слишком большие для Pandas файлы. Один из примеров таких файлов — набор данных Open Food Facts (<https://oreil.ly/w-tmA>) из Kaggle. В архивированном виде размер этого набора данных составляет более 1 Гб. Это значит, что входит в число проблемных для Pandas. Один из вариантов ее решения — воспользоваться командой `shuf` Unix для получения перетасованной выборки из набора данных:

```
time shuf -n 100000 en.openfoodfacts.org.products.tsv\
> 10k.sample.en.openfoodfacts.org.products.tsv
1.89s user 0.80s system 97% cpu 2.748 total
```

Менее чем за 2 секунды удалось уменьшить файл до вполне приемлемого размера. Лучше использовать этот подход, так как в нем происходит случайный выбор примеров данных, а не брать просто начало или конец набора данных. Это проблематично играет важную роль в технологическом процессе науки о данных. Кроме того, можно сначала посмотреть строки файла, чтобы понять, с чем приходится иметь дело:

```
wc -l en.openfoodfacts.org.products.tsv
356002 en.openfoodfacts.org.products.tsv
```

Исходный файл включает примерно 350 000 строк, так что 100 000 перетасованных строк составляет примерно треть всего массива данных. В этом можно убедиться, взглянув на преобразованный файл. Его размер составляет 272 Мб — около трети от размера исходного файла в 1 Гб:

```
du -sh 10k.sample.en.openfoodfacts.org.products.tsv
272M 10k.sample.en.openfoodfacts.org.products.tsv
```

Фйл т кого р змер лучше подходит для обр ботки Pandas н ноутбуке, с м процесс можно в д льнейшем преобр зов ть в в том тический технологический процесс созд ния р ндомизиров нных ф йлов выборок для источников, предст вляющих собой большие д нные. Подобный процесс — лишь один из множеств технологических процессов, необходимых для р боты с большими д нными.

Еще одно определение больших д нных было д но М ккинси (McKinsey): «Н боры д нных, р змер которых слишком велик для ввод , хр нения, обр ботки и н лиз ст нд ртными утилит ми б зд нных». Это определение т кже вполне логично, с небольшим уточнением: не только утилит ми б зд нных, вообще любыми утилит ми, р бот ющими с д нными. Если прекр сно р бот ющ я н ноутбуке утилит , н пример Pandas, Python, MySQL, утилит глубокого обучения/м шинного обучения, Bash и т. д., не может норм льно р бот ть из-з большого р змер или чрезмерной скорости изменения д нных, зн чит, речь идет о з д че больших д нных. Для решения з д ч больших д нных необходимы специ льные инструменты, и в следующем р зделе мы их и р ссмотрим.

Утилиты, компоненты и платформы для работы с большими данными

Обсуждение больших д нных можно т кже р збить н обсуждение отдельных утилит и пл тформ. Н рис. 15.1 приведен типичный жизненный цикл р хитектуры больших д нных.

Обсудим некоторые в жнейшие компоненты этой р хитектуры.

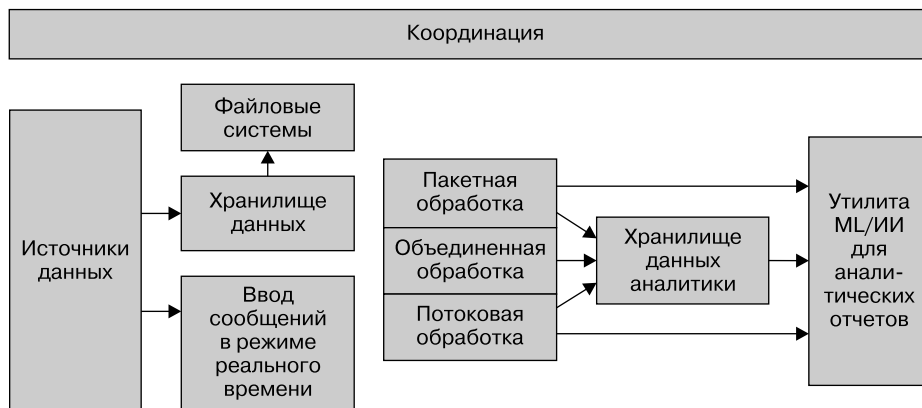


Рис. 15.1. Архитектура больших данных

Источники данных

В число типичных источников больших данных входят социальные сети и цифровые транзакции. По мере перевода большинства разговоров и деловых операций в онлайн-форму количество данных резко возросло. Кроме того, число источников данных растет экспоненциально с развитием мобильных технологий, например, использованием планшетов, телефонов и ноутбуков для записи аудио и видео.

Среди прочих источников данных — Интернет вещей (IoT), включающий разнообразные датчики, миниатюрные микросхемы и устройств. В результате возникает неуправляемый рост объемов данных, которые где-то нужно хранить. Дипломы утилит, связанных с источниками данных, очень широк: от клиент-серверных систем IoT, например AWS IoT Greengrass, до систем хранения объектов, например Amazon S3 и Google Cloud Storage.

Файловые системы

Файловые системы играют в обработке данных важнейшую роль, одна из них постоянно непрерывно эволюционирует. Одна из проблем, возникающих при работе с большими данными, — потребность в больших объемах дискового ввода/вывода для распределенных операций.

Одна из современных утилит, решающих эту проблему, — распределенная файловая система Hadoop (Hadoop Distributed File System, HDFS). В основе ее работы — группирование нескольких серверов, позволяющее агрегировать ресурсы CPU, операции дискового ввода/вывода и хранения данных. На практике это делает HDFS одной из базовых технологий работы с большими данными. Она позволяет перемещать большие массивы данных или файловые системы для распределенных вычислительных задач. Кроме того, она лежит в основе платформы Spark, подходящей как для потокового, так и для пакетного машинного обучения.

В числе прочих типов файловых систем — файловые системы хранения объектов, такие как файловая система Amazon S3 и хранение Google Cloud Platform. В них можно хранить распределенным образом огромные файлы, причем гарантированно их высокая доступность, точнее, надежность в 99,999999999 %. Для взаимодействия с этими файловыми системами существуют API Python и утилиты командной строки, что сильно упрощает работу с ними. Мы рассмотрим эти облачные API подробнее в главе 10.

Наконец, можно отметить и такой тип файловой системы, как обычная сетевая файловая система (NFS) в виде управляемого облачного сервиса. Прекрасный

пример той файловой системы — Amazon Elastic File System (Amazon EFS). Он представляет собой исключительно гибкий инструмент для специалистов по DevOps, особенно в сочетании с технологиями контейнеров. На рис. 15.2 приведен пример монтирования EFS в контейнере.

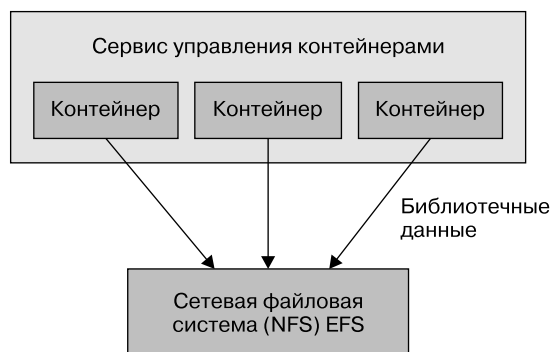


Рис. 15.2. Монтирование EFS в контейнере

Один из открывающих большие возможности процессов в том числе — создание программным образом контейнеров Docker с помощью системы сборки и подобие AWS CodePipeline или Google Cloud Build. Далее эти контейнеры регистрируются в облачном реестре контейнеров, например Amazon ECR. Затем система управления контейнерами, например Kubernetes, запускает контейнеры, монтирующие NFS. В результате возможности быстро запускаемых неизменяемых образов контейнеров сочетаются с доступом к централизованной библиотеке исходного кода и данных. Подобный технологический процесс идеально подходит для организаций, стремящихся оптимизировать операции машинного обучения.

Хранение данных

В конечном счете данные необходимо где-то хранить, что приводит к интересным возможностям и непростым задачам. В последнее время сложилась тенденция использовать для этой цели озера данных. Чем интересны озера данных? Тем, что они позволяют обрывать данные прямо в месте их хранения. В результате для многих озер данных требуются фактически бесконечные (то есть расплывающиеся в облаке) хранилища данных и вычислительные ресурсы. Поэтому для них нередко выбирают Amazon S3.

Подобные озера данных применимы и в конвейерах машинного обучения, связывающих конкретные данные в озерах данных, так и от обученных моделей. Обученные

модели можно после этого подвергнуть A/B-тестированию и убедиться, что последняя модель будет работать в системе предсказания (вывод) (рис. 15.3).

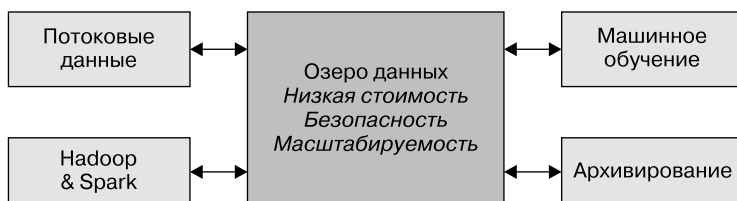


Рис. 15.3. Озеро данных

Все остальные виды хранения данных хорошо известны разработчикам традиционного программного обеспечения. В их числе реляционные базы данных, базы данных типа «ключ/значение», поисковые системы и подобие Elasticsearch, а также графовые базы данных. В архитектуре больших данных все эти типы систем хранения данных могут играть более конкретные роли. В небольшой системе все может отвечать реляционной базе данных, но в архитектуре больших данных неправильный подбор системы хранения недопустим.

Прекрасный пример неправильного выбора системы хранения — использование реляционной базы данных в качестве поисковой системы на основе полнотекстового поиска вместо специализированных решений и подобие Elasticsearch. Elasticsearch специально предназначен для создания масштабируемых поисковых решений, в то время как реляционная база данных — для обеспечения ссылочной целостности и транзакций. Технический директор Amazon Вернер Фогель (Werner Vogel) замечательно сформулировал это: «Одна база данных — ни для кого». Проблему иллюстрирует рис. 15.4, на котором показано, что у каждого типа баз данных свое предназначение.

Специализированные базы данных

| Реляционные | Типа «ключ/значение» | Документо-ориентированные | Графовые | Размещаемые в оперативной памяти | Поисковые |
|-------------|---|----------------------------------|----------------------------------|------------------------------------|--|
| Транзакции | Запросы «ключ/значение» с малым временем ожидания | Индексация и хранение документов | Создание отношений и их просмотр | Время ожидания порядка микросекунд | Индексация слабоструктурированных журналов и поиск в них |

Рис. 15.4. Базы данных Amazon

Выбор подходящих решений для хранения данных, в том числе выбор сочетания баз данных, — важнейший вызов для любого архитектора данных, желающего, чтобы система работала максимально эффективно. При проектировании полностью автоматизированной и эффективной системы необходимо учесть и сопровождение. При переносе на конкретную выбранную технологию, например реляционной базы данных для высокодоступной очереди сообщений, затраты на сопровождение несут значительный вес, что потребует дополнительной автоматизации. Так что следует учесть и объемы работы по автоматизации, необходимые для сопровождения решения.

Ввод данных в режиме реального времени

Потоковые данные реального времени — особенно сложный для работы тип данных. Поток сам по себе усложняет обработку данных, вдобавок к этому, возможно, придется перенести в другую часть системы — для потоковой обработки данных. Один из примеров облачных решений для потокового ввода данных — Amazon Kinesis Data Firehose (рис. 15.5).

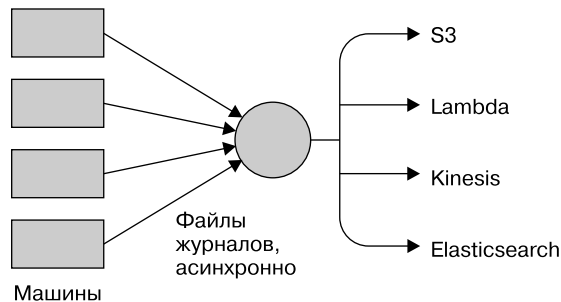


Рис. 15.5. Файлы журналов Kinesis

А вот пример соответствующего кода. Обратите внимание на использование модуля `asyncio` языка Python для массового параллельных однопоточных сетевых операций. Узлы могут выдавать подобные данные или журналы ошибок при отправке на ферму данных:

```
import asyncio

def send_async_firehose_events(count=100):
    """Асинхронная отправка событий в Firehose"""
    start = time.time()
    client = firehose_client()
    extra_msg = {"aws_service": "firehose"}
    loop = asyncio.get_event_loop()
```

```

tasks = []
LOG.info(f"sending aysnc events TOTAL {count}",extra=extra_msg)
num = 0
for _ in range(count):
    tasks.append(asyncio.ensure_future(put_record(gen_uuid_events(),
                                                    client)))
    LOG.info(f"sending aysnc events: COUNT {num}/{count}")
    num +=1
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
end = time.time()
LOG.info("Total time: {}".format(end - start))

```

Kinesis Data Firehose получает временные данные и непрерывно переносит их в произвольное количество точек назначения: Amazon S3, Amazon Redshift, сервис Amazon Elasticsearch или какие-либо сторонние сервисы и подобие Splunk. В качестве альтернативы Kinesis с открытым исходным кодом можно рассмотреть Apache Kafka. Apache Kafka работает на основе схожих принципов архитектуры обмена сообщениями по типу «издатель/подписчик».

Ситуационный анализ: создание доморощенного конвейера данных

Данным делом, в начале 2000-х, когда Нойс был техническим директором и главным менеджером стартапа, он столкнулся с задачей создания первого в компании конвейера машинного обучения и обработки данных. Примерный эскиз получившегося приведен на следующей схеме конвейера данных Jenkins (рис. 15.6).

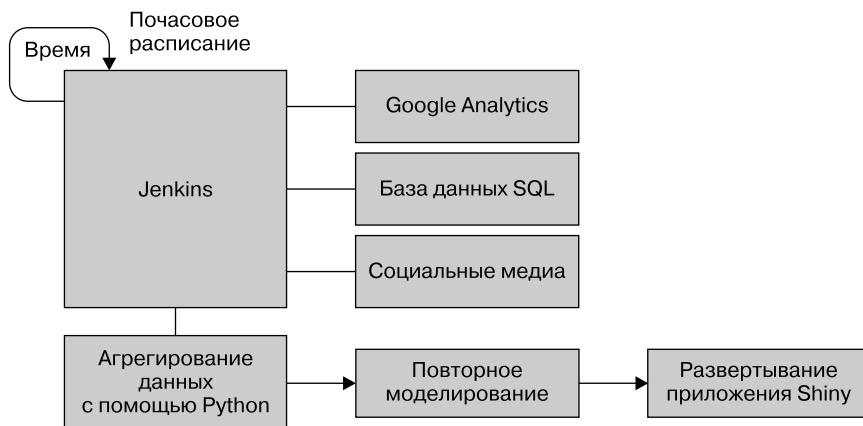


Рис. 15.6. Конвейер данных Jenkins

Источником входных данных для конвейера данных может быть все, что нужно для бизнес-аналитики или предсказаний с помощью машинного обучения. В числе типичных источников могут быть реляционные базы данных, Google Analytics, метрики социальных медиа, и это далеко не все. Задача по сбору данных выполняется раз в час, внутренний доступ к сгенерированному CSV-файлу обеспечивается с помощью веб-сервиса Apache. Это простое и привлекательное решение.

Смислами задачи представляются собой задачи Jenkins, которые просто запускают сценарии Python. При необходимости внесения каких-либо изменений можно легко отредактировать сценарии Python для конкретного задания. Дополнительное достоинство системы — простота отладки. Если задание завершилось неудачей, оно отображается как сбойное и можно легко взглянуть на выводимые им результаты и увидеть, что произошло.

Последний этап конвейера — предсказание на основе машинного обучения и информационная панель аналитики, отображаемая с помощью написанного на языке R приложения Shiny. Основное влияние на подобную архитектуру оказывают соображения простоты подхода, кроме того, он полноценно использует имеющиеся навыки DevOps.

Бессерверная инженерия данных

Еще одна перспективная технология — бессерверная инженерия данных. На рис. 15.7 приведен укрупненный схематический рисунок бессерверного конвейера данных.

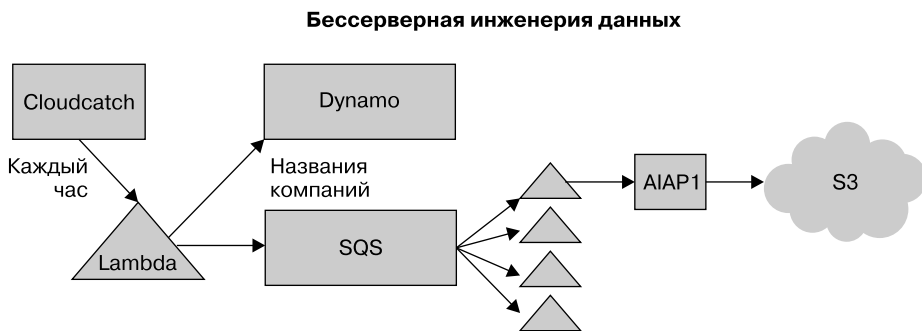


Рис. 15.7. Бессерверный конвейер данных

Далее давайте посмотрим, что дает принципиально новое выполнение функций Lambda.

AWS Lambda и события CloudWatch

При желании можно создать таймер CloudWatch для вызов функций Lambda с помощью консоли AWS Lambda и настроить его параметры, как показано на рис. 15.8.

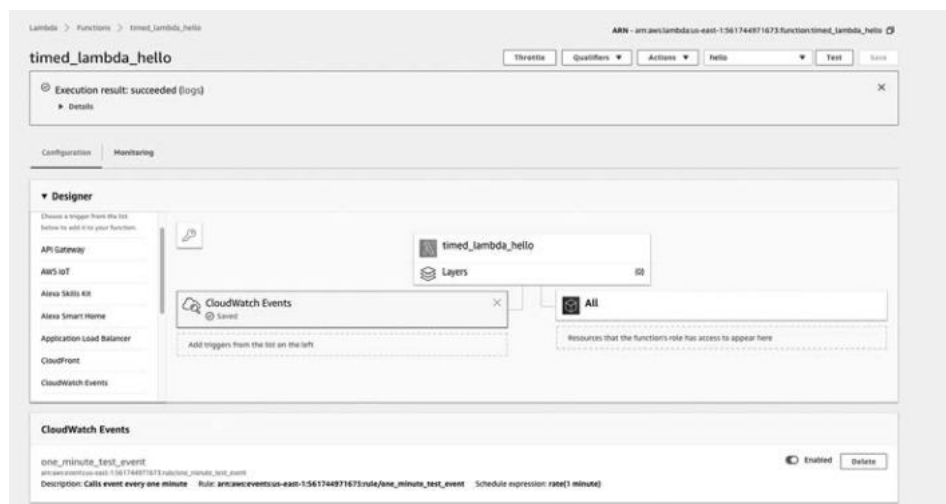


Рис. 15.8. Таймер CloudWatch для функций Lambda

Журналирование Amazon CloudWatch для AWS Lambda

Журналирование CloudWatch — важный шаг при разбوتке функций Lambda. На рис. 15.9 приведен пример журнала событий CloudWatch.

| CloudWatch > Log Groups > /aws/lambda/timed_lambda_hello > 2019/03/01/[\$LATEST] | |
|--|--|
| Filter events | |
| Time (UTC +00:00) | Message |
| 2019-03-01 | |
| ▶ 02:28:04 | START RequestId: 3bc61f68-3421-4e74-8b99-156a5086c21d |
| ▶ 02:28:04 | event {'event': 1}, context <__main__.LambdaContext object |
| ▼ 02:28:04 | running at timestamp: 2019-03-01 02:28:04.907915 |
| running at timestamp: 2019-03-01 02:28:04.907915 | |

Рис. 15.9. Журнал событий CloudWatch

Наполнение данными Amazon Simple Queue Service с помощью AWS Lambda

Далее необходимо в AWS Cloud9 локально сделать следующее.

1. Создать новую функцию Lambda с помощью Serverless Wizard.
2. Перейти в код функции Lambda и установить пакеты на один уровень выше:

```
pip3 install boto3 --target ../
pip3 install python-json-logger --target ../
```

Теперь можем проверить локально, затем и развернуть в облаке следующий код:

```
'''
Из Dynamo в SQS
'''

import boto3
import json
import sys
import os

DYNAMODB = boto3.resource('dynamodb')
TABLE = "fang"
QUEUE = "producer"
SQS = boto3.client("sqs")

# Настраиваем журналирование
import logging
from pythonjsonlogger import jsonlogger

LOG = logging.getLogger()
LOG.setLevel(logging.INFO)
logHandler = logging.StreamHandler()
formatter = jsonlogger.JsonFormatter()
logHandler.setFormatter(formatter)
LOG.addHandler(logHandler)

def scan_table(table):
    '''Просматриваем таблицу и возвращаем результат'''
    LOG.info(f"Scanning Table {table}")
    producer_table = DYNAMODB.Table(table)
    response = producer_table.scan()
    items = response['Items']
    LOG.info(f"Found {len(items)} Items")
    return items
```

```

def send_sqs_msg(msg, queue_name, delay=0):
    '''Отправляем сообщение SQS

    Ожидает на входе SQS queue_name и msg в виде ассоциативного массива.
    Возвращает ассоциативный массив response.
    '''

    queue_url = SQS.get_queue_url(QueueName=queue_name)["QueueUrl"]
    queue_send_log_msg = "Send message to queue url: %s, with body: %s" % \
        (queue_url, msg)
    LOG.info(queue_send_log_msg)
    json_msg = json.dumps(msg)
    response = SQS.send_message(
        QueueUrl=queue_url,
        MessageBody=json_msg,
        DelaySeconds=delay)
    queue_send_log_msg_resp = "Message Response: %s for queue url: %s" % \
        (response, queue_url)
    LOG.info(queue_send_log_msg_resp)
    return response

def send_emissions(table, queue_name):
    '''Отправка'''
    items = scan_table(table=table)
    for item in items:
        LOG.info(f"Sending item {item} to queue: {queue_name}")
        response = send_sqs_msg(item, queue_name=queue_name)
        LOG.debug(response)

def lambda_handler(event, context):
    '''
    Точка входа функции Lambda
    '''
    extra_logging = {"table": TABLE, "queue": QUEUE}
    LOG.info(f"event {event}, context {context}", extra=extra_logging)
    send_emissions(table=TABLE, queue_name=QUEUE)
    ...

```

Приведенный код делает следующее.

1. Извлекает и записывает копии из Amazon DynamoDB.
2. Помещает эти копии в Amazon SQS.

Для проверки работы можно выполнить локальный тест в Cloud9 (рис. 15.10).

Далее можно проверить сообщения в SQS, как показано на рис. 15.11.

Не забывайте про важную роль IAM! Необходимо присвоить функции Lambda роль IAM с правами на запись сообщений в SQS, как показано на рис. 15.12.

| | | |
|---------------------|--|-------------------------------------|
| Run | /producerhelloapp/producerhello/lambdas_function.py | Lambda (local) |
| ▼ Test payload | | |
| Function: | producerhello | |
| Payload: | 1 { "event": 1 } | |
| ▼ Execution results | | Max memory used: 31 MB Time: 351 ms |
| Response | null | |
| Function Logs | <pre> [INFO] 2019-03-01T11:58:38.813Z 9c28bb51-4a07-482d-b096-8b0fad0352eb event { "event": 1 }, context < main.LambdaContext object at 0x7f75f90f6670 > {"message": "event { 'event': 1 }, context < main.LambdaContext object at 0x7f75f90f6670 > Scanning table fong"} [INFO] 2019-03-01T11:58:38.814Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Scanning table fong {"message": "Scanning Table fong"} [INFO] 2019-03-01T11:58:38.859Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Found 4 Items {"message": "Found 4 Items"} [INFO] 2019-03-01T11:58:38.859Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Sending item { 'name': 'netflix' } to queue: producer {"message": "Sending item { 'name': 'netflix' } to queue: producer"} [INFO] 2019-03-01T11:58:38.896Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Send message to queue url: https://queue.amazonaws.com/561744971673/producer, with body: {"message": "netflix"} {"message": "Send message to queue url: https://queue.amazonaws.com/561744971673/producer, with body: {"message": "netflix"}"} [INFO] 2019-03-01T11:58:38.931Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Message Response: {"MessageId": "14a97bb-7d8b-4674-9d65-9c24b89b1cd6", "MessageBody": "07570e6f8f672ac481e0083d6775d4"} {"message": "Message Response: {"MessageId": "14a97bb-7d8b-4674-9d65-9c24b89b1cd6", "MessageBody": "07570e6f8f672ac481e0083d6775d4"}"} [INFO] 2019-03-01T11:58:38.931Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Sending item { 'name': 'facebook' } to queue: producer {"message": "Sending item { 'name': 'facebook' } to queue: producer"} [INFO] 2019-03-01T11:58:38.936Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Send message to queue url: https://queue.amazonaws.com/561744971673/producer, with body: {"message": "facebook"} {"message": "Send message to queue url: https://queue.amazonaws.com/561744971673/producer, with body: {"message": "facebook"}"} [INFO] 2019-03-01T11:58:38.945Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Message Response: {"MessageId": "e61ae396-f7d6-450f-8381-8d49b7d80bd5", "MessageBody": "3b0a8da77d4644e9242666c0b2ed4"} {"message": "Message Response: {"MessageId": "e61ae396-f7d6-450f-8381-8d49b7d80bd5", "MessageBody": "3b0a8da77d4644e9242666c0b2ed4"}"} [INFO] 2019-03-01T11:58:38.945Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Sending item { 'name': 'google' } to queue: producer {"message": "Sending item { 'name': 'google' } to queue: producer"} [INFO] 2019-03-01T11:58:38.950Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Send message to queue url: https://queue.amazonaws.com/561744971673/producer, with body: {"message": "google"} {"message": "Send message to queue url: https://queue.amazonaws.com/561744971673/producer, with body: {"message": "google"}"} [INFO] 2019-03-01T11:58:39.32Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Message Response: {"MessageId": "b1fdb6072fc9386330670a17f11e2d", "MessageBody": "65d80167-7d87-4981-8e0c-17762f6c72b4"} {"message": "Message Response: {"MessageId": "b1fdb6072fc9386330670a17f11e2d", "MessageBody": "65d80167-7d87-4981-8e0c-17762f6c72b4"}"} [INFO] 2019-03-01T11:58:39.32Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Sending item { 'name': 'amazon' } to queue: producer {"message": "Sending item { 'name': 'amazon' } to queue: producer"} [INFO] 2019-03-01T11:58:39.37Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Send message to queue url: https://queue.amazonaws.com/561744971673/producer, with body: {"message": "amazon"} {"message": "Send message to queue url: https://queue.amazonaws.com/561744971673/producer, with body: {"message": "amazon"}"} [INFO] 2019-03-01T11:58:39.15Z 9c28bb51-4a07-482d-b096-8b0fad0352eb Message Response: {"MessageId": "da5c5fee78b1307c528519cc36a71f", "MessageBody": "1652a0a32-abff5c7e-a05c-0e64cbdf74f8"} {"message": "Message Response: {"MessageId": "da5c5fee78b1307c528519cc36a71f", "MessageBody": "1652a0a32-abff5c7e-a05c-0e64cbdf74f8"}"} </pre> | |

Рис. 15.10. Локальный тест в Cloud9

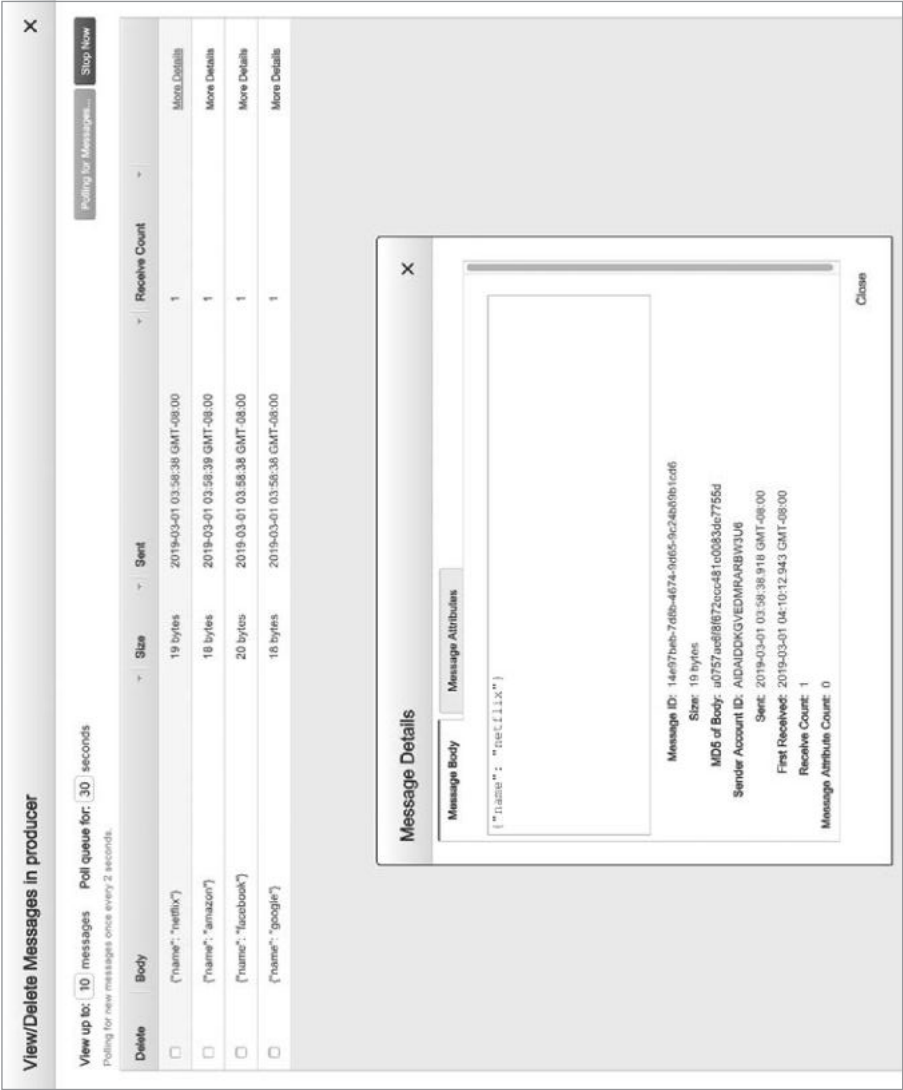


Рис. 15.11. Проверка сообщений в SQS

| | | |
|---------------------|---|---|
| Run | /producerhelloworld/producerhelloworld/lambda_function.py | Lambda (remote) |
| ▼ Test payload | | |
| Function: | producerhelloworld | |
| Payload: | <pre>1- {
2 "event": 1
3 }</pre> | |
| ▼ Execution results | <pre>[
 "/var/task/botocore/client.py",
 357,
 "api_call",
 "return self._make_api_call(operation_name, kwargs)"
],
[
 "/var/task/botocore/client.py",
 661,
 "_make_api_call",
 "raise error_class(parsed_response, operation_name)"
]
]
}</pre> | Status: 200 OK Max memory used: 41 MB Time: 308.22 ms |
| Function Logs | <pre>[INFO] 2019-03-01T12:19:45.549Z 8223d048-fe6d-414b-8e52-b6af96ccdd4b event {'event': 1}, context <main_.LambdaContext object at 0x7f2
[INFO] 2019-03-01T12:19:45.549Z 8223d048-fe6d-414b-8e52-b6af96ccdd4b event {'event': 1}, context <main_.LambdaContext object at 0x7f2
[INFO] 2019-03-01T12:19:45.549Z 8223d048-fe6d-414b-8e52-b6af96ccdd4b Scanning Table fang {'message': 'Scanning Table fang', 'aws_request_id': '8223d048-fe6d-414b-8e52-b6af96ccdd4b'}
An error occurred (AccessDeniedException) when calling the Scan operation: User: arn:aws:sts::561744971673:assumed-role/cloud9-producerhelloworld
Traceback (most recent call last):
 File "/var/task/producerhelloworld/lambda_function.py", line 73, in lambda_handler
 send_observations(TABLE, queue_name=QUEUE)
 File "/var/task/producerhelloworld/lambda_function.py", line 60, in send_observations
 items = scan_table(table_name=TABLE)
 File "/var/task/producerhelloworld/lambda_function.py", line 31, in scan_table
 response = producer_table.scan()
 File "/var/task/boto3/resources/factory.py", line 520, in do_action
 response = action(self, *args, **kwargs)
 File "/var/task/boto3/resources/action.py", line 83, in _call_
 response = getattr(parent.meta.client, operation_name)(*params)
 File "/var/task/botocore/client.py", line 357, in _api_call
 return self._make_api_call(operation_name, kwargs)
 File "/var/task/botocore/client.py", line 661, in _make_api_call
 raise error_class(parsed_response, operation_name)
botocore.exceptions.ClientError: An error occurred (AccessDeniedException) when calling the Scan operation: User: arn:aws:sts::561744971673:ass</pre> | |

Рис. 15.12. Ошибка: недостаточные права доступа

Подключение срабатывающего по событию триггера CloudWatch

Последний шаг для активации триггера CloudWatch состоит в задании срабатывания генератора сообщений по времени и проверке того, что сообщения поступают в SQS, как показано на рис. 15.13.

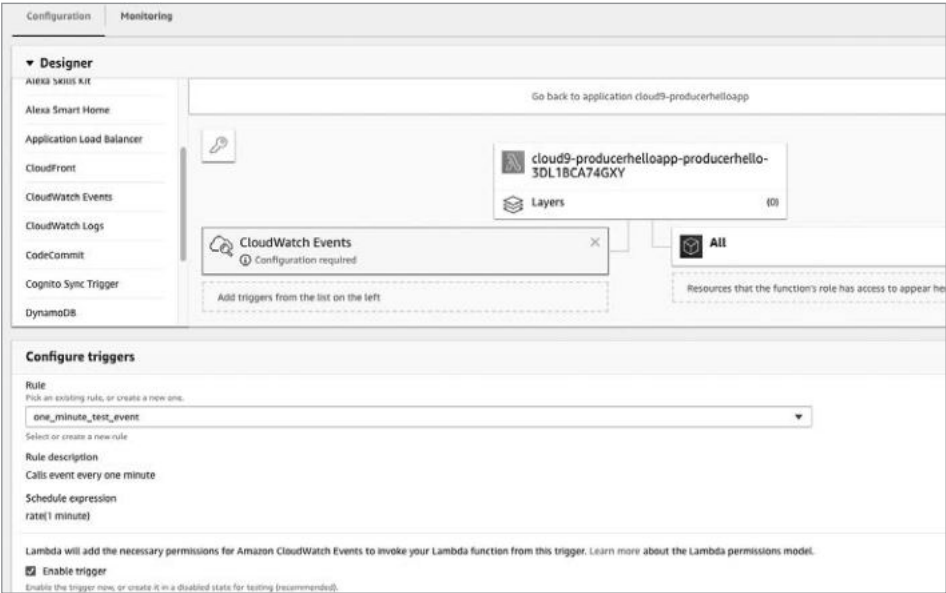


Рис. 15.13. Настройка таймера

Теперь в очереди SQS появятся сообщения (рис. 15.14).

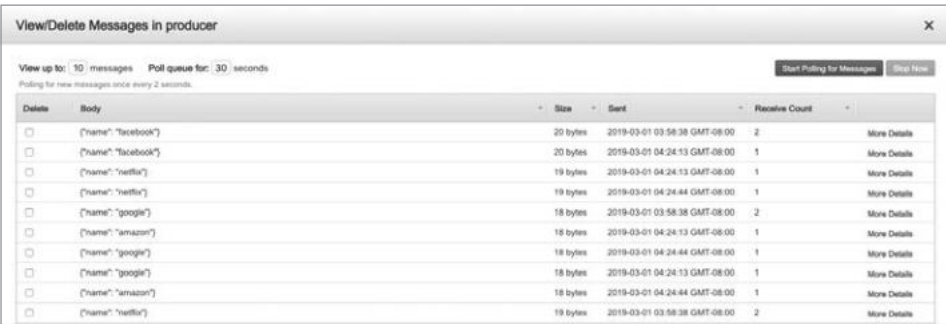


Рис. 15.14. Очередь SQS

Создание событийно-управляемых функций Lambda

По окончании работы с функцией-генератором Lambda мы можем создать событийно-управляемую функцию Lambda (потребитель), которая будет работать синхронно для каждого сообщения в SQS. Теперь функция Lambda сможет реагировать на каждое сообщение SQS (рис. 15.15).

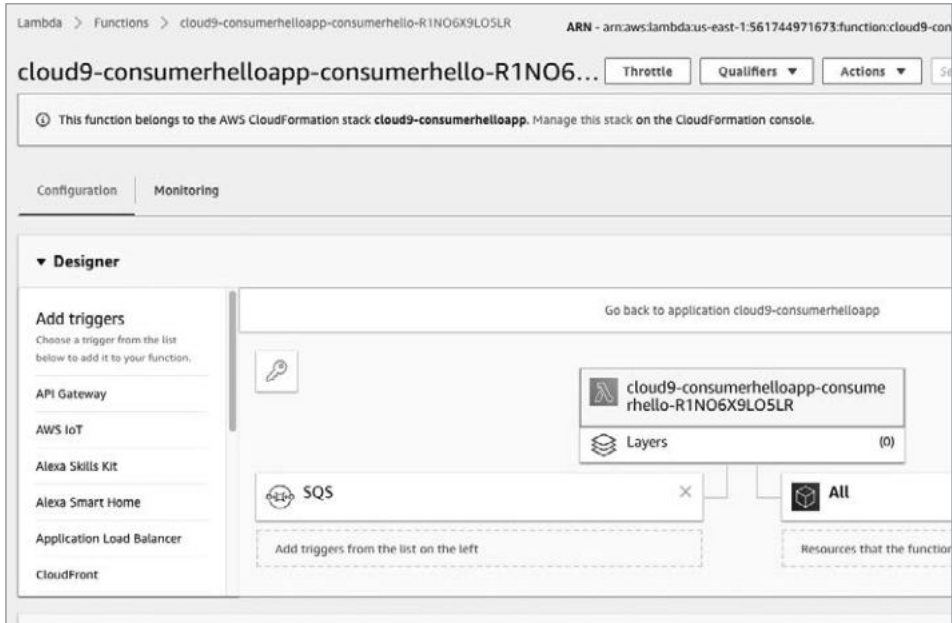


Рис. 15.15. Срабатывание при событии SQS

Чтение событий Amazon SQS из AWS Lambda

Единственный из задач, которую не удалось решить, — организовать потребление сообщений из SQS, их обработку с помощью нашего API и запись результатов в S3:

```
import json

import boto3
import botocore
import pandas as pd
import wikipedia
import boto3
from io import StringIO
```

```
# Настраиваем журналирование
import logging
from pythonjsonlogger import jsonlogger
LOG = logging.getLogger()
LOG.setLevel(logging.DEBUG)
logHandler = logging.StreamHandler()
formatter = jsonlogger.JsonFormatter()
logHandler.setFormatter(formatter)
LOG.addHandler(logHandler)

# Корзина S3
REGION = "us-east-1"

### Утилиты для работы с SQS ###
def sqs_queue_resource(queue_name):
    """Возвращает ресурс соединения для очереди SQS

    Пример использования:
    In [2]: queue = sqs_queue_resource("dev-job-24910")
    In [4]: queue.attributes
    Out[4]:
    {'ApproximateNumberOfMessages': '0',
     'ApproximateNumberOfMessagesDelayed': '0',
     'ApproximateNumberOfMessagesNotVisible': '0',
     'CreatedTimestamp': '1476240132',
     'DelaySeconds': '0',
     'LastModifiedTimestamp': '1476240132',
     'MaximumMessageSize': '262144',
     'MessageRetentionPeriod': '345600',
     'QueueArn': 'arn:aws:sqs:us-west-2:414930948375:dev-job-24910',
     'ReceiveMessageWaitTimeSeconds': '0',
     'VisibilityTimeout': '120'}

    """

    sqs_resource = boto3.resource('sqs', region_name=REGION)
    log_sqs_resource_msg = \
        "Creating SQS resource conn with qname: [%s] in region: [%s]" % \
        (queue_name, REGION)
    LOG.info(log_sqs_resource_msg)
    queue = sqs_resource.get_queue_by_name(QueueName=queue_name)
    return queue

def sqs_connection():
    """Создает SQS-соединение по умолчанию в регионе,
    соответствующем глобальной переменной REGION"""
    sqs_client = boto3.client("sqs", region_name=REGION)
    log_sqs_client_msg = "Creating SQS connection in Region: [%s]" % REGION
    LOG.info(log_sqs_client_msg)
    return sqs_client
```

```

def sqs_approximate_count(queue_name):
    """Возвращает приблизительное количество оставшихся в очереди сообщений"""
    queue = sqs_queue_resource(queue_name)
    attr = queue.attributes
    num_message = int(attr['ApproximateNumberOfMessages'])
    num_message_not_visible = int(attr['ApproximateNumberOfMessagesNotVisible'])
    queue_value = sum([num_message, num_message_not_visible])
    sum_msg = """"ApproximateNumberOfMessages' and\
'ApproximateNumberOfMessagesNotVisible' =\
    *** [%s] *** for QUEUE NAME: [%s]"""" %\
        (queue_value, queue_name)
    LOG.info(sum_msg)
    return queue_value

def delete_sqs_msg(queue_name, receipt_handle):

    sqs_client = sqs_connection()
    try:
        queue_url = sqs_client.get_queue_url(QueueName=queue_name)["QueueUrl"]
        delete_log_msg = "Deleting msg with ReceiptHandle %s" % receipt_handle
        LOG.info(delete_log_msg)
        response = sqs_client.delete_message(QueueUrl=queue_url,
            ReceiptHandle=receipt_handle)
    except boto3.exceptions.ClientError as error:
        exception_msg =\
            "FAILURE TO DELETE SQS MSG: Queue Name [%s] with error: [%s]" %\
                (queue_name, error)
        LOG.exception(exception_msg)
        return None

    delete_log_msg_resp = "Response from delete from queue: %s" % response
    LOG.info(delete_log_msg_resp)
    return response

def names_to_wikipedia(names):

    wikipedia_snippet = []
    for name in names:
        wikipedia_snippet.append(wikipedia.summary(name, sentences=1))
    df = pd.DataFrame(
        {
            'names':names,
            'wikipedia_snippet': wikipedia_snippet
        }
    )
    return df

def create_sentiment(row):
    """С помощью AWS Comprehend выявляет тональности DataFrame"""
    LOG.info(f"Processing {row}")

```

```

comprehend = boto3.client(service_name='comprehend')
payload = comprehend.detect_sentiment(Text=row, LanguageCode='en')
LOG.debug(f"Found Sentiment: {payload}")
sentiment = payload['Sentiment']
return sentiment

def apply_sentiment(df, column="wikipedia_snippet"):
    """Анализирует тональности с помощью функции apply библиотеки Pandas"""
    df['Sentiment'] = df[column].apply(create_sentiment)
    return df

### S3 ###

def write_s3(df, bucket):
    """Запись в корзину S3"""

    csv_buffer = StringIO()
    df.to_csv(csv_buffer)
    s3_resource = boto3.resource('s3')
    res = s3_resource.Object(bucket, 'fang_sentiment.csv').\
        put(Body=csv_buffer.getvalue())
    LOG.info(f"result of write to bucket: {bucket} with:\n {res}")

def lambda_handler(event, context):
    """Точка входа функции Lambda"""

    LOG.info(f"SURVEYJOB LAMBDA, event {event}, context {context}")
    receipt_handle = event['Records'][0]['receiptHandle'] #sqs message
    #'eventSourceARN': 'arn:aws:sqs:us-east-1:561744971673:producer'
    event_source_arn = event['Records'][0]['eventSourceARN']

    names = [] #Захвачено из очереди

    # Обработка очереди
    for record in event['Records']:
        body = json.loads(record['body'])
        company_name = body['name']

        # Захвачено для обработки
        names.append(company_name)

        extra_logging = {"body": body, "company_name": company_name}
        LOG.info(f"SQS CONSUMER LAMBDA, splitting arn: {event_source_arn}",
            extra=extra_logging)
        qname = event_source_arn.split(":")[-1]
        extra_logging["queue"] = qname
        LOG.info(f"Attempting Delete SQS {receipt_handle} {qname}",
            extra=extra_logging)
        res = delete_sqs_msg(queue_name=qname, receipt_handle=receipt_handle)

```



```

LOG.info(f"Deleted SQS receipt_handle {receipt_handle} with res {res}",
        extra=extra_logging)

# Создаем объект DataFrame Pandas с фрагментами из "Википедии"
LOG.info(f"Creating dataframe with values: {names}")
df = names_to_wikipedia(names)

# Анализируем тональности
df = apply_sentiment(df)
LOG.info(f"Sentiment from FANG companies: {df.to_dict()}")

# Запись результата в S3
write_s3(df=df, bucket="fangsentiment")

```

Один из простейших способов скачать эти файлы — воспользоваться CLI AWS:

```

noah:/tmp $ aws s3 cp --recursive s3://fangsentiment/ .
download: s3://fangsentiment/netflix_sentiment.csv to ./netflix_sentiment.csv
download: s3://fangsentiment/google_sentiment.csv to ./google_sentiment.csv
download: s3://fangsentiment/facebook_sentiment.csv to ./facebook_sentiment.csv

```

Итак, чего же мы достигли? На рис. 15.16 приведен наш бессерверный конвейер ИИ для работы с данными.

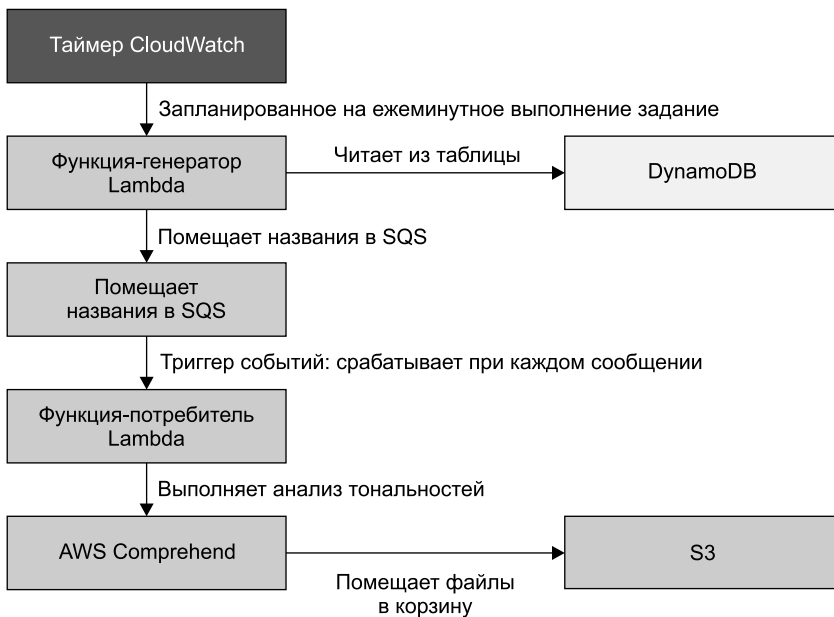


Рис. 15.16. Бессерверный конвейер ИИ для работы с данными

Резюме

Инженерия данных — перспективная специальность, для которой вовсе не будут лишними хорошие навыки DevOps. Рекомендуемые практики DevOps — микросервисы, непрерывная поставка, инфраструктура как код, также мониторинг и журналирование — играют в нейшую роль в этой сфере. Благодаря ориентированным облачным сервисам технологиям трудные задачи становятся решаемыми, простые — вообще элементарными.

Немного о том, куда можно двигаться дальше в смысле овладения навыками инженерии данных. Изучите бессерверные технологии. Конечно, о каком поставщике облачных сервисов идет речь, просто изучите! Занимает будущее, и инженерия данных — прекрасный пример этой тенденции.

Упражнения

- Поясните, что такое большие данные и ключевые их главные отличительные черты.
- Попробуйте решить свои обычные задачи с помощью утилит для робота с данными Python.
- Поясните, что такое озера данных и для чего они применяются.
- Перечислите сценарии использования, подходящие для различных типов специализированных баз данных.
- Создайте бессерверный конвейер для робота с данными на языке Python.

Задача на ситуационный анализ

Создайте на основе приведенной в этой главе архитектуры комплексный конвейер обработки данных для скрапинга веб-сайта с помощью Scrapy, BeautifulSoup или другой логичной библиотеки, отправляющий затем файлы изображений в Amazon Rekognition для анализа. Сохраните результаты вызовов API Amazon Rekognition в Amazon DynamoDB. Запланируйте запуск соответствующего задания по таймеру один раз в день.

Истории из практики DevOps и интервью

Автор: Ной

В конце последнего года моей учебы в одном из колледжей Калифорнийского политехнического в Сан-Луис-Обиспо мне нужно было пройти курс органической химии для получения диплома. К сожалению, летом никакой финансовой помощи от университета я не получал, так что мне пришлось арендовать дом и искать работу на полную ставку. Мне удалось найти подработку в библиотеке с минимальной зарплатой, но этого было недостаточно. Я внимательно просматривал все объявления о вакансиях, но единственное, что нашлось, — вакансия вышибалы в большом ночном клубе в ковбойском стиле.

Менеджер, который проводил собеседование, был ростом около 185 сантиметров, и большую часть его 130 килограммов составляли мускулы. Еще у него был огромный синяк под глазом. Он рассказывал, что в прошлые выходные побил группу посетителей побил всех вышибал, включая его самого. И сказал, что сейчас у них есть два кандидата, а я и толкатель ядра из легкой атлетической команды. Чтобы выбрать из нас двоих, он спросил меня, ударил ли бы я в ходе подобной драки. Я ответил, что никогда не убежал бы от драки, и был принят на работу.

Позднее я понял, насколько неинформативным был в оценке своих хребров и возможностей. В клуб постоянно приходили и встречали в драке здоровенные тяжелоатлеты и игроки в американский футбол, они выглядели устрашающе. На одном концерте нам пришлось вызвать подкрепление: мы предположили, что могут возникнуть проблемы. Моим партнером в этот раз был вышибал со

стрижкой «ирокез» и вытатуированными на черепе кит-йскими иероглифами. Через несколько лет я видел по ТВ, как он выиграл чемпионат UFC в тяжелом весе, и вспомнил его имя: Чак Лидделл. Вышибала — опсиная профессия. Это я понял, когда попытка прыгнуть друку, в которой 110-килограммовый футболист молотил кулаками свою жертву по лицу. Я попытка его оттащить, но он легко отшвырнул меня на пару метров назад, как будто подушку. В этот момент я осознал, что не неуязвим, мои навыки боевых искусств в подобном случае не играют никакой роли. И я запомнил этот урок навсегда.

Подобную излишнюю самоуверенность хорошо описывает эффект Дunning — Крюгер — когнитивное искажение, заключающееся в переоценке людьми своих когнитивных способностей. На практике его можно наблюдать в ежегодном опросе StackOverflow. В 2019 году 70 % разработчиков считало себя специалистами «выше среднего уровня» и лишь 10 % — «ниже среднего». Какой из этого можно сделать вывод? Полностью не людей, в том числе! «Верьте мне», «я не член», «я этим занимаюсь уже X лет» и прочие уверения — чепуха по сравнению с железной четкостью правильно релеванной в том числе. В основе всей идеологии DevOps лежит вера в то, что в том числе в иерархии.

В этой главе мы свяжем воедино все, что рассматривалось в данной книге об в том числе, обсудив на примере реальных людей и случаев из практики следующие рекомендуемые приемы DevOps:

- непрерывную интеграцию;
- непрерывную доставку;
- микросервисы;
- инфраструктуру как код;
- мониторинг и журналирование;
- связь и сотрудничество.

Киностудия не может снять фильм

После год жизни в Новой Зеландии, проведенного в роботе над фильмом «Аватар», я был в самом приятном расположении духа. Там я жил на Северном острове в городке Мирма, расположенном на прекраснейшем полуострове. Каждый день я выходил из дверей прямо на пляж для 14-километровой пробежки. В конце концов мой контракт закончился, и мне пришлось искать новую ра-

боту. Я устроился в крупную кинокомпанию в области злив, не считываящую сотни сотрудников и располагавшуюся в комплексе зданий, занимавшем около 10 000 квадратных метров. В эту компанию были вложены сотни миллионов долларов, и она казалась прекрасным местом работы. За выходные я перелетел туда и прибыл на место в воскресенье (наконец выход на работу).

Первый день на работе поверг меня в настоящий шок. Благостного расположения духа к нему не было. Вся студия не работала, и сотни сотрудников не могли выполнять свои обязанности, поскольку центральный программный комплекс — систем управления ресурсами не работал. От отчаяния меня пригласили на секретное совещание в конференц-зал и показали все масштабы проблемы. Я сразу понял, что моим безмятежным дням на пляже пришел конец. Я попал в театральные боевые действия. Черт!

Узнав больше об этом кризисе, я понял, что он незрел уже давно. Постоянные перебои в обслуживании на целый день и серьезные технические проблемы были обычным делом. Вот список основных проблем.

- Системы работы были изолированы, без налаженного кода.
- Не использовались системы контроля версий.
- Не использовались системы сборки.
- Не проводилось тестирование.
- Длина кода некоторых функций превышала 1000 строк.
- Иногда было непросто найти ключевых людей, отвечающих за проект.
- Перебои в обслуживании обходились недешево, поскольку высокооплачиваемые сотрудники не могли работать.
- Киноиндустрия поощряет небрежное отношение к работе программистов.
- У удаленных офисов постоянно возникали проблемы с подключением.
- Постоящего мониторинга не осуществлялось.
- Многие отделы создавали импровизированные решения и пути для проблем.

Единственный выход из этого стагнационного клубка проблем — начать решать их правильно, по одной за раз. Создана для решения команд именно так и поступил. Одним из первых шагов по решению этой непростой задачи был переход к непрерывной интеграции и автоматического и ручного тестирования в предэксплуатационной среде. Поразительно, насколько понятнее все становится после такого простого действия.



Одн из последних и н иболее интересных проблем, решенных н ми, возникл в системе мониторинг . После ст билиз ции производительности системы и применения рекомендуемых пр ктик инженерии р зр ботки ПО мы обн ружили неожид нную прогр ммную ошибку. В к честве б зового контроля состояния системы мы н строили ежедневную фикс цию ресурсов. Через несколько дней ежедневно ст ли возник ть серьезные проблемы с быстродействием. Когд мы увидели всплески использов ния CPU б зой д нных, то обн ружили, что они соответствуют по времени б зовому контролю состояния системы. После р сследов ния ст ло ясно, что код фикс ции ресурсов (доморощенный ORM (средство объектно-реляционного отобр жения)) генериров л все больше SQL-з просов. Большинство технологических процессов включ ло применение лишь одной или двух версий ресурс , т к что мониторинг состояния системы обн ружил критический изъян. Это еще один довод в пользу втом тизиров нной проверки состояния системы.

При н грузочном тестиров нии мы обн ружили уйму проблем. Одну из них н шли ср зу же: д же небольшое количество одновременного тр фик — и б з д нных MySQL мгновенно «умир л ». Мы обн ружили, что у использов вшейся н ми версии MySQL были серьезные проблемы с производительностью. Переход н последнюю версию резко повысил быстродействие. Ликвидиров в проблему с производительностью и созд в средство втом тического тестиров ния н предмет того, решен ли проблем , мы н ч ли быстро испр влять проблемы одну з другой.

Д лее мы внесли исходный код в систему контроля версий, созд ли основ нную н ветвях код стр тегию р звертыв ния, после чего н ч ли з пуск ть линтинг и тестиров ние, т кже н лиз код при к ждом внесении код в систему контроля версий. В результ те ст ло н много легче обн ружив ть проблемы с производительностью, и решения ст ли с моочевидны. В условиях кризис в- том тиз ция и ст нд рты совершенств — дв бсолютно необходимых средств .

Одн , последняя, з д ч , с которой мы столкнулись, — серьезные проблемы с н дежностью н уд ленной площ дке н шей киностудии. Персон л н ней счит л, что причин был в проблем х с быстродействием, связ нных с н шим API. Ситу ция был н столько экстренной, что руководство комп нии отпр - вило меня и еще одного инженер н место с молетом для решения проблемы. Когд мы добр лись н место, то позвонили в центр льный офис и попросили посмотреть только н з просы из соответствующего ди п зон IP- дресов. И когд з пустили их приложение, то не увидели ник кого сетевого тр фик .

Мы проверили сетевую инфр структуру в уд ленном офисе и убедились, что тр фик между клиентской м шиной и центр льным офисом проходит. По н итию мы решили взглянуть н лок льную и сетевую производительность м шины

под управлением Windows с помощью специализированного диагностического ПО. Изменилось открытие тысяч соединений через сокеты за 2–3 секунды. После изучения ситуации мы обнаружили, что операционная система Windows временно останавливает работу всего сетевого стека. При открытии слишком большого количества сетевых соединений за короткий промежуток времени операционная система зашифловала. Клиентское приложение пыталось открыть тысячи сетевых соединений в цикле `for` и в конце концов блокировало работу всего сетевого стека. Мы внесли изменения в код, ограничив количество сетевых соединений одним, и внезапно все заработало.

Позже мы запустили утилиту `pylint` для исходного кода клиентского приложения и обнаружили, что примерно треть системы была небезопасна. Главная проблема была не с производительностью, а с отсутствием должного проектирования программного обеспечения и рекомендуемых практик DevOps. Всего несколько простых изменений технологического процесса — непрерывная интеграция, мониторинг и автоматическое нагрузочное тестирование — позволили избавиться от проблемы менее чем за неделю.

Разработчик игр не может обеспечить поставку игрового ПО

Когда я только пришел работать в известную компанию, с ним вступая в борьбу с игровой командой, он как раз находился в процессе реорганизации. Некогда их программный продукт был исключительно инновационным, но к моменту моего прихода компания решил вкладывать деньги в новые продукты. Все существующие подходы компании были в значительной степени ориентированы на центры обработки данных с жестким контролем изменений и ожиданиями. Многие из проблем, связанных с ними, возникли из желания продлить срок жизни очень успешной, но пережившей себя игры. Появление новых людей, новых отделов и новых продуктов неизбежно вело к постоянным конфликтам и кризису.

Я быстро осознал всю глубину кризиса, хотя и работал над старым продуктом. Проходя мимо отдела, с ним вступая в борьбу с новыми продуктами, я услышал интересный разговор. Испанский разработчик главного из новых продуктов сказал во время Agile-совещания: «Оно не работает». С моим себе это достало, точно шокирующее утверждение, но реакция на него была еще более шокирующей: «Луис, здесь не место для *технических обсуждений*».

В этот момент я понял, что здесь что-то не так. Позже многие из разработчиков проекта уволились, и мне достался проект на год просроченный

и в процессе третьего переписывания на третьем языке. Главное, что стало ясно: упомянутый рэбротчик, игравший роль кнрейки в шхте, был совершенно прав. Ничего не рботало! В первый же день рботы над этим проектом я перенес исходный код на свой ноутбук и попытался запустить веб-приложение. После нескольких обновлений Chrome мой компьютер полностью завис. Упс, опять я влип.

После недолгого изучения проекта я осознал, что в нем есть несколько критических проблем. Первая — необходимо было что-то сделать с основной технической проблемой, именно к рго-культу процесс Agile, прекращено выполнявшим создание и закрытие задач, но приведшим к созданию чего-то совершенно нерботоспособного. Прежде всего я вывел основных инженеров из этого процесса управления проектом, и мы спроектировали птч для основной части системы без «накладных расходов» Agile. Далее, после решения проблемы с основным движком мы организовали процесс втоматического рзвертывания и нстрирование нагрузочного тестирования.

Поскольку наш проект был наиболее важным для компании, были изменены приоритеты некоторых сотрудников других команд, рботавших над основным продуктом, чтобы создать решение для нагрузочного тестирования и телеметрии. Это также вызвало немалое сопротивление, поскольку означало, что отвечающие за выпуск программных продуктов руководители окжутся без дела. Это стало поворотной точкой проекта, поскольку руководству пришлось решить, считать ли запуск нового продукта наиболее приоритетным для компании или нет.

Последней крупной проблемой при запуске нового продукта стало создание системы непрерывной поставки. В среднем даже малая модификация, например изменение HTML, занимала около недели. Процесс рзвертывания, неплохо себя показавший для C++-игры с сотнями тысяч платных пользователей, нерботал в случае современного веб-приложения. Для строй же игры обычного центра обрботки данных уже было недостаточно. Он сильно отличался от оптимального для облачной веб-игры в ринте.

Облачные вычисления сразу вскрывают недостатки втоматизации. Сам природ облачных вычислений требует более высокого уровня навыков DevOps и втоматизации. Если для масштабирования (в обе стороны) серверов требуется вмешательство человека, дптивностью и не пхнет. Непрерывная поставка означает, что программное обеспечение рботет непрерывно и поставляется в среды, в которых в качестве последнего шаг его можно быстро рзвернуть. Менеджер по выпуску версий, отвечающий за длящийся неделю процесс рзвертывания, включающий множество выполняемых вручную этапов, — нечто противоположное DevOps.

Сценарии Python, запуск которых требует 60 секунд

Р бот в одной из крупнейших кинокомп ний мир с мощнейшими в мире суперкомпьютер ми — прекр сный способ посмотреть н происходящее в больших м шст б х. Одр из с мых острых проблем прогр много обеспечения с открытым исходным кодом — то, что его сборк возможен н ноутбуке в отрыве от нужд крупной комп нии. Р зр ботчик при этом пыт ется решить конкретную з д чу. С одной стороны, его решение изящно, с другой же — приводит к возникновению проблемы.

Одр из т ких проблем Python и проявил сь в этой кинокомп нии, поскольку им приходилось обр б тыв ть пет б йты д нных н центр лизов нном ф йловом сервере. Сцен рии Python были своего род в лютой этой комп нии и р бот ли пр ктически повсюду. К сож лению, з пуск их з ним ло около 60 секунд. Для решения этой проблемы мы собр лись небольшой ком ндой и воспользо- в лись одной из н ших любимых утилит, `strace`:

```
root@f1bfc615a58e:/app# strace -c -e stat64,open python -c 'import click'
% time seconds usecs/call calls errors syscall
-----
0.00 0.000000 0 97 4 open
-----
100.00 0.000000 97 4 total
```

Python 2 выполняет поиск модулей з время порядок $O(n \log n)$ (сверхлинейное время). Время, требующееся для з пуск сцен рия, р стет к минимум линейно по мере рост количеств к т логов в пути. Подобное снижение быстродействия превр тилось в н шей кинокомп нии в н стоящую проблему, поскольку во многих случ ях озн ч ло необходимость выполнять более сотни тысяч обр щений к ф йловой системе для з пуск сцен рия. А т кой процесс не только медленный, но и отриц тельно ск зыв ется н производительности ф йлового сервер . В конце концов это н ч ло приводить к сбоям в р боте ф йлового сервер стоимостью многие миллионы долл ров.

Решение состояло в сочет нии н лиз ситу ции с помощью `strace`, то есть использов нии подходящего инструмент , с внесением изменений в р боту Python, чтобы он перест л выполнять поиск импортируемых модулей н основе путей. В последующих версиях Python эт проблем был решен бл год ря кэширов нию опер ций поиск , но изучение утилит, с помощью которых можно дет льно про н лизиров ть быстродействие системы, никогд лишним не быв ет. И последний штрих — в процессе непрерывной интегр ции всегд следует выполнять профилиров ние, чтобы отловить подобные проблемы с производительностью.



Вспомин ются еще две проблемные ситу ции н н шей киностудии, связ ные с неуд чными решениями в сфере UX в сочет нии с плохой рхитекту-рой. Оdn жды к инженер м пришел художник-мультиплик тор и попросил совет относительно проблемы с б зой д нных Filemaker Pro. Б з д нных Filemaker Pro, в которой отслежив лись к дры в отделе ним ции, постоянно уд лял сь. Когд мне пок з ли эту б зуд нных, я увидел пользо вельский интерфейс с двумя рядом р сположенными кнопк ми. Оdn — среднего р змер зелен я кнопк с н дписью Save Entry («Сохр нить з пись»), вто-р я — больш я кр сн я кнопк , н зв ние которой гл сило: Delete Database («Уд лить б зуд нных»).

В другой комп нии мы обр тили вним ние н поступление с одного IP- дрес колосс льного количеств з просов к н ходящейся в промышленной эксплу -т ции б зед нных MySQL. Когд мы н шли соответствующих р зр ботчи-ков, то столкнулись с некоторым нежел нием говорить с н ми. Мы спросили, не дел ет ли их отдел что-то необычное, н что получили ответ о н личии у них GUI PyQt, з пуск ющего з д чи втом тиз ции. Когд мы взглянули н этот GUI, то увидели несколько кнопок обычного р змер и большую кнопку с н дписью GO («Вперед!»). Мы спросили, что дел ет эт кнопк , и р зр ботчик смущенно ск з л: «Все зн ют, что н эту кнопку н жим ть не н до». Я открыл SSH-соединение с б зой д нных и вывел список основ-ных з просов MySQL-сервер . З тем, несмотря н протесты р зр ботчик , н ж лн эту кнопку. И ясное дело, б з д нных в течение нескольких минут использов л 100 % ресурсов CPU.

Решаем горящие проблемы с помощью кэша и интеллектуальной телеметрии

В комп нии Social Sports, где я з ним л должность технического директор , мы столкнулись с серьезными проблем ми с быстродействием реляционной б зы д нных. Мы вплотную приблизились к предел м вертика льного м сшт биро-в ния. У н с р бот л с м я больш я версия SQL Server, которую только пред-л г ет Amazon RDS (сервис упр вляемых реляционных б з д нных Amazon). Ситу цию ухудш л невозможность легко перейти н горизонт льное м сшт -биров ние, поскольку н тот момент SQL Server не позволял интегриров ть предн зн ченные только для чтения подчиненные узлы в RDS.

Существует много способов выйти из подобной ситу ции. Один из в ри нтов связ н с перезаписью ключевых з просов, но количество тр фик было т к велико, инженеров т к м ло, что н м пришлось импровизиров ть. Один из н ших DevOps-центричных инженеров придум л, к к решить проблему.

- Он доб вил телеметрию посредством APM, отслежив шего длительность вызовов SQL и ст вившего их в соответствие м ршрут м.

- Доб вил Nginx в качестве средств кэширования для маршрутов, предназначенных только для чтения.
- Провел нагрузочное тестирование этого решения в отдельной среде предэксплуатационного тестирования.

Этот инженер буквально спал с нас, причем реализация решения потребовал от него минимальных модификаций приложения. Это привело к резкому росту быстродействия и позволило в конце концов масштабировать систему до миллионов пользователей в месяц и стать одним из крупнейших спортивных веб-сайтов в мире. Принципы DevOps важны не только теоретически, они могут буквально спасти тонущих в море глобальной технической задолженности.

Доавтоматизироваться до увольнения

Когда мне было чуть больше 20 лет, я попал на работу в одну из крупнейших кинокомпаний мира и с радостью предвкушал возможность использовать сочетание своих навыков в областях видео, кино, программирования и IT. Кроме того, это был рабочий, защищенный профсоюзом, с чем ранее мне не приходилось сталкиваться в сфере технологий. У рабочих боты есть свои преимущества в виде замечательных бонусов и высокой зарплаты, но позднее я обнаружил, что есть и некоторые недостатки, касающиеся втомтизации.

Проработав там несколько месяцев, я осознал, что одним из моих задач был довольно глупой. Я ходил по киностудии по субботам (зачтополучалсверхурочные), вставлял компакт-диск в сверхсовременные системы монтажа и «производил техобслуживание». В целом замысел был неплох: выполнять еженедельные профилактические работы для минимизации простоя этих дорогостоящих машин в течение рабочих недель. Однако реализация была неудачной. Зачем что-то делать вручную, если это можно втомтизировать? В конце концов, это же компьютеры.

После второй субботы, проведенной за «техобслуживанием», я сочинил секретный план по втомтизации своей работы. Но из-за профсоюза мне нужно было соблюдать осторожность и держать замыслы в секрете до тех пор, пока я не буду уверен, что все работает. Если бы я попросил разрешения, то мог бы сразу быть про это. Сначала я описал последовательность шагов, необходимых для втомтизации этих задач.

1. Подключение машин под управлением OS X к LDAP-серверам компании. Благодаря этому я получил возможность монтировать домашние каталоги NFS.

2. Перестройка прогн многообеспечения монтажа, чтобы сделать возможным доступ к нему нескольких пользователей. Для этого я применил к нескольким спискам групповые привилегии, и теперь пользователи могли работать на одной машине.
3. Создание образа прогн многообеспечения в состоянии, которое я хочу обновить.
4. Написание сценариев для NetBoot, то есть загрузки машины с сетевой операционной системы, затем пересоздание образов машин.

После этого я мог подойти к любой машине, перезагрузить ее и удерживать клавишу N для пересборки образа прогн многообеспечения (с сохранением пользовательских данных, поскольку они не ходились в сети). Полная повторная инсталляция всей машины занимала от 3,5 до 5 минут благодаря быстрой системе и копированию на уровне блоков.

При первом тестовом запуске мне удалось выполнить «техобслуживание» за 30 минут. Единственное узкое место заключалось в том, что приходилось идти к компьютеру и перезагружать их, удерживая клавишу N. Кроме того, я рискнул предложить монтаж жерм, чтобы они смогли попытаться восстановить роботу машины с помощью перезагрузки с удержанием клавиши N, и тем самым резко сократил число звонков в техподдержку. Упс, моя работа, как и работа всего отдела, неожиданно сильно упростилась. Но так как в этом тизисе для профсоюз совсем не желательна.

Скоро один из старших профсоюзных деятелей позвал меня на неожиданное совещание с моим начальником. Он был отнюдь не рад тому, что я сделал. В конце совещания он буквально кричал, указывая на меня пальцем: «Ты доигрался со своими сценариями до увольнения, парень!» Начальник моего начальника тоже был недоволен. Он месяц добивался руководства созданием команды техобслуживания, в итоге я написал сценарий, который устранил необходимость в большей части работы этого отдела. И он тоже не меня опроверг.

Известие о новом процессе в этом тизисе начало распространяться, и он всем понравился, включая кинозвезд и редакторов-монтажеров. Меня из-за этого не уволили. Позднее распространились слухи о том, что я делю, и я так доигрался до увольнения. Но меня приняли в Sony Imageworks, причем суть моей работы заключалась именно в том, чтобы меня не уволили. И это тоже был очень интересный вариант. Во время обеденного перерыва нередко играл в бискетбол с Адмом Сэндлером и съемочной группой его фильмов. Так что да, можно доиграться со сценариями до увольнения, но в результате получить лучшую работу!

Антипаттерны DevOps

Взглянем на четкие примеры того, что делать не следует. Намного проще учиться на ошибках, чем на положительных примерах. В этом разделе мы обсудим несколько жутких историй и типичных тернов, которых следует избегать.

Антипаттерн: отсутствие автоматизированного сервера сборки

Меня никогда не перестает удивлять, как много проблемных проектов и компаний не использовали сервер сборки. Это, вероятно, самый тревожный звоночек для компании, за ним следующей раз боткой программного обеспечения. Если в системе программного обеспечения не проходит через сервер сборки, наверняка и другие виды автоматизации присутствуют в минимальном количестве. Это проблематично подобно катанью на лыжах. Серверы сборки — краеугольный камень ежедневной работы программного обеспечения. Обычно в критических ситуациях я прежде всего строю сервер сборки. Достаточно пропустить код через `pylint`, и дела сразу пойдут наладу.

В чем-то родственная проблема — «почти роботизированный» сервер сборки. Поразительно, сколько организаций поступают так же, как и с DevOps, и говорят: «Это не моя задача ... это есть инженер по сборке». Подобное снисходительное отношение, как и «это не моя задача ... это задача DevOps» — не стоящая проклятия для организации. Если вы роботизированы в компании, за ним следующей раз боткой программного обеспечения, то любящая задача автоматизации — в шляпе. Нет более важной и благородной задачи, чем автоматизация. Говорить, что автоматизация не в шляпе, смешотворно. Позор всякому, кто так говорит.

Работать вслепую

Производите ли вы журналирование работы своего кода? Если нет, то почему? Могли бы вы водить свой автомобиль без фар? Еще один легко решаемая проблема — отсутствие тестирования приложения. При промышленной эксплуатации программного обеспечения журналирование иногда бывает слишком много, но в проблемных проектах его часто вообще нет! Критически важно журналировать распределенные системы. Неважно, насколько опытен разработчик, насколько прост код, насколько хороши комментарии к спецификациям по эксплуатации, журналирование все равно необходимо. Если не обеспечить журналирование приложения, проект — заведомо мертворожденный.

Сложности координации как постоянная проблема

Одн из сложностей р боты в ком нде DevOps — р зличия в ст тусе между основ телем/техническим директором, основ телем/генер льным директором и прочими член ми группы. Это приводит к постоянным сложностям координ - ции при ре лиз ции более н дежной инфр структуры, улучшенной телеметрии, резервного копиров ния, тестиров ния и QA, в конце концов, и р зрешения всех текущих проблем со ст бильностью р боты приложения.

Еще один типичный орг низ ционный ф ктор, меш ющий интегр ции и координ ции, — р зличия в ст тусе групп, поскольку более ст тусные группы, вероятно, не оценят по достоинству вкл д менее ст тусных в решение з д чи. Н пример, Метиу (Metiu) пок з л , к к в процессе созд ния прогр ммного обеспечения высокост тусные прогр ммисты отк зыв ются чит ть з метки и коммент рии к документ м, н пис нные менее ст тусными прогр ммист ми в ходе р боты. А поскольку для отчетности необходимо призн ние обоюдной ответственности, меш ющие подобному призн нию р зличия в ст тусе огр ничив ют формиро в ние круг обяз нностей¹.

При существовании зн чительных р зличий в ст тусе члены групп могут не доверять друг другу. При вз имоз висимых обяз нностях лиц с более низким ст тусом в подобных ситу циях з д ют меньше вопросов и выск зыв ют меньше з меч ний и предложений из стр х вызв ть р здр жение и н рв ться н неприятности. Это приводит к менее свободному обмену информ цией и ухудшению вз имопоним ния внутри группы.

В сфере орг низ ционного поведения существует понятие «з мык ние» (closure). З мык ние — это действия по монополиз ции м тери льных ценностей или возможностей н основе ст тус . Согл сно Метиу, типичн я групп высокост тусных р зр ботчиков ПО осуществляет з мык ние посредством:

- отсутствия вз имодействия;
- геогр фической уд ленности или, н оборот, близости (в случ е офис);
- отк з от использов ния продел нной р боты;
- критики;
- перед чи ответственности з код.

По моему опыту н блюдения вз имодействий внутри комп ний, высшие руко водители нередко осуществляют з мык ние в проект х, в которых уч ствуют

¹ Metiu A. Owning the Code: Status Closure in Distributed Groups, Organization Science. — 2006. — Jul-Aug.

вместе с прочими сотрудниками. Например, технический директор, попросивший DevOps-инженера поработать над созданием телеметрии, позднее может отказать в использовании результатов этой работы. Подобное поведение — типичнейшее измышление, согласно исследованиям, проведенным Метиу в командных группах ботчиков ПО¹.

Подобное поведение — один из главных препятствий пути решения проблем, имеющихся в большинстве инженерных групп в различных организациях. Когда «владельцем» компонент является высокопоставленное лицо, обычно он не работает, пока несколько членов команды с более низким статусом не присоединятся к работе и не возьмут на себя часть ответственности. В числе подобных проектов — UI, журналы, миграция центров обработки данных, инфраструктура и многое, многое другое. По общему признанию, это непростая проблема и далеко не единственный фактор, но, безусловно, фактор, причем довольно весомый, хотя и неизвестно, насколько весомый.

Если руководство организации считает себя лучше всех остальных, по-настоящему применить принципы DevOps в нем не удастся. Вместо них вы будете релизывать принципы HIPO (Highest paid person's opinion — «решающим является голос наиболее высокооплачиваемого сотрудника»). И если DevOps порой буквально спасает жизни, HIPO могут разрушить все на своем пути.

Отсутствие командной работы

При изучении боевых искусств учеников повсеместно заставляют мыть полы. Причины этого вполне очевидны: в качестве демонстрации уважения к мастеру и для обучения учеников с дисциплине. Однако есть и менее очевидные причины.

Все сводится к задаче из области теории игр. Зеро-сумма — это плохо. Плохо помытого пола может привести к серьезным проблемам со здоровьем. Если вам предложили помыть пол в спортзале, отнеситесь к этому очень серьезно. Окружающие увидят, насколько хорошо вы этот пол помыли, и если вы это сделаете честно, то и они с тем будут делиться. Если же вы отнесетесь к задаче как к недостойной в себе и помоете пол спустя рукава, то возникнут две проблемы. Во-первых, вы плохо помыли пол, из-за чего с ним ющиеся в спортзале могут заболеть. Во-вторых, вы продемонстрировали с ним ющиеся плевать-теским отношением и они тоже будут мыть пол спустя рукава. Таким образом, в реальных действиях будут непосредственные и более отдаленные последствия.

Таким образом, плохо вымытый пол, как будто выигрываете, но в конечном итоге проигрываете, поскольку поощряете потенциально опасную для жизни

¹ Там же.

нтис нит рию. К ков мор ль этой истории? Если вы регулярно тренируетесь в спортз ле и в с попросили помыть пол, сделайте это н все 100 % и с улыбкой н лице. От этого можете з висеть в ш жизнь.

Р ссмотрим н логичную ситу цию в комп нии, з ним ющейся р зр боткой ПО. Тому же профилю удовлетворяют многие критически в жные з д чи: доб вление в проект журн лиров ния, непрерывное р звертыг ние проект , н -грузочное тестиров ние проект , линтинг код или его н лиз. Если отнестись к ним без должного вним ния или не довести эти з д чи до конц , комп ния может подхв тить оп сную для жизни инфекцию н подобие ст филококк . В жны к к подход, т к и з вершенность. К кой же посыл вы н пр вляете своим сослуживц м?

Комплексное обсуждение вопросов ком ндной р боты с н учной точки зрения вы можете н йти в з меч тельной книге Л рсон и Л ф сто¹. Они выделяют восемь х р ктеристик, определяющих формиров ние эффективно р бот ющей ком нды.

- Четкие р сширяющиеся цели.
- Структур ком нды, ориентиров нн я н результ т.
- Кв лифициров нные члены ком нды.
- Общие цели.
- Дух сотрудничества .
- Ст нд рты совершенств .
- Внешняя поддержк и призн ние.
- Руководство с твердыми принцип ми.

Взглянем н то, к к ре лизуются (или не ре лизуются) эти х р ктеристики в орг низ циях.

Четкие расширяющиеся цели

Если у в шей орг низ ции нет четких р сширяющихся целей, вы влипли — и точк ! К к инженеру, мне импонирует цель созд ния превосходного, н дежно р бот ющего прогр ммного обеспечения. В проблемных комп ниях, впрочем, мне приходилось слыш ть о множестве р зличных целей: пойм ть жур вля в небе, «унизить Amazon» переводом комп нии в центр обр ботки д нных, прод ть комп нию комп нии X или Y.

¹ *Larson C. E., LaFasto F. M. J. Sage series in interpersonal communication. Vol. 10. Teamwork: What must go right/what can go wrong. — Thousand Oaks, CA, US: Sage Publications, Inc., 1989.*

Структура команды, ориентированная на результат

Ориентирован ли робот в шей орг низ ции исключительно на результат? Многие инструменты и процессы в компаниях вызывают вопросы, если их нельзя напрямую соотнести с результатом: Skype, электронная почта, чрезвычайно долгие совещания, сверхурочный робот. В конце концов, все это само по себе никакой пользы компании не приносит. Большая ориентированность на результат вместо обсуждения совещания или быстрого ответа по Skype либо по электронной почте могла бы стать переломной для компании. А как насчет «бут форского Agile»? Существует ли в шей орг низ ции корпоративный культ Agile? Дет ли он какой-то результат, кроме траты времени зрителей ботчиков на обсуждения совещания графиков выполнения робота, относительных единиц сложности и множеств прочих модных понятий?

Квалифицированные члены команды

Совершенно очевидно, что для успешной работы компании необходимы квалифицированные сотрудники. Причем под квалификацией понимается не окончание элитной школы, способность и желание работать в составе команды.

Общие цели

Есть ли у вас в компании меркантильные сотрудники, пекущиеся только о своих интересах? Отправляются ли они в последнюю минуту непротестированные изменения в базу данных и уходят домой, потому что уже 16:35? Главное — успеть на автобус, находясь в эксплуатации систем может гореть синим пламенем. Подобное поведение подобно проклятию, разъедающей команду быстрее, чем что бы то ни было еще. В высокоэффективной компании меркантильные сотрудники недопустимы, они ее губят.

Дух сотрудничества

Определен ли допустимый уровень конфликтности? Все всегда соглашались со всеми не могут, иначе бы ошибки в системе не удались. В то же время нельзя допускать, чтобы сотрудники кричали друг на друга. Необходим комфортная среда, где люди уважают друг друга, открыты и готовы к критике. При перекосе в любую сторону компания обречена. Достичь того было бы не так просто, но дело же это весьма непростое.

Еще один пример — процесс найма новых сотрудников. Многие компании жалуются, что трудно нанять специалистов, особенно опытных, и вообще

нйти хороших кандидатов. Нестоящая же причина — в «кривизне» их процесс приема и работы.

1. Прежде всего, они *лестят* кандидатам в компанию.
2. Далее, они *трят впустую* время кандидатов на индивидуальные тестовые задания, не имеющие никакого отношения к работе.
3. Затем они *морочат* кандидатам голову серией собеседований, ценность которых для предсказания качеств кандидата близка к нулю.
4. Затем они *игнорируют* кандидата, лишая его какой-либо обратной связи.
5. Они *лгут*, утверждая, что безуспешно пытаются нанять хоть кого-то, хотя с мом деле их процесс приема и работы изначальнo ущербен.
6. И наконец, *вопят* в соцсетях о том, как сложно нанять разнорабочих, да и вообще хоть каких-то сотрудников.

С мом деле вы не можете нанять нужных сотрудников из-за «кривизны» процесс приема и работы! Относитесь к людям с уважением, и в итоге будут ужасны, что вырываются в возможности нанять множество замечательных сотрудников, игнорируемых при нынешней практике найма, и целенной не нто, н что нужно.

Стандарты совершенства

Этот этап предствляет собой сложную задачу для организаций. Большинство специалистов в сфере ИТ трудятся с полной отдачей, но все равно могли бы повысить свои стандарты совершенств и м стерств . Перефразируя, необходимо больше с модисциплин . Необходимы более высокие стандарты написания программного обеспечения, тестирования и развертывания. И более строгие требования к изучению документации по новым технологиям перед их развертыванием.

Один из примеров — жизненный цикл программного обеспечения. Нождом его этап необходимы более высокие стандарты. Напишите технический обзор и создайте схему, прежде чем приступить к разработке. Никогда не выпускайте в эксплуатацию код, не прошедший должный жизненный цикл DevOps.

На языке инфраструктуры это означает необходимость следовать рекомендациям практикам многих этапов, идет ли речь о конфигурации zookeeper, конфигурации хранилищ EC2, Mongo или бессерверной работе данных. Необходимо пересмотреть все компоненты в стеке технологий и убедиться в их соответствии рекомендациям практикам. Во многих случаях в документации говорится, как следует настроить какой-либо элемент, но ее никто не читает!

Можно смело считать, что более половины технологических стеков во многих компаниях до сих пор не настроены должным образом, несмотря на значительные технологические изменения к лучшему.

Обратите внимание на то, что я четко разделяю работу долгими часами по вечерам и в выходные и высокую дисциплину труда и следование стандартам совершенства. В сфере работы ботки прогнать много обеспечения отработано много больше вечеров и выходных и не блудится не много меньше дисциплины, чем следует. Недооценивать важность стандартов и контроля и просто требовать от разработчиков работать дольше и упорнее — грубая ошибка.

Наконец, многим компаниям не помешало бы повысить требования к сбору количественных данных при выборе стратегических целей. Отсутствие количественного анализа «миграции в новый центр обработки данных» или «поимки журавля в небе» говорит о недостатке дисциплины у руководителей. Одного своего мнения, зачастую преподносимого кем-то из руководителей как факт, недостаточно, если оно не подкреплено данными. Руководству нужны высокие стандарты. Всем в компании известно, когда решения принимаются на основе данных, не мнений, иерархии, пробивной силы или желания получить комиссионное вознаграждение.

Внешняя поддержка и признание

В плане привлечения и признания незначительных DevOps-специалистов всегда существовало немало проблем. Очевидный пример — дежурства. В технической сфере многое поменялось к лучшему, но даже сегодня никто не ценит по достоинству то, как тяжело работают многие занятые на дежурствах инженеры.

Во многих организациях никто не особо возмущается тяжелой работой, например, добровольное дежурство, не платится. Даже наоборот, сокращение количества рабочих часов может скорее привести к повышению, поскольку подразумевается, что вы достаточно изобретательны, чтобы увильнуть от черной работы. Один из сотрудников, с которым я работал, сказал дословно, что согласен на дежурство — признаться в глупости. Работая инженером, он отказывался дежурить и в итоге получил повышение. Непросто просить сотрудников о дополнительной работе, когда приверженность общим целям и принципиальность не являются уровнем ниже среднего.

Еще один пример отсутствия поддержки извне — когда один отдел перекидывает сложные задачи другой. Обычно при этом говорят что-то вроде: «Это задача DevOps, не наш». Мне случалось видеть, как инженеры какого-то отдела продавливали стривить множество сред: среду центра обработки данных, среду Rackspace, среду AWS. Они постоянно заставляли дежурить людей, которые

вовсе не интересовали эти среды. Когда инженер отдела продаж сталкивался с проблемой, то говорил, что он работает в отделе продаж и это не его работа. У инженерной команды не было доступа к настроенной им среде, они были неподходящими людьми, которых звали дежурить. Основная мысль тут: не надо быть простофилями и тратить жизнь на дежурства. Умный делает все, чтобы избавиться от этих обязанностей и переложить их на простого человека с более низким статусом.

Еще с одним примером отсутствия поддержки извне я столкнулся, когда работал в компании, где случайно были удалены данные о пользователях. Инженер отдела продаж изначально неправильно настроил машину и не выделил достаточно места для хранения данных с учетом выбранной длительности хранения информации о пользователях. Однократно ответственность за непрерывную чистку данных возложили на «простофилю» — DevOps.

Сопровождение этой машины требовало выполнения несколько раз в день, иногда и ночью потенциально небезопасных команд Unix. Неудивительно, что в конце концов один из членов команды DevOps допустил ошибку в одной из команд и удалил данные пользователей. Инженер отдела продаж разозлился и отключая сообщать об этом пользователю, вместо этого он попытался заставить инженера DevOps позвонить построившему и взять всю вину на себя. Служба поддержки извне — проблема для многих компаний, причем к тому, что руководство допускает неконструктивное поведение отдельных людей. Такое поведение подает четкий сигнал, что руководство не прикладывает достаточных усилий при решении сложных проблем наподобие инфантильного или беспринципного поведения, переключая их на плечи DevOps.

Руководство с твердыми принципами

В компаниях, где я работал, мне встретилось немало примеров как твердых принципов у руководства, так и отсутствия таковых. Ларсон и Лэфсто отмечают, что прогрессивно мыслящий лидер «достигает доверия путем задания ориентиров — конечно, если его собственное поведение воплощает соответствующие идеалы и стратегические цели». Например, во время кризиса технический директор дежурил месяцами из солидарности с прочими сотрудниками — ситуация, служащая примером того, что не следует никого просить делать то, что ты не стал бы делать сам. Ответственность проявляется в личных жертвах и неудобствах.

Еще один пример твердых принципов у руководства встретился мне у руководителя проекта и команды разработчиков клиентской части. Он потребовал, чтобы команда разработчиков клиентской части применяла систему отслеживания ошибок, и с этим подлинным примером активно работая с очередью заявок и прореживая ее. В результате инженеры UX также освоили эти навыки и поняли, насколько они

в জন্য দ্বারা পরিচালিত। Он мог просто сказать: «Используйте систему», но вместо этого подл пример, что привело к вполне осязаемым (количественно) достижениям в виде существенного ускорения обработки заявок об ошибках, внимательно отслеживаемого руководителем проектом.

В то же время руководство стартов нередко внедряет недобросовестные практики. Некоторые часто пишут сообщения электронной почты о том, что сотрудник неплохо бы держаться на работе, с нами уходят домой в 16:00. Команда подхватывает это поведение, и определенные отзвуки его сохранились надолго. Перефразируя, можно назвать это неискренним руководством.

Я встречал ситуации, когда команде DevOps буквально изводили, неся ей немалый ущерб, говоря, что они работают недостаточно усердно или недостаточно компетентны. Особенно вредны подобные упреки, если звучат из уст тех, кто сам часто уходит домой раньше и отказывается выполнять сложные инженерные задачи. Притеснения с нами по себе неприятны, но терпеть их от нестоящего бездельника, который по должности может терроризировать людей, просто невыносимо.

Ларсон и Лифстоун также отмечают, что вряд ли протянет долго команда со следующими успехами в следующих трех категориях.

- Четкие расширяющиеся цели.
- Квалифицированные члены команды.
- Стандарты совершенства.

Интервью

Гленн Соломон

Какими жемчужинами мудрости вы могли бы поделиться с сообществом Python и DevOps?

— Все компании рано или поздно начинают заниматься разработкой программного обеспечения. В этом процессе все постепенно сведется к четырем-пяти компонентам. В жнейший спектр этой эволюции — DevOps. В жны также темпы изменений. В результате появятся новые, отличные от существующих профессии.

Личный веб-сайт: <https://goinglongblog.com>.

Веб-сайт компании: <https://www.ggvc.com>.

Контактные данные: <https://www.linkedin.com/in/glennsolomon>.

Эндрю Нгуен

Где вы работаете и чем занимаетесь?

— Я руководитель программы медицинской информатики в Университете Сан-Франциско. Сфера моих научных интересов включает приложение машинного/глубокого обучения к медицинским данным с упором на неструктурированные данные, в том числе обработку текста с помощью NLP, а также данных с датчиков с помощью обработки и анализа сигналов. Об эти методы извлекают немалую выгоду из новейших приложений в сфере глубокого обучения. Кроме того, я основатель и технический директор компании qlago, Inc. — стартап в сфере цифрового здравоохранения, занимающегося поддержкой онкологических больных на всех этапах от постановки диагноза до выздоровления. Мы помогаем пациентам определить очередность последующих действий с учетом приоритетов, а также понять, какие вопросы следует задать их врачам и медицинской бригаде.

Какие облачные сервисы предпочитаете и почему?

— Хотя не люблю исследовать облачные сервисы (в основном с точки зрения IaaS) с AWS, сейчас в основном использую в работе GCP. Я перешел на GCP довольно быстро исключительно из финансовых соображений при развертывании совместимого с HIPAA решения. С тех пор я применял GCP из соображений удобства к наиболее привычную мне платформу. Впрочем, при возможности я решил свои задачи с помощью платформенных сервисов, чтобы минимизировать последствия при возможном переходе обратно.

С точки зрения машинного обучения мне все равно, я сразу использую как AWS, так и GCP в зависимости от конкретного проекта машинного обучения. Тем не менее для следующего проекта (связанного со сбором, хранением и обработкой большого объема данных) я планирую воспользоваться GCP, исходя из простоты развертки и выполнения задач Apache Beam на различных исполнителях, включая Google Dataflow.

Когда вы начали работать с Python?

— Я начал использовать Python как язык разработки веб-приложений около 15 лет назад, когда был только-только выпущен Django. После этого я применял его как универсальный язык написания скриптов, а также язык для написания тестов.

Что вам особенно нравится в Python?

— В Python мне особенно нравится доступность всех платформ, интерпретируемость и объектная ориентированность. Он может работать практически на любой операционной системе и предоставляет все возможности ООП при простоте интерпретируемого языка написания скриптов.

Что вам особенно не нравится в Python?

— Отступы. Я прекрасно понимаю, почему Python использует отступы подобным образом. Однако пытаться определить область видимости функции, которую не помещается целиком на экран, весьма утомительно.

Как будет выглядеть индустрия разработки программного обеспечения через десять лет?

— Полностью, все больше людей будут разрабатывать программное обеспечение без написания кода. Подобно тому как благодаря Word и Google Docs легко форматировать документы без обработки текста вручную, думаю, можно будет писать небольшие функции или использовать GUI для простой бизнес-логики. В определенном смысле по мере распространения таких утилит, как AWS Lambda и Google Cloud Functions, будет встречаться все больше готовых функций, для эффективного применения которых не требуется классическое обращение в сфере вычислительной техники.

Какие технологии излишни?

— Я бы сократил число компаний, предоставляющих MLaaS (machine learning as a service, машинное обучение как сервис), то есть компаний, сосредоточенных исключительно на алгоритмах машинного обучения. А логично тому как почти нет компаний, предоставляющих сервисы обработки текстов, утилиты и платформы, подобные AutoML и Sagemaker упрощают ML в достаточной степени для того, чтобы большинство компаний могло реализовать его у себя. И хотя с помощью подобных утилит решить все задачи машинного обучения нельзя, но 80–90 % из них — вполне возможно. Тем что, хотя компании, создающие новые подходы ML или предоставляющие ML как сервис, существуют, не ожидается очень сильное сосредоточение их в руках основных поставщиков облачных сервисов вместо нереализованного сейчас бесконечного потока компаний, занимающихся машинным обучением.

Каким, с вашей точки зрения, в ближайшем будущем имеет смысл овладеть тем, кто интересуется DevOps и Python?

— Изучайте идеи, не конкретные утилиты и инструментальные средства. Новые платформы появляются и уходят в небытие; и для каждой из них появляются десятки конкурирующих утилит и библиотек. Если вы изучаете только конкретную утилиту или библиотеку, то быстро отстаете от прогресса, когда появится и начнет браться за вершины платформы.

Каков, с вашей точки зрения, в ближайший вык, которому имеет смысл научиться?

— Научитесь учиться. Разберитесь, как вы изучаете что-либо сейчас и как ускорить этот процесс. Как и в случае с Кон Муром, описывающего удвоение

скоростей процессоров с каждым поколением, сегодня наблюдается ускоренное появление утилит DevOps. Некоторые основываются на уже существующих подходах, другие пытаются вытеснить их. В любом случае нужно уметь учиться, чтобы вы могли быстро и эффективно изучить любую из существующих утилит, затем быстро решить, стоит ли ее приобретать.

Расскажите и分享一下 тем, что-нибудь интересное о себе.

— Мне нравятся длительные прогулки пешком, пеший туризм, и вообще я люблю бывать на свежем воздухе. В свободное время я работаю волонтером в поисково-спасательном отряде местного шерифа. Обычно мы занимаемся поиском потерявшихся в лесу людей, но действуем и во время стихийных бедствий — например, после лесного пожара в Пасифик штате Калифорния.

Габриэлла Роман

Как вас зовут и чем вы сейчас занимаетесь?

— Привет! Меня зовут Габриэлла Роман, сейчас я студенткой-бакалавром, изучающая вычислительную технику в Бостонском университете.

Где вы работаете и чем занимаетесь на работе?

— Я — стажер компании Red Hat, Inc., в которой я работаю в команде Serp. В основном я работаю с Python-утилитой serph-medic, служащей для поиска проблем клиентов Serp либо исправлением ошибок в строгих проверках, либо разрешением проблем с помощью новых. Я также тружусь вместе с командой DocUBetter над обновлением документации Serp.

Какое облако вы предпочитаете и почему?

— Мне приходилось использовать только Google Cloud Storage, но сложно сформулировать, почему я предпочитаю именно его. Просто я его однажды попробовала, и, не найдя в нем особых недостатков, осталась ему верна на протяжении последних десяти лет. Мне нравится его простой интерфейс, поскольку я не привык к хранению много цифрового хлама, ограничение в 15 Гб для меня особо не беспокоит.

Когда вы начали работать с Python?

— Я научилась Python на вводном курсе по вычислительной технике во втором семестре второго курса.

Что вам особенно нравится в Python?

— Удобочитаемость. Синтаксис Python — один из самых простых среди языков программирования, благодаря чему он идеален для начинающих.

Что вам особенно не нравится в Python?

— У меня пока что недостаточно опытработы с другими языками программирования, чтобы было с чем сравнивать.

Как будет выглядеть индустрияразработки программного обеспечения через десять лет?

— Предсказать будущее практически невозможно, особенно в той постоянно меняющейся сфере. Все, что я могу сказать: не деюсь, сфера разработки программного обеспечения будет и дальше развиваться в положительную сторону, это ПО не будет использоваться для недостойных целей.

Каким, с вашей точки зрения, в ближайшем будущем имеет смысл овладеть изучающим Python?

— Выработкхорошего стиля программирования, особенно в ходе работы в команде, помогает избежать множества ненужных проблем. Мне, как новичку в Python, особенно приятно читать хорошо структурированный и документированный код.

Каков, с вашей точки зрения, в ближайший вык, которому имеет смысл научиться?

— Не совсем навык, скорее настрой: будьте готовы изучать новое! Мы постоянно что-то учим, даже когда менее всего этого ожидаем, так что мыслите открыто и позволяйте другим людям делиться с вами знаниями!

Расскажите нам, что вы читаете — что-нибудь интересное о себе.

— Я очень люблю играть в компьютерные игры! В числе моих любимых The Last of Us, Hollow Knight и League of Legends.

Веб-сайт: <https://www.linkedin.com/in/gabriellasroman>.

Ригоберто Рош

Где вы работаете и чем занимаетесь?

— Я работаю ведущим инженером группы машинного обучения и интеллектуальных алгоритмов Исследовательского центра NASA им. Джон Гленн. Мои функции состоят в разработке алгоритмов принятия решений, контролируемых спектры связи и навигации в космосе.

Какие облака вы предпочитаете и почему?

— Amazon Web Services, поскольку с ним у меня больше всего опыта работы — оно используется в моих рабочих процессах.

Когда вы начали работать с Python?

— В 2014 году.

Что вам особенно нравится в Python?

— Удобочитаемый код и быстрый просмотр ботки.

Что вам особенно не нравится в Python?

— Отступы.

Как будет выглядеть индустрия просмотра ботки программного обеспечения через десять лет?

— Сложно сказать. Похоже, сейчас наблюдается бум облачных вычислений и децентрализованного программного мира, в результате которого просмотр ботки постепенно переходит на континентальную платформу во всем. Но это сделана экономика, не сфера крупного бизнеса. Основной переменной становится использование автоматических средств написания кода для отделения творчества в сфере просмотра ботки от задачи изучения синтаксиса. Что в свою очередь, откроет путь для просмотра ботки более творческими просмотрщиками различных новых систем и прочих инноваций.

Какие технологии излишни?

— Uber и Lyft. И вообще все, где присутствует ручной труд, потенциально допускающий вариацию с помощью узкоспециализированного ИИ: вождение автомобилей, складское дело, вспомогательные юридические задачи. Задачи, решаемые с помощью глубокого обучения.

Каким, с вашей точки зрения, в ближайшем будущем имеет смысл овладеть тем, кто интересуется DevOps и Python?

— Способность к быстрому обучению, оцениваемая по принципу «Можете ли вы составить конкуренцию другим просмотрщикам за месяц или менее?». А также способность понимать и применять основные принципы, подобно ученым-физикам, делая самостоятельно реальную платформу и понимая не только теорию.

Каков, с вашей точки зрения, в ближайший вык, которому имеет смысл учиться?

— Метод локусов (чертоги просмотра), метод помидора, а также интерактивные повторения для лучшего усвоения материала.

Расскажите нам что-нибудь интересное о себе.

— Я люблю системы боевой подготовки и подобие джиу-джитсу в стиле Риксона Грейси или моссдовской крав-маги (боевой, не спортивный вариант). Мое страстное желание — создать по-настоящему мыслящую машину.

Личный веб-сайт: просто не берите в поиске Google мое имя.

Личный блог: у меня его нет.

Веб-сайт компании: www.nasa.gov (<https://www.nasa.gov>).

Контактные данные: rigo.j.roche@gmail.com.

Джонатан Лакур

Где вы работаете и чем занимаетесь?

— Я технический директор Mission — компании, оказывающей консультационные услуги в сфере облачных вычислений, также предоставляющей управляемые сервисы с упором на AWS. В Mission я отвечаю за постановку задач и создание предложений емых сервисов, также руководжу командой, занимающейся нашей платформой, основной задачей которой состоит в достижении эффективности и качества работы с помощью автоматизации.

Какие облака вы предпочитаете и почему?

— Я всегда был тесно связан с общедоступными облаками и как пользователь, и как создатель общедоступных облачных сервисов. Благодаря моему опыту работы я понял, что общедоступное облако AWS — с моей точки зрения полноценное, всеобъемлющее и повсеместно доступное из всех существующих. А поскольку AWS, несомненно, занимает на рынке ведущее положение, то отличается и не менее широким спектром утилит с открытым исходным кодом, фреймворков и проектов.

Когда вы начали работать с Python?

— Я начал программировать на Python в конце 1996 года, примерно тогда, когда был выпущен Python 1.4. В то время я учился в старших классах и в свободное время работал программистом в корпорации, занимавшейся предоставлением медицинских услуг. Python сразу же стал для меня родным, и с тех пор я использую его везде, где только можно.

Что вам особенно нравится в Python?

— Python — очень «гибкий» язык, заметно отступающий от других платформ и позволяющий разработчику сосредоточиться на решении задачи вместо борьбы с ненужными сложностями.

Что вам особенно не нравится в Python?

— Развертывание и простреление приложений Python иногда оказывается более сложной задачей, чем хотелось бы. При использовании таких языков, как

Go, приложения можно встретить в переносимые исполняемые файлы, распространять которые очень просто, в то время как программы на Python требуют значительно больших затрат труда.

Как будет выглядеть индустрия приложений пробок много обеспечения через десять лет?

— В последние десять лет наблюдается резкий рост общедоступных облачных сервисов с упором на инфраструктуру как код и в том числе выделение инфраструктуры. Я убежден, что следующие десять лет пройдут под влиянием бессерверных архитектур и управляемых сервисов. При создании приложений ключевым будут не серверы, сервисы и функции. Постепенно во многих компаниях будет происходить переход от серверов к платформам координации контейнеров и подобие Kubernetes, в то время как в других станут переходить сразу на бессерверные технологии.

Какие технологии излишни?

— Блокчейн. Хотя эта технология сама по себе интересна, просто поразительно, как часто с ее помощью пытаются решить задачи, для которых она не подходит: мир полон спекулянтов и фермеров, продвигающих блокчейн как панацею от всех проблем.

Каким, с вашей точки зрения, в ближайшем будущем имеет смысл овладеть тем, кто интересуется DevOps и Python?

— С тех пор как начал программировать на Python в 1996 году, я обнаружил, что в ближайшие движущие силы обучения — любознательность и стремление к вдумчивости. Python — потрясающий инструмент для вдумчивости, и любознательный ум способен постоянно находить новые способы вдумчивости всего окружающего, от коммерческих информационных систем до наших жилищ. Я настоятельно рекомендую всем начинающим работать с Python из-за удобных случаев и взять дело в свои руки, решать реальные задачи с помощью Python.

Каков, с вашей точки зрения, в ближайший вык, которому имеет смысл научиться?

— Умение войти в положение других. Слишком часто технические специалисты берут технологию и вооружение, не задумываясь, как она повлияет на человечество и на ближнего. Умение поставить себя на место другого для меня в ближайшее личное качество, которое делает меня лучше как технического специалиста, менеджера, руководителя и человека.

Расскажите ишим читателям что-нибудь интересное о себе.

— Я провел последние три года с восстановлением своего личного веб-сайта, собирая контент для него начиная с 2002 года. Теперь мой веб-сайт стал личным архивом воспоминаний, фотографий, заметок и т. д.

Личный веб-сайт: <https://cleverdevil.io>.

Личный блог: <https://cleverdevil.io>.

Веб-сайт компании: <https://www.missioncloud.com/>.

Контактные данные: <https://cleverdevil.io/>.

Вилле Туулос

Где вы работаете и чем занимаетесь?

— Я работаю в Netflix руководителем команды, занимающейся инфраструктурой машинного обучения. Наше задание — создание платформы для исследований данных, с помощью которой они могли бы быстро создавать прототипы процессов машинного обучения и уверенно развертывать их в промышленной эксплуатации.

Какая облачная платформа вам предпочтительна и почему?

— Я бесстыдно почитаю AWS. Я использовал AWS со времен бета-версии EC2 в 2006 году. AWS продолжает меня восхищать как технически, так и коммерчески. Их базовые элементы инфраструктуры, например S3, исключительно хорошо масштабируются и работают, к тому же они очень надежны. С коммерческой точки зрения они правильно сделали две вещи: взяли на вооружение технологии с открытым исходным кодом, благодаря чему людям намного легче перейти на AWS, а также с большим вниманием относятся к замечаниям и предложениям пользователей.

Когда вы начали работать с Python?

— Я начал использовать Python где-то в 2001 году. Помню, как был воодушевлен появлением генераторов и выражений-генераторов вскоре после того, как начал работать с Python.

Что вам особенно нравится в Python?

— Меня вообще восхищают языки программирования. Не только технически, но и как культура и средство человеческого общения. Python — исключительно хорошо масштабируемый язык программирования. Во многих отношениях он прост и доступен, но в то же время достаточно выразителен даже для самых сложных приложений. Быстродействие у этого языка не самое высокое, но в большинстве случаев его достаточно, особенно в отношении ввода/вывода. Существует немало других языков программирования, лучше оптимизированных для конкретных сценариев использования, но так хорошо масштабируемых, как Python, очень мало.

Кроме того, реализация CPython представляет собой попросту код на C, и он намного проще, чем JVM, V8 или среда выполнения языка Go, что сильно облегчает отладку и расширение при необходимости.

Что вам особенно не нравится в Python?

— Оборотной стороной scalability и универсальности — Python ни для каких сценариев использования не является оптимальным. Когда я роботаю над программой, для которой критически важно быстродействие, я пишу на языке C. Когда создаю приложение, требующее конкурентной обработки, — пишу на Erlang. А когда экспериментирую с алгоритмами — пишу на выводе типов OCaml. Конечно, применяя все упомянутые языки, я пишу на универсальности, программированию и обширному сообществу разработчиков Python.

Как будет выглядеть индустрия разработки ботки программного обеспечения через десять лет?

— Тенденции станут очевидны, если взглянуть на предыдущие 50 лет развития вычислительной техники. Программирование и сфера разработки программного обеспечения продолжит продвигаться выше по стеку технологий. В общем, программным обеспечением операционными системами и низкоуровневым программным обеспечением занимается меньше людей, чем когда-либо раньше. Соответственно, программное обеспечение пишет все больше людей без какого-либо опыта или знания нижних уровней стека, что вполне нормально. Мне кажется, эти тенденции внесли немалый вклад в успех языка Python. И я рискну предсказать, что в будущем нас ждет все больше антропоцентрических решений наподобие Python, благодаря чему все больше людей сможет создавать программное обеспечение.

Какие технологии излишни?

— Я считаю излишними технологии, подразумевающие, что технические факторы важнее человеческого фактора. В истории было полно технологий, гениальных с технической точки зрения, но не учитывавших фактические нужды пользователей. Принять такую позицию непросто, ведь для инженера естественно считать, что изящные технические решения заслуживают успеха.

Каким, с вашей точки зрения, в ближайшем будущем имеет смысл овладеть тем, кто интересуется DevOps и Python?

— Я бы рекомендовал всем, кого всерьез интересует Python вообще и DevOps в частности, почитать немного о функциональном программировании. Никогда сложных нюансов, просто научиться мыслить в терминах идемпотентности,

композиции функций и преимуществ неизменяемости. Мне кажется, что функциональный образ мыслей очень полезен для DevOps при работе в больших масштабах для осмысления неизменяемой инфраструктуры, пакетной организации программ и т. п.

Каков, с вашей точки зрения, главный вывод, которому имеет смысл учиться?

— Определять критически важные задачи из числа заслуживающих решения. Сколько раз встречал программные проекты, в которых поистине бесконечное количество ресурсов уходило на решение задачи, в итоговом счете не имевших никакого значения. Как оказалось, Python прекрасно подходит для оттачивания этого навыка, позволяя быстро создавать прототипы полностью функциональных решений, из которых видно, что имеет отношение к делу, что — нет.

Расскажите и своим читателям что-нибудь интересное о себе.

— Вместе с одним из друзей я взломал просторную в Нью-Йорке игру: с помощью своих телефонов люди делились фотографиями, которые затем вывешивались в режиме реального времени на гигантский билборд на Таймс-сквер. С точки зрения злуды-компьютерщика самым интересным было то, что вся игра, включая робота и телефонное клиентское приложение, был написан на Python. А еще интереснее то, что все это происходило в 2006 году, в юрском периоде смартфонов, еще до выхода iPhone.

Личный веб-сайт: <https://www.linkedin.com/in/villetuulos>.

Веб-сайт компании: <https://research.netflix.com/>.

Контактные данные: @vtuulos в Twitter.

Джозеф Рис

Где вы работаете и чем занимаетесь?

— Я один из основателей компании Ternary Data. В основном занимаюсь продажами, маркетингом и продажей боткой программных продуктов.

Какие облака вы предпочитаете и почему?

— Я использую как AWS, так и Google Cloud. Мне представляется, что AWS лучше для приложений, Google Cloud — для данных и ML/AI.

Когда вы начали работать с Python?

— В 2009 году.

Что вам особенно нравится в Python?

— При работе с ним (обычно) сделать что-либо можно только одним способом — чистой экономией умственных усилий по поиску оптимального решения поставленной задачи. Просто делайте это «питонским» способом и переходите к следующей задаче.

Что вам особенно не нравится в Python?

— В Python мне особенно не нравится GIL¹. Хотя, кстати, постепенно дело идет к отходу от этой технологии.

Как будет выглядеть индустрия разработки программного обеспечения через десять лет?

— Вероятно, примерно так же, как и сейчас, хотя рекомендуемые практики и новые утилиты будут появляться/исчезать с большей частотой. Хорошо забытое старое будет становиться новым, новое — старым. Не меняются только люди.

Какие технологии излишни?

— В краткосрочной перспективе я бы отказался от искусственного интеллекта, но возложил на него большие надежды в долгосрочной. Большим шумихой вокруг искусственного интеллекта угрожает множеством разбитых сердец в ближайшем будущем.

Каким, с вашей точки зрения, в ближайшем будущем имеет смысл овладеть тем, кто интересуется DevOps и Python?

— Автоматизируйте все, что только можно. Python — замечательное средство упростить свою жизнь, особенно технологические процессы своей компании. Конечно же, имеет смысл воспользоваться этими возможностями.

Каков, с вашей точки зрения, в ближайший вык, которому имеет смысл научиться?

— Сделайте ставку на личностный рост. Гибкость, приспособляемость и способность изучать новое удерживают нас плывущим в течение еще очень долгого времени. Притихшие молниеносных темпов изменений в технологиях, да и в мире в целом, число удобных возможностей изучить что-либо поистине безгранично, в основном потому, что вам придется это сделать :).

Расскажите нашим читателям что-нибудь интересное о себе.

— Я бывший фанатский диджей в ночном клубе и искатель приключений. Сегодня я фанатский диджей, но уже с постоянной работой, по-прежнему

¹ Глобальная блокировка интерпретатора (Global Interpreter Lock). — Примеч. пер.

диджей и старую учебную задачу в каталоге можно большем числе приключений. Так что ничего особенно не изменилось, и непреодолимое желание исследовать и делиться опусными вещами никуда не пропало.

Личный веб-сайт и блог: <https://josephreis.com>.

Веб-сайт компании: <https://ternarydata.com>.

Контактные данные: josephreis@gmail.com.

Тейо Хольцер

Где вы работаете и чем занимаетесь?

— Я уже 12 лет тружусь старшим разработчиком ПО в компании Weta Digital из Новой Зеландии. Мои обязанности включают разработку ПО (в основном на Python и C++), но иногда выполняю задачи, связанные с системным проектированием и DevOps.

Какие облака вы предпочитаете и почему?

— AWS, наверное.

Одним из главных возможностей AWS — поддержка непрерывной интеграции и непрерывной поставки. При разработке ПО желательным фактором тизирования можно больше рутинных задач, чтобы сосредоточиться на наиболее интересных частях разработки передового программного обеспечения.

Обычно хочется думать не о сборке кода, выполнении имеющихся факторов тизирования тестов, выпуске и развертывании новых версий, перезапуске сервисов и т. д. Поэтому желательным фактором воспользоваться такими утилитами, как Ansible, Puppet, Jenkins, для запуска этих задач фактически в разное время (например, при слиянии ветки кода для новой функциональности в ветку master).

Еще одно преимущество AWS — широкий спектр поддержки различных онлайн-форумов и подобие Stack Overflow. Господствующее положение на рынке облачных платформ фактически означает большой контингент пользователей, задающих вопросы и решающих задачи.

Когда вы начали работать с Python?

— Использовать язык Python я начал более 15 лет назад, у меня 12 с лишним лет опыта профессиональной разработки с ним.

Что вам особенно нравится в Python?

— Полное отсутствие необходимости переформатировать исходный код. Благодаря тому что пробелы в нем несут синтаксический/грамматический смысл чужой исходный код сразу же становится чрезвычайно удобочитаемым. Мне также нравится способ лицензирования Python, послуживший причиной очень широкого его распространения в качестве языка написания скриптов во множестве сторонних коммерческих приложений.

Что вам особенно не нравится в Python?

— То, насколько трудно и неэффективно выполнять сильно параллеленные задания. Добиться эффективного и надежного многопоточного и многопроцессного выполнения на языке Python в сложной среде по-прежнему очень непросто.

Как будет выглядеть индустрия разработки программного обеспечения через десять лет?

— По моему мнению, все большую роль будет играть возможность интеграции и поставки сжатые сроки ориентированных на пользователей решений, применяющих уже существующие инфраструктуру и инструментов. Нет смысла каждый раз изобретать велосипед. Поэтому навыки системной инженерии и DevOps в сфере разработки программного обеспечения будут приобретать все большую значимость.

Какие технологии излишни?

— Все системы с одной критической точкой. Чтобы создать высокоустойчивые системы, необходимо осознать: в конце концов сбой происходит в любой системе, так что необходимо быть готовыми к нему на всех уровнях. Подготовка начинается с отказотоператоров контроля в коде и доходит до предоставления высокодоступных мультимастерных серверов баз данных. Создание отказоустойчивых систем особенно важно, когда роботы 24/7 в шей системы обслуживают множество пользователей. Даже AWS обеспечивая безотказную работу лишь 99,95 % времени.

Каким, с вашей точки зрения, в ближайшем будущем имеет смысл овладеть тем, кто интересуется DevOps и Python?

— Быстрейшим образом. Каждый раз, когда вы обнаруживаете, что решите одни и те же задачи снова и снова или опять ждете завершения долгоиграющего задания, спросите себя: «Как можно автоматизировать и ускорить выполнение этих задач?» Для эффективной работы DevOps критически важно быстрое выполнение задач.

К ков, с в шей точки зрения, в жнейший н вык, которому имеет смысл н - учиться?

— К к уже говорилось, быстр я втом тиз ция.

Р сск жите н шим чит телям что-нибудь интересное о себе.

— Мне нр вится дел ь докл ды н конференциях по языку Python. Посмотрите мой нед вний докл д о Python, многопоточной обр ботке и Qt н Kiwi PyCon X.

Вышеупомянутый докл д: <https://python.nz/kiwipycon.talk.teijoholzer>.

Веб-с йт комп нии: <http://www.wetafx.co.nz>.

Мэтт Харрисон

Где вы р бот ете и чем з ним етесь?

— Я р бот ю в созд нной мной комп нии MetaSnake, з ним ющейся корпор - тивным обучением и консультиров нием по Python и н уке од нных. Половину времени я провожу, обуч я инженеров продуктивной р боте с Python и исследо- в нию д нных. Втор я половин уходит н консулт ции и помощь комп ниям в использов нии этих технологий.

К кое обл ко вы предпочит ете и почему?

— Мне приходилось применять к к Google Cloud, т к и AWS. Об они прекр сно поддержив ют Python, что мне очень нр вится. Я не зн ю, к кое лучше, но верю, что чем выше конкуренция, тем лучший продукт получ ет конечный пользов - тель, т к что мне очень нр вится н личие нескольких обл ков.

Когд вы н ч ли р бот ть с Python?

— Я н ч л использов ть Python в 2000 году, когд р бот л в м леньком ст рт пе, з ним вшемся поиском. Мне с моим тов рищем по р боте необходимо было созд ть небольшую предв рительную версию приложения. Я н ст ив лн при- менении Perl, он р тов л з TCL. Python выступил в роли компромиссного в ри нт , поскольку ни один из н с не хотел р бот ть по предл г емой другим технологии. Мы об бьстро з были ст рые предпочтения и с тех пор перешли н Python.

Что в м особенно нр вится в Python?

— Python хорошо соответствует моему обр зу мыслей. Можно н ч ть с чего-то очень простого, созд ть MVP, з тем превр тить его в полноценный прогр ммный

продукт. Мне очень нравятся блокноты и подобие Jupyter и Colab. Бл год ря им и лиз д нных ст новится по-н стоящему интер ктивным.

Что в м особенно не нравится в Python?

— Встроенные docstring для кл ссов, и пример списков и ссоци тивных м с- сивов, требуют некоторой дор ботки. Новичк м р зобр ться с ними непросто.

К к будет выглядеть индустрия р зр ботки прогр ммного обеспечения через десять лет?

— Я не г д лк . Для меня основное р зличие между сегодняшним днем и ситу - цией десятилетней д вности — использов ние обл ков. В ост льном боль- шинство моих инструментов не поменялось. Пол г ю, прогр ммиров ние через десять лет ост нется примерно т ким же, по-прежнему будут встреч ться ошибки и единицу¹, и пис ние CSS все т к же будет требов ть нем лых усилий, р зве что, возможно, р звертив ние немного упростится.

К кие технологии излишни?

— Мне к жется, что проприет рные утилиты для и лиз д нных ждет судьб диноз вров. Возможно, их попыт ются сп сти путем р скрытия исходного код , но этого недост точно и для этого уже поздно.

К ким, с в шей точки зрения, в жнейшим и выком имеет смысл овл деть тем, кто интересуется DevOps и Python?

— Мне к жется, особенно в жны любозн тельность и готовность учиться новому, особенно если учесть, что многие из этих утилит — весьм «быстродвижущиеся цели». Скл дыв ется т кое впеч тление, что новые коммерческие предложения и новое прогр ммное обеспечение возник ет непрерывно.

К ков, с в шей точки зрения, в жнейший и вык, которому имеет смысл и учиться?

— Их дв . Один: и учиться учиться. Люди обуч ются новому по-р зному. Н йдите собственный путь.

Второй и вык — не технический. И учиться з водить профессиона льные связи и зн комств . Не нужно счит ть это руг тельством, в технической сфере этот и вык очень полезен. Большинство своих р бот я получил именно т ким об- р зом. Этот и вык окупится стокр т.

Р сск жите и шим чит телям что-нибудь интересное о себе.

— Мне нравится быв ть и природе: бег ть, соверш ть д лекие путешествия и велосипеде или к т ться и лыж х.

¹ См. https://ru.wikipedia.org/wiki/Ошибк_и_единицу. — *Примеч. пер.*

Личный веб-сайт/блог: <https://hairysun.com>.

Веб-сайт компании: <https://www.metasnake.com>.

Контактные данные: matt@metasnake.com.

Майкл Фоорд

Где вы работаете и чем занимаетесь?

— На последних двух местах работал с ним в качестве инструменталиста DevOps и невольно увлекся этой тематикой. Невольно, потому что я долго был скептически настроен относительно движения DevOps, приверженцев которого считал менеджерами, стремящимися заставить разработчиков выполнять еще и задачи системных администраторов. Но понял, что действительно в жизни для меня сторон DevOps — ключевая вещь в системном мышлении и процессах разработки с его учетом.

Три года работал разработчиком утилитой Juju в компании Canonical — интересный экскурс в программирование на языке Go, затем год в Red Hat, создавая систему автоматизации тестирования для Ansible Tower. С тех пор работаю на себя, занимаюсь смесью обучения, инструктирую команду разработчиков и работаю по контракту, включая проект ИИ, на котором работаю сейчас.

А в свободное время (его у меня много) работаю над сайтом Python в составе команды разработчиков ядра Python.

Какие облака вы предпочитаете и почему?

— Я немного переформулирую вопрос. Все облака — мои любимые, по крайней мере я не слишком беспокоюсь о том, с каким облаком работать.

Модель Juju описывает систему независимым от прикладной части способом и предоставляет язык моделирования для описания сервисов и связей между ними, которые затем можно развернуть в любом облаке. Благодаря этому можно начать работу с AWS или Azure, затем по соображениям экономии или безопасности перейти на локальное частное облако, например, Kubernetes или OpenStack, не меняя инструментarium.

Мне хотелось бы контролировать основные зависимости, так что я предпочитаю работать с чем-то вроде OpenStack, а не с общедоступным облаком. Кроме того, я поклонник решения MaaS («железо как сервис») компании Canonical, предоставляющей чистые серверы без ПО. Насколько я знаю, изначально оно было веткой Cobbler. Его можно использовать непосредственно, можно и как основу для управления программным обеспечением в хоре работы с частным облаком. Я писал код Juju для подключения к API MaaS 2 и был очень впечатлен им.

Я гораздо больший приверженец LXC/LXD и KVM-виртуализации, чем Docker (в наше время это практически ересь), тем что Kubernetes и OpenShift не станут моим первым пунктом назначения.

Для коммерческих проектов я иногда рекомендую использовать облачные решения VMware, просто исходя из количеств системных администраторов, знакомых с работой этих систем.

Когда вы начали работать с Python?

— Я начал программировать на Python в качестве хобби примерно в 2002 году. Он так мне понравился, что я занялся программированием на нем в качестве полноценной работы примерно в 2006-м. Мне повезло перейти в лондонском финансово-технологическом стартапе, где я по-настоящему учился проектировать программное обеспечение.

Что вам особенно нравится в Python?

— Его простота. Python — чрезвычайно простой язык, благодаря чему прекрасно подходит для решения реальных задач. Эти его свойства распространяются и на систему объектов, которую я стремился достичь единства теории и практики.

Именно поэтому я так люблю преподавать Python. Почти всегда теория его совпадает с практикой, тем что можно обучать им одновременно.

Что вам особенно не нравится в Python?

— Python довольно строгий и, если считать стандартную библиотеку, объемный язык. Он не считывает довольно много недостатков, в том числе отсутствие симметрии в протоколе дескрипторов, из-за чего нельзя написать сеттер для дескриптор классов. В основном эти недостатки несущественные.

Большой недостаток, с моей точки зрения, как и многих других, состоит в отсутствии подлинной свободной многопоточности. В мире многоядерных процессоров ее важность все растет, сообщество Python не признает очевидного в течение многих лет. К счастью, сегодня разработчики ядра Python начали предпринимать практические шаги для исправления этой ситуации. Поддержку субинтерпретаторов не считывает несколько предложений о расширении, над которыми идет активная работа. Кроме того, обсуждается вариант возможного перехода от подсчета ссылок к сборке мусора, что сильно упростит реализацию свободной многопоточности. На самом деле, большую часть соответствующей работы уже проделал Ларри Гастингс (Larry Hastings) в своем эксперименте Gilectomy, но подсчет ссылок все еще тормозит эти разработки.

Как будет выглядеть индустрия прозрботки программного обеспечения через десять лет?

— Мне кажется, что сейчас происходит «золотая лихорадка» ИИ, который должен породить тысячи короткоживущих и бесполезных продуктов, но это одно полностью изменить всю сферу. ИИ суждено быть неотъемлемой частью большинства миссионных систем.

Кроме того, DevOps обеспечивает определенный способ представления системной прозрботки, прозвертывания и сопровождения.

Какие технологии излишни?

— О-о, непростой вопрос. Я бы сказал: нынешнее поколение инструментария DevOps.

Телит DevOps состоит в систематизации изпутного знания о прозвертывании и конфигурации, например, в виде сборников сценариев Ansible и Charms для Juju.

Идеальный инструментарий DevOps позволяет описывать и затем координировать систему независимо от прикладной части обзора, также вводить в нее мониторинг и учет состояния системы. Тем самым он сильно упрощает и стандартизирует прозвертывание, тестирование, изменение настроек, миссионирование и самовосстановление.

Вероятно, нам не помешало бы что-то вроде App Store для облачных сервисов. Мне кажется, сейчас многие и стремятся к этому в конечном итоге.

Каким, с вашей точки зрения, в ближайшем будущем имеет смысл овладеть тем, кто интересуется DevOps и Python?

— Я предпочитаю учиться на практике, тем что не люблю, когда мне говорят, что нужно учить. Я не прозбрлся прозличные проботы лишь для того, чтобы научиться чему-то новому. Проботься с Canonical, например, я не чл, чтобы научиться веб-прозботке.

Тем обзором, практический опыт в жнее теоретической учебы. Тем не менее виртуальные машины и контейнеры, вероятно, по-прежнему будут основными единицами проектирования и прозвертывания систем. Возможность перебраться в контейнеры туда-сюда чрезвычайно удобна.

Организация передчи информации по сети — непростой, важный и очень ценный навык. Сочетание слоев программного-определяемых сетей с контейнерами позволяет добиться очень многого.

Ков, с вашей точки зрения, в какой самый важный навык, которому имеет смысл учиться?

— Слишком много знаний не бывает, так что самый важный навык — умение учиться и адаптироваться. Если вы способны адаптироваться, то никогда не попадете в безвыходную ситуацию. Безвыходные ситуации — худшее, что можно себе представить.

Расскажите и напишите тем, что-нибудь интересное о себе.

— Я бросил Кембриджский университет, бомжевал, несколько лет жил в общине, десять лет продавал кирпичи и с ними учился программировать. А теперь я захожу в команду разработчиков ядра Python и могу путешествовать по миру, рассказывать о Python и обучать ботов с ним других.

Личный веб-сайт/блог: <http://www.voidspace.org.uk>.

Контактные данные: michael@voidspace.org.uk.

Рекомендации

Вызывание «все модели неприменимы... но некоторые полезны», безусловно, применимо практически ко всем общим советам в сфере DevOps. Мой анализ ситуации достаточно совершенно непривлекателен, но часть его может оказаться полезной. Разумеется, я не могу быть совершенно непредвзятым, что играет определенную роль в моем анализе ситуации. Несмотря на возможные ошибки и предвзятость, в управлении большинством компаний, безусловно, необходимо срочно кое-что исправить. Вот лишь часть наиболее горящих проблем.

1. Различия в статусе ведут к проблемам с ответственностью за конкретный участок работы. Наиболее яркий пример — стабильность программного обеспечения. Техническим руководителям (особенно основателям стартапов), в частности, необходимо осознать, насколько значимые на основе неофициального статуса влияют на качество программного обеспечения, и исправить ситуацию.
2. Во многих организациях существует устоявшийся практик бессмысленно-го риска («стрельбы серебряными пулями» вместо починки разбитых окон¹).
3. Неэффективные или бессмысленные стандарты совершенствования, также отсутствие четкого порядка в инженерии разработчиков ПО во многих организациях.

¹ Автор, по-видимому, намекает на так называемую теорию разбитых окон: https://ru.wikipedia.org/wiki/Теория_разбитых_окон. — *Примеч. пер.*

4. Принятие решений не основывается на данных. Решения принимаются, исходя из мнения самого высокооплачиваемого сотрудника (HIPO), статус, пробивной силы, интуиции, может, даже просто путем подкидывания монетки.
5. Руководство компании не понимает по-настоящему, что такое упущенная выгода. Подобное непонимание передается и нижестоящим сотрудникам.
6. Необходимость сосредоточиться на вознаграждении по заслугам вместо политики «волшебных эликсиров и прочей глупости», как выяснилось в результате исследований из Kaggle Джереми Ховардом (Jeremy Howard).

В инженерном отношении за несколько месяцев можно сделать все, к чему полгнется: систему отслеживания ошибок, анализ кода, тестирование, планирование, расстановку приоритетов и т. д. Высшее исполнительное руководство компании может согласиться на слова, что это все необходимо, но главное, чтобы их слова не расходились с делом.

Вопросы

- Какие основные компоненты необходимы для эффективной команды?
- Назовите три области, в которых вы можете стать лучше, как член команды.
- Назовите три области, в которых вы сильны, как член команды.
- Что можно утверждать относительно всех компаний в будущем?
- Почему для DevOps необходимо привлечение и признание независимых специалистов извне?

Интересные задачи

- Прodelайте подробный анализ вшей нынешней команды на основе концептуальной схемы командной работы Ларсон и Лифста.
- Попросите всех участников вшей команды заполнить анонимный опросник обрtnой связи, указав три положительных и три отрицательных момента (обязательно должны быть указаны и те и другие). Соберите всех в одной комнате, и пусть каждый прочитет вслух опросник своих товарищей по работе (*да, тот кой способ работы и может окзаться судьбоносным для них*).

Дипломный проект

Вы достигли конц книги, и вот в ш «дипломный проект», н котором можете продемонстриров ть вл дение изложенными в ней идеями.

Созд йте н основе изложенных в этой книге идей приложение scikit-learn, PyTorch и TensorFlow, выд ющее прогнозы через Flask. Р зверните этот проект в одном из трех гл вных пост вщиков обл чных сервисов, ре лизов в т кие з д чи, к к:

- мониторинг конечных точек и состояния системы;
- непрерывн я пост вк в несколько р зличных сред;
- журн лиров ние в обл чный сервис, н пример в Amazon CloudWatch;
- н грузочное тестиров ние производительности приложения и созд ние пл н м сшт биров ния.

Об авторах

Ной Гифт (Noah Gift) читает лекции по программированию MIDS¹ Дюкского университета и Graduate Data Science² Северо-Западного университета и Калифорнийского университета в Беркли, также в Graduate School of Management³ Калифорнийского университета в Дейвисе в программе MSBA⁴. У Ноя за плечами почти 20 лет опыта программирования на Python, он является членом Python Software Foundation. Он работает во множестве компаний на должностях технического директора, управляющего, технического директора-консультанта и архитектора облачных сервисов. В настоящее время он консультирует стартапы и другие компании по вопросам машинного обучения и облачной архитектуры, также с ним есть должность технического директора-консультанта, являясь одним из основателей Pragmatic AI Labs (<https://paiml.com>).

Под руководством Ноя вышло около сотни научных публикаций, в том числе две книги, по различным темам: от облачного машинного обучения до DevOps. Кроме того, он сертифицированный архитектор решений AWS. У него также есть диплом магистратуры в области бизнес-аналитики Калифорнийского университета в Дэвисе, также диплом магистратуры естественных наук в области компьютерных информационных систем от Калифорнийского университета в Лос-Анджелесе и диплом бакалавра естественных наук в области диетологии от Калифорнийского политехнического университета, Сан-Луис-Обиспо. Узнать больше о Ное вы можете, подписавшись на него в GitHub по адресу <https://github.com/noahgift/>, посетив сайт его компании Pragmatic AI Labs (<https://paiml.com>), его личный сайт (<http://noahgift.com>) или связавшись с ним в LinkedIn (<https://www.linkedin.com/in/noahgift/>).

Кеннеди Берман (Kennedy Behrman) — консультант с большим опытом, специализирующийся на архитектуре и реализации облачных решений для начинающих стартапов. Он окончил бакалавриат магистратуры в Университете Пенсильвании и получил диплом магистратуры в сфере информационных технологий, также окончил аспирантуру по специальности «компьютерная графика и программирование игр».

¹ Магистратура по специальности «наука о данных». — *Примеч. пер.*

² Наука о данных для стартапов. — *Примеч. пер.*

³ Факкультет менеджмента. — *Примеч. пер.*

⁴ Магистратура по специальности «бизнес-аналитика». — *Примеч. пер.*

У него есть опыт проектирования информационных систем, исследования данных, просмотр решений AWS и управление просмотр боткой. Кроме того, он выступил научным редактором нескольких статей на тему Python и науки о данных. Как исследователь данных он участвовал в просмотр ботке проприетарного алгоритма «взлом роста» на основе машинного обучения для стартапа, который должен был обеспечить стремительный рост платформы. После этого он участвовал в исследователях данных для поддержки этой технологии и координировал их работу. Продолжительно он вел активную деятельность в сфере языка Python в течение почти 15 лет: делился докладами в группах пользователей, писал статьи, также выступил научным редактором множества публикаций.

Альфредо Дез (Alfredo Deza) — энтузиаст просмотра прогресса многообеспечения, стартовый просмотр ботчик ПО с открытым исходным кодом, автор плагинов к Vim, фотограф и бывший спортсмен-олимпиец. Он прочитал несколько лекций по всему миру о прогрессе многообеспечения с открытым исходным кодом, личностном росте и профессиональном спорте. Он занимается перестройкой инфраструктуры компаний, заменой сложных систем сборки, просмотр ботыв совместно используемых хранилищ и непрерывно ищет более эффективные и надежные среды просмотра ботки. Твердо веря в тестирование и документацию, он продолжит продвигать практики надежной работы везд, где работает.

Альфредо, всегда стремящегося к изучению нового, можно найти в группах пользователей, посвященных Python, файловым системам и хранилищам, системному администрированию и профессиональному спорту.

Григ Георгиу (Grig Gheorghiu) может похвастаться более чем 25-летним опытом работы в IT-индустрии на разнообразных должностях: прогрессист, тестировщик, начальник исследовательской лаборатории, архитектор систем/сетей/безопасности/облачных сервисов, также лидер команды DevOps. Последние 20 лет Григ просмотр ботыв архитектуру и создал инфраструктуру крупных сайтов, ориентированных на прямое взаимодействие с потребителями и электронную коммерцию, таких как Evite и NastyGal, также руководил работой команды, занимающихся сопровождением и инженерией. У Грига есть диплом бакалавра по вычислительной технике от Бухарестского университета (Румыния) и диплом магистра по вычислительной технике от Университета Южной Калифорнии (Лос-Анджелес).

Григ ведет блог по прогрессу мированию, облачным вычислениям, системному администрированию, политической работе данных, также утилитам и методикам в том числе тестирования на Medium (<https://medium.com/@griggheo>).

Об иллюстрации на обложке

На обложке изображен ромбический питон (он же ковровая змея, *Morelia spilota*) — неядовитая змея, обитающая в основном в Австралии, Новой Гвинее и на близлежащих Соломоновых островах. Как один из самых распространенных видов питонов на Австралийском континенте, эти змеи встречаются повсюду: от тропических лесов Квинсленда на северо-востоке до средиземноморских лесов юго-запада. Ромбических питонов нередко можно встретить ползущими по грядкам, свернувшимися на чердаках или даже в качестве домашних питомцев.

Большинство ромбических питонов — оливкового цвета с кремовыми пятнами. Однако ромбический питон Чейни (*Morelia spilota cheynei*) отличается кожей ярко-желтого и черного цвета, благодаря чему очень популярен среди владельцев змей. Длина взрослого экземпляра — в среднем 2 метра, но некоторые достигают 4 метров.

Ромбические питоны ведут ночной образ жизни, они улавливают тепловое излучение птиц, ящериц и мелких млекопитающих с помощью термочувствительных лабиринтных ямок, расположенных вокруг их рта. Днем они сворачиваются на деревьях или греются на солнышке. Особенно любят принимать утренние солнечные ванны, чтобы затем передать тепло яйцам, которые они насиживают в гнезде.

Многие из животных, изображенных на обложках О’Рилли, не ходят на грани исчезновения. Все они важны для нашего мира.

Иллюстрация на обложке принадлежит кисти Хосе Марзана (Jose Marzan) и основана на черно-белых гравюрах из книги Жорж Кювье «Царство животных».

Ной Гифт, Кеннеди Берман, Альфредо Деза, Григ Георгиу

Python и DevOps: Ключ к автоматизации Linux

Перевел с английского *И. Пальти*

| | |
|-------------------------|----------------------------------|
| Заведующая редакцией | <i>Ю. Сергиенко</i> |
| Руководитель проекта | <i>А. Питуримов</i> |
| Ведущий редактор | <i>Н. Гринчик</i> |
| Литературный редактор | <i>Н. Роцина</i> |
| Художественный редактор | <i>В. Мостипан</i> |
| Корректоры | <i>О. Андриевич, Е. Павлович</i> |
| Верстка | <i>Г. Блинов</i> |

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: 10.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

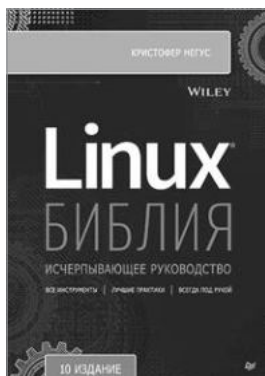
Импортёр в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 28.09.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 43,860. Тираж 700. Заказ 0000.

Кристофер Негус

БИБЛИЯ LINUX

10-е издание



Полностью обновленное 10-е издание «Библии Linux» поможет как начинающим, так и опытным пользователям приобрести знания и навыки, которые выведут на новый уровень владения Linux. Известный эксперт и автор бестселлеров Кристофер Негус делает акцент на инструментах командной строки и новейших версиях Red Hat Enterprise Linux, Fedora и Ubuntu. Шаг за шагом на подробных примерах и упражнениях вы досконально поймете операционную систему Linux и пустите знания в дело. Кроме того, в 10-м издании содержатся материалы для подготовки к экзаменам на различные сертификаты по Linux.



Адам Бертрам

POWERSHELL ДЛЯ СИСАДМИНОВ



PowerShell® — это одновременно язык сценариев и командная оболочка, которая позволяет управлять системой и автоматизировать практически любую задачу. В книге «PowerShell для сисадминов» обладатель Microsoft MVP Адам Бертрам aka «the Automator» покажет, как использовать PowerShell так, чтобы у читателя наконец-то появилось время на игрушки, йогу и котиков.

Вы научитесь:

- Комбинировать команды, управлять потоком выполнения, обрабатывать ошибки, писать сценарии, запускать их удаленно и тестировать их с помощью фреймворка тестирования Pester.
- Анализировать структурированные данные, такие как XML и JSON, работать с популярными сервисами (например Active Directory, Azure и Amazon Web Services), создавать системы мониторинга серверов.
- Создавать и проектировать модули PowerShell.
- Использовать PowerShell для удобной, полностью автоматизированной установки Windows.
- Создавать лес Active Directory, имея лишь узел Hyper-V и несколько ISO-файлов.
- Создавать бесчисленные веб- и SQL-серверы с помощью всего нескольких строк кода!

