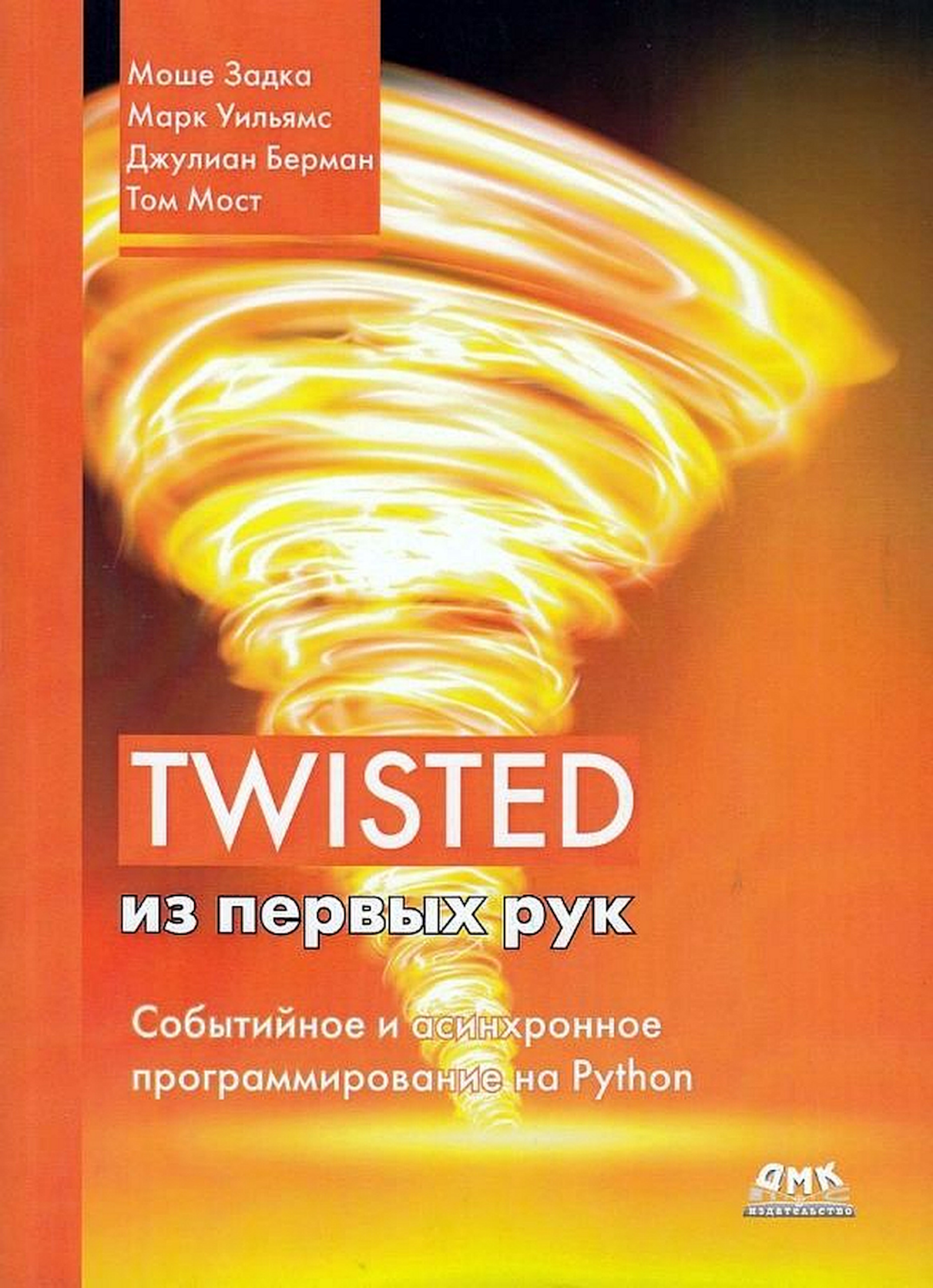




Моше Задка
Марк Уильямс
Джулиан Берман
Том Мост

TWISTED

из первых рук

Событийное и асинхронное
программирование на Python

ОМК
ИЗДАТЕЛЬСТВО

Моше Задка, Марк Уильямс, Кори Бенфилд, Брайан Уорнер,
Дастин Митчелл, Кевин Сэмюэл, Пьер Тарди

Twisted из первых рук

Expert Twisted

Event-Driven and Asynchronous Programming with Python

Moshe Zadka
Mark Williams
Cory Benfield
Brian Warner
Dustin Mitchell
Kevin Samuel
Pierre Tardy

Apress®

Twisted из первых рук

Событийное и асинхронное
программирование на Python

Моше Задка
Марк Уильямс
Кори Бенфилд
Брайан Уорнер
Дастин Митчелл
Кевин Сэмюэл
Пьер Тарди



УДК 004.438
ББК 32.973.22
315

**Задка М., Уильямс М., Бенфилд К., Уорнер Б.,
Митчелл Д., Сэмюэл К., Тарди П.**

315 Twisted из первых рук / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2020. – 338 с.: ил.

ISBN 978-5-97060-795-4

Эта книга посвящена Twisted – событийно-ориентированному сетевому фреймворку на Python, в котором можно создавать уникальные проекты. В первой части рассматриваются особенности Twisted; на практических примерах показано, как его архитектура способствует тестированию, решает общие проблемы надежности, отладки и упрощает выявление причинно-следственных связей, присущих событийно-ориентированному программированию. Детально описываются приемы асинхронного программирования, подчеркивается важность отложенного вызова функций и сопрограмм. На примере использования двух популярных приложений, `treq` и `klein`, демонстрируются сложности, возникающие при реализации веб-API с Twisted, и способы их преодоления.

Вторая часть книги посвящена конкретным проектам, использующим Twisted. В число примеров входят использование Twisted с Docker, применение Twisted в поликонтейнера WSGI, организация обмена файлами и многое другое.

Читатель должен иметь некоторый опыт работы с Python и понимать основы контейнеров и протоколов. Знакомство с Twisted и с проектами, описанными в книге, не требуется.

УДК 004.438
ББК 32.973.22

Authorized Russian translation of the English edition of Expert Twisted ISBN 978-1-4842-3741-0 © 2019 Moshe Zadka, Mark Williams, Cory Benfield, Brian Warner, Dustin Mitchell, Kevin Samuel, Pierre Tardy.

This translation is published and sold by permission of Packt Publishing, which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-4842-3741-0 (англ.)

© 2019 Moshe Zadka, Mark Williams, Cory Benfield,
Brian Warner, Dustin Mitchell, Kevin Samuel, Pierre Tardy

ISBN 978-5-97060-795-4 (рус.)

© Оформление, издание, перевод, ДМК Пресс, 2020

*Посвящается AZ, NZ и TS: Twisted добился признания,
и мы с нетерпением ждем следующего поколения
разработчиков.*

– Моше Задка (Moshe Zadka)

Содержание

Об авторах	12
Благодарности	14
Введение	15
От издательства	16
Часть I. ОСНОВЫ	17
Глава 1. Введение в событийно-ориентированное программирование с помощью Twisted	18
Примечание о версиях Python	19
Событийно-ориентированное программирование – что это?.....	19
Многократные события.....	20
Мультиплексирование и демultipлексирование	22
Мультиплексор select.....	23
История, аналоги и назначение	23
Сокеты и select	24
События сокета – как, что и почему	25
Обработка событий	26
Цикл обработки событий с select	27
Управляемые событиями клиенты и серверы.....	29
Неблокирующий ввод/вывод	31
Знаем, когда нужно остановиться	31
Отслеживание состояния	32
Наличие информации о состоянии усложняет программы	35
Управление сложностью с помощью транспортов и протоколов	36
Реакторы: работа с транспортом.....	37
Транспорты: работа с протоколами	37
Игра в пинг-понг с протоколами и транспортами	38
Клиенты и серверы со своими реализациями протоколов и транспортов	42
Реакторы Twisted и протоколы с транспортами	43
Преимущества событийно-ориентированного программирования	44
Twisted и реальный мир.....	46
События и время.....	50
Повторение событий с LoopingCall	53
Интерфейсы событий в zope.interface.....	55

Управление потоком в событийно-ориентированных программах	57
Управление потоком в Twisted с помощью производителей и потребителей	58
Активные производители	59
Потребители	61
Пассивные производители	64
Итоги	64

Глава 2. Введение в асинхронное программирование с Twisted

Обработчики событий и их композиция	66
Что такое асинхронное программирование?	69
Заполнители для будущих значений	70
Асинхронная обработка исключений	72
Введение в Twisted Deferred	76
Обычные обработчики	76
Ошибки и их обработчики	77
Композиция экземпляров Deferred	80
Генераторы и inlineCallbacks	83
yield	83
send	84
throw	86
Асинхронное программирование с inlineCallbacks	87
Сопрограммы в Python	89
Сопрограммы с выражением yield from	90
Сопрограммы async и await	91
Ожидание завершения экземпляров Deferred	96
Преобразование сопрограмм в Deferred с помощью ensureDeferred	97
Мультиплексирование объектов Deferred	98
Тестирование объектов Deferred	102
Итоги	105

Глава 3. Создание приложений с библиотеками treq и Klein

Насколько важную роль играют эти библиотеки?	107
Агрегирование каналов	108
Введение в treq	109
Введение в Klein	112
Klein и Deferred	113
Механизм шаблонов Plating в Klein	115
Первая версия агрегатора каналов	117
Разработка через тестирование с использованием Klein и treq	123
Выполнение тестов на устанавливаемом проекте	123
Тестирование Klein с помощью StubTreq	126
Тестирование treq с помощью Klein	133

Журналирование с использованием twisted.logger.....	136
Запуск приложений Twisted с помощью twist.....	141
Итоги	144
Часть II. ПРОЕКТЫ	146
Глава 4. Twisted в Docker	147
Введение в Docker	147
Контейнеры.....	147
Образы контейнеров	148
runc и containerd	149
Клиент	149
Реестр	150
Сборка	150
Многоступенчатая сборка.....	151
Python в Docker	153
Варианты развертывания	153
В виртуальном окружении.....	157
В формате Pex	159
Варианты сборки	160
Один большой образ.....	160
Копирование пакетов wheel между этапами.....	161
Копирование окружения между этапами	161
Копирование файлов Pex между этапами.....	161
Автоматизация с использованием Dockerpy	161
Twisted в Docker	162
ENTRYPOINT и PID 1.....	162
Пользовательские плагины.....	162
NColony.....	162
Итоги	165
Глава 5. Использование Twisted в роли сервера WSGI	166
Введение в WSGI	166
PEP	167
Простой пример.....	168
Базовая реализация.....	170
Пример WebOb	172
Пример Pyramid	173
Начало	174
Сервер WSGI.....	174
Поиск кода.....	177
Путь по умолчанию	177
PYTHONPATH	177

setup.py	177
Почему Twisted?	178
Промышленная эксплуатация и разработка	178
TLS	179
Индикация имени сервера	180
Статические файлы	182
Модель ресурсов	182
Чисто статические ресурсы	183
Комбинирование статических файлов с WSGI	185
Встроенное планирование задач	186
Каналы управления	189
Стратегии параллельного выполнения	191
Балансировка нагрузки	191
Открытие сокета в режиме совместного использования	192
Другие варианты	195
Динамическая конфигурация	195
Приложение Pyramid с поддержкой A/B-тестирования	195
Плагин для поддержки AMP	197
Управляющая программа	200
Итоги	201

Глава 6. Tahoe-LAFS: децентрализованная файловая

система	202
Как работает Tahoe-LAFS	203
Архитектура системы	206
Как система Tahoe-LAFS использует Twisted	208
Часто встречающиеся проблемы	208
Инструменты поддержки выполнения в режиме демона	209
Внутренние интерфейсы FileNode	210
Интеграция интерфейсных протоколов	211
Веб-интерфейс	212
Типы файлов, Content-Type, /named/	214
Сохранение на диск	215
Заголовки Range	215
Преобразование ошибок на возвращающей стороне	216
Отображение элементов пользовательского интерфейса: шаблоны Nevow	217
Интерфейс FTP	218
Интерфейс SFTP	223
Обратная несовместимость Twisted API	223
Итоги	226
Ссылки	226

Глава 7. Magic Wormhole	227
Как это выглядит	228
Как это работает	229
Сетевые протоколы, задержки передачи, совместимость клиентов	231
Сетевые протоколы и совместимость клиентов	231
Архитектура сервера	234
База данных	235
Транзитный клиент: отменяемые отложенные операции	235
Сервер транзитной ретрансляции	238
Архитектура клиента	239
Отложенные вычисления и конечные автоматы, одноразовый наблюдатель	240
Одноразовые наблюдатели	243
Promise/Future и Deferred	244
Отсроченные вызовы, синхронное тестирование	247
Асинхронное тестирование с объектами Deferred	248
Синхронное тестирование с объектами Deferred	249
Синхронное тестирование и отсроченный вызов	252
Итоги	254
Ссылки	254
 Глава 8. Передача данных в браузерах и микросервисах с использованием WebSocket	 255
Нужен ли протокол WebSocket?	255
WebSocket и Twisted	256
WebSocket, из Python в Python	258
WebSocket, из Python в JavaScript	261
Более мощная поддержка WebSocket в WAMP	263
Итоги	269
 Глава 9. Создание приложений с asyncio и Twisted	 271
Основные понятия	271
Механизм обещаний	272
Циклы событий	273
Рекомендации	274
Пример: прокси с aiohttp и treq	277
Итоги	280
 Глава 10. Buildbot и Twisted	 282
История появления Buildbot	282
Эволюция асинхронного выполнения кода на Python в Buildbot	283
Миграция синхронных API	286
Этапы асинхронной сборки	287

Код Buildbot	287
Асинхронные утилиты	288
«Дребезг»	288
Асинхронные службы	288
Кеш LRU	291
Отложенный вызов функций	291
Взаимодействие с синхронным кодом	292
SQLAlchemy	292
requests	293
Docker	295
Конкурентный доступ к общим ресурсам	296
yield как барьер конкуренции	296
Функции из пула потоков не должны изменять общее состояние	297
Блокировки Deferred	298
Тестирование	298
Имитации	300
Итоги	300
Глава 11. Twisted и HTTP/2	301
Введение	301
Цели и задачи	303
Бесшовная интеграция	303
Оптимизация поведения по умолчанию	304
Разделение задач и повторное использование кода	305
Проблемы реализации	306
Что такое соединение? Ценность стандартных интерфейсов	306
Мультиплексирование и приоритеты	309
Противодавление	315
Противодавление в Twisted	317
Противодавление в HTTP/2	319
Текущее положение дел и возможность расширения в будущем	321
Итоги	322
Глава 12. Twisted и Django Channels	323
Введение	323
Основные компоненты Channels	325
Брокеры сообщений и очереди	325
Распределенные многоуровневые системы в Twisted	327
Текущее положение дел и возможность расширения в будущем	328
Итоги	329
Предметный указатель	330

Об авторах

Марк Уильямс (Mark Williams) работает в Twisted. В eBay и PayPal Марк Уильямс работал над высокопроизводительными веб-службами Python (более миллиарда запросов в день!), над обеспечением безопасности приложений и информации, а также переносом корпоративных Python-библиотек на Python.

Кори Бенфилд (Cory Benfield) – разработчик Python с открытым исходным кодом, активно участвует в сообществе Python HTTP. Входит в число основных разработчиков проектов Requests и urllib3 и является ведущим сопровождающим проекта Hyper – коллекции инструментов поддержки HTTP и HTTP/2 для Python, а также помогает в разработке Python Cryptographic Authority для PyOpenSSL.

Брайан Уорнер (Brian Warner) – инженер по безопасности и разработчик программного обеспечения, работавший в Mozilla на Firefox Sync, Add-On SDK и Persona. Один из основателей проекта Tahoe-LAFS – распределенной и защищенной файловой системы, разрабатывает средства безопасного хранения и связи.

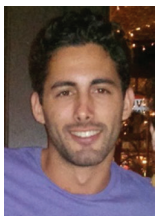
Моше Задка (Moshe Zadka) с 1995 года является участником сообщества открытого исходного кода, свой первый вклад в Python внес в 1998 году, один из основателей открытого проекта Twisted. Любит рассказывать о Twisted и Python и выступать на конференциях. Регулярно ведет блоги.

Дастин Митчелл (Dustin Mitchell) внес свой вклад в Buildbot. Является членом команды TaskCluster в Mozilla. Также работает в командах Release Engineering, Release Operations и Infrastructure.

Кевин Сэмюэл (Kevin Samuel) начал заниматься разработкой и преподаванием еще во времена, когда появилась версия Python 2.4. Работал в Восточной Европе, Северной Америке, Азии и Западной Африке. Тесно сотрудничает с командой Crossbar.io и является активным членом французского сообщества Python.

О технических рецензентах

Пьер Тарди (Pierre Tardy) – специалист по непрерывной интеграции в Renault Software Labs, в настоящее время является ведущим коммитером в Buildbot.



Джулиан Берман (Julian Berman) – разработчик программного обеспечения с открытым исходным кодом из Нью-Йорка. Автор библиотеки jsonschema для Python, периодически вносит вклад в экосистему Twisted, активный участник сообщества Python.

Шон Шоджи (Shawn Shojaie) живет в районе Калифорнийского залива, где работает инженером-программистом. Работал в Intel, NetApp. Сейчас работает в SimpleLegal, где создает веб-приложения для юридических фирм. В будние дни занимается разработкой с использованием Django и PostgreSQL, а в выходные вносит вклад в проекты с открытым исходным кодом, такие как django-pylint, и время от времени пишет технические статьи. Больше можно узнать на сайте shawnshojaie.com.

Том Мост (Tom Most) – Twisted-коммитер с десятилетним опытом разработки веб-служб, клиентских библиотек и приложений командной строки с использованием Twisted. Инженер-программист в телекоммуникационной отрасли. Сопровождает Afkak, клиента Twisted Kafka. Его адрес в интернете – freecog.net, а связаться можно по адресу twm@freecog.net.

Благодарности

Благодарю свою жену Дженнифер Задка (Jennifer Zadka), без чьей поддержки я бы не справился.

Благодарю своих родителей Якова и Пнине Задка (Yaacov and Pnina Zadka), которые хорошо меня учили.

Благодарю своего советника Яэля Каршону (Yael Karshon) за то, что он научил меня писать.

Благодарю Махмуда Хашеми (Mahmoud Hashemi) за вдохновение и поддержку.

Благодарю Марка Уильямса (Mark Williams) за то, что всегда был рядом со мной.

Благодарю Глифа Лефковица (Glyph Lefkowitz), который познакомил меня с Python, учил программировать и вообще быть хорошим человеком.

– *Моше Задка (Moshe Zadka)*

Благодарю Махмуда Хашеми (Mahmoud Hashemi) и Давида Карапетяна (David Karapetyan) за их отзывы. Благодарю Энни (Annie) за то, что была терпелива, пока я писал.

– *Марк Уильямс (Mark Williams)*

Введение

Twisted недавно отпраздновал свой шестнадцатый день рождения. За время своего существования он превратился в мощную библиотеку. За этот период на основе Twisted было создано несколько интересных приложений и многие из нас узнали много нового о том, как правильно использовать Twisted, как думать о сетевом коде и как создавать программы, основанные на событиях.

После ознакомления с вводными материалами, которые есть на сайте Twisted, часто можно услышать: «И что делать дальше? Как я могу узнать больше о Twisted?» На этот вопрос мы обычно отвечали встречным вопросом: «А что вы хотите сделать с Twisted?» В этой книге мы покажем некоторые интересные приемы использования Twisted.

Авторы данной книги использовали Twisted для разных целей и усвоили разные уроки. Мы рады представить все эти уроки, чтобы сообщество их знало и могло воспользоваться.

Вперед!

От издательства

Отзывы и пожелания

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

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

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

Скачивание исходного кода примеров

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

Список опечаток

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

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

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Часть I



ОСНОВЫ

Глава 1

Введение в событийно-ориентированное программирование с помощью Twisted

Twisted – это мощная, хорошо протестированная, развитая параллельная сетевая библиотека и фреймворк. Как мы в этой книге увидим далее, многие организации и отдельные люди при создании своих проектов эффективно использовали этот фреймворк на протяжении более чем десятилетия.

В то же время Twisted большой, сложный и старый. Его лексикон изобилует странными названиями, такими как «реактор», «протокол», «конечная точка» и «отложенный». Эти понятия описывают философию и архитектуру, сбивающую с толку как новичков, так и людей с многолетним опытом работы с Python.

Сетевые приложения, создаваемые с помощью Twisted, основываются на двух основных принципах программирования: *событийного программирования* и *асинхронного программирования*. Рост популярности языка программирования JavaScript и включение в стандартную библиотеку Python пакета `asyncio` привели к тому, что оба принципа стали основой фреймворка. Но ни один из принципов не занимает настолько доминирующего положения в программировании на Python, чтобы простого знания языка было достаточно для их освоения. Эти сложные темы доступны, пожалуй, только программистам со средним или высоким уровнем подготовки.

В этой и следующей главах представлена мотивация, лежащая в основе событийно-управляемого и асинхронного программирования, а затем показано, как Twisted использует эти принципы. Здесь закладывается основа для последующих глав, в которых рассматриваются реальные программы Twisted.

Сначала мы познакомимся с идеей событийно-ориентированного программирования без привязки к Twisted. Затем, получив представление о том, что определяет событийно-управляемое программирование, мы познакомимся с программными абстракциями в Twisted, помогающими разработчикам писать четкие и эффективные событийные программы. Попутно мы будем делать остановки, чтобы познакомиться с некоторыми уникальными частями этих абстракций, например *интерфейсами*, и исследуем документацию с их описанием на веб-сайте Twisted.

К концу этой главы вы освоите терминологию Twisted: протоколы, транспорты, реакторы, потребители и производители. Данные понятия образуют основу событийно-ориентированного программирования с Twisted. Знание этой основы необходимо для разработки полезного программного обеспечения с Twisted.

ПРИМЕЧАНИЕ О ВЕРСИЯХ PYTHON

Twisted поддерживает Python 2 и 3, поэтому все примеры кода в этой главе могут работать как с Python 2, так и с Python 3. Python 3 – это будущее, но одной из сильных сторон Twisted является его богатая история реализации протоколов; по этой причине важно, чтобы вы умели писать код, способный выполняться под управлением Python 2, даже если прежде вы никогда не писали его.

СОБЫТИЙНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ – ЧТО ЭТО?

Событие – это то, что заставляет управляемую событиями программу выполнить действие. Это широкое определение позволяет рассматривать многие программы как управляемые событиями. Рассмотрим, например, простую программу, которая в зависимости от ввода пользователя печатает либо Hello, либо World:

```
import sys
line = sys.stdin.readline().strip()
if line == "h":
    print("Hello")
else:
    print("World")
```

Появление строки в потоке стандартного ввода является событием. Наша программа приостанавливается на операторе `sys.stdin.readline()`, который просит операционную систему разрешить пользователю ввести законченную строку. Пока ввод не будет получен, программа дальше двигаться не будет. Когда операционная система прочитает очередной ввод и внутренние компоненты Python определят, что строка завершена, оператор `sys.stdin`.

`readline()` возобновит работу программы, вернув ей полученные данные. Возобновление работы – это событие, толкающее нашу программу вперед. Поэтому даже такую простую программу можно представить как управляемую событиями.

МНОГОКРАТНЫЕ СОБЫТИЯ

Программа, обрабатывающая единственное событие и завершающая работу, не получает никаких преимуществ от событийно-ориентированного подхода. Программы, в которых одновременно может происходить несколько событий, имеют более естественную организацию для их обработки. Примером такой программы может быть графический интерфейс пользователя: в любой момент пользователь может нажать кнопку, выбрать пункт из меню, прокрутить текст и т. д.

Вот версия нашей предыдущей программы с графическим интерфейсом Tkinter:

```
from six.moves import tkinter
from six.moves.tkinter import scrolledtext
class Application(tkinter.Frame):
    def __init__(self, root):
        super(Application, self). __init__(root)
        self.pack()
        self.helloButton = tkinter.Button(self,
                                           text="Say Hello",
                                           command=self.sayHello)
        self.worldButton = tkinter.Button(self,
                                           text="Say World",
                                           command=self.sayWorld)
        self.output = scrolledtext.ScrolledText(master=self)
        self.helloButton.pack(side="top")
        self.worldButton.pack(side="top")
        self.output.pack(side="top")
    def outputLine(self, text):
        self.output.insert(tkinter.INSERT, text+ '\n')
    def sayHello(self):
        self.outputLine("Hello")
    def sayWorld(self):
        self.outputLine("World")

Application(tkinter.Tk()).mainloop()
```

Эта версия нашей программы представляет пользователю две кнопки, каждая из которых может генерировать независимое событие (click) и отличается от предыдущей программы тем, что в предыдущей программе `sys.stdin.readline` может генерировать только одно событие: «готовность строки».

С этими событиями мы связали *обработчики событий*, которые вызываются нажатием любой кнопки. Кнопки Tkinter имеют свойство `command` со ссылкой

на обработчик, и вызывают этот обработчик при нажатии. Так, когда нажимается кнопка **Say Hello**, она генерирует событие, вынуждающее программу вызвать функцию `Application.sayHello`, которая, в свою очередь, выводит слово `Hello` в текстовое окно с прокруткой. То же происходит при нажатии кнопки **Say World**, которая вызывает функцию `Application.sayWorld` (см. рис. 1.1).



Рис. 1.1 ❖ Приложение Tkinter GUI
после нескольких нажатий кнопок **Say Hello** и **Say World**

Метод `tkinter.Frame.mainloop`, унаследованный нашим классом `Application`, ждет, пока связанная с ним кнопка сгенерирует событие, и запускает соответствующий обработчик. После выполнения каждого обработчика `tkinter.Frame.mainloop` переходит к ожиданию новых событий. Цикл ожидания событий и их передачи в соответствующие обработчики типичен для управляемых событиями программ и известен как *цикл событий*.

Вот основные понятия, лежащие в основе событийно-ориентированного программирования.

1. *События* показывают, что произошло то, на что должна реагировать программа. В обоих наших примерах события естественно соответствуют нажатиям кнопок, но, как мы увидим, они могут представлять все, что заставляет нашу программу выполнять какое-либо действие.
2. *Обработчики событий* определяют реакцию программы на события. Иногда обработчик события имеет вид простой инструкции, как вызов `sys.stdin.readline` в нашем примере. Но чаще всего он представлен функцией или методом, как в нашем примере `tkinter`.

3. *Цикл событий* ждет появления события и вызывает соответствующий обработчик. Не все управляемые событиями программы имеют цикл событий; в нашем первом примере `sys.stdin.readline` цикла обработки событий нет, потому что здесь происходит только одно событие. Однако большинство программ напоминают наш пример `tkinter` тем, что перед окончательным завершением обрабатывают много событий. Такие программы используют цикл событий.

МУЛЬТИПЛЕКСИРОВАНИЕ И ДЕМУЛЬТИПЛЕКСИРОВАНИЕ

Способ ожидания появления событий в цикле существенно влияет на написание управляемых событиями программ, поэтому рассмотрим его более подробно. Вернемся к нашему примеру `tkinter` с двумя кнопками. Цикл событий внутри `mainloop` должен ждать, пока пользователь не нажмет одну из двух кнопок.

Простая реализация цикла может выглядеть следующим образом:

```
def mainloop(self):
    while self.running:
        ready = [button for button in self.buttons if button.hasEvent()]
        if ready:
            self.dispatchButtonEventHandlers(ready)
```

Цикл `mainloop` в ожидании нового события постоянно *опрашивает* каждую кнопку и запускает обработчики только для имеющих готовое событие. Когда событий нет, программа никаких действий не выполняет, так как никаких действий, требующих ответа, предпринято не было. В течение периодов бездействия управляемая событиями программа свое выполнение должна приостанавливать.

Цикл `while` в нашем примере `mainloop` приостанавливает работу программы до тех пор, пока не будет нажата одна из кнопок и не понадобится вызвать функцию `sayHello` или `sayWorld`. Если пользователь не сможет щелкнуть мышью со сверхъестественной скоростью, большая часть времени цикла будет тратиться на проверку кнопок, которые не были нажаты. Это называется *активным ожиданием*.

Активное ожидание, подобное этому, приостанавливает общее выполнение программы до тех пор, пока один из ее источников событий не сообщит о том, что событие произошло. Поэтому активного ожидания достаточно для приостановки цикла событий.

Внутренний генератор списков, управляющий активным ожиданием, задает важный вопрос: что-нибудь произошло? Ответ помещается в переменную `ready` и имеет вид списка со всеми кнопками, которые были нажаты. Истинность переменной `ready` определяет ответ на вопрос: если переменная `ready` ничего не содержит (имеет ложное значение), значит, кнопки не нажи-

мались и ничего не произошло. Если переменная имеет истинное значение, следовательно, событие произошло и по меньшей мере одна из кнопок была нажата.

Генератор списков, формирующий значение для `ready`, объединяет множество событий в одном месте. Этот процесс известен как *мультиплексирование*. Обратный процесс разделения списка на отдельные события называется *демультиплексированием*. Генератор списка объединяет (мультиплексирует) все наши кнопки в переменной `ready`, тогда как метод `dispatchButtonEventHandlers` демультиплексирует (перенаправляет сигнал с одного из информационных входов на один из информационных выходов) их, вызывая обработчик каждого события отдельно.

Теперь мы можем уточнить наше понимание циклов событий, точно описав, как происходит ожидание событий.

- *Цикл событий* ожидает появления событий, мультиплексируя их источники в один список. Если получился непустой список, цикл событий демультиплексирует его и для каждого события вызывает соответствующий обработчик.

Наш мультиплексор `mainloop` тратит большую часть своего времени на опрос кнопок, которые не были нажаты. Но не все мультиплексоры настолько неэффективны. В `tkinter.Frame.mainloop` используется аналогичный мультиплексор, который опрашивает все виджеты, если операционная система не предлагает более эффективного способа. Мультиплексор `mainloop`, чтобы повысить свою эффективность, использует тот факт, что компьютеры могут проверять виджеты GUI быстрее, чем с ними может взаимодействовать человек, и вставляет вызов `sleep`, приостанавливающий программу на несколько миллисекунд. Это позволяет программе часть времени в цикле активного ожидания вообще не выполнять никаких операций и за счет небольшой задержки экономить процессорное время и электроэнергию.

Хоть `Twisted` может интегрироваться с графическими пользовательскими интерфейсами и фактически имеет специальную поддержку `tkinter`, в его основе лежит сетевой движок. В сети основными источниками событий являются *сокеты*, а не кнопки, и операционные системы предлагают эффективные механизмы для мультиплексирования событий сокетов. Цикл событий в `Twisted` использует эти механизмы. Чтобы понять подход `Twisted` к событийному программированию, нужно разобраться, как взаимодействуют сокеты с этими механизмами мультиплексирования.

МУЛЬТИПЛЕКСОР SELECT

История, аналоги и назначение

Мультиплексор `select` поддерживается большинством операционных систем. Свое имя «`select`» (выбор) этот мультиплексор получил из-за своей способно-

сти выбирать из всего списка сокетов только те, у которых есть готовые к обработке события.

Мультиплексор `select` появился в 1983 году, когда возможности компьютеров были гораздо скромнее. Как следствие, он имеет не лучшую эффективность, особенно при мультиплексировании большого количества сокетов. Каждое семейство операционных систем предоставляет свой, более эффективный мультиплексор, такой как `kqueue` в BSD и `epoll` в Linux, но между собой они не взаимодействуют. К счастью, эти мультиплексоры имеют схожий принцип работы, и мы можем обобщить их поведение. Используем для иллюстрации мультиплексор `select`.

Сокеты и `select`

В следующем коде отсутствует обработка ошибок, поэтому он будет терпеть неудачу во многих пограничных случаях, встречающихся на практике. **Этот код приводится исключительно в демонстрационных целях. Не используйте его в реальных приложениях.** Twisted стремится правильно обрабатывать ошибки и крайние случаи, поэтому имеет довольно сложную реализацию.

Теперь, предупредив вас об опасностях, начнем интерактивный сеанс Python и создадим сокеты для мультиплексирования с использованием `select`:

```
>>> import socket
>>> listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> listener.bind(('127.0.0.1', 0))
>>> listener.listen(1)
>>> client = socket.create_connection(listener.getsockname())
>>> server, _ = listener.accept()
```

Полное объяснение сокетов выходит за рамки этой книги. Фактически мы надеемся, что после обсуждения деталей вы предпочтете Twisted! Однако приведенный выше код содержит больше фундаментальных понятий, чем несущественных деталей:

- 1) `listener` (прослушиватель) – сокет, предназначенный для приема входящих соединений. Это сокет интернета (`socket.AF_INET`) и TCP (`socket.SOCK_STREAM`), доступный клиентам через внутренний локальный сетевой интерфейс (который обычно имеет адрес `127.0.0.1`) и порт, случайно выбираемый операционной системой (`0`). Данный прослушиватель выполняет настройку входящего соединения и ставит его в очередь до тех пор, пока оно не будет прочитано (`listen(1)`);
- 2) `client` (клиент) – сокет исходящего соединения. Функция `socket.create_connection` принимает кортеж (хост, порт), представляющий прослушивающий сокет, к которому требуется подключиться, и возвращает подключенный сокет. Поскольку наш сокет `listener` находится в том же процессе, мы можем получить его хост и порт с помощью `listener.getsockname()`;

- 3) `server` (сервер) – входящее соединение на сервере. После подключения клиента к нашему хосту и порту необходимо принять соединение из очереди `listener`. `listener.accept` возвращает кортеж (сокет, адрес). Так как нам нужен только сокет, адрес мы отбрасываем. Реальная программа может записать адрес в журнал или использовать его для отслеживания метрик подключения. Очередь прослушивателя с длиной 1, которую мы создали вызовом метода `listen`, хранит этот сокет, пока мы не вызовем `accept`, что позволит функции `create_connection` вернуть управление.

`client` (клиент) и `server` (сервер) – это два конца одного TCP-соединения. Установленное TCP-соединение не различает клиента и сервер; клиентский сокет обладает теми же правами на чтение, запись или закрытие соединения, что и серверный:

```
>>> data = b"xyz"
>>> client.sendall(data)
>>> server.recv(1024) == data
True
>>> server.sendall(data)
>>> client.recv(1024) == data
True
```

События сокета – как, что и почему

Внутри операционной системы для каждого сокета TCP поддерживаются буферы чтения и записи для учета ненадежности сети, а также клиентов и серверов с различной скоростью записи-чтения. Если сервер временно не доступен и не может получать данные `b"xyz"`, переданные в `client.sendall`, то они останутся в буфере записи до тех пор, пока сервер снова не станет активным. Точно так же, если клиент слишком занят и не смог вовремя вызвать `client.recv` и получить данные `b"xyz"`, отправленные вызовом `server.sendall`, они будут храниться в буфере чтения клиента, пока тот не получит их. Число, которое мы передаем `recv`, представляет максимальное количество данных, которые мы готовы удалить из буфера чтения. Если объем данных в буфере чтения меньше указанной величины, как в данном примере, `recv` удалит из буфера и возвратит *все* данные.

Двунаправленность наших сокетов подразумевает два возможных события:

- 1) *событие готовности к чтению* означает, что сокет получил какие-то данные и их можно прочитать. Серверный сокет генерирует это событие, когда в его приемный буфер попадают какие-то данные, то есть вызов `recv` сразу после события готовности к чтению немедленно вернет эти данные. Если данных не окажется в буфере, произойдет разъединение. Кроме того, прослушивающий сокет генерирует это событие, когда мы можем принять новое соединение;
- 2) *событие доступности к записи* означает, что в буфере записи сокета есть свободное место. Это тонкий момент: пока сервер успевает обрабаты-

вать данные быстрее, чем мы добавляем их в буфер отправки клиентского сокета, этот буфер остается доступным для записи.

Интерфейс `select` отражает эти возможные события и принимает до четырех аргументов:

- 1) список сокетов для мониторинга *событий готовности к чтению*;
- 2) список сокетов для мониторинга *событий доступности для записи*;
- 3) список сокетов для мониторинга «исключительных событий». В наших примерах исключительных событий не предполагается, поэтому мы всегда будем передавать пустой список;
- 4) необязательный *тайм-аут*. Это количество секунд, в течение которых `select` будет ждать, когда хотя бы один из подконтрольных сокетов сгенерирует событие. Если опустить этот аргумент, `select` будет ждать вечно.

Мы можем спросить у `select`, какие события сгенерировали наши сокеты:

```
>>> import select
>>> maybeReadable = [listener, client, server]
>>> maybeWritable = [client, server]
>>> readable, writable, _ = select.select(maybeReadable, maybeWritable, [], 0)
>>> readable
[]
>>> writable == maybeWritable and writable == [client, server]
True
```

Мы потребовали от `select` не ждать новых событий, передав тайм-аут 0. Как объяснялось выше, сокеты `client` и `server` могут быть доступны как для чтения, так и для записи, а сокет `listener` может только принимать входящие соединения, то есть может генерировать лишь события готовности к чтению.

Если опустить аргумент с тайм-аутом, `select` приостановит нашу программу до тех пор, пока один из контролируемых сокетов не станет доступным для чтения или записи. Приостановка выполнения производится так же, как в примере мультиплексирования с циклом активного ожидания, опрашивающим все кнопки в нашей упрощенной реализации `mainloop`.

Вызов `select` мультиплексирует сокеты более эффективно, чем активное ожидание, потому что операционная система возобновит нашу программу только в том случае, когда будет сгенерировано хотя бы одно событие. Внутри ядра цикл событий, в отличие от нашего `select`, ожидает событий от сетевого оборудования и затем передает их нашему приложению.

Обработка событий

`select` возвращает кортеж с тремя списками в том же порядке, в котором расположены его аргументы. Обход каждого списка *демультиплексирует* возвращаемое значение `select`. Ни один из наших сокетов не сгенерировал события готовности к чтению, хотя мы записывали данные и в `client`, и в `server`. Наши предыдущие вызовы `recv` очистили их буферы чтения, и к `listener` никаких

новых подключений не происходило с тех пор, как было принято соединение с клиентом и создан сокет `server`. Однако в буферах отправки `client` и `server` есть свободное место, поэтому они оба сгенерировали событие доступности для записи.

Отправка данных с `client` на `server` приводит к тому, что `server` генерирует событие готовности к чтению, поэтому `select` помещает его в список `readable`:

```
>>> client.sendall(b'xyz')
>>> readable, writable, _ = select.select(maybeReadable, maybeWritable, [], 0)
>>> readable == [server]
True
```

Список доступных для записи сокетов, что интересно, снова содержит наши сокеты `client` и `server`:

```
>>> writable == maybeWritable and writable == [client, server]
True
```

Если еще раз вызвать `select`, сокет `server` снова будет готов к чтению, и оба сокета – `client` и `server` – опять будут доступны для записи. Причина проста: пока данные остаются в буфере чтения сокета, для него непрерывно будет генерироваться событие готовности к чтению, а пока в передающем буфере сокета остается место, для него будет генерироваться событие доступности для записи. Мы можем это подтвердить, прочитав данные, полученные сокетом `server`, и снова вызвав `select`:

```
>>> server.recv(1024) == b'xyz'
True
>>> readable, writable, _ = select.select(maybeReadable, maybeWritable,
[], 0)
>>> readable
[]
>>> writable == maybeWritable and writable == [client, server]
True
```

Очистка буфера чтения сервера привела к тому, что он перестал генерировать событие готовности к чтению, но оба сокета – `client` и `server` – продолжают генерировать событие доступности для записи, потому что в их буферах записи еще есть место.

Цикл обработки событий с `select`

Теперь мы знаем, как `select` мультиплексирует сокеты:

- 1) различные сокеты генерируют готовности к чтению/записи, чтобы сообщить управляемой событиями программе, что та может принять входящие данные или соединения или записать исходящие данные;
- 2) `select` мультиплексирует сокеты, проверяя их готовность к чтению/записи, и приостанавливает программу, пока не появится хотя бы одно событие или не истечет тайм-аут;

- 3) сокеты продолжают генерировать события готовности к чтению/записи до тех пор, пока не изменятся обстоятельства, приведшие к этим событиям: сокет с данными в буфере чтения продолжает генерировать событие готовности к чтению, пока буфер не опустеет; прослушивающий сокет продолжает генерировать событие готовности к чтению, пока все входящие соединения не будут приняты; а сокет, в буфере отправки которого еще есть место, будет генерировать события доступности для записи, пока буфер не заполнится.

Опираясь на эти знания, мы можем набросать цикл событий с `select`:

```
import select

class Reactor(object):
    def __init__(self):
        self._readers = {}
        self._writers = {}
    def addReader(self, readable, handler):
        self._readers[readable] = handler
    def addWriter(self, writable, handler):
        self._writers[writable] = handler
    def removeReader(self, readable):
        self._readers.pop(readable, None)
    def removeWriter(self, writable):
        self._writers.pop(writable, None)
    def run(self):
        while self._readers or self._writers:
            r, w, _ = select.select(list(self._readers), list(
                self._writers), [])
            for readable in r:
                self._readers[readable](self, readable)
            for writable in w:
                if writable in self._writers:
                    self._writers[writable](self, writable)
```

Мы назвали наш цикл событий *реактором*, потому что он реагирует на события сокета. Мы можем попросить `Reactor` запускать наши обработчики событий готовности к чтению, вызывая его метод `addReader`, и обработчики событий доступности для записи, вызывая `addWriter`. Обработчики событий принимают два аргумента: сам реактор и сокет, который сгенерировал событие.

Цикл внутри метода `run` мультиплексирует сокеты с помощью `select`, а затем демultipлексирует результат на сокеты, готовые к чтению и доступные для записи. Сначала вызываются обработчики для сокетов, готовых к чтению. Далее, прежде чем запустить обработчик события доступности для записи, цикл событий проверяет, зарегистрирован ли по-прежнему сокет в словаре `_writers`. Это важно, потому что закрытие соединения генерирует событие готовности к чтению, и запущенный непосредственно перед этим обработчик чтения может удалить закрытый сокет из словарей `_readers` и `_writers`. К моменту запуска обработчика события доступности для записи закрытый сокет будет удален из словаря `_writers`.

Управляемые событиями клиенты и серверы

Этого простого цикла событий достаточно для реализации клиента, который постоянно записывает данные на сервер. Начнем с обработчиков событий:

```
def accept(reactor, listener):
    server, _ = listener.accept()
    reactor.addReader(server, read)
def read(reactor, sock):
    data = sock.recv(1024)
    if data:
        print("Server received", len(data), "bytes.")
    else:
        sock.close()
        print("Server closed.")
        reactor.removeReader(sock)

DATA=[b"*", b"*"]
def write(reactor, sock):
    sock.sendall(b"*.join(DATA))
    print("Client wrote", len(DATA), " bytes.")
    DATA.extend(DATA)
```

Функция `accept` обрабатывает событие готовности к чтению сокета `listening`, принимая входящее соединение и передавая реактору соответствующий сокет для проверки наличия в нем событий готовности к чтению. Они обрабатываются функцией `read`.

Функция `read` обрабатывает событие готовности к чтению, пытаясь получить фиксированный объем данных из приемного буфера сокета. Она выводит длину принятого сообщения – напомним, что число, переданное в `recv`, представляет *верхний предел* количества возвращаемых байтов. Если в сокете, сгенерировавшем событие готовности к чтению, данные отсутствуют, значит, другая сторона соединения закрыла свой сокет. Поэтому функция `read` в ответ закрывает свой сокет и удаляет его из набора сокетов, контролируемых реактором на появление события готовности к чтению. Закрытие сокета освобождает ресурсы операционной системы, а удаление его из реактора гарантирует, что мультиплексор `select` не будет пытаться контролировать неактивный сокет.

Функция `write` записывает последовательность звездочек (*) в сокет, сгенерировавший событие доступности для записи. После каждой успешной записи объем данных удваивается. Это имитирует поведение реальных сетевых приложений, которые не всегда записывают в соединение одинаковый объем данных. Рассмотрим веб-браузер: некоторые исходящие запросы содержат небольшое количество данных из формы, введенных пользователем, в то время как другие могут выгружать огромные файлы.

Обратите внимание, что это функции уровня модуля, а не методы нашего класса `Reactor`. Вместо этого они ассоциируются с реактором путем их регистрации в качестве обработчиков событий доступности для чтения/записи, потому что TCP-сокеты – это только один вид сокетов и события в других сокетах

может потребоваться обрабатывать иначе. Однако сама функция `select` обрабатывает все сокеты одинаково, поэтому логика вызова обработчиков событий для сокетов в возвращаемых списках должна инкапсулироваться классом `Reactor`. Позже мы рассмотрим, насколько для программ, управляемых событиями, важны инкапсуляция и интерфейсы, которые она подразумевает.

Теперь можно создать сокеты прослушвателя `listener` и клиента `client` и позволить циклу событий управлять приемом соединений и передачей данных между клиентским и серверным сокетами.

```
import socket
listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listener.bind(('127.0.0.1',0))
listener.listen(1)
client = socket.create_connection(listener.getsockname())

loop = Reactor()
loop.addWriter(client, write)
loop.addReader(listener, accept)
loop.run()
```

После запуска эта программа в какой-то момент терпит неудачу:

```
Client wrote 2 bytes.
Server received 2 bytes.
Client wrote 4 bytes.
Server received 4 bytes.
Client wrote 8 bytes.
Server received 8 bytes.
...
Client wrote 524288 bytes.
Server received 1024 bytes.
Client wrote 1048576 bytes.
Server received 1024 bytes.
^CTraceback (most recent call last):
  File "example.py", line 53, in <module>
    loop.run()
  File "example.py", line 25, in run
    writeHandler(self, writable)
  File "example.py", line 33, in write
    sock.sendall(b"".join(DATA))
KeyboardInterrupt
```

Сначала все хорошо: данные передаются из клиентского сокета в серверный. Это поведение соответствует логике обработчиков событий `accept`, `read` и `write`. Как и ожидалось, сначала клиент посылает серверу два байта `b'*`, который, в свою очередь, получает эти два байта.

Одновременная работа клиента и сервера демонстрирует эффективность событийно-ориентированного программирования. Подобно нашему графическому приложению, реагировавшему на события от двух разных кнопок, этот небольшой сетевой сервер способен откликаться на события от клиента или сервера, что позволило объединить их в одном процессе. Мультиплексирую-

щие способности `select` обеспечивают единую точку в цикле событий, где программа может реагировать на любое из них.

Но потом возникла проблема: после определенного количества повторений программа зависает, и ее приходится прерывать с помощью клавиатуры. Ключ к этому находится в выводе трассировки стека нашей программы; спустя какое-то время клиент застревает, пытаясь переслать большой объем данных серверу, как видно по трассировке стека после прерывания `KeyboardInterrupt`, ведущей прямо к вызову `sock.sendall` в нашем обработчике записи (`write`).

Сервер не поспевает за клиентом, в результате чего большую часть времени буфер отправки клиентского сокета остается заполненным до отказа. По умолчанию, если в буфере отправки нет места, функция `sendall` приостанавливает (*блокирует*) программу. Если бы `sendall` не блокировала программу и наш цикл событий был бы мог выполняться как обычно, сокет не был бы доступен для записи, и блокирующий вызов `sendall` не выполнялся бы. Однако мы не можем с уверенностью сказать, сколько данных нужно передать функции `sendall`, чтобы она заполнила буфер отправки, не заблокировав при этом программу, обработчик записи выполнялся бы до конца, и функция `select` предотвратила бы дальнейшие попытки записи, пока в буфере не освободится место. Природа сетей такова, что о подобной проблеме мы узнаем только после того, как она возникнет.

Все события, которые мы до сих пор рассматривали, заставляют программу выполнять какие-то действия. Но мы еще не уделили внимания событиям, заставляющим программу выполняемые действия *прекратить*. Нам нужен новый вид события.

НЕБЛОКИРУЮЩИЙ ВВОД/ВЫВОД

Знаем, когда нужно остановиться

По умолчанию сокеты блокируют программу, начавшую операцию, которая не может быть завершена, пока на удаленном конце не произойдет определенное действие. Для предотвращения блокировки мы можем заставить сокет выдавать событие, попросив операционную систему сделать его *неблокирующим*.

Давайте вернемся к интерактивному сеансу Python и снова построим соединение между клиентским (`client`) и серверным (`server`) сокетами. На этот раз мы сделаем сокет `client` неблокирующим и попытаемся записать в него бесконечный поток данных.

```
>>> import socket
>>> listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> listener.bind(('127.0.0.1', 0))
>>> listener.listen(1)
>>> client=socket.create_connection(listener.getsockname())
>>> server, _ = listener.accept()
```

```
>>> client.setblocking(False)
>>> while True: client.sendall(b"*"*1024)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
BlockingIOError: [Errno11] Resource temporarily unavailable
```

Мы снова заполнили буфер отправки клиента, но вместо того, чтобы приостановить процесс, `sendall` вызвала исключение. Тип исключения зависит от версии Python. В Python 3 это исключение `BlockingIOError`, а в Python 2 это общее исключение `socket.error`. В обеих версиях Python значение атрибута `errno` исключения будет установлено в `errno.EAGAIN`:

```
>>> import errno, socket
>>> try:
...     while True: client.sendall(b"*"*1024)
... except socket.error as e:
...     print(e.errno == errno.EAGAIN)
True
```

Исключение – это сгенерированное операционной системой событие, указывающее, что мы должны *прекратить* запись. Этого почти достаточно, чтобы исправить проблему в наших клиенте и сервере.

Отслеживание состояния

Однако для обработки этого исключения требуется ответить на новый вопрос: сколько данных, которые мы пытались записать, попали в буфер отправки сокета? Не ответив на этот вопрос, мы не сможем узнать, какие данные действительно были отправлены. Не зная этого, мы не сможем правильно писать программы с неблокирующими сокетами. Например, веб-браузер обязан отслеживать количество и объем выгруженных файлов. Иначе содержимое может быть повреждено в пути.

`client.sendall`, прежде чем сгенерировать исключение с ошибкой `EAGAIN`, может поместить в буфер отправки любое количество байтов. Мы должны отказаться от метода `sendall` и вместо него использовать метод `send`, который возвращает объем данных, записанных в буфер отправки сокета. Мы можем продемонстрировать это с помощью сокета `server`:

```
>>> server.setblocking(False)
>>> try:
...     while True: print(server.send(b"*" * 1024))
... except socket.error as e:
...     print("Terminated with EAGAIN:", e.errno == errno.EAGAIN)
1024
1024
...
1024
952
Terminated with EAGAIN:True
```

Мы переводим сокет `server` в неблокирующий режим, чтобы по заполнении буфера отправки он генерировал событие `EAGAIN`. Затем цикл `while` вызывает `server.send`. Вызовы, возвращающие 1024, записали все предоставленные байты в буфер отправки сокета. В конце концов, буфер записи сокета заполняется, и исключение с ошибкой `EAGAIN` завершает цикл. Однако последний успешный вызов `send` перед завершением цикла вернул 952 – он просто отбросил последнее 72 байта. Это известно как *короткая запись*. То же происходит и с блокирующими сокетами! Когда в буфере отправки не останется свободного места, `sendall` не генерирует исключение, а запускает цикл, который проверяет значение, возвращаемое вызовом `send`, и повторяет эти вызовы до тех пор, пока не будут отправлены все данные.

В нашем случае размер буфера отправки сокета не был кратным 1024, поэтому мы не смогли отправить круглое число байтов в последнем вызове `send`, прежде чем возникла ошибка `EAGAIN`. Однако в реальном мире размер буфера отправки сокета изменяется в зависимости от условий в сети, и приложения отправляют по соединениям различные объемы данных. Программы, использующие неблокирующий ввод/вывод, например как наш гипотетический веб-браузер, регулярно сталкиваются с короткими записями, подобными этим.

Используя значение, возвращаемое функцией `send`, можно гарантировать запись всех данных в соединение. Мы можем добавить свой буфер для данных, которые нужно записать. Каждый раз, когда `select` создаст для сокета событие доступности для записи, мы попытаемся отправить данные, находящиеся в этот момент в буфере. Если вызов `send` завершится без ошибки `EAGAIN`, мы запомним результат и удалим это количество байтов из начала нашего буфера, потому что `send` записывает данные в буфер отправки, выбирая их из начала полученной последовательности. Если `send` сгенерирует ошибку `EAGAIN`, указывающую, что буфер отправки полностью заполнен и не может вместить больше данных, мы оставим содержимое буфера без изменений. Так будет продолжаться, пока наш собственный буфер не опустеет, и когда это произойдет, мы будем знать, что все наши данные помещены в буфер отправки сокета. После этого операционная система должна отправить все накопленные данные на приемный конец соединения.

Теперь исправим наш простой пример клиент-сервер, разделив функцию `write` на два элемента – функцию, которая иницирует запись данных, и объект, управляющий буфером и вызывающий метод `send`:

```
import errno
import socket

class BuffersWrites(object):
    def __init__(self, dataToWrite, onCompletion):
        self._buffer = dataToWrite
        self._onCompletion = onCompletion
    def bufferingWrite(self, reactor, sock):
        if self._buffer:
```

```

    try:
        written = sock.send(self._buffer)
    except socket.error as e:
        if e.errno != errno.EAGAIN:
            raise
        return
    else:
        print("Wrote", written, "bytes")
        self._buffer = self._buffer[written:]
    if not self._buffer:
        reactor.removeWriter(sock)
        self.onCompletion(reactor, sock)

```

```

DATA=[b"*", b"*"]
def write(reactor, sock):
    writer = BuffersWrites(b"".join(DATA), onCompletion=write)
    reactor.addWriter(sock, writer.bufferingWrite)
    print("Client buffering", len(DATA), "bytes to write.")
    DATA.extend(DATA)

```

Первый аргумент метода инициализации `BuffersWrites` – это последовательность байтов для отправки, которая будет играть роль буфера. А второй аргумент, `onCompletion`, является вызываемым объектом. Как следует из названия, `onCompletion` (по завершении) будет вызван после записи всех данных в буфер отправки сокета.

Сигнатура метода `bufferingWrite` отвечает требованиям функции `Reactor.addWriter` к обработчикам событий доступности для записи. Он пытается отправить данные из буфера в сокет и запоминает число отправленных байтов. Если `send` вызывает исключение `EAGAIN`, `bufferingWrite` перехватывает это исключение и просто выходит; если вызов `send` сгенерирует какое-то другое исключение, оно продолжит распространение вверх по стеку вызовов. В обоих случаях `self._buffer` остается неизменным.

Если отправка выполнена успешно, записанное число байтов удаляется из начала буфера `self._buffer` и `bufferingWrite` завершается. Например, если вызов `send` запишет только 952 байта из 1024, `self._buffer` будет содержать последние 72 байта.

Наконец, если буфер опустел, значит, все запрошенные данные были записаны, и для экземпляра `BuffersWrites` не осталось работы. Он обращается к реактору, чтобы прекратить мониторинг своего сокета на наличие событий доступности для записи, а затем вызывает `onCompletion`, чтобы сообщить, что все данные записаны. Обратите внимание, что эта проверка выполняется в операторе `if`, который не зависит от первого оператора `if self._buffer`. Предыдущий код мог выполняться и очистить буфер; если бы заключительный код находился в блоке `else` оператора `if self._buffer`, он не был бы выполнен до следующего раза, когда реактор обнаружит, что сокет доступен для записи. Чтобы упростить управление ресурсами, мы вынесли проверку в отдельный оператор `if`.

Функция `write` выглядит аналогично предыдущей версии, за исключением

того, что теперь она делегирует отправку данных методу `bufferingWrite` объекта `BuffersWrites`. Обратите особое внимание на то, что `write` передает *себя* в `BuffersWrites` в роли вызываемого объекта `onCompletion`. Такая *косвенная рекурсия* создает тот же эффект цикла, что и в предыдущей версии. `write` никогда не вызывает себя напрямую, а передает себя объекту, который, в свою очередь, вызывается *реактором*. Эта косвенность позволяет продолжать данную последовательность без переполнения стека вызовов.

С этими изменениями наша программа больше не блокируется. Но теперь она терпит неудачу по другой причине: в какой-то момент объем данных становится слишком большим и не помещается в доступной памяти компьютера! Вот, например, что получилось на компьютере автора:

```
Client buffering 2 bytes to write.
Wrote 2 bytes
Client buffering 4 bytes to write.
Server received 2 bytes.
Wrote 4 bytes
...
Client buffering 2097152 bytes to write.
Server received 1024 bytes.
Wrote 1439354 bytes
Server received 1024 bytes.
Server received 1024 bytes.
....
Wrote 657798 bytes
Server received 1024 bytes.
Server received 1024 bytes.
....
Client buffering 268435456 bytes to write.
Traceback (most recent call last):
  File "example.py", line 76, in <module>
    loop.run()
  File "example.py", line 23, in run
    writeHandler(self, writable)
  File "example.py", line 57, in bufferingWrite
    self._onCompletion(reactor, sock)
  File "example.py", line 64, in write
    DATA.extend(DATA)
MemoryError
```

Наличие информации о состоянии усложняет программы

Несмотря на проблему, связанную с переполнением памяти компьютера, мы успешно справились с созданием управляемой событиями программы, которая для управления записью данных в сокеты использует неблокирующий ввод/вывод. Однако код получился запутанным: косвенный вызов `write` через `BuffersWrites` и реактор мешает понять логический поток исходящих данных, и очевидно, что реализация чего-то более сложного, чем отправка последовательности звездочек, потребует расширения классов и интерфейсов. Напри-

мер, как обработать исключение `MemoryError`? Наш подход непригоден для реальных приложений.

УПРАВЛЕНИЕ СЛОЖНОСТЬЮ С ПОМОЩЬЮ ТРАНСПОРТОВ И ПРОТОКОЛОВ

Программирование с неблокирующим вводом/выводом, несомненно, имеет ряд сложностей. Вот что пишет об этом администратор UNIX W. Ричард Стивенс в первом томе своей основополагающей серии *Unix Network Programming*¹:

Но будут ли оправданы затраченные усилия при написании приложения, использующего неблокируемый ввод-вывод, с учетом усложнения итогового кода? Нет, ответим мы.

(UNIX. Разработка сетевых приложений, с. 478)

Сложность нашего кода, похоже, доказывает справедливость утверждения Стивенса. Правильные абстракции, однако, могут загерметизировать сложность в управляемом интерфейсе. В нашем примере уже есть многократно используемый код: любая новая единица кода, выполняющая запись в сокет, должна использовать основную логику `BuffersWrites`. Мы *загерметизировали* сложность операции записи в неблокирующий сокет. Основываясь на этом понимании, мы можем выделить две концептуальные области:

- 1) *транспорты*: `BuffersWrites` управляет процессом записи выходных данных в неблокирующий сокет *независимо от их содержимого*. Он может пересылать фотографии, музыку – любые данные, какие только можно себе представить. Главное условие – эти данные перед передачей должны быть преобразованы в байты. `BuffersWrites` – это *транспорт*, или, другими словами, *средство транспортировки* байтов. Транспорты герметизируют (инкапсулируют) процесс чтения данных из сокета, а также прием новых соединений. Транспорт не только иницирует действия в нашей программе, но и получает от программы результаты собственных действий;
- 2) *протоколы*: программа, которую мы привели в качестве примера, генерирует данные с помощью простого алгоритма и подсчитывает получаемые данные. Более сложные программы могут создавать веб-страницы или преобразовывать в текст голосовые телефонные сообщения. Пока эти программы могут принимать и отдавать байты, они могут согласованно работать с тем, что мы называли транспортом. Они также могут управлять поведением своего транспорта. Например, при получении недопустимых данных закрывать активное соединение. В области телекоммуникаций такие правила, определяющие порядок передачи дан-

¹ У. Р. Стивенс, Б. Феннер, Э. М. РудOFF. UNIX. Разработка сетевых приложений. Т. 1. 3-е изд. СПб.: Питер, 2007. ISBN: 5-318-00535-7. – Прим. перев.

ных, описываются в виде *протокола*. Протокол, таким образом, *определяет, как генерировать и обрабатывать входные и выходные данные*. То есть протокол инкапсулирует *эффект* нашей программы.

Реакторы: работа с транспортом

Мы изменим наш реактор `Reactor` и приспособим его для работы с транспортом:

```
import select
class Reactor(object):
    def __init__(self):
        self._readers = set()
        self._writers = set()
    def addReader(self, transport):
        self._readers.add(transport)
    def addWriter(self, transport):
        self._writers.add(transport)
    def removeReader(self, readable):
        self._readers.discard(readable)
    def removeWriter(self, writable):
        self._writers.discard(writable)
    def run(self):
        while self._readers or self._writers:
            r, w, _ = select.select(self._readers, self._writers, [])
            for readable in r:
                readable.doRead()
            for writable in w:
                if writable in self._writers:
                    writable.doWrite()
```

Если раньше наши обработчики событий готовности к чтению/записи были функциями, то теперь они являются методами объектов-транспортов: `doRead` и `doWrite`. Кроме того, теперь реактор мультиплексирует не сокеты, а транспорты. С точки зрения реактора интерфейс транспорта включает:

- 1) `doRead`;
- 2) `doWrite`;
- 3) что-то, что делает состояние транспорта видимым для `select`: метод `file-no()`, возвращающий число, распознаваемое `select` как ссылка на сокет.

ТРАНСПОРТЫ: РАБОТА С ПРОТОКОЛАМИ

Теперь вернемся к функциям `read` и `write` и рассмотрим реализацию протокола. На функцию `read` возлагается две обязанности:

- 1) подсчет количества байтов, полученных из сокета;
- 2) выполнение некоторых действий в ответ на закрытие соединения.

Функция `write` имеет единственную обязанность: помещать данные в очередь для записи.

Из вышеописанного мы можем сделать набросок первого проекта интерфейса протокола:

```
class Protocol(object):
    def makeConnection(self, transport):
        ...
    def dataReceived(self, data):
        ...
    def connectionLost(self, exceptionOrNull):
        ...
```

Обязанности `read` мы разделили на два метода: `dataReceived` и `connectionLost`. Сигнатура первого говорит сама за себя, а вот второй требует некоторых пояснений: он получает один аргумент – объект исключения, если соединение было закрыто из-за исключения (например, `ECONNRESET`), или `None`, если оно было закрыто не из-за ошибки (например, когда `read` не получила никаких данных). Обратите внимание, что в нашем интерфейсе протокола отсутствует метод `write`. Это связано с тем, что запись данных, которая предполагает транспортировку байтов, попадает в зону ответственности транспорта. То есть экземпляр `Protocol` должен иметь доступ к транспорту, представляющему сетевое подключение и имеющему метод `write`. Связь между этими двумя объектами создается с помощью метода `makeConnection`, принимающего транспорт в качестве аргумента.

Почему бы не передать аргумент с транспортом методу инициализации протокола? Использование отдельного метода может показаться излишеством, но такой подход дает больше гибкости; например, представьте, как этот метод позволит нам добавить кеширование в протокол. Более того, так как транспорт вызывает методы протокола `dataReceived` и `connectionLost`, он тоже должен иметь доступ к протоколу. Если бы обоим классам `Transport` и `Protocol` требовалось передать в метод инициализации друг друга, возникла бы циклическая связь, мешающая создать оба экземпляра. Мы решили разорвать эту циклическую связь и добавить в `Protocol` отдельный метод, принимающий транспорт.

Игра в пинг-понг с протоколами и транспортами

Этого достаточно, чтобы написать более сложный протокол, использующий новый интерфейс. В предыдущем примере клиент просто посылал серверу все увеличивающуюся последовательности байтов; мы можем сделать так, чтобы после достижения необязательного максимума получатель закрывал соединение.

```
class PingPongProtocol(object):
    def __init__(self, identity, maximum=None):
        self._identity = identity
        self._received = 0
        self._maximum = maximum
    def makeConnection(self, transport):
```

```

        self.transport = transport
        self.transport.write(b'')
    def dataReceived(self, data):
        self._received += len(data)
        if self._maximum is not None and self._received >= self._maximum:
            print(self._identity, "is closing the connection")
            self.transportloseConnection()
        else:
            self.transport.write(b'')
            print(self._identity, "wrote a byte")
    def connectionLost(self, exceptionOrNone):
        print(self._identity, "lost the connection:", exceptionOrNone)

```

Метод инициализации принимает строку `identity`, идентифицирующую экземпляр протокола, и необязательный максимальный объем данных, получив который, следует закрыть соединение. `makeConnection` связывает `PingPongProtocol` с его транспортом и начинает обмен, отправляя один байт. `dataReceived` записывает объем полученных данных. Если он превышает необязательный максимум, `dataReceived` сообщает транспорту о потере соединения, что эквивалентно отключению. Иначе `dataReceived` продолжает обмен, отправляя обратно один байт. Наконец, после закрытия соединения протоколом `connectionLost` печатает сообщение.

`PingPongProtocol` описывает набор моделей поведения, сложность которых значительно превышает наши предыдущие попытки создания неблокирующего приложения клиент-сервер. В то же время его реализация отражает предшествующее ему прозаическое описание, не увязая в подробностях неблокирующего ввода/вывода. Мы смогли увеличить сложность нашего приложения и одновременно уменьшить сложность уникального управления вводом/выводом. Позже мы вернемся к изучению этих последствий, но достаточно сказать, что сужение нашего внимания позволяет устранять сложности в конкретных областях нашей программы.

Мы не можем использовать `PingPongProtocol`, пока не напишем транспорт. Создадим первый проект интерфейса транспорта:

```

class Transport(object):
    def __init__(self, sock, protocol):
        ...
    def doRead(self):
        ...
    def doWrite(self):
        ...
    def fileno(self):
        ...
    def write(self):
        ...
    def loseConnection(self):
        ...

```

В первом аргументе методу инициализации передается сокет, который обертывает транспорт. Это обеспечивает инкапсуляцию сокетов в экземпляре `Transport`, с которыми в данное время работает реактор. Во втором аргументе передается протокол, чьи методы `dataReceived` и `connectionLost` будут вызываться при получении новых данных и закрытии соединения соответственно. Методы `doRead` и `doWrite` соответствуют описанному выше интерфейсу транспорта со стороны реактора. Частью этого интерфейса является новый метод `fileno`. Объект с правильно реализованным методом `fileno` можно передать в `select`. Наш метод `fileno` будет делегировать свои вызовы сокету, что сделает транспорты неотличимыми от сокетов с точки зрения `select`.

Метод `write` предоставляет интерфейс, который наш протокол использует для отправки исходящих данных. Мы на стороне протокола также добавили новый метод `loseConnection`, который инициирует закрытие сокета и представляет активную сторону закрытия соединения для пассивного метода `connectionLost`.

Мы можем реализовать этот интерфейс, встроив `BuffersWrites` и обработку сокетов в функцию `read`:

```
import errno

class Transport(object):
    def __init__(self, reactor, sock, protocol):
        self._reactor = reactor
        self._socket = sock
        self._protocol = protocol
        self._buffer = b""
        self._onCompletion = lambda:None
    def doWrite(self):
        if self._buffer:
            try:
                written = self._socket.send(self._buffer)
            except socket.error as e:
                if e.errno != errno.EAGAIN:
                    self._tearDown(e)
                return
            else:
                print("Wrote", written, "bytes")
                self._buffer = self._buffer[written:]
        if not self._buffer:
            self._reactor.removeWriter(self)
            self._onCompletion()
    def doRead(self):
        data=self._socket.recv(1024)
        if data:
            self._protocol.dataReceived(data)
        else:
            self._tearDown(None)
    def fileno(self):
        return self._socket.fileno()
```

```

def write(self, data):
    self._buffer += data
    self._reactor.addWriter(self)
    self.doWrite()
def loseConnection(self):
    if self._buffer:
        def complete():
            self.tearDown(None)
        self._onCompletion = complete
    else:
        self._tearDown(None)
def _tearDown(self, exceptionOrNone):
    self._reactor.removeWriter(self)
    self._reactor.removeReader(self)
    self._socket.close()
    self._protocol.connectionLost(exceptionOrNone)
def activate(self):
    self._socket.setblocking(False)
    self._protocol.makeConnection(self)
    self._reactor.addReader(self)
    self._reactor.addWriter(self)

```

`doRead` и `doWrite` выполняют те же манипуляции с сокетами, которые в предыдущих примерах производили функции `read` и `write` и объект `BuffersWrites`. `doRead` также передает все полученные данные в метод протокола `dataReceived` или, получив пустую последовательность байтов, вызывает метод `connectionLost`. Наконец, `fileno` завершает требуемый реактору интерфейс, делая `Transport` пригодным для передачи в вызов `select`.

Метод `write`, как и раньше, буферизует запись, но вместо делегирования процесса записи отдельному классу он для сброса в сокет данных вызывает соседний метод `doWrite`. Если буфер пуст, вызов `loseConnection` разрывает соединение:

- 1) удаляя транспорт из реактора;
- 2) закрывая базовый сокет, чтобы освободить ресурсы сокета и вернуть их обратно в операционную систему;
- 3) передавая `None` в вызов метода `connectionLost` протокола, чтобы указать, что соединение было разорвано методом пассивного закрытия.

Если буфер содержит данные для записи, `loseConnection` переопределяет обработчик `_onCompletion` и подставляет замыкание, которое разрывает соединение, действуя так же, как было описано выше. Как и в случае с `BuffersWrites`, `Transport._onCompletion` вызывается только тогда, когда все байты в нашем буфере записи будут сброшены в базовый сокет. `loseConnection` так использует `_onCompletion`, что базовое соединение гарантированно остается открытым до тех пор, пока все данные не будут записаны. Обработчик `_onCompletion` по умолчанию устанавливается методом инициализации класса `Transport` как лямбда без эффекта. Это гарантирует, что многократные вызовы `write` смогут повторно

использовать исходное соединение. Реализации `write` и `lateConnection` совместно осуществляют транспортный интерфейс, требуемый протоколом.

Наконец, `activate` активирует транспорт:

- 1) подготавливая обернутый сокет к неблокирующему вводу/выводу;
- 2) передавая экземпляр `Transport` в свой протокол вызовом `Protocol.makeConnection`;
- 3) и наконец, регистрируя транспорт в реакторе.

На этом `Transport` завершает инкапсуляцию начала жизненного цикла своего сокета, конец которого уже инкапсулирован в `loseConnection`.

Если класс `Protocol` помог расширить сферу нашего внимания и добавить в приложение новые особенности поведения за счет определения `PingPongProtocol`, то `Transport` сузил ее, сконцентрировав на жизненном цикле ввода/вывода сокетов. Реактор – наш цикл обработки событий – определяет и отправляет события, связанные с сокетами, а протокол содержит обработчики этих событий. `Transport` выполняет посреднические функции, преобразуя события из сокетов в вызовы методов протокола и обеспечивая *последовательный* вызов этих методов, например он гарантирует вызов метода `makeConnection` протокола в начале жизненного цикла и вызов `loseConnection` – в конце. Это еще одно улучшение по сравнению с нашим примером клиента и сервера; мы сконцентрировали поток управления сокетами внутри транспорта `Transport`, не распыляя его по независимым функциям и объектам.

Клиенты и серверы со своими реализациями протоколов и транспортов

Показать обобщенность `Transport` можно, определив подтип `Listener`, который принимает входящие соединения и связывает их с уникальным экземпляром `PingPongProtocol`:

```
class Listener(Transport):
    def activate(self):
        self._reactor.addReader(self)

    def doRead(self):
        server, _ = self._socket.accept()
        protocol = PingPongProtocol("Server")
        Transport(self._reactor, server, protocol).activate()
```

Сокет, принимающий запросы на соединение, не генерирует событий доступности для записи, поэтому мы переопределяем метод `activate` и регистрируем транспорт только для чтения. Наш обработчик событий чтения `doRead` должен получить новое соединение с клиентом и протокол, а затем связать их вместе в активированном экземпляре транспорта.

Вот этап настройки для примера клиента и сервера, управляемых протоколом и транспортом:

```
listenerSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listenerSock.bind(('127.0.0.1', 0))
```

```

listenerSock.listen(1)
clientSock = socket.create_connection(listenerSock.getsockname())

loop = Reactor()
Listener(loop, listenerSock, None).activate()
Transport(loop, clientSock, PingPongProtocol("Client", maximum=100)).
    activate()
loop.run()

```

Они будут обмениваться одиночными байтами, пока клиент не получит максимум 100 байт, после чего он закроет соединение:

```

Server wrote a byte
Client wrote a byte
Wrote 1 bytes
Server wrote a byte
Wrote 1 bytes
Client wrote a byte
Wrote 1 bytes
Server wrote a byte
Wrote 1 bytes
Client wrote a byte
Wrote 1 bytes
Server wrote a byte
Server wrote a byte
Client is closing the connection
Client lost the connection: None
Server lost the connection: None

```

Реакторы Twisted и протоколы с транспортом

Мы прошли долгий путь: начав с `select`, мы дошли до набора интерфейсов, инкапсулирующих цикл обработки событий, и обработчиков, которые четко разделяют обязанности. Нашей программой управляет экземпляр `Reactor`, а транспорты `Transport` отправляют события, порождаемые сокетами, в обработчики на уровне приложения, объявленные в протоколах `Protocol`.

Наши реакторы, транспорты и протоколы – это всего лишь пробные реализации. Например, вызов `socket.create_connection` блокирует приложение, и мы не исследовали никаких других неблокирующих альтернатив. Фактически базовое разрешение имен с помощью DNS, которое производится в `create_connection`, тоже может заблокировать приложение!

Однако концептуально они вполне готовы к серьезному использованию. Реакторы, транспорты и протоколы составляют основу событийно-ориентированной архитектуры Twisted. Как мы уже видели, их архитектура, в свою очередь, основана на мультиплексировании и неблокирующем выполнении ввода/вывода, что позволяет Twisted работать более эффективно.

Но прежде чем приступить к изучению самого фреймворка Twisted, рассмотрим наши примеры с высоты птичьего полета и оценим сильные и слабые стороны событийно-ориентированного программирования.

ПРЕИМУЩЕСТВА СОБЫТИЙНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Замечание Ричарда Стивенса (W. Richard Stevens) о сложности неблокирующего ввода/вывода считается важной критикой исследованной нами событийно-ориентированной парадигмы программирования. Однако это не единственный недостаток: наша событийно-ориентированная парадигма плохо работает при высоких нагрузках на процессор.

Пример реализации клиента и сервера, записывающих последовательности байтов, растущие в экспоненциальной прогрессии, естественно, потребляет намного больше памяти и вычислительных ресурсов центрального процессора. Причина заключается в упрощенном управлении буфером: сокет просто не может принимать фрагменты данных, размер которых превышает определенную величину. Поэтому каждый раз, когда методу `send` передается слишком большой объем данных, тот копирует их в область памяти, контролируемую ядром. Затем в сеть выводится некоторая часть данных, которую мы тут же удаляем из начала буфера; поскольку тип `bytes` в Python является неизменяемым, это подразумевает еще одну операцию копирования. При попытке отправить N байт буфер будет скопирован один раз, затем второй, а потом еще несколько раз, пока не будут отправлены все N байт. Поскольку каждое копирование предполагает обход содержимого буфера, этот процесс имеет временную сложность $O(n^2)$.

Собственные механизмы буферизации в Twisted работают лучше за счет более высокой сложности, чем было показано во введении в событийно-ориентированное программирование. Однако не все задачи с большим объемом вычислений легко поддаются такому усовершенствованию: имитация методом Монте-Карло должна многократно создавать случайные выборки и проводить статистический анализ; алгоритм сортировки должен сравнить каждую пару элементов в последовательности и т. д.

Все наши программы, управляемые событиями, выполняют множество логических действий. Наши клиент и сервер выполняются в одном процессе и взаимодействуют друг с другом *конкурентно*: клиент выполняет некоторую часть работы, затем приостанавливается и позволяет серверу сделать свой ход. Клиент и сервер никогда не работают *параллельно*, как в случае с программами, выполняющимися в разных процессах и, возможно, на разных компьютерах, связанных сетью. Когда наш упрощенный механизм управления буферами копирует длинный фрагмент данных, все остальное ждет завершения этой операции. Дело обстоит иначе, когда клиент и сервер выполняются на разных компьютерах, сервер может принимать новые соединения, а клиент – кропотливо перемешивать байты. Если бы мы запустили тяжелый, с вычислительной точки зрения, алгоритм в нашем процессе, тогда до завершения этого алгоритма реактор не смог бы вызвать `select`, чтобы обнаружить новые события, на которые нужно реагировать.

Из вышесказанного следует, что событийно-ориентированное программирование плохо подходит для вычислительных задач. К счастью, многие задачи предъявляют более высокие требования к вводу и выводу, чем к вычислениям. Классическим примером могут служить сетевые серверы; на сервере чата может быть зарегистрировано много тысяч пользователей, но только небольшая их часть активна в каждый конкретный момент времени (и, как назло, не тогда, когда вам нужна помощь!). Следовательно, событийно-ориентированное программирование остается мощной парадигмой для сетевых приложений.

Событийно-ориентированное программирование обладает одной важной чертой, с лихвой восполняющей этот недостаток: акцент на *причинах* и *эффектах*. Генерирование события – это причина, а обработка этого события – эффект.

Мы воплотили это разделение в виде транспортов и протоколов: транспорт представляет *причину* – некоторый ввод или вывод в сокете, а протокол инкапсулирует *эффект*. Наш `PingPongProtocol` взаимодействует со своим транспортом через четко определенный интерфейс, который предлагает обработчики для событий более высокого уровня – причин, таких как поступление входящих байтов или завершение соединения. Обработчики производят *эффекты*, реагируя на причины, которые, в свою очередь, могут привести к появлению новых *причин*, таких как запись данных в транспорт. Их разделение обеспечивается соответствующими интерфейсами.

Это означает, что мы можем заменить один транспорт другим и симитировать выполнение протокола, вызывая методы, производящие ожидаемые эффекты. Это превращает ядро наших клиента и сервера в единицу тестируемого кода.

Рассмотрим реализацию транспорта, основанную на `BytesIO`, реализующую только сторону протокола интерфейса транспорта:

```
import io

class BytesTransport(object):
    def __init__(self, protocol):
        self.protocol = protocol
        self.output = io.BytesIO()

    def write(self, data):
        self.output.write(data)

    def loseConnection(self):
        self.output.close()
        self.protocol.connectionLost(None)
```

Этот класс можно использовать для модульного тестирования нашего класса `PingPongProtocol`:

```
import unittest

class PingPongProtocolTests(unittest.TestCase):
```

```

def setUp(self):
    self.maximum = 100
    self.protocol = PingPongProtocol("client", maximum=self.maximum)
    self.transport = BytesTransport(self.protocol)
    self.protocol.makeConnection(self.transport)

def test_firstByteWritten(self):
    self.assertEqual(len(self.transport.output.getvalue()), 1)

def test_byteWrittenForByte(self):
    self.protocol.dataReceived(b"")
    self.assertEqual(len(self.transport.output.getvalue()), 2)

def test_receivingMaximumLosesConnection(self):
    self.protocol.dataReceived(b" " * self.maximum)
    self.assertTrue(self.transport.output.closed)

```

Этот тест проверяет требования, предъявляемые к протоколу `PingPongProtocol` без настройки сокетов и выполнения фактических операций ввода/вывода. С его помощью мы можем проверить *эффект* нашей программы, не порождая конкретных *причин*. Мы просто имитируем событие чтения, вызывая метод `dataReceived` нашего экземпляра протокола, и передаем ему байты, при этом протокол генерирует событие записи, вызывая метод `write` нашего транспорта байтов и закрывая соединение вызовом `loseConnection`.

Twisted делает все возможное, чтобы отделить причину от эффекта. Как было показано выше, наиболее очевидным преимуществом такого подхода является простота тестирования. Писать исчерпывающие тесты для событийно-ориентированных программ Twisted намного проще благодаря четкой границе между транспортом и протоколами. Такое разделение обязанностей в Twisted является следствием уроков, извлеченных на долгом и извилистом пути развития фреймворка, и повлекло к появлению обширного, а иногда и загадочного лексикона. Создание такого большого числа объектов, явно отделенных друг от друга, требует множества имен.

Теперь мы готовы написать событийно-ориентированную программу с использованием Twisted. В процессе этой работы мы столкнемся с теми же проблемами проектирования, что и в первых пробных примерах, и опыт их разработки поможет нам усвоить стратегии, которые Twisted предлагает для решения этих проблем.

TWISTED И РЕАЛЬНЫЙ МИР

Начнем наше исследование Twisted с реализации новой версии клиента и сервера `PingPongProtocol`:

```

from twisted.internet import protocol, reactor

class PingPongProtocol(protocol.Protocol):

    def __init__(self):
        self._received = 0

```

```

def connectionMade(self):
    self.transport.write(b'')

def dataReceived(self, data):
    self._received += len(data)
    if (self.factory._maximum is not None and
        self._received >= self.factory._maximum):
        print(self.factory._identity, "is closing the connection")
        self.transportloseConnection()
    else:
        self.transport.write(b'')
        print(self.factory._identity, "wrote a byte")

def connectionLost(self, exceptionOrNone):
    print(self.factory._identity, "lost the connection:",
          exceptionOrNone)

class PingPongServerFactory(protocol.Factory):
    protocol = PingPongProtocol
    _identity = "Server"

    def __init__(self, maximum=None):
        self._maximum = maximum

class PingPongClientFactory(protocol.ClientFactory):
    protocol = PingPongProtocol
    _identity = "Client"

    def __init__(self, maximum=None):
        self._maximum = maximum

listener=reactor.listenTCP(port=0,
                           factory=PingPongServerFactory(),
                           interface='127.0.0.1')
address = listener.getHost()
reactor.connectTCP(host=address.host,
                   port=address.port,
                   factory=PingPongClientFactory(maximum=100))

reactor.run()

```

Новая версия класса `PingPongProtocol` практически идентична предыдущей. В ней всего три изменения:

- 1) новая версия наследует `twisted.internet.protocol.Protocol`. Этот класс предлагает реализации важных функций по умолчанию. На момент разработки транспортов и протоколов в Twisted наследование как способ повторного использования кода пользовалось большой популярностью. Трудности, связанные с общедоступными и закрытыми API и разделением ответственности, справедливо привели к снижению этой популярности. Полное обсуждение недостатков наследования выходит за рамки этой главы, но мы не советуем писать новые API, опирающиеся на наследование!

- 2) мы заменили метод `makeConnection` методом `connectionMade`, который служит обработчиком событий, генерируемых фреймворком Twisted после подготовки базового соединения. Класс `Protocol` в Twisted реализует `makeConnection` и определяет метод `connectionMade` как простую заглушку, которую мы можем переопределить. На практике редко бывает необходимо менять способ взаимодействий транспорта с протоколом, но часто желательно выполнять некоторый код, как только соединение будет установлено. Этот обработчик дает такую возможность;
- 3) максимальное количество байтов и идентификатор протокола больше не являются переменными экземпляра, теперь это атрибуты новой переменной экземпляра `factory`.

Фабрики протоколов обобщают создание протоколов и их привязку к транспортам. Это наш первый пример локализации ответственности в классах, широко используемой в Twisted. Фабрики протоколов бывают двух основных типов: серверные и клиентские. Как следует из их имен, первые управляют созданием протоколов на стороне сервера, а вторые – на стороне клиента. Обе создают экземпляры протоколов, вызывая атрибут `protocol` без аргументов. Вот почему метод инициализации в `PingPongProtocol` не принимает аргументов.

`PingPongServerFactory` наследует `twisted.internet.protocol.Factory` и присваивает его атрибуту `_identity` значение "Server". Метод инициализации принимает аргумент с реактором и необязательный аргумент `maximum`. Затем он использует реализацию суперкласса для создания экземпляров протоколов – установленных на уровне класса `PingPongProtocol` – и связывает их с собой. Именно для этого экземпляры `PingPongProtocol` имеют атрибут `factory`: класс `Factory` создает его по умолчанию.

`PingPongClientFactory` наследует `twisted.internet.protocol.ClientFactory` и присваивает его атрибуту `_identity` значение "Client". В остальном он идентичен классу `PingPongServerFactory`.

Фабрики предоставляют удобное место для хранения состояния, используемого всеми экземплярами протокола. Поскольку экземпляры протокола являются уникальными для соединений, они прекращают свое существование, когда появляется соединение, и поэтому не могут хранить состояние. То есть перемещение настроек, таких как максимально допустимое значение и идентификационные строки протоколов клиента или сервера в соответствующие фабрики, следует общей схеме Twisted.

Реактор предоставляет методы `listenTCP` и `connectTCP`, которые связывают фабрики с соединениями на стороне сервера и клиента. `listenTCP` возвращает объект `Port`, метод `getHost` которого аналогичен функции `socket.getsockname`. Однако вместо кортежа он возвращает экземпляр `twisted.internet.address.IPv4Address`, который, в свою очередь, имеет удобные атрибуты `host` и `port`.

В заключение производится запуск реактора вызовом метода `run`, как мы уже делали это в предыдущей реализации. После этого на экране появляется вывод, напоминающий вывод предыдущей версии программы:

```

Client wrote a byte
Server wrote a byte
Client wrote a byte
Server wrote a byte
Client wrote a byte
Server wrote a byte
Client wrote a byte
Server wrote a byte
Client is closing the connection
Client lost the connection: [Failure instance: ...: Connection was closed
cleanly.
]
Server lost the connection: [Failure instance: ...: Connection was closed
cleanly.
]

```

Кроме объекта `Failure`, переданного в `connectionLost`, который мы рассмотрим, когда будем обсуждать асинхронное программирование в Twisted, этот вывод демонстрирует соответствие поведения новой и предыдущей реализаций.

Адаптировав наш тест протокола, мы можем убедиться в этом:

```

from twisted.trial import unittest

from twisted.test.proto_helpers import (
    StringTransportWithDisconnection,
    MemoryReactor
)

class PingPongProtocolTests(unittest.SynchronousTestCase):

    def setUp(self):
        self.maximum = 100
        self.reactor = MemoryReactor()
        self.factory = PingPongClientFactory(self.reactor, self.maximum)
        self.protocol = self.factory.buildProtocol(address.IPv4Address(
            "TCP", "localhost", 1234))
        self.transport = StringTransportWithDisconnection()
        self.protocol.makeConnection(self.transport)
        self.transport.protocol = self.protocol

    def test_firstByteWritten(self):
        self.assertEqual(len(self.transport.value()), 1)

    def test_byteWrittenForByte(self):
        self.protocol.dataReceived(b"")
        self.assertEqual(len(self.transport.value()), 2)

    def test_receivingMaximumLosesConnection(self):
        self.protocol.dataReceived(b" " * self.maximum)
        self.assertFalse(self.transport.connected)

```

Twisted имеет собственную инфраструктуру тестирования, которую мы рассмотрим, когда будем обсуждать асинхронное программирование; но

на данный момент `SynchronousTestCase` можно рассматривать как эквивалент `unittest.TestCase` из стандартной библиотеки. Метод `setUp` теперь создает фиктивный экземпляр `MemoryReactor`, используемый вместо реального реактора, и передает его в `PingPongClientFactory`, а затем создает клиентский протокол `PingPongProtocol`, вызывая метод `buildProtocol`, унаследованный от `ClientFactory`. Этот метод, в свою очередь, ожидает получить экземпляр адреса, вместо которого передается другой фиктивный объект. Затем мы используем встроенный в Twisted класс `StringTransportWithDisconnection`, поведение и интерфейс которого соответствуют нашей реализации `BytesTransport`. Свое имя `StringTransport` получил, потому что на момент написания тип `bytes` по умолчанию представлял строки. В мире Python 3 имя `StringTransport` стало неправильным, поскольку класс по-прежнему обрабатывает последовательности байтов.

Наши методы тестирования адаптированы к интерфейсу `StringTransportWithDisconnection`: `value` возвращает записанное содержимое, а `connected` получает значение `False`, когда протокол вызывает `loseConnection`.

Реализация клиента и сервера `PingPongProtocol` с использованием Twisted делает очевидным сходство между Twisted и нашим примером: реактор мультиплексирует события из сокетов и отправляет их через транспорты в протоколы, которые затем могут создавать новые события через свои транспорты.

Эта динамика лежит в основе событийно-ориентированной архитектуры Twisted и определяет ее проектные решения, но она является относительно низкоуровневой. Многие программы никогда не реализуют собственные подклассы `Protocol`.

Далее мы рассмотрим типы событий, лежащих в основе шаблонов и API, используемых во многих программах Twisted.

СОБЫТИЯ И ВРЕМЯ

Все события, которые мы видели до сих пор, были обусловлены вводом, например нажатием на клавишу или поступлением новых данных в сокет. Программы часто должны планировать выполнение некоторых действий в определенный момент в будущем, отдельно от ввода. Рассмотрим отправку контрольных сообщений, поддерживающих соединение в открытом состоянии: каждые 30 секунд или около того сетевое приложение будет записывать по одному байту в свои соединения, чтобы удаленная сторона не закрыла их из-за отсутствия активности.

Для планирования таких действий в будущем Twisted предлагает низкоуровневый интерфейс `reactor.callLater`. Мы обычно не вызываем его напрямую, но сделаем это сейчас, чтобы объяснить его работу.

```
from twisted.internet import reactor

reactor.callLater(1.5, print, "Hello from the past.")
reactor.run()
```

`reactor.callLater` принимает число, определяющее величину задержки в секундах, и функцию для вызова. Любые другие позиционные или именованные аргументы передаются вызываемой функции. После запуска эта программа выведет сообщение "Hello from the past." примерно через 1,5 секунды.

`reactor.callLater` возвращает экземпляр `DelayedCall`, с помощью которого можно отменить вызов:

```
from twisted.internet import reactor

call = reactor.callLater(1.5, print, "Hello from the past.")
call.cancel()
reactor.run()
```

Эта программа ничего не выведет, потому что она отменяет запланированный вызов функции с помощью `DelayedCall` до того, как реактор успеет выполнить его.

Очевидно, что `reactor.callLater` сгенерирует событие по истечении указанного времени и выполнит вызываемый объект, который он получил как обработчик этого события. Однако сама механика происходящего не так очевидна.

К счастью, реализация очень проста, и одного взгляда на нее достаточно, чтобы понять, почему задержка отмеряется лишь приблизительно. Как вы помните, `select` принимает необязательный аргумент времени задержки. Когда нужно, чтобы `select` немедленно сгенерировал какое-то событие, мы вызывали его со значением задержки 0. Теперь, в дополнение к событиям из сокетов, мы можем мультиплексировать события, основываясь на времени: чтобы убедиться, что наши отложенные вызовы `DelayedCall` выполняются, можно вызывать `select` с задержкой, равной задержке следующего `DelayedCall`, который должен быть запланирован, то есть ближайшего по времени.

Рассмотрим следующую программу:

```
reactor.callLater(2, functionB)
reactor.callLater(1, functionA)
reactor.callLater(3, functionC)
reactor.run()
```

Реактор сохраняет `DelayedCall` в своей куче в порядке запланированного времени запуска:

```
def callLater(self, delay, f, *args, **kwargs):
    self._pendingCalls.append((time.time()+delay, f, args, kwargs))
    heapq.heapify(self._pendingCalls)
```

Если предположить, что первый вызов `reactor.callLater` был сделан в момент времени t и затем сразу же было сделано еще два вызова, тогда после трех вызовов куча `pendingCalls` будет выглядеть так:

```
[
    (t+1, <DelayedCall: functionA>),
```

```

    (t+2, <DelayedCall: functionB>),
    (t+3, <DelayedCall: functionC>),
]

```

Добавление элемента в кучу имеет временную сложность $O(\log n)$, поэтому повторные вызовы `callLater` в худшем случае имеют общую сложность $O(n \log n)$. Если при каждом повторном вызове реактор будет выполнять пересортировку `_pendingCalls`, повторные вызовы `callLater` будут иметь сложность $O(n) * O(n \log n) = O(n^2)$.

Теперь, прежде чем реактор вызовет свой метод `select`, он проверит наличие ожидающих экземпляров `DelayedCall`; если таковые имеются, он извлечет верхний элемент из кучи и использует разность между целевым временем выполнения и текущим временем как тайм-аут вызова `select`. Затем, перед обработкой любых событий от сокета, он извлечет из кучи все элементы, для которых прошло время запуска, и вызовет их, пропуская отмененные вызовы. В отсутствие ожидающих вызовов `DelayedCall` реактор вызовет `select` с тайм-аутом, равным `None`, что означает отсутствие тайм-аута.

```

class Reactor(object):
    ...
    def run(self):
        while self.running:
            if self._pendingCalls:
                targetTime, _ = self._pendingCalls[0]
                delay=targetTime-time.time()
            else:
                targetTime = None
            r, w, _ = select.select(self.readers,self.writers, [], targetTime)
            now = time.time()
            while self._pendingCalls and (self._pendingCalls[0][0] <= now):
                targetTime, (f, args, kwargs) = heapq.heappop()
                if not call.cancelled:
                    f(*args,**kwargs)
            ...

```

Из наших трех вызовов `reactor.callLater` для `functionA` установлена самая короткая задержка, поэтому она находится на вершине кучи `pendingCalls`. Если цикл `run` реактора начнется сразу после этого вызова (т. е. вплоть до момента времени t), тогда переменная `delay` получит значение $(t + 1) - t = 1$, а вызов `select` будет повторен не позднее, чем через секунду. Теперь `time.time` вернет $t + 1$, и реактор выполнит `DelayedCall` для `functionA` и, следовательно, саму функцию `functionA`. Однако задержки в экземплярах `DelayedCall` для `functionB` и `functionC` все еще определяют время в будущем, поэтому внутренний цикл `while` завершится и процесс начнется сначала.

Реализация наглядно показывает, почему `DelayedCall` не вызываются сразу после истечения задержки: их вызов зависит от позиции в куче `pendingCalls` и от того, сколько времени потребуется для завершения предыдущих отло-

женных вызовов. Если `functionA` работала дольше секунды, `functionB` запустится позже своего срока. Это еще более вероятно для отложенных вызовов с той же задержкой.

Повторение событий с `LoopingCall`

Метода `reactor.callLater` вполне достаточно для реализации отправки контрольных сообщений. Мы можем определить функцию, которая вызывает `callLater` и передает ей саму себя, а затем запустить косвенную рекурсию, непосредственно вызвав функцию один раз:

```
def f(reactor, delay)
    reactor.callLater(delay, f, reactor, delay)

f(reactor, 1.0)
```

Это вполне работоспособное решение, но очень неуклюжее. Мы не имеем доступа к экземпляру `DelayedCall`, представляющему следующий вызов функции `f` после первого вызова `f`, поэтому не сможем отменить ее, если другая сторона разорвет соединение. Мы могли бы сохранять эти экземпляры вручную, но, к счастью, `Twisted` предлагает более удобное решение – обертку для `callLater`, которая делает все это автоматически: `twisted.internet.task.LoopingCall`. Вот протокол, использующий `LoopingCall` для реализации отправки контрольных сообщений:

```
from twisted.internet import protocol, task

class HeartbeatProtocol(protocol.Protocol):

    def connectionMade(self):
        self._heartbeater = task.LoopingCall(self.transport.write, b"*\n")
        self._heartbeater.clock = self.factory._reactor
        self._heartbeater.start(interval=30.0)

    def connectionLost(self):
        self._heartbeater.stop()

class HeartbeatProtocolFactory(protocol.Factory):
    protocol = HeartbeatProtocol

    def __init__(self, reactor):
        self._reactor = reactor
```

Протокол создает новый экземпляр `LoopingCall`, который будет выводить символ звездочки в транспорт протокола после открытия соединения. Затем записывает в атрибут `clock` экземпляра `LoopingCall` ссылку на фабричный реактор; как увидим ниже, такая косвенность упрощает тестирование. Наконец, протокол запускает `LoopingCall` с интервалом 30 секунд, поэтому примерно раз в 30 секунд он будет вызывать `transport.write` с одним символом звездочки. В какой момент `LoopingCall` начнет отсчет 30 секунд? Счет начнется с 0, и вызов функции произойдет сразу же, или с 1, и до первого вызова пройдут полные 30 секунд? Ответ на этот вопрос зависит от программиста. Второй обяза-

тельный аргумент `LoopingCall.start – now` – определяет, следует ли вызвать указанную функцию немедленно или после того, как истечет полный интервал. По умолчанию этот аргумент получает значение `True`, поэтому наш протокол передаст звездочку своему транспорту немедленно.

Получение реактора с помощью фабрики здорово упрощает тестирование `HeartbeatProtocol`:

```
from twisted.trial import unittest
from twisted.internet import main, task
from twisted.test.proto_helpers import StringTransportWithDisconnection

class HeartbeatProtocolTests(unittest.SynchronousTestCase):
    def setUp(self):
        self.clock = task.Clock()
        self.factory = HeartbeatProtocolFactory(self.clock)
        self.protocol = self.factory.buildProtocol(address.IPv4Address(
            "TCP", "localhost", 1234))
        self.transport = StringTransportWithDisconnection()
        self.protocol.makeConnection(self.transport)
        self.transport.protocol = self.protocol

    def test_heartbeatWritten(self):
        self.assertEqual(len(self.transport.value()), 1)
        self.clock.advance(60)
        self.assertEqual(len(self.transport.value()), 2)

    def test_lostConnectionStopsHeartbeater(self):
        self.assertTrue(self.protocol._heartbeater.running)
        self.protocol.connectionLost(main.CONNECTION_DONE)
        self.assertFalse(self.protocol._heartbeater.running)
```

Метод `HeartbeatProtocolTest.setUp` практически идентичен методу `PingPongProtocolTests.setUp`, за исключением того, что вместо `MemoryReactor.Clock` он использует `twisted.internet.task.Clock`, который, как следует из имени, предлагает реализацию методов, имеющих отношение ко времени. Но, самое главное, у него есть метод `callLater`:

```
>>> from twisted.internet.task import Clock
>>> clock = Clock()
>>> clock.callLater(1.0, print, "OK")
```

Поскольку экземпляры `Clock` предназначены для использования в модульных тестах, они не имеют своего цикла `select`. Смоделировать истечение тайм-аута `select` можно вызовом `advance`:

```
>>> clock.advance(2)
OK
```

`test_heartbeatWritten` вызывает `advance`, чтобы заставить `LoopingCall` протокола записать один байт. Это аналогично вызову метода `dataReceived` протокола в `PingPongProtocolTests.test_byteWrittenForByte`; оба моделируют возникновение событий, которые реактор мог бы сгенерировать в процессе работы.

Подход к событийно-ориентированному программированию, используемый в Twisted, зависит от четкого разграничения интерфейсов, таких как Protocol и Clock. Однако до сих пор мы воспринимали природу каждого интерфейса как нечто само собой разумеющееся. Но как узнать, что Clock или MemoryReactor могут заменить реальный реактор в тестах? Ответ на этот вопрос можно узнать, только исследуя инструменты, которые Twisted использует для управления своими интерфейсами.

ИНТЕРФЕЙСЫ СОБЫТИЙ В ZOPE.INTERFACE

Формальные объявления интерфейсов, включая те, что определяют событийно-ориентированную парадигму в Twisted, находятся в пакете zope.interface.

Zope – давний, но все еще активно развивающийся проект, выпустивший несколько фреймворков для веб-приложений, самый первый из которых увидел свет еще в 1998 году. В Zope были разработаны многие технологии для использования в других проектах. Twisted использует пакет с *интерфейсами* из проекта Zope для определения своих интерфейсов.

Полное описание zope.interface выходит за рамки этой книги. Однако интерфейсы играют настолько важную роль в тестировании и документировании, что надо хотя бы кратко познакомить вас с ними, исследуя интерфейсы классов Twisted, использованные в предыдущих примерах.

Сначала определим, какие интерфейсы реализует класс Clock:

```
>>> from twisted.internet.task import Clock
>>> clock = Clock()
>>> from zope.interface import providedBy
>>> list(providedBy(clock))
[<InterfaceClass twisted.internet.interfaces.IReactorTime>]
```

Для этого мы создали экземпляр Clock. Затем импортировали providedBy из пакета zope.interface; поскольку сам фреймворк Twisted зависит от модуля zope.interface, он доступен для импорта в интерактивном сеансе. Вызов providedBy с экземпляром Clock возвращает итерируемый объект с перечнем предоставляемых *интерфейсов*.

В отличие от интерфейсов в других языках программирования, интерфейсы из zope.interface можно *реализовать* или *предоставить*. Отдельные объекты, соответствующие интерфейсу, *предоставляют* его, тогда как компоненты, которые *создают* эти объекты, *предоставляющие* интерфейс, *реализуют* этот интерфейс. Это тонкое различие соответствует «утиной типизации» в Python. Определение интерфейса может описывать метод call и, следовательно, применяться к объекту функции, созданному с помощью def или lambda. Об этих синтаксических элементах нельзя сказать, что они *реализуют* наш интерфейс, но, безусловно, они его *предоставляют*.

Интерфейс – это подкласс, наследующий zope.interface.Interface, который использует специальный API для описания необходимых методов и их сигнала-

тип, а также атрибутов. Вот выдержка из интерфейса `twisted.internet.interfaces.IReactorTime`, предоставляемого классом `Clock`:

```
class IReactorTime(Interface):
    """
    Методы для работы со временем, которые должен реализовать Reactor.
    """

    def callLater(delay, callable,*args,**kw):
        """
        Выполняет отложенный вызов функции.

        @type delay: C{float}
        @param delay: величина задержки в секундах.

        @param callable: вызываемый объект для вызова после задержки.

        @param args: аргументы для вызываемого объекта.

        @param kw: именованные аргументы для вызываемого объекта.

        @return: Объект, предоставляющий L{IDelayedCall}, с помощью которого
            можно отменить запланированный вызов, вызовом метода
            C{cancel()}.
            С помощью его методов C{delay()} и C{reset()} можно также
            перепланировать отложенный вызов.
        """
```

Обратите внимание, что «метод» `callLater` не имеет аргумента `self`. Это является следствием того, что интерфейсы нельзя использовать для создания экземпляров. В нем также отсутствует тело, а чтобы удовлетворить требованиям синтаксиса определения функций в Python, была добавлена строка документирования. В отличие от абстрактных классов, например предоставляемых модулем `abc` из стандартной библиотеки, интерфейсы не могут включать код реализации – они только описывают подмножество функциональных возможностей объекта.

Zope предоставляет вспомогательную функцию `verifyObject`, которая вызывает исключение, если объект не предоставляет требуемый интерфейс:

```
>>> from zope.interface.verify import verifyObject
>>> from twisted.internet.interfaces import IReactorTime
>>> verifyObject(IReactorTime, clock)
True
>>> verifyObject(IReactorTime, object())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...
zope.interface.exceptions.DoesNotExist: An object does not implement
interface<Interface>
```

Мы можем использовать ее, чтобы убедиться, что реактор предоставляет тот же интерфейс `IReactorTime`, что и экземпляр `Clock`:

```
>>> from twisted.internet import reactor
>>> verifyObject(IReactorTime, reactor) True
```

Мы еще вернемся к `verifyObject`, когда начнем писать свои реализации интерфейсов. А пока просто имейте в виду, что реактор можно заменить экземпляром `Clock` везде, где требуется иметь доступ к `IReactorTime.callLater`. Вообще говоря, если известно, какой интерфейс предоставляет объект, включающий нужные нам методы и атрибуты, мы сможем заменить этот объект любым другим, который предоставляет тот же интерфейс. Конечно, мы можем вручную исследовать интерфейсы, предоставляемые объектом, используя `providedBy` в интерактивном сеансе, однако ту же самую информацию можно найти в электронной документации Twisted. На рис. 1.2 показана страница с описанием `Clock`.

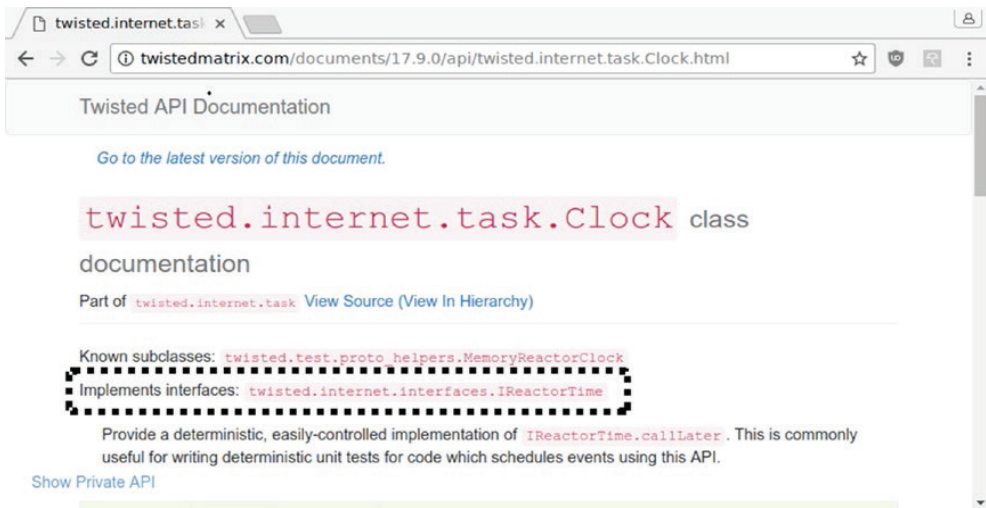


Рис. 1.2 ❖ Описание `twisted.internet.task.Clock`.
Пунктирной рамкой выделена ссылка на интерфейс `IReactorTime`

Интерфейсы, реализованные в классе `Clock`, выделены пунктирной рамкой. Щелчок на любом из них откроет страницу с его описанием, где также можно увидеть список всех известных разработчиков. Зная тип объекта, можно открыть страницу с его описанием и определить его интерфейсы.

Теперь перейдем к проблеме, для решения которой в Twisted используются определения реализаций интерфейсов.

УПРАВЛЕНИЕ ПОТОКОМ В СОБЫТИЙНО-ОРИЕНТИРОВАННЫХ ПРОГРАММАХ

`PingPongProtocol` отличается от протокола потоковой передачи, который мы реализовали в нашем последнем примере, не использующем события: каждая сторона в `PingPongProtocol` записывает байт в ответ на полученный байт, тогда

как в протоколе потоковой передачи клиент отправлял серверу последовательности байтов, приостанавливая запись, когда сервер не справлялся с нагрузкой. Регулировка скорости передачи в соответствии с возможностями получателя называется *управлением потоком*.

В сочетании с событийно-ориентированным программированием неблокирующий ввод/вывод позволяет писать программы, способные реагировать на множество различных событий, возникающих в любой момент времени. *Синхронный* ввод/вывод, как мы видели в клиентском протоколе потоковой передачи, реализованном в терминах `sendall`, приостанавливает, или *блокирует*, программу, не позволяя ей делать что-либо до завершения операции ввода/вывода. Это затрудняет реализацию параллельных вычислений, зато существенно упрощает управление потоками: отправитель, действующий быстрее получателя, просто приостанавливается операционной системой, пока получатель не примет ожидающих данных. В нашем потоковом клиенте это привело к тупиковой ситуации, потому что медленный получатель действовал в том же процессе, который был приостановлен из-за слишком высокой скорости отправки данных, и поэтому не мог наверстать упущенное. На практике отправители и получатели чаще выполняются в отдельных процессах, если не на отдельных машинах, и используемый в них синхронный и блокирующий ввод/вывод обеспечивает естественное управление потоком.

Однако простые блокирующие операции ввода/вывода редко используются в сетевых приложениях. Даже в самых простых случаях требуется одновременно управлять двумя аспектами каждого соединения: производить обмен данными и отслеживать тайм-аут для каждой операции ввода/вывода. Модуль `socket` в Python позволяет устанавливать тайм-ауты для операций `recv` и `sendall`, но внутренне они используют `select` с тайм-аутом!

У нас есть события, необходимые для реализации управления потоком. `select` сообщает нам о событиях записи, а `EAGAIN` указывает, что выходной буфер сокета заполнен и, следовательно, что получатель перегружен. Мы можем использовать их для приостановки и возобновления записи и таким способом управлять потоком почти так же, как это позволяет блокирующий ввод/вывод.

УПРАВЛЕНИЕ ПОТОКОМ В TWISTED С ПОМОЩЬЮ ПРОИЗВОДИТЕЛЕЙ И ПОТРЕБИТЕЛЕЙ

В системе управления потоками Twisted есть два компонента: *производители* и *потребители*. Производители посылают данные потребителям, вызывая метод `write` потребителя. Потребители *обертывают* производителей; каждый потребитель может быть связан с одним производителем. Эта связь гарантирует, что потребитель будет иметь доступ к своему производителю и возможность воздействовать на него, вызывая определенные методы производителя, что-бы регулировать интенсивность потока данных. Обычные транспорты, такие

как транспорт TCP, связанный с такими протоколами, как наш `PingPongProtocol`, могут быть как потребителями, так и производителями.

Исследуем порядок взаимодействий между производителями и потребителями, повторно реализовав пример клиента с потоковой передачей.

Активные производители

Начнем с производителя для клиента:

```
from twisted.internet.interfaces import IPushProducer
from twisted.internet.task import LoopingCall
from zope.interface import implementer

@implementer(IPushProducer)
class StreamingProducer(object):
    INTERVAL=0.001

    def __init__(self, reactor, consumer):
        self._data = [b"*, b*"]
        self._loop = LoopingCall(self._writeData, consumer.write)
        self._loop.clock = reactor

    def resumeProducing(self):
        print("Resuming client producer.")
        self._loop.start(self.INTERVAL)

    def pauseProducing(self):
        print("Pausing client producer.")
        self._loop.stop()

    def stopProducing(self):
        print("Stopping client producer.")
        if self._loop.running:
            self._loop.stop()

    def _writeData(self, write):
        print("Client producer writing", len(self._data), "bytes.")
        write(b"".join(self._data))
        self._data.extend(self._data)
```

Наш производитель, `StreamingProducer`, реализует `twisted.internet.interfaces.IPushProducer`. Этот интерфейс описывает производителей, которые непрерывно посылают данные своим потребителям, пока те не остановят их. Класс `StreamingProducer`, реализующий интерфейс `IPushProducer`, должен включать следующие методы:

- `resumeProducing`: возобновляет или иницирует процесс отправки данных потребителю. Поскольку наша реализация генерирует свои данные, удваивая последовательность байтов после каждой операции вывода, ей нужен какой-то цикл для отправки непрерывного потока своему потребителю. Простой цикл `while` не годится: если программа не будет возвращать управление реактору, она не сможет обрабатывать новые события до завершения цикла. Программа, управляемая событиями, такая как веб-браузер, в таком случае будет эффективно приостанавливать свою

работу на время выгрузки большого файла. `StreamingProducer` решает эту проблему, делегируя выполнение цикла записи реактору через экземпляр `LoopingCall`. Именно поэтому его метод `resumeProduction` запускает `LoopingCall`. Интервал в одну миллисекунду выбран произвольно. Наш производитель не сможет посылать данные чаще, поэтому интервал играет роль источника задержки, и величины в одну миллисекунду вполне достаточно;

- `pauseProducing`: приостанавливает процесс отправки данных потребителю. Потребитель вызывает этот метод, чтобы сообщить, что он перегружен и не может принять больше данных. В нашей реализации достаточно остановить базовый `LoopingCall`. Позднее потребитель может вызвать `resumeProducing`, когда будет готов принять больше данных. Цикл вызовов `resumeProducing` и `pauseProducing` составляет управление потоком;
- `stopProducing`: прекращает отправку данных. От `pauseProducing` отличается тем, что вызовом `stopProducing` потребитель сообщает о полном прекращении приема данных и что он больше никогда не вызовет `resumeProducing`. Наиболее очевидное применение этого метода – после закрытия соединения сокетом. Реализация `stopProducing` от `pauseProducing` в `StreamingProducer` отличается только тем, что он должен сначала проверить, запущен ли циклический вызов `LoopingCall`. Это связано с тем, что потребитель может запросить дальнейшую передачу данных, когда производитель приостановлен. Более сложные активные производители могут выполнять дополнительную очистку; например, производитель, передающий данные из файла, должен закрыть этот файл здесь, чтобы вернуть ресурсы операционной системе.

Обратите внимание, что `IPushProducer` не определяет, как его реализация должна передавать данные потребителю или получать доступ к ним. Это делает интерфейс более гибким, но также более сложным в реализации. `StreamingProducer` следует типичному шаблону, принимая потребителя в своем методе инициализации. Полный интерфейс потребителя мы рассмотрим чуть ниже, а пока просто запомните, что потребители должны определять метод `write`.

Теперь протестируем правильность реализации интерфейса `IPushProducer` в `StreamingProducer`:

```
from twisted.internet.interfaces import IPushProducer
from twisted.internet.task import Clock
from twisted.trial import unittest
from zope.interface.verify import verifyObject
```

```
class FakeConsumer(object):
```

```
    def __init__(self, written):
        self._written = written
```

```
    def write(self, data):
        self._written.append(data)
```

```
class StreamingProducerTests(unittest.TestCase):
```

```

def setUp(self):
    self.clock = Clock()
    self.written = []
    self.consumer = FakeConsumer(self.written)
    self.producer = StreamingProducer(self.clock, self.consumer)

def test_providesIPushProducer(self):
    verifyObject(IPushProducer, self.producer)

def test_resumeProducingSchedulesWrites(self):
    self.assertFalse(self.written)
    self.producer.resumeProducing()
    writeCalls = len(self.written)
    self.assertEqual(writeCalls, 1)
    self.clock.advance(self.producer.INTERVAL)
    newWriteCalls = len(self.written)
    self.assertGreater(newWriteCalls, writeCalls)

def test_pauseProducingStopsWrites(self):
    self.producer.resumeProducing()
    writeCalls = len(self.written)
    self.producer.pauseProducing()
    self.clock.advance(self.producer.INTERVAL)
    self.assertEqual(len(self.written), writeCalls)

def test_stopProducingStopsWrites(self):
    self.producer.resumeProducing()
    writeCalls = len(self.written)
    self.producer.stopProducing()
    self.clock.advance(self.producer.INTERVAL)
    self.assertEqual(len(self.written), writeCalls)

```

`FakeConsumer` принимает список, в конец которого каждый вызов `write` добавляет полученные данные. Это позволяет тесту убедиться, что `StreamingProducer` вызвал метод `write` своего потребителя, как и ожидалось.

`test_providesIPushProducer` проверяет наличие в `StreamingProducer` методов, которые определены в `IPushProducer`. Если какой-то метод отсутствует, этот тест завершится с исключением `zope.interface.exceptions.DoesNotImplement`. Подобные тесты, проверяющие соответствие реализации своим интерфейсам, являются полезными фильтрами при разработке и рефакторинге.

`test_resumeProducingSchedulesWrites` проверяет, что при каждом вызове `resumeProducing` выполняется передача данных потребителю и по прошествии заданного интервала происходит передача дополнительных данных. `test_pauseProducingStopsWrites` и `test_stopProducingStopsWrites` проверяют обратное условие: вызовы `pauseProducing` и `stopProducing` прекращают отправку новых данных по истечении каждого интервала.

Потребители

`StreamingProducer` генерирует данные, но ему пока некуда их поместить. Чтобы завершить реализацию потокового клиента, нам нужен *потребитель*. Ме-

тод инициализации в `StreamingProducer` наглядно показывает, что потребитель должен иметь метод `write`, а в описании выше мы говорили, что потребитель должен иметь дополнительные методы для управления взаимодействиями с производителями. Интерфейс `twisted.internet.interfaces.IConsumer` требует реализации трех методов:

- `write`: принимает данные от производителя. Это единственный метод, реализованный в `FakeConsumer` в тестах выше, потому что это единственный метод интерфейса `IConsumer`, который вызывает `IPushProducer`;
- `registerProducer`: связывает производителя с потребителем, гарантирует потребителю, что тот сможет вызывать методы `resumeProducing` и `pauseProducing` производителя для управления потоком данных, а также `stopProducing` для остановки производителя. Принимает два аргумента: ссылку на производителя и признак потоковой передачи. Назначение этого второго аргумента мы объясним позже, а пока достаточно знать, что наш потоковый клиент передает в этом аргументе значение `True`;
- `unregisterProducer`: разрывает связь между потребителем и производителем. На протяжении своего существования потребитель может принимать данные от нескольких производителей; снова представьте веб-браузер, который может загружать несколько файлов через одно соединение с сервером.

Не случайно, что реализации `IConsumer` и транспорты предоставляют метод `write`; как упоминалось выше, транспорт TCP, связанный с подключенными протоколами, является потребителем, с помощью которого можно зарегистрировать экземпляр `StreamingProducer`. Мы можем изменить наш пример `PingPongProtocol` и добавить в него регистрацию `StreamingProducer` с его базовым транспортом после успешной установки соединения:

```
from twisted.internet import protocol, reactor
from twisted.internet.interfaces import IPushProducer
from twisted.internet.task import LoopingCall
from zope.interface import implementer

@implementer(IPushProducer)
class StreamingProducer(object):
    INTERVAL=0.001

    def __init__(self, reactor, consumer):
        self._data = [b"*, b*"]
        self._loop = LoopingCall(self._writeData, consumer.write)
        self._loop.clock = reactor

    def resumeProducing(self):
        print("Resuming client producer.")
        self._loop.start(self.INTERVAL)

    def pauseProducing(self):
        print("Pausing client producer.")
        self._loop.stop()
```

```

def stopProducing(self):
    print("Stopping client producer.")
    if self._loop.running:
        self._loop.stop()

def _writeData(self, write):
    print("Client producer writing", len(self._data),"bytes.")
    write(b"".join(self._data))
    self._data.extend(self._data)

class StreamingClient(protocol.Protocol):

    def connectionMade(self):
        streamingProducer = StreamingProducer(
            self.factory._reactor,self.transport)
        self.transport.registerProducer(streamingProducer,True)
        streamingProducer.resumeProducing()

class ReceivingServer(protocol.Protocol):

    def dataReceived(self, data):
        print("Server received", len(data),"bytes.")

class StreamingClientFactory(protocol.ClientFactory):
    protocol = StreamingClient

    def __init__(self, reactor):
        self._reactor = reactor

class ReceivingServerFactory(protocol.Factory):
    protocol = ReceivingServer

listener = reactor.listenTCP(port=0,
                             factory=ReceivingServerFactory(),
                             interface='127.0.0.1')

address = listener.getHost()
reactor.connectTCP(host=address.host,
                  port=address.port,
                  factory=StreamingClientFactory(reactor))

reactor.run()

```

Протокол `StreamingClient` создает `StreamingProducer`, который затем регистрируется в его транспорте. Как и было обещано, во втором аргументе в вызов `registerProducer` передается `True`. Однако регистрация производителя не запускает его автоматически, поэтому нужно запустить цикл записи в `StreamingProducer`, вызвав `resumeProducing`. Обратите внимание, что `StreamingClient` никогда не вызывает метод `stopProducing` своего производителя: это делает транспорт, когда реактор сигнализирует о разрыве соединения.

Если запустить эту программу, она выведет следующее:

```

Resuming client producer.
Client producer writing 2 bytes.
Server received 2 bytes.
Client producer writing 4 bytes.
Server received 4 bytes.

```

```
Client producer writing 8 bytes.  
Server received 8 bytes.  
...  
Client producer writing 524288 bytes.  
Pausing client producer.  
Server received 65536 bytes.  
Server received 65536 bytes.  
Server received 65536 bytes.  
Server received 65536 bytes.  
Resuming client producer.  
Client producer writing 1048576 bytes.  
Pausing client producer.  
...
```

В какой-то момент программа начнет пересылать объемы данных, превышающие объем доступной памяти, что будет ярко свидетельствовать об успешном управлении потоком.

Пассивные производители

Существует еще один интерфейс производителя: `twisted.internet.interfaces.IPullProducer`. В отличие от `IPushProducer`, он посылает данные потребителю, только когда тот вызовет его метод `resumeProduction`. Такое поведение обеспечивается вторым аргументом метода `IConsumer.registerProducer`: `IPullProducer` требует, чтобы параметр `streaming` имел значение `False`. Старайтесь не использовать `IPullProducer`! Большинство транспортов действуют подобно сокетам и генерируют события доступности для записи, что устраняет необходимость в цикле записи, как это сделано в `StreamingProducer`. Когда данные необходимо извлекать из источника вручную, лучше и проще написать и протестировать `LoopingCall`.

Итоги

В этой главе вы узнали, как событийно-ориентированный подход позволяет разделить программы на *события* и их *обработчики*. Все, что происходит в программе, можно смоделировать в виде событий: ввод пользователя, получение данных через сокет и даже время. *Цикл обработки событий* использует прием *мультиплексирования* для ожидания любых возможных событий, вызывая соответствующие обработчики для происшедших. Операционные системы предоставляют низкоуровневые интерфейсы, такие как `select`, для мультиплексирования событий ввода/вывода. Событийно-ориентированное программирование сетевого ввода/вывода с использованием `select` обеспечивает неблокирующий способ работы, когда для таких операций, как `send` и `recv`, генерируются события, указывающие, что программа должна прекратить запуск обработчика событий.

Использование события остановки – `EAGAIN`, – генерируемого неблокирующими сокетами, усложняет код, если не использовать правильных абстракций. *Протоколы и транспорты* помогают разделить программный код на *причи-*

ны и эффекты: транспорты преобразуют события чтения, записи и остановки в высокоуровневые события-причины, на которые протоколы могут реагировать и, в свою очередь, генерировать новые события. Такое разделение ответственности между протоколами и транспортами позволяет реализовать обработчики событий, которые легко тестируются путем замены транспортов фиктивными объектами. Позже мы увидим другие практические преимущества разделения на протоколы и транспорты.

Протоколы, транспорты и реакторы – это составные части циклов обработки событий. Они имеют решающее значение для Twisted и определяют его общую архитектуру. Реактор в Twisted может реагировать на события, не связанные с вводом/выводом, например определяемые течением времени. Тестировать их не сложнее, чем протоколы, потому что для реакторов, как и для транспортов, имеются свои фиктивные объекты-имитации. Twisted формализует интерфейсы, которые должны реализовывать реакторы и другие объекты, с помощью `zope.interface`. Определяя, какие интерфейсы предоставляет объект, можно выбрать подходящую замену для тестирования, гарантированно эквивалентную, благодаря поддержке тех же интерфейсов. Онлайн-документация Twisted упрощает выявление поддерживаемых интерфейсов, по сравнению с исследованием фактических объектов в интерактивном сеансе Python.

Практическое использование интерфейсов в Twisted помогает успешно преодолевать многие сложности событийно-ориентированного программирования сетевых приложений, в частности сложности управления потоком данных. Интерфейсы `IPushProducer` и `IConsumer` определяют набор методов, которые позволяют получателю потоковых данных приостанавливать производителя, когда он не успевает обрабатывать получаемые данные.

Этого краткого введения достаточно, чтобы объяснить основные принципы событийно-ориентированного программирования в Twisted. Однако мы не заканчиваем обсуждение данной темы: в следующей главе вы узнаете, как Twisted упрощает событийно-ориентированное программирование, позволяя работать со значениями, которые еще предстоит вычислить.

Глава 2

Введение в асинхронное программирование с Twisted

В предыдущей главе были представлены основные принципы, лежащие в основе событийно-ориентированной архитектуры Twisted. Программы, использующие фреймворк Twisted, как и все событийно-ориентированные программы, упрощают параллельную обработку за счет усложнения управления потоками данных. Событийно-ориентированная программа не приостанавливается при выполнении блокирующих операций ввода/вывода, когда отправляет больше данных, чем может обработать принимающая сторона, и несет всю полноту ответственности за определение подобных ситуаций и их предотвращение.

Порядок передачи данных между взаимодействующими сторонами также влияет на порядок передачи в рамках одной программы. Как результат стратегии *композиции* различных компонентов в событийно-ориентированных приложениях отличаются от тех, что используются в блокирующихся программах.

ОБРАБОТЧИКИ СОБЫТИЙ И ИХ КОМПОЗИЦИЯ

Рассмотрим программу, которая не является событийно-ориентированной и использует блокирующие операции сетевого ввода/вывода:

```
def requestField(url, field):
    results = requests.get(url).json()
    return results[field]
```

Функция `requestField` извлекает содержимое по заданному адресу URL, используя библиотеку `requests`, декодирует тело ответа в формате JSON и возвращает значение запрошенного свойства `field` из полученного словаря. Библиотека `requests` использует блокирующий ввод/вывод, поэтому вызов `requestField` приостановит выполнение программы, пока не завершатся сетевые операции, инициированные HTTP-запросом. По этой причине функция может предполагать, что результаты будут доступны перед возвратом из нее. Код, вызывающий эту функцию, может положиться на то же самое предположение, потому что `requestField` будет блокировать его, пока не вычислит свой результат:

```
def someOtherFunction(...):
    ...
    url = calculateURL(...)
    value = requestField(url, 'someInteger')
    return value + 1

x = someOtherFunction(...)
```

Ни `someOtherFunction`, ни операция присваивания переменной `x` верхнего уровня не завершатся, пока `requestField` не извлечет значение свойства `someInteger` из ответа в формате JSON. Это пример *композиции*: `someOtherFunction` вызывает `requestField`, чтобы решить свою задачу. То же самое можно сделать, явно используя прием композиции функций:

```
def someOtherFunction(value):
    return value + 1

x = someOtherFunction(requestField(calculateURL(...), 'someInteger'))
```

Этот код замещает локальные переменные в `someOtherFunction` вложенными вызовами функций, но в остальном полностью эквивалентен предыдущему решению.

Композиция функций – фундаментальный инструмент организации программ. Этот прием позволяет *разложить*, или разбить, программу на отдельные модули, образующие единое целое с поведением, точно соответствующим первоначальной версии. Одновременно улучшаются удобочитаемость, возможность повторного использования и простота тестирования.

К сожалению, обработчики событий не получится скомпоновать подобно `someOtherFunction`, `requestField` и `calcURL`. Рассмотрим гипотетическую неблокирующую версию `requestField`:

```
def requestField(url, field):
    ??? = nonblockingGet(url)
```

Какой код мог бы заменить `???` в неблокирующей версии `requestField`? Это сложный вопрос, потому что `nonblockingGet` не приостанавливает выполнение программы для завершения сетевых операций, инициированных HTTP-запросом к `url`. Вместо этого внешний цикл событий, действующий *за пределами* `requestField`, мультиплексирует события чтения и записи и вызывает соответствующие обработчики для получения и отправки данных, когда это становится возможно. Не существует очевидного способа вернуть значение обработчика событий из нашей гипотетической функции `nonblockingGet`.

К счастью, представляя обработчики событий как функции, можно использовать универсальный прием композиции функций для разделения событийно-ориентированной программы на отдельные компоненты. Предположим, что гипотетическая функция `nonblockingGet` сама принимает функцию-обработчик события в аргументе и вызывает ее, когда поступает событие завершения запроса. Это высокоуровневое событие будет синтезировано из событий более низкого уровня, подобно тому, как в главе 1 транспорты генерировали

события `connectionLost` для своих протоколов. В этом случае мы можем переписать `requestField` и воспользоваться этим новым аргументом:

```
def requestField(url, field):
    def onCompletion(response):
        document = json.loads(response)
        value = document[field]

    nonblockingGet(url, onCompletion=onCompletion)
```

`onCompletion` – это *функция обратного вызова (обработчик)*, или вызываемый объект, передаваемый в качестве аргумента некоторому другому вызываемому объекту, выполняющему некоторую операцию. Когда операция завершится, будет выполнен вызов этой функции с соответствующим аргументом или аргументами. В данном случае, когда HTTP-запрос будет преобразован в объект `document`, функция `nonblockingGet` вызовет `onCompletion`. Эквивалентный обратный вызов `onCompletion` мы видели в предыдущей главе, в реализации `BufferedWrites`; там этот вызов производился после записи всех буферизованных данных в сокет.

Композиция обратных вызовов производится *внутри*, тогда как композиция обычных функций, как в примере с `someOtherFunction` выше, производится *снаружи*; значения передаются обратным вызовам в процессе выполнения вызываемого объекта, после получения желаемого результата, а не возвращаются из него.

Подобно тому, как `nonblockingGet` выделяет управляемый событиями код, выполняющий HTTP-запрос, `requestField` тоже может выделить обработку извлеченного поля, принимая свой обратный вызов. В результате получится функция `requestField`, принимающая обратный вызов `useField`, который будет вызываться обратным вызовом `onCompletion`:

```
def requestField(url, field, useField):
    def onCompletion(response):
        document = json.loads(response)
        value = document[field]
        useField(value)

    nonblockingGet(url, onCompletion=onCompletion)
```

То есть, чтобы получить событийно-ориентированную программу, эквивалентную версии, использующей блокирующий ввод/вывод, мы можем переписать `someOtherFunction` и передать ей обратный вызов `useField`:

```
def someOtherFunction(useValue):
    url = calculateURL(...)
    def addValue(value):
        useValue(value + 1)
    requestField(url, "someInteger", useField=addValue)
```

`someOtherFunction`, в свою очередь, тоже должна использовать прием внутренней композиции, принимая свой обратный вызов вместо функции `calculateURL`,

как в примере внешней композиции. Такой подход, основанный на обратных вызовах, можно использовать в любых программах; но вообще в информатике обратные вызовы принято преобразовывать в примитивы управления потоком, которые называются *продолжениями* и используются в реализации приема, называемого *передачей продолжения*, когда функции завершаются вызовом их продолжений с результатом. Прием передачи продолжения широко используется в разных языках для анализа и оптимизации программ.

Несмотря на теоретические преимущества, код, использующий прием передачи продолжений, трудно писать и читать. Кроме того, внешняя композиция – как в примере с `requestField` и `calculateURL` – и внутренняя композиция – как в примере с `requestField` и `useField` – явно не сочетаются друг с другом. Сложно, например, представить передачу `calculateURL` в качестве обратного вызова. Наконец, не стоит забывать об обработке ошибок; представьте, во что превратится обработка исключений в приеме передачи продолжений! Мы намеренно опустили обработку ошибок в этом примере, чтобы код получился достаточно коротким и простым для чтения.

К счастью, *асинхронное программирование* предлагает мощную абстракцию, упрощающую композицию обработчиков событий, и решает эти проблемы.

ЧТО ТАКОЕ АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ?

Наша первоначальная версия `requestField` выполняется *синхронно*, потому что действует линейно, шаг за шагом. Например, в двух вызовах `request.get` второй вызов выполнится, только когда завершится первый. Парадигма синхронного программирования широко распространена и соответствует природе блокирующих операций ввода/вывода. Большинство языков программирования, включая Python, по умолчанию выполняют операции синхронно, что обеспечивается блокирующими операциями ввода/вывода.

Метод передачи продолжения в событийно-ориентированной версии `requestField` является разновидностью *асинхронного программирования*: логический поток выполнения обратных вызовов в `nonblockingGet` приостанавливается до получения необходимых данных, но остальная программа продолжает выполняться. Выполнение двух отдельных вызовов `nonblockingGet` будет чередоваться без каких-либо гарантий относительно порядка их завершения; запуск одного запроса раньше другого не гарантирует, что второй завершится после первого. Это определение параллелизма, или конкуренции.

Событийно-ориентированная программа, использующая неблокирующий ввод/вывод, всегда действует асинхронно, потому что все операции ввода/вывода выполняются по событиям, которые могут поступить в любой момент и в любом порядке. Важно отметить, что асинхронной может быть не только программа, обрабатывающая события ввода/вывода; разные платформы предоставляют шаблоны ввода/вывода и планирования выполнения на основе принципиально разных примитивов. Например, Windows предлагает порты

завершения ввода/вывода (I/O Completion Ports, IOCP), информирующие программы о *завершении* запрошенной операции, а не о возможности ее выполнить. Например, программа, использующая инфраструктуру IOCP для чтения из сокета, будет уведомлена, когда завершится чтение, и сможет получить прочитанные данные. Фреймворк Twisted предлагает некоторую поддержку этого механизма в виде реактора IOCP, но в нашем случае асинхронное программирование можно понимать как следствие раздельного и частичного выполнения в событийно-ориентированной парадигме, так же как синхронное программирование является следствием блокирующего ввода/вывода.

Заполнители для будущих значений

Обратные вызовы в событийно-ориентированных программах скрывают поток управления, используя *внутреннюю* композицию, – они не возвращают значение вызывающей стороне, а передают результаты обратным вызовам, указанным в аргументах. Это нарушает линейность логики приложения и потока управления, затрудняет рефакторинг и разрывает связь между точкой появления ошибки и кодом, который должен ее обработать.

Введение объекта для значения, которое еще предстоит вычислить, позволяет использовать *внешнюю* композицию для обратных вызовов. Взгляните, как меняется неблокирующий пример `requestField`, если разрешить возвращать такой объект-заполнитель:

```
def requestField(url, field):
    def onCompletion(response):
        document = json.load(response)
        return document[field]

    placeholder = nonblockingGet(url)
    return placeholder.addCallback(onCompletion)
```

Функция `nonblockingGet` теперь возвращает *объект-заполнитель* (`placeholder`), который является не ответом, а контейнером, куда будет помещен ответ после его получения. Контейнер без операций не дает никакой выгоды, поэтому он позволяет добавлять *обработчики*, которые будут вызваны, когда результат будет готов. Вместо передачи обработчика `onCompletion` непосредственно в `nonblockingGet` мы добавляем его в объект-заполнитель, возвращаемый функцией `nonblockingGet`. Реализация внутреннего обратного вызова `onCompletion` теперь может вернуть значение – поле из документа JSON, – которое станет доступно в будущем, в момент обратного вызова.

Аналогично в функции `requestField` тоже можно исключить аргумент с обработчиком и вернуть заполнитель функции `someOtherFunction`, которая сможет добавить в него свой обработчик:

```
def someOtherFunction(...):
    url = calculateURL(...)
    def addValue(value)
```

```

    return value + 1
placeholder = requestField(url, "someInteger")
return placeholder.addCallback(addValue)

```

Использование объекта-заполнителя не избавляет от обратных вызовов, но дает абстракцию потока управления, позволяющую разместить обратные вызовы (обработчики) в одной области видимости и использовать прием *внешней композиции*. Преимущества такого подхода становятся очевиднее, когда возникает необходимость использовать несколько обратных вызовов для обработки асинхронного результата. Взгляните на следующий пример внутренней композиции обратных вызовов:

```

def manyCallbacks(url, useValue, ...):
    def addValue(result):
        return divideValue(result + 2)

    def divideValue(result):
        return multiplyValue(result // 3)

    def multiplyValue(result):
        return useValue(result * 4)

    requestField(url, "someInteger", onCompletion=addValue)

```

Управление передается из `addValue` в `divideValue`, затем в `multiplyValue` и, наконец, в обработчик `useValue`, переданный в вызов `manyCallbacks`. Чтобы изменить порядок этих трех обратных вызовов, придется переписать каждый из них. Объект-заполнитель устраняет этот недостаток, позволяя определять порядок извне:

```

def manyCallbacks(url, ...):
    def addValue(result):
        return result + 2

    def divideValue(result):
        return result // 3

    def multiplyValue(result):
        return result * 4

    placeholder = requestField(url, "someInteger")
    placeholder.addCallback(addValue)
    placeholder.addCallback(divideValue)
    placeholder.addCallback(multiplyValue)
    return placeholder

```

`divideValue` больше не зависит напрямую от `multiplyValue`, поэтому его можно поместить перед `multiplyValue` или вообще убрать из цепочки операций, не изменяя его или `multiplyValue`.

Фактическая композиция обратных вызовов выполняется внутри объекта-заполнителя, имеющего очень простую реализацию. Дадим нашему классу заполнителя имя `Deferred`, потому что он представляет отложенные вычисления, результат которых пока не известен:

```
class Deferred(object):
    def __init__(self):
        self._callbacks = []

    def addCallback(self, callback):
        self._callbacks.append(callback)

    def callback(self, result):
        for callback in self._callbacks:
            result = callback(result)
```

Созданный экземпляр `Deferred` вызовет метод `callback` с результатом, когда он станет доступным. Этот метод, в свою очередь, вызовет каждый назначенный обработчик с текущим результатом и сохранит его возвращаемое значение в текущем результате для передачи следующему обработчику. С его помощью приведенная выше функция `onCompletion` сможет преобразовать HTTP-ответ в единственное поле, извлеченное из документа в формате JSON.

Поток управления, реализованный циклом `for` в `Deferred`, последовательно вызовет все обработчики, но он не способен обрабатывать исключения лучше, чем обратные вызовы при использовании приема внутренней композиции. Чтобы решить эту проблему, нужно добавить некоторую условную логику, обнаруживающую и пересылающую исключения в нужное место.

АСИНХРОННАЯ ОБРАБОТКА ИСКЛЮЧЕНИЙ

Вот как выглядит синхронная обработка исключений в Python с использованием `try` и `except`:

```
def requestField(url):
    response = requests.get(url).content
    try:
        return response.decode('utf-8')
    except UnicodeDecodeError:
        # Обработать исключительную ситуацию
```

Обработчик, добавленный в `Deferred` вызовом его метода `addCallback`, выполняется, только если не возникло никаких исключений, а значит, является асинхронным эквивалентом блока `try`. Мы можем добавить обработку ошибок, предусмотрев аналогичный обратный вызов для блока `except`, который принимает возникшее исключение в виде аргумента. Такой обратный вызов, который вызывается с исключением, называется *обработчиком ошибок* (*errback*).

В синхронном коде можно позволить исключению распространиться вверх по стеку вызовов до вызывающего кода, убрав `try` и `except` из обработчика. Однако если исключение возникнет в цикле `for` объекта `Deferred`, оно выйдет за пределы цикла и попадет в `Deferred.callback`. Это неподходящее место для обработки исключений, потому что код, записывающий значение в `Deferred`, не может знать, как правильно обработать ту или иную ошибку, но это знает код, добавивший свои обратные вызовы. Если заключить обработку ошибок в об-

работчики, которые предназначены для этой цели и передаются в экземпляр `Deferred`, объект `Deferred` сможет вызвать их в нужный момент, не беспокоя код, вызвавший `Deferred.callback`.

На каждом шаге в цепочке обратных вызовов цикл должен перехватить любое исключение и переслать его следующему обработчику ошибок. Поскольку на каждом шаге может быть вызван как обработчик результата, так и обработчик ошибок, наш список `_callbacks` вызовов изменится и будет содержать пары `(callback, errback)`:

```
def passthrough(obj):
    return obj

class Deferred(object):
    def __init__(self):
        self._callbacks = []

    def addCallback(self, callback):
        self._callbacks.append((callback, passthrough))

    def addErrback(self, errback):
        self._callbacks.append((passthrough, errback))

    def callback(self, result):
        for callback, errback in self._callbacks:
            if isinstance(result, BaseException):
                handler = errback
            else:
                handler = callback
            try:
                result = handler(result)
            except BaseException as e:
                result = e
```

В каждой итерации цикла проверяется текущий результат. Исключения передаются следующему обработчику ошибок, а все остальное – следующему обработчику результата, как это было раньше. Любое исключение, вызванное любым обработчиком (ошибок или результата), становится текущим результатом и передается далее по цепочке обработчиков. То есть следующий код, использующий `Deferred`:

```
someDeferred = Deferred()
someDeferred.addCallback(callback)
someDeferred.addErrback(errback)
someDeferred.callback(value)
```

эквивалентен синхронному коду:

```
try:
    callback(value)
except BaseException as e:
    errback(e)
```

Обработчики ошибок распространяют исключения, возвращая их, и подавляют, возвращая любое значение, которое *не является* исключением. Следую-

ший код отфильтровывает исключения `ValueError`, позволяя всем остальным распространяться по цепочке обработчиков:

```
def suppressValueError(exception):
    if not isinstance(exception, ValueError):
        return exception

someDeferred.addErrback(suppressValueError)
```

`suppressValueError` неявно возвращает `None`, когда `isinstance(exception, ValueError)` возвращает `True`, поэтому следующий обработчик в цикле `Deferred` получит `None`. Любое другое исключение, которое вернет функция `suppressValueError` в цикле `for`, будет передано следующему обработчику ошибок. Этот код действует подобно следующей синхронной конструкции:

```
try:
    callback(value)
except ValueError:
    pass
```

Удобство нового потока управления в `Deferred` становится очевидным, если рассмотреть два места, где можно встретиться с исключениями:

- 1) исключение может вызвать любой обработчик результата в списке обратных вызовов в объекте `Deferred`. Например, ошибка в последовательности обратных вызовов в функции `manyCallback` может привести к тому, что `addValue` вернет `None`, и тогда `divideValue` вызовет исключение `TypeError`;
- 2) исключение может вызвать код, который передает фактический результат в вызов метода `callback` объекта `Deferred`. Представьте, например, что `nonblockingGet` пытается декодировать тело ответа HTTP как UTF-8 и передает результат объекту `Deferred`. Если тело содержит текст в какой-то другой кодировке, отличной от UTF-8, возникнет исключение `UnicodeDecodeError`. Это исключение означает невозможность получить фактическое значение, и о такой неустранимой ошибке обработчики в `Deferred` должны знать.

Теперь `Deferred` обрабатывает оба случая; первый обрабатывается путем вызова каждого следующего обработчика (результата или ошибки) внутри блока `try`, а второй можно обработать, перехватывая и пересылая исключения в `Deferred.callback`. Рассмотрим реализацию протокола HTTP, которая вызывает `Deferred.callback` и передает ему тело ответа, после декодирования из кодировки UTF-8:

```
class HTTP(protocol.Protocol):
    def dataReceived(self, data):
        self._handleData(data)
        if self.state == "BODY_READY":
            try:
                result = data.decode('utf-8')
            except Exception as e:
                result = e
```

```

        self.factory.deferred.callback(result)

class HTTPFactory(protocol.Factory)
    protocol = HTTP
    def __init__(self, deferred):
        self.deferred = deferred

def nonblockingGet(url):
    deferred = Deferred()
    factory = HTTPFactory(deferred)
    ...
    return deferred

```

Каждая итерация цикла `for` в объекте `Deferred` начинается с проверки природы текущего результата. В первой итерации результатом является любое значение, переданное кодом, вызвавшим метод `callback`; если возникнет исключение, код выше передаст это исключение.

Обработку исключений теперь можно сосредоточить в обработчиках ошибок, по аналогии с тем, как в обычных обработчиках сосредоточена обработка допустимых значений. Это позволяет нам перейти от синхронного управления потоком выполнения к асинхронному. Следующий код:

```

def requestField(url, field):
    results = requests.get(url).json()
    return results[field]

def manyOperations(url):
    result = requestField(url, field)
    try:
        result += 2
        result //= 3
        result *= 4
    except TypeError:
        return -1
    return result

```

превращается в:

```

def manyCallbacks(url):
    def addValue(result):
        return result + 2
    def divideValue(result):
        return result // 3
    def multiplyValue(result):
        return result * 4
    def onTypeError(exception):
        if isinstance(exception, TypeError):
            return -1
        else:
            return exception

    deferred = requestField(url, "someInteger")
    deferred.addCallback(addValue)
    deferred.addCallback(divideValue)

```

```
deferred.addCallback(multiplyValue)
deferred.addErrback(onTypeError)
return deferred
```

Twisted предлагает свою реализацию Deferred – более обширную версию, чем показанная здесь; как мы увидим в следующем разделе, настоящий класс Deferred поддерживает композицию с самим собой и предлагает дополнительные возможности, такие как тайм-ауты и отмена. Однако его поведение в своей основе соответствует нашей игрушечной реализации.

ВВЕДЕНИЕ В TWISTED DEFERRED

Лучший способ поближе познакомиться с классом Deferred в фреймворке Twisted – поиграть с ним в интерактивном сеансе Python. Для этого прежде всего нужно импортировать его из модуля `twisted.internet.defer`:

```
>>> from twisted.internet.defer import Deferred
```

Обычные обработчики

Так же как наша реализация, класс `twisted.internet.defer.Deferred` имеет метод `addCallback`, принимающий обработчик и добавляющий его в список обработчиков, хранящийся в данном экземпляре Deferred. Кроме того, этот метод в Twisted принимает также позиционные и именованные аргументы для передачи в обработчик:

```
>>> d = Deferred()
>>> def cbPrint(result, positional, **kwargs):
...     print("result =", result, "positional =", positional,
...           "kwargs =", kwargs)
...
>>> d.addCallback(cbPrint, "positional", keyword=1) is d
True
>>> d.callback("result")
result = result positional = positional kwargs = {'keyword': 1}
```

Здесь был создан экземпляр Deferred с именем `d`, в него добавлен обработчик `cbPrint` и затем вызван метод `d.callback` с аргументом `"result"`. Экземпляр `d` передал эту строку обработчику `cbPrint` в первом позиционном аргументе, а также остальные аргументы, указанные в вызове `d.addCallback`.

Обратите внимание, что `d.addCallback` вернул ссылку на сам экземпляр `d`, что позволяет составлять такие цепочки:

```
d.addCallback(...).addCallback(...).addCallback(...).
```

Далее, после первого вызова метода `d.callback`, его нельзя вызвать вновь:

```
>>> d.callback("whoops")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "site-packages/twisted/internet/defer.py", line 459, in callback
```

```

    self._startRunCallbacks(result)
File "site-packages/twisted/internet/defer.py", line 560,
in _startRunCallbacks
    raise AlreadyCalledError
twisted.internet.defer.AlreadyCalledError

```

Объясняется это тем, что Deferred запоминает факт обработки значения:

```

>>> d2 = Deferred()
>>> d2.callback("the result")
<Deferred at 0x12345 current result: 'the result'>

```

Теперь, когда мы узнали, что Deferred сохраняет результат, возникает вопрос: что случится, если в экземпляре Deferred с уже имеющимся результатом добавить еще один обработчик?

```

>>> d2.addCallback(print)
the result

```

Как видите, функция `print` была выполнена немедленно после ее добавления в список обработчиков `d2`. Объекты `Deferred`, уже хранящие результат, немедленно вызывают вновь добавляемые обработчики. Заманчиво думать, что `Deferred` всегда представляет значение, даже недоступное. Однако это ошибочное предположение, и код, опирающийся на него, является потенциальным источником трудноуловимых ошибок. Например:

```

class ReadyOK(twisted.internet.protocol.Protocol):
    def connectionMade(self):
        someDeferred = someAPI()
        def checkAndWriteB(ignored):
            self.transport.write(b"OK\n")
        someDeferred.addCallback(checkAndWriteB)
        self.transport.write(b"READY\n")

```

Как можно заключить из имени, протокол `ReadyOK` должен поприветствовать новое соединение строкой `READY`, а когда `someAPI` вызовет метод `callback` своего экземпляра `Deferred` – вывести `OK` и разорвать соединение. Строка `READY` появится раньше строки `OK`, если `someDeferred` не получит результат до завершения `connectionMade`, но это не гарантируется; если `someAPI` вернет `someDeferred` с результатом, тогда строка `OK` появится раньше строки `READY`. Такое нарушение порядка следования строк может вызвать ошибки на стороне клиентов, требующих, чтобы строка `READY` появилась первой.

Чтобы исправить проблему, нужно поместить вызов `self.transport.write(b"READY\n")` *перед* инструкцией `someDeferred = someAPI()`. Возможно, вам придется аналогично реорганизовать свой код, чтобы гарантировать, что экземпляры `Deferred` с готовыми результатами не нарушают инварианты.

Ошибки и их обработчики

Экземплярам `Deferred` также можно передать обработчики ошибок на случай появления исключений в обработчиках результатов и в коде, вызывающем `Deferred.callback`. Сначала рассмотрим первый случай:

```

>>> d3 = Deferred()
>>> def cbWillFail(number):
...     1 / number
...
>>> d3.addCallback(cbWillFail)
<Deferred at 0x123456>
>>> d3.addErrback(print)
<Deferred at 0x123456>
>>> d3.callback(0)
[Failure instance: Traceback: <class 'ZeroDivisionError'>: division by zero
<stdin>:1:<module>
site-packages/twisted/internet/defer.py:459:callback
site-packages/twisted/internet/defer.py:567:_startRunCallbacks
--- <exception caught here> ---
site-packages/twisted/internet/defer.py:653:_runCallbacks
<stdin>:2:cbWillFail
]

```

В экземпляр `d3` класса `Deferred` добавлен обработчик результата, который делит 1 на свой аргумент, и обработчик ошибок – встроенная функция `print`, – которая выведет информацию об исключении, вызванном обработчиком результата. Если методу `d3.callback` передать 0, это, естественно, вызовет ошибку `ZeroDivisionError`, а также создаст экземпляр `Failure`. Обратите внимание, что строковое представление `Failure` заключено в квадратные скобки (`[...]`). Обработчик ошибок в этом примере напечатал одну ошибку, а не список `list` с одним экземпляром `Failure`!

Объекты исключений в Python 2 не содержат ни трассировки стека, ни какой-либо другой информации об источнике ошибки. С целью дать как можно больше контекстной информации в фреймворк Twisted добавлен класс `Failure` – контейнер для асинхронных исключений, куда записывается трассировка стека. Экземпляр `Failure` создается в блоке `except`, и в него помещаются возникшее исключение и трассировка стека:

```

>>> from twisted.python.failure import Failure
>>> try:
...     1 / 0
... except:
...     f = Failure()
...
>>> f
<twisted.python.failure.Failure builtins.ZeroDivisionError: division by zero>
>>> f.value
ZeroDivisionError('division by zero',)
>>> f.getTracebackObject()
<traceback object at 0x1234567>
>>> print(f.getTraceback())
Traceback (most recent call last):
--- <exception caught here> ---
  File "<stdin>", line 2, in <module>
builtins.ZeroDivisionError: division by zero

```

Экземпляр `Failure` сохраняет объект фактического исключения в своем атрибуте `value` и сам генерирует трассировку стека некоторым способом.

Также экземпляры `Failure` имеют вспомогательные методы, упрощающие анализ исключений в обработчиках ошибок. Метод `check` принимает несколько классов исключений и возвращает тот из них, который соответствует исключению в экземпляре `Failure` или `None`:

```
>>> f.check(ValueError)
>>> f.check(ValueError, ZeroDivisionError)
<class 'ZeroDivisionError'>
```

`Failure.trap` действует подобно `check`, но дополнительно повторно вызывает исключение, зафиксированное в `Failure`, если оно не соответствует ни одному из указанных классов исключений. Это позволяет обработчикам ошибок воспроизводить поведение предложений `except`, в которых указаны классы исключений:

```
>>> d4 = Deferred()
>>> def cbWillFail(number):
...     1 / 0
...
>>> def ebValueError(failure):
...     failure.trap(ValueError):
...     print("Failure was ValueError")
...
>>> def ebTypeErrorAndZeroDivisionError(failure):
...     exceptionType = failure.trap(TypeError, ZeroDivisionError):
...     print("Failure was", exceptionType)
...
>>> d4.addCallback(cbWillFail)
<Deferred at 0x12345678>
>>> d4.addErrback(ebValueError)
<Deferred at 0x12345678>
>>> d4.addErrback(ebTypeErrorAndZeroDivisionError)
<Deferred at 0x12345678>
>>> d4.callback(0)
Failure was <class 'ZeroDivisionError'>
```

Функции `ebValueError` и `ebTypeErrorAndZeroDivisionError` действуют подобно двум блокам `except` в синхронном коде:

```
try:
    1/0
except ValueError:
    print("Failure was ValueError")
except (TypeError, ZeroDivisionError) as e:
    exceptionType = type(e)
    print("Failure was", exceptionType)
```

Наконец, экземпляры `Deferred` могут принимать экземпляры `Failure` или синтезировать их из текущего исключения.

Если в метод `callback` экземпляра `Deferred` передать экземпляр `Failure`, он вызовет свои обработчики ошибок. То есть вызов `someDeferred.callback(Failure())` является аналогом передачи исключения в метод `callback` в нашей реализации `Deferred`.

В классе `Deferred` имеется также метод `errback`, передача экземпляра `Failure` которому имеет тот же эффект, что и передача `Failure` методу `callback`. Разница лишь в том, что если вызвать `Deferred.errback` без аргументов, он сконструирует экземпляр `Failure`, облегчив захват исключения для асинхронной обработки:

```
>>> d5 = Deferred()
>>> d5.addErrback(print)
<Deferred at 0x12345678>
>>> try:
...     1/0
... except:
...     d.errback()
...
[Failure instance: Traceback:< class 'ZeroDivisionError': division
by zero
---<exception caught here>---
<stdin>:2:<module>
]
```

Композиция экземпляров Deferred

Класс `Deferred` – это абстракция управления потоком выполнения, поддерживающая возможность композиции обработчиков результатов и ошибок. Но, кроме этого, он поддерживает возможность композиции самих экземпляров `Deferred`, заставляя их ждать друг друга.

Рассмотрим экземпляр `Deferred` с именем `outerDeferred` со следующей последовательностью обработчиков результата, один из которых возвращает экземпляр `innerDeferred`, имеющий свои обработчики результата:

```
>>> outerDeferred = Deferred()
>>> def printAndPassThrough(result, *args):
...     print("printAndPassThrough",
...           " ".join(args), "received", result)
...     return result
...
>>> outerDeferred.addCallback(printAndPassThrough, '1')
<Deferred at 0x12345678>
>>> innerDeferred = Deferred()
>>> innerDeferred.addCallback(printAndPassThrough, '2', 'a')
<Deferred at 0x123456789>
>>> innerDeferred.addCallback(printAndPassThrough, '2', 'b')
<Deferred at 0x123456789>
>>> def returnInnerDeferred(result, number):
...     print("returnInnerDeferred #", number, "received", result)
...     print("Returning innerDeferred...")
...     return innerDeferred
```

```

...
>>> outerDeferred.addCallback(returnInnerDeferred, '2')
<Deferred at 0x12345678>
>>> outerDeferred.addCallback(printAndPassThrough, '3')
<Deferred at 0x12345678>
>>> outerDeferred.addCallback(printAndPassThrough, '4')
<Deferred at 0x12345678>

```

Очевидно, что вызов метода `outerDeferred.callback`, в свою очередь, вызовет обработчик `printAndPassThrough` с идентификатором 1, но что случится, когда выполнение достигнет обработчика `returnInnerDeferred`?

Ответ на этот вопрос вы найдете на рис. 2.1.

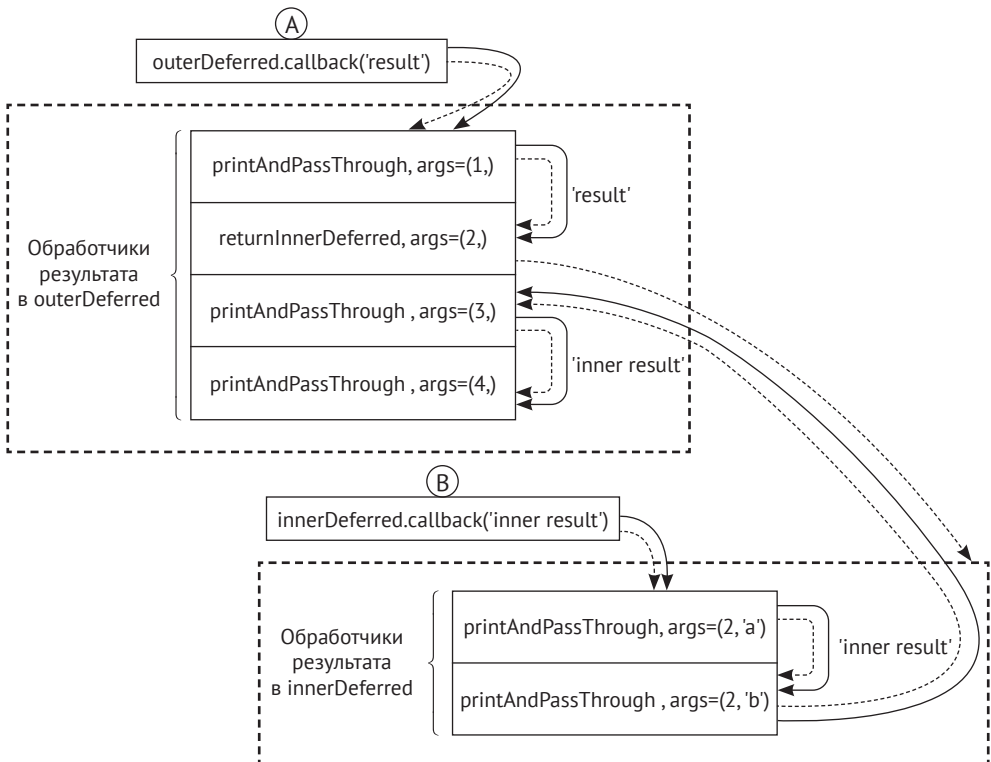


Рис. 2.1 ❖ Поток выполнения и данных между `outerDeferred` и `innerDeferred`. Поток выполнения обозначен пунктирными стрелками, а поток данных – сплошными

Блок с меткой **A** представляет вызов `outerDeferred.callback('result')`, который запускает цикл по обработчикам результата в `outerDeferred`; пунктирные и сплошные стрелки показывают потоки выполнения и данных соответственно.

Первым будет вызван обработчик `printAndPassThrough` с идентификатором 1 – он получит строку `'result'` в первом аргументе и выведет сообщение. По-

сколько он возвращает строку 'result', externalDeferred вызовет следующий обработчик с этой строкой. returnInnerDeferred выведет свой идентификатор и сообщение и вернет innerDeferred:

```
>>> outerDeferred.callback("result")
printAndPassThrough 1 received result
returnInnerDeferred 2 received result
Returning innerDeferred...
```

Цикл обхода обработчиков в outerDeferred обнаружит, что вместо фактического значения returnInnerDeferred вернул экземпляр Deferred, и приостановит выполнение, пока innerDeferred не получит готовый результат. Пунктирная стрелка на рис. 2.1 показывает, как управление передается в innerDeferred. А вот что дает попытка получить представление экземпляра outerDeferred:

```
>>> outerDeferred
<Deferred at 0x12345678 waiting on Deferred at 0x123456789>
```

Блок с меткой **В** представляет вызов innerDeferred.callback('result'), который возобновляет выполнение. Естественно, innerDeferred имеет свой список обработчиков: printAndPassThrough с идентификаторами 2 а и 2 b, которые он вызывает в цикле.

Выполнив все свои обработчики, innerDeferred вернет управление циклу обработки в outerDeferred, который последовательно вызовет printAndPassThrough с идентификаторами 3 и 4 и значением, полученным от последнего обработчика в списке innerDeferred.

```
>>> innerDeferred.callback('inner result')
printAndPassThrough 2 a received inner result
printAndPassThrough 2 b received inner result
printAndPassThrough 3 received inner result
printAndPassThrough 4 received inner result
```

Обработчики printAndPassThrough с идентификаторами 3 и 4 фактически превратились в обработчики результата, полученного экземпляром innerDeferred. Если какой-то из обработчиков в innerDeferred, в свою очередь, вернет экземпляр Deferred, его цикл обработки тоже приостановится, как и в externalDeferred.

Возможность возвращать экземпляры Deferred из обработчиков результата (и ошибок) дает возможность внешней композиции функций, возвращающих экземпляры Deferred:

```
def copyURL(sourceURL, targetURL):
    downloadDeferred = retrieveURL(sourceURL)
    def uploadResponse(response):
        return uploadToURL(targetURL, response)
    return downloadDeferred.addCallback(uploadResponse)
```

Функция copyURL использует две гипотетические функции: retrieveURL, которая загружает содержимое, находящееся по заданному адресу targetURL; и uploadToURL, которая выгружает данные по заданному адресу targetURL. Об-

работчик `uploadResponse`, добавленный в экземпляр `Deferred`, возвращающийся функцией `retrieveURL`, вызывает функцию `uploadToURL`, передает ей данные из исходного URL и возвращает полученный экземпляр `Deferred`. Не забывайте, что метод `addCallback` возвращает экземпляр `Deferred`, для которого он был вызван, поэтому `copyURL` вернет `downloadDeferred`.

Код, вызвавший `copyURL`, сначала дожждется окончания загрузки, а затем выгрузки, как и предполагалось. Реализация `copyURL` комбинирует функции, возвращающие экземпляры `Deferred`, точно так же, как обработчики, – без применения каких-либо специальных методов.

Базовый интерфейс `Deferred` в `Twisted` позволяет использовать прием внешней композиции для объединения обработчиков результатов и ошибок, а также других экземпляров `Deferred`, упрощая создание асинхронных программ.

Но класс `Deferred` – не единственный способ внешней композиции асинхронных вычислений. За почти два десятилетия, прошедших с момента появления класса `Deferred` в фреймворке `Twisted`, в Python появились механизмы уровня языка, позволяющие приостанавливать и возобновлять выполнение функций специальных типов.

ГЕНЕРАТОРЫ И INLINECALLBACKS

yield

В Python 2.5 появилась поддержка *генераторов*. Генераторы – это особые функции и методы, использующие выражение `yield`. Вызов функции-генератора возвращает итерируемый *объект генератора*. Попытка выполнить итерации с ним приведет к выполнению тела генератора до следующего выражения `yield`, после чего выполнение приостановится и итератор вернет операнд выражения `yield`.

Рассмотрим порядок выполнения следующей функции-генератора:

```
>>> def generatorFunction():
...     print("Begin")
...     yield 1
...     print("Continue")
...     yield 2
...
>>> g = generatorFunction()
>>> g
<generator object generatorFunction at 0x12345690>
>>> result = next(g)
Begin
>>> result
1
```

`generatorFunction` возвращает новый объект генератора. Обратите внимание, что при вызове функции `generatorFunction` никакая часть ее тела не выполняется – она просто возвращает объект генератора. Встроенная функция `next` сдви-

гает итератор, объект генератора `g` начинает выполнять тело функции `generatorFunction` и выводит строку `Begin`. Встретив первое выражение `yield`, генератор приостанавливает выполнение и возвращает значение, указанное в `yield`. Повторный вызов `next` возобновляет выполнение генератора, которое продолжается до следующего выражения `yield`:

```
>>> nextResult = next(g)
Continue
>>> nextResult
2
```

Следующий вызов `next` возобновит выполнение генератора. Но на этот раз просто произойдет выход из тела функции. Поскольку никаких других выражений `yield` в функции нет, где выполнение могло бы приостановиться, объект генератора ничего не вернет. В соответствии с протоколом итераций в языке Python вызов `next` для объекта-генератора, завершившего выполнение, возбуждает исключение `StopIteration`, чтобы сообщить о его исчерпании:

```
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

То есть генераторы поддерживают тот же программный интерфейс, что и любые другие итераторы: значения возвращаются либо явно, через вызов `next`, как показано выше, либо неявно, как в циклах `for`, и исключение `StopIteration` сообщает об исчерпании значений. Однако генераторы поддерживают не только программный интерфейс итераторов.

send

Генераторы способны не только возвращать, но и *принимать* значения. Операнд `yield` можно использовать справа от оператора присваивания. Выражение `yield`, на котором генератор приостанавливает выполнение, может принимать значения, переданные генератору вызовом его метода `send`. Рассмотрим следующий генератор `gPrime` с выражением `yield`:

```
def gPrime():
    a = yield 4
```

Вызов `gPrime.send(5)` заставит выражение `yield` справа от оператора присваивания вернуть число 5, то есть код внутри генератора будет эквивалентен следующему коду:

```
def gPrime():
    a = 5
```

В результате локальная переменная `a` в теле генератора получит значение 5. Кроме того, вызов `gPrime().send(5)` заставит генератор выполнить шаг и вернет 4. Давайте подробнее рассмотрим, как действует `send`, на полноценном примере, схема работы которого представлена также на рис. 2.2.

```

>>> def receivingGenerator():
...     print("Begin")
...     x = 1
...     y = yield x
...     print("Continue")
...     z = yield x + y
...     print(x + y + z)
...
>>> g = receivingGenerator()
>>> result = next(g) # A
Begin
>>> result
1
>>> nextResult = g.send(2) # B
Continue
>>> nextResult
3
>>> g.send(3) # C
6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

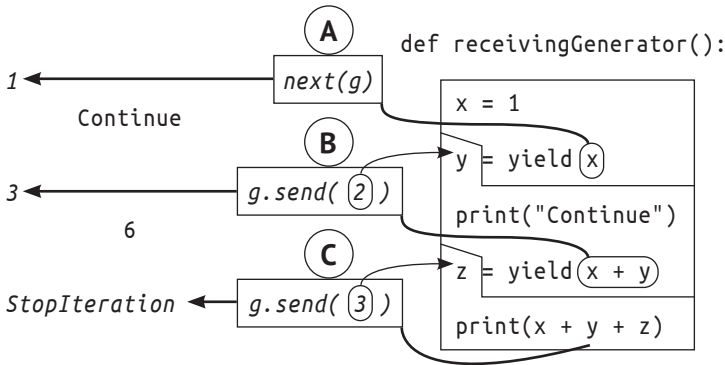


Рис. 2.2 ❖ Потоки выполнения и данных в `receivingGenerator`.
Выполнение протекает сверху вниз, сплошными стрелками показано,
как перемещаются данные

Выполнение генератора `receivingGenerator` начинается с первого вызова `next`, так же как в примере с генератором `generatorFunction`; генераторы всегда должны запускаться однократной попыткой выполнить итерацию. Прямоугольник с меткой **A** на рис. 2.2 соответствует первому вызову `next`. Как и в предыдущем примере, `g` выполняется, пока не встретит первое выражение `yield`, и этот вызов `next` вычислит операнд `yield`. Так как в данном случае таким операндом является локальная переменная `x`, которая имеет значение 1, вызов `next` вернет 1. Стрелка, идущая от прямоугольника с выражением `yield x` через прямоугольник с меткой **A**, показывает, как значение 1 покидает генератор через вызов `next`.

Теперь, когда генератор запущен, мы можем возобновить его выполнение вызовом метода `send`, который показан на рис. 2.2 в прямоугольнике с меткой **В**. Вызов `g.send(2)` передаст значение 2 в генератор, который, в свою очередь, присвоит его переменной `y`. Затем выполнится инструкция `print("Continue")`, и генератор приостановится на следующем выражении `yield`. Это выражение вычислит свой операнд `x + y`, получит значение 3 и вернет его как результат вызова `g.send(2)`. Стрелка, идущая от прямоугольника с выражением `x + y` через прямоугольник с меткой **В**, показывает, как значение 3 покидает генератор.

Вызов `g.send(3)`, который показан на рис. 2.2 в прямоугольнике с меткой **С**, передаст 3 в генератор и снова возобновит его выполнение. На этот раз генератор выведет результат `x + y + z = 6`. Но теперь он не приостановится, как на предыдущих шагах, потому что в нем больше нет выражений `yield`. Следуя протоколу итераций, генераторы возбуждают исключение `StopIteration` при исчерпании, поэтому вызов `g.send(3)` возбудит исключение `StopIteration`, как показано на рис. 2.2 и в примере сеанса.

throw

Подобно тому, как метод `send` позволяет передавать значения в генераторы, метод `throw` позволяет возбудить исключение в них. Взгляните на следующий код:

```
>>> def failingGenerator():
...     try:
...         value = yield
...     except ValueError:
...         print("Caught ValueError")
...
>>> tracebackG = failingGenerator()
>>> next(tracebackG)
>>> tracebackG.throw(TypeError())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in failingGenerator
TypeError
>>> catchingG = failingGenerator()
>>> next(catchingG)
>>> catchingG.throw(ValueError())
Caught ValueError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

В генераторе `failingGenerator` выражение `yield` заключено в блок `try`, который перехватывает исключение `ValueError` и выводит сообщение. Если передать этому генератору любое другое исключение, оно будет возвращено в вызывающий код.

Мы создали новый объект генератора `failingGenerator` с именем `tracebackG` и, как обычно, запустили его вызовом `next`. Обратите внимание, что выражение `yield` в `failingGenerator` не имеет операнда. В таких случаях Python подставляет

на место отсутствующего операнда значение `None`, поэтому `next` вернет `None` (интерактивный сеанс Python не выводит `None`, получив его от функции). Внутри генератора первое выражение `yield` вернет `None`, потому что функция `next` не передает в генератор никаких значений. То есть вызов `next(g)` эквивалентен вызову `g.send(None)`. Эта эквивалентность пригодится нам, когда мы приступим к изучению *сопрограмм*.

Далее, вызовом метода `throw` мы возбуждаем исключение `TypeError` в `tracebackG`. Генератор возобновляет выполнение с точки, где встретил выражение `yield`, но вместо вычисления значения `yield` возбудит исключение `TypeError`, переданное методом `throw`. Появившаяся трассировка стека показывает, что `failingGenerator` прервал выполнение и вернул сгенерированное исключение `TypeError` обратно из вызова `tracebackG.throw`. В этом есть определенный смысл: вызов `throw` возобновил выполнение генератора, который, в свою очередь, возбудил исключение `TypeError`, и, естественно, необработанное исключение распространилось вверх по стеку вызовов.

Новый генератор, с именем `catchingG`, демонстрирует, что случится, если блок `except` в `failingGenerator` встретит исключение `ValueError`. Как и ожидалось, выражение `yield` возбудило исключение, переданное в вызов `throw`, и в соответствии с механизмом обработки исключений в Python блок `except` перехватил его и вывел сообщение. Однако далее в теле генератора нет других выражений `yield`, на которых генератор мог бы приостановиться, поэтому возбуждается еще одно исключение, `StopIteration`, сообщающее об исчерпании `failingGenerator`.

Асинхронное программирование с inlineCallbacks

Приостановку и возобновление выполнения генераторов можно сравнить с выполнением обработчиков результатов и ошибок в `Deferred`:

- генератор приостанавливает выполнение, достигнув выражения `yield`, а `Deferred` приостанавливает выполнение своих обработчиков, когда один из них возвращает другой экземпляр `Deferred`;
- выполнение приостановленного генератора можно возобновить, передав значение через его метод `send`, а `Deferred` ожидает, пока другой `Deferred` не выполнит свои обработчики результата, получив его;
- приостановленный генератор может принимать исключения через метод `throw` и перехватывать их, а `Deferred` ожидает, пока другой `Deferred` не выполнит свои обработчики ошибок, получив исключение.

Чтобы лучше понять эту эквивалентность, сравним два примера кода:

```
def requestFieldDeferred(url, field):
    d = nonblockingGet(url)

    def onCompletion(response):
        document = json.load(response)
        return document[field]

    def onFailure(failure):
        failure.trap(UnicodeDecodeError)
```

```
d.addCallback(onCompletion)
d.addErrback(onFailure)

return d

def requestFieldGenerator(url, field):
    try:
        document = yield nonblockingGet(url)
    except UnicodeDecodeError:
        pass

    document = json.load(response)
    return document[field]
```

`requestFieldDeferred` добавляет в экземпляр `Deferred`, возвращаемый функцией `nonblockingGet`, обработчик результата, который декодирует ответ в формате JSON и извлекает свойство `field`, и обработчик ошибок, который подавляет исключение `UnicodeDecodeError`.

`requestFieldGenerator`, напротив, возвращает экземпляр `Deferred`, полученный от функции `nonblockingGet`. Когда ответ станет доступен или возникнет исключение, выполнение генератора можно будет возобновить. Оба обработчика, результата и ошибок, находятся в одной области видимости с вызовом `nonblockingGet`. Включение тела функции в вызывающий ее код называют *встраиванием (inlining)*.

Мы не сможем использовать `requestFieldGenerator` в текущем его виде: Python 2 не позволяет функциям-генераторам возвращать значения, поэтому нам нужна какая-то обертка, которая иницирует выполнение генератора, примет экземпляр `Deferred` и вызовет метод `send` или `throw` генератора, когда `Deferred` получит ожидаемое значение или исключение.

Фреймворк Twisted предлагает такую обертку в виде `twisted.internet.defer.inlineCallbacks`. Она декорирует вызываемые объекты, возвращаемые функциями-генераторами, иницирует выполнение генератора и вызывает его метод `send` или `throw`, когда экземпляр `Deferred` получит значение или исключение. Код, вызывающий декорированную функцию-генератор, в свою очередь, получит экземпляр `Deferred` вместо объекта генератора. Таким образом, `inlineCallbacks` гарантирует бесшовную совместимость с существующими API, которые ожидают получить `Deferred`.

Вот как выполнить декорирование нашего генератора `requestFieldGenerator` с помощью `inlineCallbacks`:

```
from twisted.internet import defer

@defer.inlineCallbacks
def requestFieldGenerator(url, field):
    try:
        document = yield nonblockingGet(url)
    except UnicodeDecodeError:
        pass

    document = json.load(response)
    defer.returnValue(document[field])
```

```
def someCaller(url, ...):
    requestFieldDeferred = requestFieldGenerator(url, "someProperty")
    ...
```

Функция `returnValue` возбуждает специальное исключение со своим аргументом; `inlineCallbacks` перехватывает это исключение и вызывает `requestFieldGenerator` с полученным значением. Инструкция `return` в Python 3 возбуждает эквивалентное исключение, которое также перехватывается в `inlineCallbacks`, поэтому вызов `returnValue` можно опустить, если известно, что код будет выполняться только под управлением Python 3.

Позволяя включать обработчики результатов и ошибок в одну локальную область, генераторы делают асинхронные программы на основе Twisted простыми для понимания и похожими на синхронные программы. Особенно от такого способа определения функций и потока управления выигрывают короткие программы.

Однако генераторы приносят новые проблемы. Самая большая из них заключается в том, что код, вызывающий функцию или метод-генератор, не может знать, будет ли полученный ими объект-генератор использовать значение, отправленное с помощью `send`, или молча проигнорирует его. Например, вот два генератора, предлагающих один и тот же интерфейс:

```
def listensToSend():
    a = 1
    b = yield a
    print(a+b)

def ignoresSend():
    a = 1
    yield a
    print(a)
```

Случайная замена `listensToSend` на `ignoreSend` приведет к трудноуловимой ошибке. Обе реализации являются допустимыми в Python, но предназначены для использования в разных обстоятельствах: `listensToSend` поддерживает возможность возобновления выполнения с передачей значения извне и может декорироваться декоратором `inlineCallbacks`, а `ignoreSend` просто возвращает значение, подобно конвейеру, обрабатывающему строки в файле одну за другой. Программный интерфейс генераторов в Python делает эти два разных варианта использования неразличимыми.

К счастью, в последние версии Python 3 добавлен новый синтаксис, предусмотренный специально для генераторов в стиле `inlineCallbacks`.

СОПРОГРАММЫ В PYTHON

Генераторы являются особым случаем *сопрограмм*¹, которые могут приостанавливаться, давая возможность поработать другим сопрограммам, и возоб-

¹ Их иногда называют *корутинами* (от англ. *coroutines*). – Прим. перев.

новляться при получении значения. Наш генератор, декорированный декоратором `inlineCallbacks`, похож на сопрограммы своей способностью возвращать и принимать значения, но, в отличие от сопрограмм, он не может напрямую вызвать другой генератор, подобно обычной функции. Для передачи управления другому генератору он должен использовать специальный механизм внутри `inlineCallbacks`. Этот механизм, управляющий запросами на выполнение и возвращающий результаты инициатору, известен как *трамполайнинг* (от англ. *trampoline* – трамплин). Чтобы понять, почему он получил такое необычное название, представьте, что управление как бы отскакивает от `inlineCallbacks` между разными генераторами.

Сопрограммы с выражением `yield from`

В Python 3.3 появился новый синтаксис, позволяющий генератору напрямую делегировать выполнение другому генератору: `yield from`. Следующий код, работающий только под управлением Python 3.3+, демонстрирует поведение генератора, возвращающего значение, полученное из другого генератора:

```
>>> def e():
...     a = yield 1
...     return a + 2
...
>>> def f():
...     print("Begin f")
...     c = yield from e()
...     print(c)
...
>>> g = f()
>>> g.send(None)
Begin f
1
>>> g.send(2)
4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

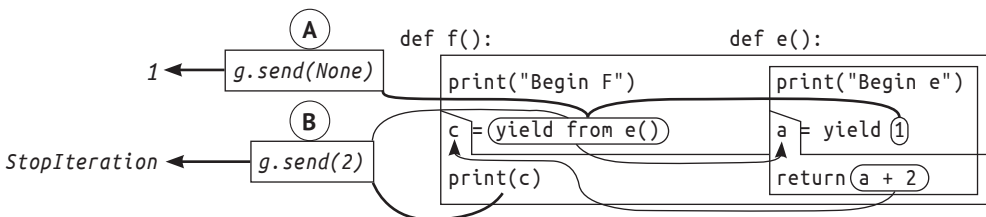


Рис. 2.3 ❖ Потоки выполнения и данных между генераторами `e` и `f`.
Выполнение протекает сверху вниз, а потоки данных
показаны сплошными стрелками

Генератор `e` действует в точности как функции-генераторы, описанные в предыдущем разделе: если вызвать ее и затем передать полученный объект генератора функции `next` (или вызвать его метод `send` с аргументом `None`), он вернет 1 – операнд в выражении `yield`; после этого можно передать значение в генератор с помощью метода `send`, и тот вернет операнд следующего выражения `yield` или оператора `return` (напомню, что в Python 3 генераторы могут возвращать значения).

Объект-генератор `g`, возвращаемый функцией `f`, *возвращает значение из генератора*, полученного вызовом `e`, приостанавливаясь на некоторое время, чтобы дать возможность выполниться подчиненному генератору. Вызовы `next`, `send` и `throw` для `g` передаются в подчиненный генератор `e`, поэтому генератор `g` выглядит в точности как генератор `e`. На рис. 2.3 прямоугольник с меткой **A** представляет начальный вызов `g.send(None)`, который запускает выполнение `g`. Выполнение передается через `yield from` в функции `f()` генератору, возвращаемому функцией `e()`, который, в свою очередь, приостанавливается на выражении `yield` и возвращает 1 в `g.send(None)`.

Генератор, передающий управление другому генератору с помощью выражения `yield from`, возобновляется, когда подчиненный генератор вернет значение. Прямоугольник с меткой **B** на рис. 2.3 представляет второй вызов `g.send(2)`, который передает значение 2 через приостановленный генератор `f` в подчиненный генератор `e`, который, в свою очередь, возобновляется и присваивает 2 переменной `a`. Затем выполнение передается оператору `return`, и подчиненный генератор `e` завершает работу со значением 4. Теперь `f` возобновляет работу слева от выражения `yield from` и присваивает полученное значение 4 переменной `c`. После вызова `print` в генераторе не остается выражений `yield` или `yield from` для выполнения, поэтому `f()` завершается, и вызов `g.send(2)` генерирует ошибку `StopIteration`.

Этот синтаксис избавляет от необходимости использовать трамполайнинг в `inlineCallbacks` для передачи вызовов из одного генератора в другой, потому что позволяет генераторам прямо делегировать выполнение другим генераторам. Благодаря выражению `yield from` генераторы в Python действуют подобно настоящим сопрограммам.

Сопрограммы `async` и `await`

К сожалению, выражение `yield from` страдает все той же неопределенностью, что и `yield`: генераторы, принимающие значения, и генераторы, которые их игнорируют, для вызывающего кода выглядят одинаково. Эта неоднозначность была устранена в версиях Python выше 3.5 добавлением новых синтаксических конструкций, основанных на `yield from`, которые помогают отличить сопрограммы: `async` и `await`.

При применении к определению функции или метода спецификатор `async` превращает эту функцию или метод в сопрограмму:

```
>>> async def function(): pass
... 
```

```
>>> c = function()
>>> c
<coroutine object function at 0x9876543210>
```

Сопрограммы, в отличие от генераторов, не поддерживают итерации:

```
>>> list(function())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'coroutine' object is not iterable
>>> next(function())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'coroutine' object is not iterable
```

Подобно генераторам, сопрограммы имеют методы `send` и `throw`, с помощью которых вызывающий код может возобновлять их выполнение:

```
>>> function().send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> function().throw(Exception)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in function
Exception
```

Сопрограммы могут *ждать*, пока выполняются другие сопрограммы, используя семантику, похожую на семантику выражения `yield from` в генераторах:

```
>>> async def returnsValue(value):
...     return 1
...
>>> async def awaitsCoroutine(c):
...     value = await c
...     print(value)
...
>>> awaitsCoroutine(returnsValue(1)).send(None)
1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Этот пример демонстрирует возможность композиции сопрограмм, но вызов сопрограммы, которая немедленно возвращает значение, никак не мотивирует использование `async` и `await` в асинхронном программировании. Нам нужна возможность отправлять произвольные значения в приостановленную сопрограмму, но поскольку целью `async` и `await` является предоставление API, несовместимого с простыми генераторами, мы не сможем использовать `await` для вызова простого генератора, как в `yield from`, или игнорировать операнд, как в `yield`:

```

>>> def plainGenerator():
...     yield 1
...
>>> async def brokenCoroutineAwaitsGenerator():
...     await plainGenerator()
...
>>> brokenCoroutineAwaitsGenerator().send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in brokenCoroutineAwaitsGenerator
TypeError: object generator can't be used in 'await' expression
>>> async def brokenCoroutineAwaitsNothing():
...     await
    File "<stdin>", line 2
        await
        ^
SyntaxError: invalid syntax

```

Чтобы узнать, как возобновлять сопрограммы с передачей значений, вернемся к `yield from`. В предыдущем примере мы использовали `yield from` с другим генератором, благодаря чему вызовы методов `send` и `throw` генератора-обертки делегировались внутреннему генератору. Вполне возможно собрать целую цепочку генераторов, каждый из которых будет делегировать выполнение следующему генератору с помощью `yield from`, но в самом конце такой цепочки должно находиться что-то, что отправит значение обратно. Рассмотрим, например, стек из пяти генераторов (рис. 2.4).

```

>>> def g1(): yield from g2
...
>>> def g2(): yield from g3
...
>>> def g3(): yield from g4
...
>>> def g4(): yield from g5
...
>>> def g5(): yield 1

```

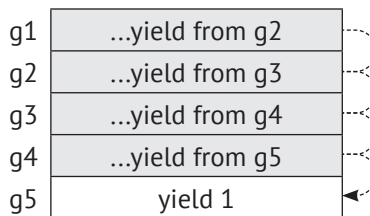


Рис. 2.4 ❖ Стек генераторов. Генераторы `g1`, `g2`, `g3` и `g4` последовательно делегируют выполнение генератору `g5`

`g1`, `g2`, `g3` и `g4` не могут продвинуться вперед, пока `g5` не вернет значение, которое последовательно будет передано от `g4` к `g1`. Однако `g5` необязательно дол-

жен быть генератором; как демонстрирует следующий пример, выражению `yield from` достаточно передать любой *итерируемый объект*:

```
>>> def yieldsToIterable(o):
...     print("Yielding from object of type", type(o))
...     yield from o
...
>>> list(yieldsToIterable(range(3)))
Yielding from object of type <class 'range'>
[0, 1, 2]
```

`yieldsToIterable` делегирует выполнение своему аргументу, роль которого в данном примере играет объект `range`. Выполняя итерации через генератор `yieldsToIterable` в процессе конструирования списка, этот пример демонстрирует, что объект `range` может участвовать в итерациях подобно генератору.

Сопрограммы, объявленные с `async def`, используют ту же реализацию, что и `yield from`, и поэтому способны в выражениях `await` принимать особые типы итерируемых объектов и генераторов.

Вопреки предыдущим примерам генераторы *можно* использовать в выражениях `await`, если снабдить их декоратором `types.coroutine`. Сопрограмма, в которой используется выражение `await` с таким декорированным генератором, получит значение, возвращаемое этим генератором:

```
>>> import types
>>> @types.coroutine
... def makeBase():
...     return (yield "hello from a base object")
...
>>> async def awaitsBase(base):
...     value = await base
...     print("From awaitsBase:", value)
...
>>> awaiter = awaitsBase(makeBase())
>>> awaiter.send(None)
'hello from a base object'
>>> awaiter.send("the result")
From awaits base: the result
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

После запуска вызовом `send(None)` сопрограмма `awaitsBase` выполнит переход к оператору `yield` в генераторе `base` и обычным для генераторов путем вернет строку `"hello from base object"`. Эта сопрограмма делегирует выполнение генератору `base`, поэтому вызов `send("the result")` возобновит выполнение генератора `base` и передаст ему эту строку. `base` немедленно вернет это значение, которое тут же будет получено выражением `await` в сопрограмме.

В выражении `await` также можно использовать итерируемые объекты, если они реализуют специальный метод `__await__`, возвращающий итератор. Конеч-

ное значение этого итератора – то есть значение, полученное последним или завернутое в исключение `StopIteration`, – станет результатом выражения `await`. Объекты, соответствующие этому интерфейсу, называют *future-подобными*. Позже, когда мы приступим к знакомству с механизмом `asyncio`, мы увидим, что этот интерфейс поддерживают его экземпляры `Future`, что и обусловило такое название.

Рассмотрим поток управления на примере простой реализации *future-подобного* объекта:

```
class FutureLike(object):
    _MISSING="MISSING"

    def __init__(self):
        self.result = self._MISSING

    def __next__(self):
        if self.result is self._MISSING:
            return self
        raise StopIteration(self.result)

    def __iter__(self):
        return self

    def __await__(self):
        return iter(self)

async def awaitFutureLike(obj):
    result = await obj
    print(result)

obj = FutureLike()
coro = awaitFutureLike(obj)
assert coro.send(None) is obj
obj.result = "the result"
try:
    coro.send(None)
except StopIteration:
    pass
```

Экземпляры `FutureLike` являются итерируемыми объектами, потому что их метод `__iter__` возвращает объект, имеющий метод `__next__`. В данном случае итерации по экземпляру `FutureLike` будут каждый раз возвращать один и тот же экземпляр, пока его атрибут `result` не получит некоторое значение, после чего первая же итерация сгенерирует исключение `StopIteration` с этим значением. Это эквивалентно возврату значения из генератора.

Кроме того, экземпляры `FutureLike` являются *future-подобными*, потому что их метод `__await__` возвращает итератор, благодаря чему `awaitFutureLike` может передать `await` экземпляр `FutureLike`. Как обычно, сопрограмма запускается вызовом `send(None)`. Он возвращает экземпляр `FutureLike`, который ожидается сопрограммой `awaitFutureLike` и является тем же экземпляром, который мы передали ей. После записи значения в атрибут `result` объекта `FutureLike` мы можем

возобновить сопрограмму, которая получит результат, выведет его и завершится с исключением `StopIteration`.

Обратите внимание, что второй вызов `coro.send` *тоже* передает `None` в сопрограмму. Сопрограммы, ожидающие разрешения `future`-подобных объектов, получают последние значения, предоставленные итераторами этих объектов. Сопрограммы все еще требуется возобновить, чтобы они смогли использовать эти значения, но они всегда игнорируют аргумент метода `send`.

Фреймворк Twisted предлагает объект, поддерживающий протокол ожидания, и адаптирует сопрограммы для беспрепятственного взаимодействия с существующими API. Как мы уже видели, сопрограммы полностью отделены от механизма `asyncio`, поэтому возможностей Twisted API, которые мы обсудили в этом разделе, недостаточно для их интеграции. С дополнительными API, решающими эту проблему, мы познакомимся в следующей главе.

ОЖИДАНИЕ ЗАВЕРШЕНИЯ ЭКЗЕМПЛЯРОВ DEFERRED

Начиная с версии Twisted 16.4.0 класс `Deferred` поддерживает интерфейс `future`-подобных объектов благодаря добавлению методов `__next__`, `__iter__` и `__await__`. Это позволяет заменить экземпляр `FutureLike` в предыдущем примере экземпляром `Deferred`:

```
from twisted.internet.defer import Deferred
```

```
async def awaitFutureLike(obj):
    result = await obj
    print(result)
```

```
obj = Deferred()
coro = awaitFutureLike(obj)
assert coro.send(None) is obj
obj.callback("the result")
try:
    coro.send(None)
except StopIteration:
    pass
```

Выражение `await` получает от экземпляра `Deferred` значение, которое тот вернет в конце обычного цикла обработки:

```
>>> from twisted.internet.defer import Deferred
>>> import operator
>>> d = Deferred()
>>> d.addCallback(print, "was received by a callback")
<Deferred at 0x7eff85886160>
>>> d.addCallback(operator.add, 2)
<Deferred at 0x7eff85886160>
>>> async def awaitDeferred():
...     await d
...
>>> g = awaitDeferred()
```

```
>>> g.send(None)
<Deferred at 0x7eff85886160>
>>> d.callback(1)
1 was received by a callback
>>> g.send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in awaitDeferred
  File "twisted/src/twisted/internet/defer.py", line 746, in send
    raise result.value
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Обработчик нашего экземпляра Deferred – функция `print` – выполняется и возвращает `None`, что приводит к сбою с ошибкой `TypeError` при попытке прибавить 2 к первому аргументу. По этой причине возобновление сопрограммы завершается ошибкой `TypeError`, хранящейся в Deferred.

В этом случае композиция сопрограммы и Deferred привела к ошибке, но данный пример наглядно иллюстрирует, что ошибки и данные естественным образом передаются между ними.

Поддержка протокола `await` в классе Deferred позволяет использовать Twisted API в сопрограммах, но что, если понадобится, чтобы Twisted API использовал одну из наших сопрограмм?

ПРЕОБРАЗОВАНИЕ СОПРОГРАММ В DEFERRED С ПОМОЩЬЮ ENSUREDEFERRED

Фреймворк Twisted позволяет заворачивать сопрограммы в экземпляры Deferred, что дает возможность использовать сопрограммы там, где ожидаются экземпляры Deferred.

`twisted.internet.defer.ensureDeferred` принимает объект сопрограммы и возвращает экземпляр Deferred, который получает значение, возвращаемое сопрограммой:

```
>>> from twisted.internet.defer import Deferred, ensureDeferred
>>> async def asyncIncrement(d):
...     x = await d
...     return x + 1
...
>>> awaited = Deferred()
>>> addDeferred = ensureDeferred(asyncIncrement(awaited))
>>> addDeferred.addCallback(print)
<Deferred at 0x12345>
>>> awaited.callback(1)
2
>>>
```

Наша сопрограмма `asyncIncrement` ждет, пока указанный объект вернет число, затем прибавляет 1 к этому числу и возвращает результат. Мы пре-

образовали ее в экземпляр `Deferred` с помощью `ensureDeferred`, сохранили его в `addDeferred`, а затем добавили обработчик `print`. Вызов метода `callback` экземпляра `awaited`, разрешения которого ожидает сопрограмма `asyncIncrement`, в свою очередь, вызывает метод `callback` экземпляра `addDeferred`, полученного вызовом `sureDeferred`, и при этом нам не потребовалось вызывать метод `send` сопрограммы. Проще говоря, `addDeferred` ведет себя так же, как обычный экземпляр `Deferred`, созданный вручную. Исключения в такой цепочке передаются аналогично:

```
>>>from twisted.internet.defer import Deferred, ensureDeferred
>>> async def asyncAdd(d):
...     x = await d
...     return x + 1
...
>>>awaited = Deferred()
>>>addDeferred = ensureDeferred(asyncAdd(awaited))
>>>addDeferred.addErrback(print)
Unhandled error in Deferred:

<Deferred at 0x7eff857f0470>
>>>awaited.callback(None)
[Failure instance: Traceback: <class 'TypeError'>: ...
...
<stdin>:3:asyncAdd
]
```

Сопрограммы больше напоминают синхронный код, чем обработчики в `Deferred`, и Twisted существенно упрощает их использование, поэтому возникает резонный вопрос: зачем вообще нужны все эти сложности с `Deferred`? Ответ прост: дело в том, что они уже используются; экземпляры `Deferred` используются повсюду в Twisted, поэтому, даже если вы будете применять их очень редко, вам все равно придется с ними ознакомиться. Другая причина, по которой вы можете отказаться от использования сопрограмм, заключается в том, что вам, возможно, придется писать код, совместимый с Python 2. Однако эта проблема постепенно теряет свою актуальность, так как жизненный цикл Python 2 приближается к концу и многие продукты, такие как PyPy – альтернативная среда выполнения Python, включающая динамический компилятор (Just In Time, JIT), способный значительно ускорить выполнение кода на Python, – добавляют поддержку Python 3.

Однако есть менее очевидные и более существенные причины, по которым поддержка объектов `Deferred` в Twisted сохраняет свою ценность.

МУЛЬТИПЛЕКСИРОВАНИЕ ОБЪЕКТОВ DEFERRED

Как поступить, если понадобится получить результаты двух асинхронных операций и одна из них может завершиться раньше другой? Представьте, к примеру, что мы пишем программу, посылающую одновременно два запроса HTTP:

```
def issueTwo(url1, url2):
    urlDeferreds = [retrieveURL(url1), retrieveURL(url2)]
    ...
```

Можно написать сопрограмму, которая позволит дожидаться выполнения обоих запросов:

```
async def issueTwo(url1, url2):
    urlDeferreds = [retrieveURL(url1), retrieveURL(url2)]
    for d in urlDeferreds:
        result = await d
    doSomethingWith(result)
```

Реактор продолжит извлекать url1 и url2, пока issueTwo ожидает завершения любого из них; ожидание, пока будет получено содержимое url1, не мешает реактору запустить извлечение url2. Такая параллельная работа является главным достоинством асинхронного и событийно-ориентированного программирования!

Однако эффективность отходит на второй план с ростом сложности операций. Представьте, что нам нужно вернуть не оба ответа, а тот, что получен первым. Мы не сможем написать сопрограмму `fastestOfTwo`, используя только `await`, потому что не знаем, какая из операций завершится первой. Только реактор может узнать, что значение для сопрограммы готово, анализируя базовые события, и если бы у нас имелись лишь сопрограммы, цикл событий должен был бы предоставить примитив синхронизации, ожидающий сразу несколько сопрограмм и проверяющий их завершение.

К счастью, поддержка отложенных вычислений в фреймворке Twisted дает возможность объединить несколько экземпляров Deferred без применения специальных механизмов синхронизации на уровне реактора. Эта возможность реализована в виде `twisted.internet.defer.DeferredList` – класса, принимающего список экземпляров Deferred и вызывающего свои обработчики, когда все экземпляры Deferred в списке получают значение.

Рассмотрим следующий код:

```
>>> from twisted.internet.defer import Deferred, DeferredList
>>> url1 = Deferred()
>>> url2 = Deferred()
>>> urlList = DeferredList([url1, url2])
>>> urlList.addCallback(print)
<Deferred at 0x123456>
>>> url2.callback("url2")
>>> url1.callback("url1")
[(True, "url1"), (True, "url2")]
```

urlList объединяет два экземпляра Deferred – url1 и url2 – и получает в качестве обработчика функцию print. Этот обработчик выполнится, только когда будут вызваны методы callback обоих объектов в списке, url1 и url2. То есть с данными настройками urlList следует стратегии «все или ничего», подобно сопрограмме issueTwo в примере выше.

Первым признаком, позволяющим судить о широте возможностей, скрытых в `DeferredList`, является список, который он передает своим обработчикам. Каждый элемент списка является кортежем с двумя элементами. Второй элемент кортежа, как нетрудно догадаться, – это значение, полученное экземпляром `Deferred`, причем индекс кортежа в списке с результатами совпадает с индексом соответствующего экземпляра `Deferred` во входном списке, то есть кортеж со вторым элементом "url1" имеет индекс 0 и соответствует экземпляру `url1` с индексом 0 в `DeferredList`.

Первый элемент кортежа служит признаком успешного завершения `Deferred`. В данном случае оба экземпляра, `url1` и `url2`, получили строковые значения, а не `Failure`, поэтому первые элементы кортежей в списке с результатами имеют значение `True`.

Сымитируем ошибку в одном из объектов `Deferred` в списке `DeferredList`, чтобы посмотреть, как происходит передача ошибок:

```
>>> succeeds = Deferred()
>>> fails = Deferred()
>>> listOfDeferreds = DeferredList([succeeds, fails])
>>> listOfDeferreds.addCallback(print)
<Deferred at 0x1234567>
>>> fails.errback(Exception())
>>> succeeds.callback("OK")
[(True, 'OK'), (False, <twisted.python.failure.Failure builtins.Exception:>)]
```

Теперь второй кортеж в возвращаемом списке имеет в первом элементе значение `False` и во втором – экземпляр `Failure`, представляющий исключение `Exception`, заставившее соответствующий экземпляр `Deferred` потерпеть неудачу.

Этот специальный список пар (*признак успеха, значение или ошибка*) сохраняет всю возможную информацию об ошибках, используя механизм захвата трассировки стека в `Failure`. Кроме того, гибкость `DeferredList` позволяет организовать фильтрацию агрегированных результатов в единственном обработчике.

Теперь, познакомившись с поведением `DeferredList` по умолчанию, рассмотрим дополнительную возможность, применение которой поможет нам реализовать `fastestOfTwo: fireOnOneCallback`.

Параметр `fireOnOneCallback` сообщает экземпляру `DeferredList`, что тот должен вызвать свои обработчики, когда хотя бы один из экземпляров `Deferred` в его списке получит значение:

```
>>> noValue = Deferred()
>>> getValue = Deferred()
>>> waitsForOne = DeferredList([noValue, getValue], fireOnOneCallback=True)
>>> waitsForOne.addCallback(print)
<Deferred at 0x12345678>
>>> getValue.callback("the value")
('the value', 1)
```

Теперь `waitForOne` вызовет `print`, когда хотя бы один из экземпляров `Deferred` в его списке получит значение. И снова `DeferredList` передаст своему обработчику кортеж с двумя элементами, но на этот раз первый элемент кортежа будет содержать полученное значение, а второй – индекс соответствующего экземпляра `Deferred` в списке. Экземпляр `getValue` получил значение "the value" и является вторым в списке, переданном в `DeferredList`, поэтому обработчик получил кортеж ("the value", 1).

Теперь мы можем реализовать `fastestOfTwo`:

```
def fastestOfTwo(url1, url2):
    def extractValue(valueAndIndex):
        value, index = valueAndIndex
        return value

    urlList = DeferredList([retrieveURL(url1), retrieveURL(url2)],
                           fireOnOneCallback=True,
                           fireOnOneErrback=True)
    return urlList.addCallback(extractValue)
```

`DeferredList` позволяет также мультиплексировать ошибки, предлагая параметр `fireOnOneErrback`. Вызов обработчиков `DeferredList` при первой же ошибке и ее разворачивание – настолько распространенный шаблон, что для этой цели в `Twisted` был добавлен класс-обертка `twisted.internet.defer.gatherResults`:

```
>>> from twisted.internet.defer import Deferred, gatherResults
>>> d1, d2 = Deferred(), Deferred()
>>> results = gatherResults([d1, d2])
>>> results.addCallback(print)
<Deferred at 0x123456789>
>>> d1.callback(1)
>>> d2.callback(2)
>>> [1, 2]
>>> d1, d2 = Deferred(), Deferred()
>>> fails = gatherResults([d1, d2])
>>> fails.addErrback(print)
<Deferred at 0x1234567890>
>>> d1.errback(Exception())
[[Failure instance: Traceback ...: <class 'Exception'>: ]]
```

Напомню, что метод `__str__` в классе `Failure` возвращает строку, заключенную в квадратные скобки `[]`, поэтому вывод ошибки содержит два набора квадратных скобок: один набор добавлен методом `__str__`, а другой – вмещающим экземпляром `list`.

Отметьте также, что в отсутствие ошибок `gatherResults` ждет, пока завершатся все экземпляры `Deferred`, поэтому его не получится использовать для реализации `fastestOfTwo`.

`DeferredList` и `gatherResults` предлагают высокоуровневый API, помогающий реализовать весьма сложную логику. Однако результат каждого зависит от набора их параметров, а также от результата каждого из экземпляров `Deferred`,

которые они обертывают. Изменение любого из аспектов может привести к неожиданным результатам и нежелательным ошибкам.

Есть еще один важный аспект, касающийся использования `Deferred`: поскольку `Deferred.callback` почти всегда вызывается реактором, а не пользовательским кодом, который косвенно манипулирует сокетом, между точкой появления ошибки и ее причиной может существовать разрыв.

Twisted помогает преодолеть сложности, присущие асинхронному коду, предлагая специальную поддержку тестирования объектов `Deferred`.

ТЕСТИРОВАНИЕ ОБЪЕКТОВ DEFERRED

В предыдущей главе мы видели, что пакет `trial.unittest` в фреймворке Twisted предлагает класс `SynchronousTestCase`, имитирующий интерфейс `unittest.TestCase`. На самом деле интерфейс `SynchronousTestCase` является надмножеством `unittest.TestCase`, и в число его дополнительных возможностей входят функции проверки объектов `Deferred`.

Исследуем эти возможности, написав тесты для функции `fastestOfTwo`, объявленной в предыдущем разделе. Но сначала обобщим ее и перепишем так, чтобы вместо адресов URL она принимала два любых экземпляра `Deferred`:

```
def fastestOfTwo(d1, d2):
    def extractValue(valueAndIndex):
        value, index = valueAndIndex
        return value

    urlList = DeferredList([d1, d2],
                           fireOnOneCallback=True,
                           fireOnOneErrback=True)
    return urlList.addCallback(extractValue)
```

В первом тесте для этой обновленной версии `fastestOfTwo` мы проверим, что она возвращает экземпляр `Deferred`, который еще не разрешился, если не разрешился ни один из переданных ей экземпляров `Deferred`:

```
from twisted.internet import defer
from twisted.trial import unittest

class FastestOfTwoTests(unittest.SynchronousTestCase):
    def test_noResult(self):
        d1 = defer.Deferred()
        self.assertNoResult(d1)
        d2 = defer.Deferred()
        self.assertNoResult(d2)
        self.assertNoResult(fastestOfTwo(d1, d2))
```

Как можно догадаться по имени, `SynchronousTestCase.assertNoResult` проверяет отсутствие результата в переданном экземпляре `Deferred` и является ценным инструментом проверки наших ожиданий.

Однако объекты `Deferred` наиболее полезны, когда они имеют результат. В случае с `fastestOfTwo` мы ожидаем, что возвращаемый экземпляр `Deferred`

будет иметь значение одного из двух экземпляров Deferred, разрешившегося первым:

```
def test_resultIsFirstDeferredsResult(self):
    getResultFirst = defer.Deferred()
    neverGetsResult = defer.Deferred()
    fastestDeferred = fastestOfTwo(getResultFirst, neverGetsResult)
    self.assertNoResult(fastestDeferred)
    result = "the result"
    getResultFirst.callback(result)
    actualResult = self.successResultOf(fastestDeferred)
    self.assertIs(result, actualResult)
```

SynchronousTestCase.successResultOf либо возвращает результат первого разрешившегося экземпляра Deferred, либо заставляет тест потерпеть неудачу. Для проверки он создает результат со строкой "the result", передает его в вызов getResultFirst.callback и затем извлекает получившееся значение из fastestDeferred. После этого тест проверяет, действительно ли fastestOfTwo вернула результат, который первый стал доступным.

Обратите внимание, что здесь сначала проверяется отсутствие результата в Deferred, возвращаемом из fastestOfTwo, и только потом вызывается getResultFirst.callback. Эта проверка может показаться излишней, учитывая, что для этой проверки уже существует тест test_noResult, но не забывайте, что Deferred может разрешиться *до того*, как ваш код добавит обработчики. В этом случае fastestOfTwo может ошибочно вернуть Deferred, который уже содержит результат "the result", и если не учитывать этот аспект, тест будет считаться пройденным. В таком простом коде это невероятная ситуация, но неявные предположения о том, *когда* Deferred получает результат, могут вкрасься в код и привести к тому, что тесты не будут замечать ошибок. Всегда старайтесь проверять исходное состояние экземпляров Deferred, не полагаясь на неявные предположения, чтобы избежать подобных ошибок, а еще лучше – добавьте проверку наличия и отсутствия результата в Deferred.

Также можно добавить тест для случая, когда fastestOfTwo получает уже разрешившийся экземпляр Deferred:

```
def test_firedDeferredIsFirstResult(self):
    result = "the result"
    fastestDeferred = fastestOfTwo(defer.Deferred(),
                                   defer.succeed(result))
    actualResult = self.successResultOf(fastestDeferred)
    self.assertIs(result, actualResult)
```

Функция twisted.internet.defer.succeed принимает аргумент и возвращает Deferred, для которого уже был вызван метод callback с этим аргументом, поэтому здесь во втором аргументе fastestOfTwo получит экземпляр Deferred, разрешившийся строкой "the result" еще до фактического вызова fastestOfTwo.

Для полноты нужно также проверить, что случится, когда fastestOfTwo получит два экземпляра уже разрешившихся Deferred:

```
def test_bothDeferredsFired(self):
    first = "first"
    second = "second"
    fastestDeferred = fastestOfTwo(defer.succeed(first),
                                   defer.succeed(second))
    actualResult = self.successResultOf(fastestDeferred)
    self.assertIs(first, actualResult)
```

Класс `DeferredList` добавляет экземпляры `Deferred` в свой список в указанном порядке. С флагом `fireOnOneCallback=True` результат всего списка будет представлять разрешившийся экземпляр `Deferred`, находящийся ближе к началу списка. Поэтому в этом тесте ожидается, что `fastestDeferred` будет содержать результат `first`.

Обработка ошибок является важной частью любого тестирования, поэтому тесты для функции `fastestDeferred` также должны проверить, как она обрабатывает ошибки `Failure`. Мы рассмотрим только случай, когда `Deferred` потерпел неудачу еще до передачи в `fastestOfTwo`, чтобы сократить тест:

```
def test_failDeferred(self):
    class ExceptionType(Exception):
        pass
    fastestDeferred = fastestOfTwo(defer.fail(ExceptionType()),
                                   defer.Deferred())
    failure = self.failureResultOf(fastestDeferred)
    failure.trap(defer.FirstError)
    failure.value.subFailure.trap(ExceptionType)
```

По аналогии с `SynchronousTestCase.successResultOf`, метод `SynchronousTestCase.failureResultOf` возвращает текущую ошибку `Failure` из `Deferred`; если экземпляр `Deferred` еще не разрешился или разрешился с действительным результатом, `failureResultOf` вызовет завершение теста неудачей.

Поскольку возвращаемый объект является экземпляром `Failure`, внутри теста доступны все методы и атрибуты, которые можно использовать в обработчиках ошибок. `DeferredList` с флагом `fireOnOneErrback = True` обертывает ошибки исключением `twisted.internet.defer.FirstError`, поэтому в вызове `trap` мы проверяем исключение этого типа; если `Failure` завернуть в любое другое исключение, `trap` повторно возбудит его. Фактический экземпляр `Failure`, вызвавший исключение `FirstError`, доступен в его атрибуте `subFailure`, а поскольку мы передали экземпляр `ExceptionType`, в следующей инструкции, после перехвата исключения вызовом `trap`, мы проверяем, что сбой произошел именно по этой причине.

`assertNoResult` с `successResultOf` и `failResultOf` способствуют написанию тестов с явными предположениями о состоянии экземпляров `Deferred`. Как показывает `fastestOfTwo`, даже простое использование `Deferred` следует проверять на неявные зависимости от порядка следования и обработки ошибок. Все вышесказанное относится и к сопрограммам, а также любым другим примитивам параллельного выполнения. Пакет тестирования, входящий в состав Twisted,

естественно, имеет прекрасные инструменты для выявления типичных проблем параллельного выполнения в контексте Deferred.

Итоги

В этой главе мы продолжили изучение приемов событийно-ориентированного программирования, начатое в предыдущей главе, и узнали, что обработчики событий являются своего рода *обратными вызовами*. Благодаря широким возможностям стиля *передачи продолжений* с помощью обратных вызовов можно выражать весьма сложную логику. Обратные вызовы могут передавать значения другим обратным вызовам напрямую, а не через вызывающий код. Этот подход мы называли *внутренней композицией*, потому что она имеет место в теле каждого обратного вызова.

Внутренняя композиция усложняет поддержку программ, основанных на использовании обратных вызовов: каждый обратный вызов должен знать имя и сигнатуру обратного вызова, следующего за ним, чтобы суметь вызвать его. Переупорядочение последовательности обратных вызовов или удаление одного из них может потребовать изменения других. Решить эту проблему можно с помощью парадигмы *асинхронного программирования*, которая позволяет программам продолжить работу, не дожидаясь готовности результатов. *Объект-заполнитель*, представляющий асинхронный результат, может собирать коллекцию обратных вызовов (обработчиков) и запускать их, когда реальное значение станет доступно. Этот заполнитель получает результаты от обратных вызовов и тем самым обеспечивает возможность *внешней* композиции, что, в свою очередь, избавляет логические модули от необходимости знать, как и где они используются. Событийно-ориентированный код, использующий такие асинхронные объекты-заполнители, можно оформлять точно так же, как обычный линейный код.

Роль асинхронных объектов-заполнителей в фреймворке Twisted играют экземпляры Deferred. Мы видели, что они вызывают свои обработчики в цикле, передавая результат от одного к другому и вызывая обработчики ошибок при любом исключении. Этот цикл обработки внутри Deferred делает их мощной *абстракцией потока управления*.

Важной особенностью этой абстракции потока управления является возможность по-разному реагировать на разные ошибки. Класс Failure в фреймворке Twisted собирает информацию о трассировке вместе с исключением и предоставляет вспомогательные методы, с помощью которых можно фильтровать ошибки и повторно вызывать исключения. Мы видели, как обработчики результатов и ошибок могут представлять синхронный код, использующий try и except.

Экземпляры Deferred могут не только объединять обработчики, но и объединяться друг с другом. Когда обработчик результата или ошибки возвращает экземпляр Deferred, цикл обработки в соответствующем экземпляре Deferred

приостанавливается до разрешения этого нового экземпляра `Deferred`. Это означает, что функции и методы, возвращающие `Deferred`, можно использовать в качестве обработчиков без каких-либо особых усилий со стороны разработчика.

Несмотря на свою эффективность, объекты `Deferred` – не единственный способ композиции асинхронных операций. *Генераторы* в Python могут приостанавливать выполнение и возобновлять его после получения значений из внешних источников. Этот поток управления легко отображается в поток управления, предлагаемый объектами `Deferred` с его обратными вызовами, с помощью `inlineCallbacks`.

Однако генераторы не так однозначны, в том смысле, что они могут представлять простые итераторы или потоки управления, похожие на `Deferred`. Для преодоления этой неоднозначности в Python 3.5 была добавлена специальная поддержка *сопрограмм*, которые являются генераторами, способными приостанавливать себя и передавать управление другим сопрограммам без использования `inlineCallbacks`. Сопрограммы могут напрямую дожидаться разрешения экземпляров `Deferred` с помощью `await` и преобразовываться в экземпляры `Deferred` с помощью `sureDeferred`. Эти механизмы позволяют беспрепятственно использовать сопрограммы Twisted.

Не все программы можно выразить только с использованием сопрограмм: наша функция `fastestOfTwo`, например, ожидает появления одного из двух событий. К счастью, абстракция `DeferredList`, построенная поверх `Deferred`, позволяет мультиплексировать асинхронные результаты.

В фреймворке Twisted также есть специальная поддержка тестирования объектов `Deferred`. Класс `SynchronousTestCase` предлагает методы `assertNoResult`, `successResultOf` и `failResultOf`, позволяющие точно описывать ожидаемое состояние экземпляров `Deferred`. С помощью этого набора инструментов легко можно выявить любые проблемы параллельного выполнения, затрагивающие все примитивы – сопрограммы, генераторы и `Deferred`.

Глава 3

Создание приложений с библиотеками `treq` и `Klein`

В предыдущих главах мы детально исследовали основы Twisted. Знание этих основ необходимо, но недостаточно для создания действующих приложений. В этой главе мы рассмотрим способы организации программ и их высокоуровневых API, построив агрегатор каналов с использованием двух мощных веб-библиотек, входящих в состав фреймворка Twisted: `treq` и `Klein`.

Библиотека `treq` (<https://treq.readthedocs.io>) служит оберткой для `twisted.web.client.Agent` и реализует API в духе популярной синхронной HTTP-библиотеки `requests`. Удобные и разумные настройки по умолчанию упрощают отправку асинхронных HTTP-запросов, а фиктивные объекты, предлагаемые модулем `treq.testing`, упрощают тестирование.

Библиотека `Klein` (<https://klein.readthedocs.io>) служит отличной оберткой для веб-фреймворка `twisted.web.server`. Она позволяет писать динамические и асинхронные веб-приложения с использованием хорошо знакомой парадигмы маршрутизации в духе Werkzeug (<https://werkzeug.readthedocs.io/>).

Насколько важную роль играют эти библиотеки?

Фреймворк Twisted реализует базовую функциональность, которую применяют `Klein` и `treq`. Почему бы тогда не использовать эту функциональность напрямую? Интерфейсы обеих библиотек значительно отличаются от собственных интерфейсов Twisted; например, `twisted.web` применяет прием *обхода объектов* вместо *маршрутизации*, чтобы связать URL-пути с кодом на Python. Класс `twisted.web.server.Site` не сопоставляет пути и строки запросов со строковыми шаблонами, такими как `"/some/"`; вместо этого он сопоставляет сегменты пути с вложенными объектами `Resource`. Эта парадигма преобладала в веб-приложениях на Python, когда велась разработка `twisted.web`. Поэтому вместо добавления новой абстракции маршрутизации непосредственно в Twisted авторы `Klein` решили поэкспериментировать с другим подходом отдельно. Их усилия увенчались успехом, а независимость `Klein` позволила им развивать

и адаптировать свою библиотеку, не оказывая разрушительного влияния на приложения, использующие `twisted.web.server`.

Аналогично библиотека `treq` инкапсулирует типовые шаблоны использования `twisted.web.client.Agent` в высокоуровневый API; например, `Agent` требует определить все запросы, включая полезные нагрузки, в виде байтовых строк в объектах `IBodyProducer`, тогда как методы запросов в `treq` принимают байтовые строки напрямую. Использование `treq` не мешает применять `Agent`, все возможности которого остаются доступными в `Twisted`.

Инструмент `pip`, используемый для установки сторонних пакетов на Python, в наше время работает достаточно надежно, поэтому установка дополнительных зависимостей не выглядит обременительной. В следующей главе мы также увидим, как можно использовать `Docker` для повышения надежности разработки и развертывания приложений на основе `Twisted`, применяющих сторонние библиотеки. Наконец, обе библиотеки, `Klein` и `treq`, можно найти в репозитории `Twisted GitHub`, и разрабатываются и используются они ключевыми разработчиками `Twisted`. Эти библиотеки настолько надежны, насколько это вообще возможно.

АГРЕГИРОВАНИЕ КАНАЛОВ

Веб-агрегирование уходит корнями в другую, более открытую эпоху развития интернета. В расцвете сайты обслуживали файлы *каналов* по HTTP и структурировали их так, чтобы другие сайты могли использовать эти файлы для своих нужд. Открытые стандарты, такие как RSS (Really Simple Syndication – очень простое распространение; или Rich Document Format Site Summary – обогащенный формат документов со сводной информацией о сайте) и *Atom*, описывают эти структуры и позволяют любому желающему создавать свои программы, выступающие в роли потребителей этих каналов. Службы, *агрегирующие* каналы с нескольких сайтов в одном месте, стали популярным способом для пользователей оставаться в курсе последних событий. Расширения этих форматов, такие как RSS с вложениями, дали возможность добавлять в каналы ссылки на внешние источники, что привело к появлению таких методов распространения информации, как подкастинг.

Прекращение работы над `Google Reader` в 2013 году совпало с падением популярности каналов. Сайты удалили свои каналы, и некоторые клиентские программы утратили способность потреблять их. Несмотря на это, до сих пор не появилось единой замены для агрегирования веб-каналов, и этот подход остается эффективным способом организации контента из множества различных онлайн-источников.

Многие стандарты определяют свои варианты RSS. Там, где необходимо работать непосредственно с форматом канала, мы будем использовать только следующее подмножество RSS 2.0, определяемое центром Беркмана Гарвардского университета (<http://cyber.harvard.edu/rss/rss.html>):

- 1) корневым элементом файла канала RSS 2.0 служит тег `<channel>` и описывается его элементами `<title>` и `<link>`;
- 2) веб-страницы в `<channel>` описываются элементами `<item>`, каждый из которых имеет свои элементы `<title>` и `<link>`.

Работая над агрегатором каналов на основе Klein и treq, мы будем использовать прием разработки через тестирование. Но прежде чем заняться агрегатором каналов, познакомимся поближе с предметной областью и напишем несколько пробных программ. Затем используем полученные знания для проектирования, реализации и итеративного улучшения нашего приложения. Поскольку невозможно отобразить содержимое каналов, не загрузив их, начнем с изучения способов отправки HTTP-запросов с помощью treq.

ВВЕДЕНИЕ в treq

Перед отображением каналов агрегатор должен сначала загрузить их, поэтому начнем с исследования возможностей библиотеки treq. Обратите внимание, что примеры, представленные далее, должны работать под управлением обеих версий Python – 2 и 3.

Создайте новое виртуальное окружение с помощью своего излюбленного инструмента и установите в него библиотеку treq из PyPI. Сделать это можно с помощью нескольких инструментов, но для единообразия мы советуем использовать virtualenv (<https://virtualenv.pypa.io/en/stable/>) и pip (<https://pip.pypa.io/en/stable/>):

```
$ virtualenv treq-experiment-env
...
$ ./treq-experiment-env/bin/pip install treq
...
$ ./treq-experiment-env/bin/python experiment.py
```

где `experiment.py` содержит следующий код:

```
from argparse import ArgumentParser
from twisted.internet import defer, task
from twisted.web import http
import treq

@defer.inlineCallbacks
def download(reactor):
    parser = ArgumentParser()
    parser.add_argument("url")
    arguments = parser.parse_args()
    response = yield treq.get(
        arguments.url, timeout=30.0, reactor=reactor)
    if response.code != http.OK:
        reason = http.RESPONSES[response.code]
        raise RuntimeError("Failed:{}".format(response.code, reason))
    content = yield response.content()
    print(content)

task.react(download)
```

Функция `download` извлекает URL из аргументов командной строки, используя модуль `argparse` из стандартной библиотеки, и затем посылает GET-запрос с помощью `treq.get`. Клиентский API библиотеки `treq` принимает URL в виде строк `bytes` или `unicode`, кодируя последние в соответствии со сложными правилами для текстовых URL. Это упрощает разработку программ, потому что `ArgumentParser.parse_args` возвращает объекты `str`, представляющие аргументы командной строки в Python 2 и 3; в Python 2 это строки `bytes`, а в Python 3 – строки `unicode`. Мы избавлены от необходимости беспокоиться о кодировании или декодировании строк `str` с URL-адресами в соответствии с типом, соответствующим конкретной версии Python, потому что `treq` сделает это автоматически.

Клиентский API библиотеки `treq` принимает параметр `timeout` с интервалом времени, по истечении которого завершает запросы, которые не успели *начаться*. Аргумент `reactor` определяет объект реактора для выполнения сетевых взаимодействий и внутренних операций. Это форма *внедрения зависимостей*: `treq` *зависит* от реактора, но вместо импортирования `twisted.internet.reactor` библиотека `treq` дает возможность программисту самому предоставить эту зависимость. Далее вы увидите, как внедрение зависимостей упрощает тестирование и анализ кода.

Функция `treq.get` возвращает `Deferred`, который разрешается в объект `treq.response._Response` (символ подчеркивания в начале имени говорит не о том, что мы не должны взаимодействовать с экземплярами этого типа, а лишь о том, что мы не должны сами конструировать их). Он реализует интерфейс `twisted.web.iweb.IRequest`, то есть содержит код ответа в своем атрибуте `code`. Наша программа проверяет его, чтобы убедиться, что запрос был благополучно получен и обработан сервером; в противном случае она генерирует исключение `RuntimeError` с кодом ответа и соответствующим описанием, извлекаемым из словаря `twisted.web.http.RESPONSES`.

Экземпляр `Deferred` может также разрешиться в объект `Failure`. Если, например, интервал времени, указанный в параметре `timeout`, истек до того, как был сконструирован экземпляр `Response`, объект `Deferred` разрешится в ошибку `CancelledError`.

Библиотека `treq` также имеет удобные вспомогательные методы. Один из таких методов – `content` – возвращает `Deferred`, разрешающийся строкой `bytes` с полным телом ответа. Все действия, связанные с получением и сборкой ответа, `treq` выполняет автоматически.

Наконец, наш пример нигде не вызывает `reactor.run` или `reactor.stop` непосредственно. Вместо этого используется функция из библиотеки `Twisted`, с которой мы пока не встречались: `twisted.internet.task.react`. Функция `react` автоматически запускает и останавливает реактор. Она принимает единственный обязательный аргумент – вызываемый объект, которому передает запущенный реактор; сам вызываемый объект должен вернуть `Deferred`, заставляющий реактор остановиться, когда разрешится действительным значением или ошибкой

Failure. Функция `download` возвращает именно такой экземпляр `Deferred` благодаря использованию декоратора `twisted.internet.defer.inlineCallbacks`. Так как сама функция `react` принимает вызываемый объект в первом аргументе, ее тоже можно использовать в роли декоратора. Мы могли бы переписать наш пример, как показано ниже:

```
..
from twisted.internet import defer, task
...
@task.react
@defer.inlineCallbacks
def main(reactor):
...
```

Фактически это один из самых популярных способов реализации коротких сценариев, использующих Twisted. В дальнейшем мы будем применять `react` именно как декоратор.

Если запустить эту программу и передать ей URL веб-канала, она извлечет его содержимое. Мы можем добавить в программу код, использующий библиотеку `feedparser` для Python, чтобы вывести сводную информацию о канале. Для этого сначала нужно установить `feedparser` в виртуальное окружение с помощью `pip`:

```
$ ./treq-experiment-env/bin/pip install feedparser
```

Затем сохранить программу, следующую ниже, в файле `feedparser_experiment.py` и запустить ее, передав RSS URL:

```
$ ./treq-experiment-env/bin/python feedparser_experiment.py
http://planet.twistedmatrix.com

from __future__ import print_function
from argparse import ArgumentParser
import feedparser
from twisted.internet import defer, task
from twisted.web import http
import treq

@task.react
@defer.inlineCallbacks
def download(reactor):
    parser = ArgumentParser()
    parser.add_argument("url")
    arguments = parser.parse_args()
    response = yield treq.get(arguments.url, reactor=reactor)
    if response.code != http.OK:
        reason = http.RESPONSES[response.code]
        raise RuntimeError("Failed:{}".format(response.code,
                                                reason))

    content = yield response.content()
    parsed = feedparser.parse(content)
```

```
print(parsed['feed']['title'])
print(parsed['feed']['description'])
print("*** ENTRIES ***")
for entry in parsed['entries']:
    print(entry['title'])
```

Эта программа должна вывести примерно следующее:

```
Planet Twisted
Planet Twisted - http://planet.twistedmatrix.com/
*** ENTRIES ***
Moshe Zadka: Exploration Driven Development
Hynek Schlawack: Python Application Deployment with Native Packages
Hynek Schlawack: Python Hashes and Equality
...
```

ВВЕДЕНИЕ В KLEIN

Теперь, научившись извлекать содержимое каналов с помощью `treq`, посмотрим, как отображать его в веб-страницах с помощью `Klein`.

Не будем нарушать принятого порядка и создадим для экспериментов новое виртуальное окружение, куда установим библиотеку `Klein` командой `pip install klein`. Затем запустим следующий пример:

```
import klein

application = klein.Klein()

@application.route('/')
def hello(request):
    return b'Hello!'

application.run("localhost", 8080)
```

Теперь откройте в браузере страницу `http://localhost:8080/`. (Возможно, вам понадобится изменить номер порта `8080`, если он уже занят какой-то другой программой.) Вы увидите строку `Hello!`, созданную обработчиком `hello` маршрута `'/'`.

Приложение начинается с создания экземпляра класса `Klein`. Вызываемые объекты ассоциируются с маршрутами с использованием метода `Klein.route` в роли декоратора. В первом аргументе методу `route` передается шаблон URL в стиле `Werkzeug`; допустимые директивы перечислены в документации к библиотеке `Werkzeug`, доступной по адресу: `http://werkzeug.readthedocs.io/en/latest/routing/`. Давайте немного изменим программу и добавим в нее одну из таких директив, извлекающую целое число из пути в URL:

```
import klein

application = klein.Klein()

@application.route('/<int:amount>')
def increment(request, amount):
```

```

newAmount = amount + 1
message = 'Hello! Your new amount is:{ } '.format(newAmount)
return message.encode('ascii')

application.run("localhost",8080)

```

Если запустить эту программу и открыть в браузере адрес `http://localhost:8080/1`, вы увидите страницу, изображенную на рис. 3.1.

Шаблон URL определяет компонент пути, который библиотека Klein извлекает, преобразует в указанный тип и передает функции-обработчику в позиционном аргументе. Аргумент `amount` – это первый элемент пути и должен быть целым числом; в противном случае запрос потерпит неудачу с кодом 404. Список *преобразователей типов* можно найти в документации к библиотеке Werkzeug.

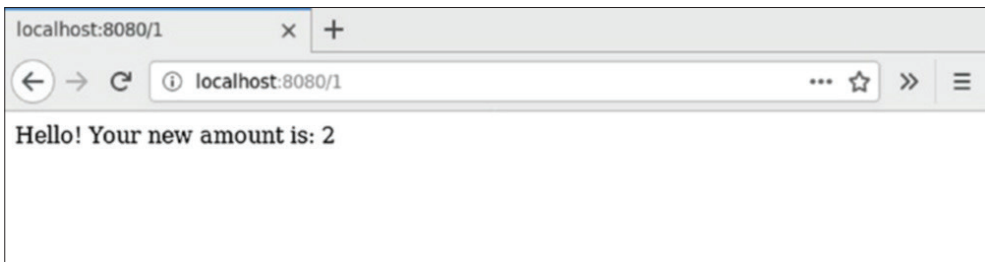


Рис. 3.1 ❖ Результат работы обработчика `increment`

Отметьте также, что обработчики не могут возвращать строки `unicode`; для программ, выполняющихся под управлением Python 3, это означает, что строки `str` должны преобразовываться в строки `bytes` перед возвратом из обработчика маршрута Klein. По этой причине в программе выше мы кодируем переменную `message` как `ascii` после ее формирования. В версии Python 3.5 и выше можно использовать форматирование строк байтов, но на момент написания этой книги все еще широко использовалась версия Python 3.4. Кроме того, в Python 2 этот код неявно *декодирует* `message` как `ascii`. Такое неудачное поведение приводит к появлению туманных сообщений об ошибках при использовании других кодировок, отличных от `ascii`, но является распространенным шаблоном в Twisted-коде для работы со строками `str`, которые содержат только ASCII, и этот подход должен работать как в версии Python 2, так и в версии Python 3.

Klein и Deferred

Библиотека Klein является составной частью проекта Twisted, поэтому изначально имеет поддержку объектов `Deferred`. Функции-обработчики возвращают экземпляры `Deferred`, которые в какой-то момент разрешатся в действительное значение или `Failure`. Чтобы увидеть, как это происходит, изменим программу

и симитируем медленную сетевую операцию, вернув экземпляр `Deferred`, который должен разрешиться через секунду после получения запроса:

```
from twisted.internet import task
from twisted.internet import reactor
import klein

application = klein.Klein()

@application.route('/<int:amount>')
def slowIncrement(request, amount):
    newAmount = amount + 1
    message = 'Hello! Your new amount is:{} '.format(newAmount)
    return task.deferLater(reactor, 1.0, str.encode, message, 'ascii')

application.run("localhost", 8080)
```

Как и ожидалось, эта программа возвращает страницу `http://localhost:8080/1` только по истечении секунды. Такая задержка достигается использованием метода `twisted.internet.task.deferLater`, который принимает провайдера `twisted.internet.interfaces.IReactorTime`, величину задержки в секундах и функцию с аргументами для нее, которая будет вызвана по истечении указанного времени. Обратите внимание, что в данном случае, выбирая функцию и аргументы для нее, мы руководствовались тем фактом, что методы экземпляров хранятся в соответствующих классах и в первом аргументе получают ссылку на соответствующий экземпляр; то есть вызов `str.encode(message, 'ascii')`, где `message` является экземпляром `str`, эквивалентен вызову `message.encode('ascii')`. Это еще один шаблон, который часто можно встретить в `Twisted`-коде.

Этот последний пример демонстрирует ограничение, присущее использованию декораторов для регистрации маршрутов: аргументы для декорированной функции должны полностью предоставляться инфраструктурой маршрутизации. Это затрудняет определение функций-обработчиков, ссылающихся на некоторое состояние или зависящих от некоторого существующего объекта. Код в нашем примере зависит от реактора, чтобы удовлетворить требования `deferLater`, но мы не можем передать реактор обработчику, потому что только `Klein` может вызвать его. Из многих способов решения этой проблемы `Klein` поддерживает только один: приложения `Klein` для конкретного экземпляра. Перепишем наш пример `slowIncrement` и используем эту возможность.

```
from twisted.internet import task
from twisted.internet import reactor
import klein

class SlowIncrementWebService(object):
    application = klein.Klein()

    def init(self, reactor):
        self._reactor = reactor

    @application.route('/<int:amount>')
    def slowIncrement(self, request, amount):
```

```

newAmount = amount + 1
message = 'Hello! Your new amount is:{}'.format(newAmount)
return task.deferLater(self._reactor, 1.0, str.encode, message,
                       'ascii')

```

```

webService = SlowIncrementWebService(reactor) webService.application.
run("localhost", 8080)

```

Класс `SlowIncrementWebService` инициализирует приложение `Klein` и сохраняет ссылку на него в переменной уровня класса `application`. Методы класса можно декорировать методом `route` этой переменной, в точности как в примере функции `slowIncrement` уровня модуля, которая декорировалась методом `route` объекта `Klein`. Но поскольку теперь декорируются методы экземпляра, мы можем обращаться к переменным экземпляра, таким как `reactor`. Это позволяет *параметризовать* веб-приложения, не опираясь на объекты уровня модуля.

Сами объекты `Klein` локализуют свое внутреннее состояние, реализуя протокол дескриптора. `webService.application` возвращает экземпляр `Klein` для конкретного запроса, который содержит все маршруты и их обработчики, зарегистрированные в приложении `SlowIncrementWebService`. Таким образом, библиотека `Klein` поддерживает надежную инкапсуляцию и минимизирует объем общего изменяемого состояния.

Механизм шаблонов `Plating` в `Klein`

И последнее, что нам нужно рассмотреть, прежде чем мы будем готовы написать простую версию агрегатора каналов, – это механизм шаблонов для создания веб-страниц. Конечно, можно было бы использовать `Jinja2`, `Мако` или любой другой механизм шаблонов для создания веб-страниц в `Python`, но в состав библиотеки `Klein` входит свой механизм шаблонов, который называется *Plating*. Давайте изменим пример `SlowIncrementWebService` и используем в нем `klein.Plating` для создания удобочитаемого ответа:

```

from twisted.internet import task, reactor
from twisted.web.template import tags, slot
from klein import Klein, Plating

class SlowIncrementWebService(object):
    application = Klein()
    commonPage = Plating(
        tags=tags.html( tags.head(
            tags.title(slot("title")),
            tags.style("#amount { font-weight: bold; }"
                      "#message { font-style: italic; }")),
        tags.body(
            tags.div(slot(Plating.CONTENT))))

    def __init__(self, reactor):
        self._reactor = reactor

    @commonPage.routed(
        application.route('/<int:amount>'),

```

```

tags.div(
    tags.span("Hello! Your new amount is: ", id="message"),
    tags.span(slot("newAmount"), id="amount")),
)

def slowIncrement(self, request, amount):
    slots = {
        "title": "Slow Increment",
        "newAmount": amount + 1,
    }
    return task.deferLater(self._reactor, 1.0, lambda: slots)

```

```

webService=SlowIncrementWebService(reactor)
webService.application.run("localhost",8080)

```

Новый объект `commonPage` типа `Plating` – это главное изменение в `SlowIncrementWebService`. Так как `Plating` основывается на механизме шаблонов `twisted.web.template` в `Twisted`, мы должны познакомиться с ним, прежде чем продолжить.

`twisted.web.template` конструируется из экземпляров `twisted.web.template.Tag` и `twisted.web.template.slot`. Экземпляры `Tag` представляют теги HTML, такие как `html`, `body` и `div`. Они создаются путем вызова методов с соответствующими именами экземпляра фабрики `twisted.web.template.tags`, например:

```
tags.div()
```

представляет тег `div`, который отображается так:

```
<div></div>
```

Позиционные аргументы в этих методах представляют вложенные теги, то есть добавить тег `span` внутрь тега `div` можно так:

```
tags.div(tags.span("A span."))
```

Это простое дерево тегов будет отображаться так:

```
<div><span>A span.</span></div>
```

Обратите внимание, что текстовое содержимое тега тоже представлено как дочерний элемент.

Именованные аргументы этих методов определяют атрибуты тегов, учитывая это, вот как можно включить изображение в наше дерево `div`:

```
tags.div(tags.img(src="picture.png"), tags.span("A span."))
```

При отображении это дерево будет выглядеть так:

```
<div><span>A span.</span></div>
```

`twisted.web.template` резервирует один именованный аргумент для внутренних нужд: `render`. Это строка, определяющая *метод отображения*, который должен использоваться для преобразования тега в разметку HTML. Мы еще вернемся к методам отображения, имеющимся в библиотеке `Klein`.

Иногда удобнее записывать атрибуты тега перед дочерними элементами, но именованные аргументы всегда должны предшествовать позиционным аргументам. Чтобы воспользоваться этим улучшением, не нарушая синтаксиса языка Python, tags можно вызывать с дочерними элементами. Например, вот как можно переписать предыдущее дерево div, добавляя дочерние элементы подобным способом:

```
tags.div()(tags.img(src="picture.png"), tags.span("A span."))
```

Слоты служат заполнителями, которые, как будет показано ниже, могут заполняться по их именам во время отображения шаблона. Они позволяют параметризовать содержимое тегов и атрибутов. С учетом этого дерево div можно переписать так:

```
tags.div(tags.img(src=slot('imageUrl')), tags.span(slot("spanText")))
```

Мы можем передать имя файла "anotherImage.png" в слот imageUrl и текст "Different text." в слот spanText и получить в результате:

```
<div><span>Different text.</span></div>
```

Когда слоты заполняются строками с литералами HTML, twisted.web.template экранирует их, чтобы избежать неправильной интерпретации пользовательских данных как директив шаблонов. Это, в свою очередь, устраняет многие типичные ошибки, открывающие уязвимость к атакам вида «межсайтовый скриптинг» (cross-site scripting, XSS). Однако слоты можно заполнять другими вызовами tags, что открывает путь к повторному использованию шаблонов. Это означает, что такая цепочка вызовов:

```
tags.div(slot("child")).fillSlots(child="<div>")
```

отобразится в:

```
<div>&lt;div&gt;</div></div>
```

А эта цепочка:

```
tags.div(slot("child")).fillSlots(child=tags.div())
```

отобразится в:

```
<div><div></div></div>
```

ПЕРВАЯ ВЕРСИЯ АГРЕГАТОРА КАНАЛОВ

Теперь, познакоившись с основами twisted.web.template, вернемся к примеру объекта klein.Plating:

```
commonPage = Plating(
    tags=tags.html(
        tags.head(
            tags.title(slot("title")),
            tags.style("#amount { font-weight: bold; }")
```

```

        "#message { font-style: italic; }")),
    tags.body(
        tags.div(slot(Plating.CONTENT))))))

```

Дерево тегов, которое передается в аргументе `tags`, описывает структуру всех страниц HTML, которые будет отображать этот экземпляр `Plating`. В нем имеются два слота: `title` и `Plating.CONTENT`. Слот `title` ничем не отличается от любых других; в нем мы будем передавать нужное нам значение при отображении страницы, являющейся частью этого дерева тегов. Но слот `Plating.CONTENT` представляет местоположение в дереве тегов, куда `Plating` будет вставлять содержимое конкретной страницы. В нашем примере приложение отображает только одну страницу, производную от `commonPage`:

```

@commonPage.routed(
    application.route('/<int:amount>'),
    tags.div(
        tags.span("Hello! Your new amount is: ", id="message"),
        tags.span(slot("newAmount"), id="amount")),
    )
def slowIncrement(self, request, amount):
    slots={
        "title": "Slow Increment",
        "newAmount": amount+1,
    }
    return task.deferLater(self._reactor, 1.0, lambda: slots)

```

Мы определяем производную страницу, обертывая маршрут `route` декоратором `routed` базовой страницы. Вторым позиционным аргументом декоратора `routed` представляет дерево тегов для заполнения слота `Klein.CONTENT` в базовой странице. Эта страница `slowIncrement` обертывает тот же маршрут, который мы определили выше, и в качестве своего содержимого передает дерево тегов, включающее слот с увеличенной суммой.

В `Klein` слоты заполняются возвратом из обработчика страницы словаря, который отображает имена слотов в значения, или `Deferred`, который разрешается в единицу. Этот обработчик все так же действует медленно, используя `deferLater` для отсрочки возврата словаря со слотами на одну секунду.

В результате получается веб-страница, обладающая индивидуальными чертами, как показано на рис. 3.2.

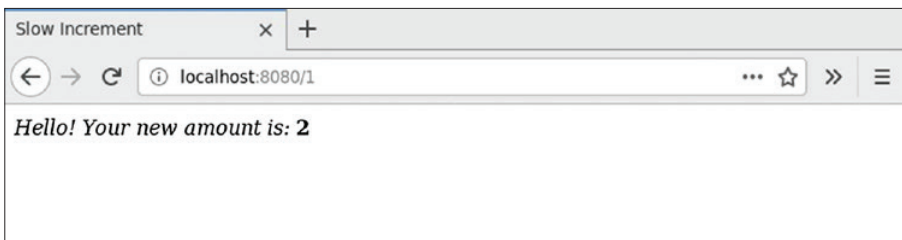


Рис. 3.2 ❖ Стилизация страницы `Slow Increment`

Механизм шаблонов в Klein предлагает уникальную возможность: запросить возврат словаря со слотами в виде JSON, указав параметр запроса `json`. На рис. 3.3 показано, как будет выглядеть наша страница «Slow Increment», если передать этот параметр.

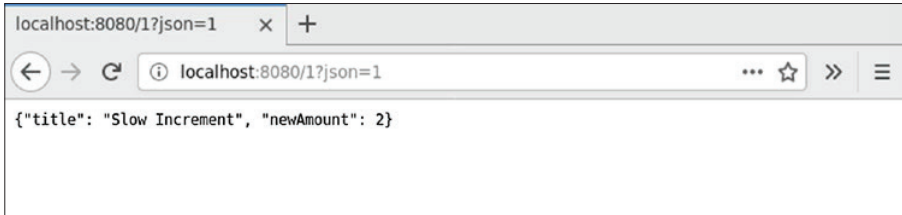


Рис. 3.3 ❖ Содержимое страницы в формате JSON

Это позволяет пользователям Plating писать обработчики, отображающие страницы в двух форматах, HTML и JSON, которые могут генерировать простые автономные страницы или служить вычислительными компонентами для сложных одностраничных мобильных или веб-приложений (Single Page Applications, SPA). Мы не будем превращать HTML-интерфейс нашего агрегатора каналов в SPA, потому что эта книга рассказывает о Twisted, а не о JavaScript, но мы продолжим исследовать сериализацию в формат JSON.

Теперь напишем упрощенную версию агрегатора каналов и попутно исследуем его архитектуру. Определим класс `SimpleFeedAggregation`, принимающий URL-адреса каналов и использующий `treq` для их извлечения, когда пользователь откроет главную страницу. Мы будем отображать каждый канал в виде таблицы, заголовки которой связаны с каналом, а строки – с элементами канала.

Для начала установите библиотеки `feedparser` и `treq` в виртуальное окружение Klein, как мы делали это выше в виртуальном окружении `treq`.

```
import feedparser

from twisted.internet import defer, reactor
from twisted.web.template import tags, slot
from twisted.web import http
from klein import Klein, Plating
import treq

class SimpleFeedAggregation(object):
    application = Klein()
    commonPage = Plating(
        tags=tags.html(
            tags.head(
                tags.title("Feed Aggregator 1.0")),
            tags.body(
                tags.div(slot(Plating.CONTENT))))))

    def __init__(self, reactor, feedURLs):
        self._reactor = reactor
```

```

        self._feedURLs = feedURLs

@defer.inlineCallbacks
def retrieveFeed(self, url):
    response = yield treq.get(url, timeout=30.0, reactor=self._reactor)
    if response.code != http.OK:
        reason = http.RESPONSES[response.code]
        raise RuntimeError("Failed:{}".format(response.code,
                                                reason))

    content = yield response.content()
    defer.returnValue(feedparser.parse(content))

@commonPage.routed(
    application.route('/'),
    tags.div(render="feeds:list")(slot("item")))
def feeds(self, request):

    def renderFeed(feed):
        feedTitle = feed[u"feed"][u"title"]
        feedLink = feed[u"feed"][u"link"]
        return tags.table(
            tags.tr(tags.th(tags.a(feedTitle, href=feedLink)))
        )([
            tags.tr(tags.td(tags.a(entry[u'title'], href=entry[u'link'])))
            for entry in feed[u'entries']
        ])

    return {
        u"feeds": [
            self.retrieveFeed(url).addCallback(renderFeed)
            for url in self._feedURLs
        ]
    }

webService = SimpleFeedAggregation(reactor,
    [ "http://feeds.bbc.co.uk/news/technology/rss.xml",
      "http://planet.twistedmatrix.com/rss20.xml" ])
webService.application.run("localhost",8080)

```

Метод `retrieveFeed` напоминает функцию `download` из нашей первой программы, использовавшей библиотеку `treq`, а метод `feeds` снабжен декоратором `Plating`, подобно приложению `slowIncrement` на основе библиотеки `Klein`. Но в данном случае шаблон маршрута содержит тег `div` со специализированным методом отображения. `Klein` интерпретирует `feeds:list` как требование продублировать тег `div` для каждого элемента в списке и поместить его в слот. Если, например, наш метод `feeds` вернет словарь:

```
{"feeds": ["first", "second", "third"]}
```

`Klein` отобразит его в следующую разметку HTML:

```
<div>first</div><div>second</div>third</div>
```

Наш метод `feeds` возвращает словарь, ключ `feeds` которого содержит не просто список, но экземпляры `Deferred`. В этом случае используется уникальная

возможность `twisted.web.template` отображать результаты, хранящиеся в экземплярах `Deferred`: встречая такой экземпляр, механизм шаблонов приостанавливается, пока тот не разрешится в действительное значение, которое затем отображается, или не завершится с ошибкой.

Каждый экземпляр `Deferred` в списке `feeds` возвращается вызовом `retrieveURL`, который создает канал для URL с использованием `treq` и `feedparser`. Обработчик `genderFeed` преобразует канал в дерево тегов, которое, в свою очередь, добавляется в таблицу ссылок. Для этого используется возможность `twisted.web.template` внедрять элементы `tags` в слоты.

Если теперь открыть страницу в браузере, на ней первым появится канал BBC, затем более объемный и долго извлекаемый канал Twisted Matrix, как показано на рис. 3.4 и 3.5.

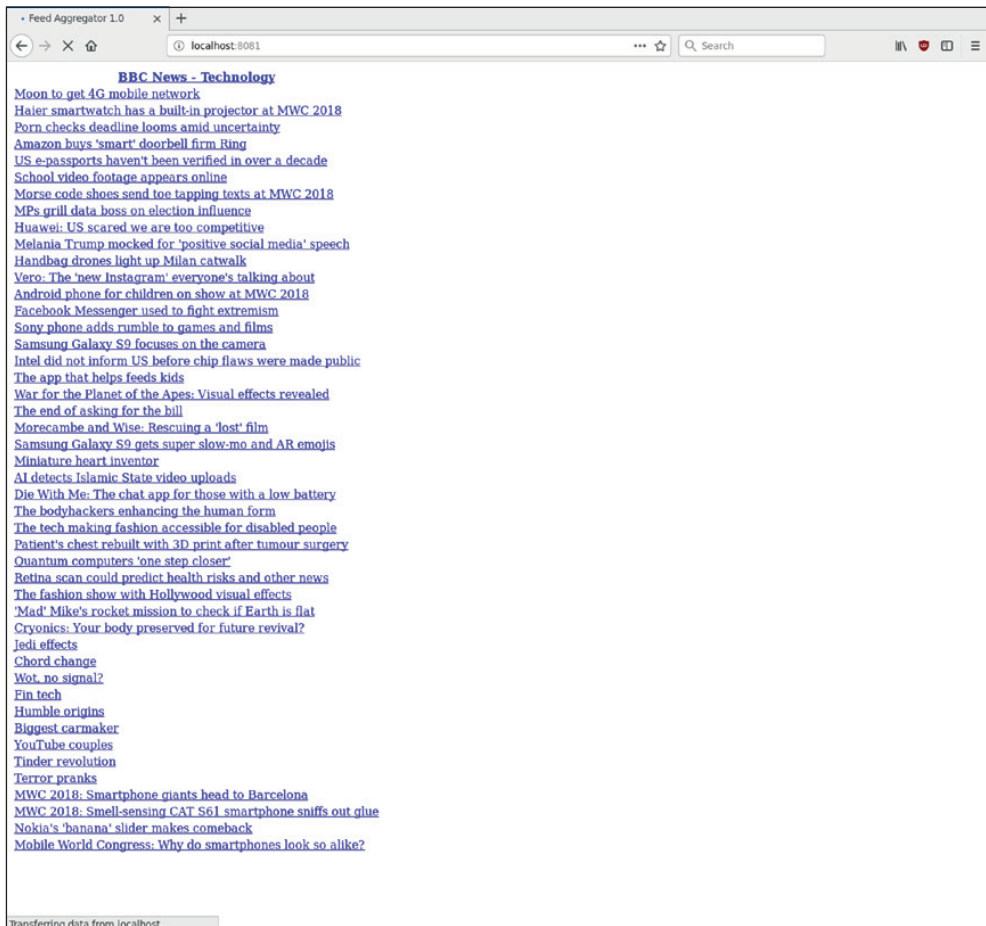


Рис. 3.4 ❖ Неполная страница, содержащая только канал BBC

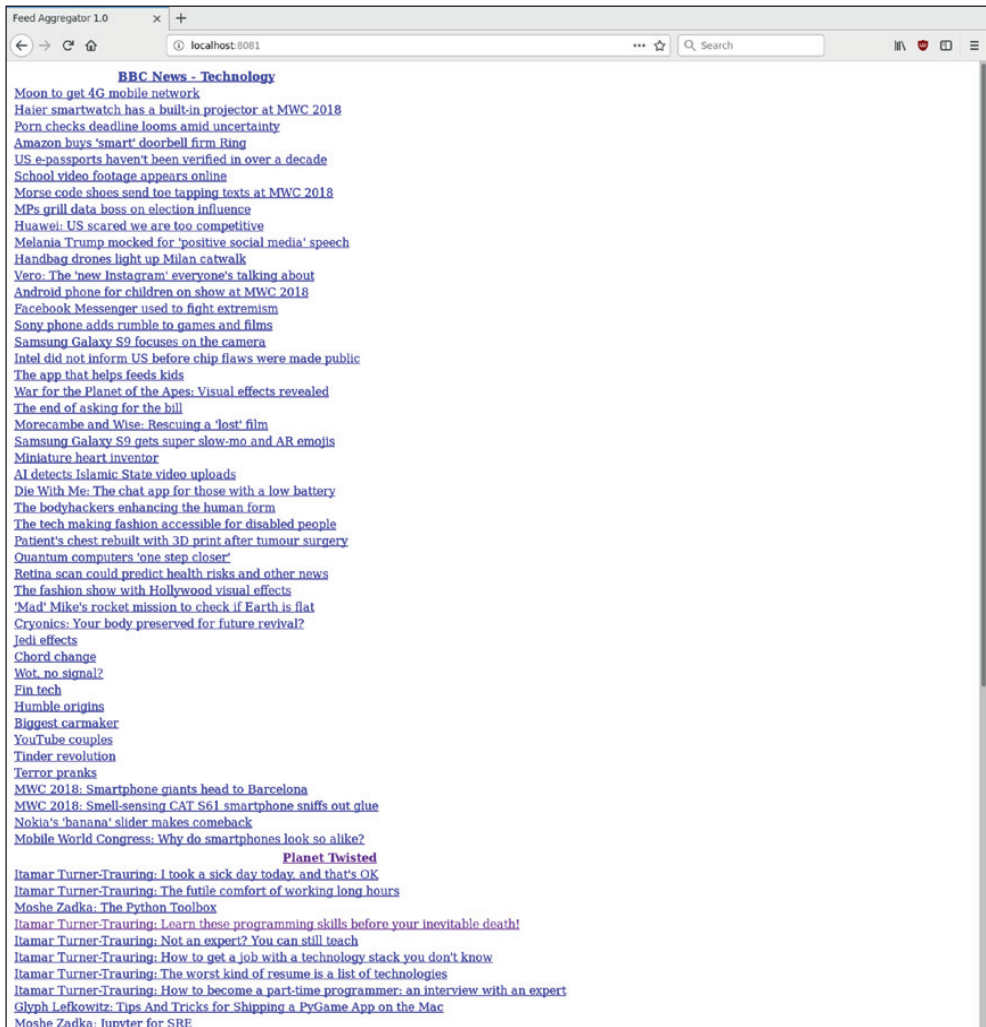


Рис. 3.5 ❖ Полная страница с обоими каналами, BBC и Twisted Matrix

Наш класс `SimpleFeedAggregation` благополучно извлекает и отображает каналы. Его базовая архитектура отражает поток данных в службе: выполняется обход адресов URL каналов, запускается параллельное их извлечение вызовом `trek.get`. Поток данных нередко определяет дизайн программ на основе `Twisted`.

Однако в нашей реализации:

- 1) обработка ошибок плохо организована. Несмотря на информативность исключения `RuntimeError`, возбуждаемого `SimpleFeedAggregation.retrieveFeed`, сведения передаются пользователям в неудобочитаемом виде, особенно запросившим результат в формате `JSON`;

- 2) имеется ошибка. На самом деле пользователи не смогут запросить данные в формате JSON, потому что деревья тегов, представляющие каналы, не сериализуются в JSON.

Но прежде чем исправлять эти и другие проблемы, напомним набор тестов, чтобы дальнейшая реализация агрегатора каналов велась с использованием методики разработки через тестирование.

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ KLEIN и treq

Создание тестов требует времени и сил. Методика разработки через тестирование облегчает эту задачу, превращая тестирование в часть процесса разработки. Начнем с *интерфейса*, который должен реализовать некоторый блок кода. Затем напишем пустую реализацию – класс с пустыми методами, а потом тесты, проверяющие данные, возвращаемые этой реализацией, с учетом известных входных данных. Вначале любая попытка запустить тесты должна завершаться неудачей, а разработка сводится к заполнению реализации, обеспечивающей прохождение тестов. В результате на ранней стадии мы выясним, конфликтуют ли части реализации друг с другом, и в конце получим полный набор тестов.

Создание тестов требует времени, поэтому важно начинать с самого ценного интерфейса. Для веб-приложения таким интерфейсом является HTTP-интерфейс, который будут использовать клиенты, поэтому наши первые тесты будут действовать подобно HTTP-клиентам, использующим приложение Feed-Aggregation.

Выполнение тестов на устанавливаемом проекте

Разработка через тестирование требует снова и снова запускать тесты, поэтому, прежде чем приступить к их созданию, мы должны выполнить настройки, с помощью которых `trial`, механизм автоматического запуска тестов в Twisted, смог их найти.

Команда `trial` принимает единственный обязательный аргумент, представляющий *полный путь* к чему-либо, что содержит или представляет выполняемые тесты. В дизайне `trial` использован обычный шаблон в стиле xUnit, как в модуле `unittest` для Python, поэтому тесты оформляются как подклассы `twisted.trial.unittest.TestCase` или `twisted.trial.unittest.SynchronousTestCase`. Эти имена сами являются полными квалифицированными именами; начиная с пакета самого верхнего уровня, они определяют полный путь через атрибуты, вплоть до конкретной функции, класса или метода. Например, следующая команда запускает метод `test_sillyEmptyThing` в тесте `ParsingTests`, который размещается в наборе тестов Twisted для проверки протокола обмена асинхронными сообщениями (Asynchronous Message Protocol, AMP):

```
trial twisted.test.test_amp.ParsingTests.test_sillyEmptyThing
```

Получив такое полное квалифицированное имя, `trial` последовательно переходит от модуля к модулю в дереве пакета и отыскивает тест, подобно тому как это делает `python -m unittest discover`. Например, все тесты, имеющиеся в фреймворке `Twisted`, можно запустить командой `trial twisted`.

Поскольку тесты определяются полным квалифицированным именем, они должны быть доступными для импорта. `trial` делает еще шаг вперед и требует, чтобы они находились в одном из *путей поиска модулей* Python времени выполнения. Это согласуется с соглашением в `Twisted` о включении тестов в библиотеку в специальные пакеты `test`.

Python позволяет программистам определять пути поиска несколькими способами. Устанавливая переменную окружения `PYTHONPATH` или непосредственно манипулируя значением `sys.path`, можно импортировать код из определенных для проекта местоположений. Однако сообщать интерпретатору Python пути, в которых он сможет найти код, – не лучшее решение, потому что возникает зависимость от конкретной конфигурации и точек входа. Для локализации путей поиска Python в дереве каталогов конкретного проекта лучше создавать виртуальные окружения, а затем устанавливать в них проекты с их зависимостями. Возможность управлять своими приложениями точно так же, как мы управляем его зависимостями, дает нам более высокую согласованность за счет использования тех же инструментов и шаблонов.

Полное обсуждение виртуальных окружений в Python выходит за рамки этой книги. Тем не менее мы определим минимальный макет и конфигурацию проекта, покажем, как связать проект с виртуальным окружением, а потом предоставим пример вызова `trial` для пустого набора тестов.

Каталог проекта имеет следующую структуру:

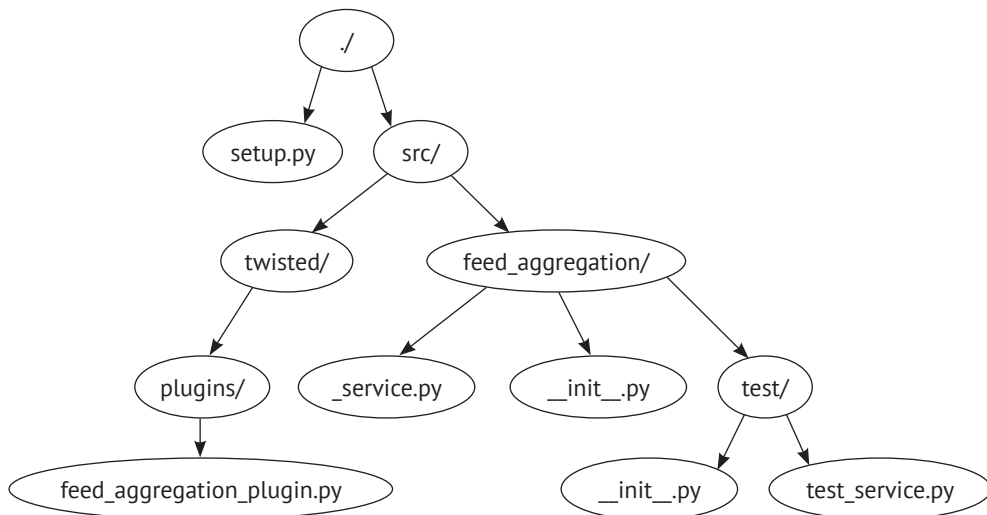


Рис. 3.6 ❖ Структура каталогов для проекта `Feed Aggregation` агрегатора каналов

То есть в некотором каталоге, принятом в качестве текущего рабочего, существуют файл `setup.py` и каталог `src/`. Каталог `src/`, в свою очередь, содержит пакет верхнего уровня `feed_aggregation` и подмодуль `_service`. `feed_aggregation.test.test_service` будет содержать тесты для кода в `_service`.

`src/twisted/plugins/feed_aggregation_plugin.py` будет содержать плагин на основе Twisted, упрощающий запуск приложения на основе Klein.

Поместим наш класс `FeedAggregation` в `feed_aggregation._service`:

```
class FeedAggregation(object):
    pass
```

Это приватный модуль, поэтому сделаем наш класс общедоступным, экспортировав его в `feed_aggregation/__init__.py`:

```
from feed_aggregation._service import FeedAggregation
__all__ = ["FeedAggregation"]
```

Размещение реализации в приватном подмодуле и последующее его экспортирование в пакете верхнего уровня `__init__.py` – это распространенный прием в Twisted. Таким способом гарантируется, что инструменты документирования, линтеры (инструменты оценки качества кода) и интегрированные среды разработки будут видеть общедоступные API как общедоступные пакеты, без доступа к приватным деталям реализации.

Оставим `feedaggregation/test/__init__.py` пустым, но поместим тривиальный подкласс `SynchronousTestCase` в `feed_aggregation/test/test_service.py`, чтобы команда `trial` могла что-то запустить, когда мы завершим настройку:

```
from twisted.trial.unittest import SynchronousTestCase
```

```
class FeedAggregationTests(SynchronousTestCase):
    def test_nothing(self):
        pass
```

`twisted/plugins/feed_aggregation_plugin.py` тоже оставим пустым и перейдем к `setup.py`:

```
from setuptools import setup, find_packages

setup(
    name="feed_aggregation",
    install_requires=["feedparser", "Klein", "Twisted", "treq"],
    package_dir={"": "src"},
    packages=find_packages("src") + ["twisted.plugins"],
)
```

Здесь объявляется имя проекта `feed_aggregation` и его зависимости: `feedparser` (для анализа каналов), `Klein` (для реализации нашего веб-приложения), `Twisted` (для `trial`) и `treq` (для загрузки каналов). Здесь же определяются настройки для `setuptools`, согласно которым поиск пакетов должен производиться в каталоге `src` и `feed_aggregation_plugin.py` должен быть включен в `twisted/plugins`.

Допустим, мы активировали новое виртуальное окружение для нашего проекта и перешли в его корневой каталог. Тогда мы можем выполнить команду:

```
pip install -e .
```

Флаг `-e` требует, чтобы команда `pip install` выполнила установку проекта в редактируемом режиме, который поместит указатель из виртуального окружения в корневой каталог проекта. В результате этого изменения появятся в виртуальном окружении, как только мы их сохраним.

Теперь команда `trial feed_aggregation` должна вывести следующее:

```
feed_aggregation.test.test_service
FeedAggregationTests
test_nothing ... [OK]
-----
Ran 1 tests in 0.001s
PASSED (successes=1)
```

демонстрируя, что мы фактически сделали наш проект доступным для `trial` через наше виртуальное окружение.

Тестирование `Klein` с помощью `StubTreq`

Теперь, имея возможность запускать тесты, можно заменить `FeedAggregationTests.test_nothing` методами, которые что-то тестируют. Этим «что-то», как говорилось выше, должен быть HTTP-интерфейс, посредством которого клиенты смогут обращаться к нашему приложению на `Klein`.

Нередко для тестирования HTTP-служб запускают облегченный веб-сервер, имитирующий реальное окружение для службы и, возможно, привязанный к предопределенному номеру порта на локальном хосте, и используют клиентскую HTTP-библиотеку для подключения к нему. Однако это довольно медленный способ, и, что еще хуже, порты являются ресурсом операционной системы, дефицит которого может вызвать нестабильность в выполнении тестов, которые их приобретают.

К счастью, мощь транспортов и протоколов `Twisted` позволяет запускать пары HTTP-клиент/сервер в памяти и тестировать их взаимодействия. В частности, в состав библиотеки `treq` входит мощная утилита тестирования `treq.testing.StubTreq`. Экземпляры `StubTreq` имеют тот же интерфейс, что и модуль `treq`, поэтому код, получающий `treq` через внедрение зависимостей, может использовать реализацию заглушки в тестах. Проект `treq` гарантирует соответствие API утилиты `StubTreq` и модуля `treq`, поэтому нам в наших тестах не требуется выполнять никаких дополнительных проверок.

`StubTreq` принимает в первом аргументе экземпляр `twisted.web.resource.Resource`, чьи ответы определяют результаты различных вызовов `treq`. Поскольку экземпляры `Klein` имеют метод `resource()`, генерирующий `twisted.web.resource.Resource`, мы можем связать `StubTreq` с нашим веб-приложением и получить HTTP-клиента в памяти, подходящего для наших тестов.

Заменяем `test_nothing` методом, использующим `StubTreq` для отправки запроса по корневому адресу URL нашей службы:

```
# src/feed_aggregation/tests/test_service.py
from twisted.trial.unittest import SynchronousTestCase
from twisted.internet import defer
from treq.testing import StubTreq
from .. import FeedAggregation

class FeedAggregationTests(SynchronousTestCase):
    def setUp(self):
        self.client = StubTreq(FeedAggregation().resource())

    @defer.inlineCallbacks
    def test_requestRoot(self):
        response = yield self.client.get(u'http://test.invalid/')
        self.assertEqual(response.code, 200)
```

Метод `setUp` создает экземпляр `StubTreq`, связанный с `twisted.web.resource.Resource` нашего приложения `FeedAggregation`. Метод `test_requestRoot` использует этого клиента для отправки запроса GET ресурсу и проверяет, был ли получен ответ, сообщаящий об успехе.

Обратите внимание, что для нашего теста значение имеет только часть адреса URL, передаваемого в `self.client.get`, которая определяет путь. Но `treq` и `StubTreq` могут посылать запросы только по полному адресу, включающему схеме и доменное имя хоста, поэтому мы использовали домен `.invalid`, чтобы удовлетворить это требование. Домен верхнего уровня `.invalid` имеет особое значение – он никогда не преобразуется в реальный интернет-адрес, что делает его идеальным выбором для использования в тестах.

Если теперь запустить новую версию `FeedAggregationTests` командой `trial feed_aggregation`, она потерпит неудачу с ошибкой `AttributeError`, потому что экземпляры нашего класса `FeedAggregation` не имеют метода `resource`. Однако добавление правильной реализации этого метода не приведет к успешному выполнению теста; нам также нужно сконструировать приложение Klein, отвечающее на запрос с адресом `/`. Изменим модуль `_service`, чтобы устранить эти недостатки.

```
# src/feed_aggregation/_service.py
from klein import Klein

class FeedAggregation(object):
    _app=Klein()
    def resource(self):
        return self._app.resource()

    @_app.route("/")
    def root(self, request):
        return b""
```

Новый метод `resource` делегирует вызов приложению Klein, связанному с классом. Это пример реализации закона *Деметры* – свода правил проектирования программного обеспечения, – запрещающего вызывать методы атрибутов экземпляра; вместо этого в экземпляре должны иметься методы делегирования, такие как `FeedAggregation.resource`, обертывающие методы его

атрибутов, чтобы код, использующий `FeedAggregation`, оставался в неведении о его внутренней реализации. Нашему приложению `Klein` мы дали имя `_app`, чтобы подчеркнуть, что оно является частью внутреннего, приватного API класса `FeedAggregation`.

Метод `root` действует как самый обычный обработчик для пути `/` в URL и вместе с `FeedAggregation.resource` обеспечивает успешное выполнение теста `FeedAggregation.test_requestRoot`.

Мы завершили первый цикл разработки на основе тестирования. Сначала мы написали минимальный тест, который терпит неудачу, а затем обеспечили его прохождение, добавив минимально необходимый для этого код.

Теперь двинемся дальше и заменим `FeedAggregationTests` более полным набором тестов, проверяющим представление информации о каналах в обоих форматах, HTML и JSON.

```
# src/feed_aggregation/test/test_service.py
import json
from lxml import html
from twisted.internet import defer
from twisted.trial.unittest import SynchronousTestCase
from treq.testing import StubTreq
from .. import FeedAggregation

class FeedAggregationTests(SynchronousTestCase):
    def setUp(self):
        self.client = StubTreq(FeedAggregation().resource())

    @defer.inlineCallbacks
    def get(self, url):
        response = yield self.client.get(url)
        self.assertEqual(response.code, 200)
        content = yield response.content()
        defer.returnValue(content)

    def test_renderHTML(self):
        content = self.successResultOf(self.get(u"http://test.invalid/"))
        parsed = html.fromstring(content)
        self.assertEqual(parsed.xpath(u'/html/body/div/table/tr/th/a/text()'),
                          [u"First feed", u"Second feed"])
        self.assertEqual(parsed.xpath('/html/body/div/table/tr/th/a/@href'),
                          [u"http://feed-1/", u"http://feed-2/"])
        self.assertEqual(parsed.xpath('/html/body/div/table/tr/td/a/text()'),
                          [u"First item", u"Second item"])
        self.assertEqual(parsed.xpath('/html/body/div/table/tr/td/a/@href'),
                          [u"#first", u"#second"])

    def test_renderJSON(self):
        content = self.successResultOf(self.get(
            u"http://test.invalid/?json=true"))
        parsed = json.loads(content)
        self.assertEqual(
            parsed,
            {u"feeds": [{u"title": u"First feed", u"link": u"http://feed-1/",
```

```
u"items": [{u"title": u"First item", u"link": u"#first"}]},
{u"title": u"Second feed", u"link": u"http://feed-2/",
u"items": [{u"title": u"Second item", u"link": u"#second"}]}}
```

Это более сложный набор тестов. В нем определено два теста, `test_renderHTML` и `test_renderJSON`, проверяющих структуру и содержимое ответов в формате HTML и JSON, которые должна вернуть наша веб-служба `FeedAggregation`. Метод `test_requestRoot` мы заменили методом `get`, который может использоваться в обоих тестах – `test_renderHTML` и `test_renderJSON` – для извлечения из нашей веб-службы содержимого с указанным URL-адресом. И оба теста – `test_renderHTML` и `test_renderJSON` – используют `SynchronousTestCase.successResultOf`, чтобы дождаться, пока экземпляр `Deferred`, возвращаемый методом `get`, разрешится и из него можно будет извлечь полученный ответ.

Метод `test_renderHTML` использует библиотеку `lxml` (<https://lxml.de/>) для анализа и проверки разметки HTML, возвращаемой нашим приложением `Klein`. Поэтому мы должны добавить `lxml` в список `install_requires` в файле `setup.py`. Обратите внимание, что виртуальное окружение можно синхронизировать с зависимостями проекта, выполнив команду `pip install -e`.

`XPath` определяет местонахождение определенных элементов в дереве DOM и извлекает их содержимое и атрибуты. Ожидаемая структура таблицы соответствует той, что мы разработали в нашем прототипе: каналы помещаются в таблицы `table`, заголовки которых являются ссылками на домашние страницы каналов, а строки – на элементы каждого канала.

Метод `test_renderJSON` запрашивает информацию о каналах в формате JSON, преобразует полученный результат в словарь и сравнивает его с ожидаемым результатом.

Эти новые тесты, естественно, терпят неудачу, потому что существующая реализация `FeedAggregation` возвращает ответ с пустым телом. Давайте добавим в `FeedAggregation` минимально необходимый код, обеспечивающий успешное прохождение тестов.

```
# src/feed_aggregation/_service.py
from klein import Klein, Plating
from twisted.web.template import tags as t, slot

class FeedAggregation(object):
    _app = Klein()
    _plating = Plating(
        tags=t.html(
            t.head(t.title("Feed Aggregator 2.0")),
            t.body(slot(Plating.CONTENT)))

    def resource(self):
        return self._app.resource()

    @_plating.routed(
        _app.route("/"),
        t.div(render="feeds:list")(slot("item")),
    )
```

```
def root(self, request):
    return {u"feeds": [
        t.table(t.tr(t.th(t.a(href=u"http://feed-1/")(u"First feed"))),
            t.tr(t.td(t.a(href=u"#first")(u"First item")))),
        t.table(t.tr(t.th(t.a(href=u"http://feed-2/")(u"Second feed"))),
            t.tr(t.td(t.a(href=u"#second")(u"Second item"))))
    ]}
```

У нас пока нет тестов, проверяющих извлечение содержимого каналов, поэтому данная реализация еще не получает RSS-каналы. Вместо этого, чтобы удовлетворить тесты, она возвращает жестко закодированные данные, соответствующие проверяемым в тестах. Кроме того, она напоминает наш прототип: метод `root` обрабатывает путь к корневому URL, который использует метод отображения `list` для преобразования последовательности `twisted.web.template.tags` в HTML.

Эта версия `FeedAggregation` успешно преодолевает тест `test_renderHTML`, но терпит неудачу в тесте `test_renderJSON`:

```
(feed_aggregation) $ trial feed_aggregation
feed_aggregation.test.test_service
FeedAggregationTests
test_renderHTML ... [OK]
test_renderJSON ... [ERROR]
[ERROR]
=====
[ERROR]
Traceback (most recent call last):
...
exceptions.TypeError: Tag('table', ...) not JSON serializable
feed_aggregation.test.test_service.FeedAggregationTests.test_renderJSON
=====
[ERROR]
Traceback (most recent call last):
...
twisted.trial.unittest.FailTest: 500 != 200
feed_aggregation.test.test_service.FeedAggregationTests.test_renderJSON
-----
Ran 2 tests in 0.029s
FAILED (failures=1, errors=1, successes=1)
```

Вторая ошибка соответствует утверждению `self.assertEqual(response.code, 200)` в `FeedAggregationTests.get`, но она является лишь следствием истинной – первой ошибки: библиотека `Klein` не смогла сериализовать в формат JSON `tag`, который вернул вызов `FeedAggregation.root`.

Самое простое решение – определить момент, когда клиент потребует ответа в формате JSON и вернуть ему сериализованный словарь. Текущий дизайн требует копирования данных, необходимых для выполнения тестов, поэтому, решая эту проблему, попутно добавим контейнерные классы для хранения

данных из канала, а также класс верхнего уровня, который будет хранить адрес источника канала и управлять его представлением. Это позволит нам получить данные один раз и потом преобразовывать их в любой формат – HTML или JSON. Мы можем реорганизовать FeedAggregation так, что он будет принимать экземпляры контейнеров каналов в своем методе инициализации, благодаря чему тесты смогут использовать для проверки свои тестовые данные. Напишем `_service.py` в соответствии с этим решением, а чтобы сохранить наш код компактным и ясным, используем библиотеку Хайнека Шлавака (Hynek Schlawack) `attrs` (<https://attrs.readthedocs.io>); не забудьте добавить ее в вызов `install_requires` в `setup.py`.

```
# src/feed_aggregation/_service.py
import attr
from klein import Klein, Plating
from twisted.web.template import tags as t, slot

@attr.s(frozen=True)
class Channel(object):
    title = attr.ib()
    link = attr.ib()
    items = attr.ib()

@attr.s(frozen=True)
class Item(object):
    title = attr.ib()
    link = attr.ib()

@attr.s(frozen=True)
class Feed(object):
    _source = attr.ib()
    _channel = attr.ib()

    def asJSON(self):
        return attr.asdict(self._channel)

    def asHTML(self):
        header = t.th(t.a(href=self._channel.link)
                      (self._channel.title))
        return t.table(t.tr(header))(
            [t.tr(t.td(t.a(href=item.link)(item.title)))
             for item in self._channel.items])

@attr.s
class FeedAggregation(object):
    _feeds = attr.ib()
    _app = Klein()
    _plating = Plating(
        tags=t.html(
            t.head(t.title("Feed Aggregator 2.0")),
            t.body(slot(Plating.CONTENT))))

    def resource(self):
        return self._app.resource()
```

```

@_plating.routed(
    _app.route("/"),
    t.div(render="feeds:list")(slot("item")),
)
def root(self, request):
    jsonRequested = request.args.get(b"json")
    def convert(feed):
        return feed.asJSON() if jsonRequested else feed.asHTML()
    return {"feeds": [convert(feed) for feed in self._feeds]}

```

Библиотека `attrs` упрощает определение контейнерных классов, таких как `Channel` и `Item`. Декоратор класса `attr.s` генерирует метод `init` контейнерного класса, который устанавливает переменные экземпляра, соответствующие переменным класса `attr.ib`.

Библиотека `attrs` также упрощает определение классов, экземпляры которых являются *неизменяемыми*, если передать аргумент `frozen` декоратора со значением `True`. Неизменяемость – важное свойство наших контейнерных классов: они представляют внешние данные, изменение которых после получения наверняка было бы ошибкой. Библиотеки `attrs` и `lxml` необходимо добавить в список `install_requires` внутри `setup.py`.

Класс `Feed` обортывает исходный URL-адрес канала и экземпляр `Channel`, представляющий его содержимое, и предлагает два метода представления. Метод `asJSON` использует `attrs.asdict` для рекурсивного преобразования экземпляра канала в словарь, который затем сериализуется в формат JSON, а метод `asHTML` возвращает дерево `twisted.web.template.tags`, которое будет отображаться механизмом шаблонов `Plating` из библиотеки `Klein`.

Метод `FeedAggregation.root` теперь проверяет параметр запроса `json`, доступный в словаре `args`, и определяет формат ответа, JSON или HTML, и вызывает `asJSON` или `asHTML` соответственно.

Наконец, класс `FeedAggregation` теперь сам декорирован декоратором `attr.s` и получает метод инициализации, который принимает итерируемые объекты `Feed` для отображения.

В результате всех этих изменений необходимо переписать метод `FeedAggregationTests.setUp`, чтобы передать итерируемый объект `Feed` в его экземпляр `FeedAggregation`:

```

# src/feed_aggregation/test/test_service.py
...
from .._service import Feed, Channel, Item
FEEDS = (
    Feed("http://feed-1.invalid/rss.xml",
        Channel(title="First feed", link="http://feed-1/",
            items=(Item(title="First item", link="#first"),)),
    Feed("http://feed-2.invalid/rss.xml",
        Channel(title="Second feed", link="http://feed-2/",
            items=(Item(title="Second item", link="#second"),)),
)

```

```

class FeedAggregationTests(SynchronousTestCase):
    def setUp(self):
        self.client = StubTreq(FeedAggregation(FEEDS).resource())
    ...

```

Эта последняя версия имеет свои преимущества, самое очевидное из них: `test_renderJSON` теперь успешно выполняется, а сами тестовые данные находятся в одном месте с тестами, что упрощает их синхронизацию с утверждениями.

Однако в ней есть и свои недостатки. `FeedAggregation` стал бесполезен как служба агрегирования каналов, в отсутствие возможности получения RSS-каналов, и тесты теперь импортируются и зависят от наших контейнерных классов. Тесты, зависящие от внутренних деталей реализации, слишком хрупкие и трудно поддаются рефакторингу.

Мы исправим эти недостатки чуть ниже, написав логику извлечения канала.

Тестирование treq с помощью Klein

Выше мы использовали `StubTreq` для тестирования приложения на Klein. Теперь пойдем в обратную сторону и протестируем работу `treq`.

И снова начнем с написания тестов. Добавим их в модуль `test_service` с новыми инструкциями импортирования и нашим новым набором тестов.

```

# src/feed_aggregation/test/test_service.py
import attr
...
from hyperlink import URL
from klein import Klein
from lxml.builder import E
from lxml.etree import tostring
...
from .. import FeedRetrieval

@attr.s
class StubFeed(object):
    _feeds = attr.ib()
    _app = Klein()

    def resource(self):
        return self._app.resource()

    @_app.route("/rss.xml")
    def returnXML(self, request):
        host = request.getHeader(b 'host')
        try:
            return self._feeds[host]
        except KeyError:
            request.setResponseCode(404)
            return b'Unknown host: ' + host

    def makeXML(feed):
        channel = feed._channel
        return tostring(

```

```

        E.rss(E.channel(E.title(channel.title), E.link(channel.link),
                        *[E.item(E.title(item.title), E.link(item.link))
                          for item in channel.items],
                        version = u"2.0")))

```

```

class FeedRetrievalTests(SynchronousTestCase):
    def setUp(self):
        service = StubFeed(
            {URL.from_text(feed._source).host.encode('ascii'): makeXML(feed)
              for feed in FEEDS})
        treq = StubTreq(service.resource())
        self.retriever = FeedRetrieval(treq=treq)

    def test_retrieve(self):
        for feed in FEEDS:
            parsed = self.successResultOf(
                self.retriever.retrieve(feed._source))
            self.assertEqual(parsed, feed)

```

Класс `FeedRetrievalTests`, так же как `FeedAggregationTests`, зависит от некоторых новых понятий. `StubFeed` – это приложение `Klein`, маршрут `/rss.xml` которого возвращает XML-документ для хоста, указанного в запросе. Это позволяет ему возвращать разные ответы для `http://feed-1.invalid` и `http://feed-2.invalid`. В качестве меры предосторожности в ответ на запросы с неизвестным именем хоста возвращается вполне информативный ответ: 404 «Not Found».

Функция `makeXML` преобразует экземпляр `Feed` и связанные с ним экземпляры `Item` в XML-документ, совместимый с RSS 2.0. Мы используем фабрику тегов `E` из `lxml.builder`, чей API напоминает `twisted.web.template.tags`, но для тегов XML, и сериализуем полученное дерево тегов в байты с помощью `lxml.etree.tostring` (несмотря на свое имя, в Python 3 этот метод *возвращает* строку `bytes`).

Метод настройки `FeedRetrievalTests.setUp` создает список каналов `Feed` и передает его экземпляру `StubFeed`, который связывает список с экземпляром `StubTreq`. Тот, в свою очередь, передает список экземпляру `FeedRetrieval`, который будет хранить результаты попыток извлечения каналов. Параметризация класса реализацией `treq` – это пример внедрения зависимостей, упрощающий процесс написания тестов.

Обратите внимание, что имя хоста для каждого канала мы получаем из URL в элементе `link` с помощью `hyperlink.URL`. Экземпляры класса `URL` из библиотеки `Hyperlink` (<https://hyperlink.readthedocs.io>) являются неизменяемыми объектами, представляющими URL-адреса. Библиотека `Hyperlink` была выделена из модуля `twisted.python.url` и предлагает более широкие возможности. Теперь `Twisted` зависит от этой библиотеки, поэтому она неявно доступна во всех проектах, зависящих от `Twisted`. Однако, как в случае с любой другой зависимостью, предпочтительнее объявить ее явно, поэтому мы должны добавить пакет `hyperlink` в список `install_requires` в файле `setup.py`. Вот как теперь должен выглядеть код в `setup.py`:

```

# setup.py
from setuptools import setup, find_packages

```

```

setup(
    name="feed_aggregation",
    install_requires=["attrs", "feedparser", "hyperlink", "Klein",
                     "lxml", "Twisted", "treq"],
    package_dir={"": "src"},
    packages=find_packages("src")+["twisted.plugins"],
)

```

(Напомню, что о необходимости добавить `attrs` и `lxml` уже говорилось выше.)

Один из тестов в наборе `FeedAggregationTests` – `test_retrieve` – утверждает, что `FeedRetrieval.retrieve` преобразует содержимое канала, полученное из адреса URL в `_source`, в объект `Feed`, соответствующий его XML-представлению.

Теперь, когда у нас есть тест для проверки извлечения каналов, можно реализовать эту операцию. Сначала добавим `FeedRetrieval` в `src/feed_aggregation/__init__.py`, чтобы его можно было импортировать без помощи приватного API:

```

# src/feed_aggregation/ init .py
from ._service import FeedAggregation, FeedRetrieval

__all__ = ["FeedAggregation", "FeedRetrieval"]

```

И реализуем минимальный код, который обеспечит успешное выполнение теста:

```

# src/feed_aggregation/_service.py
...
import treq
import feedparser

@attr.s
class FeedRetrieval(object):
    _treq = attr.ib()

    def retrieve(self, url):
        feedDeferred = self._treq.get(url)
        feedDeferred.addCallback(treq.content)
        feedDeferred.addCallback(feedparser.parse)

    def toFeed(parsed):
        feed = parsed['feed']
        entries = parsed['entries']
        channel = Channel(feed['title'], feed['link'],
                          tuple(Item(e['title'], e['link'])
                                for e in entries))
        return Feed(url, channel)

    feedDeferred.addCallback(toFeed)
    return feedDeferred

```

Как и ожидалось, `FeedRetrieval` принимает реализацию `treq` в виде единственного аргумента через декоратор класса `attr.s` и `_treq attr.ib`. Его метод `retrieve` следует той же схеме, что и наша пробная программа: сначала он использует `treq` для извлечения содержимого по указанному адресу URL, а затем

использует `feedparser` для преобразования полученного документа XML в словарь Python.

Далее, метод `toFeed` извлекает заголовок канала, ссылку, заголовки элементов и ссылки на них и затем собирает их в контейнеры `Channel`, `Item` и `Feed`.

Эта версия `FeedRetrieval` обеспечивает благополучное выполнение теста, но в ней отсутствует обработка ошибок. Как быть, если канал был удален или был получен недействительный документ XML? В текущей реализации выполнение экземпляра `Deferred`, возвращаемого методом `FeedRetrieval.retrieve`, завершится с исключением, что может стать проблемой.

Веб-сайт и служба JSON никогда не должны возвращать информацию с трассировкой стека. В то же время должен быть какой-то обработчик, который сохранит трассировку для нужд отладки. К счастью, в Twisted имеется развитая система журналирования, которую можно использовать для трассировки поведения нашего приложения.

Журналирование с использованием `twisted.logger`

В фреймворке Twisted уже довольно давно имеется своя система журналирования. Начиная с версии Twisted 15.2.0 предпочтительным методом журналирования событий в программах на Twisted считается `twisted.logger`.

Подобно модулю `logging` в стандартной библиотеке, приложения имеют возможность писать сообщения в журнал с разными уровнями важности, вызывая соответствующие методы экземпляра `twisted.logger.Logger`. Например, следующий код выводит сообщения с уровнем важности `info`.

```
from twisted.logger import Logger
Logger().info("A message with{key}", key="value")
```

Подобно модулю `logging`, методы журналирования, такие как `Logger.info`, принимают строку формата и значения для интерполяции; в отличие от модуля `logging`, здесь строка формата оформляется в *новом стиле*, и она отправляется вместе с журналируемым событием. Также, в отличие от стандартной системы журналирования `logging`, класс `twisted.logger.Loggers` не является родоначальником иерархии, а просто передает получаемые сообщения *наблюдателям*. Сохранение строки формата обеспечивает одну из самых мощных возможностей `twisted.logger`: он способен выводить журналируемые сообщения в традиционном формате, понятном человеку, а также генерировать объекты, сериализованные в формат JSON. В последнем случае открывается возможность сложной фильтрации и накопления в таких системах, как Kibana. Мы посмотрим, как переключаться между этими форматами, когда напишем плагин для нашего приложения агрегирования каналов.

Кроме того, класс `Logger` использует протокол дескриптора для сбора информации о связанном классе, поэтому мы создадим экземпляр `Logger` для нашего класса `FeedRetrieval`. Затем организуем вывод сообщений до запроса канала и после его успешного получения или ошибки с исключением. Однако прежде

мы должны определить, каким значением должен разрешаться экземпляр `Deferred`, возвращаемый методом `FeedRetrieval.retrieve`, при появлении исключения. Это не может быть экземпляр `Feed`, потому что у нас не будет документа XML для преобразования в экземпляр `Channel`; но `FeedAggregation` ожидает получить объект, имеющий методы `asJSON` и `asHTML`, реализации которых существуют только в `Feed`.

Эту проблему можно решить, воспользовавшись полиморфизмом и определив новый класс `FailedFeed`, представляющий ошибку получения канала в `FeedRetrieval`. Он будет иметь тот же интерфейс, что и `Feed`, предлагая свои методы `asJSON` и `asHTML`, которые отображают ошибку в соответствующем формате.

Как обычно, сначала напишем тесты. Ошибки, которые могут возникнуть в `FeedRetrieval.retrieve`, можно разделить на две категории: ответ с любым кодом состояния, отличным от 200, и любое другое исключение. Первую категорию ошибок мы смоделируем с использованием своего типа исключений `ResponseNotOK`. Оно будет генерироваться в методе `retrieve` и обрабатываться внутри. Мы сможем симитировать его в тестах, запросив канал с хоста, о существовании которого `StubFeed` ничего не знает. Вторую категорию ошибок можно симитировать, передав в `StubFeed` ссылку на хост, возвращающий пустую строку, которую `feedparser` не сможет проанализировать. Итак, добавим несколько тестов в класс `FeedRetrievalTests`.

```
# src/feed_aggregation/test/test_service.py
from .. import FeedRetrieval
from .._service import Feed, Channel, Item, ResponseNotOK
from xml.sax import SAXParseException

...

class FeedRetrievalTests(SynchronousTestCase):
    ...
    def assertTag(self, tag, name, attributes, text):
        self.assertEqual(tag.tagName, name)
        self.assertEqual(tag.attributes, attributes)
        self.assertEqual(tag.children, [text])

    def test_responseNotOK(self):
        noFeed = StubFeed({})
        retriever = FeedRetrieval(StubTreq(noFeed.resource()))
        failedFeed = self.successResultOf(
            retriever.retrieve("http://missing.invalid/rss.xml"))
        self.assertEqual(
            failedFeed.asJSON(),
            {"error": "Failed to load http://missing.invalid/rss.xml: 404"})
        self.assertTag(failedFeed.asHTML(),
            "a", {"href": "http://missing.invalid/rss.xml"},
            "Failed to load feed: 404")

    def test_unexpectedFailure(self):
        empty = StubFeed({b"empty.invalid": b""})
```

```

retriever = FeedRetrieval(StubTreq(empty.resource()))
failedFeed = self.successResultOf(
    retriever.retrieve("http://empty.invalid/rss.xml"))
msg = "SAXParseException('no element found',)"
self.assertEqual(
    failedFeed.asJSON(),
    {"error": "Failed to load http://empty.invalid/rss.xml: " + msg}
)
self.assertTag(failedFeed.asHTML(),
    "a", {"href": "http://empty.invalid/rss.xml"},
    "Failed to load feed: " + msg)
self.assertTrue(self.flushLoggedErrors(SAXParseException))

```

Метод `assertTag` проверяет имя, атрибуты и дочерние элементы в заданном теге в дереве `twisted.web.template` и тем самым упрощает реализацию методов `test_responseNotOK` и `test_unexpectedFailure`.

Метод `test_responseNotOK` создает пустое приложение `StubFeed`, которое на любой запрос отвечает кодом состояния 404. Затем он проверяет, что при попытке извлечь содержимое по адресу URL возвращается `Deferred`, разрешившийся экземпляром `FailedFeed`, и отображает его в формат JSON и в дерево тегов. Документ JSON должен содержать URL кода состояния HTTP, а разметка HTML должна содержать гиперссылку на канал, получить который не удалось, и код состояния.

Метод `test_unexpectedFailure` создает `StubFeed`, отвечающий пустой строкой на запрос к `empty.invalid`. Документы HTML и JSON, сгенерированные полученным экземпляром `FailedFeed`, проверяются на наличие исходного URL, а также строкового представления исключения, вызвавшего отказ. Мы выбрали строковое представление `repr`, потому что сообщения во многих исключениях, таких как `KeyError`, трудно понять без имени класса исключения.

Обратите особое внимание на последнюю строку в методе `test_unexpectedFailure`. `trial`, в отличие от `unittest` в Python, тест потерпит неудачу, если не работает зарегистрированные ошибки, возникшие в тестируемом коде. Это не относится к исключениям, возбуждаемым в самом тесте.

`SynchronousTestCase.flushLoggedErrors` возвращает список `twisted.python.failure.Failure` с ошибками, записанными в журнал до этого момента; если ему передать типы исключений, он вернет только ошибки, соответствующие этим типам. Префикс `flush` в имени `flushLoggedErrors` означает, что это деструктивный вызов, то есть данный экземпляр `Failure` не появится в списках, возвращаемых двумя последовательными вызовами. Тест будет терпеть неудачу, если после его завершения останется непустой список зарегистрированных ошибок. Проверка появления хотя бы одного исключения `SAXParseException` в `feedparser` имеет побочный эффект – она очищает список зарегистрированных ошибок, что обеспечивает успешное завершение теста.

Теперь напишем код, который обеспечит успешное выполнение этого теста. Мы покажем новую версию `FeedRetrieval` целиком, чтобы вы могли увидеть полный контекст обработки ошибок.

```

# src/feed_aggregation/_service.py
...
import treq import feedparser
from twisted.logger import Logger
from functools import partial
...

@attr.s(frozen=True)
class FailedFeed(object):
    _source = attr.ib()
    _reason = attr.ib()

    def asJSON(self):
        return {"error": "Failed to load{ }: {}".format(
            self._source, self._reason)}

    def asHTML(self):
        return t.a(href=self._source)(
            "Failed to load feed: {}".format(self._reason))

class ResponseNotOK(Exception):
    """Ответ для кода состояния, отличного от 200."""

@attr.s
class FeedRetrieval(object):
    _treq = attr.ib()
    _logger = Logger()

    def retrieve(self, url):
        self._logger.info("Downloading feed{url}", url=url)
        feedDeferred = self._treq.get(url)

        def checkCode(response):
            if response.code != 200:
                raise ResponseNotOK(response.code)
            return response

        feedDeferred.addCallback(checkCode)
        feedDeferred.addCallback(treq.content)
        feedDeferred.addCallback(feedparser.parse)

    def toFeed(parsed):
        if parsed[u'bozo']:
            raise parsed[u'bozo_exception']
        feed=parsed[u'feed']
        entries = parsed[u'entries']
        channel = Channel(feed[u'title'], feed[u'link'],
            tuple(Item(e[u'title'], e[u'link'])
                for e in entries))
        return Feed(url, channel)

    feedDeferred.addCallback(toFeed)

    def failedFeedWhenNotOK(reason):
        reason.trap(ResponseNotOK)
        self._logger.error("Could not download feed{url}:{code}",
            url=url, code=str(reason.value))

```

```

    return FailedFeed(url, str(reason.value))

def failedFeedOnUnknown(failure):
    self._logger.failure("Unexpected failure downloading{url}",
                        failure=failure, url=url)
    return FailedFeed(url, repr(failure.value))

feedDeferred.addErrback(failedFeedWhenNotOK)
feedDeferred.addErrback(failedFeedOnUnknown)
return feedDeferred

```

Класс `FailedFeed` реализует методы `asJSON` и `asHTML` в соответствии с интерфейсом `Feed`. Поскольку метод инициализации является приватным, мы можем определить новый аргумент `reason` для описания ошибки загрузки канала.

Исключение `ResponseNotOK` представляет категорию ошибок, обусловленных получением кода состояния, отличного от 200, и первое изменение в самом методе `retrieve`: обратный вызов `checkCode` возбуждает исключение `ResponseNotOK`, когда `treq.get` возвращает недопустимый код состояния, который затем передается исключению.

Метод `toFeed` тоже изменился в соответствии с API отчетов об ошибках в `feedparser`. В `feedparser` выбран мягкий подход к парсингу, то есть сам метод `feedparser.parse` никогда не генерирует исключений; вместо этого он возвращает словарь с ключом `bozo`, имеющим значение `True`, и с ключом `bozo_exception`, содержащим фактическое исключение.

Эта ошибка попадает во вторую категорию неожиданных ошибок. Конечно, неожиданных ошибок может быть намного больше, и наш код должен обрабатывать их все.

Обработчик ошибок `failedFeedWhenNotOK` обрабатывает первую категорию, перехватывая исключение `ResponseNotOK` и регистрируя сообщение об ошибке с URL-адресом канала и кодом состояния ответа, а вторую категорию ошибок обрабатывает метод `failFeedOnUnknown`, который регистрирует сообщение с уровнем `critical`, в которое включает трассировку стека, полученную с помощью вспомогательного метода `Logger.failure`. Оба обработчика возвращают экземпляр `FailedFeed`, представляющий соответствующие ошибки в соответствии с ожиданиями тестов.

Порядок добавления обработчиков ответов и ошибок в `feedDeferred` имеет значение. Напомню, что когда обработчик ответа возбудит исключение, оно будет передано следующему обработчику ошибок. Добавляя обработчики ошибок после всех обработчиков ответов, мы ясно показываем, что они обрабатывают любые возникшие исключения. Также, поскольку обработчики ошибок сами могут возбуждать исключения, которые получают следующие за ними зарегистрированные обработчики ошибок, мы добавляем более конкретный обработчик `failedFeedWhenNotOK` перед универсальным `failedFeedOnUnknown`. То есть подобная последовательность добавления обработчиков ошибок эквивалентна следующему синхронному коду:

```

try:
    ...

```

```

except ResponseNotOK:
    self._logger.error(...)
    return FailedFeed(...)
except:
    self._logger.failure(...)
    return FailedFeed(...)

```

Запуск приложений Twisted с помощью twist

Мы разделили проект на две функционально независимые половины: FeedAggregation, обрабатывающий входящие веб-запросы, и FeedRetrieval, получающий и анализирующий RSS-каналы. Feed и FailedFeed связывают их, предлагая общий интерфейс, но чтобы объединить приложение в одно рабочее целое, нужно внести последний штрих.

Так же как наша пробная программа SimpleFeedAggregation, после получения входящего HTTP-запроса FeedAggregation должен вызвать FeedRetrieval. Такой поток управления предполагает, что экземпляр FeedAggregation должен обернуть экземпляр FeedRetrieval, чего легко добиться с помощью механизма внедрения зависимостей; вместо списка элементов Feed мы можем передать в FeedAggregation метод retrieve экземпляра FeedRetrieval и список URL каналов для запроса. Изменим для этого тесты в FeedAggregationTests:

```

# src/feed_aggregation/test/test_service.py
...
class FeedAggregationTests(SynchronousTestCase):
    def setUp(self):
        service = StubFeed(
            {URL.from_text(feed._source).host.encode('ascii'): makeXML(feed)
              for feed in FEEDS})
        treq = StubTreq(service.resource())
        urls = [feed._source for feed in FEEDS]
        retriever = FeedRetrieval(treq)
        self.client = StubTreq(
            FeedAggregation(retriever.retrieve, urls).resource())
        ...

```

Теперь добавим поддержку этого нового API в FeedAggregation:

```

# src/feed_aggregation/_service.py
@attr.s
class FeedAggregation(object):
    _retrieve = attr.ib()
    _urls = attr.ib()
    _app = Klein()
    _plating = Plating(
        tags=t.html(
            t.head(t.title("Feed Aggregator 2.0")),
            t.body(slot(Plating.CONTENT))))

    def resource(self):
        return self._app.resource()

```

```

    @_plating.routed(
        _app.route("/"),
        t.div(render="feeds:list")(slot("item")),
    )
    def root(self, request):
        def convert(feed):
            return feed.asJSON() if request.args.get(b"json") else feed.asHTML()
        return {"feeds": [self._retrieve(url).addCallback(convert)
                          for url in self._urls]}

```

Метод инициализации `FeedAggregation` принимает два новых аргумента: вызываемый объект `retrieve`, который принимает URL-адрес и возвращает `Deferred`, разрешающийся в экземпляр `Feed` или `FailedFeed`, и итерируемый объект `urls`, представляющий список URL-адресов RSS-каналов. Обработчик `root` объединяет их, применяя `_retrieve` к каждому адресу в списке `_urls`, а затем отображает результаты с помощью обратного вызова `convert`.

Теперь, когда мы можем объединить половину приложения, представляющую службу, с половиной, реализующей извлечение каналов, мы можем написать плагин для приложения `Twisted` в файле `src/twisted/plugins/feed_aggregation_plugin.py`, загружающий и запускающий службу агрегирования каналов.

Программа командной строки `twist`, входящая в состав фреймворка `Twisted`, позволяет запускать различные службы на основе `Twisted`, как, например, статический веб-сервер: `twist web --path=/path/to/serve`. Она допускает возможность расширения с использованием механизма плагинов `Twisted`. Давайте напишем плагин, запускающий нашу веб-службу агрегирования каналов.

```

# src/twisted/plugins/feed_aggregation_plugin.py
from twisted import plugin
from twisted.application import service, strports
from twisted.python.usage import Options
from twisted.web.server import Site
import treq
from feed_aggregation import FeedAggregation, FeedRetrieval
from zope.interface import implementer

class FeedAggregationOptions(Options):
    optParameters = [["listen", "l", "tcp:8080", "How to listen for requests"]]

@implementer(plugin.IPlugin, service.IServiceMaker)
class FeedAggregationServiceMaker(service.Service):
    tapname = "feed"
    description = "Aggregate RSS feeds."
    options = FeedAggregationOptions

    def makeService(self, config):
        urls = ["http://feeds.bbci.co.uk/news/technology/rss.xml",
                "http://planet.twistedmatrix.com/rss20.xml"]
        aggregator = FeedAggregation(FeedRetrieval(treq).retrieve, urls)
        factory = Site(aggregator.resource())
        return strports.service(config['listen'], factory)

makeFeedService = FeedAggregationServiceMaker()

```

`twisted.application.service.IService` – это модуль кода, запускаемый утилитой `twist`, `twisted.application.service.IServiceMaker` позволяет `twist` отыскивать провайдеров `IService`, а `twisted.plugin.IPlugin` помогает `twisted.plugin` найти плагины. Класс `FeedAggregationServiceMaker` реализует оба этих интерфейса, поэтому его экземпляры в `twisted/plugins` будут обнаружены утилитой `twist`.

Атрибут `tapname` представляет имя команды, под которой будет доступна наша служба в `twist`, а атрибут `description` содержит описание, которое `twist` будет выводить перед пользователями команд. Атрибут `options` содержит экземпляр `twisted.python.usage.Options`, который преобразует параметры командной строки в словарь и передает его в метод `makeService`. Наш подкласс `FeedAggregationOptions` определяет единственный параметр командной строки, `--listen` или `-l`, который представляет *строку описания конечной точки*, по умолчанию `tcp:8080`. Что это такое и как работают эти строки, рассказывается чуть ниже.

`FeedAggregationServiceMaker.makeService` принимает конфигурацию, проанализированную классом `Options`, и возвращает экземпляр `IService`, который запускает нашу веб-службу `FeedAggregation`. Здесь экземпляр `FeedAggregation` конструируется точно так же, как в тестах, только на этот раз в `FeedRetrieval` передается фактическая реализация `treq`.

Класс `twisted.web.server.Site` в действительности является фабрикой, которая знает, как отвечать на запросы HTTP. В первом аргументе он принимает `twisted.web.resource` – ресурс, который будет отвечать на входящие запросы, как это делал `StubTreq` в наших тестах, поэтому мы снова используем `FeedAggregation.resource`, чтобы создать его на основе приложения `Klein`.

Функция `strports.service` анализирует строку с описанием конечной точки в экземпляре `IService`, который управляет указанным портом. Строки описания конечных точек придают приложениям `Twisted` дополнительную гибкость, позволяя определить, какие протоколы и транспорты следует использовать для связи с клиентами.

Значение по умолчанию `tcp:8080` заставляет `Twisted` занять TCP-порт 8080 на всех доступных сетевых интерфейсах и ассоциировать транспорт TCP с экземплярами протоколов, созданными фабрикой `Site`. Вместо него можно использовать `ssl:port=8443;privateKey=server.pem`, чтобы настроить прием TLS-соединений через порт 8443 с использованием сертификата в `server.pem`. Протоколы, созданные фабрикой `Site`, затем привязываются к TLS-транспортам, которые автоматически шифруют и дешифруют соединения с клиентами. Парсеры `strports` можно расширять сторонними плагинами, например `txorcon` (<https://txorcon.readthedocs.io/en/latest/>) позволяет запустить сервер TOR, передавая строку описания конечной точки `onion:`.

Теперь можно попробовать вызвать службу агрегирования каналов с помощью программы `twist`:

```
$ twist feed
2018-02-01T12:12:12-0800 [-] Site starting on 8080
2018-02-01T12:12:12-0800 [twisted.web.server.Site#info] Starting factory
```

```
<twisted.web.serve
2018-02-01T12:12:12-0800 [twisted.application.runner._runner.Runner#info]
Starting reactor.
2018-02-01T12:13:13-0800 [feed_aggregation._service.FeedRetrieval#info]
Downloading feed
2018-02-01T12:13:13-0800 [feed_aggregation._service.FeedRetrieval#info]
Downloading feed
...
```

`twist` настраивает `twisted.logger` для отправки журналируемых сообщений в стандартный вывод. Сообщения `FeedRetrieval` имеют уровень важности `info` и свидетельствуют, что клиент получил доступ к нашему приложению.

Также программа `twist` может выводить журналируемые сообщения в формате JSON, если передать ей ключ `--log-format=json`:

```
$ twist --log-format=json feed
...
{"log_namespace": "...FeedRetrieval", "url": "http://feeds.bbc.co.uk/news/
technology/rss.x
{"log_namespace": "...FeedRetrieval", "url": "http://planet.twistedmatrix.
com/rss20.xml", .
...
```

В этом примере вывода опущено много деталей, чтобы сделать его более читабельным. Но обратите внимание, что параметр `url`, который передается в вызов `logger.info` в методе `FeedRetrieval._retrieve`, является свойством возвращаемых объектов JSON. Это позволит службе агрегирования журналов извлекать данные из журнала без применения эвристик, таких как регулярные выражения. Как и в `strports`, это изменение поведения не потребовало от нас изменить код приложения.

Итоги

В данной главе представлены библиотеки `Klein` и `treq`. Они обе являются высокоуровневыми обертками для веб-API в фреймворке `Twisted`, упрощающими решение типичных задач.

Мы написали службу агрегирования каналов RSS 2.0 с использованием библиотеки `feedparser`, начав с простого прототипа, а затем, руководствуясь принципом разработки через тестирование, создали полнофункциональное `Twisted`-приложение, запускаемое с помощью утилиты командной строки `twist`. Мы протестировали нашу веб-службу с помощью `treq.testing.StubTreq`, не выполняя реальных сетевых запросов, а также с помощью `SynchronousTestCase` убедились, что параллельные операции завершаются детерминированно при разных входных данных. Мы увидели, как механизм шаблонов `Plating` в библиотеке `Klein` позволяет создавать веб-службы, способные возвращать ответ в любом из форматов – JSON или HTML, – и как можно регистрировать структурированные данные с помощью `twisted.logger`.

Использование сторонних библиотек, не поддерживающих параллельное выполнение, таких как `feedparser`, `lxml` и `attrs`, демонстрирует, как программы Twisted интегрируются с современной экосистемой Python. В то же время в нашей программе применялись классические идеи Twisted, такие как `Deferreds`; наша служба агрегирования каналов показала, какую мощь дает объединение обширной коллекции библиотек для Python с механизмами Twisted.

Часть II



ПРОЕКТЫ

Глава 4

Twisted в Docker

Технология Docker нередко используется в архитектурах микросервисов, основанных на создании различных компонентов, взаимодействующих по сети. Фреймворк Twisted с его собственной поддержкой сетевых парадигм часто хорошо подходит для реализации архитектур на основе Docker.

Docker, и контейнеры в целом, – это довольно новая технология. И сам этот инструмент, и взгляды на его использование быстро развиваются. В этой главе вы познакомитесь с основами использования Docker, чтобы впоследствии прийти к пониманию, как использовать Twisted в Docker.

Обратите внимание, что технология Docker основана на Linux. Другие операционные системы тоже имеют аналогичные возможности, но сама технология Docker основана на использовании определенных возможностей ядра Linux. Docker для Windows может запускать «Контейнеры Windows», но мы не будем рассматривать эту тему в данной главе.

Реализации Docker для Mac и Docker для Windows используют виртуальную машину, работающую под управлением Linux и достаточно тесно интегрированную с ведущей операционной системой (OS X и Windows соответственно), чтобы обеспечить гладкое взаимодействие. Тем не менее важно помнить, что контейнеры Docker всегда работают под управлением ядра Linux, даже если запускаются на ноутбуке с Mac или Windows.

ВВЕДЕНИЕ В DOCKER

Вследствие новизны и популярности Docker под названием «Docker» часто понимают совершенно разные составляющие. Понять, *что такое Docker* на самом деле, непросто. Здесь мы пытаемся разложить термин «Docker» на отдельные понятия. Обратите внимание, что каждое из них часто упоминается под названием «Docker», как и все они вместе.

Контейнеры

Контейнеры – это процессы, действующие в более полной изоляции, чем традиционные процессы UNIX.

В контейнере действуют только процессы, запущенные корневым процессом контейнера, который имеет идентификатор процесса 1 (Process ID, PID) внутри контейнера. Обратите внимание, что иногда контейнер может использовать нумерацию процессов, общую с нумерацией процессов в хост-системе. Добиться этого можно, добавив в командную строку Docker аргумент `--pid host`.

Аналогично контейнеры имеют свой сетевой адрес. Это означает, что процессы внутри контейнера могут прослушивать любой порт без оглядки на хост-систему или другие выполняющиеся контейнеры. Тем не менее контейнер можно запустить со специальным аргументом `--net host`, чтобы заставить его использовать пространство сетевых имен хост-системы.

Наконец, каждый контейнер имеет свою файловую систему. Это означает, например, что мы можем устанавливать в разные контейнеры разные версии Python и даже конфликтующие пакеты Python. Организовать совместное использование файловой системы с хост-системой довольно сложно.

Однако в Docker поддерживается возможность «монтирования томов». Параметр, обеспечивающий монтирование тома, обеспечивает доступность («монтирует») каталога в хост-системе из контейнера. Синтаксис этого параметра прост: нужно через двоеточие указать путь к каталогу в хост-системе (слева) и путь к каталогу в контейнере (справа).

То есть если запустить Docker с параметром `--volume /:/from-host`, контейнеру станут доступны все файлы в хост-системе. Обратите внимание, что они будут доступны контейнеру в его каталоге `/from-host`.

Контейнеры изолированы точно в той степени, в какой желательно их изолировать. Это похоже на флаги системного вызова `clone`, указывающие, что будет совместно использоваться родительским и дочерним процессами: например, флаг `CLONE_FILES` обеспечивает совместный доступ к таблице дескрипторов файлов.

Образы контейнеров

Образ контейнера – это изолированный набор выполняющихся процессов. А чтобы запустить контейнер, нужен *образ контейнера*. В этом отношении образы контейнеров подобны выполняемым файлам на диске.

Образ контейнера имеет *многослойную* структуру. Каждый слой представляет файловую систему. Окончательная файловая система, которую увидит контейнер (часто называется объединенной файловой системой), образуется наложением всех слоев, причем более высокие слои перекрывают более низкие. Верхний слой может изменять, добавлять и даже «удалять» файлы, имеющиеся в предыдущем слое. Хотя сам нижний слой не подвергается влиянию слоев, лежащих выше, окончательная файловая система, видимая внутри контейнера, будет подвергнута таким влияниям.

Это важно, потому что удаление файлов в верхнем слое *не экономит пространство*. Например, если в первом слое имеется тарболл, который затем распаковывается, занятое им пространство не усекается. Верхние слои часто

включают команды, такие как `rm /path/to/file.tar.gz`. Подобные удаленные файлы не будут доступны в окончательной файловой системе, но это никак не повлияет на конечный размер образа контейнера – количество байтов, которое необходимо загрузить, чтобы запустить контейнер, – потому что тарболл все равно будет включен в образ.

Образы контейнеров *именуются* (или, если говорить точнее, снабжаются тегами) после их окончательного размещения. Обычная схема именования: `[optional host/][optional user/]name[:optional tag]`. В образах, которые никогда не покидают хост, где они были собраны, обычно отсутствуют части имени `host` и `user`, хотя бывают исключения.

Если часть имени `tag` не указана, по умолчанию используется тег `:latest`. Если отсутствует часть `host`, по умолчанию используется `docker.io`.

Обратите внимание, что один и тот же образ контейнера может иметь несколько тегов.

Образы контейнеров перемещаются между *реестрами* и *хостами*: они «помещаются» в реестры и «извлекаются» на хосты.

runс и containerd

Для запуска контейнера из образа используется специальная программа `runс` («`run container`» – «запустить контейнер»). Эта программа отвечает за настройку механизмов изоляции: она использует средства ядра Linux, такие как `cgroups` и пространства имен, чтобы правильно изолировать файловую систему, пространство имен процесса и сетевые адреса.

Обычно пользователи контейнера не вызывают команду `runс` непосредственно. Однако она вызывается механизмами стека Docker и почти всеми альтернативными стеками контейнеров, такими как Rocket.

Для управления запущенными контейнерами нужно знать, какие контейнеры работают и в каком состоянии находятся. Для этих нужд используется программа-демон `containerd`, которая запускает контейнеры из образов, вызывая `runс`.

Интересно отметить, что в предыдущих версиях Docker команда `runс` была встроена в `containerd`, поэтому кое-где до сих пор указывается, что контейнеры запускаются «демоном Docker».

Клиент

Команда `docker run` на самом деле не запускает контейнеры – она обращается к демону `containerd` и просит его запустить контейнер командой `runс`.

По умолчанию для взаимодействий с демоном используется сокет из домена UNIX. Эти сокеты являются специальным механизмом межпроцессных взаимодействий в операционных системах на основе UNIX. Их API напоминает интерфейс TCP-сокетов, но их можно использовать только для взаимодействий внутри одного компьютера, что позволяет ядру обеспечить более высокую скорость обмена. Вместо IP-адресов и портов сокеты из домена UNIX используют

пути к файлам. Это позволяет применять к ним обычную модель разрешений для файлов UNIX.

По умолчанию `docker` подключается к сокету `/var/run/docker.sock`. В зависимости от особенностей настройки Docker он может быть доступен для группы `docker` или `root`. Клиент Docker также может подключаться к серверу по протоколу TLS с взаимной аутентификацией с использованием сертификатов TLS.

То же верно для всех других подкоманд `docker`, таких как `build`, `images` и т. д. (Обратите внимание, что `docker login` является исключением из этого правила, но мы не будем вдаваться в исследование особенностей удаленного доступа к реестру.)

Поскольку команда `docker` в основном используется для отправки демону запросов на вызов удаленных процедур (Remote Procedure Calls, RPC), мы называем ее «клиентом».

Реестр

Образы контейнеров Docker хранятся в (обычно удаленном) *реестре*. Образы хранятся в реестре в виде метаданных и наборов *слоев*. Метаданные определяют порядок следования слоев, а также некоторые другие детали, касающиеся организации образа контейнера.

Обратите внимание, что благодаря такому способу хранения каждый слой будет храниться в единственном экземпляре. В практике нередко случаи, когда несколько образов совместно используют одни и те же слои, но все они будут использовать одну и ту же копию слоя.

Также отметьте, что в программное обеспечение встроен реестр по умолчанию `docker.io` – если реестр не указан явно, предполагается, что должен использоваться реестр по умолчанию, обычно с именем «DockerHub».

Это несколько иное применение слова «Docker», что лишний раз подчеркивает запутанность терминологии.

Сборка

Обычно образы создаются с помощью команды `docker build`. При этом используется файл конфигурации, который называют `Dockerfile`. Этот файл начинается со строки `FROM`, определяющей родительский образ. Если образ начинает собираться с нуля, то `Dockerfile` должен начинаться со строки `FROM scratch`, указывающей, что сборка начинается с образа `scratch` – чистого образа, не имеющего слоев. Однако это редкая ситуация.

Чаще образ начинает собираться, как минимум, с обычного дистрибутива Linux, доступного в стандартном реестре Docker – DockerHub. Например, в реестре доступны Debian, Ubuntu и CentOS.

Все последующие строки в `Dockerfile` описывают «этапы сборки». Каждый этап сборки создает слой, который затем кешируется. То есть при изменении `Dockerfile` будут выполняться только изменившиеся строки (и те, которые следуют за ними).

Вот пример короткого Dockerfile, который создает образ с демонстрационным веб-сервером Twisted.

```
FROM debian:latest
RUN python3 -m pip install --user Twisted
ENTRYPOINT ["python3", "-m", "twisted", "web"]
```

Этот файл не является демонстрацией передовых приемов, которые мы рассмотрим далее в главе, но показывает три важных раздела, которые присутствуют почти во всех файлах Dockerfile:

- строка FROM. Здесь мы требуем использовать в качестве основы последнюю (latest) версию Debian. Обратите внимание: отсутствие символов слеша означает, что этот базовый образ будет взят из «библиотеки» DockerHub – полуофициального набора базовых образов;
- строка RUN описывает команду, которая должна быть запущена внутри строящегося контейнера. В данном случае мы устанавливаем Twisted в домашний каталог пользователя;
- строка ENTRYPOINT описывает программу, которая должна быть запущена при запуске контейнера.

Многоступенчатая сборка

В описании выше не упомянута одна важная новая функция, добавленная в docker build в середине 2017 года: поддержка многоступенчатой сборки. Многоступенчатая сборка характеризуется наличием в Dockerfile более одной строки FROM.

В этом случае процесс сборки создает новый образ, и в конце все незавершенные образы отбрасываются. Однако во время выполнения сборки другие образы остаются доступными для команды Dockerfile – COPY.

Если использовать команду COPY --from=<image>, она скопирует файл не из контекста, а из предыдущего образа. Теоретически многоступенчатые сборки могут включать сколько угодно этапов, но на практике очень редко требуется больше двух. Последовательные образы нумеруются с нуля. Большинство «многоступенчатых» сборок в действительности являются «двухступенчатыми». На первом этапе собираются все артефакты из «толстого» образа, заполненного компиляторами и инструментами сборки. На втором этапе собираются все артефакты, полученные на первом этапе, и создается окончательный образ для распространения. Как следствие чаще всего используется форма инструкции COPY --from=0.

Эта возможность может пригодиться, когда требуется воссоздать сложное окружение сборки для создания некоторых продуктов для развертывания, и желательно не помещать это сложное окружение в окончательный контейнер времени выполнения, чтобы уменьшить размер, количество слоев и потенциальные риски безопасности.

Ниже приводится пример организации многоступенчатой сборки. Обратите внимание, что в этом случае окончательный результат предназначен не для

непосредственного использования, а для использования в других сборках. Это обычная схема, когда создается стандартная база общих элементов. Такой подход имеет несколько преимуществ, например экономию места в реестре и на сервере (если на одном сервере выполняется несколько разных образов, как это часто бывает). Еще одно преимущество: после исправления ошибок в базовых пакетах обновленные их версии требуется сохранить только в одном месте.

```
FROM python:3
RUN mkdir /wheels
RUN pip wheel --wheel-dir /wheels pyrsistent

FROM python:3-slim
COPY --from=0 /wheels /wheels
RUN pip install --no-index --find-links /wheel pyrsistent
```

И снова рассмотрим эти строки одну за другой:

```
FROM python:3
```

Базовый образ `python:3` – это еще один пример стандартного образа в «библиотеке DockerHub». Он включает Python 3, а также другие инструменты, необходимые для компиляции программ, хотя и без дополнительных зависимостей.

```
RUN mkdir /wheels
```

Создает каталог для компиляции пакетов `wheel`. Обратите внимание: этот этап не будет включен в конечный результат, поэтому можно не беспокоиться о создании дополнительных слоев. На самом деле дополнительные слои – это благо, они создают больше точек кеширования. Для данного примера это не так существенно, но во многих случаях, чтобы подготовить основу для сборки, требуется установить гораздо большее число зависимостей.

```
RUN pip wheel --wheel-dir /wheels pyrsistent
```

Команда `pip wheel` часто используется в многоступенчатых сборках. Она собирает пакеты `wheel` в соответствии с указанными требованиями и все необходимые зависимости. Если платформа совместима, будет использоваться двоичный формат `wheel` из PyPI, созданный для `manylinux`, но при желании это поведение можно изменить с помощью `pip wheel --no-binary :all:`.

```
FROM python:3-slim
```

Базовый образ `python:3-slim` похож на `python:3`, но не содержит сложного набора зависимостей времени сборки. Обратите внимание, что во многих дистрибутивах Python файлы `setup.py` автоматически обнаруживают отсутствие компиляторов или зависимостей и автоматически отключают сборку собственных модулей. Например, `pyrsistent` имеет оптимизированную реализацию на языке C, которую мы хотим получить в нашем образе. Поэтому сборка и установка `pyrsistent` были реализованы на предыдущем этапе.

```
COPY --from=0 /wheels /wheels
```

Эта команда скопирует пакет `pyrsistent` и все его зависимости, собранные на первом этапе (этап 0), в текущий этап. Вторая строка `FROM` указывает, что это многоступенчатая сборка, но эта строка `COPY` делает многоступенчатую сборку особенно полезной.

```
RUN pip install --no-index --find-links /wheel pyrsistent
```

Наконец, мы устанавливаем библиотеку в локальное окружение Python. Мы указываем параметры `--no-index` и `--find-links` в команде `pip`, чтобы она использовала пакеты `wheel`, полученные на первом этапе, и не пыталась извлечь их из каталога дистрибутивов PyPI.

PYTHON В DOCKER

Существует множество способов развертывания приложений Python в Docker, как и на любой платформе UNIX. Они не все равноценны – одни лучше, другие хуже. Далее мы рассмотрим варианты, которые обычно дают хороший результат.

Варианты развертывания

В полноценном окружении

Под развертыванием в «полноценном окружении» понимается использование специального интерпретатора Python для приложения. Этот интерпретатор может быть специально собран из исходного кода в ходе сборки Docker или до него, или он может быть получен из метадистрибутива, такого как `conda` или `nix`.

Установка специального интерпретатора Python имеет свои преимущества: мы можем настроить свои параметры сборки, закрепить версию интерпретатора и даже, в особо экстремальных случаях, применить свои исправления. Однако это также означает, что задача поддержки интерпретатора в актуальном состоянии теперь полностью ложится на нас.

Однако мы установим этот интерпретатор, он будет использоваться только для нашего приложения. Затем командой `pip install` установим пакеты (если интерпретатор устанавливается из метадистрибутива, например `conda` или `nix`, пакеты можно установить из этого метадистрибутива). Это особенно удобно, когда желательно использовать интерпретатор из дистрибутива `conda`, в котором присутствует большое число пакетов с реализацией численных методов.

Вот пример `Dockerfile`, который собирает интерпретатор Python, загружая необходимые пакеты.

```
FROM buildpack-deps:stretch
```

```
ENV PYTHON_VERSION 3.6.4
```

```
ENV PREFIX https://www.python.org/ftp/python
```

```
ENV LANG C.UTF-8

ENV GPG_KEY 0D96DF4D4110E5C43FBFB17F2D347EA6AA65421D

RUN apt-get update
RUN apt-get install -y --no-install-recommends \
    tcl \
    tk \
    dpkg-dev \
    tcl-dev \
    tk-dev

RUN wget -O python.tar.xz \
    "$PREFIX/${PYTHON_VERSION%%[a-z]*}/Python-${PYTHON_VERSION}.tar.xz"
RUN wget -O python.tar.xz.asc \
    "$PREFIX/${PYTHON_VERSION%%[a-z]*}/Python-${PYTHON_VERSION}.tar.xz.asc"
RUN export GNUPGHOME="$(mktemp -d)" && \
    gpg --keyserver ha.pool.sks-keyservers.net --recv-keys "$GPG_KEY" && \
    gpg --batch --verify python.tar.xz.asc python.tar.xz
RUN mkdir -p /usr/src/python
RUN tar -xJC /usr/src/python --strip-components=1 -f python.tar.xz

WORKDIR /usr/src/python

RUN gnuArch="$(dpkg-architecture --query DEB_BUILD_GNU_TYPE)"
RUN ./configure \
    --build="$gnuArch" \
    --enable-loadable-sqlite-extensions \
    --enable-shared \
    --prefix=/opt/custom-python/

RUN make -j
RUN make install
RUN ldconfig /opt/custom-python/lib
RUN /opt/custom-python/bin/python3 -m pip install twisted

FROM debian:stretch

COPY --from=0 /opt/custom-python /opt/custom-python
RUN apt-get update && \
    apt-get install libffi6 libssl1.1 && \
    ldconfig /opt/custom-python/lib
ENTRYPOINT ["/opt/custom-python/bin/python3", "-m", "twisted", "web"]
```

Сборка своего интерпретатора Python – весьма нетривиальная задача. Поэтому рассмотрим содержимое файла строку за строкой:

```
FROM buildpack-deps:stretch
```

buildpack-deps – это базовый образ для сборки. Так как мы собираемся использовать Debian «stretch» – последнюю стабильную версию Debian на момент написания этих строк, мы берем за основу пакет, совместимый с этой версией.

```
ENV PYTHON_VERSION 3.6.4
ENV PREFIX https://www.python.org/ftp/python
```

Эти настройки позволяют с легкостью изменить используемую версию Python – это важно для внедрения новых исправлений безопасности и ошибок. Чем проще будет организовано обновление Python, тем лучше.

```
ENV LANG C.UTF-8
```

Явное определение кодировки и языка необходимо, чтобы избежать странной ошибки, возникающей в процессе сборки Python. Файл `Dockerfile` отлично подходит для размещения подобных настроек и обходных решений, гарантирующих успешную сборку.

```
ENV GPG_KEY 0D96DF4D4110E5C43FBFB17F2D347EA6AA65421D
```

Это открытый ключ GnuPG, соответствующий закрытому ключу, которым подписан тарболл с исходными текстами Python. GnuPG (Gnu Privacy Guard) – это инструмент, использующий криптографию для поддержания безопасности. В данном случае ключ позволяет убедиться, что тарболл с исходными текстами не был подделан. Нелишне было бы усовершенствовать защиту и использовать несколько разных проверок подлинности источников. Такие файлы `Dockerfile` и аналогичные им часто используются в окружениях непрерывной интеграции, где они запускаются многократно и автоматически. Достаточно всего одной уязвимости, чтобы серьезно скомпрометировать инфраструктуру. Гарантируя прерывание сборки, если безопасность источника не подтвердилась, можно предотвратить дорогостоящую остановку производства.

Определение отпечатка ключа в `Dockerfile`, который, вероятно, хранится в системе управления версиями, является одним из способов повысить доверие к хранимому коду.

```
RUN apt-get update
RUN apt-get install -y --no-install-recommends \
    tcl \
    tk \
    dpkg-dev \
    tcl-dev \
    tk-dev
```

Для сборки пакета нам понадобятся некоторые дополнительные библиотеки. Здесь мы устанавливаем их.

```
RUN wget -O python.tar.xz \
    "$PREFIX/${PYTHON_VERSION%%[a-z]*}/Python-${PYTHON_VERSION}.tar.xz"
```

Далее загружается тарболл с исходными текстами Python. Переменные, что были определены выше, позволяют сохранить эту строку короткой и лаконичной. Кроме того, такая командная строка будет работать даже с версиями, не являющимися стабильными, такими как 3.6.1rc2, – это может пригодиться, если понадобится использовать этот `Dockerfile` с небольшими изменениями для проверки совместимости с версиями, кандидатами на выпуск.

```
RUN wget -O python.tar.xz.asc \
    "$PREFIX/${PYTHON_VERSION%%[a-z]*}/Python-${PYTHON_VERSION}.tar.xz.asc"
```

Отдельно загружается сигнатура открытого ключа. Даже притом что загрузка выполняется с веб-сайта, поддерживающего TLS, и используется схема https, а не http, проверка сигнатуры является хорошей дополнительной мерой защиты.

```
RUN export GNUPGHOME="$(mktemp -d)" && \
  gpg --keyserver ha.pool.sks-keyservers.net --recv-keys "$GPG_KEY" && \
  gpg --batch --verify python.tar.xz.asc python.tar.xz
```

Эта команда проверяет открытый ключ. Обратите внимание: эта команда не меняет локального состояния. Однако как любая команда, потерпевшая неудачу, она остановит процесс `docker build`, если проверка ключа не увенчается успехом.

```
RUN mkdir -p /usr/src/python
```

Создается каталог для распаковки исходных текстов. Обратите внимание, что мы выполняем многоступенчатую сборку, поэтому можно не беспокоиться об очистке данного каталога – в конце сборки будет очищен весь контейнер!

```
RUN tar -xJC /usr/src/python --strip-components=1 -f python.tar.xz
```

Тарболл с исходными текстами Python распаковывается во вновь созданный каталог.

```
WORKDIR /usr/src/python
```

Выполняется переход в каталог с исходными текстами. Эта команда подготавливает почву для последующих команд сборки, которые получаются короче и проще для понимания, если выполняются внутри этого каталога.

```
RUN gnuArch="$(dpkg-architecture --query DEB_BUILD_GNU_TYPE)" && \
  ./configure \
  --build="$gnuArch" \
  --enable-loadable-sqlite-extensions \
  --enable-shared \
  --prefix=/opt/custom-python/
```

Запускается сценарий `./configure` с настроенными параметрами сборки. Ключ `--prefix=/opt/custom-python/` гарантирует, что сборка будет выполнена в требуемый каталог. Другие ключи обеспечивают правильную сборку Python:

- архитектура определяется с использованием `dpkg-Architecture` и явно передается в сценарий `configure`. Этот подход надежнее, чем инструмент автоматического определения в сценарии `configure`;
- добавляется модуль `sqlite`. Поскольку этот модуль является встроенным, многие сторонние модули будут зависеть от него, не объявляя эту зависимость явно, поэтому важно гарантировать, что он войдет в состав собранной версии Python;
- разрешается использовать динамические библиотеки. В данном примере в этом нет необходимости, но вообще это решение обеспечит возможность встраивания Python.

```
RUN make -j
```

Определить точное число процессоров непросто. В этом примере мы просто запускаем утилиту `make` и сообщаем (передавая флаг `-j`), что она может запустить максимально возможное число процессов сборки. Но вообще рекомендуется устанавливать разумный предел параллелизма, передавая с флагом `-j` максимально возможное число процессов, например `-j 4`.

```
RUN make install
```

Копирует скомпилированные файлы с правильными привилегиями в каталог установки.

```
RUN ldconfig /opt/custom-python/lib
```

Добавляет каталог в путь поиска библиотек, иначе Python (если собран с динамическими библиотеками) не будет работать.

```
RUN /opt/custom-python/bin/python3 -m pip install twisted
```

Устанавливает Twisted. Среди многих других преимуществ Twisted этот фреймворк содержит свой простой веб-сервер, который удобно использовать для демонстраций.

```
FROM debian:stretch
```

Сборка образа для эксплуатации начинается с определения подходящего минимального дистрибутива Debian, с учетом версии, использовавшейся для сборки.

```
COPY --from=0 /opt/custom-python /opt/custom-python
```

Копирует окружение целиком, включая установленные сторонние библиотеки: в данном случае фреймворк Twisted и его зависимости.

```
RUN apt-get update && \
    apt-get install libffi6 libssl1.1 && \
    ldconfig /opt/custom-python/lib
```

Устанавливаются необходимые библиотеки, и выполняется команда `ldconfig` в образе, предназначенном для эксплуатации.

```
ENTRYPOINT ["/opt/custom-python/bin/python3", "-m", "twisted", "web"]
```

Определяется точка входа для запуска демонстрационного веб-сервера, встроенного в Twisted. Если создать и запустить этот образ, Docker автоматически запустит веб-сервер, а если экспортировать порт, к этому веб-серверу можно будет обратиться из браузера.

В виртуальном окружении

Альтернативой полноценному окружению является «облегченное» окружение или, как его еще называют, виртуальное окружение. В Python 2.7 виртуальные окружения создаются с помощью пакета `virtualenv`. Установить его можно с помощью `pip`, но возникает одна проблема: обычно виртуальные окружения

создаются с целью исключить изменение реального окружения, а установка `virtualenv` сама по себе изменяет реальное окружение. Есть другой путь – получить `virtualenv` так же, как мы получили Python. И еще один: установить его с помощью команды

```
pip install --user virtualenv
```

Она поместит пакет в каталог пользователя (в Docker, обычно в каталог `/root`). Это часто означает, что `virtualenv` оказывается вне пути поиска программ в командной оболочке, но так как пакет находится в пути поиска Python

```
python -m virtualenv <directory>
```

его можно использовать для создания виртуальных окружений.

В Python 3.x эти проблемы отсутствуют: команда `python -m venv` предлагает лучший способ создания виртуального окружения для Python 3.x. Обратите внимание, что часть документации не была обновлена, и на самом деле `virtualenv` работает в Python 3.x, что затрудняет своевременное обновление всех этих инструментов. Однако наличие встроенного модуля `venv` значительно упрощает организацию виртуальных окружений.

Одно из преимуществ установки кода в виртуальное окружение – в каталог виртуального окружения можно установить ровно то, что необходимо этому окружению, кроме самого интерпретатора, разумеется. Это преимущество особенно ценно для сборки образов Docker.

Объединив все описанные идеи, мы приходим к следующему файлу `Dockerfile`:

```
FROM python:3
```

Поскольку мы собираемся создать виртуальное окружение, у нас уже должно иметься полноценное окружение. Самый простой путь добиться этого – начать с образа контейнера `python`.

```
RUN python -m venv /opt/venv-python
```

Создаем виртуальное окружение в `/opt/venv-python`.

```
RUN /opt/venv-python/bin/pip install Twisted
```

Устанавливаем Twisted в него. Обратите внимание, что установка Twisted подразумевает установку нескольких пакетов с расширениями на языке C, а для этого необходим компилятор C. Контейнер `python:3` имеет все необходимое для сборки расширений на C.

```
FROM python:3-slim
```

Образ контейнера `python:3-slim` не имеет инструментов сборки. А поскольку именно этот контейнер мы будем распространять, это означает, что в промышленном окружении у нас не будет компилятора C.

```
COPY --from=0 /opt/venv-python /opt/venv-python
```

Копируем виртуальное окружение. Как вы помните, виртуальное окружение включает несколько предопределенных путей. Поэтому копирование выполняется с сохранением этих путей.

```
ENTRYPOINT ["/opt/venv-python/bin/python", "-m", "twisted", "web"]
```

Точка входа практически идентична той, что вы видели выше. Единственное отличие – путь, который на этот раз ведет в виртуальное окружение.

В формате Pex

Наибольшую автономность пакетам обеспечивает формат Pex (Python EXecutable) – формат выполняемых файлов для Python, – разработанный в Twitter. В нем используются возможности ОС UNIX, Python и архиватора Zip для создания единого файла, содержащего весь код приложения и сторонние зависимости.

Файл в формате Pex должен быть отмечен как выполняемый, например командой `chmod +x`, и включать строку определения интерпретатора для его выполнения (`#!/`), которая вызывает интерпретатор Python. Благодаря уникальной особенности архиватора Zip, согласно которой параметры архива определяются по последним, а не по первым байтам в файле, остальная часть файла – это обычный архив Zip.

Когда интерпретатор Python получает архив Zip, возможно с произвольным содержимым в начале, он интерпретирует его как дополнение к `sys.path` и запускает файл `__main__.py`, находящийся в архиве. Для файлов в формате Pex дополнительно генерируется файл `__main__.py`, который начинает выполнение с точки входа или модуля Python, указанных в параметрах, подставленных построителем файлов Pex.

Файл в формате Pex можно создать командой `pex` (устанавливается командой `pip install pex`), с помощью библиотеки `pex` для Python или более современных инструментов, таких как Pants, Bazel и Buck.

```
FROM python:3
RUN python -m venv /opt/venv-python
```

Создаем виртуальное окружение. Это виртуальное окружение *не будет разворачиваться* в окончательном образе, оно необходимо только для создания файла в формате Pex.

```
RUN /opt/venv-python/bin/pip install pex
```

Устанавливаем утилиту `pex`.

```
RUN mkdir /opt/wheels /opt/pex
```

Создаем два каталога для разных продуктов.

```
RUN /opt/venv-python/bin/pip wheel --wheel-dir /opt/wheels Twisted
```

Используем `pip` для сборки пакетов `wheel`. Утилита `pip` имеет замечательный алгоритм разрешения зависимостей, и мы используем его здесь. Объективно

он не лучше аналогичного алгоритма в `pep`, но имеет более давнюю историю развития. То есть если пакеты столкнутся с проблемой в процессе разрешения зависимостей, они смогут передать утилите `pip` дополнительные подсказки, которые помогут правильно установить их. Утилита `pep`, используемая гораздо реже, не дает таких гарантий.

```
RUN /opt/venv-python/bin/pep --find-links /opt/wheels --no-index \
    Twisted -m twisted -o /opt/pep/twisted.pep
```

Создаем файл `Peep`. Обратите внимание, что здесь мы требуем от `pep` игнорировать каталог `PyPI` и собирать пакеты только из каталога, куда `pip` поместит все созданные пакеты `wheel`. Файл `Peep` настраивается на запуск интерпретатора Python с флагами `-m twisted`, сохраняем его в `/opt/pep`. Расширение файла не является обязательным, но оно пригодится, когда позднее мы вернемся к этому образу контейнера и попробуем вспомнить, как все работает.

```
FROM python:3-slim
```

И снова мы предотвращаем включение инструментов сборки в промышленное окружение, определив образ без этих инструментов для второго этапа.

```
COPY --from=0 /opt/pep /opt/pep
```

Копируем каталог, содержащий единственный файл. Также обратите внимание, что на этот раз файл является перемещаемым: его можно (хотя здесь эта возможность не используется) скопировать в любой другой каталог.

```
ENTRYPOINT ["/opt/pep/twisted.pep", "web"]
```

Часть логики, которая в предыдущих примерах находится в `ENTRYPOINT` (в частности, вызов интерпретатора `python -m twisted`), теперь встроена в файл `Peep`, благодаря чему определение `ENTRYPOINT` получилось короче.

Варианты сборки

Независимо от способа запуска Python сборка контейнера Docker тоже имеет несколько вариантов.

Один большой образ

Один из возможных вариантов – полностью отказаться от многоступенчатой сборки и создать контейнер со всеми инструментами, необходимыми для создания окружения. Такие контейнеры часто получаются большими и многослойными.

Этот подход, безусловно, прост и понятен, легко поддается отладке, но у него есть свои недостатки. Первый – размер контейнера. Также большое количество слоев замедляет развертывание контейнера. Наконец, размещение большого количества пакетов в контейнере может привести к появлению большого числа потенциальных целей для злоумышленников, чем хотелось бы.

Копирование пакетов wheel между этапами

Другой вариант – собрать все пакеты wheel в рамках отдельного этапа сборки, включая любые двоичные пакеты wheel, а затем скопировать их в образ с промышленным окружением. В этом случае необходим дополнительный этап создания промышленного окружения, в котором будут иметься средства для создания виртуального окружения и куда будут установлены все требуемые пакеты wheel. Однако с появлением `venv` – встроенного модуля в Python 3 – это больше не является проблемой.

Есть еще две проблемы: пакеты wheel остаются после установки, потому что невозможно удалить файл после переключения слоев; из-за этого часто создаются дополнительные слои (хотя путем переупорядочения строк и использования длинных команд с обратным слешем этого иногда можно избежать).

Копирование окружения между этапами

Еще одним вариантом сборки является копирование окружения (полноценного или виртуального) из этапа сборки в промышленный этап. Преимущество этого подхода – в его скорости и простоте, а недостаток – в отсутствии проверки совместимости, зависимостей и местоположения. Тем не менее основные проблемы, возникающие при таком подходе, относительно легко обнаруживаются с помощью тестов.

Копирование файлов Рех между этапами

Наконец, если на этапе сборки создать исполняемый файл Рех, его можно просто скопировать. Все зависимости будут загружены и добавлены в файл Рех при сборке. Это обеспечит надежную проверку, поэтому для проверки достаточно лишь запустить контейнер.

Кроме того, этот файл допускает возможность перемещения, поэтому не имеет значения, откуда он скопирован и куда. Файлы Рех и Docker часто образуют хорошую компанию. Однако из-за ограничений, характерных для Рех (например, плохая поддержка двоичных пакетов wheel перед сборкой или плохая поддержка PyPy), иногда файл получается незапускаемым.

Автоматизация с использованием Dockerpy

Пакет `dockerpy` позволяет автоматизировать этапы создания образов контейнеров Docker с Python. Обычно для запуска контейнеров используется специализированный управляющий фреймворк, однако этот пакет часто оказывает ценную помощь в создании и тестировании контейнеров. Библиотека `dockerpy` позволяет настроить контекст для передачи демону Docker – используя модуль Python `tarfile`, можно создать именно тот контекст, который необходим.

TWISTED В DOCKER

ENTRYPOINT и PID 1

Процесс, который запускается командой в инструкции `ENTRYPOINT`, получает числовой идентификатор процесса 1. Процесс с этим идентификатором несет особую ответственность с ОС Linux. Когда процесс завершается, его родителем временно становится процесс с идентификатором 1. То есть когда завершается дочерний процесс, процесс с PID 1 должен «утилизировать его» – дожидаться, пока тот вернет код завершения, и удалить запись о нем из таблицы процессов.

Это довольно обременительная ответственность, и многие программы не предполагают ее. Если программа не предусматривает утилизацию дочерних процессов, может наступить переполнение таблицы процессов. В лучшем случае это приведет к аварийной остановке контейнера. В худшем случае, когда пределы для процесса не были тщательно установлены, это может привести к сбою всей машины (виртуальной или физической), где выполняется контейнер.

К счастью, *любая* программа на основе Twisted готова получить PID 1. Это связано с тем, что инфраструктура процессов Twisted автоматически готова управлять ожидаемыми и неожиданными потомками. То есть если контейнер создается для запуска приложений WSGI, приложений Klein или мастера Buildbot, их смело можно назначать точкой входа.

По этой причине, если понадобится запустить какой-либо пользовательский код, подумайте о его реализации в виде `tap`-плагины. В этом случае точкой входа можно назначить Twisted.

Пользовательские плагины

Разрабатывая приложения Twisted для запуска в Docker, мы почти всегда реализуем его в виде плагина. Это позволяет определить простую инструкцию `ENTRYPOINT`:

```
["/path/to/python", "-m", "twisted", "custom_plugin"]
```

Проще говоря, плагин может получать любые аргументы, передаваемые команде `docker run`, потому что эти аргументы напрямую добавляются в аргументы `ENTRYPOINT`. Кроме того, плагин может напрямую читать любые переменные окружения, переданные в `docker run` через `--env`.

Функция `makeService` в плагине возвращает работающую службу. Обратите внимание, что в этой функции плагин может выполнить любую инициализацию – цикл обработки событий в этот момент еще не запущен.

NColony

Иногда требуется запустить внутри контейнера Docker несколько процессов. Это может быть служебный процесс для удаления временных файлов или,

возможно, несколько одинаковых процессов для организации параллельных вычислений на нескольких процессорах. В таких случаях желательно использовать ведущий процесс – супервизор, – запускающий рабочие процессы и осуществляющий их мониторинг и повторный запуск при необходимости.

Роль такого супервизора может осуществлять утилита NColony, входящая в состав фреймворка Twisted. Это тонкая обертка вокруг `twisted.runner.gprocmon`, поддерживающая несколько гибких параметров настройки. NColony использует конфигурацию в формате каталога с файлами JSON, описывающими процессы.

Эти файлы можно создавать вручную, открывая их в редакторе и записывая в них определения в формате JSON. Однако вместе с NColony поставляется утилита командной строки `python -m ncolony ctl`, упрощающая создание таких файлов, а также библиотека для Python – `ncolony.ctllib`.

Одно из преимуществ каталога с файлами заключается в том, что эта модель прекрасно взаимодействует с моделью слоев в контейнерах Docker. Локальный базовый контейнер может иметь `ENTRYPOINT ["python", "-m", "twisted", "ncolony", ...]` и даже несколько базовых процессов в своем конфигурационном каталоге, обычно `/var/run/ncolony/config/`. А каждый конкретный контейнер может добавлять свои файлы, созданные на этапе сборки с использованием, например, `python -m ncolony ctl`. Получившийся контейнер может запускаться и как побочный, и как основной процесс.

Вот пример, подробно иллюстрирующий многое из того, о чем говорилось в этой главе:

```
FROM python:3
RUN python3 -m venv /application/env
RUN /application/env/bin/pip install ncolony
RUN mkdir /application/config /application/messages
RUN /application/env/bin/python -m ncolony \
    --config /application/config \
    --messages /application/messages \
    ctl \
    --cmd /application/env/bin/python \
    --arg=-m \
    --arg=twisted \
    --arg=web

FROM python:3-slim
COPY --from=0 /application/ /application/
ENTRYPOINT [" /application/env/bin/python", \
    "-m", \
    "twisted", \
    "ncolony", \
    "--config", "/application/config", \
    "--messages", "/application/messages"]
```

Разберем данный пример строку за строкой.

```
FROM python:3
```

Один из способов воссоздать окружение для программ на Python – использовать официальный образ Docker. Он основан на дистрибутиве Debian и содержит Python, а также все инструменты, необходимые для сборки модулей и расширений Python.

```
RUN python3 -m venv /application/env
```

Создаем виртуальное окружение в `/application/env`. Как упоминалось выше, Python 3 имеет встроенную поддержку виртуальных окружений, и мы в полной мере используем ее.

```
RUN /application/env/bin/pip install ncolony
```

Для большей эффективности лучше скопировать файл зависимостей – в идеале включающий также контрольные суммы (хеши) – и передать его команде `pip install`. Однако намного нагляднее, когда используется имя пакета непосредственно.

```
RUN mkdir /application/config /application/messages
```

Для нормальной работы NColony требуется два каталога: один для конфигурационных файлов и другой для сообщений. Мы создаем оба каталога в `/application`. Конфигурация – это набор процессов, которые требуется запустить, и их параметры. Сообщения – это запросы, обычно для повторного запуска одного или нескольких процессов.

```
RUN /application/env/bin/python -m ncolony \
```

Запускаем утилиту NColony, которую установили в виртуальное окружение `/application/env`.

```
--config /application/config \  
--messages /application/messages \
```

Передаем параметры утилите NColony. В этом случае каталог `messages` не используется, но вообще желательно передать всем командам оба каталога.

```
ctl \
```

`ctl` – это подкоманда утилиты NColony, которая управляется конфигурацией.

```
--cmd /application/env/bin/python \
```

Запускаем тот же интерпретатор Python, что использовался для запуска NColony. Обратите внимание, что в общем случае в этом нет необходимости, но было бы странно писать код, использующий совершенно другой интерпретатор.

```
--arg=-m \  
--arg=twisted \
```

Процесс, контролирующий работу NColony, не обязательно должен быть процессом Twisted, но в нашем случае эту роль будет играть именно процесс Twisted – фактически еще один `tap`-плагин.

```
--arg=web
```

При запуске без аргументов `tap`-плагин отображает демонстрационное веб-приложение. Это может пригодиться для быстрой демонстрации и проверки, как в этом случае:

```
FROM python:3-slim
```

Вторая строка `FROM` создает образ `Docker` для развертывания в промышленном окружении. Обратите внимание: все, что было создано до этого момента, *будет отброшено* после завершения сборки. Единственная причина выполнения всех предыдущих шагов – создание всего, что будет скопировано из этого эфемерного этапа. Данный исходный образ – это минимальный дистрибутив `Debian`, плюс установленный `Python 3`.

```
COPY --from=0 /application/ /application/
```

Копируется все содержимое каталога `application`. Поскольку этот каталог включает виртуальное окружение и конфигурацию `NColony`, он – единственный, который мы должны скопировать. Простота данной строки объясняет ценность всей работы, проделанной для настройки этого каталога.

```
ENTRYPOINT ["/application/env/bin/python", \ "-m", \
            "twisted", \ "ncolony", \
            "--config", "/application/config", \
            "--messages", "/application/messages"]
```

В заключение определяется точка входа. Так как сама утилита `NColony` является `tap`-плагином, мы снова запускаем команду `python -m twisted <plugin>`.

В этом примере мы могли бы запустить веб-сервер непосредственно, указав его в точке входа. Однако мы предпочли привести более практичный пример, проясняющий основную механику запуска `NColony` в `Docker`.

Итоги

Технологии `Docker`, `Python` и `Twisted` прекрасно дополняют друг друга. `Docker`, с многоступенчатыми сборками и реестрами, предоставляет стандартный способ создания и упаковки приложений. Фреймворк `Twisted`, с его примитивами управления процессами, предлагает возможность создания процесса с `PID 1` в `Docker`, который либо выполняет полезную работу сам по себе, как, например, веб-сервер, либо обеспечивает мощную основу (в лице `NColony`), которая хорошо укладывается в модель `Docker`.

`Docker` – это практичный способ создания, упаковки и запуска приложений `Twisted`, а `Twisted` – замечательный инструмент для запуска приложений на `Python` внутри `Docker`.

Глава 5

Использование Twisted в роли сервера WSGI

ВВЕДЕНИЕ В WSGI

WSGI – Web Standard Gateway Interface (стандартный интерфейс веб-шлюза) – это стандарт, регламентирующий взаимодействие программ на Python с веб-сервером. Он основан на стандарте CGI – Common Gateway Interface (обобщенный интерфейс шлюза), – регламентирующем взаимодействия между веб-сервером и сценариями. С ростом нагрузок возникла необходимость иметь постоянный процесс Python внутри веб-сервера. Первоначально каждый сервер имел свой уникальный способ запуска приложений на Python. Это означало, что для каждого приложения требовалось выбрать конкретную реализацию веб-сервера, которую потом нельзя было сменить. WSGI разрабатывался как низкоуровневый стандарт для веб-приложений, написанных на Python, регламентирующий взаимодействия с веб-серверами, которые могут запускать внутренний интерпретатор Python.

Стандарт WSGI определяет интерфейс между WSGI-совместимым *веб-приложением* и WSGI-совместимым *веб-сервером*.

Фреймворк Twisted имеет свой веб-сервер, реализующий свой уникальный веб-API и поддерживающий стандарт WSGI. Благодаря поддержке стандарта WSGI он может запускать любые веб-приложения на Python, которые также поддерживают стандарт WSGI.

Обычно веб-приложения на Python не взаимодействуют напрямую с WSGI API. Для этой цели используются специализированные веб-фреймворки, такие как Django, Flask или Pyramid, которые берут на себя всю ответственность за взаимодействие с WSGI API и предлагают высокоуровневый интерфейс для использования в веб-приложениях. Однако эти интерфейсы весьма специфичны для веб-фреймворков, и, как следствие, приложение будет нелегко перенести, скажем, с Django на Pyramid.

Выбор веб-фреймворка в этой ситуации подобен выбору языка программирования, а выбор веб-сервера – выбору операционной системы. Обычно при

переходе между операционными системами мы ожидаем, что большую часть кода удастся перенести без изменений (высокая переносимость), но не ожидаем того же при смене языка программирования.

С точки зрения веб-серверов поддержка стандарта WSGI означает независимость от используемого веб-фреймворка – приложение на Pyramid запускается точно так же, как приложение на Flask. В свою очередь, поддержка стандарта WSGI с точки зрения веб-фреймворков означает независимость от используемого веб-сервера – они действуют под управлением Apache точно так же, как под управлением uwsgi.

Стандарт WSGI родился не в вакууме. На момент его разработки уже имелось множество серверов и веб-фреймворков на Python. Поэтому WSGI проектировался с целью упростить его реализацию по обе стороны – на стороне серверов и на стороне веб-фреймворков. Его сходство с CGI является результатом этого стремления. Многие из этих фреймворков уже поддерживали CGI, и добавление поддержки WSGI не потребовало больших усилий.

WSGI был разработан довольно давно, еще в 2003 году. Поэтому названия многих фреймворков, упоминающихся в нем, например Quixote и Webware, теперь воспринимаются как отголоски давних экспериментов с веб-фреймворками. В то время доминирующие позиции занимал единственный веб-сервер – Apache, – популярность которого с той поры серьезно снизилась.

Но, несмотря на смену популярных фреймворков и серверов, стандарт WSGI выдержал проверку временем.

Стандарт WSGI определяет довольно сложный API, абстрагирующий HTTP. И современным веб-приложениям необходим доступ к большей части функций этого сложного API. Описание стандарта охватывает два документа и иногда может показаться ошеломляюще сложным.

В этой главе мы подробно рассмотрим стандарт WSGI и составляющие его части.

PEP

Все основные улучшения, добавляемые в Python, проходят процесс PEP (Python Enhancement Proposal – предложения по улучшению Python). Поддержка WSGI как важная особенность была первоначально описана в документе PEP 0333, созданном в декабре 2003 года и окончательно завершенном в августе 2004 года.

Это предложение PEP сохраняет свою актуальность для Python 2.x. Но в сентябре 2010 года было создано и в октябре 2010 года завершено предложение PEP 3333, описывающее реализацию WSGI для Python 2.x и Python 3.x.

Этот документ добавляет довольно незначительные изменения в PEP 0333. Основные отличия касаются правильной реализации WSGI в Python 2.x и Python 3.x. Чтобы понять, зачем это было необходимо, важно понимать разницу между Python 2.x и Python 3.x.

Одним из главных отличий между Python 2.x и Python 3.x стала обработка Юникода. В частности, серьезные изменения претерпели типы `bytes`, `string`

и `unicode`. Стандарт WSGI, как непосредственно (в конечном счете) связанный с передачей байтов через TCP-соединения, обязан был уточнить, какие типы Python 3.x и где используются.

Подробное объяснение этих изменений выходит за рамки нашей книги, тем не менее мы должны осветить некоторые моменты. Обе версии, Python 2.7+ и Python 3.x, имеют тип `bytes`, представляющий последовательность байтов, и тип `unicode`, представляющий последовательность кодовых пунктов Юникода. Тип `string`, однако, в Python 2.7+ эквивалентен типу `bytes`, а в Python 3.x – типу `unicode`.

Кодировка – это (возможно, частичная) карта соответствий между байтами и Юникодом. ASCII – одна из таких кодировок. Она отображает байты со значениями меньше 128 в кодовые пункты Юникода с теми же значениями, а все остальные байты объявляет недействительными. Latin-1 (или ISO-8859-1) – еще одна кодировка, которая отображает все байты в кодовые пункты Юникода с теми же значениями, а если кодового пункта Юникода с таким значением нет, соответствующий байт объявляется недействительным.

Протокол HTTP, лежащий в основе Всемирной паутины, делит любую веб-страницу на заголовок и тело; и если тело является текстовым, заголовок должен указывать кодировку этого текста.

Однако остается проблема кодирования самих заголовков: в PEP 3333 указывается, что заголовки должны быть представлены в кодировке Latin-1 (также известной как ISO-8859-1), тогда как фреймворк Twisted представляет их в кодировке UTF-8. Самое безопасное на практике – обеспечить соответствие заголовков общему подмножеству UTF-8 и Latin-1: кодировке ASCII. В этом случае, независимо от используемой цепочки кодирования/декодирования, заголовки останутся без изменений.

Согласно PEP 3333, заголовки должны определяться значениями встроенного типа `string`, то есть типа `bytes` в Python 2.x и `unicode` и Python 3.x, при этом содержимое всегда должно быть значением типа `bytes`.

PEP 3333, как и PEP 0333, также описывает идею промежуточного программного обеспечения WSGI (middleware), которое с точки зрения приложения выглядит как сервер, а с точки зрения сервера – как приложение. Однако, несмотря на наличие нескольких реализаций промежуточного программного обеспечения WSGI, некоторые популярные фреймворки – в частности Django и Pyramid – имеют свои собственные реализации промежуточного программного обеспечения. Flask, в свою очередь, использует промежуточное программное обеспечение WSGI.

Простой пример

Простейшее WSGI-приложение действительно выглядит очень просто:

```
def application(
    environment,
    start_response):
```

```
start_response('200 OK', [('Content-Type', 'text/html')])
return [b'hello world']
```

Рассмотрим этот пример строку за строкой и разберем назначение каждой из трех частей, которые должно иметь любое WSGI-приложение:

```
def application(
```

В языке Python определения функций преследуют две цели:

- создать *объект функции*;
- присвоить этому объекту имя.

Это определение функции, например, создает объект функции и присваивает ему имя `application`.

Это означает, что теперь имя `application` ссылается на вызываемый объект. То есть, согласно PEP 3333, приложения WSGI являются вызываемыми объектами.

```
environment,
```

Первый параметр – это так называемое «окружение». Имя `environment` восходит к истокам WSGI, когда этот стандарт еще был быстрым приближением стандарта CGI.

Стандарт CGI описывает, как веб-серверы должны выполнять сценарии. Часть этого стандарта определяет переменные окружения, доступные сценариям. В действительности большая часть информации о веб-запросе в CGI доступна в виде переменных окружения. Стандарт WSGI заимствовал из CGI те же имена переменных и понятие окружения и дал первому параметру приложения WSGI соответствующее имя.

Параметр `environment` – это словарь Python, отображающий конкретные имена в данные о веб-запросе. В приведенном примере приложения этот параметр игнорируется, поскольку оно использует только константы. Если бы этим наши возможности ограничивались, мы имели бы только статические HTML-страницы. Но на практике большинство приложений так или иначе используют пользовательский ввод.

```
start_response):
```

Второй параметр, с именем `start_response`, играет важную и часто не до конца понимаемую роль. Это вызываемый объект, принимающий два аргумента: код HTTP-ответа и HTTP-заголовки.

```
start_response('200 OK', [('Content-Type', 'text/html')]))
```

Первое, что делает наше приложение, – вызывает функцию `start_response`. Первый аргумент – `'200 OK'` – это обычный HTTP-ответ, сообщающий об успехе. Второй аргумент – это список заголовков. В данном случае клиенту отправляется только один заголовок – `Content-Type`. Он указывает, что должен интерпретироваться браузером как разметка HTML.

```
return [b'hello world']
```

Следующая строка возвращает список строк байтов. Поскольку мы явно не указали кодировку в заголовке Content-Type, браузер будет использовать свою настройку кодировки по умолчанию. В данном случае это достаточно безопасно – механизм определения кодировки в современных браузерах всегда будет правильно работать с байтами в диапазоне ASCII.

В общем случае, чтобы не полагаться на интеллектуальные способности браузеров, предпочтительнее использовать кодировку UTF-8 и явно указывать ее в Content-Type.

Это важно, потому что разметка HTML *всегда* определяется в терминах Юникода. Браузер переведет этот ответ в строку `u'hello world'` и покажет сообщение пользователю.

В оставшейся части главы мы будем предполагать, что этот код находится в файле `wsgi_hello.py`.

Базовая реализация

В PEP 333 (и 3333) было решено, что нет необходимости внедрять поддержку WSGI в ядро Python, но опыт доказал ошибочность этого решения. Модуль `wsgiref` реализует простой веб-сервер, поддерживающий приложения WSGI.

Следующая команда будет работать в любой `bash`-подобной оболочке, где кавычки позволяют передавать многострочный текст. Эту разбивку мы сделали для удобства – заменив первые два символа перевода строки точками с запятой и просто удалив остальные, вы получите полностью переносимую команду, которую, однако, будет сложнее читать.

```
python -c '  
from wsgiref import simple_server  
import wsgi_hello  
simple_server.make_server(  
    "127.0.0.1",  
    8000,  
    wsgi_hello.application  
) .serve_forever()  
'
```

Рассмотрим каждую строку из этого примера в отдельности:

```
python -c '
```

Интерпретатор Python имеет параметр `-c`, который интерпретирует следующий за ним аргумент как код на языке Python и выполняет его. Это удобный способ выполнить короткую программу, не сохраняя ее в отдельном файле.

```
from wsgiref import simple_server
```

Импортирует модуль `wsgiref.simple_server`. Этот модуль реализует однопоточный синхронный веб-сервер. Этот сервер не предназначен для использования в промышленном окружении, но иногда он очень удобен для простых демонстраций.

```
import wsgi_hello
```

Здесь важно вспомнить, что код, показанный выше, хранится в файле с именем `wsgi_hello.py`. Также важно, чтобы:

- этот файл находился в текущем рабочем каталоге;
- при использовании - с текущий рабочий каталог находился в пути поиска модулей Python.

Важность всего этого станет очевидна, когда мы обсудим тонкости поиска кода приложения WSGI.

```
simple_server.make_server(
```

Это главная функция в модуле `simple_server` – она создает экземпляр простого сервера.

```
"127.0.0.1",
```

Многие примеры (в том числе и в официальной документации) используют пустую строку `""`. Она заставляет сервер WSGI привязаться к адресу `0.0.0.0`, обозначающему так называемый «любой» интерфейс. Обратите внимание, что `wsgiref` не предназначен для промышленной эксплуатации – мы используем его только для тестирования и демонстрации примеров, и если привязать его к интерфейсу `0.0.0.0`, к нему смогут подключиться посторонние клиенты.

Чтобы избежать этой опасности, мы привязываем сервер к локальному интерфейсу `"127.0.0.1"`. В этом случае к серверу смогут подключиться только программы, выполняющиеся на этом же компьютере. Это удобно, так как мы легко сможем протестировать работу сервера с помощью браузера, запустив его на этом же компьютере.

```
8000,
```

Обычно для веб-серверов используется порт с номером 80, как определено стандартом IANA. Однако в системах UNIX порты с номерами ниже 1024 могут назначаться службам только при наличии привилегий суперпользователя (`root`). Это предотвращает «захват» системных портов непривилегированными пользователями. В настоящее время, несмотря на снижение важности конкретной модели потоков выполнения, накладывающей это ограничение, и когда непривилегированные пользователи редко имеют право входа в систему, где выполняется веб-сервер, это правило все еще используется для уменьшения угрозы и, что особенно важно, продолжает применяться в современных UNIX-подобных системах, таких как Linux.

В веб-разработке стало традицией присваивать веб-службам порты, «похожие» на порт 80, такие как 8888 или 8080.

```
wsgi_hello.application
```

Это фактическое приложение WSGI. Как уже отмечалось, приложение WSGI – это вызываемый объект Python.

```
).serve_forever()
```

После создания сервера запускается его бесконечный цикл приема и обработки запросов.

Это, пожалуй, самый простой способ быстро протестировать приложение WSGI, не имеющее никаких зависимостей, кроме стандартной библиотеки Python.

Пример WebOb

Пакет WebOb может служить примером низкоуровневого веб-фреймворка. Обычно он не используется непосредственно, хотя это возможно.

```
import webob

def application(environment, start_response):
    request = webob.Request(environment)
    response = webob.Response(
        text='Hello world!')
    return response(environment, start_response)
```

Рассмотрим этот пример поближе:

```
import webob
```

Библиотека WebOb достаточно мала и содержит все необходимое нам на верхнем уровне.

```
def application(environment, start_response):
```

Само приложение WSGI. В данном случае это самая обычная функция, как если бы мы не использовали никакого фреймворка.

```
request = webob.Request(environment)
```

Объект запроса конструируется на основе словаря `environment`. Хотя наше приложение не использует объект запроса, при его создании анализируется большое число параметров: строка URL и параметры в строке запроса, а также файлы cookie и многое другое.

```
response = webob.Response()
```

Создается объект ответа. Этот шаг освобождает нас от необходимости помнить множество мелких деталей.

```
text='Hello world!')
```

Например, здесь мы записываем текст ответа в свойство `text`, не беспокоясь о его преобразовании в список строк байтов.

```
return response(environment, start_response)
```

Объект ответа знает, как вызвать `start_response` и записать тело.

Пример Pyramid

Pyramid – это фреймворк, предназначенный для создания приложений с минимальными накладными расходами и прекрасно масштабируемый для использования в больших проектах.

```
from pyramid import config, response

def hello_world(request):
    return response.Response('Hello World!')

with config.Configurator() as conf:
    conf.add_route('hello', '/')
    conf.add_view(hello_world, route_name='hello')
    application = conf.make_wsgi_app()
```

Рассмотрим этот пример строку за строкой.

```
from pyramid import config, response
```

Фреймворк Pyramid имеет довольно много компонентов. Но в данном примере нам необходимы только эти два модуля.

```
def hello_world(request):
```

Обратите внимание, что `hello_world` – это самая обычная функция Python. Она не имеет никаких специфических особенностей. Это открывает возможность ее повторного использования: например, мы можем написать тест для нее или использовать в какой-то другой функции.

```
    return response.Response('Hello World!')
```

Создается объект ответа, примерно так же, как это делается при использовании библиотеки `WebOb` или `werkzeug`.

```
with config.Configurator() as conf:
```

Использование конфигулятора в качестве диспетчера контекста означает, что в конце блока, если не возникнет никаких исключений, он автоматически зафиксирует конфигурацию и сохранит ее.

```
    conf.add_route('hello', '/')
```

Маршрутизация в Pyramid выполняется в два этапа. Отображение адреса URL в «логическое имя» – первый из них.

```
    conf.add_view(hello_world, route_name='hello')
```

Второй этап – отображение логического имени в представление.

```
    application = conf.make_wsgi_app()
```

В заключение мы преобразуем конфигурацию в приложение WSGI.

Начало

Документация, описывающая запуск приложений WSGI через Twisted, верна, но содержится в нескольких документах. Здесь мы шаг за шагом покажем законченный действующий пример запуска приложения WSGI.

Сервер WSGI

Роль сервера WSGI в Twisted играет плагин `web`. Далее мы будем использовать уникальный способ вызова плагина `python -m twisted`. Хотя этот способ немного сложнее, он пригодится вам для использования в промышленном окружении.

Несмотря на то что он не использует WSGI, вам будет полезно увидеть, как запускается плагин `web` – знание многих параметров пригодится вам и при использовании сервера WSGI, и при отладке ошибок на «слушающей стороне».

Если допустить, что фреймворк Twisted уже установлен, можно выполнить команду

```
$ python -m twisted web --port tcp:8000
```

и получить веб-сервер, выполняющий «демонстрационный пример». Роль этого демонстрационного примера играет веб-приложение, которое просто выводит приветственное сообщение – в данном случае в порт 8000.

Запустить любое приложение WSGI не составляет никакого труда – выше мы уже определили шесть таких приложений!

```
$ python -m twisted web --port tcp:8000 --wsgi wsgi_hello.application
$ python -m twisted web --port tcp:8000 --wsgi werkzeug_hello.application
$ python -m twisted web --port tcp:8000 --wsgi flask_hello.application
$ python -m twisted web --port tcp:8000 --wsgi webob_hello.application
$ python -m twisted web --port tcp:8000 --wsgi pyramid_hello.application
$ python -m twisted web --port tcp:8000 --wsgi django_hello.application
```

Обратите внимание, что данный подход даже проще, чем в случае с базовой реализацией. Для базовой реализации нам пришлось бы написать небольшой сценарий на языке командной оболочки, включающий вызов интерпретатора Python с параметром `-c` и блоком из четырех инструкций. Конечно, это хорошо, что Python и командная оболочка UNIX могут взаимодействовать друг с другом, но намного приятнее иметь возможность обойтись без этого.

Параметр `--port` обладает намного более мощными возможностями, чем может показаться.

```
$ python -m twisted web --port tcp:8000:interface=127.0.0.1 \
--wsgi wsgi_hello.application
```

Эта команда запустит веб-сервер, который будет принимать запросы только на локальном интерфейсе. Это особенно удобно для разработки веб-приложений с использованием сетей в кафе!

Но особую мощь параметру командной строки `--port` дает возможность использовать плагины. Некоторые плагины имеют особенно большое значение, и мы познакомимся с ними ниже.

Обратите внимание, что, в отличие от других полноценных серверов WSGI, Twisted не имеет конфигурационного файла. Есть лишь несколько параметров командной строки для небольших настроек, а для многих важных настроек используются значения по умолчанию, например размер пула потоков выполнения WSGI.

Настройку этих параметров можно реализовать в плагине.

```
# поместить этот код в twisted/plugins/twisted_book_wsgi.py
from zope import interface
from twisted.python import usage, threadpool
from twisted import plugin
from twisted.application import service, strports
from twisted.web import wsgi, server
from twisted.internet import reactor
import wsgi_hello
@interface.implementer(service.IServiceMaker, plugin.IPlugin)
class ServiceMaker(object):
    tapname = "twisted_book_wsgi"
    description = "WSGI for book"
    class options(usage.Options): pass
    def makeService(self, options):
        pool = threadpool.ThreadPool(minthreads=1, maxthreads=100)
        reactor.callWhenRunning(pool.start)
        reactor.addSystemEventTrigger('after', 'shutdown', pool.stop)
        root = wsgi.WSGIResource(reactor, pool, wsgi_hello.application)
        site = server.Site(root)
        return strports.service('tcp:8000', site)
        serviceMaker = ServiceMaker()
```

Пропустим инструкции импортирования и поближе рассмотрим оставшиеся:

```
@interface.implementer(service.IServiceMaker, plugin.IPlugin)
```

Так пишутся `tap`-плагины для Twisted. Этот декоратор отмечает класс как:

- нечто, что является плагином (`plugin.IPlugin`);
- нечто, что знает, как преобразовать командную строку в службу (`service.IServiceMaker`).

Это реализовано с использованием фреймворка `zope.interface`, который позволяет явно отмечать интерфейсы и их реализации, а также открывает программный доступ к этой информации. Этот программный интерфейс обеспечивает работу системы плагинов в Twisted.

```
class ServiceMaker(object):
```

Имя класса не имеет большого значения. Единственное, что важно, — экземпляр этого класса должен иметь имя `serviceMaker`.

```
tapname = "twisted_book_wsgi"
```

Имя плагина, которое можно использовать в первом аргументе в команде `python -m twisted`:

```
description = "WSGI for book"
```

Обычно желательно определять более информативное описание, потому что оно появляется в справке при попытке выполнить команду `python -m twisted` без аргументов.

```
class options(usage.Options): pass
```

Поскольку это минимальный плагин, мы «жестко» определяем все настройки. На самом деле здесь нет особой «жесткости» – сохраняется возможность принять решение о выборе порта и приложения. Такой подход вполне оправдан для плагинов, особенно если разработчики следуют методологии двенадцати факторов и запрашивают все настройки из переменных окружения.

Однако часто бывает полезно сделать возможным назначение из командной строки хотя бы порта.

```
def makeService(self, options):
```

Эта функция принимает экземпляр `options`, созданный после парсинга командной строки.

```
pool = threadpool.ThreadPool(minthreads=1, maxthreads=100)
```

Эту строку не следует рассматривать как пример хорошей конфигурации. В действительности это не самая удачная конфигурация пула потоков. Тем не менее иногда некоторая подстройка количества потоков имеет смысл. Во многом это зависит от приложения, компьютера и особенностей использования.

```
reactor.callWhenRunning(pool.start)
```

Настройка запуска пула с запуском реактора.

```
reactor.addSystemEventTrigger('after', 'shutdown', pool.stop)
```

Настройка остановки пула с остановкой реактора.

```
root = wsgi.WSGIResource(reactor, pool, wsgi_hello.application)
```

Создание корневого ресурса. Здесь пул потоков объединяется с конкретным приложением WSGI.

```
site = server.Site(root)
```

Создание объекта `Site`, который фактически анализирует запросы HTTP, получаемые от объекта ресурса.

```
return strports.service('tcp:8000', site)
```

Создание конечной точки и начало цикла приема входящих запросов HTTP.

```
serviceMaker = ServiceMaker()
```

Как уже упоминалось, фактический плагин зависит от экземпляра, а не от класса. Здесь создается экземпляр класса, который был определен выше.

Этот плагин позволяет запускать одно и то же приложение «hello world» с лучше (или, в данном случае, хуже) настроенным пулом потоков выполнения. Также можно создать плагин для удовлетворения других потребностей, часть из которых мы рассмотрим далее в этой главе.

Поиск кода

Самое важное, что должен сделать сервер Twisted WSGI, – найти приложение WSGI, которое нужно запустить. Однако такой поиск традиционно всегда был непрост.

Путь по умолчанию

При запуске интерпретатора Python с параметром `-s` или `-m` в путь поиска импортируемых модулей автоматически включается текущий каталог (`.`). Выше, в примере базовой реализации, мы использовали параметр `-s`, а в примере с сервером Twisted WSGI – параметр `-m`.

Однако когда сценарий запускается непосредственно, в путь поиска добавляется каталог сценария, а не текущий каталог. Поэтому если вместо команды `python -m twisted` использовать утилиту `twist`, текущий каталог не попадет в путь поиска модулей.

Проще говоря, текущий каталог не всегда включается в путь поиска, и мы не можем положиться на этот аспект. Если для демонстрационных целей данный подход вполне применим, то для использования в промышленном окружении нужно что-то более надежное.

PYTHONPATH

Один из таких надежных способов – настроить значение переменной окружения `PYTHONPATH`. Первый вопрос: какое значение выбрать: в одних случаях можно выбрать `PYTHONPATH = .`, в других – `PYTHONPATH = $(pwd)`. Преимущество первого варианта в том, что он позволяет следовать за командной оболочкой, но это преимущество одновременно является недостатком, потому что такая простая команда, как `cd`, может нарушить работу.

Преимущество второго варианта в его конкретности, но он тоже имеет свой недостаток, заключающийся в действии на расстоянии, когда запуск Python через некоторое время может внезапно импортировать старое приложение WSGI. Особую проблему это представляет для проектов, выполняющих поиск модулей в пути `PYTHONPATH`, например реализаций плагинов Twisted. Появление дополнительного плагина может оказаться неприятным сюрпризом.

setup.py

Самое лучшее решение – написать файл `setup.py` и превратить код в полноценный пакет. Правда, вам придется выбрать имя пакета, но обычно для этой цели

вполне подойдет имя модуля самого верхнего уровня. Также желательно выбрать номер версии, но если у вас нет намерения распространять свой пакет, вполне подойдет имя с версией 0.0.0dev1.

Для нужд разработки часто проще выполнить установку в виртуальное окружение с помощью команды `pip install -e ..`. Это позволит отслеживать изменения в исходных файлах и уменьшит хлопоты при интеграции с системой виртуальных окружений в Python или любыми другими системами виртуальных окружений, такими как Nix или Conda.

Почему Twisted?

Конечно, применение фреймворка Twisted – не единственный способ запуска приложений WSGI. Для этой цели также можно использовать Gunicorn, uwsgi и модуль `mod_wsgi` веб-сервера Apache. Однако Twisted предлагает ряд интересных преимуществ.

Промышленная эксплуатация и разработка

В состав большинства веб-фреймворков входят свои реализации веб-серверов, часто основанные на `wsgiref`. В документации к этим серверам обычно можно встретить предупреждение: «DO NOT USE THIS SERVER IN A PRODUCTION SETTING. It has not gone through security audits or performance tests» (Этот сервер не предназначен для использования в промышленном окружении. Он не тестировался на предмет безопасности и производительности)¹. В худшем случае на это предупреждение не обращают внимания – по незнанию или по другим причинам – и запускают веб-сайты поверх серверов, предназначенных исключительно для разработки.

В лучшем случае к этим предупреждениям относятся со всем вниманием и используют сервер разработки только для разработки, а в промышленном окружении применяют сервер промышленного класса. Это неизбежно влечет изменение окружения – некоторые тонкие отличия в реализации WSGI, например, приводят к тому, что некоторые особенности поведения под управлением промышленного сервера не воспроизводятся в процессе разработки. Кроме того, разработчики лишены возможности наблюдать нормальную работу промышленного веб-сервера. Журналы, сообщения об ошибках и аварии – все это может существенно отличаться и приводить к потере связи между разработкой и эксплуатацией.

Наконец, при использовании двух веб-серверов следует придерживаться определенной логики, выбирая, где, что и когда запускать. Часто может возникать путаница, приводящая к случайному запуску сервера разработки в промышленном окружении. Поскольку сервер разработки все-таки имеет некоторый запас прочности, такая ошибка не приводит к немедленному отказу, а проявляется в виде множества необъяснимых проблем, таких как проблема производительности.

¹ Эта цитата взята из документации Django.

Сервер Twisted, напротив, может использоваться и для разработки, и в промышленном окружении. Его можно запустить из командной строки, как было показано выше, передавая только имя приложения. Если впоследствии окажется удобнее написать плагин, то такой плагин можно использовать и в разработке. Это позволяет нивелировать разницу между промышленным окружением и окружением разработки.

Некоторые из наиболее развитых серверов разработки поддерживают полезную функцию автоматической перезагрузки кода. Однако, потратив немного времени на настройку, такого же поведения можно добиться и от Twisted. Прежде всего нужно установить наш код командой `pip install -e`, в результате чего достаточно будет просто перезапустить сервер. Затем, вместо того чтобы запускать сервер непосредственно, можно использовать команду

```
$ watchmedo shell-command \
  --patterns="*.py" \
  --recursive \
  --command='python -m twisted web --wsgi=wsgi_hello.application' \
  .
```

Она обеспечит автоматический перезапуск сервера при изменении любого файла. Здесь используются возможности пакета `watchdog` из каталога PyPI.

TLS

TLS (Transport Layer Security – защита транспортного уровня) – это последняя версия того, что раньше называлось SSL (Secure Socket Layer – уровень защищенных сокетов). TLS – это протокол шифрования и обмена ключами, работающий поверх TCP.

Этот протокол обеспечивает:

- шифрование: соединения по протоколу TLS защищены от прослушивания;
- проверку подлинности конечной точки: при использовании TLS можно убедиться, что соединение установлено с ожидаемой конечной точкой.

Первый пункт из этого списка часто используется для обоснования важности использования протокола TLS, но второй пункт еще более важен. Многие приложения WSGI почти не передают конфиденциальных данных, тем не менее, поскольку они посылают HTML, JavaScript и CSS в потенциально уязвимые браузеры, важно гарантировать, что никакие вредоносные программы не будут переданы по сети.

Проверка подлинности конечных точек в TLS основана на проверке сертификатов, подписанных центрами сертификации. Получить подпись центра сертификации можно двумя путями – убедить сторонний центр сертификации, что вы являетесь законным представителем конечной точки, или создать свой центр сертификации. Создать реальный центр сертификации практически невозможно, но часто такое решение предпочтительнее для организации обмена внутри центров обработки данных, когда один и тот же человек или группа отвечает за обе стороны соединения.

Допустим, что имеется ключ в файле `key.pem` и сертификат в файле `cert.pem`, тогда следующая команда

```
$ python -m twisted web \
    --port ssl:port=8443:privateKey=key.pem:certKey=cert.pem \
    --wsgi wsgi_hello.application
```

запустит сервер TLS с приложением. Обратите внимание, что в этом случае словарь `environment` будет иметь ключ `wsgi.url_scheme` со значением `"https"`. Приложения WSGI могут воспользоваться этим фактом, чтобы убедиться, что обмен происходит по протоколу TLS.

Это одно из преимуществ непосредственной реализации протокола TLS на сервере WSGI. В ее отсутствие потребуются исследовать малоизвестные и нестандартные заголовки HTTP, чтобы узнать, является ли запрос безопасным.

Индикация имени сервера

Приложения WSGI имеют доступ ко многим заголовкам, в том числе и к заголовку `Host`. Анализируя этот заголовок, приложение WSGI может узнать, какое имя сервера использовал клиент для обращения к нему, и использовать его как один из параметров для настройки своего поведения. Например, представьте, что приложение должно возвращать разный контент при обращении к нему из обычного браузера по адресу `example.com` и из мобильного браузера по адресу `m.example.com`.

Если при этом приложение должно также поддерживать защищенные соединения по протоколу TLS, мы должны иметь сертификаты для `m.example.com` и `example.com` и знать, какой из них использовать. Для этой цели TLS поддерживает расширение «Server Name Indication» (SNI – индикация имени сервера), которое позволяет клиенту сообщить имя сервера, с которым он хотел бы установить соединение.

Для поддержки SNI в WSGI необходимо:

- получить соответствующие сертификаты и ключи;
- для каждого имени хоста *объединить* сертификат и ключ в один файл (в UNIX это можно сделать командой `cat`). Этот файл должен иметь имя `<host>.pem`, например `m.example.com.pem`;
- поместить все эти файлы в один каталог, например `/var/lib/keys`;
- установить пакет `txsni` из каталога PyPI;
- запустить приложение:

```
$ python -m twisted web \
    --port txsni:/var/lib/keys:tcp:8443 \
    --wsgi wsgi_hello.application
```

Этот пример прекрасно будет работать в случаях, когда требуется обслуживать один и тот же контент с двух разных доменных имен, например `example.com` и `www.example.com`.

Чтобы обслуживать разный контент для разных поддоменов, например динамическое содержимое из `app.example.com` и статические файлы из `static`.

example.com, можно использовать не раз упоминавшийся параметр `--port` и передавать в нем свой плагин, создающий ресурс `twisted.web.vhost.NameVirtualHost`.

Вот пример такого плагина:

```
from zope import interface
from twisted.python import usage, threadpool
from twisted import plugin
from twisted.application import service, strports
from twisted.web import wsgi, server, static, vhost
from twisted.internet import reactor

import wsgi_hello

@interface.implementer(service.IServiceMaker, plugin.IPlugin)
class ServiceMaker(object):
    tapname = "twisted_book_vhost"
    description = "Virtual hosting for book"
    class options(usage.Options):
        optParameters = [
            ["port", "p", None,
             "strports description of the port to "
             "start the server on."]]

    def makeService(self, options):
        application = wsgi_hello.application
        pool = threadpool.ThreadPool(minthreads=1, maxthreads=100)
        reactor.callWhenRunning(pool.start)
        reactor.addSystemEventTrigger('after', 'shutdown', pool.stop)
        dynamic = wsgi.WSGIResource(reactor, pool, application)
        files = static.File('static')
        root = vhost.NameVirtualHost()
        root.addHost(b'app.example.org', dynamic)
        root.addHost(b'static.example.org', files)
        site = server.Site(root)
        return strports.service(options['port'], site)

serviceMaker = ServiceMaker()
```

Наибольший интерес представляют следующие строки:

```
root = vhost.NameVirtualHost()
root.addHost(b'app.example.org', dynamic)
root.addHost(b'static.example.org', files)
```

Здесь создается корневой ресурс, который пересылает все запросы к адресу `app.example.org` в ресурс `dynamic`, а все запросы к адресу `static.example.org` – в ресурс `files`. Обратите внимание: глядя на имя `example.org`, можно с уверенностью сказать, что оно предназначено для целей тестирования и соответствует IP-адресу `127.0.0.1` в вашем файле `hosts`.

Отметьте также, что в этом примере мы не использовали имя по умолчанию. По этой причине попытка обратиться к приложению с использованием другого имени (например, `localhost`) вызовет ошибку 404. Чтобы устранить ее, можно определить в свойстве `default` объекта `NameVirtualHost` корень по умолчанию для всех других имен.

Статические файлы

Использование Twisted в роли сервера WSGI позволяет организовать на одном веб-сервере обслуживание не только динамических приложений, но и статических ресурсов, таких как изображения, сценарии на JavaScript и таблицы стилей CSS, а также любые другие файлы.

Фреймворк Twisted изначально создавался как высокопроизводительное сетевое приложение, и с точки зрения обслуживания статических файлов веб-сервер Twisted способен удовлетворить все потребности, кроме самых сложных. Однако при удовлетворении этих потребностей большинство приложений будут обслуживаться через сеть распространения контента (Content Distribution Network, CDN).

Но при использовании CDN скорость обслуживания статических файлов играет второстепенную роль. Однако в таких случаях из кода на Python удобно устанавливать заголовки `Cache-Control`. Разработчики, которые пишут WSGI-приложения на Python, обычно хорошо владеют этим языком и предпочитают использовать его вместо изучения другого узкоспециализированного языка, такого как язык конфигурации, встроенный в большинство серверов.

Однако, чтобы понять, как это сделать, нужно глубже вникнуть в программный интерфейс веб-сервера Twisted и разобраться с некоторыми понятиями, вскользь представленными ранее.

Модель ресурсов

Большинство современных веб-серверов используют модель маршрутизации (если вообще имеют), основанную на *сопоставлении с шаблонами*. Flask, Django и Pyramid, как мы видели выше, так или иначе сопоставляют URL с шаблонами.

Модуль `web` в Twisted появился намного раньше, еще до того, как прием сопоставления URL с шаблонами приобрел популярность, и когда широко применялся альтернативный подход – обработка веб-ресурсов в виде дерева. Именно этот альтернативный подход использован в модуле `web` в Twisted. В результате модель ресурсов в нем предполагает наличие дочерних ресурсов, или ресурсов-потомков.

Это не так важно, если использовать только WSGI: ресурс WSGI помечает себя как `isLeaf = True`, указывая, что не имеет дочерних элементов, и обход дерева прекращается при его достижении. Это позволяет ресурсу WSGI передать путь в инфраструктуру веб-приложения, чтобы оно могло выполнить маршрутизацию самостоятельно. Поскольку в роли корневого ресурса мы использовали ресурс WSGI, который затем передали конструктору `Site`, это означает, что модель дерева ресурсов существует только теоретически.

Однако при использовании комбинаций различных ресурсов особенности этой модели приобретают особую важность.

Чисто статические ресурсы

Чтобы лучше понять, как происходит обслуживание статических файлов Twisted-модулем web, напишем для начала плагин, обслуживающий исключительно статические ресурсы.

```
from zope import interface

from twisted.python import usage, threadpool
from twisted import plugin
from twisted.application import service, strports
from twisted.web import static, server
from twisted.internet import reactor

@interface.implementer(service.IServiceMaker, plugin.IPlugin)
class ServiceMaker(object):
    tapname = "twisted_book_static"
    description = "Static for book"
    class options(usage.Options):
        pass
    def makeService(self, options):
        root = static.File('static')
        site = server.Site(root)
        return strports.service('tcp:8000', site)

serviceMaker = ServiceMaker()
```

Новая для нас здесь только одна строка:

```
root = static.File('static')
```

Она определяет ресурс File. Ресурс File тоже является листовым ресурсом, который сопоставит оставшуюся часть URL с путем в файловой системе. При этом он будет применять относительный путь static в текущем рабочем каталоге. Такой подход можно использовать для иллюстрации, но в промышленных приложениях обычно предпочтительнее использовать полный путь.

Один из способов получить полный путь – это упаковать файлы вместе с кодом Python. Для этого потребуется немного помудрить над установкой и поиском во время выполнения.

Вот пример setup.py и плагина, который использует его:

```
import setuptools
setuptools.setup(
    name='static_server',
    license='MIT',
    description="Server: Static",
    long_description="Static, the web server",
    version="0.0.1",
    author="Moshe Zadka",
    author_email="zadka.moshe@gmail.com",
    packages=setuptools.find_packages(where='src') + ['twisted/plugins'],
```

```
package_dir={"": "src"},
include_package_data=True,
install_requires=['twisted', 'setuptools'],
)
```

Самое интересное здесь – строка `include_package_data=True`. Но, чтобы действительно добавить данные, необходимо объявить пути к ним, поэтому добавим в `MANIFEST.in` строку:

```
include src/static_server/a_file.html
```

Вот плагин, обслуживающий этот файл (в данном случае находящийся в пути `/`):

```
import pkg_resources

from zope import interface

from twisted.python import usage, threadpool
from twisted import plugin
from twisted.application import service, strports
from twisted.web import static, server, resource
from twisted.internet import reactor

@interface.implementer(service.IServiceMaker, plugin.IPlugin)
class ServiceMaker(object):
    tapname = "twisted_book_pkg_resources"
    description = "Static for book"
    class options(usage.Options):
        pass
    def makeService(self, options):
        root = resource.Resource()
        fname = pkg_resources.resource_filename("static_server",
                                                "a_file.html")
        static_resource = static.File(fname)
        root.putChild("", static_resource)
        site = server.Site(root)
        return strports.service('tcp:8000', site)

serviceMaker = ServiceMaker()
```

Новые строки, представляющие интерес:

```
fname = pkg_resources.resource_filename("static_server",
                                        "a_file.html")
static_resource = static.File(fname)
```

Здесь для поиска файла во время выполнения используется пакет `pkg_resources`, входящий в состав `setuptools`.

Обратите внимание, что этот прием будет работать, даже если наш пакет развернут в виде `zip`-файла с использованием такого инструмента, как `rex` (или `ziparr`): `pkg_resources` способен автоматически распаковать файл.

Этот метод также можно использовать для включения в пакет файлов шаблонов при использовании таких систем, как `Jinja2` и `Chameleon`.

Комбинирование статических файлов с WSGI

Обслуживать статические ресурсы для приложения WSGI можно также через собственный веб-сервер Twisted.

```
import os

from zope import interface
from twisted.python import usage, threadpool
from twisted import plugin
from twisted.application import service, strports
from twisted.web import wsgi, server, static, resource
from twisted.internet import reactor

import wsgi_hello

class DelegatingResource(resource.Resource):
    def __init__(self, wsgi_resource):
        resource.Resource.__init__(self)
        self._wsgi_resource = wsgi_resource

    def getChild(self, name, request):
        request.prepath = []
        request.postpath.insert(0, name)
        return self._wsgi_resource

@interface.implementer(service.IServiceMaker, plugin.IPlugin)
class ServiceMaker(object):
    tapname = "twisted_book_combined"
    description = "twisted_book_combined"

    class options(usage.Options): pass

    def makeService(self, options):
        application = wsgi_hello.application
        pool = threadpool.ThreadPool()
        reactor.callWhenRunning(pool.start)
        reactor.addSystemEventTrigger('after', 'shutdown', pool.stop)
        wsgi_resource = wsgi.WSGIResource(reactor, pool, application)
        static_resource = static.File('.')
        root = DelegatingResource(wsgi_resource)
        root.putChild('static', static_resource)
        site = server.Site(root)
        return strports.service('tcp:8000', site)

serviceMaker = ServiceMaker()
```

Рассмотрим новый код строку за строкой:

```
class DelegatingResource(resource.Resource):
```

Определяется класс с именем `DelegatingResource`. Он наследует класс `resource.Resource` и будет служить нам корневым ресурсом. Обратите внимание, что это не листовой ресурс, и поэтому сайт будет проходить через него.

```
def __init__(self, wsgi_resource):
```

Делегат иницируется ресурсом WSGI.

```
resource.Resource.__init__(self)
```

Как обычно, вызываем конструктор суперкласса. Это очень важно – Resource не будет работать правильно, если не вызвать его конструктор.

```
self.wsgi_resource = wsgi_resource
```

Сохраняем ресурс WSGI в атрибуте.

```
def getChild(self, name, request):
```

Имя getChild немного сбивает с толку. Его семантика заключается в получении *динамического* потомка. Статический потомок, то есть добавленный в ресурс Resource вручную, предотвратит вызов этого метода. Корень никогда не будет вызываться в `render`: даже URL, такой как `/`, даст в результате обхода потомков пустую строку в `name`.

```
request.prepath = []  
request.postpath.insert(0, name)
```

Мы перемещаем имя из `prepath` в `postpath` и таким нехитрым способом обманываем делегированный ресурс, который является корневым. Обратите внимание, что этот трюк работает только для корневых ресурсов.

```
return self.wsgi_resource
```

После трюка с путем, позволяющего сделать вид, что число проходов выполнено на один меньше, мы возвращаем ресурс WSGI.

```
static_resource = static.File('.')
```

Создаем статический ресурс. Он ничем не отличается от чисто статического ресурса.

```
root = DelegatingResource(wsgi_resource)
```

Создаем делегирующий ресурс как корневой.

```
root.putChild('static', static_resource)
```

Как отмечалось выше, дочерний элемент, введенный вручную, переопределяет метод `getChild`. То есть для любого пути, который начинается с `/static/`, будет обслуживаться статический ресурс.

Встроенное планирование задач

В следующем примере мы пытаемся создать приложение WSGI, зависящее от параметра, который мы можем изменить.

```
class _Application(object):  
    def __init__(self, greeting='hello world'):  
        self.greeting = greeting
```

```
def __call__ (self, environment, start_response):
    start_response('200 OK', [('Content-Type',
                                'text/html; charset=utf-8')])
    return [self.greeting.encode('utf-8')]
```

```
application = _Application()
```

Рассмотрим этот код строку за строкой:

```
class _Application(object):
```

Как упоминалось выше, единственное предположение, которое можно сделать о приложениях WSGI, состоит в том, что они являются вызываемыми объектами. В этом случае мы создаем вызываемый объект, определяя класс с методом `__call__`.

```
def __init__ (self, greeting='hello world'):
```

Инициализируется стандартное приветствие.

```
self.greeting = greeting
```

В конструкторе мы не делаем ничего интересного – только инициализируем атрибуты.

```
def __call__ (self, environment, start_response):
```

Поскольку это приложение WSGI, оно вызывается со стандартными параметрами.

```
start_response('200 OK', [('Content-Type',
                                'text/html; charset=utf-8')])
```

Это тот же вызов `start_response`, что и прежде, только на этот раз явно указан набор символов. Поскольку клиентам разрешено передавать произвольные строки Юникода и мы возвращаем ответ в кодировке `utf-8`, то должны сообщить браузеру об этом.

```
return [self.greeting.encode('utf-8')]
```

Чтобы иметь возможность представлять приветствия в виде строк, мы должны преобразовать их в байты.

```
application = _Application()
```

Нам нужен не класс, а экземпляр приложения.

```
import time
from zope import interface
from twisted.python import usage, reflect, threadpool, filepath
from twisted import plugin
from twisted.application import service, strports, internet
from twisted.web import wsgi, server, static
from twisted.internet import reactor
import wsgi_param
```

```
def update(application, reactor):
    stamp = time.ctime(reactor.seconds())
    application.greeting = "hello world, it's {}".format(stamp)

@interface.implementer(service.IServiceMaker, plugin.IPlugin)
class ServiceMaker(object):
    tapname = "twisted_book_scheduled"
    description = "Changing application"
    class options(usage.Options): pass
    def makeService(self, options):
        s = service.MultiService()
        pool = threadpool.ThreadPool()
        reactor.callWhenRunning(pool.start)
        reactor.addSystemEventTrigger('after', 'shutdown', pool.stop)
        root = wsgi.WSGIResource(reactor, pool, wsgi_param.application)
        site = server.Site(root)
        strports.service('tcp:8000', site).setServiceParent(s)
        ts = internet.TimerService(1, update, wsgi_param.application, reactor)
        ts.setServiceParent(s)
        return s

serviceMaker = ServiceMaker()
```

Рассмотрим этот код строку за строкой:

```
def update(application, reactor):
```

Эта функция будет вызываться периодически для обновления приложения.

```
stamp = time.ctime(reactor.seconds())
```

Здесь мы используем `reactor.seconds()` вместо `time.time()`. По мере разрабатывания кода это поможет нам тестировать его.

```
application.greeting = "hello world, it's {}".format(stamp)
```

Устанавливает атрибут `greeting` приложения. Так как атрибут является общедоступным, его можно считать частью API класса.

Обратите внимание, что здесь используется изменчивое глобальное состояние, что в общем случае считается опасным шаблоном программирования и вдвойне опасным в многопоточном окружении. Несмотря на то что в главном цикле Twisted нет других потоков выполнения, весь механизм WSGI работает внутри пула потоков Twisted.

Однако в этом конкретном случае изменение не несет опасности – поток увидит либо старое приветствие, либо новое. Это обусловлено действием глобальной блокировки интерпретатора Python, которая гарантирует, что потоки выполнения в Python будут видеть только согласованное состояние, а также тем, что это всего лишь замена одной строки другой.

```
s = service.MultiService()
```

Создает службу, которая запускает несколько служб. Это позволяет одной службой осуществлять обслуживание веб-клиентов и обновление.

```
strports.service('tcp:8000', site).setServiceParent(s)
```

На этот раз вместо возврата результата `strports.service` мы назначаем его родителем объект `MultiService`. Это закрепит его за `MultiService` в роли потомка.

```
ts = internet.TimerService(1, update, wsgi_param.application, reactor)
```

Здесь мы создаем таймер, который срабатывает каждую секунду и вызывает функцию `update` с параметрами `wsgi_param.application` и `reactor`.

```
ts.setServiceParent(s)
```

Присоединяем таймер к возвращаемому значению.

```
return s
```

И возвращаем экземпляр `MultiService`.

Определенно это не лучший способ отображения часов, но во многих случаях подобное отделение извлечения значения от его отображения имеет определенный смысл. Представьте биржевое приложение: лучше извлекать цены на акции раз в секунду и отображать их из памяти при веб-запросе, чем заставлять каждый веб-запрос ждать, пока служба выполнит все необходимые операции (которые могут потребовать времени).

Этот пример показал преимущества выполнения службы по графику. Таким же способом, например, можно запланировать очистку журнала. Этот прием позволяет хранить конфигурацию приложения в одном месте, а не добавлять зависимость от службы, такой как `cron`.

Каналы управления

Часто бывает полезно иметь возможность менять конфигурацию веб-приложения во время выполнения, без перезапуска или пересборки, например:

- для изменения уровня журналирования с целью решить проблему;
- для изменения доли трафика контрольной/тестовой страницы при проведении A/B-тестирования;
- для отключения некоторой особенности при получении жалоб от клиентов.

Это означает, что помимо «канала приложения», по которому конечный пользователь взаимодействует с приложением, нужен дополнительный канал управления, позволяющий влиять на поведение. Безопаснее всего организовать доступ к этому каналу через другой порт и, возможно, по другому протоколу – в этом случае доступность канала управления можно ограничить с помощью обычного брандмауэра и настроек сети, а не только с использованием средств защиты на уровне приложения.

Поскольку фактически `Twisted` – это фреймворк для обработки сетевых событий, он идеально подходит для добавления каналов управления в приложения `WSGI`. Так как такие каналы управления по своей природе пересекают границы потоков выполнения, необходимо позаботиться о безопасности данных в многопоточном окружении.

Он также позволяет добавлять в приложения WSGI новые и интересные особенности.

Следующий плагин демонстрирует, как организовать управление приветствием по сети.

```
from zope import interface

from twisted.python import usage, reflect, threadpool, filepath
from twisted import plugin
from twisted.application import service, strports, internet
from twisted.web import wsgi, server, static
from twisted.internet import reactor, protocol
from twisted.protocols import basic

import wsgi_param

class UpdateMessage(basic.LineReceiver):

    def lineReceived(self, line):
        self.factory.application.greeting = line.decode('utf-8')
        self.transport.writeSequence([b"greeting is now: ", line, b"\r\n"])
        self.transportloseConnection()

@interface.implementer(service.IServiceMaker, plugin.IPlugin)
class ServiceMaker(object):
    tapname = "twisted_book_control"
    description = "Changing application"
    class options(usage.Options): pass
    def makeService(self, options):
        s = service.MultiService()
        pool = threadpool.ThreadPool()
        reactor.callWhenRunning(pool.start)
        reactor.addSystemEventTrigger('after', 'shutdown', pool.stop)
        root = wsgi.WSGIResource(reactor, pool, wsgi_param.application)
        site = server.Site(root)
        strports.service('tcp:8000', site).setServiceParent(s)
        factory = protocol.Factory.forProtocol(UpdateMessage)
        factory.application = wsgi_param.application
        strports.service('tcp:8001', factory).setServiceParent(s)
        return s

serviceMaker = ServiceMaker()
```

Рассмотрим этот код строку за строкой:

```
class UpdateMessage(basic.LineReceiver):
```

Объявление подкласса протокола `basic.LineReceiver`. Он разбивает сообщения на строки, что дает возможность ограничивать сообщения.

```
def lineReceived(self, line):
```

Этот метод вызывается при получении строки. Обратите внимание, что строка `line` *не* будет содержать символов завершения (по умолчанию возврат каретки и перевод строки, `\r\n`).

```
self.factory.application.greeting = line
```

Копируем строку в переменную, определяющую текст приветствия.

```
factory = protocol.Factory.forProtocol(UpdateMessage)
```

Создаем фабрику, которая производит экземпляры UpdateMessage при подключении клиента.

```
factory.application = wsgi_param.application
```

Копируем ссылку на наше приложение WSGI в фабрику. Это позволит объекту протокола обратиться к приложению, чтобы изменить приветствие.

```
strports.service('tcp:8001', factory).setServiceParent(s)
```

Связываем протокол с портом, номер которого на единицу больше номера порта приложения.

СТРАТЕГИИ ПАРАЛЛЕЛЬНОГО ВЫПОЛНЕНИЯ

Единственное ограничение, которое накладывает использование Twisted в роли WSGI-сервера, заключается в том, что фреймворк запускает только один процесс. Поскольку в Python имеется глобальная блокировка интерпретатора, это означает, что на многопроцессорном или многоядерном компьютере сервер WSGI будет использовать лишь одно ядро. Обычно это не является проблемой: в некоторых окружениях нижний уровень предоставляет приложениям одноядерный «компьютер». Например, это часто имеет место при использовании платформ виртуализации или инфраструктур управления контейнерами.

Однако иногда на уровне приложения желательно иметь решение с полноценной поддержкой выполнения на нескольких процессорах. Здесь мы продемонстрируем несколько таких решений.

Балансировка нагрузки

Самый простой способ – запустить несколько процессов Twisted WSGI и поместить перед ними балансировщик нагрузки. Одним из популярных балансировщиков нагрузки является HAProxy. Полное описание HAProxy выходит за рамки рассматриваемой темы, тем не менее ниже приводится пример конфигурации HAProxy. Для простоты эта конфигурация предусматривает обработку простого трафика HTTP, но вообще HAProxy часто используется для обслуживания SSL-трафика.

```
defaults
    log      global
    mode     http
frontend localnodes
    bind *:8080
    mode http
    default_backend nodes
```

```
backend nodes
mode http
balance roundrobin
option forwardfor
http-request set-header X-Forwarded-Port %[dst_port]
http-request add-header X-Forwarded-Proto https if { ssl_fc }
option httpchk HEAD / HTTP/1.1\r\nHost:localhost
server web01 127.0.0.1:9000 check
server web02 127.0.0.1:9001 check
server web03 127.0.0.1:9002 check
```

Последние три строки особенно важны: они направляют трафик трем разным локальным веб-серверам.

Теперь нам нужно каким-то способом запустить все четыре процесса – балансировщик HAProxy и три веб-сервера. В этом примере мы используем ncolony.

```
$ alias add="python -m ncolony --messages /var/run/messages \
--config /var/run config add"
$ add --cmd haproxy --arg=-f --arg=/my/haproxy.cfg haproxy
$ add --cmd python --arg=-m --arg=twisted \
--arg=web --arg=--wsgi \
--arg=wsgi_hello.application \
--arg=--port --arg=tcp:9001 web1
$ add --cmd python --arg=-m --arg=twisted \
--arg=web --arg=--wsgi \
--arg=wsgi_hello.application \
--arg=--port --arg=tcp:9002 web2
$ add --cmd python --arg=-m --arg=twisted \
--arg=web --arg=--wsgi \
--arg=wsgi_hello.application \
--arg=--port --arg=tcp:9003 web3
$ python -m twisted ncolony --messages /var/run/messages \
--config /var/run config add
```

Открытие сокета в режиме совместного использования

Относительно недавно в Linux появилась поддержка режима `SO_REUSEPORT` сокетов. Он позволяет нескольким серверам прослушивать один и тот же порт. Однако, поскольку эта функция появилась лишь недавно, Twisted не поддерживает ее по умолчанию.

Чтобы воспользоваться этой новой возможностью, нужно вторгнуться в нижние уровни Twisted.

```
import socket
import attr
from zope import interface
from twisted.python import usage, threadpool
from twisted import plugin
from twisted.application import service, internet as tainternet
from twisted.web import wsgi, server
```

```

from twisted.internet import reactor, tcp, interfaces as tiinterfaces, defer
import wsgi_hello

@interface.implementer(tiinterfaces.IStreamServerEndpoint)
@attr.s
class ListenerWithReuseEndPoint(object):
    port = attr.ib()
    reactor = attr.ib(default=None)
    backlog = attr.ib(default=50)
    interface = attr.ib(default="")

    def listen(self, protocolFactory):
        p = tcp.Port(self.port, protocolFactory, self.backlog, self.interface,
                     self.reactor)
        self._sock = sock = p.createInternetSocket()
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
        sock.bind((self.interface, self.port))
        sock.listen(self.backlog)
        return defer.succeed(reactor.adoptStreamPort(sock.fileno(),
                                                    p.addressFamily(),
                                                    protocolFactory))

@interface.implementer(service.IServiceMaker, plugin.IPlugin)
class ServiceMaker(object):
    tapname = "twisted_book_reuseport"
    description = "Reuse port"
    class options(usage.Options): pass
    def makeService(self, options):
        application = wsgi_hello.application
        pool = threadpool.ThreadPool(minthreads=1, maxthreads=100)
        reactor.callWhenRunning(pool.start)
        reactor.addSystemEventTrigger('after', 'shutdown', pool.stop)
        root = wsgi.WSGIResource(reactor, pool, application)
        site = server.Site(root)
        endpoint = ListenerWithReuseEndPoint(8000)
        service = tainternet.StreamServerEndpointService(endpoint, site)
        return service

serviceMaker = ServiceMaker()

```

Это, безусловно, самый сложный плагин из всех, что были представлены в данной книге. В промышленном коде большую часть логики было бы лучше вынести за пределы плагина.

Но в данном случае демонстрация всего этого кода в одном листинге помогает лучше понять его.

```
@interface.implementer(tiinterfaces.IStreamServerEndpoint)
```

Имя модуля кажется странным. В разветвленной иерархии модулей Twisted некоторые имена повторяются в разных ее точках. Поэтому для импортирования модулей применяется удобное соглашение, заключающееся в использовании некоторых начальных букв из иерархии, чтобы сделать цель более понятной. В данном случае `tiinterfaces` означает `twisted.internet.interfaces`.

Мы реализуем интерфейс `IStreamServerEndpoint`, так как нам нужно реализовать конечную точку нового типа – открывающую сокеты в режиме `REUSEPORT`.

```
@attr.s
```

Поскольку в этом классе имеется множество атрибутов данных, мы используем пакет `attrs`, чтобы упростить код.

```
class ListenerWithReuseEndPoint(object):
    port = attr.ib()
    reactor = attr.ib(default=None)
    backlog = attr.ib(default=50)
    interface = attr.ib(default="")
```

Мы принимаем те же самые аргументы, что и `reactor.listenTCP`. Это сделано преднамеренно.

```
def listen(self, protocolFactory):
```

Это единственный метод в интерфейсе `IStreamServerEndpoint`.

```
p = tcp.Port(self.port, protocolFactory, self.backlog, self.interface,
             self.reactor)
self._sock = sock = p.createInternetSocket()
```

Низкоуровневые механизмы поддержки TCP в Twisted, реализованные в модуле `tcp.Port`, обеспечивают правильную настройку неблокирующего режима для сокета. Мы сохраняем ссылку на объект сокета, чтобы предотвратить его утилизацию сборщиком мусора. Это важно, потому что мы будем создавать новый объект сокета на уровне Python из того же дескриптора файла.

```
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
```

Вот истинная причина всей этой канители – установка параметра `SO_REUSEPORT`.

```
sock.bind((self.interface, self.port))
```

Сокет привязывается к интерфейсу.

```
sock.listen(self.backlog)
```

И переводится в режим прослушивания.

```
return defer.succeed(reactor.adoptStreamPort(sock.fileno(),
                                              p.addressFamily,
                                              protocolFactory))
```

Извлекаем дескриптор файла из объекта сокета и позволяем Twisted «закрепить» его. В ответ фреймворк возвращает `IListeningPort`. Согласно контракту `listen` должен вернуть объект `Deferred`, поэтому мы заключаем результат в `defer.succeed`.

Чтобы включить все это в работу, снова используем `ncolony`.

```
$ alias add="python -m ncolony --messages /var/run/messages \
--config /var/run/config add"
```

```
$ add --cmd python --arg=-m --arg=twisteded \
    --arg=twisted_book_reuseport web1
$ add --cmd python --arg=-m --arg=twisteded \
    --arg=twisted_book_reuseport web2
$ add --cmd python --arg=-m --arg=twisteded \
    --arg=twisted_book_reuseport web3
$ python -m twist ncolony --messages /var/run/messages \
    --config /var/run/config add
```

Как и в предыдущем примере, здесь запускаются три веб-сервера. Обратите внимание, что на этот раз все три команды идентичны и отпала необходимость в балансировщике нагрузки.

Другие варианты

Вообще в Twisted есть еще несколько вариантов организации параллельной обработки данных. Можно создать сокет, а затем запускать процессы, которые будут прослушивать его. Это означает, что придется связать управление процессами с кодом, прослушивающим сокет, что очень неудобно. Например, для мониторинга процессов больше нельзя использовать `ncolony` или `twisted.runner.gprocmon`. Если «родительский» процесс завершится, мы окажемся перед дилеммой: перезапустить его и остановить все дочерние процессы или дождаться, пока все дочерние процессы завершатся сами.

Другой вариант – прослушивание сокета в одном процессе и передача файловых дескрипторов через сокет домена UNIX. Это решение непереносимо и требует глубокого понимания магии системного вызова `socket`.

В общем случае варианты с совместным использованием сокета и балансировкой нагрузки выглядят более предпочтительными. Обратите внимание, что так же, как любое улучшение производительности, эффект конкретного выбора (например, совместного использования сокета или балансировки нагрузки) должен измеряться в окружении, максимально приближенном к промышленному окружению.

ДИНАМИЧЕСКАЯ КОНФИГУРАЦИЯ

Как отмечалось выше, использование Twisted в роли сервера WSGI позволяет добавлять в приложения каналы управления, помогающие производить перенастройку во время выполнения. Здесь мы рассмотрим полноценный пример реализации такого управления с использованием протокола асинхронного обмена сообщениями (Asynchronous Messaging Protocol, AMP) в качестве протокола управления. Пример включает само приложение, а также управляющее приложение.

Приложение Pyramid с поддержкой А/В-тестирования

Под А/В-тестированием понимается показ разных версий веб-приложения разным пользователям и проверка влияния разных возможностей на разные

показатели. Например, в приложении электронной коммерции можно поэкспериментировать с размещением кнопки «Оформить заказ» и проверить, как это повлияет на количество клиентов, совершивших покупку.

Существует множество полнофункциональных вариантов А/В-тестирования для веб-фреймворков на Python. У нас нет возможности написать в книге полнофункциональную альтернативу, поэтому мы покажем только одну из основных частей: изменение вывода. В общем случае выходные данные должны быть постоянными при показе данному пользователю, но для этого требуется согласованная конструкция сеанса, а эта тема выходит за рамки нашей книги.

Наше «тестирование» будет выполняться только по запросу и решать, какую версию страницы показать. Выбор будет делаться случайно. Однако мы задействуем важную особенность А/В-тестов – смещенность (предвзятость) выбора. Если мы считаем, что тестовая страница может оказать отрицательное влияние, мы обычно настраиваем ее показ небольшому проценту пользователей.

По умолчанию тестовая страница демонстрируется 0 % пользователей. Мы будем использовать внешний механизм для увеличения этого процента.

```
import random

from pyramid import config, response

FEATURES = dict(capitalize=0.0, exclaim=0.0)

def hello_world(request):
    if random.random() < FEATURES['capitalize']:
        message = 'Hello world'
    else:
        message = 'hello world'
    if random.random() < FEATURES['exclaim']:
        message += '!'

    return response.Response(message)

with config.Configurator() as conf:
    conf.add_route('hello', '/')
    conf.add_view(hello_world, route_name='hello')
    application = conf.make_wsgi_app()
```

Рассмотрим этот код строку за строкой:

```
FEATURES = dict(capitalize=0.0, exclaim=0.0)
```

Мы определяем поддержку двух «особенностей»: `capitalize` – вывод приветствия с заглавной буквы и `exclaim` – добавление восклицательного знака в конец. Обратите внимание, что в этом примере эти две особенности реализованы как независимые, то есть пользователи могут получить четыре разных приветствия.

Это, в общем-то, хорошая имитация реального окружения, где проводится А/В-тестирование, – теоретически каждый пользователь может получить любую из 2^n возможных страниц при проведении n экспериментов.

```
if random.random() < FEATURES['capitalize']:
```

Это упрощенная реализация на Python логики так называемого «предвзятого подбрасывания монеты». В среднем она дает значение True в доле случаев, равной значению `FEATURES['capitalize']`.

```
message = 'Hello world'
```

Сообщение, начинающееся с заглавной буквы:

```
else:
```

```
message = 'hello world'
```

Сообщение, начинающееся со строчной буквы:

```
if random.random() < FEATURES['exclaim']:
    message += '!'
```

Если необходимо, добавить восклицательный знак.

Плагин для поддержки AMP

Для корректировки процентов мы используем протокол AMP. Есть много альтернативных вариантов, но этот вариант отличается сбалансированностью между гибкостью и наглядностью. И еще один немаловажный факт – поддержка AMP встроена в Twisted, поэтому нам не понадобится использовать сторонние пакеты.

```
from zope import interface

from twisted.python import usage, threadpool
from twisted import plugin
from twisted.application import service, strports
from twisted.web import wsgi, server
from twisted.internet import reactor, protocol
from twisted.protocols import amp

import pyramid_dynamic

class GetCapitalize(amp.Command):
    arguments = []
    response = [(b'value', amp.Float())]

class GetExclaim(amp.Command):
    arguments = []
    response = [(b'value', amp.Float())]

class SetCapitalize(amp.Command):
    arguments = [(b'value', amp.Float())]
    response = []

class SetExclaim(amp.Command):
    arguments = [(b'value', amp.Float())]
    response = []

class AppConfiguration(amp.CommandLocator):

    @GetCapitalize.responder
    def get_capitalize(self):
```

```
        return {'value': pyramid_dynamic.FEATURES['capitalize']]

    @GetExclaim.responder
    def get_exclaim(self):
        return {'value': pyramid_dynamic.FEATURES['exclaim']]

    @SetCapitalize.responder
    def set_capitalize(self, value):
        pyramid_dynamic.FEATURES['capitalize'] = value
        return {}

    @SetExclaim.responder
    def set_exclaim(self, value):
        pyramid_dynamic.FEATURES['exclaim'] = value
        return {}

@interface.implementer(service.IServiceMaker, plugin.IPlugin)
class ServiceMaker(object):
    tapname = "twisted_book_configure"
    description = "WSGI for book"
    class options(usage.Options):
        pass
    def makeService(self, options):
        application = pyramid_dynamic.application
        pool = threadpool.ThreadPool(minthreads=1, maxthreads=100)
        reactor.callWhenRunning(pool.start)
        reactor.addSystemEventTrigger('after', 'shutdown', pool.stop)
        root = wsgi.WSGIResource(reactor, pool, application)
        site = server.Site(root)
        control = protocol.Factory()
        control.protocol = lambda: amp.AMP(locator=AppConfiguration())
        ret = service.MultiService()
        strports.service('tcp:8000', site).setServiceParent(ret)
        strports.service('tcp:8001', control).setServiceParent(ret)
        return ret

serviceMaker = ServiceMaker()
```

Рассмотрим поближе новый для нас код:

```
class GetCapitalize(amp.Command):
    arguments = []
    response = [(b'value', amp.Float())]

class GetExclaim(amp.Command):
    arguments = []
    response = [(b'value', amp.Float())]

class SetCapitalize(amp.Command):
    arguments = [(b'value', amp.Float())]
    response = []

class SetExclaim(amp.Command):
    arguments = [(b'value', amp.Float())]
    response = []
```

Эти классы определяют *команды* AMP. В AMP команды являются основными сообщениями. Теоретически команды можно посылать в обоих направлениях, но чаще они посылаются клиентом серверу.

Мы намеренно сделали так, чтобы команды Get/Set допускали возможность чтения/записи только одного поля. Тем самым мы показываем, что атомарность этих операций не гарантируется. Сложно гарантировать атомарность при использовании словаря без применения сложных механизмов синхронизации, поэтому полезно с помощью API показать, что, например, невозможно *гарантировать* одновременное присваивание элементу 'capitalize' значения 1 и элементу 'exclaim' значения 0.

Мы могли бы написать API с претензией на атомарность, например организовав одновременное изменение обоих атрибутов. Мы могли бы даже реализовать API так, чтобы он действовал атомарно, например возвращая словарь FEATURES целиком, чтобы код получал доступ либо к старому, либо к новому словарю, и отсутствовала возможность получить промежуточный результат. Но, как бы то ни было, система может переключить поток выполнения между строками

```
if random.random() < FEATURES['capitalize']:
```

и

```
if random.random() < FEATURES['exclaim']:
```

что нарушит всю нашу атомность. Поэтому мы решили явно показать, что изменения не являются атомарными:

```
class AppConfiguration(amp.CommandLocator):
    @GetCapitalize.responder
    def get_capitalize(self):
        return {'value': pyramid_dynamic.FEATURES['capitalize']}

    @GetExclaim.responder
    def get_exclaim(self):
        return {'value': pyramid_dynamic.FEATURES['exclaim']}

    @SetCapitalize.responder
    def set_capitalize(self, value):
        pyramid_dynamic.FEATURES['capitalize'] = value
        return {}

    @SetExclaim.responder
    def set_exclaim(self, value):
        pyramid_dynamic.FEATURES['exclaim'] = value
        return {}
```

Мы написали простой класс, который служит мостом между командами чтения/записи и словарем `pyramid_dynamic.FEATURES`.

```
control = protocol.Factory()
control.protocol = lambda: amp.AMP(locator=AppConfiguration())
```

В атрибут `protocol` фабрики `control` записывается функция, которая создает новый протокол `amp.AMP` с пользовательским локатором. Существуют и другие

способы привязки протокола AMP к определенному локатору, но это открывает столь же широкие возможности перед программистом, пишущим плагины, которыми обладает тот, кто пишет код для обработки команд.

Управляющая программа

Возможно, в других местах управляющий код мог бы использовать синхронный стиль и блокировать сетевые вызовы. Но в этой книге у нас есть замечательная возможность показать, как писать клиентов с использованием Twisted. Мы решили написать этот код так, чтобы он был совместим с Python 2 и Python 3.

```
from twisted.internet import task, defer, endpoints, protocol
from twisted.protocols import amp

from twisted.plugins import twisted_book_configure

@task.react
@defer.inlineCallbacks
def main(reactor):
    endpoint = endpoints.TCP4ClientEndpoint(reactor, "127.0.0.1", 8001)
    prot = yield endpoint.connect(protocol.Factory.forProtocol(amp.AMP))
    res1 = yield prot.callRemote(twisted_book_configure.GetCapitalize)
    res2 = yield prot.callRemote(twisted_book_configure.GetExclaim)
    print(res1['value'], res2['value'])
    yield prot.callRemote(twisted_book_configure.SetCapitalize, value=0.5)
    yield prot.callRemote(twisted_book_configure.SetExclaim, value=0.5)
    res1 = yield prot.callRemote(twisted_book_configure.GetCapitalize)
    res2 = yield prot.callRemote(twisted_book_configure.GetExclaim)
    print(res1['value'], res2['value'])
```

Рассмотрим этот код поближе:

```
@task.react
```

Декоратор `react` немедленно вызовет функцию `main` с реактором в аргументе.

```
@defer.inlineCallbacks
```

Мы использовали декоратор `inlineCallbacks`, чтобы улучшить работу кода с потоками.

```
def main(reactor):
```

Обратите внимание, что здесь мы принимаем аргумент `reactor` с реактором вместо его импортирования.

```
endpoint = endpoints.TCP4ClientEndpoint(reactor, "127.0.0.1", 8001)
```

Создается клиентская конечная точка:

```
prot = yield endpoint.connect(protocol.Factory.forProtocol(amp.AMP))
```

Создается клиентская фабрика и соединение:

```
res1 = yield prot.callRemote(twisted_book_configure.GetCapitalize)
res2 = yield prot.callRemote(twisted_book_configure.GetExclaim)
```

Извлекаются значения. Обратите внимание, что здесь используются классы команд, которые были определены выше:

```
print(res1['value'], res2['value'])
```

Вывод значений перед изменением:

```
yield prot.callRemote(twisted_book_configure.SetCapitalize, value=0.5)
yield prot.callRemote(twisted_book_configure.SetExclaim, value=0.5)
```

Установка новых значений:

```
res1 = yield prot.callRemote(twisted_book_configure.GetCapitalize)
res2 = yield prot.callRemote(twisted_book_configure.GetExclaim)
print(res1['value'], res2['value'])
```

Вывод обновленных значений. Этот код позволяет убедиться, что значения изменились.

Эти три части – приложение, плагин и управляющая программа – образуют веб-сервер, внутренние параметры которого можно настраивать динамически.

Итоги

Сервер Twisted WSGI прост в установке, настройке и использовании. Фактически он даже проще базовой реализации. Несмотря на простоту использования, он идеально подходит для использования в промышленном окружении, благодаря чему стирается грань между окружениями разработки и эксплуатации – грань, которая часто затрудняет воспроизведение ошибок, возникших в промышленном окружении.

Поскольку Twisted WSGI основан на сервере Twisted Web, он автоматически наследует такие особенности, как реализация TLS промышленного уровня, поддержка SNI, Let Encrypt и протокола HTTP/2. Его также можно настроить как статический файловый веб-сервер, что позволяет обслуживать статические ресурсы, такие как изображения, файлы JavaScript и CSS, из того же процесса, что и динамическое приложение.

Он не определяет конкретного формата для файла конфигурации. Вместо этого для любой конфигурации, более глубокой, чем выбор порта или имени приложения WSGI, можно написать плагин Twisted, который обеспечит окончательную настройку на языке, знакомом всем инженерам, работающим над приложением.

Самый большой недостаток Twisted как контейнера WSGI – невозможность автоматического использования преимуществ многопроцессорных архитектур. Для его преодоления можно настроить запуск нескольких процессов WSGI. В общем случае отделение задач «как прослушивать сокет» от «как управлять несколькими процессами» позволяет найти хорошие решения для каждой из них.

Глава 6

Tahoe-LAFS: децентрализованная файловая система

Tahoe-LAFS – это распределенная система хранения, созданная в 2006 году для компании AllMyData (ныне уже не существующей), предоставлявшей услуги по хранению личных резервных копий. Перед ликвидацией компания открыла исходный код, и теперь проект развивается и поддерживается сообществом энтузиастов.

Система позволяет выгружать данные с компьютера в сеть серверов, называемую «сеткой» (grid), а затем извлекать их оттуда. Помимо создания резервных копий (например, на случай выхода из строя жесткого диска ноутбука), система предлагает также гибкие способы обмена файлами или каталогами с другими пользователями в той же сети. То есть система может также выступать в роли «сетевого диска» (SMB или NFS) или протокола передачи файлов (FTP или HTTP).

Особенностью Tahoe является «безопасность, независимая от поставщика услуги». Все файлы шифруются и хешируются локально, перед тем как покинуть ваш компьютер. Серверы, где хранятся эти файлы, видят только зашифрованное содержимое и не могут вносить необнаруживаемые изменения (из-за хеширования). Кроме того, зашифрованное содержимое кодируется в сегменты с определенной избыточностью и выгружается на несколько независимых серверов. Это означает, что данные способны пережить потерю нескольких серверов, что повышает надежность их хранения и доступность.

В результате появляется возможность выбирать серверы для хранения данных, исходя исключительно из их производительности, стоимости и времени безотказной работы, не заботясь об их безопасности. Большинство других сетевых хранилищ не обеспечивают такого уровня безопасности: злоумышленник, скомпрометировавший хостинг-провайдера, получает возможность просмотреть, изменить или полностью удалить ваши данные. В Tahoe же конфиденциальность и целостность никак не зависят от поставщика услуг хранения.

КАК РАБОТАЕТ ТАНОЕ-LAFS

«Сеть» Tahoe состоит из одного или нескольких посредников, нескольких серверов и нескольких клиентов:

- клиенты знают, как выгружать и загружать данные;
- серверы хранят зашифрованные данные;
- посредники помогают клиентам и серверам найти друг друга и вступить в контакт.

Узлы этих трех типов взаимодействуют с использованием специального протокола под названием «Foolscap», созданного на основе протокола «Perspective Broker» в Twisted, но с улучшенной безопасностью и гибкостью.

Для идентификации и доступа ко всем файлам и каталогам в Tahoe используются «мандатные ссылки» (capability strings). Это сегменты данных в кодировке base32, содержащих ключ шифрования, хеши, защищающие целостность, и информацию о местоположении. Для краткости мы используем названия «filecaps» (ссылки на файлы) и «dircaps» (ссылки на каталоги).

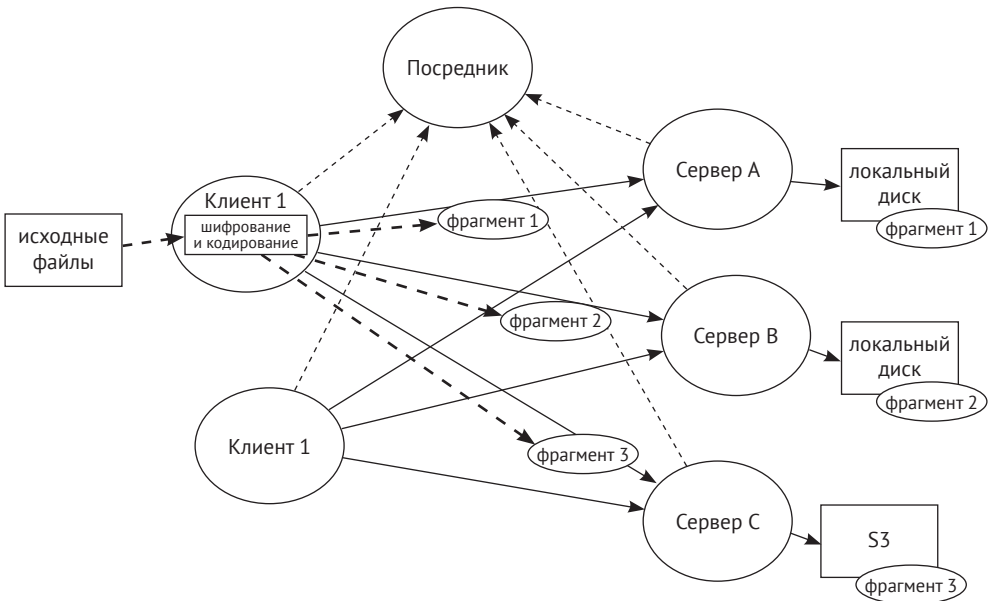


Рис. 6.1 ❖ Схема сети Tahoe-LAFS

(Примеры в этой главе сокращены для удобства чтения, но обычно длина ссылок на файлы составляет примерно 100 символов.)

Иногда ссылки могут быть трех разных видов: «writescap» (ссылка для записи) дает любому, кому она известна, возможность изменить файл, тогда как «readcap» (ссылка для чтения) позволяет только читать содержимое. Существу-

ет даже «verifysar» (ссылка для проверки), позволяющая владельцу проверить зашифрованные сегменты, хранящиеся на серверах (и сгенерировать новые, если какие-то оказались потеряны), но не позволяющая читать или изменять содержимое. Ссылки для проверки можно безопасно передать делегированному агенту восстановления, поддерживающему целостность ваших файлов, пока ваш компьютер не подключен к сети.

Самая простая команда Tahoe – PUT. Она принимает данные в незашифрованном виде, выгружает их в совершенно новый неизменяемый файл и возвращает сгенерированную ссылку на файл filecap:

```
$ tahoe put kittens.jpg
200 OK
URI:CHK:bz3lwnno6stuspq5a:mwmb5vaecnd3jz3qc:2:3:3545
```

Эта ссылка – единственный способ получить файл обратно. Вы можете записать ее на листке бумаги или сохранить в другом файле либо в каталоге Tahoe. Она – это все, что необходимо и достаточно для восстановления файла. Загрузка файла выглядит следующим образом (команда tahoe get записывает загруженные данные в стандартный вывод, поэтому мы используем синтаксис перенаправления в файл >):

```
$ tahoe get URI:CHK:bz3lwnno6stuspq5a:mwmb5vaecnd3jz3qc:2:3:3545
>downloaded.jpg
```

Мы часто (и ошибочно) называем ссылки на файлы (filecap) универсальными идентификаторами ресурсов (URI), в том числе и в самих ссылках. «CHK» означает «Content-Hash Key» – хеш-ключ содержимого, – который описывает конкретный тип кодирования неизменяемого файла: другие ссылки имеют другие идентификаторы. Ссылки на неизменяемые файлы всегда являются ссылками для чтения: никто в мире не сможет изменить файл после его загрузки, даже тот, кто выгрузил его.

Tahoe также поддерживает *изменяемые* файлы, то есть файлы, содержимое которых можно изменить позже. Для этой цели поддерживается три команды: create – создает изменяемый слот, publish – записывает новые данные в слот (перезаписывает все, что было раньше) и retrieve – возвращает текущее содержимое слота.

Изменяемые слоты имеют ссылки как для записи, так и для чтения. Команда create возвращает ссылку для записи, но любой может «понижить ее статус» до ссылки для чтения. Это позволит поделиться ссылкой с другими и оставить право записи только для себя.

Каталоги в Tahoe – это просто файлы, содержащие специально закодированную таблицу, отображающую имена дочерних файлов или каталогов в ссылки. Эти каталоги можно рассматривать как промежуточные узлы в ориентированном графе.

Создать каталог можно с помощью команды mkdir. По умолчанию она создает изменяемый каталог, но при желании можно создавать полностью за-

полненные неизменяемые каталоги. В Tahoe есть команды `cp` и `ls` для копирования файлов и получения списка содержимого каталога – они знают, как обрабатывать пути к файлам, включающие привычные нам символы слеша.

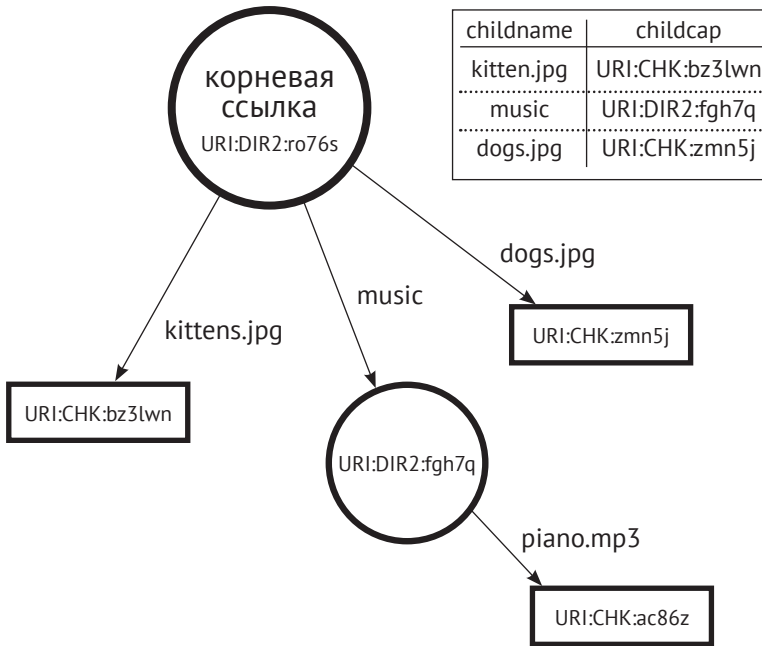


Рис. 6.2 ❖ Граф с корневой ссылкой, каталогами и файлами

Интерфейс командной строки поддерживает также «псевдонимы», которые просто хранят ссылку на корневой каталог (rootcap) в локальном файле (`~/.tahoe/private/aliases`), что позволяет в других командах сокращать ссылку на каталог до префикса, напоминающего имя сетевого диска (например, диск `E:` в Windows). Это уменьшает длину командной строки и существенно упрощает использование команд:

```

$ tahoe mkdir
URI:DIR2:ro76sdlt25ywixu25:lgxvueurtm3
$ tahoe add-alias mydrive URI:DIR2:ro76sdlt25ywixu25:lgxvueurtm3
Alias 'mydrive' added
$ tahoe cp kittens.jpg dogs.jpg mydrive:
Success: files copied
$ tahoe ls URI:DIR2:ro76sdlt25ywixu25:lgxvueurtm3
kittens.jpg
dogs.jpg
$ tahoe mkdir mydrive:music
$ tahoe cp piano.mp3 mydrive:music
$ tahoe ls mydrive:

```

```
kittens.jpg
music
dogs.jpg
$ tahoe ls mydrive:music
piano.mp3
$ tahoe cp mydrive:dogs.jpg /tmp/newdogs.jpg
$ ls /tmp
newdogs.jpg
```

Инструменты командной строки основаны на HTTP API, который мы рассмотрим позже.

АРХИТЕКТУРА СИСТЕМЫ

Клиентский узел – это долгоживущий демон шлюза, который принимает запросы на выгрузку и загрузку по «интерфейсному» протоколу. Чаще всего роль внешнего интерфейса играет HTTP-сервер, который прослушивает локальный интерфейс (127.0.0.1).

Для извлечения данных используется запрос HTTP GET, обработка которого осуществляется в несколько этапов:

- извлечение из ссылки на файл ключа дешифрования и индекса хранения;
- выяснение, что необходимо, чтобы удовлетворить запрос клиента, в том числе сегменты с данными и промежуточные узлы хеш-дерева;
- использование индекса хранения для определения, на каких серверах могут храниться сегменты запрошенного файла;
- отправка этим серверам запроса на загрузку;
- трассировка отправленных и выполненных запросов, чтобы избежать дублирования;
- трассировка времени отклика сервера, чтобы впоследствии выбирать более быстрые серверы;
- проверка и отклонение поврежденных сегментов;
- переключение на более быстрые серверы, если они доступны и после потери соединения;
- сборка сегментов в зашифрованный файл;
- расшифровывание зашифрованного файла и передача результата клиенту.

Все эти этапы выполняются циклом обработки событий, который постоянно готов к приему новых запросов `read()`, или ответов от серверов, или к обработке истечения времени ожидания, когда принимается решение попробовать послать запрос другому серверу. Этот цикл способен манипулировать десятками и сотнями одновременных соединений и таймеров, и активность в любом из них вызывает события в других. Цикл событий Twisted идеально подходит для этого.

Для передачи данных в другом направлении используются запросы HTTP PUT и POST, обработка которых включает практически те же этапы, но выполняемые в обратном порядке:

- клиентский узел принимает данные по интерфейсному протоколу и накапливает их во временном файле;
- файл хешируется с целью создать «конвергентный ключ шифрования», который также используется для устранения повторяющихся копий файлов;
- ключ шифрования хешируется с целью получить индекс хранилища;
- индекс хранилища определяет, к каким серверам можно обратиться (список серверов сортируется по-разному для каждого индекса хранилища и содержит серверы в порядке убывания приоритета);
- этим серверам отправляется запрос на выгрузку;
- если файл был выгружен ранее, сервер сообщит нам, что у него уже есть соответствующий сегмент и нам не нужно сохранять его снова;
- если сервер отклонил запрос (из-за нехватки места на диске) или ответил недостаточно быстро, выполняется попытка обратиться к другому серверу;
- производится сбор ответов, пока каждому сегменту не будет сопоставлен соответствующий сервер;
- каждый сегмент исходного файла шифруется и кодируется, на что требуется довольно много вычислительных ресурсов (по крайней мере, по сравнению с затратами на сетевые операции), поэтому эти операции выполняются в отдельном потоке, чтобы использовать преимущества параллельной обработки на нескольких ядрах;
- после шифрования и кодирования выполняется выгрузка сегментов на ранее назначенные серверы;
- когда все серверы подтвердят получение, создается окончательное хеш-дерево;
- из корня хеш-дерева и ключа шифрования создается ссылка на файл;
- возвращается ответ HTTP со ссылкой в теле.

Клиенты также реализуют другие интерфейсные протоколы, отличные от HTTP:

- FTP: передавая конфигурационный файл с именами пользователей, паролями и корневыми ссылками, клиентский узел Tahoe может выдавать себя за FTP-сервер с отдельным виртуальным каталогом для каждого пользователя;
- SFTP: то же, что и FTP, но поверх SSH;
- Magic-Folder: Dropbox-подобный инструмент двусторонней синхронизации каталогов.

Клиенты общаются с посредником на языке Foolspar и узнают у него о доступных серверах. На этом же языке Foolspar они говорят и с серверами.

Сервер-хранилище в Tahoe-LAFS может хранить сегменты на локальном диске или отправлять их в удаленную службу хранения, такую как S3 или Azure. Сервер общается с клиентами на языке Foolspar и, например, с системой S3 на языке HTTP-команд.

Узел, на котором действует сервер-хранилище, должен принимать подключения от произвольного числа клиентов, каждый из которых может одновременно посылать несколько запросов на выгрузку/загрузку сегментов. При использовании серверами удаленных хранилищ, таких как S3, каждый клиентский запрос может породить несколько запросов к S3, каждый из которых может завершиться сбоем или по тайм-ауту (и его необходимо повторить).

Все типы узлов также поддерживают HTTP-службу для проверки их состояния и управления ими. В настоящее время для этого применяется Nevow, но мы попробуем использовать встроенные средства поддержки шаблонов в Twisted (`twisted.web.template`).

КАК СИСТЕМА ТАНОЕ-LAFS ИСПОЛЬЗУЕТ TWISTED

Система Tahoe-LAFS широко использует Twisted: нам трудно представить, как ее можно было бы написать каким-то другим способом.

Приложение основано на иерархии MultiService в фреймворке Twisted, которая управляет запуском и остановкой компонентов Uploader, Downloader, IntroducerClient и др. Это позволяет запускать отдельные службы для модульного тестирования без запуска полного узла.

Самой крупной службой является Node. Она представляет клиента, сервер или посредника. Это родительский экземпляр MultiService для всего остального. Чтобы завершить службу (и дождаться прекращения сетевой активности), достаточно вызвать `stopService()` и дождаться, пока разрешится полученный экземпляр Deferred. По умолчанию узлы прослушивают выделенные временные порты и сообщают о своем местонахождении посреднику. Все состояния ограничены «базовым каталогом» узла. Это упрощает запуск нескольких клиентов/серверов в одном процессе для тестирования всей сети. Сравните эту организацию с более ранней, когда каждому серверу хранения требовалась отдельная база данных MySQL и использовались фиксированные порты TCP. В прежней организации системе было невозможно выполнить реалистичный тест без использования, как минимум, пяти отдельных компьютеров. В Tahoe пакет интеграционных тестов разворачивает сетку с 10 серверами в одном процессе, проверяет некоторую особенность, а затем останавливает серверы, и все это – за несколько секунд. Эти действия выполняются десятки раз, когда вы запускаете команду `tox` для выполнения набора тестов.

Поддержка большого числа внешних интерфейсов обеспечивается за счет мощного набора протоколов, интегрированных в Twisted. Вам не придется писать своего HTTP-клиента, сервер, FTP-сервер или SSH/SFTP-сервер: все они входят в состав Twisted.

ЧАСТО ВСТРЕЧАЮЩИЕСЯ ПРОБЛЕМЫ

Мы имеем довольно успешный опыт использования Twisted. Если бы сегодня мы вновь оказались перед выбором, то снова выбрали бы Twisted. В своей практике мы столкнулись с очень небольшим числом недостатков:

- загрузка зависимостей: некоторые пользователи (обычно упаковщики) считают, что Tahoe зависит от слишком большого количества библиотек. В течение многих лет мы старались избегать добавления новых зависимостей, потому что инструменты упаковки Python были незрелыми, но теперь есть инструмент `pip`, который делает нашу жизнь намного проще;
- упаковка/распространение: приложения на Python сложно упаковать в единственный выполняемый файл, поэтому в настоящее время пользователи должны знать о специальных инструментах для Python, таких как `pip` и `virtualenv`, чтобы установить Tahoe на свои домашние компьютеры;
- Python 3: в настоящее время Twisted отлично поддерживает Python 3, но для этого разработчикам потребовалось много лет. За это время мы добились многого и научились свободно смешивать в нашем коде машиночитаемые байты со строками, читаемыми человеком. Теперь, когда Python 3 является предпочтительной реализацией (а 2020 год – крайний срок окончания поддержки Python 2), мы прикладываем все силы, чтобы обновить наш код для работы под управлением Python 3.

Инструменты поддержки выполнения в режиме демона

Twisted предлагает удобный инструмент под названием `twistd`, который позволяет писать приложения в виде плагинов, что перекладывает на Twisted ответственность за поддержку выполнения в режиме демона (например, отсоединение от управляющего терминала `tty`, запись журналируемых сообщений в файл вместо `stdout` и, возможно, понижение привилегий после открытия привилегированных портов TCP). Когда появилась система Tahoe, ни «`pip`», ни «`virtualenv`» еще не существовало, поэтому мы создали похожие инструменты. Для объединения поддержки выполнения в режиме демона с нестандартным управлением зависимостями инструмент командной строки Tahoe предлагает подкоманды `tahoe start` и `tahoe stop`.

В настоящее время мы, вероятно, не стали бы создавать эти подкоманды и вместо этого предложили бы пользователям запускать `twistd` или `twist` (форма запуска на переднем плане). Мы также будем искать способы, чтобы вообще отказаться от выполнения в режиме демона.

В самом начале управлять демоном `twistd` было непросто, поэтому с этой целью в Tahoe использовались файлы «`.tar`». Это было наследие шаблона, который я использовал в Buildbot, первые версии которого использовали файлы «`.tar`» для записи состояния (своего рода «замороженная» копия приложения, которую можно снова разморозить, когда в следующий раз понадобится его запустить). Tahoe никогда не сохраняла туда динамическое состояние, но процесс создания узла `tahoe create-node` создавал файл `.tar` с кодом инициализации для создания и запуска нового экземпляра узла. В результате `tahoe start` превратилась в простую обертку вокруг `twistd -y node.tar`.

Для запуска узлов разных типов (клиенты, серверы, посредники и т. д.) использовались разные типы `.tar`-файлов. Это было не самое лучшее решение.

Файлы `.tar` содержали всего несколько строк: инструкцию импорта и код создания экземпляра объекта приложения `Application`. Все это ограничивало возможности реорганизации кодовой базы и изменения поведения объектов: простое переименование класса `Client` было способно нарушить работоспособность всех существующих приложений. Мы без всякой задней мысли создали общедоступный API (со всеми вытекающими отсюда проблемами совместимости), где «общедоступными» оказались все старые `.tar`-файлы, использовавшиеся в предыдущих версиях Tahoe.

Мы исправили эту проблему, заставив `tahoe start` игнорировать содержимое файла `.tar` и обращать внимание только на его имя. Большая часть конфигурации узла уже была сохранена в отдельном конфигурационном файле с именем `tahoe.cfg`, поэтому переход получился довольно простым. Когда команда `tahoe start` видит `client.tar`, она создает экземпляр `Client`, инициализирует его, используя файл конфигурации, и запускает демона.

ВНУТРЕННИЕ ИНТЕРФЕЙСЫ `FileNode`

Внутри система Tahoe определяет объекты `FileNode`, которые можно создавать из ссылок на существующие файлы или с нуля, при выгрузке данных в первый раз. Они предлагают несколько простых методов, которые скрывают все детали шифрования, создания избыточных сегментов, выбора сервера и проверки целостности. Методы загрузки определены в интерфейсе с именем `IReadable`:

```
class IReadable(Interface):
```

```
    def get_size():
        """Возвращает длину (в байтах) этого объекта, доступного для чтения."""

    def read(consumer, offset=0, size=None):
        """Загружает часть (или все) содержимого файла, делая его доступным
        для заданного экземпляра IConsumer. Возвращает Deferred, который
        разрешается (экземпляром consumer), когда consumer не зарегистрирован
        (потому что ему передан последний байт, или consumer возбудил
        исключение в вызове write(), или он отказался от приема следующих
        данных). Загруженный сегмент начинается с позиции 'offset' и
        содержит 'size' байт (или остаток до конца файла, если size==None). """
```

Twisted использует `zope.interface` для классов, которые поддерживают определение `Interface` (в действительности `Interface` – это `zope.interface.Interface`). Мы используем данное обстоятельство как форму проверки типа: интерфейс может утверждать, что читаемый объект является провайдером `IReadable`. Существует несколько видов узлов `FileNode`, но все они реализуют интерфейс `IReadable`, а интерфейсный код использует только методы, имеющиеся в этом интерфейсе.

Интерфейс `read()` не возвращает данные непосредственно: вместо этого он принимает «потребителя», которому следует передавать данные по мере их

поступления. При этом для потоковой передачи данных без излишней буферизации используется система производитель/потребитель, реализованная в Twisted (и описанная в главе 1). Это позволяет Tahoe передавать файлы объемом в несколько гигабайт без использования гигабайтов памяти.

Объекты `DirectoryNode` создаются аналогично. Они имеют методы (определены в `IDirectoryNode`) для перечисления своих потомков и для перехода по дочерним ссылкам (по имени) к каким-то другим узлам. Изменяемые каталоги включают методы для добавления или замены потомка по имени.

```
class IDirectoryNode(IFileSystemNode):
    """Представляет узел файловой системы, играющий роль контейнера и
    поддерживающий отображение имен в дочерние узлы. Служит аналогом
    каталога в Tahoe. Имена потомков определяются строками Юникода,
    и все потомки реализуют некоторый интерфейс IFileSystemNode
    (файл, подкаталог или неизвестный узел).
    """

    def list():
        """Возвращает Deferred, который разрешается словарем, отображающим
        имя потомка (строка Юникода) в кортеж (node, metadata_dict), где
        'node' -- экземпляр IFileSystemNode, а 'metadata_dict' -- словарь
        с метаданными."""

    def get(name):
        """Возвращает Deferred, который разрешается узлом потомка с указанным
        именем name, реализующим IFileSystemNode. Имя потомка должно
        быть строкой Юникода.
        Возбуждает исключение NoSuchChildError, если потомок с указанным
        именем не найден."""
```

Обратите внимание, что эти методы возвращают экземпляры `Deferred`. Каталоги хранятся в файлах, файлы – в сегментах, а сегменты – на серверах. Заранее неизвестно, которые из серверов ответят на запрос загрузки, поэтому мы используем `Deferred`, чтобы «дождаться», когда данные станут доступны.

Этот граф объектов узлов используется каждым интерфейсным протоколом.

ИНТЕГРАЦИЯ ИНТЕРФЕЙСНЫХ ПРОТОКОЛОВ

Чтобы понять, как Tahoe использует преимущества отдельной поддержки протоколов в Twisted, рассмотрим несколько «интерфейсных (front-end) протоколов». Они обеспечивают мост между внешними программами и внутренними интерфейсами `IFileNode`/`IDirectoryNode`/`IReadable`.

Все обработчики протоколов используют внутренний объект с именем `Client`, наиболее важным методом которого является `create_node_from_uri`. Он принимает ссылку на файл или каталог (строку) и возвращает соответствующий объект `FileNode` или `DirectoryNode`. Соответственно, вызывающая сторона может использовать свои методы для чтения или изменения распределенного файла.

ВЕБ-ИНТЕРФЕЙС

Клиентский демон Tahoe-LAFS поддерживает локальную HTTP-службу для управления большинством своих операций. В ее состав входит веб-приложение для просмотра файлов и папок («WUI»: Web User Interface – веб-интерфейс пользователя) и машинно ориентированный интерфейс управления («WAPI»: Web Application Programming Interface – программный интерфейс для веб-приложений), которые мы ласково называем «wooeу» («вуи») и «warру» («вапи»).

Оба реализованы на основе встроенного сервера `twisted.web`. Иерархия объектов-ресурсов направляет запросы некоторому листовому узлу, который реализует методы, такие как `gender_GET`, для обработки деталей запроса и конструирования ответа. По умолчанию прослушивается порт 3456, но его можно изменить в файле `tahoe.cfg`, определив другой дескриптор конечной точки.

Фактически Tahoe использует проект «Nevow», реализующий обертку вокруг `twisted.web`, но в настоящее время встроенная функциональность Twisted достигла достаточно высокого уровня, поэтому мы постепенно удаляем код, использующий Nevow.

Самый простой вызов WAPI – GET, который извлекает файл. HTTP-клиент отправляет ссылку на файл, Tahoe преобразует ее в `FileNode`, загружает содержимое и возвращает данные в HTTP-ответе. Запрос выглядит так:

```
curl -X GET http://127.0.0.1:3456/uri/URI:CHK:bz3lwno6stus:mwmb5vae...
```

В результате получается `twisted.web.http.Request` с массивом «путей», включающим два элемента: строку «uri» и ссылку на файл. Веб-сервер Twisted запускает корневой ресурс, к которому вы можете подключить обработчики для разных имен. Наш корневой ресурс `Root` создается с помощью объекта `Client`, описанного выше, и снабжается обработчиком имени `uri`:

```
from twisted.web.resource import Resource
class Root(Resource):
    def __init__(self, client):
        ...
        self.putChild("uri", URIHandler(client))
```

Все запросы, начинающиеся с `uri/`, передаются этому ресурсу `URIHandler`. Если запросы имеют дополнительные компоненты пути (то есть ссылку на файл), для них вызывается метод `GetChild`, который отвечает за поиск ресурса, способного обработать запрос. Мы создаем `FileNode` или `DirectoryNode` из данной ссылки на файл/каталог и затем обертываем его конкретным объектом веб-обработчика, который знает, как обращаться с HTTP-запросами:

```
class URIHandler(Resource):
    def __init__(self, client):
        self.client = client

    def getChild(self, path, request):
        # 'path', как предполагается, -- это ссылка на файл или каталог
```

```

try:
    node = self.client.create_node_from_uri(path)
    return directory.make_handler_for(node, self.client)
except (TypeError, AssertionError):
    raise WebError("%s is not a valid file- or directory- cap" % name)

```

node – это объект `FileNode`, который оборачивает ссылку на файл из запроса GET. Обработчик определяется вспомогательной функцией, которая проверяет доступные интерфейсы узла и решает, какую обертку создать:

```

def make_handler_for(node, client, parentnode=None, name=None):
    if parentnode:
        assert IDirectoryNode.providedBy(parentnode)
    if IFileNode.providedBy(node):
        return FileNodeHandler(client, node, parentnode, name)
    if IDirectoryNode.providedBy(node):
        return DirectoryNodeHandler(client, node, parentnode, name)

    return UnknownNodeHandler(client, node, parentnode, name)

```

В нашем примере она вернет `FileNodeHandler`. Этот обработчик имеет множество параметров, и фактический код в `web/filenode.py` выглядит совсем иначе, но в упрощенном виде его можно представить так:

```

class FileNodeHandler(Resource):
    def __init__(self, client, node, parentnode=None, name=None):
        self.node = node
        ...

    @inlineCallbacks
    def render_GET(self, request):
        version = yield self.node.get_best_readable_version()
        filesize = version.get_size()
        first, size, contentsize = 0, None, filesize
        ... # эти значения будут изменены в соответствии
            # с заголовком Range, если он есть
        request.setHeader("content-length", b"%d" % contentsize)
        yield version.read(request, first, size)

```

Встроенный веб-сервер Twisted, в отличие от Nevow, не позволяет объектам `Resource` возвращать `Deferred`, что довольно удобно. Вот краткое описание происходящего:

- сначала мы запрашиваем у `FileNode` его лучшую версию для чтения. Этого не требуется для неизменяемых файлов (для них существует только одна версия), но изменяемые файлы могут иметь несколько версий в сетке. «Лучшая» означает самая свежая. Мы возвращаем объект «версии», реализующий интерфейс `IReadable`;
- затем вычисляем размер файла. Для неизменяемых файлов размер «вшит» в ссылку на файл, поэтому метод `get_size()` возвращает его немедленно. Размер изменяемых файлов определяется после получения объекта версии;

- используем размер файла и заголовок Range (если имеется), чтобы выяснить смещение и объем данных для чтения;
- устанавливаем заголовок Content-Length, чтобы сообщить HTTP-клиенту, какой объем данных он должен ожидать;
- запускаем загрузку вызовом метода `IReadable.read()`. Объект `Request` также является экземпляром `IConsumer`, поэтому код загрузки создает `IProducer` для подключения к нему. В ответ возвращается `Deferred`, который разрешится после доставки последнего байта из файла потребителю;
- когда разрешится последний экземпляр `Deferred`, сервер будет знать, что может закрыть TCP-соединение, или повторно инициализировать его для следующего запроса.

Здесь мы опустили множество деталей, которые раскроем ниже.

Типы файлов, Content-Type, /named/

Модель хранения в Tahoe отображает ссылки на файлы в строки байтов, без имен, дат и других метаданных. *Каталоги* содержат имена и даты в записях, указывающих на дочерние элементы, но базовая ссылка на файл просто дает набор байтов.

Однако каждый HTTP-запрос включает заголовок `Content-Type`, что позволяет браузеру выяснить, как отобразить полученную страницу (HTML, JPG или PNG) или какие метаданные записать при сохранении на диск. Кроме того, большинство браузеров предполагают, что последним компонентом в пути URL является имя файла, и функция «сохранить на диск» использует его в качестве имени файла по умолчанию.

Чтобы обработать это несоответствие, в WAPI Tahoe есть функция, позволяющая загрузить ссылку на файл с произвольным именем в последнем элементе пути. Браузер каталогов WUI помещает эти специальные URL-адреса в разметку HTML-страницы с содержимым каталога, поэтому команда **Сохранить ссылку как...** работает правильно. Вот как выглядит полный URL:

```
http://127.0.0.1:3456/named/URI:CHK:bz3lwno6stus:mwmb5vae../kittens.jpg
```

Очень похоже на каталог с дочерним элементом внутри. Чтобы избежать визуальной путаницы, мы обычно вставляем дополнительную строку в такие URL:

```
http://127.0.0.1:3456/named/URI:CHK:bz3lwn../@@named=/kittens.jpg
```

Это добавление реализовано с помощью ресурса `Named`, который создает-ся с использованием `FileNodeHandler` и дополнительно запоминает последний компонент пути в URL в атрибуте `self.filename` (при этом игнорируются любые промежуточные компоненты, такие как `@@named=string`). Затем, вызывая `render_GET`, мы передаем это имя файла в утилиту `Twisted`, которая отображает расширение из имени файла в строку типа, используя эквивалент `/etc/mime.types`. Благодаря этому мы можем установить заголовки `Content-Type` и `Content-Encoding`.

```
# from twisted.web import static
ctype, encoding = static.getTypeAndEncoding(
    self.filename,
    static.File.contentTypes,
    static.File.contentEncodings,
    defaultType="text/plain")
request.setHeader("content-type", ctype)
if encoding:
    request.setHeader("content-encoding", encoding)
```

Сохранение на диск

Когда пользователь щелкает на ссылке, браузер пытается отобразить полученный документ: разметка HTML пропускается через механизм шаблонов, изображения выводятся в окне, аудиофайлы воспроизводятся и т. д. Если тип файла не распознается, браузер предложит сохранить файл на диск. HTML-интерфейс «WUI» предлагает возможность принудительного сохранения на диск: для любого URL-адреса, ссылающегося на файл, нужно просто добавить в URL аргумент запроса `?save=True`. Обнаружив этот аргумент, веб-сервер добавит заголовок `Content-Disposition`, указывающий браузеру, что тот всегда должен сохранять ответ, не пытаясь его обработать:

```
if boolean_of_arg(get_arg(request, "save", "False")):
    request.setHeader("content-disposition",
        'attachment; filename="%s"' % self.filename)
```

Заголовки Range

Веб-интерфейс позволяет клиентам HTTP запрашивать только часть файла, добавляя в запрос заголовок `Range`. Этот прием часто используется потоковыми мультимедийными проигрывателями (такими как VLC или iTunes), когда для перемещения по фильму или аудиофайлу используется элемент управления «ползунок». Схема кодирования в `Tacho` предлагает эффективную поддержку такого рода произвольного доступа с использованием хеш-деревьев Меркла.

Создание хеш-дерева Меркла начинается с разбивки данных на сегменты и применения к каждому криптографической хеш-функции (SHA256). Затем создается второй слой, в который включаются хеши каждой пары хешей сегментов (его длина равна половине длины первого слоя). Процесс свертки повторяется до тех пор, пока не останется единственный «корневой хеш» в верхней части бинарного дерева, состоящего из промежуточных хеш-узлов и хешей сегментов в нижнем слое. Корневой хеш хранится в ссылке на файл, а все остальное (промежуточные хеши и хеши сегментов) отправляется на сервер в теле запроса. В процессе извлечения любой отдельный сегмент можно проверить по имеющемуся корню без загрузки остальных сегментов, запрашивая у сервера сопутствующие хеш-узлы, лежащие на пути от этого сегмента к корню. Это позволяет быстро проверить произвольные сегменты с минимальной передачей данных.

Веб-интерфейс анализирует заголовок Range, устанавливает заголовки Content-Range и Content-Length в ответе и изменяет значения first и size, которые мы передаем в метод read().

Парсинг заголовка Range – непростая задача, потому что он может включать список (потенциально перекрывающихся) диапазонов, которые, в свою очередь, могут включать начало или конец файла и выражаться в различных единицах (не только в байтах). К счастью, серверам разрешено игнорировать заголовки Range, которые не получается разобрать: хоть это и неэффективно, зато они могут просто вернуть весь файл, как если бы заголовка Range не существовало, а клиент сможет игнорировать те части данных, которые ему не нужны.

```
first, size, contentsize = 0, None, filesize
request.setHeader("accept-ranges", "bytes")

rangeheader = request.getHeader('range')
if rangeheader:
    ranges = self.parse_range_header(rangeheader)

    # ranges = None означает, что заголовок не удалось разобрать и
    # его можно игнорировать, как если бы его вообще не было. Если
    # диапазонов больше одного, то просто возвращается первый.
    # В настоящее время мы пока не генерируем multipart/byteranges.
    if ranges is not None:
        first, last = ranges[0]

        if first >= filesize:
            raise WebError('First beyond end of file',
                           http.REQUESTED_RANGE_NOT_SATISFIABLE)

        else:
            first = max(0, first)
            last = min(filesize-1, last)

            request.setResponseCode(http.PARTIAL_CONTENT)
            request.setHeader('content-range', "bytes %s-%s/%s" %
                             (str(first), str(last),
                              str(filesize)))

            contentsize = last - first + 1
            size = contentsize

request.setHeader("content-length", b"%d" % contentsize)
```

Преобразование ошибок на возвращающей стороне

Внутренний API Tahoe генерирует множество исключений, когда что-то идет не так. Например, если из строя вышло слишком много серверов, попытка восстановить файл может не увенчаться успехом (по крайней мере, пока некоторые серверы не вернутся в режим нормальной работы). Мы постарались преобразовать эти исключения в более или менее понятные коды ошибок HTTP с помощью обработчика исключений, который выполняется в конце цепочки обработки HTTP. Ядром обработчика является метод `humanize_failure()`. Он про-

сматривает объект `twisted.python.failure.Failure`, которым обертываются все исключения, возникшие в `Deferred`:

```
def humanize_failure(f):
    # вернуть текст, responsecode
    if f.check(EmptyPathnameComponentError):
        return ("The webapi does not allow empty pathname components, "
                "i.e. a double slash" , http.BAD_REQUEST)
    if f.check(ExistingChildError):
        return ("There was already a child by that name, and you asked me "
                "to not replace it." , http.CONFLICT)
    if f.check(NoSuchChildError):
        quoted_name = quote_output(f.value.args[0], encoding="utf-8")
        return ("No such child: %s" % quoted_name, http.NOT_FOUND)
    if f.check(NotEnoughSharesError):
        t = ("NotEnoughSharesError: This indicates that some "
            "servers were unavailable, or that shares have been "
            "lost to server departure, hard drive failure, or disk "
            "corruption. You should perform a filecheck on "
            "this object to learn more.\n\nThe full error message is:\n"
            "%s" ) % str(f.value)
        return (t, http.GONE)
    ...
```

Первая половина возвращаемого значения – это строка для добавления в тело HTTP-ответа; вторая – собственно код ошибки HTTP.

Отображение элементов пользовательского интерфейса: шаблоны Nevow

Таое WUI предлагает интерфейс менеджера файлов: панели каталогов, списки файлов, кнопки загрузки/выгрузки/удаления и т. д. Все эти элементы реализованы в виде разметки HTML, которая создается на стороне сервера из шаблонов Nevow.

Каталог `web/` содержит файл XHTML для каждой страницы с заполнителями, которые заполняются классом `DirectoryNodeHandler`. Каждый заполнитель – это XML-элемент, имя которого начинается с пространства имен «slot». Вот как выглядит шаблон списка каталогов:

```
<table class="tahoe-directory" n:render="sequence" n:data="children" >
  <tr n:pattern="header">
    <th>Type</th>
    <th>Filename</th>
    <th>Size</th>
  </tr>
  <tr n:pattern="item" n:render="row" >
    <td><n:slot name="type"/></td>
    <td><n:slot name="filename"/></td>
    <td align="right"><n:slot name="size"/></td>
  </tr>
```

Код, заполняющий эту форму, находится в файле `directory.py`. Он выполняет обход всех дочерних элементов отображаемого каталога, проверяет их тип и использует объект контекста `ctx` для заполнения каждого слота по имени. Для файлов тег `T.a`, поддерживаемый механизмом шаблонов `Nevow`, создает гиперссылку с атрибутом `href=`, содержащим URL для загрузки с использованием префикса `/named/`, описанного выше:

```
...
elif IImmutableFileNode.providedBy(target):
    dlurl = "%s/named/%s/@@named=%s"%(root, quoted_uri, nameurl)
    ctx.fillSlots("filename", T.a(href=dlurl, rel="noreferrer")[name])
    ctx.fillSlots("type", "FILE")
    ctx.fillSlots("size", target.get_size())
```

`Nevow` также предлагает инструменты для создания HTML-форм. Они используются для создания формы выбора файла для загрузки и элемента ввода имени «создать каталог».

ИНТЕРФЕЙС FTP

Внешние протоколы позволяют другим приложениям обращаться к внутреннему графу файлов в той форме, которая соответствует их модели данных. Например, интерфейс FTP связывает каждую «учетную запись» (пару имя пользователя/пароль) со ссылкой на корневой каталог. Когда FTP-клиент подключается к этой учетной записи, ему предоставляется файловая система с корнем в этом узле каталога и простирается только вниз (в дочерние файлы и подкаталоги). На обычном FTP-сервере все учетные записи видят одну и ту же файловую систему, но имеют разные разрешения (Алиса не может читать файлы Боба) и разные начальные каталоги (для Алисы открывается каталог `/home/alice`, а для Боба – каталог `/home/bob`). На FTP-сервере Tahoe Алиса и Боб будут видеть совершенно разные файловые системы, которые могут вообще не пересекаться (если они не договорились о совместном использовании некоторой части своего пространства).

FTP-интерфейс Tahoe построен на основе FTP-сервера `Twisted` (`twisted.protocols.ftp`). Для управления учетными записями (включая «порталы», «области» и «аватары») этот FTP-сервер использует фреймворк «Cred», входящий в состав `Twisted`. В результате сервер состоит из нескольких компонентов:

- конечная точка: определяет TCP-порт для сервера и другие параметры, например задающие используемые сетевые интерфейсы (сервер, к примеру, можно ограничить прослушиванием только интерфейса `127.0.0.1`);
- `FTPFactory` (`twisted.protocols.ftp.FTPFactory`): обобщенный FTP-сервер. Это «фабрика протокола», которая вызывается каждый раз, когда подключается новый клиент, и отвечает за создание экземпляра протокола, управляющего этим конкретным соединением. Запуская конечную точку, вы должны передать ей фабричный объект;

- объект проверки, реализующий интерфейс `ICredentialsChecker` и осуществляющий аутентификацию. Он проверяет некоторые учетные данные и (в случае успеха) возвращает «идентификатор аватара». В протоколе FTP учетными данными являются имя пользователя и пароль. В SFTP – открытый ключ SSH. «Идентификатор аватара» – это просто имя пользователя. Внешний интерфейс FTP в Tahoe можно настроить на использование `AccountFileChecker` (в `auth.py`), который хранит соответствия имя пользователя/пароль/корневая ссылка в локальном файле. Он также может использовать `AccountURLChecker`, который обращается к HTTP-серверу (принимающему имя пользователя и пароль и возвращающему корневую ссылку). `AccountURLChecker` использовался для централизованного управления учетными записями еще в `AllMyData`;
- аватар: объект на стороне сервера, который представляет конкретного пользователя. Он также зависит от типа службы, поэтому должен реализовать некоторый конкретный интерфейс, в данном случае интерфейс с именем `IFTPShell` (определяющий такие методы, как `makeDirectory`, `stat`, `list` и `openForReading`);
- область (`realm`): любой объект, реализующий интерфейс `IRealm` и преобразующий идентификатор аватара в аватар. Этот объект также должен поддерживать несколько интерфейсов: клиент, которому требуется определенный вид доступа, может запросить определенный интерфейс, а объект области должен вернуть другой аватар в зависимости от запрошенного интерфейса. В Tahoe FTP область – это класс `Dispatcher`, который знает, как создать узел корневого каталога из информации об учетной записи и обернуть его;
- `Portal` (`twisted.cred.portal.Portal`): объект, который управляет объектами проверки и области. Экземпляр `Portal` передается конструктору `FTPFactory`, и ему делегируются все операции, связанные с авторизацией;
- `Handler` (`allmydata.frontends.ftpd.Handler`): объект, реализующий `IFTPShell` и транслирующий понятия FTP в понятия Tahoe.

Сервер Tahoe FTP выполняет следующие операции:

- создает экземпляр `MultiService`, который добавляется к узлу Node верхнего уровня;
- создает приемник вызовом `strports.service` для прослушивания порта FTP;
- настраивает этот приемник с помощью `FTPFactory`;
- настраивает фабрику с помощью `Portal`;
- создает экземпляр `Dispatcher` для использования в роли «области» в экземпляре `Portal`;
- добавляет в `Portal` экземпляр `AccountFileChecker` и/или `AccountURLChecker`.

Когда клиент подключается к FTP-серверу, имя пользователя и пароль передаются в `AccountFileChecker`, который хранит в памяти содержимое из файла с учетными записями. Поиск учетной записи реализован очень просто:

```

class FTPAvatarID:
    def __init__(self, username, rootcap):
        self.username = username
        self.rootcap = rootcap

@implementer(checkers.ICredentialsChecker)
class AccountFileChecker(object):
    def requestAvatarId(self, creds):
        if creds.IUsernamePassword.providedBy(creds):
            return self._checkPassword(creds)
        ...

    def _checkPassword(self, creds):
        try:
            correct = self.passwords[creds.username]
        except KeyError:
            return defer.fail(error.UnauthorizedLogin())

        d = defer.maybeDeferred(creds.checkPassword, correct)
        d.addCallback(self._cbPasswordMatch, str(creds.username))
        return d

    def _cbPasswordMatch(self, matched, username):
        if matched:
            return self._avatarId(username)
        raise error.UnauthorizedLogin

    def _avatarId(self, username):
        return FTPAvatarID(username, self.rootcaps[username])

```

Если имя пользователя отсутствует в списке или пароль не совпадает с хранящимся в файле, то `requestAvatarId` вернет экземпляр `Deferred` с исключением `UnauthorizedLogin`, и `FTPFactory` вернет соответствующий код ошибки FTP. Если проверка прошла успешно, возвращается объект `FTPAvatarID`, содержащий имя пользователя и URI корневой ссылки (которая является обычной строкой).

В случае успеха `Portal` обращается к своей области (экземпляра `Dispatcher`), чтобы преобразовать идентификатор аватара в обработчик. Объект области тоже имеет довольно простую реализацию:

```

@implementer(portal.IRealm)
class Dispatcher(object):
    def __init__(self, client):
        self.client = client

    def requestAvatar(self, avatarID, mind, interface):
        assert interface == ftp.IFTPShell
        rootnode = self.client.create_node_from_uri(avatarID.rootcap)
        convergence = self.client.convergence
        s = Handler(self.client, rootnode, avatarID.username, convergence)
        def logout(): pass
        return (interface, s, None)

```

Сначала мы убеждаемся, что запрошен интерфейс `IFTPShell`, а не какой-то другой (с которым мы не знаем, что делать). Затем используем API графа фай-

лов в Tahoe для преобразования URI корневой ссылки в узел каталога. Обсуждение «конвергенции» выходит за рамки данной главы, но отметим, что этот механизм обеспечивает безопасное удаление дубликатов и предоставляется обработчику Handler, чтобы дать возможность расширения интерфейса для каждой учетной записи.

Затем создаем обработчик Handler, обертывающий клиента (предоставляющего методы для создания новых файловых узлов) и корневой узел (обеспечивающий доступ к «домашнему каталогу» пользователя и его содержимому), и возвращаем его порталу. Этого достаточно, чтобы дать возможность подключаться к FTP-серверу.

Позднее, когда клиент выполнит команду «ls», будет вызван метод обработчика list(). Наша реализация отвечает за трансляцию FTP-понятия вывода списка с содержимым каталога в Tacho-понятие пошагового обхода узлов от корневого каталога до некоторого другого узла каталога.

```
def list(self, path, keys=()):
    d = self._get_node_and_metadata_for_path(path)
    def _list((node, metadata)):
        if IDirectoryNode.providedBy(node):
            return node.list()
        return { path[-1]: (node, metadata) }
    d.addCallback(_list)

    def _render(children):
        results = []
        for (name, childnode) in children.iteritems():
            results.append( (name.encode("utf-8"),
                           self._populate_row(keys, childnode) ) )
        return results

    d.addCallback(_render)
    d.addErrback(self._convert_error)

    return d
```

Сначала вызывается вспомогательный метод «следующий по указанному пути от корня». Он возвращает Deferred, который в конечном итоге разрешается узлом и метаданными для файла или каталога, указанного в пути (например, для пути foo/bar у корневого каталога будет запрошен вложенный в него каталог foo, а затем у этого подкаталога будет запрошен его дочерний элемент bar). Если путь указывает на каталог, вызывается метод node.list() интерфейса IDirectoryNode, чтобы получить дочерние элементы: этот метод возвращает словарь, отображающий имена дочерних элементов в кортежи (дочерний узел, метаданные). Если путь указывает на файл, мы действуем так, будто он указывает на каталог с единственным файлом.

Далее этот словарь нужно преобразовать в нечто, что сможет принять FTP-сервер. Команда LIST в протоколе FTP может запрашивать разные атрибуты: иногда клиенту нужны имена владельцев/групп, иногда разрешения, а бывает,

его интересует только список имен дочерних элементов. Интерфейс IFTPShell в Twisted выражает это, передавая методу list() последовательность «ключей» (строк), указывая в них, какие атрибуты необходимы. Наш метод _populate_row() превращает одну пару потомок+метаданные в список значений.

```
def _populate_row(self, keys, (childnode, metadata)):
    values = []
    isdir = bool(IDirectoryNode.providedBy(childnode))
    for key in keys:
        if key == "size":
            if isdir:
                value = 0
            else:
                value = childnode.get_size() or 0
        elif key == "directory":
            value = isdir
        elif key == "permissions":
            value = IntishPermissions(0600)
        elif key == "hardlinks":
            value = 1
        elif key == "modified":
            if "linkmtime" in metadata.get("tahoe", {}):
                value = metadata["tahoe"]["linkmtime"]
            else:
                value = metadata.get("mtime", 0)
        elif key == "owner":
            value = self.username
        elif key == "group":
            value = self.username
        else:
            value = "???"
    values.append(value)

    return values
```

Для каждого ключа мы добавляем в список значение, извлекаемое с помощью интерфейса IFileNode или IDirectoryNode. В большинстве случаев все сводится к простому поиску в метаданных или вызову метода объекта узла. Но есть один необычный случай – разрешения (permissions): подробнее о нем рассказывается ниже.

Последний шаг – присоединить обработчик ошибок _convert_error. Этот метод преобразует некоторые ошибки Tahoe в их ближайшие FTP-эквиваленты, что более полезно, чем сообщение «внутренняя ошибка сервера», которую получит клиент без такого преобразования.

```
def _convert_error(self, f):
    if f.check(NoSuchChildError):
        childname = f.value.args[0].encode("utf-8")
        msg = "'%s' doesn't exist" % childname
        raise ftp.FileNotFoundError(msg)
    if f.check(ExistingChildError):
```

```

msg = f.value.args[0].encode("utf-8")
raise ftp.FileExistsError(msg)

return f

```

ИНТЕРФЕЙС SFTP

SFTP – это протокол передачи файлов, основанный на слое шифрования защищенной командной оболочки SSH. Он предоставляет POSIX-подобный API для удаленных клиентов: операции `open`, `seek`, `read` и `write`, выполняемые с дескрипторами файлов. Протокол FTP, напротив, предлагает только передачу отдельных файлов в стиле «все или ничего». FTP гораздо лучше подходит для модели представления файлов в Tahoe, но SFTP безопаснее при работе с удаленными серверами.

Преимущество использования фреймворка Cred заключается в том, что он позволяет использовать один и тот же механизм аутентификации с разными протоколами. FTP и SFTP, несмотря на их различия, используют одну и ту же базовую модель доступа: клиенты идентифицируются по некоторым учетным данным и получают доступ к определенному домашнему каталогу. В Tahoe оба протокола, FTP и SFTP, используют одинаковые классы `FTPAvatarID` и `AccountFileChecker`, упоминавшиеся выше. `AccountFileChecker` определяет «интерфейсы учетных записей», охватывающие все возможные виды аутентификации: `IUsernamePassword`, `IUsernameHashedPassword` и `ISSHPrivateKey` (последний используется с протоколом SFTP и позволяет идентифицировать пользователей по открытому ключу SSH вместо пароля).

Они различаются только реализацией областей (класс `Dispatcher`), возвращающей разные типы обработчиков для двух протоколов.

ОБРАТНАЯ НЕСОВМЕСТИМОСТЬ TWISTED API

Система Tahoe не поддерживает таких понятий, как списки управления доступом (`Access Control List`, `ACL`), имена пользователей или биты разрешений на чтение/запись/выполнение: она следует принципу возможностей объектов: «если вы можете сослаться на объект, значит, вы можете использовать его». Ссылки на файлы невозможно угадать, поэтому единственный способ сослаться на файл – узнать ссылку на него у того, кто этот файл выгрузил, или у того, кто узнал ее у первоначального владельца файла.

Большинство файлов хранится в каталогах, поэтому управление доступом осуществляется через обход структуры каталогов, что довольно безопасно, потому что каталоги в Tahoe не имеют ссылок на «родителей». Вы можете поделиться ссылкой на один из своих каталогов, просто передав ее: получивший эту ссылку не сможет с ее помощью перейти на уровень выше в иерархии каталогов.

Поэтому FTP-сервер всегда возвращает в поле «permissions» (разрешения) значение «0600», означающее «разрешены чтение и запись только текущему

пользователю». Во многом это значение является косметическим: клиенты FTP используют его только для заполнения столбца «mode» (режим) в подробном выводе (`ls -l`) списка каталогов. Мы могли бы добавить точности и возвращать значение «0400» для неизменяемых объектов, но нас такая точность мало заботила в свое время и мы так и не удосужились внести соответствующие изменения в код.

Однако даже статическое значение превратилось в источник проблем, когда неожиданно изменился один из API в Twisted. В начале своего развития фреймворк Twisted использовал целые числа для представления режимов/разрешений файлов (в точности как в ядре Unix и большинстве программ на C). Однако впоследствии был сделан вывод, что такое сходство с Unix является чрезмерным, и в Twisted-11.1.0 для хранения подобной информации в виде наборов логических значений был создан специальный класс `filepath.Permissions`.

Но поддержка этого класса была добавлена в FTP-сервер намного позже. До версии Twisted 14.0.2 функция `list()` ожидала получить «разрешения» в виде целого числа. Начиная с Twisted-15.0.0 эта функция ожидает получить экземпляр `Permissions`. Более того, она принимает *только* экземпляр `Permissions`: целое число вызовет исключение.

По сути, интерфейс `IFTPShell` существенно изменился между версиями 14.0.2 и 15.0.0, что вызвало поток отчетов об ошибках, возникавших в FTP-команде `ls` у тех, кто обновил версию Twisted (этот интерфейс не был охвачен сквозными тестами, а в Twisted 14.0.2 все еще использовалось ручное тестирование, из-за чего мы не заметили проблему).

Обычно смена устаревшего API проводится в Twisted поэтапно, на протяжении нескольких выпусков, и только потом вносятся несовместимые изменения, но в данном случае мы недосмотрели, возможно, потому что наиболее распространенной реализацией `IFTPShell` является встроенный класс `FTPShell`, который был обновлен в то же время. То есть еще одна причина проблемы состояла в том, что `IFTPShell` был изменен без выдерживания периода устаревания, как если бы это был внутренний API, но на самом деле он был общедоступным.

Самое простое решение этой проблемы – добавить в `setup.py` системы Tahoe требование Twisted `>= 15.0.0` и изменить код, чтобы он возвращал объект `Permissions`. Но это усложнило бы жизнь людям, использующим Tahoe в дистрибутивах Linux, включающих устаревшую версию Twisted. (Дистрибутив Debian 8.0 «jessie» вышел в 2015 году с Twisted-14.0.2 и не заменял ее до 2017 года.) В то время разработчики Tahoe пытались обеспечить совместимость с широким спектром версий Twisted. Мы считали неправильным для пользователей обновить свою систему Twisted, только чтобы удовлетворить требования Tahoe.

Поэтому, чтобы обеспечить работоспособность Tahoe при использовании и старых, и новых версий Twisted, мы должны были вернуть что-то, что при необходимости могло бы вести себя как целое число и как экземпляр `Permissions`. Исследовав, как в Twisted 14.0.2 используется значение, мы обнаружи-

ли, что к нему всегда применялись поразрядные операции AND (И) в процессе форматирования:

```
# twisted-14.0.2: twisted/protocols/ftp.py line 428
def formatMode(mode):
    return ''.join([mode&(256>>n) and 'rwx'[n % 3] or '-' for n in range(9)])
```

Это позволило нам создать вспомогательный класс, наследующий `Permissions`, но переопределяющий метод `and`, чтобы он возвращал целое число, если вызов выполнялся из старой версии Twisted:

```
# filepath.Permissions был добавлен в Twisted-11.1.0.
# Twisted <15.0.0 ожидает значения int,
# но использует его только в операции '&'.
# Twisted >=15.0.0 ожидает filepath.Permissions.
# Следующий класс удовлетворяет обоим требованиям.
class IntishPermissions(filepath.Permissions):
    def __init__(self, statModeInt):
        self._tahoe_statModeInt = statModeInt
        filepath.Permissions.__init__(self, statModeInt)

    def __and__(self, other):
        return self._tahoe_statModeInt&other
```

В настоящее время ситуация иная. Мы больше не советуем пользователям устанавливать Tahoe (или любое приложение Python) в общесистемное местоположение, например в `/usr/local/bin`, а также не советуем запускать Tahoe с системными библиотеками Python. Вместо этого пользователи, выполняющие сборку из исходных кодов, должны устанавливать Tahoe в новое виртуальное окружение, куда легко можно установить последние версии всех зависимостей и изолировать их от системной версии Python.

Утилита `pip2i` упрощает эту задачу: `pip2i install tahoe-lafs` создаст новое виртуальное окружение, установит в него Tahoe со всеми зависимостями и затем создаст символическую ссылку на выполняемый файл `tahoe` в каталоге `~/local/bin/`, который, вероятно, находится в вашей переменной `$PATH`. В настоящее время `pip2i` является рекомендуемым методом установки Tahoe из исходных текстов.

Общесистемную установку следует выполнять с использованием диспетчера пакетов системы. Например, `apt install tahoe-lafs` получит рабочую версию `/usr/bin/tahoe` для Debian и Ubuntu, использующую общесистемные зависимости (такие как Twisted) из `/usr/lib/python2.7/dist-packages`. Разработчики Debian (и другие создатели дистрибутивов) отвечают за совместимость общесистемных библиотек со всеми упакованными приложениями: Tahoe, Magic-Wormhole, Buildbot, Mercurial, Trac и т. д. Когда в Tahoe изменяется зависимость от версии Twisted, разработчики дистрибутивов должны учесть это и создать обновленные пакеты. И если система обновит библиотеку, подобную Twisted, которая неожиданно окажется несовместимой с Tahoe, это обновление можно отменить до того момента, когда проблема будет исправлена.

Итоги

Tahoe-LAFS – крупный проект, начатый в 2006 году, когда сам фреймворк Twisted еще находился в начале своего пути. Он содержит обходные решения для предотвращения ошибок, которые больше не существуют, и методы, на смену которым пришли новые возможности Twisted. Иногда кажется, что код скорее отражает исторические страхи и черты характера разработчиков, чем служит хорошим примером для обучения.

В нем также сквозит подавляемый гнев (вполне справедливый), копившийся многие годы работы с базой кода Twisted. И хотя Tahoe-LAFS, возможно, не является именем нарицательным, основные идеи, реализованные в этой системе, оказали влияние и были заимствованы многими другими децентрализованными системами хранения (написанными на Go, Node.js, Rust и т. д.).

Главный цикл событий в Twisted и множество готовых реализаций протоколов имеют решающее значение для нашего набора возможностей. Если вам не нравится решение, управляемое событиями, попробуйте реализовать нечто подобное с потоками и блокировками (на клиенте вам потребуется запустить для каждого сервера два потока – для записи и для чтения, еще один поток понадобится для каждого интерфейсного запроса, и во всех потоках придется скрупулезно использовать блокировки, защищающие от одновременного доступа). Шанс ошибиться в такой реализации слишком велик.

Стандартная библиотека Python включает несколько замечательных реализаций протоколов, но почти все они написаны в блокирующем стиле, что ограничивает их применение для параллельной обработки. Надеемся, что положение дел изменится, когда Python 3 и `asyncio` наберут обороты. В то же время Twisted – лучший инструмент для такого проекта, как этот.

Ссылки

- Домашняя страница Tahoe-LAFS: <https://tahoe-lafs.org/>.
- Страница Tahoe-LAFS в репозитории GitHub: <https://github.com/tahoe-lafs/tahoe-lafs>.
- Nevow: <https://github.com/twisted/nevow>.
- Foolscape: <https://foolscap.lothar.com/>.
- pipsi: <https://github.com/mitsuhiko/pipsi/>.

Глава 7

Magic Wormhole

Magic Wormhole (<http://magic-wormhole.io/>) – это инструмент безопасной передачи файлов с девизом: «Безопасная передача данных с одного компьютера на другой». Он особенно удобен, когда требуется выполнить однократную передачу в таких ситуациях, как:

- вы только что сели рядом с кем-то на конференции и хотите переслать ему со своего ноутбука тарболл с вашим проектом;
- вы разговариваете по телефону и хотите переслать ему фотографию, которую рассматриваете на своем компьютере;
- вы только что создали новую учетную запись для коллеги и хотите безопасно получить его открытый ключ SSH с его компьютера;
- вы хотите скопировать свой закрытый ключ GPG со старого компьютера на новый ноутбук;
- ваш собеседник в IRC попросил вас отправить ему файл журнала с вашего компьютера.

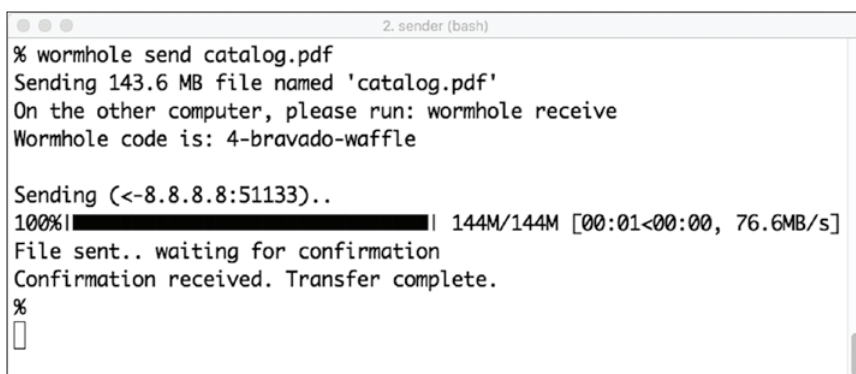
Отличительной особенностью этого инструмента является использование **кода червоточины** (wormhole code) – короткой фразы, такой как «4-bravado-waffle», которая разрешает передачу и должна сообщаться отправителем получателю. Когда Алиса попытается послать файл Бобу, она увидит на экране подобную фразу. Алиса должна как-то сообщить ее Бобу, например по телефону, в SMS или с помощью мессенджера. Код включает число и несколько слов и подбирается так, чтобы его легко можно было различить на слух даже в шумной обстановке.

Эти коды имеют одноразовый характер. Безопасность обеспечивается простыми правилами: файл получит первый из получателей, кто введет правильный код. Это *важные* правила: никто не сможет получить файл, потому что он зашифрован, и только первый, кто правильно введет код, сможет вычислить ключ для расшифровывания. И соблюдение этих правил зависит только от поведения клиентского программного обеспечения: ни сервер, ни тот, кому удалось перехватить трафик, не сможет нарушить их. Magic Wormhole предлагает уникальное сочетание строгой конфиденциальности и простоты в использовании.

КАК ЭТО ВЫГЛЯДИТ

В настоящее время Magic Wormhole доступен только в виде инструмента командной строки на Python, но уже ведутся работы по его переносу на другие языки и платформы. Наиболее важным из всех этих проектов являются разработка приложения с графическим интерфейсом (позволяющего перетаскивать файлы, предназначенные для передачи) и мобильного приложения.

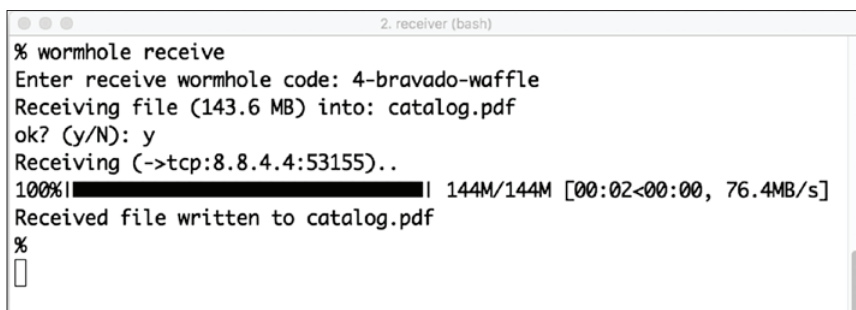
1. Алиса запускает команду `wormhole send FILENAME` на своем компьютере и видит на экране код червоточины («4-bravado-waffle»).
2. Она диктует его Бобу по телефону.
3. Боб вводит код на своем компьютере.
4. Два компьютера соединяются, затем файл расшифровывается и передается.



```
2. sender (bash)
% wormhole send catalog.pdf
Sending 143.6 MB file named 'catalog.pdf'
On the other computer, please run: wormhole receive
Wormhole code is: 4-bravado-waffle

Sending (<-8.8.8.8:51133)..
100%|████████████████████████████████████████| 144M/144M [00:01<00:00, 76.6MB/s]
File sent.. waiting for confirmation
Confirmation received. Transfer complete.
%
█
```

Рис. 7.1 ❖ Как выглядит сеанс передачи на стороне отправителя



```
2. receiver (bash)
% wormhole receive
Enter receive wormhole code: 4-bravado-waffle
Receiving file (143.6 MB) into: catalog.pdf
ok? (y/N): y
Receiving (->tcp:8.8.4.4:53155)..
100%|████████████████████████████████████████| 144M/144M [00:02<00:00, 76.4MB/s]
Received file written to catalog.pdf
%
█
```

Рис. 7.2 ❖ Как выглядит сеанс передачи на стороне получателя

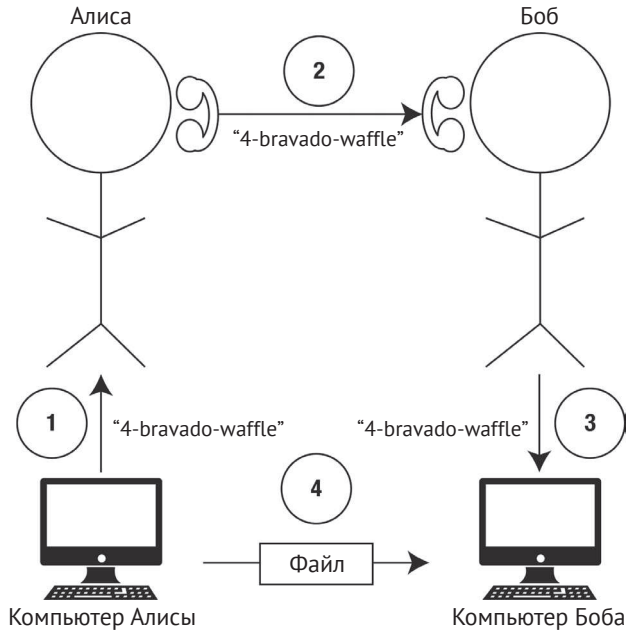


Рис. 7.3 ❖ Диаграмма процесса работы Magic Wormhole

КАК ЭТО РАБОТАЕТ

Клиенты Magic Wormhole (отправитель и получатель) подключаются к одному и тому же серверу **рандеву** и обмениваются несколькими короткими сообщениями. Эти сообщения используются для запуска специального криптографического протокола согласования ключей с названием **SPAKE2**, который является версией базового протокола обмена ключами Диффи–Хеллмана (Diffie–Hellman, более подробные сведения о нем вы найдете на ресурсах, перечисленных в разделе «Ссылки» в конце главы).

Вводя пароль (случайно сгенерированный код червоточины), каждая сторона запускает свою половину конечного автомата протокола SPAKE2. Каждая половина генерирует сообщение для отправки другой стороне. Когда эти сообщения будут доставлены, обе стороны объединяют их с собственным внутренним состоянием для создания ключа сеанса. Если обе стороны используют один и тот же код червоточины, они получают идентичные ключи сеанса. При каждом новом запуске протокола генерируется новый случайный ключ сеанса. Они используют этот ключ для шифрования всех последующих сообщений, что позволяет им установить безопасное соединение для выяснения остальных деталей передачи файлов.

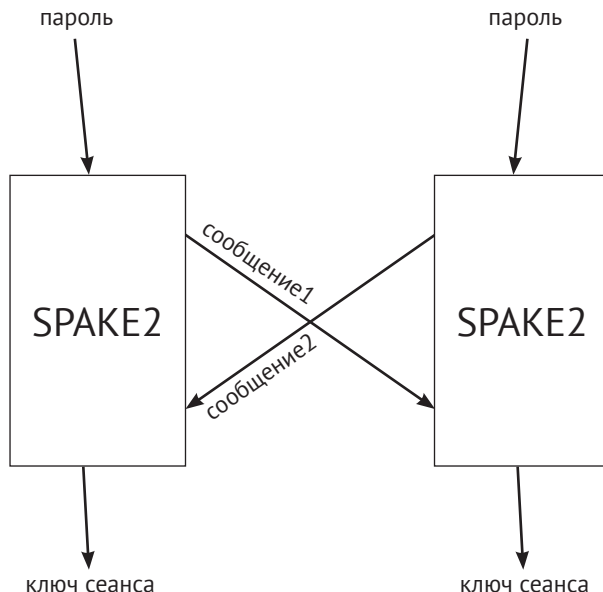


Рис. 7.4 ❖ Диаграмма работы протокола SPAKE2

У злоумышленника, перехватившего соединение, будет только один шанс угадать код. Если он ошибется, ключи сеансов получатся разными, и злоумышленник не сможет расшифровать остальные сообщения. Настоящие клиенты заметят несоответствие и завершатся с сообщением об ошибке, даже не пытаясь послать какие-либо данные.

После установки безопасного соединения клиенты обмениваются информацией о передаваемых данных и затем приступят к созданию **транзитного** соединения, через которое будет происходить передача основных данных. Для этого обе стороны сначала открывают прослушивающий сетевой сокет TCP. Они выясняют все IP-адреса, которые могут сослаться на этот сокет (их может быть несколько), и составляют список **подсказок для подключения**, которые они шифруют ключом сеанса и отправляют друг другу через сервер randevu.

Каждая сторона пытается установить прямое соединение, используя каждую подсказку для подключения. Первое успешно установленное соединение используется для передачи файла. Этот способ хорошо работает, если обе стороны находятся в одной локальной сети (например, когда оба компьютера подключены к одной и той же конференции по Wi-Fi). Поскольку они оба пытаются соединиться друг с другом (независимо от того, какая сторона отправляет файл), этот способ сработает, если хотя бы один компьютер имеет публичный IP-адрес. На практике прямую связь можно установить примерно в двух третях случаев.

Если оба компьютера находятся за разными межсетевыми экранами с NAT, они не смогут установить прямое соединение. В такой ситуации использует-

ся центральный сервер **транзитной ретрансляции**, задача которого состоит в том, чтобы «склеить» два входящих TCP-соединения.

Во всех случаях содержимое файла шифруется ключом сеанса, поэтому ни сервер randevu, ни транзитный ретранслятор не смогут увидеть его.

Этот же протокол можно использовать в других приложениях, импортировав библиотеку `wormhole` и выполняя вызовы API. Например, приложение для обмена мгновенными сообщениями, такое как Signal или Wire, могло бы использовать это решение для добавления открытых ключей ваших друзей в адресную книгу: в этом случае вам не пришлось бы копировать строку с большим ключом, а достаточно было бы сообщить другу код червоточины.

СЕТЕВЫЕ ПРОТОКОЛЫ, ЗАДЕРЖКИ ПЕРЕДАЧИ, СОВМЕСТИМОСТЬ КЛИЕНТОВ

Общее время передачи от момента запуска инструмента отправителем до получения последнего байта получателем складывается из продолжительности трех этапов:

- ввод получателем кода червоточины;
- согласование ключей и установка транзитного соединения;
- передача файла по зашифрованному каналу.

Длительность первого этапа целиком зависит от людей: программа может ждать несколько дней, пока получатель, наконец, наберет код червоточины. Длительность последнего этапа зависит от размера файла и скорости передачи данных по сети. Только второй этап находится под полным контролем протокола, поэтому он должен выполняться как можно быстрее. Мы постарались минимизировать количество необходимых сообщений и используем протокол реального времени с малой задержкой для ускорения этого этапа.

Сервер randevu поддерживает постоянный широковещательный канал (то есть сервер типа «издатель/подписчик») для каждой пары клиентов. Отправитель подключается к серверу, оставляет сообщение для получателя и ждет ответа. Затем, когда получатель запустит программу `wormhole` на своей стороне, она получит это сообщение и отправит несколько своих. Если у любого из клиентов наблюдаются проблемы с сетью, его соединение может быть разорвано, и для его восстановления может потребоваться время.

Сетевые протоколы и совместимость клиентов

Фреймворк Twisted позволяет с легкостью создавать свои протоколы поверх TCP или UDP, как было показано в первой главе этой книги. Мы могли бы создать простой протокол на основе TCP для взаимодействия с сервером randevu. Но, размышляя о будущем, мы хотели бы видеть клиентов Magic Wormhole на других языках и платформах, таких как веб-страницы или мобильные операционные системы. Протокол, который мы создали для приложения командной

строки, возможно, нелегко реализовать на других языках, или программам может потребоваться доступ к сети, запрещенный для них:

- веб-браузеры могут использовать WebSockets и WebRTC, но сценариям, выполняющимся в них, недоступен низкоуровневый протокол TCP;
- расширения для браузеров доступно все то же, что доступно веб-страницам, и даже больше, но они должны быть реализованы в специализированной версии JavaScript, где двоичные протоколы не очень естественны;
- операционные системы iOS/Android могут использовать протокол HTTP, но механизмы управления питанием могут запретить долгоживущие соединения, а запросы с использованием других протоколов, отличных от HTTP, могут не активировать встроенный радиопередатчик.

Поэтому, чтобы добиться совместимости между платформами, мы должны ограничиваться возможностями, доступными веб-браузерам.

Самый простой такой протокол должен выполнять простые HTTP-запросы GET и POST, используя превосходный пакет `trex`, который предлагает программный интерфейс, подобный интерфейсу, предлагаемому Twisted. Однако не ясно, как часто клиент должен опрашивать сервер: мы можем посылать запросы раз в секунду, тратя значительную часть пропускной способности для проверки появления ответа, который может прийти только через час. Или можно сэкономить пропускную способность и выполнять проверки раз в минуту, добавив задержку в 60 секунд в утилиту, которой порой достаточно подождать всего одну-две секунды. Проблема в том, что даже с частотой опроса один раз в секунду в работу добавляется ненужная задержка. Когда соединение работает, все необходимые действия завершаются настолько быстро, насколько сеть может передавать сообщения.

Один из способов уменьшить задержку – запустить «долгий опрос HTTP» (известный также как COMET). При таком подходе клиент будет выполнять запросы GET или POST как обычно, но сервер ретрансляции будет делать вид, что ему требуется очень много времени для доставки ответа (фактически сервер просто приостанавливает обработку запроса, пока другой клиент не подключится, чтобы получить файл). Одна из проблем заключается в том, что сервер должен в течение 30–60 секунд *что-то* ответить, например ошибкой «повторите попытку», иначе клиентская HTTP-библиотека может перестать работать. Кроме того, сообщения в последовательности (например, второе и третье), отправленные клиентами, не доставляются немедленно: время, необходимое для отправки запроса, следует добавить к задержке каждого сообщения.

Другой возможный подход называется «Server Sent Events» (события, посылаемые сервером), при использовании которого веб-контент представлен JavaScript-объектом `EventSource`. Это более строгий способ организации продолжительного опроса: клиент посылает обычный запрос GET, но записывает в заголовок `Accept` специальное значение `text/event-stream`, чтобы сообщить серверу, что соединение должно оставаться открытым. Ожидается, что ответ будет со-

держат поток закодированных событий, по одному в строке. Это довольно легко реализовать на сервере; однако для Twisted нет готовой библиотеки с поддержкой данного подхода. Сообщения передаются только в одном направлении (от сервера к клиенту), но это все, что нам нужно для нашего протокола, потому что для передачи в обратном направлении можно использовать запросы POST. Самым большим недостатком является то, что некоторые веб-браузеры (см. <http://caniuse.com/eventsource>, в частности IE и Edge) не поддерживают его.

Мы выбрали решение на основе **WebSockets**. Это стандартный протокол, реализованный в большинстве браузеров и доступный в виде библиотеки на многих языках программирования. Его легко использовать в Python и Twisted благодаря превосходной библиотеке **Autobahn** (описывается в следующей главе). Соединение выглядит как долгоживущий HTTP-сеанс, что упрощает интеграцию с существующими стеками HTTP (и обеспечивает возможность использовать его при наличии прокси-серверов и терминаторов TLS). Пакеты keep-alive обрабатываются автоматически. К тому же это протокол реального времени, поэтому сообщения доставляются максимально быстро.

Если бы у нас не было библиотеки Autobahn, мы могли бы пересмотреть свое решение. Протокол WebSockets довольно сложен в реализации, потому что имеет особую структуру (чтобы серверы не могли по ошибке спутать трафик WebSockets с каким-то другим протоколом: никому не захочется, чтобы веб-страница, сгенерированная злоумышленником, заставляла браузер посылать команды DELETE внутреннему FTP-серверу компании).

В будущем сервер randevu, вероятно, будет использовать несколько протоколов, а не только WebSockets. Наиболее привлекательной альтернативой является технология WebRTC, поскольку она включает поддержку ICE и STUN. Эти протоколы поддерживают возможность прямого соединения двух компьютеров, находящихся за NAT, поэтому два клиента могут устанавливать прямое транзитное соединение, даже если оба находятся за брандмауэрами. Технология WebRTC в основном используется для аудио/видеочатов, но включает API для обычной передачи данных. И WebRTC хорошо поддерживается большинством браузеров. Реализовать Magic Wormhole для передачи файлов между браузерами будет довольно просто, и такая реализация может работать даже лучше, чем существующий инструмент командной строки.

Проблема в том, что поддержка этой технологии *вне* браузеров практически отсутствует, отчасти из-за того, что основная сфера ее применения – аудио/видеочаты. Создатели большинства библиотек всю свою энергию направляют на поддержку аудиокодеков и алгоритмов сжатия видео, почти не уделяя внимания базовому слою для организации соединений. Самые многообещающие разработки из тех, что я видел, написаны на C++, но почти ни одна из них не имеет интерфейса для Python, что затрудняет сборку и упаковку.

Еще один претендент – протокол **libp2p**, разработанный для IPFS. Он основан на множестве узлов в большой распределенной хеш-таблице (Distributed Hash Table, DHT), а не на центральном сервере, но это проверенный временем

протокол, имеющий хорошие реализации, по крайней мере на Go и JavaScript. Версия libp2p на Python имела бы очень хорошие перспективы.

АРХИТЕКТУРА СЕРВЕРА

Сервер рандеву реализован на основе `twisted.application.service.MultiService` и прослушивает порт WebSockets.

Протокол WebSockets – это, по сути, тот же http, и библиотека Autobahn позволяет использовать один и тот же порт для обоих. В будущем это позволит нам размещать страницы и другие ресурсы веб-версии Magic Wormhole на том же компьютере, где выполняется служба рандеву. Вот как выглядит сам сервер рандеву:

```
from twisted.application import service
from twisted.web import static, resource
from autobahn.twisted.resource import WebSocketResource
from .rendezvous_websocket import WebSocketRendezvousFactory

class Root(resource.Resource):
    def __init__(self):
        resource.Resource.__init__(self)
        self.putChild(b"", static.Data(b"Wormhole Relay\n", "text/plain"))

class RelayServer(service.MultiService):
    def __init__(self, rendezvous_web_port):
        service.MultiService.__init__(self)
        ...
        root = Root()
        wsrf = WebSocketRendezvousFactory(None, self._rendezvous)
        root.putChild(b"v1", WebSocketResource(wsrf))
```

`self._rendezvous` – это наш объект `Rendezvous`, с внутренним API поддержки действий сервера рандеву: добавления сообщений в канал, подписка на каналы и т. д. Когда мы добавим дополнительные протоколы, все они будут использовать один и тот же объект.

`WebSocketResource` – это класс из библиотеки Autobahn. Он используется для добавления обработчика протокола WebSocket в любую конечную точку HTTP. Мы добавляем его как дочерний элемент «v1» в Root, то есть если наш сервер находится по адресу `magic-wormhole.io`, тогда служба рандеву будет доступна по URL: `ws://magic-wormhole.io/v1`. Мы зарезервировали v2/ и другие аналогичные пути для будущих версий протокола.

Объекту `WebSocketResource` нужно передать фабрику: мы используем нашу фабрику `WebSocketRendezvousFactory` из соседнего модуля. Эта фабрика создает экземпляры протокола нашего класса `WebSocketRendezvous`, который имеет метод `onMessage`, проверяющий содержимое каждого сообщения, анализирует его и вызывает соответствующее действие:

```
def onMessage(self, payload, isBinary):
    msg = bytes_to_dict(payload)
```

```

try:
    if "type" not in msg:
        raise Error("missing 'type'")
    self.send("ack", id=msg.get("id"))

    mtype = msg["type"]
    if mtype == "ping":
        return self.handle_ping(msg)
    if mtype == "bind":
        return self.handle_bind(msg)
    ...

```

База данных

Если оба клиента подключатся одновременно, сервер рандеву будет напрямую передавать сообщения от одного другому. В ином случае начальное сообщение должно быть помещено в буфер и храниться, пока не подключится второй клиент, иногда всего несколько секунд, а иногда часы или дни.

Ранние версии сервера рандеву хранили эти сообщения в памяти. Но каждый раз, когда сервер перезагружался (например, для обновления операционной системы), эти сообщения терялись, и любые клиенты, ожидающие подключения принимающей стороны, терпели неудачу.

Чтобы устранить эту проблему, мы добавили сохранение всех сообщений в базу данных SQLite. Каждый раз, когда поступает начальное сообщение, сервер сразу добавляет его в таблицу, и только после этого пересылает копию другому клиенту. Объект `Rendezvous` поддерживает соединение с базой данных, и каждый его метод выполняет инструкции `SELECT` и `INSERT`.

Клиенты тоже были переписаны и теперь допускают возможность потери соединения, как описано в следующем разделе, для чего в них реализован конечный автомат, повторно посылающий любое сообщение, которое не было подтверждено сервером.

Как побочный эффект всех этих нововведений появилась поддержка «автономного режима»: два клиента могут обмениваться сообщениями, не подключаясь к серверу одновременно. Конечно, при этом отсутствует возможность прямой передачи файлов, зато можно обмениваться открытыми ключами для приложения обмена сообщениями.

ТРАНЗИТНЫЙ КЛИЕНТ: ОТМЕНЯЕМЫЕ ОТЛОЖЕННЫЕ ОПЕРАЦИИ

После вычисления ключа сеанса клиенты могут безопасно обмениваться данными, но их данные все еще передаются сервером рандеву. Это слишком медленно для этапа передачи файлов: отправитель должен передать каждый байт серверу, а тот – переслать их другому клиенту. Было бы быстрее (и дешевле) использовать прямое соединение. Однако иногда клиенты не могут установить прямое соединение (например, они оба находятся за брандмауэрами с NAT), и в этом случае они вынуждены использовать сервер «транзитной ретрансля-

ции». **Транзитный клиент** отвечает за выбор наилучшего из возможных способов соединения.

Как было описано выше, каждый клиент открывает порт TCP, определяет свои IP-адреса, а затем отправляет комбинации адрес + порт другой стороне (через зашифрованный канал randevu). Чтобы расширить возможности использования новых механизмов в будущем (возможно, WebRTC), эти комбинации представлены в виде обобщенного набора «подсказок для подключения» разного типа. В настоящее время клиент распознает три вида подсказок: прямое соединение TCP, TCP с передачей через транзитный узел и TCP со скрытой службой Tor. Каждая подсказка имеет приоритет, благодаря чему клиент может поощрять использование более дешевых соединений.

Обе стороны пытаются инициировать подключение, поочередно используя подсказки в порядке убывания приоритетов. Любые подсказки, использующие транзитный узел, задерживаются на несколько секунд в пользу прямого подключения.

Первое успешно установленное подключение выигрывает гонку, после чего мы используем `defer.cancel()`, чтобы отменить все остальные попытки. Они могут находиться в состоянии ожидания (как транзитные соединения, для которых установлена двухсекундная задержка), пытаться завершить разрешение DNS или уже установили соединения, но ожидают завершения согласования.

Операция отмены отложенных вычислений аккуратно обрабатывает все эти случаи и дает программе, создавшей экземпляр `Deferred`, возможность избежать выполнения работы, результаты которой теперь все равно будут игнорироваться. И если экземпляр `Deferred` был связан с другими экземплярами `Deferred`, вызов `cancel()` проследует по цепочке до первого, еще не разрешившегося. Для нас это означает отмену попытки установить подключение или закрытие соединения, которое уже было установлено к этому моменту.

Структурируя все этапы процесса посредством `Deferred`, нам не нужно отслеживать их: достаточно сделать единственный вызов `cancel()`, чтобы аккуратно завершить данный процесс.

Мы управляем этой гонкой с помощью служебной функции в `src/wormhole/transition.py`:

```
class _ThereCanBeOnlyOne:
    """Принимает список экземпляров Deferred и возвращает объединяющий Deferred.
    Когда первый из претендентов успешно разрешится, остальные будут отменены,
    и объединяющий Deferred разрешится результатом выигравшего претендента.
    В случае ошибки объединяющий Deferred разрешится ошибкой.
    """
    def __init__(self, contenders):
        self._remaining = set(contenders)
        self._winner_d = defer.Deferred(self._cancel)
        self._first_success = None
        self._first_failure = None
        self._have_winner = False
        self._fired = False
```

```

def _cancel(self, _):
    for d in list(self._remaining):
        d.cancel()
    # поскольку все остальные претенденты в _remaining разрешатся ошибкой,
    # мы должны к этому моменту вызвать _maybe_done() и
    # разрешиться результатом self._winner_d

def run(self):
    for d in list(self._remaining):
        d.addBoth(self._remove, d)
        d.addCallbacks(self._succeeded, self._failed)
        d.addCallback(self._maybe_done)
    return self._winner_d

def _remove(self, res, d):
    self._remaining.remove(d)
    return res

def _succeeded(self, res):
    self._have_winner = True
    self._first_success = res
    for d in list(self._remaining):
        d.cancel()

def _failed(self, f):
    if self._first_failure is None:
        self._first_failure = f

def _maybe_done(self, _):
    if self._remaining:
        return
    if self._fired:
        return self._fired = True
    if self._have_winner:
        self._winner_d.callback(self._first_success)
    else:
        self._winner_d.errback(self._first_failure)

def there_can_be_only_one(contenders):
    return _ThereCanBeOnlyOne(contenders).run()

```

Мы экспортируем этот механизм как функцию, а не класс. Нам нужно превратить коллекцию Deferred в один новый экземпляр Deferred, а конструктор класса может вернуть только новый экземпляр (не Deferred). Если бы мы экспортировали `_ThereCanBeOnlyOne` как основной API, вызывающий код должен был бы использовать неуклюжий синтаксис `d = ClassXYZ(args).run()` (именно его мы спрятали за фасадом нашей функции) и мог бы допустить следующие ошибки:

- вызвать `run()` дважды;
- определить подкласс, для которого мы не можем обещать совместимость в будущем.

Обратите внимание, что если *все* претенденты Deferred потерпят неудачу, объединяющий экземпляр Deferred также потерпит неудачу. В этом случае функция `errback` получит экземпляр `Failure`, соответствующий первому отка-

завшему экземпляру `Deferred`. Идея заключается в том, чтобы дать возможность сообщить пользователю полезную информацию. Для каждого целевого хоста возможны три варианта поведения:

- соединение успешно установлено (быстрое или медленное);
- ошибка, обусловленная особенностями целевого хоста: он использует IP-адрес, недостижимый для нас, или его брандмауэр фильтрует пакеты;
- ошибка, обусловленная другими причинами, не связанными с особенностями целевого хоста, например мы вообще не подключены к интернету.

В последнем случае все ошибки подключения будут одинаковыми, поэтому не имеет значения, о какой из них сообщить. Во втором случае сообщения о первом отказе будет вполне достаточно, чтобы пользователь смог понять, что пошло не так.

СЕРВЕР ТРАНЗИТНОЙ РЕТРАНСЛЯЦИИ

Код сервера транзитной ретрансляции находится в пакете `magic-wormhole-transit-relay`. В настоящее время он использует свой протокол на основе TCP, но я надеюсь добавить интерфейс `WebSockets`, чтобы его также можно было использовать в браузерах.

Основой ретранслятора является протокол, связывающий друг с другом пару экземпляров (по одному для каждого клиента). Каждый экземпляр имеет ссылку на «собеседника», и каждый раз, когда ему поступают данные, он передает их своему собеседнику:

```
class TransitConnection(protocol.Protocol):
    def dataReceived(self, data):
        if self._sent_ok:
            self._total_sent += len(data)
            self._buddy.transport.write(data)
        return
    ...

    def buddy_connected(self, them):
        self._buddy = them
        ...
        # Объединение в пару производитель/потребитель.
        # Мы используем streaming=True, поэтому ожидается, что производитель
        # реализует интерфейс IPushProducer и используется pauseProducing() для
        # его приостановки и resumeProducing() для возобновления.
        self._buddy.transport.registerProducer(self.transport, True)

        # Объект Transit вызовет buddy_connected() для обоих протоколов,
        # поэтому будет создано две пары производитель/потребитель.

    def buddy_disconnected(self):
        self._buddy = None
        self.transportloseConnection()
```

```
def connectionLost(self, reason):
    if self._buddy:
        self._buddy.buddy_disconnected()
    ...
```

Остальной код связан с выбором подключений для объединения. Транзитные клиенты посылают строку квитирования сразу после подключения, и ретранслятор ищет двух клиентов, приславших одинаковые строки. Оставшаяся часть метода `dataReceived` реализует конечный автомат, который ожидает получения строки квитирования и затем сравнивает ее с другими соединениями в поисках совпадения.

Когда собеседники будут найдены, между ними устанавливаются отношения производитель/потребитель: TCP-транспорт Алисы регистрируется как производитель для Боба, и наоборот. Если восходящий канал Алисы передает данные быстрее, чем нисходящий канал Боба, транспорт TCP, подключенный к `Transit-Connection` на стороне Боба, заполнится и вызовет `pauseProducing()` транспорта Алисы, который удалит ее сокет TCP из списка чтения реактора (до вызова `resumeProducing()`). То есть некоторое время ретранслятор не будет читать данные из этого сокета, что приведет к заполнению входящего буфера ядра, после чего стек TCP ядра уменьшит окно TCP, что вынудит компьютер Алисы прекратить передачу, пока не освободятся приемные буферы.

В результате всего этого Алиса будет наблюдать скорость передачи, не превышающую той, с которой Боб может получать данные. Без этой связи производитель/потребитель Алиса будет передавать данные в ретранслятор настолько быстро, насколько позволяет ее соединение, и ретранслятор должен будет хранить их, пока Боб не загрузит эти данные к себе. Раньше, пока мы не добавили данный механизм регулировки, ретранслятору иногда не хватало памяти, когда люди посылали очень большие файлы очень медленным получателям.

АРХИТЕКТУРА КЛИЕНТА

На стороне клиента есть возможность использовать библиотеку `Wormhole` из пакета `wormhole` для установки соединений в стиле червоточины через сервер, библиотеку `Transit` для создания прямых зашифрованных соединений TCP (возможно, через ретранслятор) и инструмент командной строки для управления передачей файлов. Большая часть кода находится в библиотеке `Wormhole`.

Объект `Wormhole` создается с помощью простой фабричной функции и имеет программный интерфейс на основе `Deferred` для выбора типа соединения и отправки/получения сообщений:

```
import wormhole

@inlineCallbacks
def run():
    w = wormhole.create(appid, relay_url, reactor)
    w.allocate_code()
```

```
code = yield w.get_code()
print "wormhole code:", code
w.send_message(b"outbound message")
inbound = yield w.get_message()
yield w.close()
```

Для создания объекта `Wormhole` мы используем фабричную функцию `create` вместо конструктора класса. Это позволяет нам сохранить фактический класс закрытым и изменять его реализацию, не нарушая совместимость в будущем. Например, на самом деле есть два вида объектов `Wormhole`. По умолчанию используется интерфейс на основе `Deferred`, но если в `create` передать необязательный аргумент `delegate=`, она вернет альтернативную версию, которая будет вызывать методы объекта делегата вместо разрешения `Deferred`.

Функция `create` принимает внешний объект `Reactor`, а не импортирует его изнутри, чтобы дать вызывающему приложению возможность определять тип используемого реактора. Это также упрощает модульное тестирование, потому что мы можем передать фиктивный реактор, например не использующий сетевые сокеты или осуществляющий операции по времени.

Внутренне объект `Wormhole` использует более десятка небольших конечных автоматов, каждый из которых отвечает за небольшую часть процесса соединения и согласования ключей. Например, короткое целое число в начале кода червоточины («4» в 4-bravado-waffle) называется *номерным знаком*, и эти знаки распределяются, используются и высвобождаются с помощью специализированного конечного автомата. Аналогично, на сервере размещается *почтовый ящик*, посредством которого два клиента могут обмениваться сообщениями: у каждого клиента есть конечный автомат, который управляет представлением об этом почтовом ящике и знает, когда его нужно открыть, закрыть или отправить сообщения в нужный момент.

Отложенные вычисления и конечные автоматы, одноразовый наблюдатель

Основной поток сообщений довольно прост, но полный протокол весьма сложен. Эта сложность проистекает из цели допустить возможность неожиданно-го разрыва соединения (с последующим повторным подключением), а также остановки сервера (с последующей перезагрузкой).

Каждый ресурс, который клиент может выделить или зарезервировать, должен быть освобожден в нужное время. По этой причине процесс получения номерных знаков и почтовых ящиков организован так, чтобы всегда двигаться вперед, независимо от сбоев в подключениях.

Но еще большая сложность обусловлена другой целью: приложения, использующие библиотеку, могут сохранить свое состояние на диск, завершить работу, а затем перезапуститься через некоторое время и продолжить с того места, где они остановились. Это необходимо для приложений обмена сообщениями,

которые периодически запускаются и закрываются. Для нормальной работы приложению необходимо знать, когда поступило сообщение и как сериализовать состояние протокола (вместе с другими данными, необходимыми приложению). Такие приложения должны использовать **Delegate API**.

Объекты *Deferred* являются хорошим выбором для систем, управляемых потоками данных, в которых любые действия могут выполняться ровно один раз, но их трудно упорядочить. Однако для состояний, которые могут двигаться вперед, а затем откатываться, или для событий, способных возникать многократно, более предпочтительно выглядят конечные автоматы. В ранних версиях *wormhole* значительный упор был сделан на объекты *Deferred*, из-за чего было сложно обрабатывать потерю и повторную установку соединений. В текущей версии объекты *Deferred* применяются только на верхнем уровне, а ниже используются конечные автоматы.

Объект *Wormhole* использует более десятка взаимосвязанных конечных автоматов, каждый из которых реализован с помощью **Automat**. Библиотека *Automat* не является частью *Twisted*, но была написана членами сообщества и впервые использована в *Twisted ClientService* (это утилита, которая поддерживает соединение с заданной конечной точкой, заново устанавливая его, когда соединение теряется или происходит сбой в процессе подключения; *Magic Wormhole* использует *ClientService* для подключения к серверу рандеву).

В качестве конкретного примера на рис. 7.5 показан конечный автомат **Allocator**, который управляет распределением номерных знаков. Они распределяются сервером рандеву по запросу отправляющей стороны (если отправитель и получатель согласовали код червоточины каким-то иным способом, тогда обе стороны вводят его непосредственно в программах клиентов).

В любой конкретный момент соединение с сервером рандеву либо установлено, либо нет, и переходы между этими двумя состояниями вызывают отправку сообщения *connected* или *lost* большинству конечных автоматов, включая *Allocator*. Конечный автомат *Allocator* остается в одном из двух состояний ожидания (*S0A* – ожидание + отключен или *S0B* – ожидание + подключен) столько, сколько потребуется. Если код более высокого уровня решит, что ему нужен номерной знак, он посылает событие *allocate*. Если в этот момент *Allocator* был подключен, он посылает через механизм управления соединениями *Rendezvous Connector* сообщение *allocate* (прямоугольник, подписанный как *RC.tx_allocate*), а затем переходит в состояние *S1B*, в котором пребывает, ожидая ответа. Когда приходит ответ (*gx_allocated*), он выбирает случайные слова для остальной части кода червоточины, информирует конечный автомат *Code* о выделении номерного знака (*C.allocated()*) и переходит в конечное состояние *S2*: завершено.

Пока не получен ответ *gx_allocated*, мы не знаем, был запрос доставлен серверу или нет. Поэтому мы должны обеспечить 1: повторную отправку запроса после восстановления соединения – и 2: идемпотентность запроса, чтобы сервер реагировал на два или более запросов так же, как он реагировал бы на один запрос. Это гарантирует правильное поведение сервера в обоих случаях.

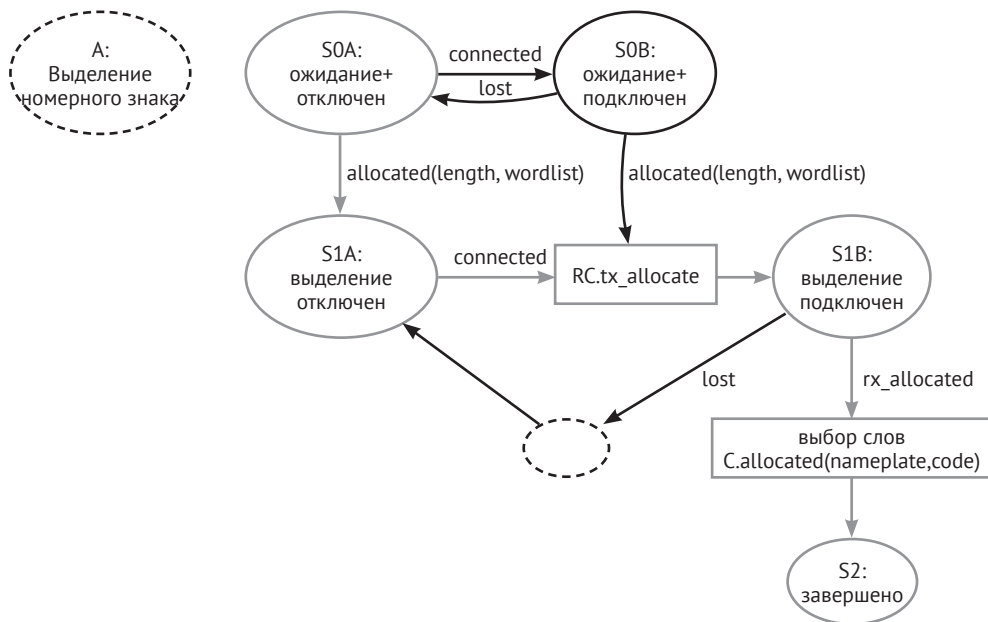


Рис. 7.5 ❖ Конечный автомат Allocator

Номерной знак может быть запрошен до того, как будет установлено соединение. В пути от S1A до S1B запрос `allocate` передается в любом случае: соединение может быть установлено перед тем, как возникла необходимость выделить номерной знак, или будет повторно установлено после отправки запроса.

Этот шаблон можно заметить в большинстве наших конечных автоматов. Более сложные примеры можно увидеть в реализациях конечных автоматов `Nameplate` и `Mailbox`, которые создают или подписываются на именованный канал на сервере рандеву. В обоих случаях состояния выстраиваются в два столбца: «отключен» слева или «подключен» справа. Вертикальная позиция в столбце определяет достигнутое (или что еще нужно сделать). Потеря соединения вызывает переход справа налево. Восстановление соединения вызывает переход слева направо и, как правило, отправку нового сообщения с запросом (или повторную передачу предыдущего). Получение ответа вызывает переход вниз.

Конечный автомат верхнего уровня **Boss** – это место, где конечные автоматы уступают место объектам `Deferred`. Приложения, импортирующие библиотеку `Magic Wormhole`, могут запросить объект `Deferred`, который разрешится, когда произойдет важное событие. Например, приложение может создать объект `Wormhole` и добавить такой обработчик:

```

from twisted.internet import reactor
from wormhole.cli.public_relay import RENDEZVOUS_RELAY
import wormhole
# Передайте в APPID нечто, что отличает ваше приложение

```

```

w = wormhole.create(APPID, RENDEZVOUS_RELAY, reactor)
w.allocate_code()
d = w.get_code()

def allocated_code(code):
    print("the wormhole code is:{}".format(code))

d.addCallback(allocated_code)

```

Конечный автомат Allocator доставит сообщение allocated автомату Code (C.allocated). Автомат Code передаст код автомату Boss (B.got_code), автомат Boss доставит его объекту Wormhole (W.got_code), а объект Wormhole передаст его в любой ожидающий объект Deferred (который был создан вызовом get_code()).

ОДНОРАЗОВЫЕ НАБЛЮДАТЕЛИ

В следующем фрагменте из src/wormhole/wormhole.py показан шаблон «одноразового наблюдателя», используемого для управления доставкой кодов червоточины, как из хранилища (описанного выше), так и из интерактивного ввода:

```

@implementer(IWormhole, IDDeferredWormhole)
class _DeferredWormhole(object):
    def __init__(self):
        self._code = None
        self._code_observers = []
        self._observer_result = None
        ...

    def get_code(self):
        if self._observer_result is not None:
            return defer.fail(self._observer_result)
        if self._code is not None:
            return defer.succeed(self._code)
        d=defer.Deferred()
        self._code_observers.append(d)
        return d

    def got_code(self, code):
        self._code = code
        for d in self._code_observers:
            d.callback(code)
        self._code_observers[:] = []

    def closed(self, result):
        if isinstance(result,Exception):
            self._observer_result = failure.Failure(result)
        else:
            # ожидающие экземпляры Deferred разрешились ошибкой
            self._observer_result = WormholeClosed(result)
        ...
        for d in self._code_observers:
            d.errback(self._observer_result)

```

Метод `get_code()` может быть вызван любое количество раз. При использовании стандартного инструмента передачи файлов клиент отправителя выделяет код и ждет вызова `get_code()`, чтобы вывести его на экран (для передачи получателю). Получатель *сообщает* полученный код (передавая его в аргументе командной строки либо вводя его в интерактивном режиме), поэтому его клиент не заботится о вызове `get_code()`. В других приложениях могут быть свои причины вызывать этот метод несколько раз.

Нам нужно, чтобы все эти вызовы возвращали один и тот же ответ (или ошибку) и чтобы их цепочки обратных вызовов были независимыми.

PROMISE/FUTURE И DEFERRED

Термин **Future** пришел из модели акторов, предложенной Карлом Хьюиттом (Carl Hewitt), и языков программирования Joule и E и других ранних объектно-ориентированных систем (в которых используется похожий термин Promise). Под терминами Future и Promise подразумевается объект, ссылающийся на значение, которое *пока* недоступно, но (возможно) будет доступно когда-то в будущем, или его вычисление завершится ошибкой, и тогда объект Future/Promise никогда не будет ссылаться на какое-либо действительное значение.

Этот механизм позволяет программам оперировать чем-то, чего пока не существует. Это может показаться бессмысленным, но не спешите с выводами, потому что с такими несуществующими значениями можно сделать много полезного. Например, можно запланировать выполнение некоторых операций, *когда они станут доступны*, их можно передать в функции, которые сами планируют эти операции. В более продвинутых системах **конвейеры обработки будущих результатов** позволяют отправить сообщение объекту Promise, и если он фактически находится на другом компьютере, сообщение будет передано в целевую систему, для чего может потребоваться выполнить несколько переходов. В целом они помогают программистам описывать свои будущие намерения компилятору или интерпретатору, чтобы тот мог лучше распланировать операции.

Объекты **Deferred** очень похожи на Promise, но являются уникальными для Twisted. Они, скорее, служат инструментом управления обратными вызовами, чем полноценным механизмом Promise. Прежде чем заняться исследованием отличий между ними, нужно разобраться с особенностями работы реального механизма Promise.

В E – объектно-ориентированном языке, обладающем наиболее полной реализацией Promise, существует функция `makePromiseResolverPair()`, которая возвращает два *отдельных* объекта: Promise и Resolver. Вычислить значение для Promise можно только с помощью Resolver, а узнать результат вычислений можно лишь с помощью Promise. Язык поддерживает специальный синтаксис, блок `when`, инструкции в котором будут выполнены только после того, как Promise разрешится в некоторое конкретное значение. Если бы система Magic

Wormhole была написана на E, метод `get_code()` вернул бы объект Promise, а вывод полученного кода выглядел бы так:

```
p = w.get_code();
when (p) {
    writeln("The code is:", p);
}
```

Объекты Promise доступны в современном JavaScript (ES6) благодаря множеству точек пересечения между сообществом пользователей объектно-ориентированных языков и организации по стандартизации TC39. Эти объекты не имеют специального синтаксиса, позволяющего дожидаться разрешения, и используют удобные анонимные функции JavaScript (включая синтаксис **стрелочных функций**, добавленный в ES6). Соответствующий код на JavaScript мог бы выглядеть так:

```
p = w.get_code();
p.then(code=>{console.log("The code is:",code)});
```

Существенная разница между Promise в языках E и JS и Deferred в Twisted заключается в способах составления цепочек. Метод `then()` в Javascript возвращает *новый* экземпляр Promise, который запускается, если и когда завершится функция обратного вызова (если обратный вызов вернет промежуточный экземпляр Promise, тогда `then()` будет ждать, пока разрешится этот объект). То есть для одного «родительского» объекта Promise можете сконструировать две отдельные цепочки обработки, например:

```
p = w.get_code();

function format_code(code){
    return slow_formatter_that_returns_a_promise(code);
}

p.then(format_code).then(formatted => {console.log(formatted)});

function notify_user(code){
    return display_box_and_wait_for_approval(code);
}

p.then(notify_user).then(approved => {console.log("code delivered!)});
```

В JavaScript эти два действия будут выполняться «параллельно», или, по крайней мере, ни одно из них не будет мешать другому.

В Twisted, напротив, объекты Deferred создают цепочку обратных вызовов *без создания* дополнительных экземпляров Deferred.

```
d1 = w.get_code()
d = d1.addCallback(format_code)
assert d1 is d # addCallback вернет тот же самый экземпляр Deferred!
```

Немного похоже на шаблон «конструирования цепочек вызовов» в JavaScript, широко используемый в веб-фреймворках (например, d3.js, jQuery), которые создают объекты с использованием цепочек вызовов методов:

```
s = d3.scale()
    .linear()
    .domain([0,100])
    .range([2,40]);
```

Такое цепочечное поведение объектов `Deferred` может преподносить неприятные сюрпризы, особенно при попытке запустить параллельные вычисления:

```
d1 = w.get_code()
d1.addCallback(format_code).addCallback(print_formatted)
# неверно!
d1.addCallback(notify_user).addCallback(log_delivery)
```

В этом примере функция `notify_user` будет вызвана только *после* завершения `print_formatted`, но она не получит код червоточины, вместо этого она получит *результат вызова* `print_formatted`. Наш шаблон кодирования (две строки, каждая из которых начинается с `d1.addCallback`) обманчив. На самом деле код выше в точности эквивалентен следующему:

```
d1 = w.get_code()
d1.addCallback(format_code)
d1.addCallback(print_formatted)
d1.addCallback(notify_user) # теперь ошибка более очевидна!
d1.addCallback(log_delivery)
```

Чтобы реализовать желаемое поведение, нужно создать новый экземпляр `Deferred`, который будет разрешаться тем же значением, но позволит нам определить новую цепочку выполнения:

```
def fanout(parent_deferred, count):
    child_deferreds = [Deferred() for i in range(count)]
    def fire(result):
        for d in child_deferreds:
            d.callback(result)
    parent_deferred.addBoth(fire)
    return child_deferreds

d1 = w.get_code()
d2, d3 = fanout(d1,2)
d2.addCallback(format_code)
d2.addCallback(print_formatted)
d3.addCallback(notify_user)
d3.addCallback(log_delivery)
```

Этот код выглядит довольно неуклюжим, поэтому в своих проектах я обычно создаю служебный класс с именем `OneShotObserverList`. Этот класс «наблюдателя» имеет метод `when_fired()` (который возвращает новый, независимый экземпляр `Deferred`) и метод `fire()` (который запускает их все). `when_fired()` можно вызвать либо до, либо после вызова `fire()`.

Приведенный выше код из Magic Wormhole (`get_code()/got_code()`) является подмножеством полной реализации `OneShotObserverList`. Процесс соединения может завершиться неудачей по разным причинам, но в любом случае он вы-

зовет функцию `closed()` с экземпляром `Failure` (в случае преднамеренного закрытия соединения функция `closed()` будет вызвана с другим значением, отличным от `Failure`, которое затем будет обернуто исключением `WormholeClosed`). Этот код гарантирует, что каждый экземпляр `Deferred`, возвращаемый функцией `get_code()`, разрешится ровно один раз – успехом (и с кодом червоточины) или ошибкой.

ОТСРОЧЕННЫЕ ВЫЗОВЫ, СИНХРОННОЕ ТЕСТИРОВАНИЕ

Еще одной особенностью `Promise` в языке `E` является возможность **отсрочить вызов** (eventual send). Она предназначена для постановки в очередь вызова метода в одной из будущих итераций цикла событий. В `Twisted` ту же роль играет `reactor.callLater(0, callable, argument)`. В `E` и `JavaScript` объекты `Promise` автоматически предоставляют эту гарантию для своих обратных вызовов.

Отсроченный вызов – это простой и надежный способ избежать ошибок, возникающих при нарушении порядка вызовов. Например, представьте обобщенный шаблон наблюдателя (обладающий более широкими возможностями, чем простой `OneShotObserverList`, описанный выше):

```
class Observer:
    def __init__(self):
        self.observers = set()

    def subscribe(self, callback):
        self.observers.add(callback)

    def unsubscribe(self, callback):
        self.observers.remove(callback)

    def publish(self, data):
        for ob in self.observers:
            ob(data)
```

Теперь представьте, что получится, если одна из функций обратного вызова обратится к методу `subscribe` или `unsubscribe`, изменив список наблюдателей в середине цикла? В зависимости от особенностей выполнения итерации вновь добавленный обратный вызов может получить или не получить текущее событие. В `Java` итератор может даже сгенерировать исключение `ConcurrentModificationException`.

Повторный вход – еще один возможный сюрприз: если какой-то обратный вызов отправит новое сообщение тому же наблюдателю, тогда функция `publish` будет вызвана повторно, до завершения первого вызова, что может нарушить многие предположения, сделанные программистом (особенно если функция сохраняет состояние в переменных экземпляра). Наконец, если обратный вызов возбудит исключение, увидят ли событие остальные наблюдатели?

Эти мелкие неожиданности все вместе известны как «опасности нарушения координации». К ним относятся: пропуск событий, дублирование событий, неопределенный порядок выполнения и бесконечные циклы.

Внимательное отношение к программированию может помочь избежать многих из этих проблем: можно, например, продублировать список наблюдателей перед итерацией, перехватывать исключения в обратных вызовах и использовать флаг для обнаружения повторного входа. Но гораздо проще и надежнее использовать механизм отсроченного вызова:

```
def publish(self, data):
    for ob in self.observers:
        reactor.callLater(0, ob, data)
```

Я с успехом использовал это решение во многих проектах (Foolscap, Tahoe-LAFS), и оно помогло избавиться от целого класса ошибок. Недостатком этого подхода является усложнение тестирования, потому что результат отсроченного вызова нельзя проверить синхронно. Кроме того, отсутствие прямой связи трассировки стека с причиной ошибки усложняет отладку: если обратный вызов сгенерирует исключение, трассировка не поможет понять, *почему* была вызвана эта функция. Объекты Deferred страдают схожими проблемами, но в них можно использовать себе в помощь функцию `defer.setDebugging(True)`.

Работая над Magic Wormhole, я экспериментировал, используя синхронные модульные тесты вместо механизма отсроченных вызовов.

АСИНХРОННОЕ ТЕСТИРОВАНИЕ С ОБЪЕКТАМИ DEFERRED

В Twisted имеется своя система модульного тестирования под названием **Trial**, которая основана на пакете `unittest`. Она предлагает специализированные методы для тестирования Deferred. Наиболее очевидной особенностью является возможность дождаться, пока разрешится экземпляр Deferred, который вернул тест (или разрешить запуск следующего теста). В сочетании с `inlineCallbacks` эта особенность позволяет убедиться, что действия происходят в определенном порядке:

```
@inlineCallbacks
def test_allocate_default(self):
    w = wormhole.create(APPID, self.relayurl, reactor)
    w.allocate_code()
    code = yield w.get_code()
    mo = re.search(r"^\d+-\w+-\w+$", code)
    self.assert_(mo, code)
    # w.close() потерпит неудачу, потому что соединение еще не установлено
    yield self.assertFailure(w.close(), LonelyError)
```

В этом тесте вызов `w.allocate_code()` иницирует выделение кода червоточины, а `w.get_code()` возвращает объект Deferred, который в конечном итоге разрешится полным кодом. Между этими событиями объект Wormhole должен связаться с сервером и выделить номерной знак (вместо использования реального сервера тест запускает в функции `setUp()` локальный сервер randevu). Инструкция `yield w.get_code()` получит этот экземпляр Deferred, дождется, пока

он разрешится, и присвоит результат переменной `code`, чтобы мы могли проверить его позже.

В действительности тестовая функция возвращает `Deferred` и передает управление циклу обработки событий, а затем в какой-то момент в будущем приходит ответ сервера, и возобновляется выполнение функции с того места, где она остановилась. Если ошибка препятствует функции `get_code()` разрешить объект `Deferred`, тест будет ждать две минуты (время тайм-аута по умолчанию), а затем объявит об ошибке.

Функция `self.assertFailure()` принимает `Deferred` и список `(*args)` с типами исключений. Она ждет разрешения объекта `Deferred`, а потом проверяет, разрешился ли он с одним из этих исключений: если был вызван метод `.callback()` объекта `Deferred` (то есть ошибки не возникло), тогда `assertFailure` объявляет тест неудачным. Если был вызван метод `.errback()` с ошибкой не из списка, тест также объявляется неудачным.

Этим мы преследуем три цели. `Wormhole API` требует вызывать метод `w.close()` по завершению. Этот метод возвращает `Deferred`, который разрешается после закрытия всех соединений. Мы используем его, чтобы предотвратить переход к следующему тесту, пока не будут освобождены все ресурсы, выделенные предыдущим (закрыты все сетевые сокеты и остановлены все таймеры), что также позволяет избежать появления ошибки «неочищенного реактора» из `Trial`.

Этот объект `Deferred` дает приложениям возможность обнаружить ошибки соединения. В этом тесте мы работаем только с одним клиентом, поэтому к нему никто не может подключиться, и в результате вызова его метода `close` он разрешится с ошибкой `LonelyError`. Мы используем `assertFailure`, чтобы убедиться, что не возникло никаких *других* ошибок, которые мы могли допустить и которые должны обнаруживаться модульными тестами, например как `NameError`.

Третья цель – не допустить неудачное завершение всего процесса тестирования. В других тестах, где происходит успешное соединение, мы используем простую инструкцию `yield w.close()` в конце. В этих случаях ошибка `LonelyError` выглядела бы как проблема для `Trial`, и тест, где она возникла, будет помечен как неудачный. Использование `assertFailure` сообщает `Trial`, что для этого объекта `Deferred` нормой считается, если он разрешается определенной ошибкой.

СИНХРОННОЕ ТЕСТИРОВАНИЕ С ОБЪЕКТАМИ DEFERRED

В действительности `test_allocate_default` является **интеграционным тестом**, потому что он использует одновременно несколько компонентов системы (включая сервер рандеву и локальный сетевой интерфейс). Такие тесты, как правило, обеспечивают сквозную проверку, но выполняются несколько медленнее. Кроме того, они не обеспечивают гарантированный охват кода.

Тесты, ожидающие разрешения `Deferred` (возвращаемого одним из тестов, получаемого в середине функции `@inlineCallbacks` или передаваемого в вызов

`assertFailure`), предполагают, что вы не уверены, *когда* произойдет ожидаемое событие. Такое разделение хорошо, когда приложение ожидает, пока библиотека что-то предпримет: приведение в действие механизма вызова обработчиков – это задача библиотеки, а не приложения. Но в модульных тестах мы должны точно знать, чего ожидать.

Библиотека `Trial` предлагает три инструмента управления объектами `Deferred`, которые *не* ждут их разрешения: `successResultOf`, `failResultOf` и `assertNoResult`. Они просто проверяют, находится ли `Deferred` в определенном состоянии.

Чаще всего они используются в паре с классом `Mock`, чтобы «проникнуть» в некоторый тестируемый код и вызвать определенные внутренние переходы в известное время.

В качестве примера рассмотрим тесты поддержки `tor` в `Magic Wormhole`. Она добавляет аргумент командной строки, который обеспечивает маршрутизацию всех соединений через демон `Tor`, то есть `wormhole send --tor` не будет передавать ваш IP-адрес серверу randevu (или получателю). Код поиска (или запуска) подходящего демона `Tor` находится в классе `TorManager` и зависит от внешней библиотеки `txtorcon`. Мы можем заменить `txtorcon` библиотекой `Mock`, а затем использовать пример выше для проверки работоспособности нашего кода в `TorManager`.

Эти тесты проверяют весь наш код, использующий `Tor`, без фактического привлечения реального демона `Tor` (что, очевидно, было бы медленнее, ненадежнее и непереносимо). Они действуют, предполагая, что `txtorcon` работает как предполагается. Мы не проверяем фактическую работу `txtorcon`, а просто записываем и проверяем все, что было передано в `txtorcon`, затем имитируем правильные ответы `txtorcon` и проверяем, как наш код реагирует на эти ответы.

Самый простой тест проверяет, что происходит, если `txtorcon` не установлен: нормальная работа не должна быть нарушена, но попытка использовать `--tor` должна вызвать сообщение об ошибке. Чтобы упростить имитацию, модуль `tor_manager.py` обрабатывает ошибку импорта, присваивая переменной `txtorcon` значение `None`:

```
# tor_manager.py
try:
    import txtorcon
except ImportError:
    txtorcon = None
```

Этот модуль имеет функцию `get_tor()`, которая возвращает объект `Deferred`, разрешающийся либо объектом `TorManager`, либо ошибкой `NoTorError`. Объект `Deferred` возвращается, потому что при нормальной работе на соединение с портом управления `Tor` требуется время. Но в этом конкретном случае мы знаем, что он должен разрешиться немедленно (с ошибкой `NoTorError`), потому что ошибка `ImportError` возникает тут же. Итак, вот как выглядит тест:

```
from ..tor_manager import get_tor
class Tor(unittest.TestCase):
```

```
def test_no_txtorcon(self):
    with mock.patch("wormhole.tor_manager.txtorcon", None):
        d = get_tor(None)
        self.failureResultOf(d, NoTorError)
```

Метод `mock.patch` гарантирует, что переменная `txtorcon` получит значение `None`, даже притом что пакет `txtorcon` всегда импортируется во время тестов (в нашем файле `setup.py` пакет `txtorcon` перечислен в зависимостях в дополнительном разделе `[dev]`). Объект `Deferred`, возвращаемый вызовом `get_tor()`, уже находится в состоянии ошибки. Вызов `self.failureResultOf(d, *errortypes)` убедится, что этот объект `Deferred` уже разрешился одной из указанных ошибок. А так как `failureResultOf` немедленно проверяет объект `Deferred`, он тут же возвращает управление. Наш тест `test_no_txtorcon` не возвращает `Deferred` и не использует `@inlineCallbacks`.

Похожий тест выполняет проверку предварительных условий внутри `get_tor()`. Все проверки типов, которые должна производить функция, мы выполняем с помощью ее вызова. Например, аргумент `launch_tor` – это логический флаг, который указывает, должен ли `tor_manager` запустить новую копию `Tor` или может использовать существующую. Передавая в этом аргументе значение, отличное от `True` и `False`, мы ожидаем, что `Deferred` разрешится ошибкой `TypeError`:

```
def test_bad_args(self):
    d = get_tor(None, launch_tor="not boolean")
    f = self.failureResultOf(d, TypeError)
    self.assertEqual(str(f.value), "launch_tor= must be boolean")
```

Этот тест выполняется полностью синхронно, без ожидания, пока разрешится какой-либо объект `Deferred`. Набор тестов, подобных этому, проверяет каждую строку и каждую ветвь в модуле `tor_manager` за 11 мс.

Другая типичная проверка – что объект `Deferred` еще не разрешился, потому что пока не наступило условие, которое позволило бы ему разрешиться. За такой проверкой обычно следует строка, иницилирующая событие, и проверка, что `Deferred` разрешился некоторым конкретным значением или исключением.

Класс `Transit` в `Magic Wormhole` управляет (прямыми, как мы надеемся) TCP-соединениями между клиентами, которые используются для передачи основного массива данных. Каждая сторона прослушивает порт и формирует список «подсказок для подключения» на основе каждого IP-адреса, который он имеет (включая локальные адреса, которые вряд ли будут доступны). Затем каждая сторона иницирует соединение, используя сразу все подсказки, представленные другой стороной. Первое успешно установленное соединение, в рамках которого благополучно выполнялась процедура согласования, объявляется выигравшим, а все остальные отменяются.

Для управления этой гонкой используется вспомогательная функция `there_can_be_only_one()` (описанная выше). Она принимает несколько объектов `De-`

ferred и возвращает другой Deferred, который разрешается, когда успешно разрешится один из переданных ей объектов Deferred. В Twisted есть свои функции, которые делают нечто подобное (например, DeferredList), но нам нужна была функция, которая отменила бы все остальные Deferred.

Для этой проверки мы использовали `assertNoResultOf(d)` и `value = successResultOf(d)` из Trial:

```
class Highlander(unittest.TestCase):
    def test_one_winner(self):
        cancelled = set()
        contenders = [Deferred(lambda d, i=i: cancelled.add(i))
                      for i in range(5)]
        d = transit.there_can_be_only_one(contenders)
        self.assertNoResult(d)
        contenders[0].errback(ValueError())
        self.assertNoResult(d)
        contenders[1].errback(TypeError())
        self.assertNoResult(d)
        contenders[2].callback("yay")
        self.assertEqual(self.successResultOf(d), "yay")
        self.assertEqual(cancelled, set([3,4]))
```

Этот тест сначала проверяет, что объединенный объект Deferred еще не разрешился. Затем проверяет, что он не разрешится даже в случае сбоя некоторых из подчиненных объектов Deferred. Потом имитируется событие, влекущее успешное разрешение одного из подчиненных объектов Deferred, и проверяется, разрешился ли объединенный Deferred с правильным значением и отменились ли остальные претенденты.

Функции `successResultOf()` и `failResultOf()` имеют один недостаток: их нельзя вызвать несколько раз с одним и тем же объектом Deferred, потому что они добавляют в него свой обработчик, который мешает всем последующим обработчикам (включая добавленные другими вызовами `successResultOf()`). Конечно, трудно придумать вескую причину, когда такое могло бы понадобиться, и тем не менее подобное поведение может вызвать некоторую путаницу, если у вас есть подпрограмма, которая проверяет состояние объекта Deferred, и вы используете ее несколько раз. А вот `assertNoResult` можно вызывать сколько угодно раз.

Синхронное тестирование и отсроченный вызов

Сообщество Twisted уже несколько лет движется к этому стилю с использованием непосредственного выполнения и фиктивных объектов. Я только недавно начал применять его, но вполне доволен результатами: мои тесты получаются более быстрыми, тщательными и детерминированными. Но иногда нахожу более ценным стиль «отсроченный вызов». В `there_can_be_only_one()` соперничающие объекты Deferred практически не зависят от обработчиков, прикрепленных к общему объекту Deferred, но меня все равно волнует вероятность

ошибиться, и я чувствую себя комфортнее, когда обратный вызов выполняется в другой итерации цикла событий.

Кроме того, все, что связано с настоящими реакторами, сложно проверить, не дожидаясь, пока разрешится Deferred. Поэтому я постоянно ищу возможность объединить стиль немедленного тестирования с удобством отсроченных вызовов.

Когда я впервые начал использовать отсроченные вызовы и Глиф увидел, как я вызываю `reactor.callLater(0, f)`, он написал более удачную версию, которую мы используем в Foolscape и Tahoe-LAFS. Она поддерживает отдельную очередь обратных вызовов и в каждый конкретный момент имеет только один `callLater`, ожидающий обработки: этот подход намного эффективнее, когда имеются тысячи активных вызовов, и он позволяет избежать зависимости от `reactor.callLater`, поддерживающего порядок активации таймеров с одинаковыми значениями.

Одна из приятных особенностей его функции `finally()` заключается в наличии дополняющей ее специальной функции `flushEventualQueue()`, которая повторяет обход очереди снова и снова, пока та не опустеет. Это позволяет писать такие тесты:

```
class Highlander(unittest.TestCase):
    def test_one_winner(self):
        cancelled = set()
        contenders = [Deferred(lambda d, i=i: cancelled.add(i))
                      for i in range(5)]
        d = transit.there_can_be_only_one(contenders)
        flushEventualQueue()
        self.assertNoResult(d)
        contenders[0].errback(ValueError())
        flushEventualQueue()
        self.assertNoResult(d)
        contenders[1].errback(TypeError())
        flushEventualQueue()
        self.assertNoResult(d)
        contenders[2].callback("yay")
        flushEventualQueue()
        self.assertEqual(self.successResultOf(d), "yay")
        self.assertEqual(cancelled, set([3,4]))
```

Но `flushEventualQueue` имеет свой недостаток: она находится в экземпляре-синглтоне диспетчера отсроченных вызовов, которому свойственны все проблемы, связанные с использованием реактора. Для их преодоления этот диспетчер должен передаваться в функцию `there_can_be_only_one()` в аргументе, так же как современный код Twisted передает реактор в функции, которые в нем нуждаются, а не импортирует его напрямую. Фактически если бы мы положились на `reactor.callLater(0)`, то могли бы протестировать этот код с экземпляром `Clock()` и вручную имитировать ход времени, чтобы опустошить очередь. Возможно, этот шаблон будет использоваться в будущих версиях.

Итоги

Magic Wormhole – это приложение для передачи файлов, обеспечивающее высокую безопасность за счет использования криптографического алгоритма SPAKE2. В его основе лежит библиотечный API, пригодный для встраивания в другие приложения. Приложение использует Twisted для управления сразу несколькими TCP-соединениями, что часто обеспечивает быструю прямую передачу данных между двумя клиентами. Библиотека Autobahn поддерживает соединения через WebSockets, что обеспечит совместимость с будущими клиентами, действующими в браузерах. Набор тестов использует вспомогательные функции Twisted для проверки состояния каждого объекта Deferred, по мере того как они проходят через разные этапы, что позволяет проводить быстрое и синхронное тестирование.

Ссылки

- Домашняя страница Magic Wormhole: <http://magic-wormhole.io>.
- Страница в GitHub: <https://github.com/warner/magic-wormhole>.
- SPAKE2: <http://www.lothar.com/blog/54-spake2-random-elements/>.
- WebSockets: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.
- Библиотека requests: <http://python-requests.org/>.
- Библиотека treq: <https://github.com/twisted/treq>.
- Библиотека Autobahn: <https://crossbar.io/autobahn/>.
- Протокол libp2p: <https://libp2p.io/>.
- Библиотека Automat: <https://github.com/glyph/Automat>.
- Технология асинхронных и параллельных вычислений Futures: https://ru.wikipedia.org/wiki/Futures_and_promises.
- Promise в JavaScript: https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Ispolzovanie_promisov.
- Promise в E: <http://wiki.erights.org/wiki/Promise>.
- Отсроченный вызов: [https://en.wikipedia.org/wiki/E_\(programming_language\)](https://en.wikipedia.org/wiki/E_(programming_language)).
- Опасности нарушения координации: <http://erights.org/talks/thesis/>.
- Утилита eventual(): <https://github.com/warner/foolscap/blob/master/src/foolscap/eventual.py>.

Глава 8

Передача данных в браузерах и микросервисах с использованием WebSocket

Нужен ли протокол WebSocket?

Протокол WebSocket начинал разрабатываться как конкурент HTTP-запросов AJAX. Когда потребовалось организовать связь между браузером и сервером в режиме реального времени, этот протокол стал хорошей альтернативой устаревшим решениям, таким как долгий опрос или Comet. Поскольку веб-сокеты создают постоянное соединение и не имеют заголовков, они оказались самой быстрой и легковесной альтернативой для обмена большим количеством коротких сообщений.

В настоящее время, однако, все большую популярность обретает протокол HTTP2, также предлагающий постоянное соединение и быструю передачу данных.

Так нужен ли нам протокол WebSocket?

Да, нужен! Во-первых, WebSocket API предназначен для использования не только на сервере, но и в клиентских приложениях. Поэтому независимо от реализации вы можете подключиться к событиям жизненного цикла соединения, реагировать на разъединение, прикреплять данные к сеансу и т. д. Это очень удобно для организации надежных взаимодействий и создания приятных впечатлений у пользователя.

Во-вторых, HTTP2 имеет сжатые заголовки, а WebSocket вообще не имеет их, что еще больше уменьшает накладные расходы. На самом деле реализации HTTP2 принудительно шифруют данные, даже когда этого не требуется, в то время как WebSocket предлагает возможность выбора, когда и на что расходовать ресурсы компьютера, активировать SSL или нет.

Более того, серверы HTTP2, как правило, по своей инициативе отправляют статические ресурсы (CSS, изображения, сценарии JS и т. д.) в браузеры, но

прикладные данные обычно посылают по запросу. В этой части WebSocket превосходит HTTP2: он может отправлять уведомления пользователям, рассылать события, сигнализировать об изменениях...

Однако в WebSocket есть одна странность: он не привязан к доменному имени, и браузерам не нужны никакие специальные настройки для совместного использования ресурсов разными источниками (CORS). Вы легко сможете подключиться из веб-страницы к локальному серверу WebSocket на вашем компьютере. Это можно рассматривать и как достоинство, и как недостаток, в зависимости от того, что нужно сделать.

Все эти характеристики делают WebSocket отличным инструментом для рассылки уведомлений на сайте, реализации чатов, торговли, многопользовательских игр или создания графиков и диаграмм в режиме реального времени. Не говоря уже о том, что эту технологию можно использовать в качестве связующего звена между всеми вашими компонентами и превратить ее в коммуникационный уровень, координирующий всю систему.

Используя WebSocket, ваш веб-сервер сможет общаться с вашими процессами кеширования или с вашей платформой аутентификации. Вы сами сможете управлять массой устройств IoT¹. В конце концов, Raspberry Pi де-факто поддерживает Python.

В общем и целом WebSocket – беспроблемный вариант, так как эта технология доступна в большинстве браузеров вплоть до IE10, а это около 94 % рынка, если судить по данным caniuse.com. В худшем случае для экзотических браузеров, не имеющих поддержки WebSocket, можно найти прокладки. Поскольку порядок создания соединений в WebSocket и HTTP совместимы, этот протокол почти наверняка будет работать в любой сети, пропускающей трафик HTTP. Вы даже можете использовать порты 80 и 443 для этих двух протоколов.

WEBSOCKET И TWISTED

На стороне сервера протокол WebSocket широко поддерживается всеми популярными языками, но требует применения методов асинхронного программирования из-за постоянных соединений. Поскольку к серверу может подключаться большое число клиентов, использование потоков выполнения часто оказывается не лучшим решением для реализации сервера WebSocket. Но асинхронный ввод/вывод подходит идеально; и Twisted является весьма привлекательной платформой в этом отношении.

Более того, протокол WebSocket можно использовать вне браузера для организации взаимодействий между всеми компонентами на ваших серверах в режиме реального времени. Это позволит вам создать свою собственную микросервисную архитектуру, распределяя функции по мелким компонентам или распространяя информацию напрямую, без использования для этого центральной базы данных.

¹ Internet of Things – интернет вещей.

Чтобы показать, как извлечь выгоду из WebSocket в среде Twisted, мы используем экосистему Autobahn. Autobahn – это набор библиотек, распространяемых на условиях лицензии MIT и написанных на разных языках. Он позволяет создавать клиенты и серверы WebSocket и поставляется с набором тестов для проверки уровня соответствия стандартам любой системы WebSocket.

И не только.

Конечно, при желании можно выработать свои соглашения о взаимодействиях с использованием WebSocket; и Autobahn, безусловно, поможет вам в этом. Но, выбрав такой путь, вы будете делать то же самое, что и все остальные, и изобретать свой велосипед (скорее всего, с квадратными колесами).

По большому счету все варианты применения WebSocket можно разделить на две категории:

- вызов удаленного кода и получение результата – как в AJAX, только лучше, быстрее и проще. Эта технология известна уже давно и называется «RPC» – Remote Procedure Calls (вызов удаленных процедур);
- отправка сообщений для уведомления других частей системы о происходящем. Это тоже очень распространенный шаблон, часто называемый «PUB/SUB» – Publish/Subscribe (публикация/подписка).

Позже мы подробно расскажем, что это значит. Но на данный момент для вас важно понять, что для правильной работы требуется написать много хорошо продуманного кода, управляющего сериализацией, аутентификацией, маршрутизацией, обработкой ошибок и исключительных ситуаций.

Зная это, авторы Autobahn решили создать протокол более высокого уровня, который они назвали WAMP – WebSocket Application Messaging Protocol (протокол обмена сообщениями между приложениями WebSocket)¹. Это открытый стандарт, зарегистрированный в IANA[], а его реализации способны избавить вас от тяжелой работы.

Самое замечательное, что WAMP можно использовать там, где поддерживается WebSocket, то есть практически везде. Вам не придется бороться с HTTP здесь, MQTT там и AMQP в остальных местах. Вы сможете использовать единый протокол и избавиться от лишних хлопот.

К счастью, библиотека Autobahn для Python поддерживает как сам протокол WebSocket, так и WAMP, используя Twisted. Исследованием этой поддержки мы и займемся в данной главе. Поэтому, прежде чем начать, установите пакет autobahn, например, с помощью pip:

```
pip install autobahn[twisted]
```

Как обычно, для этого рекомендуется создать виртуальное окружение Python 3. В этой главе мы используем версию Autobahn 17.10.1 – она прекрасно работает с Python 2.7 и 3.3+. Более того, она может работать даже с PyPy и Jython

¹ Не путайте со стеком «Windows Apache MySQL PHP», который был популярен в сети до появления AJAX.

и поддерживает `asyncio`, что может пригодиться тем, кто не желает быть привязанным к `Twisted`, но в этой главе мы будем использовать `Twisted` и `Python 3`.

Поскольку `WebSocket` является очень интересной технологией для веб-сайтов, позже мы задействуем сценарии на `JavaScript`. Однако `WebSocket` можно использовать не только в вебе – этот замечательный протокол прекрасно подходит для взаимодействий между процессами на сервере.

WEBSOCKET, ИЗ PYTHON В PYTHON

Роль программы «Привет, мир» в мире сетей часто играет эхо-сервер, именно его мы и напишем в нашей первой попытке. Хотя `Twisted` поддерживает конструкции `async/await`, мы используем сопрограммы, чтобы обеспечить более широкую поддержку `Python 3`.

Вот как выглядит эхо-сервер с `WebSocket`, использующий библиотеку `Autobahn`:

```
import uuid

from autobahn.twisted.websocket import (
    WebSocketServerProtocol,
    WebSocketServerFactory
)

class EchoServerProtocol(WebSocketServerProtocol):

    def onConnect(self, request):
        """Вызывается при подключении клиента"""
        # Вывести IP-адрес клиента, обслуживаемого этим экземпляром протокола
        print(u"Client connecting:{0}".format(request.peer))

    def onOpen(self):
        """Вызывается при открытии соединения WebSocket"""
        print(u"WebSocket connection open.")

    def onMessage(self, payload, isBinary):
        """Вызывается при получении очередного сообщения от клиента

        Параметры:

        payload (str|bytes): содержимое сообщения
        isBinary (bool): содержит ли сообщение закодированный текст (False)
                        или двоичные данные (True).
                        Значение по умолчанию False.
        """
        # Просто вывести полученное сообщение
        if isBinary:
            # Это двоичное сообщение и может содержать любые байты.
            # Здесь мы воссоздаем UUID из байтов, полученных от клиента.
            uid = uuid.UUID(bytes=payload)
            print(u"UUID received:{0}".format(uid))
        else:
            # Это закодированный текст. Обратите внимание, что он HE
```

```

        # декодируется автоматически, isBinary – это просто флаг, который
        # клиент любезно устанавливает для каждого сообщения. Вы должны
        # знать, какой набор символов используется (здесь utf8), и вызвать
        # ".decode()", чтобы преобразовать байты в строковый объект.
        print(u"Text message received:{0}".format(payload.decode( 'utf8'))))

# Это -- эхо-сервер, поэтому отправим обратно все, что получили
self.sendMessage(payload, isBinary)

def onClose(self, wasClean, code, reason):
    """Вызывается при закрытии соединения WebSocket клиентом

    Параметры:

    wasClean (bool): соединение только будет закрыто, или это уже
                     произошло.
    code (int): любой код из коллекции WebSocketClientProtocol.CLOSE_*
    reason (str): простое текстовое сообщение, описывающее причину
                  закрытия соединения
    """
    print(u"WebSocket connection closed:{0}".format(reason))

if __name__ == '__main__':
    from twisted.internet import reactor

    # В качестве схемы протокол WebSocket использует WS. Поэтому WebSocket URL
    # выглядит точно так же, как HTTP URL, но со схемой WS вместо HTTP.
    factory=WebSocketServerFactory(u"ws://127.0.0.1:9000")
    factory.protocol=EchoServerProtocol

    print(u"Listening on ws://127.0.0.1:9000")
    reactor.listenTCP(9000, factory)
    reactor.run()

```

Запустите эхо-сервер, выполнив в терминале команду:

```

$ python echo_websocket_server.py
Listening on ws://127.0.0.1:9000

```

Надеюсь, вы догадались, что «echo_websocket_server.py» – это имя файла, в котором хранится код эхо-сервера.

Вот как выглядит эхо-клиент с WebSocket, использующий библиотеку Autobahn:

```

# coding: utf8
import uuid

from autobahn.twisted.util import sleep
from autobahn.twisted.websocket import (
    WebSocketClientProtocol,
    WebSocketClientFactory
)

from twisted.internet.defer import Deferred, inlineCallbacks

class EchoClientProtocol(WebSocketClientProtocol):

```

```
def onConnect(self, response):
    # Вывести IP-адрес сервера
    print(u"Server connected:{0}".format(response.peer))

@inlineCallbacks
def onOpen(self):
    print("WebSocket connection open.")

    # Посылать сообщения раз в секунду
    i=0
    while True:

        # Послать текстовое сообщение.
        # Его НУЖНО закодировать вручную.
        self.sendMessage(u"☺ Hello wørld{!}".format(i).encode('utf8'))

        # При отправке двоичных данных нужно сообщить об этом,
        # установив флаг "isBinary". Здесь мы создаем случайный
        # уникально-универсальный идентификатор и посылаем его
        # как последовательность байтов.
        self.sendMessage(uuid.uuid4().bytes, isBinary=True)
        i+=1
        yield sleep(1)

def onMessage(self, payload, isBinary):
    # Не будем конвертировать сообщения, чтобы увидеть их
    # в исходной форме
    if isBinary:
        print(u"Binary message received:{!r}bytes".format(payload))
    else:
        print(u"Encoded text received:{!r}".format(payload))

def onClose(self, wasClean, code, reason):
    print(u"WebSocket connection closed:{0}".format(reason))

if __name__ == '__main__':
    from twisted.internet import reactor

    factory=WebSocketClientFactory(u"ws://127.0.0.1:9000")
    factory.protocol=EchoClientProtocol

    reactor.connectTCP(u"127.0.0.1",9000, factory)
    reactor.run()
```

Запустите клиента, выполнив в другом терминале команду:

```
python echo_websocket_client.py
```

Важно отметить, что клиент должен запускаться после сервера, потому что в этих простых примерах не реализовано обнаружение или повторное восстановление соединения.

Сразу после этого вы увидите в терминале клиента следующие строки:

```
WebSocket connection open.
```

```
Encoded text received: b'\xc2\xa9 Hell\xc3\xb8 w\xc3\xb8rld 0 !'
```

```
Binary message received: b'\xecA\xd9u\xa3\xa1K\xc3\x95\xd5\xba~\x11ss\xa6' bytes
```

```

Encoded text received: b'\xc2\xa9 Hell\xc3\xb8 w\xc3\xb8rld 1 !'
Binary message received: b'\xb3NAv\xb300o\x97\xaf\xde\xeaD\xc8\x92F' bytes
Encoded text received: b'\xc2\xa9 Hell\xc3\xb8 w\xc3\xb8rld 2 !'
Binary message received: b'\xc7\xda\xb6h\xbd\xbaC\xe8\x84\x7f\xce:,\x15\x
                          xc4$' bytes
Encoded text received: b'\xc2\xa9 Hell\xc3\xb8 w\xc3\xb8rld 3 !'
Binary message received: b'qw\x8c@\xd3\x18D\xb7\xb90;\xee9Y\x91z' bytes

```

А в терминале сервера:

```

WebSocket connection open.
Text message received: ☺ Hellø wørld 0 !
UUID received: d5b48566-4b20-4167-8c18-3c5b7199860b
Text message received: ☺ Hellø wørld 1 !
UUID received: 3e1c0fe6-ba73-4cd4-b7ea-3288eab5d9f6
Text message received: ☺ Hellø wørld 2 !
UUID received: 40c3678a-e5e4-4fce-9be8-6c354ded9cbc
Text message received: ☺ Hellø wørld 3 !
UUID received: eda0c047-468b-464e-aa02-1242e99a1b57

```

Это означает, что сервер и клиент обменялись несколькими сообщениями.

Также обратите внимание, что сервер только лишь отвечает на сообщения. Однако он мог бы вызывать `self.sendMessage()`, даже не получив никакого сообщения, и таким способом принудительно послать данные клиенту.

Теперь повторим то же самое, но в форме веб-приложения.

WEBSOCKET, ИЗ PYTHON В JAVASCRIPT

Отправка данных в браузер – классический вариант использования WebSocket. Ограниченное количество страниц в книге не позволяет нам показать традиционный пример чата. Однако, как известно, любой чат должен сообщать, сколько пользователей находится в сети. Вот как может выглядеть упрощенная реализация этой функции.

Сначала напишем сервер на Python.

```

from autobahn.twisted.websocket import (
    WebSocketServerProtocol,
    WebSocketServerFactory
)

class SignalingServerProtocol(WebSocketServerProtocol):
    connected_clients=[]

    def onOpen(self):
        # Каждый раз, когда открывается очередное соединение WebSocket,
        # нужно сохранить ссылку на клиента в атрибуте класса, общем
        # для всех экземпляров протокола. Это упрощенная реализация, но
        # она прекрасно подходит для целей демонстрации.
        self.connected_clients.append(self)
        self.broadcast(str(len(self.connected_clients)))

```

```
def broadcast(self, message):
    """ Посылает сообщение всем подключенным клиентам

        Параметры:
            message (str): посылаемое сообщение
    """
    for client in self.connected_clients:
        client.sendMessage(message.encode('utf8'))

def onClose(self, wasClean, code, reason):
    # Если клиент отключился, ссылку на него нужно удалить из
    # атрибута класса.
    self.connected_clients.remove(self)
    self.broadcast(str(len(self.connected_clients)))

if __name__ == '__main__':
    from twisted.internet import reactor

    factory = WebSocketServerFactory(u"ws://127.0.0.1:9000")
    factory.protocol = SignalingServerProtocol

    print(u"Listening on ws://127.0.0.1:9000")
    reactor.listenTCP(9000, factory)
    reactor.run()
```

Запустите этот сервер командой:

```
python signaling_websocket_server.py
```

Теперь перейдем к реализации HTML + JS:

```
<!DOCTYPEhtml> <html><head></head><body>

<h1>Connected users: <span id="count">...</span></h1>

<!--
    Короткая ссылка на CDN-версию библиотеки autobahn.js
    За дополнительной информацией обращайтесь по адресу:
    https://github.com/crossbario/autobahn-js
-->
<script src="http://goo.gl/1pfDD1"></script>
<script>
/* Если вы используете старый браузер, эта часть кода должна выглядеть иначе.
   Этот код будет работать, начиная с версии IE11.*/
var sock = new WebSocket("ws://127.0.0.1:9000");

/* Так же как в версии на Python, можно добавить обработчики sock.onopen() и
   sock.onclose(). Но в данном примере нам требуется реагировать только
   на получаемые сообщения: */

sock.onmessage = function(e){
    var span = document.getElementById('count');
    span.innerHTML=e.data;
}
</script>
</body></html>
```

Теперь откройте файл с этой разметкой HTML в веб-браузере.

После этого вы увидите страницу, начинающуюся строкой «Connected users: x», где x будет изменяться с открытием той же страницы в новой вкладке или после ее закрытия.

Вы заметите, что даже браузеры со строгой политикой CORS, такие как Google Chrome, не запрещают устанавливать WebSocket-соединение из страниц, открытых по протоколу «file://», как это было бы с запросом AJAX. Протокол WebSocket работает в любом контексте, принимает удаленные или локальные доменные имена, даже если файл загружен с локального диска, а не с веб-сервера.

БОЛЕЕ МОЩНАЯ ПОДДЕРЖКА WebSocket в WAMP

WebSocket – простой, но мощный инструмент; однако это довольно низкоуровневый протокол. Если вы решите создать полноценную систему, использующую WebSocket, вам придется предусмотреть:

- способ сопряжения двух сообщений для имитации HTTP-цикла запрос/ответ;
- некоторый механизм сериализации с использованием JSON или msgpack, или чего-то еще;
- соглашения по обработке ошибок и их отладке;
- механизм рассылки сообщений отдельным группам клиентов;
- средства аутентификации и что-то, что поможет связать идентификатор сеанса из HTTP с соединением WebSocket;
- систему привилегий, чтобы можно было ограничивать действия клиентов и доступную им информацию.

В конечном итоге вы напишете нестандартную, менее документированную и непроверенную альтернативу WAMP.

Протокол WAMP с успехом решает все перечисленные выше задачи. Он работает поверх WebSocket, поэтому наследует все его характеристики и преимущества, а также добавляет множество новых и интересных возможностей:

- вы можете определить функции и объявить их общедоступными в сети. Любой клиент сможет вызывать их из любого места (да, удаленно) и получать результаты. Это RPC-часть протокола WAMP – вы можете рассматривать ее как поддержку AJAX-запросов на стероидах или намного более простую альтернативу технологиям CORBA/XMLRPC/SOAP;
- вы можете определить события. Любой клиент (даже удаленный) сможет сказать «эй, меня интересует это событие». После этого другой код в любом месте сможет сказать «эй, это событие случилось», и все заинтересованные клиенты получат уведомление. Это PUB/SUB-часть протокола WAMP – вы можете использовать ее как еще более простую версию RabbitMQ;
- все ошибки будут автоматически распространяться по сети. То есть если клиент X вызовет функцию на клиенте Y и та завершится с ошибкой, клиент X получит сообщение об этом;

- идентификация и аутентификация являются частью стандарта и прекрасно сочетаются с механизмом сеансов HTTP;
- все разбито на пространства имен. Вы можете фильтровать их, использовать шаблонные символы, устанавливать разрешения и даже балансировать нагрузку.

Мы не сможем обсудить все это в такой короткой главе, но я постараюсь дать вам представление хотя бы о поддержке RPC и PUB/SUB.

WAMP – это маршрутизируемый протокол, то есть полученный запрос не передается непосредственно в код, который должен этот запрос обработать, а попадает в WAMP-совместимый маршрутизатор, который обеспечивает передачу сообщений соответствующим фрагментам кода.

В этом смысле WAMP не является протоколом клиент/сервер: любой код, который посылает WAMP-запрос, является клиентом. То есть весь код, включая веб-страницы, процессы на серверах, внешние службы и вообще все, что взаимодействует по протоколу WAMP, – это клиенты, или маршрутизаторы, WAMP, общающиеся друг с другом.

По этой причине маршрутизатор WAMP является единой точкой отказа и потенциальным узким местом. К счастью, эталонная реализация Crossbar.io в Twisted действует достаточно надежно и быстро. Эту реализацию можно установить с помощью простой команды `pip`, и, чтобы запустить наш следующий пример, вы должны выполнить:

```
pip install crossbar
```

Если вы используете Windows, вам также нужно установить зависимость `windowsapi`. Сделайте это прямо сейчас¹.

После установки вам должна быть доступна команда `crossbar2`:

```
$ crossbar version
```

```

Crossbar.io : 17.11.1 (Crossbar.io COMMUNITY)
Autobahn    : 17.10.1 (with JSON, MessagePack, CBOR, UBJSON)
Twisted      : 17.9.0-EPollReactor
LMDB         : 0.93/lmdb-0.9.18
Python       : 3.6.2/CPython
OS           : Linux-4.4.0-98-generic-x86_64-with-Ubuntu-16.04-xenial
Machine      : x86_64
Release key  : RWT/n6IQ4dKesCP8YywjIwH30ST8eq5D21ih4EFbJZazzsqEX6CnaT3k

```

Пакет `crossbar.io` играет разные роли; ему необходим конфигурационный файл, в котором было бы указано, что вы хотите сделать. К счастью, он способен сгенерировать базовую конфигурацию автоматически:

```
crossbar init
```

Эта команда создаст каталоги `web` и `.crossbar`, а также файл `README`. Вы можете игнорировать или даже удалить каталог `web` и файл `README`. Сейчас нас интересует только сгенерированный файл `.crossbar/config.json`. Его не нужно изменять, чтобы запустить пример, описываемый ниже, потому что конфигурация по умолчанию просто «разрешает все». Открыв файл, вы найдете множество настроек, которые трудно понять без контекста. Чтобы разобраться в основах протокола WAMP, не нужно копать так глубоко, поэтому мы продолжим.

Далее нужно просто запустить маршрутизатор `crossbar`. Запускать следует в том же каталоге, в котором находится каталог `.crossbar`:

```
$ crossbar start
2017-11-23T19:06:43+0200 [Controller 11424] New node key pair generated!
2017-11-23T19:06:43+0200 [Controller 11424] File permissions on node public key fixed!
2017-11-23T19:06:43+0200 [Controller 11424] File permissions on node private key fixed!
2017-11-23T19:06:43+0200 [Controller 11424]
2017-11-23T19:06:43+0200 [Controller 11424] ( _ | _ ) / _ \ _ / _ | _ ) \ / _ | | _ \
2017-11-23T19:06:43+0200 [Controller 11424] \ _ , | \ \ / . _ / . _ / | _ ) / ~ \ | \ . | \ _ \
2017-11-23T19:06:43+0200 [Controller 11424]
2017-11-23T19:06:43+0200 [Controller 11424] Version: Crossbar.io COMMUNITY 17.11.1
2017-11-23T19:06:43+0200 [Controller 11424] Public Key:
81da0aa76f36d4de2abcd1ce5b238d00a
...
```

Crossbar.io можно считать аналогом Apache или Nginx: это программное обеспечение, которое вы должны настроить и запустить, а весь остальной ваш код будет вращаться вокруг него. Crossbar.io действительно может выполнять функции статического веб-сервера, WSGI-сервера и даже диспетчера процессов. Но мы используем его только как поддержку WAMP, и для этого нам не нужно больше ничего делать. Пусть он работает в фоновом режиме, а мы сосредоточимся на программном коде вашего клиента.

Вся прелесть WAMP в том, что клиентам не нужно знать о существовании друг друга. Им достаточно знать, где находится маршрутизатор. По умолчанию маршрутизатор прослушивает `localhost:8080` и определяет «область» (группу клиентов, которые могут видеть друг друга) с именем `realm1`. То есть все, что мы должны сделать, – это подключиться к маршрутизатору, используя имеющуюся информацию.

Чтобы наглядно показать, что клиентам WAMP не нужно знать о существовании друг друга или что этот протокол не является протоколом клиент/сервер, в нашем первом примере я использую две веб-страницы.

Одна страница будет иметь поле ввода и кнопку **sum** (сложить). Другая – свое поле ввода и функцию `sum()`, доступную для удаленного вызова. Когда вы щелкнете на кнопке **sum**, значение из первого поля ввода будет отправлено второй

странице, которая вызовет функцию `sum()`, передав ей полученное и свое локальное значение, получит результат и отправит его обратно.

Страницы будут работать вообще без серверного кода!

Первая страница, первый клиент:

```
<!DOCTYPEhtml> <html><head></head><body>

  <form name="sumForm"><input type="text" name="number" value="3"></form>

  <script src="http://goo.gl/1pfDD1"></script>

  <script>

    // Подключиться к маршрутизатору WAMP
    var connection = new autobahn.Connection({
      url:"ws://127.0.0.1:8080/ws",
      realm:"realm1"
    });

    // Обратный вызов для обработки события успешного подключения
    connection.onopen = function (session,details){
      // Зарегистрировать функцию под именем "sum", чтобы любой
      // WAMP-клиент в группе "realm1" смог вызвать ее. Это и есть RPC.
      session.register('sum', function(a){
        // В действительности это самая обычная функция. Но параметры
        // и возвращаемое значение должны быть сериализуемыми.
        // По умолчанию в JSON.
        return parseInt(a) + parseInt(document.sumForm.number.value);
      });
    }

    // Установить соединение
    connection.open();
  </script>
</body></html>
```

Если открыть файл с этим кодом в браузере, можно заметить, что терминале, где запущен маршрутизатор Crossbar.io, появилось примерно такое сообщение о подключении нового клиента:

```
2017-11-23T20:11:41+0200 [Router 13613] session "5770155719510781" joined
realm "realm1"
```

Теперь вторая страница и второй клиент:

```
<!DOCTYPEhtml> <html><head></head><body>

  <form name="sumForm" method="post" >
    <input type="text" name="number" value="5">
    <button name="sumButton">Sum!</button>
    <span id="sumResult">...</span>
  </form>

  <script src="http://goo.gl/1pfDD1"></script>
  <script>
    var connection = new autobahn.Connection({
```

```

url:"ws://127.0.0.1:8080/ws",
realm:"realm1"
});

connection.onopen = function (session,details){
    // Отправляя форму (т. е. щелкая на кнопке), мы вызываем "sum()"
    // Нам не нужно знать, где находится "sum()" или как она выполняется,
    // нам достаточно знать, что где-то есть нечто с этим именем.
    document.sumForm.addEventListener('submit', function(e){
        e.preventDefault();
        // Первый параметр -- пространство имен функции. Второй --
        // аргументы для функции. Этот вызов вернет Promise, который
        // мы используем для записи результата в тег span после его получения
        session.call('sum',[document.sumForm.number.value]).then(
            function(result){
                document.getElementById('sumResult').innerHTML = result;
            });
    });
}
connection.open();
</script>
</body></html>

```

Откройте этот файл в другой вкладке браузера. И снова в терминале с роутером появится сообщение о подключении нового клиента.

Теперь щелкните на кнопке **Sum!** на первой странице. Она вызовет код во второй странице и практически сразу получит результат. Конечно, то же самое можно проделать и в Python. Имейте в виду, что это очень упрощенный пример, он не заботится ни о надежности, ни о безопасности. Но я надеюсь, что общий смысл вам ясен. Вы можете использовать этот механизм RPC для определения и вызова кода в любом месте, в любом браузере или в любом процессе на любом сервере, подключенном к маршрутизатору.

Теперь у вас в руках имеется полезный инструмент RPC, но не меньше пользы может принести его собрат PUB/SUB – еще один хороший инструмент. Для демонстрации я добавлю клиента на Python (который фактически будет выполняться на сервере Crossbar).

Этот клиент на Python будет раз в секунду просматривать каталог и составлять список файлов, находящихся в нем. Для каждого расширения файла, найденного в каталоге, он будет посылать событие со списком всех соответствующих файлов. Бесполезно? Возможно. Круто? Конечно!

```

import os

from twisted.internet.defer import inlineCallbacks
from twisted.logger import Logger

from autobahn.twisted.util import sleep
from autobahn.twisted.wamp import ApplicationSession
from autobahn.twisted.wamp import ApplicationRunner

class DirectoryLister(ApplicationSession):

```

```
log= Logger()

@inlineCallbacks
def onJoin(self, details):
    while True:

        # Составить список файлов и сгруппировать их по расширению
        files = {}
        for f in os.listdir('.'):
            file, ext = os.path.splitext(f)
            if ext.strip():
                files.setdefault(ext, []).append(f)

        # Послать событие "filewithext.xxx" для каждого расширения файла
        # где "xxx" -- расширение. Мы подключим список файлов к событиям,
        # чтобы любой клиент, заинтересованный в таких событиях, смог
        # получить список файлов.
        # Это "издатель" (PUB) в паре "PUB/SUB".
        for ext, names in files.items():
            # Обратите внимание, что не требуется объявлять событие до
            # его использования. События можно посылать по мере их
            # появления.
            yield self.publish('filewithext' +ext , names)

        yield sleep(1)

# ApplicationRunner позаботится обо всем необходимым.
if __name__ == '__main__':
    runner=ApplicationRunner(url=u"ws://localhost:8080/ws", realm=u"realm1")
    print(u"Connecting to ws://localhost:8080/ws")
    runner.run(DirectoryLister)
```

Запустите этот код командой:

```
python directory_lister.py
```

Он тут же приступит к созданию списка файлов в текущем каталоге и начнет посылать события о найденных файлах.

Теперь нам нужен клиент, чтобы сообщить, что его интересуют эти события. Мы можем написать этого клиента на Python или на JS. Поскольку в WAMP любой код является клиентом, напомним его на JS, чтобы посмотреть, как взаимодействуют клиенты, написанные на разных языках.

```
<!DOCTYPEhtml> <html><head></head><body>

<div id="files">...</div>

<script src="http://goo.gl/1pfDD1"></script>

<script>

// Параметры подключения к маршрутизатору WAMP
var connection = new autobahn.Connection({
    url:"ws://127.0.0.1:8080/ws",
    realm:"realm1"
});
```

```

connection.onopen = function (session,details){
    // Вывести на странице HTML список файлов
    var div=document.getElementById('files');
    div.innerHTML="";
    function listFile(params,meta,event){
        var ul=document.getElementById(event.topic);
        if (!ul){
            div.innerHTML += "<ul id='" + event.topic + "'></ul>";
            ul=document.getElementById(event.topic);
        }
        ul.innerHTML="";
        params[0].forEach(function(f){
            ul.innerHTML += "<li>" + f + "</li>";
        })
    }

    // Сообщить маршрутизатору, что нас интересует событие с этим именем.
    // Это "подписчик" (SUB) в паре "PUB/SUB".
    session.subscribe('filewithext.py',listFile);

    // Любой клиент, как эта веб-страница, может подписаться на
    // произвольное количество событий. Поэтому здесь мы подписываемся
    // на события с файлами ".py" и ".txt".
    session.subscribe('filewithext.txt',listFile);
}

connection.open();
</script>
</body></html>

```

У меня в каталоге присутствует файл с расширением .py и файл с расширением .html: два моих клиента. Для демонстрации я создам в том же каталоге пустой текстовый файл с именем empty.txt. То есть мы должны каждую секунду получать три события.

Открыв веб-страницу, вы увидите в ней список, включающий файлы:

- empty.txt;
- directory_lister.py.

Добавив или удалив файлы, вы увидите, что этот список почти тут же изменится. Если создать другого клиента JS с иным набором подписок, он отобразит другой список файлов.

Итоги

В этой главе мы рассмотрели только крохотную часть из того, что можно сделать с WebSocket, Twisted, Autobahn и WAMP.

Попробуйте изменить приведенные примеры, чтобы расширить их возможности, и поэкспериментируйте с ними, чтобы лучше понять происходящее. Для более полного представления добавьте в код вывод диагностических сообщений.

В примерах использования WebSocket добавьте в раздел `if __name__ == "__main__"` следующие строки:

```
import sys
from twisted.python import log
log.startLogging(sys.stdout)
...
```

В примеры использования WAMP добавьте в тело класса `ApplicationSession`:

```
from twisted.logger import Logger
...
```

```
class TheAppClass(ApplicationSession):
    log=Logger()
    ...
```

А вот несколько идей для желающих заняться дальнейшими исследованиями:

- перепишите примеры в более современном стиле – с использованием конструкций `async/await`;
- попробуйте другие способы передачи сообщений, например потоки данных;
- увеличьте надежность своего кода, используя автоматическое восстановление подключения или балансировку нагрузки (только для комбинации Twisted/WAMP);
- напишите клиента на каком-нибудь другом языке: Java, C#, PHP; в результате вы получите клиентов WebSocket и WAMP для многих популярных платформ;
- поищите информацию о механизмах поддержки безопасности: SSL, Authentication, Permissions... Они сложны в настройке, но очень надежны;
- познакомьтесь поближе с другими особенностями Crossbar.io, такими как управление процессами, WSGI-сервер, статическая обработка файлов. Вы будете удивлены широтой доступных возможностей.

Глава 9

Создание приложений с `asyncio` и Twisted

Пакет `asyncio`, входящий в состав стандартной библиотеки Python, начиная с версии 3.4, стандартизирует API для реализации асинхронных сетевых программ, управляемых событиями. Кроме собственных сетевых примитивов и механизмов конкурентного выполнения, `asyncio` также определяет обобщенный интерфейс цикла событий для асинхронных библиотек и окружений. Эта обобщенная прослойка позволяет приложениям использовать Twisted и `asyncio` вместе в одном процессе.

В этой главе мы посмотрим, как объединить Twisted и `asyncio`, написав простой HTTP-прокси с `treq`, высокоуровневым HTTP-клиентом, основанным на Twisted; и `aihttp`, HTTP-клиентом и серверной библиотекой, основанной на `asyncio`.

Пакет `asyncio` и его экосистема продолжают развиваться. С увеличением числа тех, кто использует `asyncio`, постоянно разрабатываются новые API и реализуются новые идиомы. Поэтому наш HTTP-прокси является примером, а не рецептом интеграции Twisted и `asyncio`. Для начала мы познакомимся с фундаментальными понятиями, которые обеспечивают совместимость между ними и определяют дальнейшие возможности более тесной интеграции `asyncio` и Twisted в будущем.

ОСНОВНЫЕ ПОНЯТИЯ

Библиотека `asyncio` и Twisted имеют много общих деталей в дизайне и реализации, отчасти потому, что сообщество Twisted активно участвовало в разработке `asyncio`. Документ PEP 3156, описывающий `asyncio`, является продолжением документа PEP 3153, в свою очередь написанного членом команды разработчиков Twisted. Как результат `asyncio` заимствовала из Twisted понятия протоколов, транспортов, производителей и потребителей и предлагает окружение, хорошо знакомое программистам, которые используют Twisted в своей работе.

Однако это родство по происхождению никак не влияет на процесс интеграции библиотек, *использующих* asyncio, с библиотеками, использующими Twisted. Любая инфраструктура, управляемая событиями, должна реализовать два понятия, чтобы обеспечить связь между ними: *механизм обещаний* (*promise*), представляющих значения, которые станут доступны некогда в будущем, и *циклы событий*, обеспечивающие ввод/вывод.

МЕХАНИЗМ ОБЕЩАНИЙ

Вы уже знакомы с объектами Deferred в Twisted, позволяющими связать бизнес-логику и обработку ошибок со значениями, которые станут доступны позже. Объекты Deferred более широко известны в литературе по информатике как *обещания* (*promise*). Как рассказывалось в главе 2, такие обещания упрощают разработку программ, управляемых событиями, обеспечивая возможность внешней композиции обратных вызовов без специальной поддержки со стороны используемого языка программирования.

В asyncio основой механизма обещаний является класс asyncio.Future. В отличие от Deferred, экземпляры Future выполняют свои обратные вызовы *асинхронно*; Future.add_done_callback планирует выполнение обратного вызова в следующей итерации цикла событий. Сравните поведение классов Deferred и Future в следующем примере, при выполнении под управлением Python 3.4 или выше:

```
>>> from twisted.defer import Deferred
>>> d = Deferred()
>>> d.addCallback(print)
<Deferred at 0x1234567890>
>>> d.callback("value")
>>> value
>>> from asyncio import Future
>>> f.add_done_callback(print)
>>> f.set_result("value")
>>>
```

Оба метода – Deferred.addCallback и Future.add_done_callback – обеспечивают вызов указанной функции со значением, представленным соответствующей абстракцией обещания, когда оно станет доступно. Однако вызов Deferred.callback немедленно запускает все зарегистрированные обратные вызовы, тогда как Future.set_result откладывает все действия до следующей итерации цикла обработки событий.

С одной стороны, это исключает ошибки, связанные с повторным входом, вероятность которых существует в Deferred, потому что, используя asyncio, можно быть уверенным, что добавление обратного вызова не приведет к его немедленному запуску, даже если экземпляр Future уже имеет конкретное значение. С другой стороны, при применении asyncio необходимо учитывать, что код выполняется в цикле обработки событий, а это усложняет его использование и дизайн. Например: в каком таком цикле событий экземпляр Future с име-

нем `f` в примере выше запланировал обратный вызов `print`? Чтобы ответить на этот вопрос, мы должны рассмотреть систему обработки событий в `asyncio` и выяснить, чем она отличается от реактора Twisted.

Циклы событий

Как рассказывалось в главе 1, цикл обработки событий называется в Twisted *реактором*. В главе 3, для создания и настройки реактора для нашего приложения агрегирования каналов, мы использовали `twisted.internet.task.react` и фреймворк приложений Twisted. Оба этих способа получения реактора предпочтительнее, чем импортирование `twisted.internet.reactor`. Это объясняется тем, что выбор реактора зависит от контекста, в котором он используется; разные платформы предлагают свои примитивы мультиплексирования ввода/вывода, поэтому приложения Twisted, работающие в macOS, должны использовать `kqueue`, тогда как приложения в Linux должны использовать `epoll`; в тестах часто предпочтительнее использовать реакторы-заглушки, чтобы уменьшить общее потребление ресурсов; и, как мы увидим далее, приложения часто объединяют Twisted с другими фреймворками, запуская его внутри другого цикла обработки событий. Код, импортирующий реактор, вместо получения его в аргументах сам не может быть импортирован до выбора реактора, что значительно усложняет его использование. По этой причине в Twisted были добавлены такие методы, как `react`, упрощающие параметризацию приложений реакторами.

В отличие от Twisted, где пришлось разрабатывать новые API для управления выбором и установкой реактора, библиотека `asyncio` с самого начала включала *стратегии на основе цикла событий*. `asyncio` включает стратегию по умолчанию, которую разработчики могут заменить на `asyncio.set_event_loop_policy` и получить с помощью `asyncio.get_event_loop_policy`.

Стратегия по умолчанию связывает циклы событий с потоками выполнения; `asyncio.get_event_loop` возвращает цикл для текущего потока, создавая его при необходимости, а `asyncio.set_event_loop` устанавливает его.

Именно так наш пример с `Future` связал себя с циклом событий. Инициализатор `asyncio.Future` принимает цикл событий в именованном аргументе `loop`; если оставить в нем значение по умолчанию `None`, экземпляр `Future` извлечет текущий цикл по умолчанию вызовом `asyncio.get_event_loop`.

Исторически `asyncio` ожидала, что пользователь явно передаст текущий цикл событий там, где это необходимо, из-за чего ошибка в `get_event_loop` приводила к непредвиденному поведению, когда функция вызывалась ниже уровня модуля. Но начиная с версии Python 3.5.3 функция `get_event_loop` была перепирана и теперь надежно возвращает действующий цикл событий при обращении к ней из обратных вызовов. Более поздний код `asyncio` предпочитает явно вызывать `get_event_loop` вместо явной передачи ссылок через стек вызовов или через переменные экземпляра.

Циклы событий в `asyncio` отличаются от реакторов Twisted не только своей вездесущностью, но и функциональными возможностями. Например, реак-

торы могут *генерировать системные события* в определенных точках своего жизненного цикла. Фреймворк Twisted часто управляет ресурсами, выделяя их перед выполнением прикладного кода и явно освобождая перед завершением процесса с помощью `IReactorCore.addSystemEventTrigger`; например, время жизни пула потоков, используемого реализацией по умолчанию DNS в Twisted, тесно связано с временем жизни реактора событием `shutdown`. На момент написания этих строк циклы событий в `asyncio` не имели эквивалентного API.

РЕКОМЕНДАЦИИ

Из-за различий между `asyncio.Future` и `Deferred` в Twisted, а также между циклами событий двух библиотек при их объединении необходимо следовать определенным рекомендациям:

- 1) всегда запускайте реактор Twisted поверх цикла событий `asyncio`;
- 2) вызывая код `asyncio` из Twisted, преобразуйте экземпляры `Future` в экземпляры `Deferred` с помощью `Deferred.fromFuture`. Точно так же преобразуйте в `Deferred` сопрограммы `asyncio.Task`;
- 3) вызывая Twisted из `asyncio`, преобразуйте экземпляры `Deferred` в экземпляры `Future` с помощью `Deferred.asFuture`. Передавайте этому методу аргумент с активным циклом событий `asyncio`.

Первая рекомендация обусловлена тем, что `IReactorCore` имеет более богатый интерфейс, чем циклы событий в `asyncio`. Но чтобы понять суть второй и третьей рекомендаций, необходимо поближе познакомиться с сопрограммами `asyncio`, `Future` и `Task` и различиями между ними.

Выше мы видели, что объекты `Future` действуют подобно объектам `Deferred`. В главе 2 мы узнали, что *сопрограммы* – функции и методы, объявленные с помощью `async def`, – это особенность языка; они не связаны напрямую с `asyncio`, Twisted и любой другой библиотекой. Напомню, что сопрограмма может с помощью `await` дожидаться, пока разрешится *fututre-подобный объект*, а объекты `Deferred` как раз являются *fututre-подобными объектами*, поэтому сопрограмма может использовать `await`, чтобы дожидаться разрешения `Deferred`.

Совершенно очевидно, что экземпляры `asyncio.Future` тоже являются *fututre-подобными объектами*, поэтому сопрограммы также могут ждать их разрешения. Идиоматический код `asyncio` редко создает явные экземпляры `Future`, чтобы тут же приостановиться в ожидании, пока они разрешатся, вместо этого предпочтение отдается ожиданию других сопрограмм. Например:

```
>>> import asyncio
>>> from twisted.internet import defer, task, reactor
>>> aiosleep = asyncio.sleep(1.0, loop=asyncio.get_event_loop())
>>> txsleep = task.deferLater(reactor, 1.0, lambda: None)
>>> asyncio.iscoroutine(aiosleep)
True
>>> isinstance(txsleep, defer.Deferred)
True
```

`aiosleep` – это объект, который приостанавливает сопрограмму `asyncio` как минимум на одну секунду, `txsleep` делает то же самое для кода `Twisted`, который использует `Deferred`. Но если `txsleep` является самым обычным объектом `Deferred`, то `aiosleep` фактически является сопрограммой, окончания выполнения которой могут дожидаться другие сопрограммы с помощью `await`.

Чтобы `aiosleep` могла выполниться, ей *необходимо* передать инструкции `await`, как и любую другую сопрограмму. По этой причине сопрограммы не подходят для выполнения фоновых операций типа «запустил и забыл», которые должны действовать, не блокируя вызывающий код. `txsleep` – экземпляр `Deferred` – разрешится примерно через 1 секунду, независимо от того, подключены ли к нему какие-либо обработчики.

Библиотека `asyncio` предлагает свое решение в форме класса `Task`. Задача (экземпляр `Task`) обертывает сопрограмму экземпляром `Future` и ожидает его разрешения от имени создавшего ее кода. Задачи позволяют с помощью `asyncio.gather` одновременно ждать выполнения нескольких сопрограмм. Например, следующий код будет работать только четыре секунды вместо шести:

```
import asyncio

sleeps = asyncio.gather(asyncio.sleep(2), asyncio.sleep(4))
asyncio.get_event_loop().run_until_complete(sleeps)
```

Объекты `Deferred` можно связать с объектами `Future` с помощью `Deferred.fromFuture` и `asFuture`. Функции из `asyncio` для создания задач, например `asyncio.AbstractEventLoop.create_task` и `asyncio.ensure_future`, позволяют сопрограммам, ожидающим разрешения объектов `asyncio`, взаимодействовать с `Twisted` посредством интерфейса `Deferred`.

Порядок взаимодействий между `asyncio` и `Twisted` проще объяснить на примере. Следующий код демонстрирует, как воплотить все три наши рекомендации:

```
import asyncio
from twisted.internet import asyncioreactor

loop = asyncio.get_event_loop()
asyncioreactor.install(loop)
from twisted.internet import defer, task

originalFuture = asyncio.Future(loop=loop)
originalDeferred = defer.Deferred()
originalCoroutine = asyncio.sleep(3.0)

deferredFromFuture = defer.Deferred.fromFuture(originalFuture)
deferredFromFuture.addCallback(print, "from deferredFromFuture")
deferredFromCoroutine = defer.Deferred.fromFuture(
    loop.create_task(originalCoroutine))
deferredFromCoroutine.addCallback(print, "from deferredFromCoroutine")
futureFromDeferred = originalDeferred.asFuture(loop)
futureFromDeferred.add_done_callback(
    lambda result: print(result, "from futureFromDeferred"))
```

```
@task.react
def main(reactor):
    reactor.callLater(1.0, originalFuture.set_result, "1")
    reactor.callLater(2.0, originalDeferred.callback, "2")
    return deferredFromCoroutine
```

Сначала вызовом `asyncioreactor.install` настраивается реактор для Twisted. Эта функция принимает цикл событий `asyncio` в аргументе и связывает его с реактором Twisted. Как объяснялось выше, `asyncio.get_event_loop` запрашивает у глобальной (и в данном случае по умолчанию) стратегии создание цикла событий и его кеширование для извлечения последующими вызовами `get_event_loop`.

`originalFuture`, `originalCoroutine` и `originalDeferred` представляют три типа объектов, которые далее будут преобразованы в и из `Deferred: Future`, сопрограмму, ожидающую выполнения асинхронного кода, и `Deferred`.

Затем мы обертываем `originalFuture` экземпляром `Deferred` вызовом метода класса `Deferred.fromFuture` и добавляем обратный вызов `print`. Напомню, что в первом аргументе обратному вызову передается результат, полученный экземпляром `Deferred`, а в остальных – аргументы, переданные в `addCallback`.

Далее `originalCoroutine` обертывается экземпляром `Task` с помощью `create_task`, после чего он передается в `Deferred.fromFuture`. Потом выполняются те же действия, что и с `deferredFromFuture`.

Как мы видели выше, экземпляры `Future`, в отличие от `Deferred`, выполняются, только если работает цикл событий `asyncio`, при этом в каждый конкретный момент времени может выполняться несколько циклов событий `asyncio`. Поэтому, чтобы связать `originalDeferred` с `Future` через `asFuture`, требуется явная ссылка на цикл событий. После этого мы добавляем обратный вызов `print`, который выполнится, когда `originalDeferred`, а значит, и `futureFromDeferred`, разрешится некоторым значением. Задача немного осложняется тем, что `Future.add_done_callback` принимает только обратные вызовы с одним аргументом. Для преодоления этого ограничения мы используем лямбда-выражение, которое выводит результат и информативное сообщение.

Ни один из этих объектов не будет выполняться без цикла обработки событий, поэтому мы используем `task.react`, чтобы запустить реактор. Мы спланировали выполнение так, что `originalFuture` разрешится значением "1" через одну секунду, а `originalDeferred` – значением "2" через две секунды. Наконец, мы завершаем работу реактора, когда завершится `deferredFromCoroutine` и, следовательно, `originalCoroutine`.

Если запустить эту программу, она выведет следующие строки:

```
1 from deferredFromFuture
<Future finished result='2'> from futureFromDeferred
None from deferredFromCoroutine
```

Первую строку выведет обратный вызов `print`, добавленный в `deferredFromFuture`, вторую строку – обратный вызов в `futureFromDeferred` (обратите внимание,

что обратные вызовы в Future получают аргумент с соответствующим объектом Future) и третью строку – обратный вызов в deferredFromCoroutine.

Этот пример иллюстрирует все три рекомендации, соблюдение которых необходимо для интеграции asyncio и Twisted абстрактным способом, который трудно применять для решения реальных задач. Однако, как отмечалось выше, невозможно дать более конкретный совет, который был бы универсально применимым. Но теперь мы познакомились с основными игроками и можем на примере посмотреть, как они взаимодействуют.

ПРИМЕР: ПРОКСИ С АИОНТТР И TREQ

aiohttp (<https://aiohttp.readthedocs.io>) – достаточно зрелая библиотека для реализации клиентов и серверов HTTP с поддержкой asyncio, предназначенная для использования с версией Python 3.4 и выше.

treq – это высокоуровневая библиотека для реализации клиентов HTTP, с которой мы познакомились в главе 3, действующая поверх Twisted.

С помощью этих двух библиотек мы создадим простой HTTP-прокси. Клиенты, настроенные на работу с прокси-сервером, будут отправлять все запросы ему; а прокси-сервер будет пересылать их целевому серверу и возвращать ответы клиентам. Для общения с клиентами мы используем серверную часть aiohttp, а для получения страниц от их имени – treq.

Прокси-серверы HTTP используются для фильтрации и кеширования контента, а также для передачи HTTP-запросов POST, PUT и других. Мы же будем считать нашу работу выполненной, если нам удастся просто передать запросы GET от клиентов и вернуть им ответы!

Начнем с запуска простейшего сервера aiohttp. Создайте новое виртуальное окружение в версии Python 3.4 или выше, установите aiohttp, Twisted и treq, а затем запустите следующую программу:

```
import asyncio
from twisted.internet import asyncioreactor

asyncioreactor.install(asyncio.get_event_loop())

from aiohttp import web
from twisted.internet import defer, task

app = web.Application()

async def handle(request):
    return web.Response(text=str(request.url))

app.router.add_get('/{path:.*}', handle)

async def serve():
    runner = web.AppRunner(app)
    await runner.setup()
    site = web.TCPSite(runner, 'localhost', 8000)
    await site.start()
```

```
def asDeferred(f):
    return defer.Deferred.fromFuture(asyncio.ensure_future(f))

@task.react
@defer.inlineCallbacks
def main(reactor):
    yield asDeferred(serve())
    yield defer.Deferred()
```

Как и в предыдущем примере, мы начинаем с создания асинхронного реактора Twisted и заворачиваем в него кешированный цикл событий.

Затем мы импортируем модуль `web` из `aiohttp` и создаем приложение `Application` – базовую абстракцию веб-приложения, предлагаемую библиотекой. Добавляем в приложение маршрут с регулярным выражением, которому соответствуют все URL (`.*`), и добавляем сопрограмму `handle` в качестве обработчика. Эта сопрограмма принимает экземпляр `aiohttp.web.Request`, представляющий запрос клиента, и возвращает его URL.

Сопрограмма `serve` создает объекты `AppRunner` и `Site`, необходимые для настройки нашего приложения и привязки его к сетевому порту.

Наше приложение, его обработчик и сопрограмма `serve` заимствованы непосредственно из документации `aiohttp` и не изменятся, даже если мы вообще не будем использовать Twisted. Взаимодействие, запущенное добавлением асинхронного реактора, реализует функция `main`, которую вызывает `task.react`. Как обычно, это экземпляр `Deferred`, хотя на этот раз он использует `inlineCallbacks`. Эту функцию можно было бы реализовать как сопрограмму, используя определение `async def`, и преобразовать ее в `Deferred` с помощью `ensureDeferred`; но мы решили использовать `inlineCallbacks`, чтобы показать, как разные стили можно использовать взаимозаменяемо.

Вспомогательная функция `asDeferred` принимает сопрограмму или экземпляр `Future`. Затем она использует `asyncio.ensure_future` для преобразования полученного аргумента в `Future`; сопрограмма будет преобразована в `Task`, а `Future` – собственно в `Future`. Полученный результат можно затем передать в `Deferred.fromFuture`.

Мы используем эту функцию, чтобы завернуть сопрограмму `serve` в экземпляр `Deferred`, а затем навсегда заблокировать реактор в ожидании `Deferred`, который никогда не разрешится.

Запустив эту программу, мы получим простую эхо-службу, возвращающую URL. Если в браузере открыть страницу `http://localhost:8000`, вы увидите URL, который использовали для доступа к ней; добавив элементы пути, например `http://localhost:8000/a/b/c`, вы получите другой URL.

Теперь, получив основу, можем реализовать прокси-сервер:

```
import asyncio
from twisted.internet import asyncioreactor

asyncioreactor.install(asyncio.get_event_loop())

from aiohttp import web
```

```

from twisted.internet import defer, task

app = web.Application()

async def handle(request):
    url=str(request.url)
    headers = Headers({k: request.headers.getAll(k)
                       for k in request.headers})
    proxyResponse = await asFuture(treq.get(url, headers=headers))
    print("URL:", url,"code:", proxyResponse.code)
    response = web.StreamResponse(status=proxyResponse.code)
    for key, values in proxyResponse.headers.getAllRawHeaders():
        for value in values:
            response.headers.add(key.decode(), value.decode())
    await response.prepare(request)
    body = await asFuture(proxyResponse.content())
    await response.write(body)
    await response.write_eof()
    return response

app.router.add_get('/{path:.*}', handle)

async def serve():
    runner = web.AppRunner(app)
    await runner.setup()
    site = web.TCPSite(runner, 'localhost',8000)
    await site.start()

def asFuture(d):
    return d.asFuture(asyncio.get_event_loop())

def asDeferred(f):
    return defer.Deferred.fromFuture(asyncio.ensure_future(f))

@task.react @defer.inlineCallbacks
def main(reactor):
    yield asDeferred(serve())
    yield defer.Deferred()

```

Код выше имеет два отличия от минимальной реализации на основе aiohttp: функция `handle` и новая вспомогательная функция `asFuture`.

Функция `handle` извлекает целевой URL из запроса клиента. Напомню, что клиенты прокси-серверы HTTP передают целевой URL в строке запроса; aiohttp делает его доступным через `request.url`.

Затем из запроса aiohttp извлекаются все клиентские заголовки и преобразуются в экземпляр `twisted.web.http_headers.Headers`, чтобы затем включить их в исходящий запрос `treq`. Заголовки HTTP могут быть многозначными, поэтому aiohttp преобразует их в словарь с ключами, нечувствительными к регистру символов, каждый из которых может иметь множество значений; вызов `request.headers.getAll(key)` вернет список всех значений, соответствующих заданному ключу-заголовку. Этот словарь отображает ключи в списки значений, что соответствует инициализатору `Headers` в Twisted. Обратите внимание, что aiohttp декодирует заголовки в текст, тогда как заголовки в Twisted представле-

ны последовательностями байтов; к счастью, Twisted автоматически кодирует текстовые ключи и значения в байты.

После создания копии заголовков клиента, пригодной для использования с `treq`, мы посылаем запрос GET. На этом этапе цикл обработки событий в `asyncio` выполняет планирование сопрограммы `handle`, поэтому все, что передается в инструкцию `await`, должно быть совместимо с `asyncio`. Библиотека `treq`, однако, работает в терминах `Deferred`, разрешения которых *можно* дождаться, но при попытке запланировать их `asyncio` потерпит неудачу. Чтобы решить проблему, нужно завернуть `Deferred` в экземпляр `Future`, связанный с тем же циклом событий, в котором запланировано выполнение функции `handler`.

Именно это и делает вспомогательная функция `asFuture`. Поскольку в начале программы мы привязали наш реактор к глобальному циклу событий вызовом `get_event_loop`, все последующие вызовы `get_event_loop` будут возвращать тот же цикл. В том числе вызовы внутри `aihttp` и внутри нашего кода. Именно так `asFuture` связывает экземпляр-обертку `Future` с правильным циклом событий.

Как показано в нашем примере, `asyncio` ждет разрешения `Future`, обертывающего `Deferred`, точно так же, как Twisted ждет разрешения экземпляров `Deferred`. Соответственно, наш обработчик возобновляет выполнение и присваивает объект ответа, возвращаемый библиотекой `treq`, переменной `proxyResponse`. Сразу после этого мы выводим сообщение, описывающее URL и код состояния.

Затем мы создаем `aihttp.web.StreamResponse` и записываем в него код состояния, полученный при обращении к целевому URL, чтобы клиент получил тот же код, что и прокси-сервер. Потом производится обратная трансляция заголовков копированием ключей и значений из `Headers` в `StreamResponse`. Вызов `twisted.web.http_headers.Headers.getAllRawHeaders` возвращает ключи и значения заголовков в виде строк байтов, поэтому их нужно декодировать перед записью в `StreamResponse`.

Затем мы отправляем заголовки клиенту вызовом `StreamResponse.prepare`. Теперь осталось только получить и отправить тело, что мы и делаем, вызывая метод `content` экземпляра `Response`; этот метод тоже возвращает `Deferred`, поэтому мы заворачиваем его в `asFuture` для совместимости с `asyncio`.

Вот фрагмент вывода программы после настройки веб-браузера на использование ее в роли прокси-сервера и попытки открыть страницу `http://twisted-matrix.com/`:

```
URL: http://twistedmatrix.com/ code: 200
URL: http://twistedmatrix.com/trac/chrome/common/css/bootstrap.min.css code:200
URL: http://twistedmatrix.com/trac/chrome/common/css/trac.css code: 200
...
```

Итоги

В этой главе мы узнали, как объединить Twisted и `asyncio` в одном приложении. Поскольку эти две библиотеки используют общие идеи *обещаний* и *циклов событий*, мы можем запускать код Twisted поверх `asyncio`.

Чтобы использовать `asyncio` и `Twisted` в одном приложении, необходимо выполнить следующие три рекомендации: всегда запускайте реактор поверх цикла событий `asyncio`; при вызове методов `asyncio` из `Twisted` преобразуйте экземпляры `Future` в `Deferred` с помощью `Deferred.asFuture`; при вызове методов `Twisted` из `asyncio` выполняйте обратное преобразование с помощью `Deferred.fromFuture`.

Поскольку библиотека `asyncio` продолжает развиваться и совершенствоваться, невозможно дать более конкретные рекомендации по ее интеграции с `Twisted`. Вместо этого мы показали практическое применение всего, что узнали, на примере простого прокси-сервера HTTP, использующего `aihttp` и `treq`. Несмотря на минимализм, наш прокси-сервер достаточно близок к реальному приложению, чтобы вы могли научиться применять эти рекомендации и преодолеть разрыв между двумя методиками асинхронного программирования в Python.

Глава 10

Buildbot и Twisted

Buildbot – это фреймворк для автоматизации процессов сборки, тестирования и выпуска программного обеспечения. Он пользуется большой популярностью в организациях и проектах со сложными и необычными требованиями к сборке, тестированию и выпуску. Фреймворк имеет обширный набор настроек и поддерживает широкий выбор систем управления версиями, фреймворков сборки и тестирования и механизмов отображения состояния. Buildbot написан на Python, поэтому легко поддается расширению добавлением специализированных реализаций ключевых компонентов. Buildbot можно сравнить с Django: он обеспечивает основу для создания сложных, настраиваемых приложений, но его не так просто установить или использовать, как, например, Joomla или WordPress.

История появления BUILDBOT

Предшественник Buildbot был написан Брайаном Уорнером (Brian Warner) в 2000–2001 годах, когда он работал в компании по производству маршрутизаторов. Он и его коллеги устали от каждодневных хлопот по проверке кода в CVS, который работал на машинах с ОС Solaris, но не работал на машинах с Linux.

Первоначально исходный код фреймворка был закрытым и использовал пакеты `asynctest` и `pickle` для реализации системы RPC, в которой все действия выполняли рабочие процессы. Центральный процесс сборки принимал информацию о состоянии от рабочих процессов и выводил ее в динамическую веб-страницу. Он был создан по образу и подобию Tinderbox компании Mozilla.

В процессе поиска примеров использования `asynctest` Брайан наткнулся на фреймворк Twisted и нашел его более продвинутым и быстро развивающимся. Покинув компанию в начале 2002 года, он переписал реализацию системы сборки в процессе изучения Twisted, и в результате появился фреймворк Buildbot.

Примерно до 2009 года Buildbot не поддерживал базы данных. До этого базы данных были довольно сложны в развертывании, и хранение данных непосредственно на диске не было редкостью и считалось эффективным решением. Все еще только начиналось: диски были быстрыми, сети – медленными,

а «большие» приложения непрерывной интеграции выполняли всего лишь десятки параллельных сборок.

Начиная с 2009 года фреймворк Buildbot начала использовать компания Mozilla, и ее потребности быстро превысили возможности этой простой модели. В течение нескольких лет в Mozilla работали тысячи рабочих и более 50 центральных процессов. Для поддержки таких широкомасштабных требований они пригласили Брайана и предложили добавить поддержку баз данных, чтобы центральные процессы сборки могли координировать свою работу. Новая реализация не сохраняла результаты сборки в базе данных – они оставались в файлах pickle отдельных ведущих процессов.

Веб-интерфейс был полностью синхронным и отображал статические HTML-представления результатов сборки. Кроме того, создание некоторых страниц могло блокировать процесс сборки на несколько минут, пока шла загрузка результатов из базы данных и файлов pickle. В Mozilla простой просмотр динамической страницы с результатами мог вызвать сбой всего процесса, поэтому доступ к таким страницам был запрещен.

Примерно в это же время Дастин Митчелл (Dustin Mitchell) взялся за обслуживание проекта и организацию усилий по модернизации приложения. Эта попытка завершилась выпуском Buildbot 0.9.0 в октябре 2016 года. Целью проекта было преобразование Buildbot в серверное приложение с поддержкой базы данных и интерактивным веб-интерфейсом. В конфигурации с несколькими ведущими процессами результаты сборки теперь доступны всем ведущим процессам и обновляются динамически, по мере поступления результатов от рабочих процессов. HTTP API поддерживает интеграцию с другими инструментами непрерывной интеграции, а новые асинхронные интерфейсы допускают подключение сторонних плагинов.

Развитие этой версии продвигалось тяжело, на ее разработку потребовалось почти полдесятилетия напряженного труда команды разработчиков, в которую вошли: Пьер Тарди (Pierre Tardy), Том Принс (Tom Prince), Амбер Юст (Amber Yust) и Михаил Соболев (Mikhail Sobolev). Зато в ней было решено множество сложных проблем, связанных с асинхронным выполнением кода на Python, как будет описано далее в этой главе.

Эволюция асинхронного выполнения кода на Python в Buildbot

В Twisted уже имелась хорошая поддержка протоколов, в том числе Perspective Broker, когда Брайан начал писать Buildbot. Реакторы и объекты Deferred были хорошо отлажены и базировались на прочной теоретической основе. Тем не менее асинхронное выполнение все еще оставалось относительно новой концепцией для подавляющего большинства разработчиков, и асинхронный код часто называли «Twisted Python»¹.

¹ Словосочетание Twisted Python можно перевести как «перекрученный, переплетенный, перевитый код на Python». – *Прим. перев.*

Рассмотрим для примера метод `Builder.startBuild` из `Buildbot`, существовавший примерно в 2005 году (позже он был переписан). Он последовательно выполнял две асинхронные операции: сначала проверял связь с выбранным рабочим процессом, а затем вызывал его метод `startBuild`. Все эти действия выполнялись с помощью нескольких методов экземпляра:

```
# buildbot/process/builder.py @ 41cdf5a
class SlaveBuilder(pb.Referenceable):
    def attached(self, slave, remote, commands):
        # ...
        d = self.remote.callRemote("setMaster",self)
        d.addErrback(self._attachFailure,"Builder.setMaster")
        d.addCallback(self._attached2)
        return d

    def _attached2(self, res):
        d = self.remote.callRemote("print","attached")
        d.addErrback(self._attachFailure,"Builder.print 'attached'")
        d.addCallback(self._attached3)
        return d

    def _attached3(self, res):
        # теперь можно сказать, что они действительно подключены
        return self

    def _attachFailure(self, why, where):
        assert type(where) is str
        log.msg(where)
        log.err(why)
        return why
```

Этот неуклюжий синтаксис требовал передачи переменных через несколько методов, поток управления читался с большим трудом и загрязнял пространство имен методов. Это привело к появлению большого числа странных проблем с таинственно исчезающими необработанными ошибками или выполнением обратных вызовов в неожиданном порядке. В условных выражениях и циклах, использующих асинхронные операции, было очень трудно разобратся и, соответственно, правильно отладить их.

В настоящее время мы привыкли делить функции на синхронные (возвращающие конкретное значение) и асинхронные (возвращающие экземпляры `Deferred`). Но в те мрачные времена деление не было таким четким, и в `Buildbot` имелись функции, которые могли возвращать и экземпляры `Deferred`, и конкретные значения, в зависимости от обстоятельств. Излишне говорить, что такие функции трудно было вызвать правильно, поэтому впоследствии они были реорганизованы в строго синхронные или асинхронные.

По мере развития `Twisted` и, что более важно, с появлением в `Python` новых возможностей, таких как генераторы, декораторы и выражения `yield`, ситуация постепенно улучшалась. `deferredGenerator` в `Twisted` позволял записывать поток управления в обычном для `Python` стиле с помощью операторов `if`, `while` и `for`. Но синтаксис все еще оставался неуклюжим – для выполнения асинхронной

операции требовалось написать три строки кода, и если какую-то из них опустить, происходил неясный сбой:

```
# buildbot/buildslave/base.py @ 8b4e7a9
class BotBase(service.MultiService):
    @defer.deferredGenerator
    def remote_setBuilderList(self, wanted):
        retval = {}
        # ...
        dl = defer.DeferredList([
            defer.maybeDeferred(self.builders[name].disownServiceParent)
            for name in to_remove])
        wfd = defer.waitForDeferred(dl)
        yield wfd
        wfd.getResult()
        # ...
        yield retval # вернуть значение
```

С выходом версии Python 2.5 и появлением выражений `yield` разработчики Twisted реализовали декоратор `inlineCallbacks`. Он похож на `deferredGenerator`, но позволяет организовать выполнение асинхронной операции единственной строкой кода:

```
# master/buildbot/data/buildrequests.py @ 8b4e7a9
class BuildRequestEndpoint(Db2DataMixin, base.Endpoint):
    @defer.inlineCallbacks
    def get(self, resultSpec, kwargs):
        buildrequest = yield self.master.db.buildrequests.getBuildRequest(k
        wargs['buildrequestid']
        if buildrequest:
            defer.returnValue((yield self.db2data(buildrequest)))
        defer.returnValue(None)
```

Это намного более дружелюбное решение, за исключением того, что при его использовании легко забыть вернуть `Deferred`. Такие ошибки приводят к тому, что асинхронные операции выполняются «параллельно» с вызывающими функциями и часто не имеют никаких проблем, пока не завершатся неудачей, и вызывающая функция продолжит выполнение как ни в чем не бывало. Несколько таких коварных ошибок преодолели довольно обширное тестирование и сохранялись на протяжении нескольких версий Buildbot.

С переходом на Python 3 с новым синтаксисом `async/await` Twisted и Buildbot получили более естественный способ реализации асинхронного кода, хотя и не избавились от проблемы забытой инструкции `await`. С этим синтаксисом функция, представленная выше, читается еще более естественно:

```
class BuildRequestEndpoint(Db2DataMixin, base.Endpoint):
    async def get(self, resultSpec, kwargs):
        buildrequest = await self.master.db.buildrequests.getBuildRequest
            (kwargs['buildrequestid'])
        if buildrequest:
            return (await self.db2data(buildrequest))
        return None
```

Исторически асинхронный код использовался в Python только в сетевых приложениях, где важна высокая производительность, а большинство приложений на Python следовало простой синхронной модели. Сообщество NodeJS показало, что стандартизированная, совместимая асинхронность позволяет создать динамичную экосистему свободно комбинируемых библиотек, утилит и структур. Теперь в Python есть `async/await`, а библиотека `asyncio` позволяет коду, использующему Twisted, взаимодействовать с кодом, применяющим другие асинхронные окружения, облегчая развитие аналогичной экосистемы.

Миграция синхронных API

На ранних этапах Buildbot работал как единый процесс и сохранял свое состояние в файлах `pickle` на диске. Действия с файлами выполнялись синхронно, поэтому большинство операций в Buildbot производились без участия объектов `Deferred`.

Приблизительно в 2010 году, когда непрерывная интеграция начала набирать популярность в сообществе разработчиков программного обеспечения и стало расти число установок Buildbot, файлы `pickle` превратились в узкое место. Пришло время добавить поддержку полноценной базы данных, и мы столкнулись с выбором: реорганизовать все функции для работы с базой данных так, чтобы они возвращали объекты `Deferred`, или производить обращения к базе данных синхронно из основного потока выполнения, блокируя другие операции до их завершения. Первый вариант выглядел более привлекательным, но если изменить какую-то функцию так, чтобы она возвращала объект `Deferred`, тогда все другие функции, которые ее вызывают, тоже придется изменить и заставить их возвращать объекты `Deferred`, и так по всей базе кода. Buildbot – это фреймворк, и во многих проектах, использующих его, имеется масса пользовательского кода, вызывающего его функции. Если изменить функции так, чтобы они возвращали объекты `Deferred`, пользователям фреймворка придется переписать и повторно протестировать свой код.

В интересах целесообразности мы решили выполнять большинство обращений к базе данных в главном потоке. Большая часть информации о состоянии сборки – результаты, этапы и журналы – все так же сохранялась на диске. Это позволило нам вовремя реализовать необходимую функциональность, но она страдала предсказуемыми проблемами с производительностью. На самом деле в более крупных проектах, таких как Mozilla, запросы к базе данных могут задерживать ведущий процесс настолько, что рабочие процессы будут останавливаться по тайм-ауту, останавливать сборку и пытаться повторно установить соединение.

Эта ситуация повторилась во многих других местах в Buildbot, потому что новая функциональность была добавлена в код, который когда-то был простым и синхронным. Если бы мы могли начать все сначала, не будучи ограниченными требованиями к обратной совместимости, мы бы сделали каждый

общедоступный метод асинхронным и принимали Deferred в каждом вызове пользовательского кода.

Этапы асинхронной сборки

Этапы сборки было сложнее всего перевести на асинхронные рельсы. Buildbot реализует ряд «стандартных» этапов сборки, но мы также позволяем пользователям выполнять свои собственные этапы. Такие пользовательские этапы сборки вызывают ряд методов, которые выполняют сам этап, выводят информацию в журнал, обновляют состояние и т. д. Исторически все эти вызовы были синхронными, потому что прежде они обновляли состояние в памяти, которое позже записывалось на диск.

В версии Buildbot 0.9 все эти структуры данных, хранящиеся на диске, были удалены, и вся информация стала храниться в базе данных. Она также поддерживала «оперативные» обновления, поэтому кеширование результатов этапа сборки до его завершения было невозможно. В результате все синхронные методы обновления состояния стали асинхронными – но существующие пользовательские этапы сборки вызывали их синхронно!

Мы выбрали необычное решение данной проблемы: мы определили этапы сборки «старого» (синхронного) и «нового стиля» с разным поведением. При выполнении этапов сборки старого стиля Buildbot собирает все необработанные объекты Deferred и, когда этап завершается, ждет, пока все они разрешатся. Поскольку большинство методов предоставляют информацию о ходе выполнения этапа, вызывающий код не предполагает получение какого-либо возвращаемого значения. Мы добавили простой метод, различающий старые и новые реализации этапов сборки и активирующий механизм совместимости только для старых этапов. Стратегия оказалась чрезвычайно успешной, а в редких случаях, когда она терпела неудачу, проблема решалась просто: нужно было лишь переписать этап в новом стиле.

Мы разработали этот механизм совместимости перед тем, как переписать встроенные этапы сборки в «новом» стиле. Это дало возможность протестировать и доработать механизм, прежде чем приступить к реализации встроенных этапов в более надежном новом стиле.

Код BUILDBOT

Фреймворк Buildbot – необычный кандидат для реализации в асинхронном стиле. Большинство таких приложений опираются на цикл запрос/ответ, тогда как асинхронное программирование предполагает гораздо более высокую степень параллелизма, чем синхронная модель на основе потоков выполнения. С другой стороны, Buildbot поддерживает долговременные соединения между ведущим и ведомыми (рабочими) процессами и выполняет последовательные операции в этих рабочих процессах. Даже процедура приема нового соединения от рабочего процесса предусматривает выполнение сложной последова-

тельности операций проверки дубликатов рабочих процессов, опроса их возможностей и настройки для выполнения сборки.

Синхронный подход к созданию подобных приложений предполагает запуск потока выполнения для каждого рабочего процесса плюс потоки выполнения для вспомогательных объектов, таких как планировщики или источники изменений. Даже по самым скромным оценкам такая архитектура может порождать тысячи потоков выполнения, со всеми вытекающими проблемами планирования и конкуренции.

Асинхронные утилиты

Фреймворк Twisted предлагает широкий спектр удобных асинхронных инструментов, однако в Buildbot обнаружилось несколько вариантов поведения, не поддерживаемых этими инструментами. Подобно тому как очереди и блокировки поддерживают создание синхронных многопоточных потоковых приложений, эти инструменты поддерживают создание асинхронных приложений.

«Дребезг»

Ведущий процесс Buildbot может взаимодействовать с сотнями рабочих процессов, получать события с обновленным состоянием и журналируемые данные. Часто эти события легко поддаются объединению – например, несколько строк журналируемых данных можно объединить в один блок, – но должны обрабатываться максимально быстро для поддержки оперативного журналирования и динамического обновления состояния.

Решение состоит в том, чтобы устранить «дребезг» таких событий и вызывать обработчик только один раз, если однотипные события быстро следуют друг за другом. Механизм устранения дребезга вводит задержку и гарантирует вызов декорированного метода не менее одного раза в течение этого периода и может объединить несколько событий, поступивших в течение этого интервала.

В этом механизме периодически могут возникать ошибки, приводящие к вызову метода, когда в этом нет никакого смысла. Например, бессмысленно продолжать добавлять строки в журнал этапа сборки, если этот этап был отмечен как завершенный. Чтобы избежать подобной проблемы, декорированные методы могут воспользоваться методом «stop», который дождетсЯ (асинхронно) завершения любых выполняющихся вызовов, обеспечив тем самым четкий переход между состояниями.

Асинхронные службы

Buildbot основан на превосходном фреймворке Twisted Application Framework, который предлагает (кроме всего прочего) интерфейсы IService и IServiceCollection, позволяющие создавать иерархии служб. Buildbot реализует службу buildmaster на основе этой иерархии и добавляет в нее средства управления рабочими процессами, источниками изменений и т. д. в виде дочерних служб.

Рабочие процессы и источники изменений добавляются как потомки соответствующих служб управления.

Такая организация имеет решающее значение для структуры Buildbot, особенно в поддержке запуска и завершения приложений. Что еще более важно, она позволяет фреймворку Buildbot динамически перенастраивать себя во время выполнения. Например, если в конфигурацию добавить дополнительный рабочий процесс, тогда процедура перенастройки создаст новую службу и добавит как дочерний элемент в службу управления рабочими процессами.

В фреймворке Application есть только одна проблема: `startService` действует синхронно.

Так как у нас есть службы, реализующие операции с базой данных и очередями сообщений, очень важно, чтобы они запускались фреймворком Application в правильном порядке. При соблюдении этого порядка мы можем быть уверены, что все рабочие процессы, сборщики и т. д. правильно регистрируются в базе данных и подключаются к соответствующим очередям сообщений, прежде чем мы начнем рассылать запросы на сборку. Например, когда процедура перенастройки запустит новый рабочий процесс, этот процесс должен быть добавлен в базу данных. Фактически рабочий процесс не запустится, пока эта асинхронная операция не завершится.

Инициализацию зависимостей можно рассматривать как задачу, независимую от настройки зависимых служб, но для нас было довольно удобно сделать функцию `startService` асинхронной.

```
class AsyncMultiService(AsyncService, service.MultiService):
    def startService(self):
        service.Service.startService(self)
        dl = []
        # если в ходе перенастройки служба подключает другую службу,
        # тогда служба должна быть запущена дважды, поэтому мы
        # не используем итераторы, а копируем в список
        for svc in list(self):
            # обработать любые объекты Deferred, игнорируя ошибки
            # и успешное выполнение
            dl.append(defer.maybeDeferred(svc.startService))
        return defer.gatherResults(dl, consumeErrors=True)
    [...]
```

Buildbot добавляет класс `AsyncMultiService`, наследующий `MultiService`, поддерживающий асинхронные методы `startService` для дочерних служб. Он обрабатывает крайние случаи, возникающие при добавлении и удалении служб, а это означает, что `addService`, `setServiceParent` и `disownServiceParent` также становятся асинхронными.

У нас появилась роскошная возможность переписать этот механизм, потому что все вызовы `addService` и `startService` находятся под нашим полным контролем. Внести эти изменения в сам фреймворк Twisted не получится без создания совершенно новой, взаимно несовместимой иерархии классов.

Фактически, поскольку Twisted выполняет вызов метода `startService` службы верхнего уровня, при обработке асинхронного поведения требуется проявлять некоторую осторожность. На верхнем уровне Buildbot находится служба `BuildMaster`. Ее метод `startService` возвращает экземпляр `Deferred`, который никогда не разрешится ошибкой, и использует `try/except`, чтобы перехватить любые ошибки и остановить реактор. Поскольку реактор на начальном этапе еще не запущен, `startService` приостанавливается, ожидая запуска реактора:

```
class BuildMaster(...):
    @defer.inlineCallbacks
    def startService(self):
        [...]
        # мы ждем запуска реактора, поэтому смело можем вызвать
        # reactor.stop() в случае фатальных ошибок
        d = defer.Deferred()
        self.reactor.callWhenRunning(d.callback, None)
        yield d

        startup_succeed = False
        try:
            [...]
        except:
            f = failure.Failure()
            log.err(f, 'while starting BuildMaster')
            self.reactor.stop()
```

Наша система не очень хорошо обрабатывает зависимости между службами одного уровня. Например, `WorkerManager` зависит от `MessageQueueConnector`, но обе они являются дочерними для `masterService`. Служба `MessageQueueConnector` управляет внешней очередью сообщений и не может принимать сообщения или запросы на регистрацию, пока не будет установлено соединение с брокером (диспетчером) очереди. Возможность отправки запросов на регистрацию нужна службе `WorkerManager`. Обе службы запускаются параллельно, так как являются потомками одной и той же родительской службы. В настоящее время эта проблема решается сохранением любых сообщений или запросов на регистрацию во временной очереди, пока соединение не будет установлено. Мы могли бы улучшить систему, добавив слой инициализации зависимостей, не зависящий от иерархии служб. Однако сконструировать такую систему очень непросто, если вы хотите иметь эффективный и простой интерфейс, не требующий переделки методов `startService` всех служб.

Альтернативный дизайн, используемый в классе `ClientService`, появившемся в Twisted 16.1.0, предполагает немедленный возврат из `startService`, что позволяет процедуре запуска выполняться параллельно. Это решение требует, чтобы запуск службы не мог завершиться сбоем или предоставлением какого-то другого механизма сообщения о сбое. При перенастройке во время выполнения Buildbot полагается на механику обработки ошибок в `AsyncMultiService`, которая должна привести к изящному завершению, если новая

конфигурация содержит ошибку. `ClientService`, напротив, повторяет попытки соединения до бесконечности, поэтому процесс запуска никогда не завершается сбоем, даже если имеется неустраняемая ошибка. Подход с немедленным возвратом также требует внимательного отношения к ситуациям, когда метод службы может быть вызван до завершения процедуры запуска, обычно эта проблема решается путем защиты каждого метода ожиданием завершения запуска.

Кеш LRU

Кеширование имеет большое значение для масштабирования любого приложения, и Buildbot не исключение. Самой распространенной считается стратегия вытеснения наиболее давно использовавшихся записей (Least Recently Used, LRU), когда из кеша удаляются дольше всего не использовавшиеся записи, если требуется освободить место для новых записей. Ситуацию, когда в ответ на запрос можно вернуть данные из кеша, называют «попаданием в кеш»; а ситуацию, когда для извлечения данных требуется обратиться к их источнику, называют «промахом кеша».

Кеши LRU получили широкое распространение, и в каталоге PyPI можно было найти несколько реализаций. Но в то время все они были синхронными и предназначались для использования в многопоточном окружении.

В асинхронной реализации промах кеша должен включать ожидание выборки, в течение которой могут поступить другие запросы для той же записи. Эти запросы должны не инициировать дополнительные запросы на выборку, а ждать завершения первого запроса на выборку. Это требует пристального внимания к работе с объектами `Deferred`, особенно в части обработки ошибок.

Отложенный вызов функций

Во многих случаях бывает желательно вызвать какую-либо функцию, но нас не интересует результат ее выполнения или когда именно она будет вызвана. В асинхронной системе такие функции лучше вызывать позже, после завершения текущей итерации реактора. Это позволяет более справедливо распределять работу, поскольку реактор может обрабатывать другие события перед вызовом функций.

Самое простое решение – вызвать `reactor.callLater(0, callableForLater)`; это эквивалентно вызову `process.nextTick` в Node. Однако подобный подход имеет свой недостаток – сложность тестирования. В зависимости от особенностей выполнения теста функция `callableForLater` может не успеть завершиться до окончания теста, что может приводить к периодическим ошибкам при тестировании. Также этот подход не позволяет обрабатывать исключения или ошибки, возникающие в `callableForLater`.

Метод `buildbot.util.eventual.eventually` в Buildbot служит оберткой вокруг `reactor.callLater`. Также в модуле `buildbot.util.eventual` имеется дополнительный метод `flushEventualQueue`, который можно использовать в тестах для ожидания

завершения всех отложенных вызовов функций. Он регистрирует ошибки, возникающие в вызываемых функциях, в журнале Twisted.

Взаимодействие с синхронным кодом

В отличие от экосистемы JS, асинхронность не является стандартным и единственным способом выполнения операций ввода/вывода в Python. Со временем экосистема Python обросла большим количеством надежных и полезных библиотек, большинство из которых действуют синхронно. Buildbot как инструмент интеграции должен поддерживать все эти библиотеки.

Мы разработали несколько рекомендаций по использованию таких синхронных библиотек из нашего асинхронного ядра.

SQLAlchemy

SQLAlchemy – хорошо известная библиотека, помогающая отделить код SQL от программного кода на Python. Она поддерживает несколько диалектов SQL и упрощает поддержку баз данных. Библиотека SQLAlchemy предлагает свой предметно-ориентированный язык (Domain Specific Language, DSL) для описания запросов SQL, позволяющий хранить и повторно использовать фрагменты SQL, а также автоматически защищающий запросы от атак вида «инъекция SQL».

В настоящее время Buildbot поддерживает SQLite, MySQL и PostgreSQL.

SQLAlchemy поддерживает понятие пула соединений с базой данных; движок SQL повторно использует свое соединение с базой данных от запроса к запросу. В Buildbot мы отображаем этот пул соединений в пул потоков выполнения, благодаря чему каждая операция с базой данных выполняется внутри своего потока выполнения.

Все операции с базами данных реализованы в выделенном модуле db и следуют единому шаблону.

- Компонент поддержки базы данных должен наследовать `buildbot.db.base.DBConnectorComponent`.
- Каждый общедоступный метод предполагает вызов из асинхронного кода и возвращает `Deferred`.
- Для обращения к области видимости асинхронного метода из синхронного кода мы используем вложенные функции, чтобы избежать передачи параметров.
- Переход из асинхронного контекста в синхронный выполняется с помощью `self.db.pool.do(..)`.
- В имена функций или методов, которые используют блокировки, мы всегда добавляем префикс `thd`.

```
class StepsConnectorComponent(base.DBConnectorComponent):
```

```
    def getStep(self, stepid=None, buildid=None, number=None, name=None):
        # создать дескриптор для доступа к таблице в базе данных
```

```

tbl = self.db.model.steps

# для ускорения выхода в случае ошибки мы предварительно вычисляем
# запрос внутри главного потока
if stepid is not None:
    wc = (tbl.c.id == stepid)
else:
    if buildid is None:
        return defer.fail(RuntimeError(
            'must supply either stepid or buildid'))
    if number is not None:
        wc = (tbl.c.number == number)
    elif name is not None:
        wc = (tbl.c.name == name)
    else:
        return defer.fail(RuntimeError(
            'must supply either number or name'))
    wc = wc & (tbl.c.buildid == buildid)

# эта функция может появиться в профилировщике, поэтому
# ей лучше дать осмысленное имя
def thdGetStep(conn):
    q = self.db.model.steps.select(whereclause=wc)
    # следующая строка выполняет синхронную операцию ввода/вывода
    # и может заблокироваться. Именно поэтому необходим пул потоков.
    res = conn.execute(q)
    row = res.fetchone()

    rv = None
    if row:
        rv = self._stepdictFromRow(row)
    res.close()
    return rv
return self.db.pool.do(thdGetStep)

```

requests

Многие инструменты, с которыми взаимодействует Buildbot, управляются посредством HTTP API. Фреймворк Twisted имеет свою клиентскую библиотеку `twisted.web.client`, подобную `urllib` в Python. Однако нам больше понравилась замечательная библиотека `python-requests`. Она имеет простой и мощный API, использует принцип преимущества соглашений перед конфигурацией (отсюда и девиз «HTTP для людей»), поддерживает пул соединений и работу через прокси и, что особенно важно для надежности, автоматически выполняет повторные попытки.

Естественно, многие программисты на Python пожелают использовать подобные API в Buildbot. Но `requests` имеет синхронный API, потому что людям нравится синхронность.

Существует также библиотека `treq`, реализующая API запросов с использованием клиента Twisted, но она пока не обладает всеми функциями обеспечения надежности запросов.

Первоначально в сообществе Buildbot была написана библиотека `txrequests` – простая обертка вокруг сеанса `requests`, в котором все запросы выполняются в `ThreadPool`, по аналогии с `SQLAlchemy`. Также для Buildbot реализован класс `HttpClientService`, который абстрагирует API запросов и позволяет выбирать на роль низкоуровневой реализации `req` или `txrequests`.

В `HttpClientService` было реализовано несколько важных функций, которые стали результатом обобщения нашего опыта разработки кода с использованием `txrequests`: они абстрагируют различия между двумя реализациями. Класс `HttpClientService` включает также инфраструктуру модульного тестирования, которая позволяет тестировать компоненты без привлечения фиктивного HTTP-сервера. Он также поддерживает совместное использование сеансов несколькими компонентами, поэтому, например, два компонента, взаимодействующих с GitHub, могут использовать один и тот же сеанс HTTP.

```
class GitHubStatusPush(http.HttpStatusPushBase):
    @defer.inlineCallbacks
    def reconfigService(self, token, startDescription=None,
                       endDescription=None, context=None, baseUrl=None,
                       verbose=False, **kwargs):
        yield http.HttpStatusPushBase.reconfigService(self, **kwargs)
        [...]
        self._http = yield httpclientservice.HTTPClientService.getService(
            self.master, baseUrl, headers={
                'Authorization': 'token ' + token,
                'User-Agent': 'Buildbot'
            },
            debug=self.debug, verify=self.verify)
        self.verbose = verbose
        [...]
    def createStatus(self,
                    repo_user, repo_name, sha, state, target_url=None,
                    context=None, issue=None, description=None):
        payload = {'state': state}
        if description is not None:
            payload['description'] = description

        if target_url is not None:
            payload['target_url'] = target_url

        if context is not None:
            payload['context'] = context

        return self._http.post(
            '/'.join(['/repos', repo_user, repo_name, 'statuses', sha]),
            json=payload)
        [...]

class TestGitHubStatusPush(unittest.TestCase, ReporterTestMixin):
    [...]
    @defer.inlineCallbacks
```

```

def setUp(self):
    self.master = fakemaster.make_master(testcase=self,
                                          wantData=True, wantDb=True,
                                          wantMq=True)

    yield self.master.startService()
    # getFakeService исправит HTTPClientService и гарантирует, что
    # любая дальнейшая конфигурация HTTPClientService будет иметь
    # те же аргументы.
    self._http = yield fakehttpclientservice.HTTPClientService.
        getFakeService(
            self.master, self,
            HOSTED_BASE_URL, headers={
                'Authorization': 'token XXYZZ',
                'User-Agent': 'Buildbot'
            },
            debug=None, verify=None)
    self.sp = GitHubStatusPush('XXYZZ')
    yield self.sp.setServiceParent(self.master)

@defer.inlineCallbacks
def test_basic(self):
    build = yield self.setupBuildResults(SUCCESS)
    # гарантировать правильный вызов txrequests
    self._http.expect(
        'post',
        '/repos/buildbot/buildbot/statuses/d34db33fd43db33f',
        json={'state': 'pending',
              'target_url': 'http://localhost:8080/#builders/79/builds/0',
              'description': 'Build started.',
              'context': 'buildbot/Builder0'})

# этот код создаст и проверит http-запрос
against expectations
self.sp.buildFinished(build)

```

Docker

Еще одним примером используемых нами библиотек может служить стандартная библиотека `docker` для Python. Это еще одна синхронная библиотека, использующая `python-requests` для реализации протокола Docker HTTP.

Протокол Docker сложен и может часто меняться, поэтому мы решили отказаться от пользовательской реализации своего клиента на основе `HTTPClientService`. Но стандартная библиотека `docker` является синхронной, поэтому нам пришлось обернуть ее так, чтобы она не блокировала работу главного потока выполнения.

С этой целью мы использовали `twisted.internet.threads.deferToThread`. Эта вспомогательная функция использует общий пул потоков выполнения по умолчанию, которым автоматически управляет Twisted.

```

class DockerBaseWorker(AbstractLatentWorker): [...]
    def stop_instance(self, fast=False):

```

```

    if self.instance is None:
        # Для предосторожности. Возможно, что-то пытается предупредить,
        # что экземпляр никогда не будет подключен, потому что мы пока
        # ничего не запустили.
        return defer.succeed(None)

    instance = self.instance
    self.instance = None
    return threads.deferToThread(self._thd_stop_instance, instance, fast)

def _thd_stop_instance(self, instance, fast):
    docker_client = self._getDockerClient()
    log.msg('Stopping container %s... ' % instance['Id'][:6])
    docker_client.stop(instance['Id'])
    if not fast:
        docker_client.wait(instance['Id'])
    docker_client.remove_container(instance['Id'], v=True, force=True)
    if self.image is None:
        try:
            docker_client.remove_image(image=instance['image'])
        except docker.errors.APIError as e:
            log.msg('Error while removing the image: %s ', e)

```

Конкурентный доступ к общим ресурсам

Конкурентное программирование – это сложная область с большим количеством ловушек. Запуская несколько программ параллельно, вы должны гарантировать, что они не будут одновременно работать с одними и теми же данными. В Twisted легко одновременно запустить несколько экземпляров одной и той же функции в двух разных цепочках Deferred (а также в генераторах или со-программах inlineCallbacks). Эта типичная проблема называется «повторным входом». Конечно, в асинхронном программировании данная функция не будет запускаться дважды одновременно. Она выполняется в потоке реактора. Поэтому можно выполнять любые операции чтения/изменения/записи общего состояния, не заботясь о конкуренции.

Это верно... за исключением следующих случаев.

yield как барьер конкуренции

Можно считать, что Twisted реализует кооперативную многозадачность, пока дело не доходит до операций ввода/вывода. В этом случае yield, await и d.addCallback() становятся вашими барьерами конкуренции – эти инструкции не должны изменять общего состояния.

```

class MyClass(object):
    [...]
    # Следующую функцию нельзя запускать сразу несколько раз,
    # потому что она изменяет атрибут self.data между "yield"
    # Эта функция небезопасна для повторного входа
    # (т. е. нереентерабельная).
    def unsafeFetchAllData(self, n):

```

```

self.data = []
for i in range(n):
    # контекст главного потока выполнения может измениться к моменту
    # следующего вызова функции.
    current_data = yield self.fetchOneData(i)
    # ПЛОХО! изменяется общее состояние между yield!
    self.data.append(current_data)

# Правильная реализация, не использующая блокировок
def safeFetchAllData(self, n):
    # перемещаем необходимые данные в локальную переменную
    data = []
    for i in range(n):
        current_data = yield self.fetchOneData(i)
        data.append(current_data)
    # даже если fetchAllData будет вызвана несколько раз
    # в параллельных потоках, self.data всегда будет иметь
    # согласованное состояние.
    self.data = data

```

Функции из пула потоков не должны изменять общее состояние

Иногда бывает нужно выполнить некоторые сложные вычисления или использовать библиотеку, выполняющую блокирующий ввод/вывод. Обычно эти операции производятся внутри вспомогательного потока выполнения, отличного от потока реактора, чтобы избежать блокировки реактора на время обработки.

Поэтому, используя потоки выполнения, важно позаботиться о защите общего состояния от одновременного доступа. Чтобы избежать использования любых блокировок в потоках, мы следуем в Buildbot простому правилу. Все функции или методы, выполняющиеся вне потока реактора, не должны иметь побочных эффектов, влияющих на состояние приложения, — они взаимодействуют с остальной частью приложения только через параметры и возвращаемые значения.

```

from twisted.internet import defer
from twisted.internet import threads

class MyClass(object):
    [...]
    def unsafeFetchAllData(self, n):
        def thdfetchAllData():
            # ПЛОХО! изменяет общее состояние из стороннего потока!
            self.data = []
            for i in range(n):
                with open("hugefile-{}.dat".format(i)) as f:
                    for line in f:
                        self.data.append(line)
            return threads.deferToThread(thdfetchAllData)
        @defer.inlineCallbacks

```

```
def safeFetchAllData(self, n):
    def thdfetchAllData():
        data = []
        for i in range(n):
            with open("hugefile-{}.dat".format(i)) as f:
                for line in f:
                    data.append(line)
        # мы не изменяем общего состояния, а передаем результаты
        # в главный поток через возвращаемое значение
        return data
    data = yield threads.deferToThread(thdfetchAllData)
    self.data = data
```

Этот пример иллюстрирует загрузку данных из больших файлов, но тому же шаблону может следовать любая синхронная операция или операция, для которой нет доступной асинхронной библиотеки.

Блокировки Deferred

Как показывает наш опыт, следование двум предыдущим рекомендациям защищает от 99 % проблем с конкурентным выполнением. Для предотвращения оставшегося 1 % в Twisted имеются замечательные примитивы конкурентного выполнения. Но, прежде чем использовать их, подумайте дважды, так как это часто свидетельствует о недостаточной продуманности дизайна.

- `DeferredSemaphore` реализует семафор на случай, когда возможно не более N конкурирующих попыток доступа к общему ресурсу.
- `DeferredLock` реализует простую блокировку. Эквивалентен `DeferredSemaphore` с $N=1$, но имеет более простую реализацию.
- `DeferredQueue` реализует очередь, которую можно читать посредством `Deferred`.

Исходный код этих классов довольно поучителен и стоит того, чтобы взглянуть в него. В отличие от аналогов для потоков выполнения, они имеют очень простые реализации благодаря применению принципов асинхронного программирования. В случаях, когда имеющейся функциональности недостаточно, их можно легко расширить и добавить необходимые возможности. Например, `DeferredQueue` не позволяет определить длину очереди, что является существенным недостатком для мониторинга служб в промышленном окружении.

Тестирование

В настоящее время автоматизированное тестирование является насущной необходимостью для любых серьезных дел, но 15 лет назад, особенно в мире открытого исходного кода, дело обстояло иначе. Такие инструменты, как Buildbot, Jenkins и Travis-CI, значительно улучшили ситуацию, и теперь редко можно найти библиотеку или приложение с открытым исходным кодом, которые не имели бы пусть даже самых простых тестов.

Набор тестов для Buildbot имеет непростую историю. Ранние версии имели набор интеграционных тестов, но они были фрагментарными, трудными для понимания, и охватывали не очень большую часть кода. В какой-то момент они оказались сложнее, чем стоило бы, и мы решили полностью их удалить и создать набор модульных тестов. С тех пор мы написали модульные тесты для какой-то части существующего кода, но, что особенно важно, потребовали, чтобы любой новый или переработанный код сопровождался новыми тестами. После нескольких лет напряженной работы тестированием охвачено около 90 % кода Buildbot, при этом большая часть нетестируемого кода сохраняется только для обратной совместимости. Большой охват очень важен для такого фреймворка, как Buildbot.

Фреймворк тестирования Twisted Trial незаменим для тестирования асинхронной базы кода. Благодаря многолетнему опыту асинхронного тестирования список возможностей Trial теперь считается стандартом для фреймворков тестирования асинхронного кода.

Тесты являются асинхронными по умолчанию, то есть они могут возвращать объекты Deferred. Фреймворк тестирования обеспечивает их ожидание и выполняет каждый тест в новом экземпляре реактора. В Trial также имеется класс SynchronousTestCase, который пропускает настройку реактора и работает еще быстрее.

Появление необработанных объектов Deferred – распространенная ошибка. Trial вводит принцип «грязного реактора», пытаясь выявить такие ситуации.

Например, взгляните на следующий код:

```
@defer.inlineCallbacks
def writeRecord(self, record):
    db = yield self.getDbConnection()
    db.append(self.table, record) # ПЛОХО: отсутствует yield
```

и соответствующий тест:

```
@defer.inlineCallbacks
def test_writeRecord(self):
    record = ('foo', 'bar')
    yield self.filer.writeRecord(record)
```

Когда Deferred в этом тесте разрешится, Trial изучит список реактора с ожидающими операциями ввода/вывода и таймерами. Если операция append еще не завершилась, ожидающая операция чтения из сокета или записи в сокет вызовет исключение DirtyReactor. Любой объект Deferred, который будет утилизирован сборщиком мусора в необработанном состоянии ошибки, также вызовет сбой теста. К сожалению, если операция, как append в примере выше, успеет благополучно завершиться до завершения теста, Trial не обнаружит ошибку. По этой причине редкие и непостоянные ошибки могут вызывать некоторое разочарование у пользователей и разработчиков.

Механизм поддержки сопрограмм в Python 3.5 привнес новые возможности, улучшающие отслеживание таких ошибок (RuntimeError: coroutine [...] was never awaited), но их можно использовать только с сопрограммами.

ИМИТАЦИИ

Модульное тестирование требует хорошей изоляции тестируемых блоков кода. Большинство компонентов Buildbot зависят от других компонентов, таких как базы данных, очереди сообщений и функции доступа к данным. В Buildbot принято соглашение о включении ссылки на экземпляр BuildMaster в виде `self.master` в каждый объект службы. Благодаря этому другие объекты становятся доступными через свойства `self.master`, например `self.master.data.buildrequests`. Для нужд тестирования класс `buildbot.test.fake.fakemaster.FakeMaster` определяет фиктивный объект мастера, который может предоставить доступ к аналогичному массиву фиктивных компонентов (имитаций).

Многие из этих фиктивных компонентов являются простыми классами, предназначенными исключительно для тестирования. Однако такие имитации не точно воспроизводят поведение реальных компонентов, что может оказаться проблемой. Для небольших компонентов этот риск, как правило, невелик, и при должной осторожности можно предполагать, что они верны.

Однако базы данных имеют сложные API с десятками методов и сложными правилами взаимодействий. Поэтому для более надежного тестирования желательно всегда выполнять проверки с привлечением настоящей базы данных. Buildbot поддерживает SQLite, встроенную в Python, поэтому разработчикам будет несложно организовать такое тестирование. Тем не менее настройка и запуск базы данных для каждого теста, пусть и хранящейся в памяти, производятся довольно медленно. Поэтому в Buildbot используется полная реализация API базы данных, использующая только простые структуры данных Python. Чтобы гарантировать соответствие этого API интерфейсу реальной базы данных, он тоже должен пройти те же модульные тесты, что и настоящая реализация. Результатом является имитация, которая гарантированно дает надежные результаты в модульных тестах компонентов, зависящих от нее, – «проверенная имитация». Эта имитация работает быстрее, чем реальная база данных, и обеспечивает вполне надежные результаты тестирования.

Итоги

Buildbot – большой, зрелый фреймворк, использовавший Twisted с самого начала. Его история демонстрирует развитие – иногда с неправильными поворотами – асинхронного Python в течение последнего десятилетия. И его последние версии предлагают целый ряд практических возможностей для программистов, использующих Twisted.

Глава 11

Twisted и HTTP/2

ВВЕДЕНИЕ

HTTP/2 – это последняя редакция почтенного протокола, лежащего в основе почти всей Всемирной паутины: протокол передачи гипертекста (HyperText Transfer Protocol, HTTP), первоначально разработанный Тимом Бернерсом-Ли (Tim Berners-Lee) в CERN (Европейская организация по ядерным исследованиям) в 1989 году. С тех пор HTTP является двигателем веба. Протокол занимает настолько доминирующее положение, что почти все, что большинство людей считают «интернетом», фактически является частью Всемирной паутины и использует HTTP.

По своей сути HTTP – это протокол, позволяющий вашему браузеру взаимодействовать с веб-сайтами. Он определяет, как браузер должен представлять запрашиваемые «ресурсы», такие как веб-страницы или изображения, и как серверы должны возвращать эти ресурсы в ответ. Он также поддерживает выгрузку данных. Однако HTTP используется не только для взаимодействий с веб-сайтами, но и для обмена информацией между машинами посредством «веб API», что дает программистам возможность создавать приложения, взаимодействующие с данными, хранящимися на других компьютерах. Веб API используют большинство крупных компаний, о которых вы слышали!

На первом этапе протокол пережил множество ревизий и в 1996 году был утвержден в документе RFC 1945, опубликованном инженерной рабочей группой по интернету (Internet Engineering Task Force, IETF). Этот документ представлял видение первой долгосрочной версии протокола и определил его основные свойства. К ним относятся текстовый формат, читаемый человеком; зависимость от словаря глаголов с четко определенным поведением, таких как GET, POST и DELETE; и инструменты управления кешированием. За редакцией HTTP/1.0 быстро последовала редакция HTTP/1.1, определившая ряд улучшений выразительности и эффективности протокола. Редакция HTTP/1.1 была утверждена в RFC 2068 в 1997 году и обновлена в знаменитом RFC 2616 в 1999 году. Эта редакция HTTP просуществовала практически в неизменном

виде почти 15 лет¹. Все программы и службы, достигшие зрелости за это время, были построены на основе данного протокола эпохи 1990-х.

К сожалению, редакция HTTP/1.1 имела ряд недостатков, делавших протокол все менее пригодным для интернета 2010-х. Вследствие своей текстовой природы он был слишком многословным, требовал передачи намного большего количества байтов, чем необходимо. В нем также отсутствует какая-либо форма мультиплексирования², то есть для каждой пары запрос/ответ требуется отдельное TCP-соединение, что вызывает проблемы, о которых рассказывает чуть ниже. Кроме того, анализ протокола сложен и выполняется медленнее, чем анализ большинства двоичных протоколов.

Сочетание этих недостатков приводило к проблемам в соединениях HTTP/1.1, выражающихся в виде задержек, уменьшения пропускной способности и напрасного расходования ресурсов операционной системы. Пытаясь преодолеть эти проблемы, в Google начали экспериментировать с альтернативными реализациями HTTP, которые сохраняли ту же семантику, но использовали другой формат передачи данных. После нескольких лет тестирования этого экспериментального протокола, названного SPDY³, стало ясно, что он способен решить многие проблемы, присущие HTTP/1.1, и рабочая группа HTTP в IETF решила использовать SPDY как основу для новой редакции протокола HTTP: версии 2.

HTTP/2 содержит множество улучшений. Он потерял свой текстовый характер и превратился в поток двоичных кадров с префиксами, определяющими длину этих кадров. Добавилась специальная форма сжатия, подходящая для использования с заголовками HTTP и значительно сокращающая издержки, связанные с передачей данных. Он поддерживает мультиплексирование и управление потоком, позволяя поддерживать несколько диалогов запрос/ответ через одно TCP-соединение. И наконец, добавилась явная поддержка со-

¹ Редакция HTTP/1.1 была дополнена в RFC 7230 и связанных с ним документах RFC в 2014 году. Они не вносили существенных изменений в протокол – их целью была систематизация способов развертывания HTTP/1.1, разработанных за прошедшие 15 лет.

² Редакция HTTP/1.1 определяет идею «конвейерной обработки», которая позволяет агенту отправлять несколько запросов, не дожидаясь ответа на предыдущие. В принципе, такая конвейерная обработка обеспечивает некоторую поддержку мультиплексирования. Но, к сожалению, она плохо справляется со своей задачей и страдает от ряда проблем, наиболее серьезной из которых является требование к серверам отвечать на запросы в порядке их доставки. Если серверу нужно сгенерировать объемный ответ, это может привести к долгому ожиданию ответа на последующий запрос. Кроме того, если сервер получает запрос, имеющий побочный эффект (например, изменяет некоторые данные), он должен прекратить обработку всех других запросов в этом конвейерном соединении, пока данный запрос не будет полностью обработан, если нет уверенности, что другие запросы являются безопасными. На практике эти ограничения настолько обременительны, что во всех основных браузерах поддержка конвейерной передачи отключена, и поэтому данная технология не получила широкого распространения.

³ Произносится как «спиди».

гласования расширений, что существенно упрощает возможность расширения HTTP/2 в будущем.

С момента стандартизации в 2015 году протокол HTTP/2 достиг значительного успеха. Его поддержка была реализована во всех основных браузерах и в большинстве веб-серверов, и теперь он быстро вытесняет HTTP/1.1 из веба. Учитывая такое успешное распространение, многие разработчики захотят воспользоваться преимуществами нового протокола в своих приложениях, в том числе в приложениях, основанных на Twisted.

Фреймворк Twisted содержит свою реализацию HTTP-сервера. В 2016 году началась работа по расширению этого HTTP-сервера и добавлению в него поддержки HTTP/2. Впервые эта поддержка стала доступна в версии Twisted 16.3, вышедшей в июле 2016 года. В оставшейся части этой главы мы обсудим некоторые особенности данной реализации и рассмотрим несколько методов, которые пригодятся в асинхронном программировании.

Цели и задачи

Работа по интеграции поддержки HTTP/2 в Twisted с самого начала преследовала ряд конкретных целей.

Бесшовная интеграция

Первая и самая важная цель разработки проекта по поддержке HTTP/2 состояла в том, чтобы максимально интегрировать ее в существующий веб-сервер Twisted, который является частью `twisted.web`. Разработчики стремились реализовать поддержку HTTP/2 так, чтобы существующие веб-приложения на Twisted могли использовать ее без изменения исходного кода. Это позволило бы обеспечить максимально широкий доступ к HTTP/2 для существующих и новых веб-приложений.

К счастью, HTTP/2 поддерживает ту же «семантику», что и HTTP/1.1. Это означает, что для любого действительного сообщения HTTP/1.1 имеется хотя бы одно эквивалентное представление в HTTP/2. Несмотря на то что конкретная последовательность байтов, отправляемых в сеть, отличается, абстрактный смысл сеанса HTTP можно точно передать как в HTTP/1.1, так и в HTTP/2. Это означает, что можно, по крайней мере в принципе, разрешить пользователям `twisted.web` прозрачно включать поддержку HTTP/2 без изменения кода.

Такая «бесшовная» интеграция в Twisted стала возможной благодаря широкому использованию интерфейсов для определения слоя абстракции. Интерфейс – это формальное описание функций, которые можно вызывать для семейства родственных объектов. Например, вот как можно описать интерфейс «транспортного средства», используя `zope.interface`:

```
from zope.interface import interface

class IVehicle(interface):
    def turn_on():
```

```
pass  
  
def turn_off():  
    pass
```

Определив интерфейс, можно писать программы, способные управлять любым типом транспортного средства, используя этот интерфейс, а не конкретную реализацию. Подобные интерфейсы являются формой *полиморфизма* (термин, используемый в объектно-ориентированном программировании), который выступает альтернативой наследованию классов. В этом разделе мы больше не будем касаться идеи использования интерфейсов для полиморфизма, но отметим еще раз, что определение интерфейсов позволяет писать код, способный очень изящно применять альтернативные реализации одного и того же интерфейса.

В случае с протоколом HTTP мы могли бы определить набор интерфейсов, определяющих HTTP-операции на семантическом уровне (без ссылки на конкретный формат), и дать пользователям возможность писать код, использующий эти интерфейсы. Например, мы могли бы определить интерфейс `HTTPServer`, действующий в терминах обобщенных объектов `HTTPRequest` и `HTTPResponse` и защищающий код пользователя от конкретных свойств базового соединения.

К сожалению, определить интерфейсы таким способом очень непросто, и на практике возник ряд трудностей, которые необходимо было решить, чтобы сделать эту цель достижимой. Мы рассмотрим их далее в этой главе. Однако, преодолев эти трудности, мы смогли создать окончательную реализацию, которую *почти* бесшовно можно было объединить с существующей реализацией HTTP/1.1.

В результате этих усилий, начиная с Twisted 16.3, любое приложение, использующее `twisted.web`, могло получить автоматическую поддержку HTTP/2, установив дополнительный модуль `http2` при установке или обновлении Twisted. После этого Twisted будет обнаруживать все необходимые функции и, при возможности, автоматически использовать HTTP/2.

Оптимизация поведения по умолчанию

HTTP/2 – сложный протокол со множеством параметров настройки, которые могут влиять на его эффективность. Размеры кадров, управление приоритетами, стратегии сжатия, ограничение числа одновременных потоков и даже размеры буферов – все эти настройки влияют на эффективность протокола.

Поскольку поддержку HTTP/2 в Twisted планировалось сделать максимально прозрачной для пользователя, весьма вероятно, что большинство пользователей не заметят, что она существует. Поэтому было очень важно, чтобы по умолчанию протокол действовал настолько эффективно, насколько это возможно, – если пользователи не знают о наличии какой-то особенности, они не смогут правильно настроить ее для своего варианта использования.

Это прямо вытекает из предыдущей цели: особенности, которые должны быть полностью прозрачными для пользователя, также должны иметь разум-

ные настройки по умолчанию, подходящие для самого широкого круга вариантов использования. Если этого не сделать, пользователи получат неоптимальное поведение своего программного обеспечения, даже не подозревая об этом, а когда в конечном итоге они узнают об этом, им придется заниматься сложным профилированием и отладкой, чтобы исправить проблему.

По этой причине поддержка HTTP/2 в Twisted должна пройти по тонкой линии. Конфигурация по умолчанию должна давать хорошие результаты практически при любых обстоятельствах и, как минимум, не хуже, чем поддержка HTTP/1.1. В противном случае пользователи, активировавшие эту поддержку, проиграют, что делает ее совершенно бесполезной.

Разделение задач и повторное использование кода

Последняя и самая важная цель состояла в том, чтобы изобретать не слишком много велосипедов. Большой ошибкой при проектировании сетевых приложений является создание пользовательских компонентов, а не объединение уже существующих реализаций. Это особенно явно проявляется при работе с такими фреймворками, как Twisted, которые требуют осторожности при интеграции существующих решений, чтобы избежать блокировки цикла событий. Причина этого в том, что в разных окружениях используются разные конкретные механизмы предотвращения блокировки цикла событий, поэтому очень заманчиво написать отдельный код для каждого окружения. Однако цена такого решения слишком высока: невозможность повторного использования больших кусков кода в нескольких фреймворках.

К счастью, экосистема Python уже содержала «sans io»-реализацию¹ HTTP/2, которую можно использовать для парсинга и сериализации протокола HTTP/2, но она не выполняет никаких операций ввода/вывода. Подобные реализации предназначены для внедрения в такие фреймворки, как Twisted, и обеспечивая возможность повторного использования кода.

Это один из важнейших шаблонов проектирования в сетевом программировании, поэтому повторим еще раз: везде, где возможно, старайтесь разделить свой парсер протокола от конкретной реализации ввода/вывода. Парсер протоколов должен работать только с буферами байтов в памяти, потребляя или производя их, и не должен иметь механизма получения байтов из сети или их передачи в сеть. Данный шаблон проектирования упрощает перенос парсера протоколов из одного шаблона ввода/вывода в другой, а также его тестирование и расширение.

Эта цель меняет характер работы. Реализация HTTP/2 в Twisted берет на себя ту часть работы, которая связана с записью байтов в сеть и чтением их из сети,

¹ «sans-io»-реализация – это реализация протокола, не осуществляющая фактических операций ввода/вывода, т. е. «без ввода/вывода» («sans-io»). Иными словами, «sans-io»-реализация протокола полностью определена в терминах синхронных функций, синхронно возвращающих результаты, неблокирующихся и не использующих какой-либо формы ввода/вывода. – *Прим. перев.*

установкой и обработкой таймеров, а также преобразованием событий HTTP/2 в интерфейс `twisted.web`. Реализация HTTP/2 без ввода/вывода отвечает за преобразование потоков байтов в события HTTP/2 и результатов вызовов функций из `twisted.web` в потоки байтов для вывода.

Такое повторное использование кода также освобождает больше времени на оптимизацию частей реализации, где Twisted может внести наибольший вклад. Реализация в Twisted в основном направлена на уменьшение задержек передачи данных в сеть, эффективное распространение противодействия, предотвращение ненужных системных вызовов и уменьшение накладных расходов на ввод/вывод. Это гораздо проще сделать, если основная логика протокола выделена в отдельный проект.

Проще говоря, это лучший подход при работе над «стандартными» задачами. Он способствует уменьшению размера базы кода, позволяет тратить меньше времени на решение задач, которые уже были решены, и сосредоточиться на повышении эффективности и масштабируемости решения.

ПРОБЛЕМЫ РЕАЛИЗАЦИИ

Определив цели проектирования, можно приступить к работе над кодом. Для многих разработчиков это самая интересная часть, но в процессе программирования часто возникает множество непредвиденных сюрпризов. Кроме того, некоторые аспекты дизайна, показавшиеся достаточно простыми при обсуждении идеи, могут оказаться очень сложными в реализации. В этом разделе рассматривается ряд конкретных проблем, возникших в конкретной реализации.

Что такое соединение? Ценность стандартных интерфейсов

В `twisted.web` есть ряд объектов, совместно используемых в реализации поддержки HTTP. Простейшим примером может служить связь транспорта `TCPTransport` с объектами `HTTPChannel` и `Request`, как показано на рис. 11.1.

Работая над реализацией поддержки HTTP/2, мы обнаружили, что стандартный обработчик HTTP-запросов в Twisted (`twisted.web.http.Request`) ожидает ссылку на объект HTTP-соединения в форме `twisted.web.http.HTTPChannel` (или с похожим интерфейсом: как ни странно, ожидаемый интерфейс никогда прежде не был определен). Конструктор `Request` обращался к только что полученному объекту канала и извлекал атрибут `transport`, чтобы сохранить его у себя. Все последующие вызовы `Request.write` для записи тела ответа транслировались в вызов `transport.write`. `transport.write` – функции `write` объекта `transport`. Это может быть любой объект, реализующий `twisted.internet.interfaces.ITransport` – еще один из интерфейсов в `zope.interface`, широко используемых в Twisted. В данном случае `ITransport` является особенно распространенным интерфейсом и используется для представления транспорта любого вида, поддерживающего операцию записи данных. Обычно это низкоуровневый потоковый протокол, такой как TCP, но вообще это может быть все, что обеспечи-

вает интерфейс записи в поток. В старой модели HTTP/1 это почти всегда был транспортный протокол TCP.

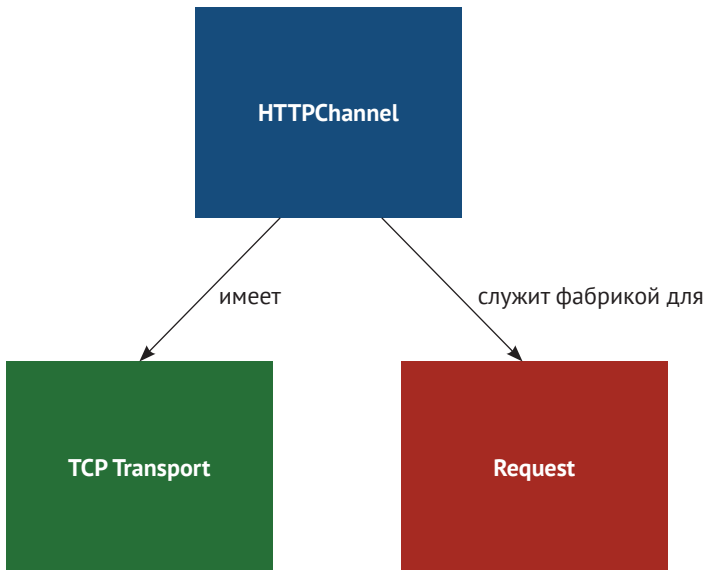


Рис. 11.1 ❖ Три наиболее важных объекта, используемых в реализации поддержки HTTP в Twisted

Это нарушение абстракции нормально работает в HTTP/1.1, потому что после отправки заголовков тело ответа можно рассматривать как произвольный поток байтов. Однако это не подходит для HTTP/2: мультиплексирование, поддержка приоритетов и управление потоком делают чрезвычайно важным предотвращение произвольной записи данных приложениями в TCP-соединение.

Работая над HTTP/2, мы должны были решить эту проблему, но мы не могли просто взять и удалить эти свойства: они являются частью общедоступного API объектов Request и должны быть сохранены¹.

Самым простым решением было бы реализовать в объекте `twisted.web.http.HTTPChannel` для протокола HTTP/1.1 интерфейс `ITransport` и в большинстве методов этого интерфейса вызывать методы базового транспорта. Такой подход обеспечил бы лучшую инкапсуляцию собственных ресурсов `HTTPChannel`, избавил бы пользователей от необходимости обращаться к нему, чтобы отправить тело ответа, а также решил бы некоторые семантические проблемы в предыдущем дизайне. По сути, `HTTPChannel` должен *быть* транспортом для ответов, а не объектом, *имеющим* транспорт, через который можно отправлять ответы.

¹ Преимущества обратной совместимости лучше объясняются в статье: <https://twisted-matrix.com/documents/current/core/development/policy/compatibility-policy.html>.

Однако из-за необходимости обеспечить обратную совместимость мы не могли удалить свойство `transport` из `HTTPChannel`, поэтому первым важным шагом было не инкапсулировать транспорт, а воспрепятствовать его использованию.

После этого можно было бы изменить внутреннюю реализацию `Request` и использовать `HTTPChannel` вместо каждого обращения к транспорту. По сути, каждое обращение к `self.transport` в теле метода `Request` было изменено на `self.channel`. Это гарантировало, что реализация обработки HTTP-запросов в `Twisted` теперь должным образом учитывает предполагаемую абстракцию между TCP- и HTTP-соединениями.

К сожалению, мы не смогли создать четкое разделение из-за политики совместимости `Twisted`. Существует слишком много приложений, использующих HTTP/1.1 и `twisted.web`, и некоторые из них неизбежно записывают ответы напрямую в транспорт (или используют его как-то иначе, например для получения сертификатов TLS). По этой причине свойство `transport` нельзя удалить из `HTTPChannel` и оно должно присутствовать в любом объекте, реализующем HTTP/2.

Как отмечалось в предыдущем разделе, мультиплексирование в HTTP/2 требует нескольких взаимодействующих объектов для обеспечения необходимых абстракций. Это также означает, что имеется два отдельных объекта, которые *вместе* предоставляют тот же интерфейс, что и `HTTPChannel`. Объекту `Request` достаточно ограниченного подмножества интерфейса `HTTPChannel`. Это подмножество было выделено в `H2Stream` для совместимости.

В связи с тем, что объекту `Request` необходим доступ к свойству `transport` канала, `H2Stream` также нуждается в этом свойстве. Однако совершенно необязательно, чтобы код реализации HTTP/2 продолжал следовать тому же нарушению абстракции, что и в коде HTTP/1.1: благодаря отсутствию требований устаревших API он просто должен дать доступ к свойству. По этой причине все объекты `H2Stream` имеют свойство `transport`, которое всегда имеет значение `None`.

Это хороший пример ситуации, которую можно было бы значительно упростить, если бы существовали стандартные интерфейсы между объектами `Request` и `HTTPChannel`. Первоначально не предполагалось, что каждому из этих объектов может потребоваться поддерживать несколько возможных реализаций своего объекта-партнера, поэтому интерфейсы, используемые между этими двумя объектами, не были определены формально. Отсутствие формального определения означает, что *фактическим* интерфейсом этих объектов является вся поверхность их API: все методы и все свойства.

Такие широкие и неявные интерфейсы приводят к огромным трудностям при попытке создания дополнительных уровней абстракции. Если человек, повторно реализующий объект, должен полностью эмулировать весь общедоступный API, ему значительно сложнее будет предложить альтернативные реализации и создать соответствующие абстракции.

С другой стороны, практически все, что необходимо объекту `Request` от `HTTPChannel`, фактически *определено* в виде интерфейса `ITransport`. Поскольку основная работа `Request` заключалась в записи данных в транспорт `HTTPChannel`

и вполне разумно было предположить, что этот транспорт реализует `ITransport`, мы легко смогли определить, какие методы необходимо добавить в `HTTPChannel` и каким должно быть их поведение. После этого не составило сложностей определить, какой эффективный API должен представлять `H2Stream`.

Из-за недостаточно внимательного отношения к возможности расширения в первые годы разработки `Twisted Web` интегрировать `HTTP/2` оказалось значительно сложнее, чем предполагалось. Однако могло бы быть и хуже: благодаря широкому использованию интерфейсов в `Twisted` исправление этих нарушений абстракции далось гораздо проще, чем могло бы быть в противном случае.

Это должно стать важным уроком для будущих инженеров: при проектировании системы особое внимание следует уделять высокоуровневым интерфейсам между компонентами. Эти интерфейсы должны явно определяться в коде, потому что они наглядно описывают, что каждый компонент ожидает от других, и дают намного более гибкие возможности расширения и улучшения компонентов в будущем.

Мультиплексирование и приоритеты

Одна из самых сложных особенностей `HTTP/2` – поддержка мультиплексирования. Эта одна из основных особенностей `HTTP/2` была добавлена в протокол с целью дать возможность посылать и получать несколько пар запрос/ответ через одно `TCP`-соединение. Этот подход дает несколько преимуществ перед протоколом `HTTP/1.1`, использующим несколько одновременных `TCP`-соединений:

- 1) расходуется меньше системных ресурсов. Для каждого `TCP`-соединения создается дескриптор файла как на стороне клиента, так и на стороне сервера, что увеличивает объем работы, которую обе операционные системы должны выполнить для поддержания сетевых соединений. Это также увеличивает расход памяти как в ядре, чтобы разместить структуры данных для соединений, так и в приложениях `Twisted`, размещающих ряд структур данных для управления каждым транспортом;
- 2) обеспечивает лучшую пропускную способность и более высокую скорость передачи данных. Распространенные `TCP`-алгоритмы управления заторами в сети были разработаны с расчетом на то, что между любыми двумя хостами одновременно будет не более одного `TCP`-соединения¹.

¹ В частности, алгоритмы предполагают, что события потери пакетов для каждого `TCP`-соединения в системе независимы: событие потери пакетов в одном соединении не имеет никакого отношения к аналогичным событиям в других. В случае нескольких `TCP`-соединений между одними и теми же двумя хостами, через которые передаются большие объемы данных, это предположение оказывается неверным: события потери пакетов чаще всего происходят из-за насыщения канала, и в результате потери почти наверняка будут случаться одновременно в большинстве или во всех соединениях. Это влечет уменьшение пропускной способности всех `TCP`-соединений вдвое, вследствие чего канал оказывается не полностью загруженным в течение длительных периодов времени.

При использовании множества соединений между двумя хостами, особенно если через них передаются большие объемы данных (что не редкость в интернете), пропускная способность параллельных соединений не достигает максимально возможной величины;

- 3) поддерживает соединение в «разогретом» состоянии. Если TCP-соединения простаивают в течение длительного периода времени, они могут закрываться (промежуточными или конечными узлами) или переводиться в состояние «медленного старта», когда прежде полученная информация о заторах в канале устаревает и уничтожается. В результате, когда такое соединение понадобится повторно, оно в течение периода медленного старта будет иметь низкую пропускную способность, а если соединение было закрыто, также потребуется время, чтобы вновь инициировать соединение. «Горячие» соединения, то есть находящиеся в постоянном или почти постоянном использовании, позволяют избежать этих проблем, что уменьшает задержку и увеличивает пропускную способность.

Мультиплексирование достигается в HTTP/2 путем разделения одного соединения HTTP/2 на несколько двунаправленных «потоков». Каждый поток несет один HTTP-запрос и соответствующий ему ответ. Это достигается простым присваиванием каждому потоку уникального идентификатора и передачей этого идентификатора в каждом кадре данных. Такой подход позволяет разделить один упорядоченный поток данных, предоставляемый TCP-соединением, на несколько логических потоков данных, как показано на рис. 11.2.

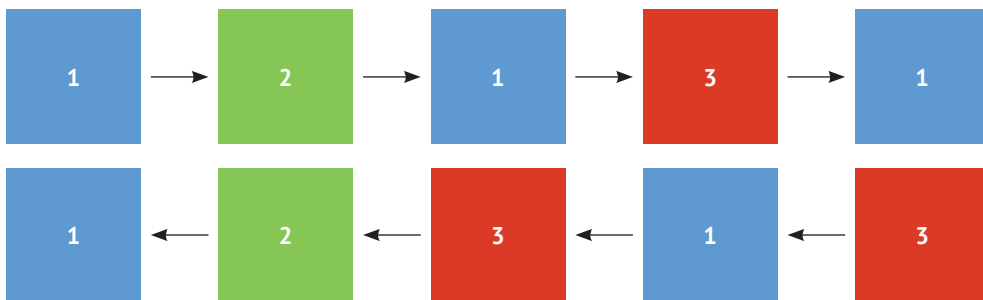


Рис. 11.2 ❖ Потоки – это чередующиеся блоки данных. Они могут следовать в любом порядке

Однако просто снабдить все данные соответствующим идентификатором потока недостаточно. Чтобы понять причину, представьте, что может произойти с гипотетическим веб-сайтом, который действует как облачная картинная галерея. Веб-сайт решает две задачи: отправляет изображения и принимает пользовательский ввод для внесения в них изменений. Каждый пользовательский ввод инициирует запрос/ответ: кроме того, когда пользователь выполняет прокрутку или редактирование, сервер передает другой файл изображения.

Запросы/ответы, как правило, очень невелики: это могут быть, например, документы JSON всего из нескольких сотен байтов. Изображения намного больше: возможно, много мегабайтов. Кроме того, для отправки изображения не требуется сложных и продолжительных вычислений: они хранятся на диске и постоянно доступны для веб-сервера.

Соответственно, проблема заключается в том, что сервер может заполнить соединение HTTP/2 данными для потоков изображений и заблокировать данные запросов/ответов. Запросы/ответы содержат относительно немного данных, но эти данные имеют более высокий *приоритет*, чем данные изображений. В большинстве своем пользователи готовы ждать окончания загрузки миниатюр, но гораздо реже склонны ждать, пока загрузятся все изображения и интерфейс наконец-то среагирует на их действия.

Эта проблема присуща большинству механизмов передачи данных с мультиплексированием. А можно ли гарантировать более быструю передачу данных с высоким приоритетом и при этом обеспечить максимальное использование соединения? Существует много решений этой проблемы, и в HTTP/2 используется схема, согласно которой клиенты устанавливают *приоритеты потоков*.

Приоритеты позволяют клиентам информировать сервер об относительной важности данных в разных потоках, чтобы сервер мог решить, как лучше распределить свои ограниченные ресурсы между различными запросами, посылаемыми клиентом. Основным ресурсом веб-сервера является пропускная способность, но более сложные серверы могут использовать эту информацию для распределения, например, процессорного времени, файловых дескрипторов или дискового пространства, то есть практически любого ограниченного ресурса.

Простейшая возможная схема приоритетов потоков – просто назначить каждому потоку числовой приоритет. Поток с большим значением приоритета важнее, чем поток с меньшим значением, и его следует обслуживать первым. Такая схема недостаточно выразительна: она позволяет указать, что некоторые данные важнее других, но не позволяет выразить, *насколько* важнее эти данные.

Простейшая возможная схема, которую можно применить на практике, – присвоить каждому потоку числовой вес. Этот вес отражает относительную важность потока: если поток X имеет вдвое больший вес, чем поток Y, значит, он примерно в два раза важнее. Преимущество этого подхода заключается в том, что его можно использовать для пропорционального распределения ресурсов: потоку X из предыдущего примера должно быть выделено в два раза больше ресурсов, чем потоку Y. Это позволяет клиентам показать, что для них важнее получить своевременный ответ на поток X, чем на поток Y, и насколько это важнее.

Данный подход использовался протоколом SPDY, предшествовавшим протоколу HTTP/2. Однако когда пришло время, рабочая группа по разработке спецификации HTTP/2 посчитала этот подход недостаточно выразительным

и непригодным для некоторых вариантов использования. В частности, он не позволяет клиенту легко выразить ограничение «использовать ресурсы для потока А, только если по какой-то причине их нельзя использовать для потока В». То есть он позволяет клиенту сказать: «поток А бесполезен» без результатов, которые несет поток В, поэтому не нужно тратить время на А, если поток В продолжает действовать».

По этой причине HTTP/2 имеет гораздо более сложную систему приоритетов. Эта система позволяет клиенту определить *дерево* приоритетов, каждый узел которого *зависит от* родительского узла над ним. Эти приоритеты не влияют на «управляющие» данные, такие как заголовки HTTP: они используются, только чтобы указать приоритет запрашиваемого ресурса.

В реализации веб-сервера в фреймворке Twisted мы столкнулись с невозможностью распределения ресурсов, не относящихся к полосе пропускания, из-за отсутствия информации о приложении пользователя. В результате мы решили распределять только пропускную способность. Чтобы добиться максимальной эффективности, мы приняли простое допущение: величина пропускной способности прямо связана с числом посылаемых кадров, поэтому мы занялись распределением кадров. Например, если у нас есть потоки А и В с весами 32 и 64 соответственно, идеальная реализация приоритетного алгоритма выделит потоку А 1/3 полосы пропускания и потоку В оставшиеся 2/3. Для точного соблюдения этого условия потребовалось бы подсчитывать данные, передаваемые в вызов `transport.write`, что повлекло бы многократное копирование данных в буферы и из буферов. Такое многократное копирование памяти действует чрезвычайно медленно, если не использовать высокопроизводительный буфер (чего не было доступно в Twisted на момент разработки и что не входило в сферу этой работы), а значит, мы должны были избежать этого.

Чтобы избежать таких дорогостоящих вычислений, мы можем оставить записанные данные как есть и распределять полосу пропускания, давая каждому потоку количество *кадров*, пропорциональное его относительному весу. Каждый раз, когда в буфере отправки появляется место для отправки дополнительных данных, реализация веб-сервера в Twisted проверит, какой из потоков, имеющих данные для отправки, следует передать следующим, с учетом веса. Затем мы отправляем из этого потока один блок данных с размером, не превышающим максимального размера кадра¹, и повторяем процедуру. Такое мультиплексирование на основе кадров широко используется в сетевых протоколах, и его легко приспособить для любого другого протокола.

Создание и управление этим деревом приоритетов осуществляет сторонняя библиотека `priority`. Эта библиотека создает и поддерживает состояние

¹ Внимательный читатель может заметить, что Twisted не устанавливает верхнюю границу объема данных, передаваемых в `write()`, и блок данных может оказаться больше максимального размера кадра HTTP/2. Если это произойдет, нам все равно придется копировать память; это неизбежно.

приоритета, отправленного клиентом, и предоставляет итерируемый объект, который последовательно сообщает реализации Twisted, какой поток следует обслужить следующим. Дерево также включает информацию из приложения Twisted о наличии у каждого потока каких-либо данных, доступных для отправки. Потоки, не имеющие данных для отправки, считаются *заблокированными*, а доли пропускной способности TCP-соединения, назначенные этим потокам, распределяются между их дочерними потоками.

Необходимость пропускать все данные через цикл вокруг дерева приоритетов привносит дополнительные сложности в конвейер отправки данных, отсутствующие в реализации HTTP/1.1. В HTTP/1.1 все операции записи ответов передают данные напрямую в базовый объект TCP-соединения, который отвечает за буферизацию и отправку данных. В HTTP/2 дело обстоит иначе – мы должны чередовать отправку записанных данных в соответствии с относительными приоритетами потоков.

Что еще более важно, реализация должна реагировать на изменения приоритетов потоков, передаваемых клиентом: если клиент увеличивает приоритет потока, это должно отразиться в данных как можно скорее. Если реализация будет стремиться записать все данные в объект TCP-соединения как можно быстрее, в результате может получиться большой буфер, ожидающий отправки, места в котором распределены в соответствии со старыми приоритетами. В ситуациях, когда пропускная способность TCP-соединения намного ниже скорости, с какой приложение Twisted генерирует данные, это может привести к задержкам до нескольких секунд, прежде чем изменение приоритета отразится в реальных данных, что неприемлемо.

По этой причине реализация HTTP/2 в Twisted должна поддерживать свою, внутреннюю буферизацию данных и асинхронно отправлять данные, вызывая `transport.write`. Это было реализовано путем многократного использования `IReactor.callLater` для планирования вызова функции, которая будет отправлять доступный фрагмент данных с наивысшим приоритетом.

Использование `callLater` позволяет избежать переполнения буфера отправки при наличии противодействия из TCP-соединения (подробнее об этом рассказывается в следующем разделе) и гарантировать, что все доступные данные будут отправлены без блокировки вызовов `write`.

Вот как выглядит ядро функции отправки данных (с обработкой ошибок и некоторых крайних случаев):

```
class H2Connection:
    def _sendPrioritisedData(self, *args):
        stream = None
        while stream is None:
            try:
                stream = next(self.priority)
            except priority.DeadlockError:
                # Все потоки заблокированы или не имеют данных.
                # Ждать, когда появятся новые данные.
```

```

        self._sendingDeferred = Deferred()
        self._sendingDeferred.addCallback(self._sendPrioritisedData)
        return

# Ожидание транспорта. Это реализовано в другом месте в классе
# как часть реализации IPushProducer.
if self._consumerBlocked is not None:
    self._consumerBlocked.addCallback(self._sendPrioritisedData)
    return

remainingWindow = self.conn.local_flow_control_window(stream)
frameData = self._outboundStreamQueues[stream].popleft()
maxFrameSize = min(self.conn.max_outbound_frame_size, remainingWindow)

if frameData is _END_STREAM_SENTINEL:
    # Здесь нет обработки ошибок. Теоретически может
    # возникнуть исключение ProtocolError, но мы с такой
    # ситуацией ни разу не столкнулись. Если в будущем подобное
    # случится, это окажется неприятным сюрпризом.
    self.conn.end_stream(stream)
    self.transport.write(self.conn.data_to_send())

    # Освободить поток
    self._requestDone(stream)
else:
    # Учесть максимальный размер кадра.
    if len(frameData) > maxFrameSize:
        excessData = frameData[maxFrameSize:]
        frameData = frameData[:maxFrameSize]
        self._outboundStreamQueues[stream].appendleft(excessData)

    # Если по какой-то причине максимальный размер кадра окажется
    # равным нулю и мы ничего не сможем послать в таком кадре, то
    # не посылать ничего.
    if frameData:
        self.conn.send_data(stream, frameData)
        self.transport.write(self.conn.data_to_send())

    # Если данных не осталось, заблокировать поток.
    if not self._outboundStreamQueues[stream]:
        self.priority.block(stream)

    # Кроме того, если окно потока исчерпано, остановить его.
    if self.remainingOutboundWindow(stream) <= 0:
        self.streams[stream].flowControlBlocked()

self._reactor.callLater(0, self._sendPrioritisedData)

```

Эту функцию можно разбить на четыре логические части. Первая проверяет наличие потоков, которые считаются «способными к прогрессу» (то есть имеют данные для отправки и место в своем окне¹ управления). Если таких пото-

¹ Подробнее об окне управления рассказывается в следующем разделе, описывающем механизм противодействия.

ков нет, значит, нам нечего отправлять, поэтому создается экземпляр `Deferred`, который разрешится, когда какой-либо поток разблокируется.

Вторая часть проверяет наличие места в буфере отправки. Это еще один сигнал от `Deferred`: если в `self._consumerBlocked` имеется `Deferred`, это значит, что `Twisted` сообщил нам о заполнении буфера отправки и мы должны воздержаться от записи в него. И снова мы выполняем возврат, ничего не сделав, и гарантируем, что когда `Deferred` разрешится, эта функция будет вызвана. В обоих случаях функция не будет вызвана, пока не исчезнет условие, вызвавшее ее блокировку.

Третья и четвертая части организуют отправку фактических данных. В этом случае у нас есть поток с данными для отправки и место в буфере. Мы извлекаем часть данных (ранее записанных вызовом `write`) из очереди. Если это объект `_END_STREAM_SENTINEL`, значит, тело ответа отправлено полностью, и мы должны завершить отправку потока. В противном случае мы создаем кадр данных для отправки и, если необходимо, выполняем дополнительные манипуляции с состоянием.

В качестве последнего шага, если отправка данных состоялась, мы планируем вызов этого метода в будущем с помощью `callLater`, как отмечалось ранее.

Эта реализация значительно сложнее логики отправки данных в `HTTP/1.1`, но она является основой мультиплексирования в `HTTP/2`. Дополнительная вычислительная сложность делает `HTTP/2` медленнее в коде на `Python`, чем `HTTP/1.1`, но значительно увеличивает пропускную способность протокола в сети.

Подход представляет модель отправки данных с мультиплексированием, равно как и любой другой логики буферизованной отправки: единственная функция, которую можно многократно вызывать на каждом этапе и которую легко перепланировать, если по какой-то причине она не сможет выполнить свою работу (например, потому что транспорт не может принять больше данных или потому что нет данных для отправки).

Противодавление

Частая ошибка, которую допускают начинающие программисты при работе с асинхронными системами, такими как `Twisted`, заключается в том, что они не учитывают, как будут обрабатывать условия перегрузки. Фреймворки, выполняющие асинхронные сетевые операции, такие как `Twisted`, значительно увеличивают объем сетевого трафика, который могло бы обрабатывать приложение, но код приложения, написанный с использованием фреймворка, может не справиться с объемом данных, который `Twisted` и операционная система могут передать.

Все сетевые приложения рискуют столкнуться с ситуацией, когда данные поступают быстрее, чем они способны обработать. Простым примером может служить веб-приложение, обрабатывающее один запрос за 10 мс. Если это приложение постоянно получает менее 100 запросов в секунду, оно прекрасно справляется с такой нагрузкой.

Что случится, если приложению будет поступать больше 100 запросов в секунду? Есть множество способов справиться с такой нагрузкой, но большинство приложений на основе Twisted буферизуют данные¹.

Этот подход оправдан, когда график нагрузки имеет «пилообразную» форму: если нагрузка ненадолго превысила порог в 100 запросов в секунду, а затем упала ниже этого уровня, тогда запросы просто будут какое-то время обрабатываться с более продолжительными задержками, из-за чего некоторое время будут проводить в буфере. И если в какой-то момент приложение Twisted будет обслуживать данные из буфера быстрее, чем поступают новые данные, буфер будет постепенно опустошаться.

Однако если нагрузка будет выше 100 запросов в секунду продолжительное время или *существенно* превысит этот уровень (например, в сотню или тысячу раз), тогда буферизация станет проблематичной. Задержка обработки каждого запроса возрастет до уровня, неотличимого от сбоя (большинство пользователей предпочитают ждать ответа не дольше пары секунд, поэтому для таких пользователей задержка в 20 секунд эквивалентна сбою). Хуже того, если высокая нагрузка сохранится, буфер будет продолжать расти, и если не предпринять никаких мер, со временем он займет всю доступную память. В лучшем случае система остановит такой процесс, в худшем – процесс начнет интенсивно использовать файл подкачки (на жаргоне это называется «выпадет в своп»), что значительно замедлит его работу и уменьшит скорость обработки, еще больше уменьшив шансы приложения справиться с перегрузкой.

Чтобы избежать этих проблем, масштабируемые приложения Twisted должны быть готовы к перегрузке. Наиболее распространенный способ справиться с ней – создать систему, которая будет управлять *противодавлением*. Противодавление – это сигнал, посылаемый одной системой другой, который говорит: «вы посылаете данные слишком быстро, я не успеваю их обрабатывать, пожалуйста, посылайте помедленнее». Правильное распространение противодавления через асинхронное приложение позволяет этому приложению передавать такие объемы данных, с которыми сможет справиться та часть системы, которая их обрабатывает.

Хорошим примером противодавления, как ни странно, является блокировка ввода/вывода. При отправке данных по протоколу TCP с использованием блокирующихся операций ввода/вывода, если удаленный узел не прочитает данные достаточно быстро, процедура отправки заблокируется, пока удаленный узел не будет готов получить следующую порцию данных. При таком подходе отправляющее приложение принудительно замедляется и отправляет данные не быстрее, чем удаленное приложение сможет прочитать их из сокета.

¹ Многое зависит от того, на что потрачены 10 мс. Если большая часть из этих 10 мс тратится на ожидание других событий (например, запросов к базе данных), Twisted будет выполнять буферизацию. Если эти 10 мс тратятся исключительно на вычисления, тогда фреймворк будет вести себя иначе. Пока мы предполагаем, что имеет место первая ситуация.

Противодавление в Twisted

В настоящее время противодавление распространяется в Twisted благодаря тому, что транспорты и протоколы реализуют два интерфейса: `IPushProducer` и `IConsumer`. Класс `Transport` реализует интерфейс `IPushProducer`, а класс `Protocol` реализует интерфейс `IConsumer`, но в более сложных системах (таких как реализация HTTP/2 в Twisted) один и тот же объект может реализовать оба интерфейса, как `IConsumer` (для входящих данных), так и `IPushProducer` (для исходящих данных).

Это довольно простые интерфейсы:

```
class IPushProducer(IProducer):
    """
    Активный производитель, также известный как потоковый
    производитель. Как ожидается, производитель будет производить
    (посылать) данные потребителю на постоянной основе, если не был
    приостановлен. Приостановленный производитель возобновляет отправку
    после вызова метода resumeProduction(). Для производителя, который
    не может быть приостановлен, эти функции могут содержать пустые операции.
    """

    def pauseProducing():
        """
        Приостанавливает отправку данных.

        Сообщает производителю, что он генерирует слишком много данных
        и должен приостановиться до вызова resumeProducing().
        """

    def resumeProducing():
        """
        Возобновляет отправку данных.

        Сообщает производителю, что тот должен снова добавить себя в главный
        цикл и начать посылать данные потребителю.
        """

class IProducer(Interface):
    """
    Пассивный производитель данных.

    Обычно генерирование данных завершается с вызовом метода write класса,
    реализующего L{IConsumer}.
    """

    def stopProducing():
        """
        Останавливает отправку данных.

        Сообщает производителю, что его потребитель завершил работу и он
        должен прекратить отправку данных.
        """

class IConsumer(Interface):
    """
```

Потребитель, принимающий данные от производителя.

"""

```
def registerProducer(producer, streaming):
```

"""

Регистрирует потребителя для получения данных от производителя.

Назначает себя потребителем указанного производителя. Когда в этом объекте заканчиваются данные (например, когда вызов функции `send(2)` для сокета успешно перемещает последние данные из буфера в пространстве пользователя в буфер в пространстве ядра), он запрашивает у производителя возобновить передачу вызовом `resumeProducing()`.

Если роль производителя играет `L{IPushProducer}`, этот объект должен вызывать `C{pauseProducing}` всякий раз, когда буфер записи заполнится, и `C{resumeProducing}` -- когда освободится.

@type producer: `L{IProducer}` производитель

@type streaming: `C{bool}`

@param streaming: `C{True}` если `C{producer}` является `L{IPushProducer}`,
`C{False}` если `C{producer}` является `L{IPullProducer}`.

@raise RuntimeError: Если производитель уже зарегистрирован.

@return: `L{None}`

"""

```
def unregisterProducer():
```

"""

Останавливает прием данных от производителя, не разрывая соединения.

"""

```
def write(data):
```

"""

Производитель будет посылать данные вызовом этого метода.

Реализация не должна блокироваться и выполнять буферизацию, если необходимо. Если производитель прислал достаточно данных и является `L{IPushProducer}`, потребитель может вызвать его метод `C{pauseProducing}`.

"""

Наиболее важными частями этих интерфейсов являются методы `IPushProducer.pauseProducing`, `IPushProducer.resumeProducing` и `IConsumer.write`. Все остальные решают задачи администрирования, такие как информирование потребителя о производителе и уведомление производителя, что потребитель больше не может принимать данные.

Когда `IConsumer` испытывает слишком большую нагрузку и хочет, чтобы данные перестали поступать, он может вызвать метод `pauseProducing` своего производителя. Когда он готов продолжить прием данных, он может вызвать метод `resumeProducing`. В этом случае производитель возобновит отправку данных вызовом метода `write` и будет продолжать это делать, пока `IConsumer` снова не вызовет `pauseProducing`.

Противодавление в HTTP/2

В реализации HTTP/2 имеется два метода поддержки противодавления, оба используют алгоритмы управления потоком. Первый используется также в реализации HTTP/1.1, потому что он фактически встроен в ТСП, который использует оба протокола, HTTP/1.1 и HTTP/2. ТСП поддерживает такой параметр, как размер приемного окна, с помощью которого получатель сообщает отправителю о емкости приемных буферов. Если получатель на одном конце ТСП-соединения прекратит чтение из сокета, отправитель на другом конце обнаружит, что ему запрещено отправлять дополнительные данные.

Кроме того, HTTP/2 поддерживает четыре собственных окна управления потоком: два для соединения в целом (одно для данных, отправляемых клиентом серверу, и одно для данных, отправляемых сервером клиенту) и по два для каждого потока (опять же, по одному для каждого направления). Эти окна ограничивают объем данных, который может отправить каждый узел: они управляют количеством данных, которые можно отправить в данном потоке, в то время как окно для ТСП-соединения управляет общим объемом данных, которые можно отправить через соединение.

Каждое окно также может использоваться для распространения противодавления: если в любом из параметров с размером окна передать ноль, удаленный узел прекратит отправку некоторых или всех своих данных. То есть клиент может послать сигнал противодавления серверу Twisted. Аналогично, приложение Twisted имеет возможность послать сигнал противодавления клиенту: если веб-приложение на сервере обрабатывает данные медленнее, чем клиент может их отправлять, мы можем замедлить отправку данных¹.

Эта стратегия имеет два аспекта: ее реализация в сервере Twisted должна предусматривать отправку и прием сигналов противодавления, а также манипулировать окнами управления потоком HTTP/2. Поговорим сначала об отправке и приеме сигналов противодавления.

Один из основных недостатков `IConsumer/IPushProducer` состоит в том, что эти два интерфейса имеют связь один-к-одному. То есть каждый потребитель может иметь только одного производителя, и каждый производитель может иметь только одного потребителя. Это противоречит идеологии HTTP/2, когда имеется несколько потоков данных, каждый из которых может распространять сигналы противодавления независимо от других.

Самый простой способ обойти эту проблему – определить соединение HTTP/2 в терминах двух объектов вместо одного. Первый объект владеет базовым транспортом ТСП и регистрирует себя как производителя и потребителя этого транспорта – эту задачу решает класс `twisted.web._http2.H2Connection`.

¹ Обратите внимание, что эта ситуация отличается от случая, когда узел вообще решил прервать прием данных. Если принимающий узел решит прервать получение данных в потоке HTTP/2, он может сразу остановить этот поток данных с помощью специального кадра HTTP/2 `RST_STREAM`. Этот метод не имеет прямого отношения к противодавлению, но помнить о нем стоит.

Когда клиент запускает новые потоки данных, `H2Connection` создает новый объект для обработки потоковых данных, который также играет роль производителя и потребителя в прикладном коде – эту задачу решает класс `twisted.web._http2.H2Stream`. Для взаимодействий между собой эти два объекта (`H2Connection` и `H2Stream`) используют свой интерфейс, поддерживаемый только в HTTP/2 и позволяющий соединению сообщать потоку, когда тот должен приостановить своего производителя (`H2Stream.flowControlBlocked`) и когда размер окна был изменен (`H2Stream.windowUpdated`). Внутри этих методов `H2Stream` вызывает `pauseProducing` и `resumeProducing` в своем приложении. Точно так же `H2Stream` позволяет приложению вызвать `pauseProducing`, чтобы приостановить отправку данных в поток. После вызова `H2Stream` прекратит передачу данных приложению и начнет буферизовать их.

Эта довольно запутанная связь изображена на рис. 11.3.

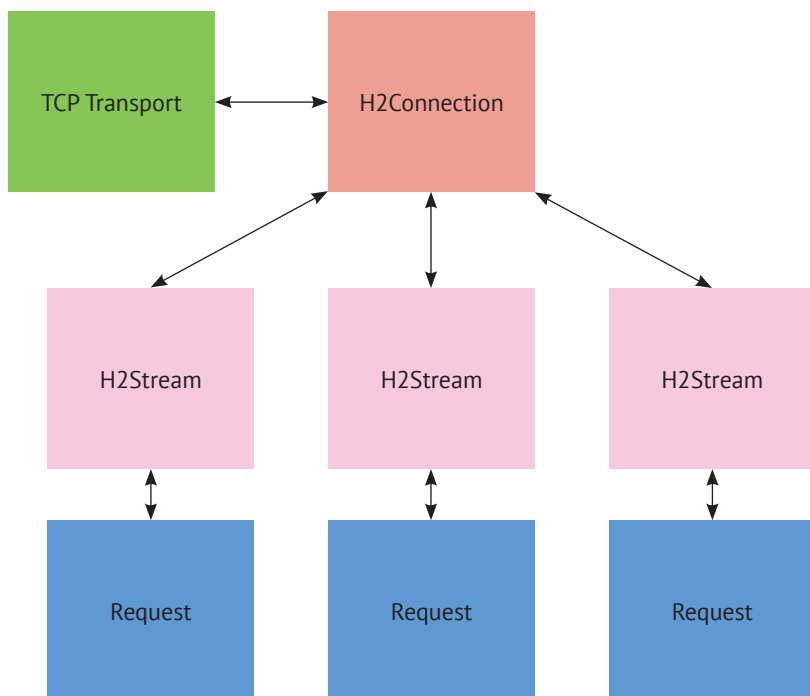


Рис. 11.3 ❖ Отношение производитель/потребитель между различными объектами в соединении HTTP/2. Каждый ряд представляет отдельный случай. Обратите внимание, что эти отношения не всегда реализуются с интерфейсами `IProducer/IConsumer`, как обсуждалось в этом разделе

Поток данных может «заблокироваться», если любое из окон управления, связанных с этим потоком, получит размер, равный нулю. То есть, если блокируется поток TCP (транспорт вызовет метод `pauseProducing` объекта `H2Connec-`

tion), все объекты `H2Stream`, принадлежащие этому соединению, вызовут `pauseProducing` в своих приложениях. Если обнулится размер окна соединения, это тоже приведет к тому, что все объекты `H2Stream` вызовут `pauseProducing` в своих приложениях. Наконец, если обнулится размер окна какого-то одного потока, метод `pauseProducing` вызовет объект `H2Stream`, связанный с этим потоком, а другие – нет.

Однако этот буфер не является неограниченным. Он ограничен окном управления потоком. `H2Connection` предлагает еще один метод для `H2Stream`: `H2Connection.openStreamWindow`. Он вызывается объектом `H2Stream` *после доставки данных в приложение*, а не до этого. Это означает, что если приложение приостановило производителя, окно потока не откроется и в какой-то момент будет исчерпано удаленным узлом, которому будет запрещено отправлять данные в этот поток, пока приложение не начнет обрабатывать очередь.

Важно отметить, что даже если приложение не сможет обрабатывать больше данных, `H2Connection` не мешает клиенту посылать данные в соединение TCP. Это связано с тем, что HTTP/2 использует несколько кадров управления окнами и состоянием соединения. Эти дополнительные кадры не способны вызывать переполнение буфера, поэтому нет причин препятствовать их отправке клиентом.

Приложения, которые надлежащим образом управляют противодавлением, получают более широкие возможности с HTTP/2, чем с HTTP/1.1. Медленные части приложения или части, взаимодействующие с медленными клиентами, могут успешно замедляться, не ограничивая общий параллелизм системы. Это также гарантирует, что приложения, обслуживающие данные через HTTP/2, способны правильно обрабатывать перегрузку, ухудшая качество обслуживания управляемым образом, что предотвращает полную перегрузку.

Приложения могут использовать механизм управления противодавлением, чтобы гарантировать, что их обработчик запросов будет регистрировать `IPushProducer` для каждого обрабатываемого экземпляра `Request`. Класс `twisted.web.http.Request` реализует интерфейс `IConsumer` именно для этой цели.

Следует отметить, что интерфейсы `IConsumer`/`IPushProducer` ограничены и не обязательно должны предлагать все богатство API противодавления. Пример более удачного интерфейса, который в конечном итоге может заменить `IConsumer`/`IPushProducer`, можно найти в библиотеке `tubes`¹.

ТЕКУЩЕЕ ПОЛОЖЕНИЕ ДЕЛ И ВОЗМОЖНОСТЬ РАСШИРЕНИЯ В БУДУЩЕМ

Реализация HTTP/2 в Twisted вошла в состав версии Twisted 16.3, вышедшей в июле 2016 года. Эта реализация ограничена рядом необязательных зависимостей, которые необходимо установить, чтобы включить ее, а также некото-

¹ <https://twisted.github.io/tubes/>.

рыми требованиями к версии OpenSSL, используемой фреймворком Twisted. Эти ограничения фактически переводят поддержку HTTP/2 в состояние «бета».

Многие предприимчивые пользователи установили эту версию, включили поддержку HTTP/2 и помогли выявить некоторые ошибки. В результате теперь стек HTTP/2 в Twisted без проблем работает на огромном количестве машин. Это огромный успех, положительно сказавшийся на развитии проекта.

Есть несколько направлений для дальнейшего продолжения данной работы. Первый и самый важный – реализация клиента HTTP/2, способного прозрачно заменить клиента HTTP/1.1. Эта часть работы еще не завершена, но некоторый фундамент уже заложен.

Другая важная задача – реализация API для использования преимуществ HTTP/2. В частности, HTTP/2 позволяет серверу принудительно посылать сообщения, что дает возможность заранее начать отправку ресурсов, которые могут понадобиться клиенту для отображения страницы. Интересным улучшением могла бы стать реализация возможности для программ Twisted программно посылать ресурсы с использованием соответствующего API. Этого можно было бы добиться посредством анализа заголовков Link в традиционных приложениях WSGI.

Наконец, полезным дополнением мог бы стать API, позволяющий настраивать более детально стек HTTP/2. В настоящее время приложения Twisted не могут изменять конфигурацию HTTP/2, ни глобальную, ни отдельно для каждого соединения. Добавление этой поддержки является естественной эволюцией в направлении полноценной реализации HTTP/2.

Итоги

В этой главе мы познакомились с протоколом HTTP/2, который определяется в документе RFC 7540. Мы обсудили расширение twisted.web с поддержкой этого протокола, сосредоточив внимание на некоторых аспектах интеграции, а также конкретных проблемах, возникших во время разработки. Мы также рассмотрели важность противодействия для конкурентного программирования и увидели, насколько важную роль играют интерфейсы для расширяемости. Наконец, мы описали текущее состояние и будущие направления развития поддержки HTTP/2 в Twisted.

Глава 12

Twisted и Django Channels

ВВЕДЕНИЕ

В следующих разделах мы углубимся в структуру библиотеки Django Channels и технологии, использованные при ее создании. Также мы рассмотрим некоторые интересные детали библиотеки, которые можно применять при создании сложных многоуровневых распределенных приложений, поддерживающих возможность горизонтального масштабирования.

Python стал одним из первых языков программирования, для которых был определен стандартный интерфейс между веб-приложениями и веб-серверами, не использующий технологии CGI (Common Gateway Interface). Технология CGI, несмотря на свою эффективность, не отличается высокой производительностью, поэтому разработчики пришли к пониманию необходимости создания более богатого интерфейса между серверами и приложениями, в идеале с использованием примитивов и функций языка.

В 2003 году группа разработчиков ядра Python приняла документ PEP 333, определивший интерфейс шлюза веб-сервера (Web Server Gateway Interface, WSGI). WSGI – это спецификация API, которая позволяет веб-серверам, способным создавать объекты и вызывать функции Python (либо из Python, либо через C API), вызывать веб-приложения стандартным способом. Цель WSGI в том, чтобы отделить веб-фреймворки от веб-серверов, чтобы любой веб-сервер мог запускать любое веб-приложение на Python.

С этой точки зрения интерфейс WSGI оказался чрезвычайно успешным. Большинство читателей этой книги не помнят мир, существовавший до появления WSGI, поэтому описание выше может вызвать недоумение: как мог существовать мир, в котором веб-фреймворки и веб-серверы не были разделены? В мире, сформировавшемся после появления WSGI, сообщество Python стало свидетелем распространения замечательных фреймворков веб-приложений (таких как Django и Flask) и веб-серверов (таких как uWSGI, gunicorn и Twisted), пользующихся гибкостью WSGI.

Однако этот подход далек от идеала. В частности, сервер WSGI вызывает приложение WSGI обращением к синхронной функции Python и блокирует

ся, пока она не вернет управление. Такой принципиально синхронный вызов в WSGI означает, что WSGI-приложения очень непросто писать в асинхронном стиле. Это доставляет программистам определенные неудобства: они не могут использовать Twisted или `async` и `await`. Это делает веб-приложения на Python неэффективными: для обработки каждого параллельного веб-запроса требуется запустить новый поток выполнения. Неэффективность такого подхода понятна: в конце концов, именно по этой причине появился Twisted!

Библиотека Django Channels – это попытка дать возможность писать конкурентные приложения на Django с сохранением обратной совместимости с приложениями WSGI. Разработчики проделали гигантскую работу, потому что фактически «Django Channels» охватывает широкий спектр связанных между собой технических проектов. К ним относятся новый интерфейс для обмена данными между серверами и приложениями, интерфейс асинхронного серверного шлюза (Asynchronous Server Gateway Interface, ASGI); эталонный веб-сервер, реализующий серверную часть этого интерфейса (Daphne); и приложение Django, которое позволяет Django обрабатывать ASGI-запросы.

Результатом появления Django Channels стала переориентация Django с запросов и ответов на «события». Это позволило приложениям на основе Django Channels обрабатывать не только HTTP-запросы и ответы, но также и веб-сокеты и даже простые соединения TCP/UDP. Это было достигнуто делением всего веб-стека на три части:

- 1) сервер ASGI. Отвечает за прием входящих соединений и преобразование данных несущего протокола (например, HTTP) в сообщения ASGI, добавление этих сообщений в очереди («каналы»), прием сообщений из каналов и их обратное преобразование в данные несущего протокола. В эталонной реализации таким сервером является Daphne – веб-сервер на основе Twisted;
- 2) «внутренний канал», который в основном играет роль хранилища данных и может использоваться в качестве брокера сообщений. В простых приложениях таким внутренним каналом может быть просто общая память, но в больших приложениях для этого обычно используется Redis;
- 3) один или несколько «обработчиков». Эти обработчики прослушивают все или некоторые каналы и запускают соответствующий код при появлении сообщений. Обработчики могут быть последовательными и многопоточными. К этой части относится традиционный прикладной код Django.

Части 1 и 3 могут интегрироваться с Twisted. Однако большинство пользователей Django предпочитают придерживаться привычного синхронного стиля, по крайней мере пока, поэтому здесь нет почти ничего интересного.

Намного интереснее было бы обсудить Daphne и дизайн системы каналов в Channels. Организация Channels может служить полезным примером структурирования сложной, многоуровневой и распределенной системы с использованием Twisted и брокеров сообщений. В этой главе мы взглянем на Django

go Channels именно с этой стороны. Попутно обсудим связь Django Channels с Autobahn, реализацией WebSocket для Twisted.

ОСНОВНЫЕ КОМПОНЕНТЫ CHANNELS

Основу Django Channels составляет набор известных и надежных программных инструментов. Это главное преимущество библиотеки. Для столь распространенного программного обеспечения, как Django, надежность намного важнее «крутизны».

Как уже отмечалось, библиотека Channels делится на три компонента, и все они основываются на одном базовом программном обеспечении.

Первый компонент – Daphne – это веб-сервер, реализованный поверх twisted.web. В число целей проектирования Daphne входят также поддержка WebSocket, протокола, который не поддерживается в ядре Twisted, поэтому Daphne добавляет некоторые изменения в twisted.web, чтобы также использовать поддержку WebSocket из Autobahn. Следует подчеркнуть, что Daphne – это удивительно небольшой фрагмент кода, который в основном отвечает за преобразование протоколов HTTP и WebSocket в сообщения Channels, добавление их в очереди и чтение из очередей с обратным преобразованием в соединения.

Третий компонент – «обработчики» каналов – это процессы на Python, использующие веб-фреймворк Django и Channels. Они отвечают за прием и отправку сообщений в соответствующие очереди и сокрытие абстракции каналов от прикладного кода. Самое замечательное, что обычный код, использующий Django, практически не нужно изменять – вы сможете бесшовно обновить свои приложения на Django, раньше не использовавшие Channels.

Второй компонент – единственный в стеке, написанный не на Python: Redis. Redis – это база данных с открытым исходным кодом, хранящая пары ключ/значение в памяти и поддерживающая возможность хранения структурированных данных. Основная ее функция – хранение данных, но она обладает рядом свойств, которые позволяют использовать ее в роли брокера сообщений, включая возможность безопасного управления очередями.

Каждый из этих компонентов может быть развернут независимо от других и реализует свою часть топологии каналов. Все вместе они формируют законченное веб-приложение, где Daphne предоставляет поддержку протокола и взаимодействует с клиентами, Django поддерживает бизнес-логику, а Redis предоставляет услуги брокера сообщений между первыми двумя службами.

БРОКЕРЫ СООБЩЕНИЙ И ОЧЕРЕДИ

Важнейшая особенность Django Channels состоит в том, что обычные приложения Django могут работать как прежде, но приобретают одну дополнительную особенность: возможность горизонтального масштабирования независимо от веб-серверов, обслуживающих HTTP-трафик. По сути, обычные приложе-

ния Django, которые раньше блокировали обслуживание веб-трафика, внезапно могут стать асинхронными, что позволяет веб-серверу не блокироваться в ожидании доставки ответа. Как этого добиться без изменения кода?

Ключом к успеху является добавление брокера сообщений. Брокер сообщений, или система очередей, – это распространенный компонент в распределенных системах. Его цель – передача сообщений от нескольких производителей меняющемуся количеству потребителей, не требуя, чтобы производители или потребители знали о существовании друг друга.

Обычно в качестве основной абстракции брокеры сообщений используют *очередь* FIFO. Компоненты системы, генерирующие сообщения, добавляют эти сообщения в конец очереди FIFO. Эти сообщения удаляются из начала очереди одним или несколькими «обработчиками», отвечающими за выполнение некоторых действий, в зависимости от сообщения. Такая организация имеет много преимуществ: ее можно использовать как инструмент обнаружения служб или для устранения тесных связей между отправителями и получателями сообщений.

Преимущество подобного брокера сообщений заключается в том, что он отделяет окружения выполнения различных компонентов. В WSGI веб-приложение тесно связано с моделью выполнения веб-сервера, потому что веб-сервер должен вызывать функцию на Python, которая блокирует сервер до окончания выполнения. Такая тесная связь не позволяет веб-серверу и веб-приложению использовать разные подходы к организации конкурентного выполнения: в конечном итоге оба должны запускать однопоточный синхронный код¹.

Благодаря наличию брокера сообщений между веб-сервером и веб-приложением каждый из них может иметь свою парадигму выполнения. Более того, они могут использовать любую парадигму, которая позволит им отправлять сообщения и получать ответы через брокера сообщений. В этом случае Daphne, асинхронный веб-сервер на основе Twisted, может взаимодействовать с брокером сообщений, используя свою модель асинхронного программирования, а традиционные однопоточные синхронные обработчики Django могут выполняться в обработчиках, не мешая серверу.

Что еще более важно, рабочих процессов может быть намного больше, чем веб-серверов. Это позволяет значительно повысить производительность традиционных веб-приложений: вместо того чтобы тратить драгоценное время в ожидании на глобальной блокировке интерпретатора Python, каждый вызов приложения можно выполнять в отдельном процессе.

Это позволяет приложениям Django быть одновременно и синхронными, и конкурентными. Каждый обработчик запросов Django может быть реализо-

¹ Это не совсем так: twisted.web, асинхронный веб-сервер, способен запускать приложения WSGI с синхронной блокировкой. Для этого он вызывает приложение WSGI в фоновом потоке выполнения и использует Deferred для получения результата. Это в целом работоспособный прием, но ядро бизнес-логики при этом отправляется в синхронный пул фоновых потоков, что плохо подходит для масштабирования!

ван как обычная функция на Python с синхронной блокировкой, но приложение может запускать столько конкурентных процессов, сколько понадобится. Что особенно ценно, количество процессов-обработчиков может динамически меняться независимо от количества веб-серверов. Это обеспечивает независимое горизонтальное масштабирование каждого компонента приложения, что позволяет гораздо эффективнее использовать ресурсы.

Брокеры сообщений – это широко применяемый инструмент для добавления асинхронности в принципиально синхронные программы. При наличии возможности запускать несколько экземпляров однопоточного синхронного кода в отдельных процессах или потоках становится возможным увеличить степень асинхронности в приложении без существенной его переделки.

Кроме того, брокеры сообщений позволяют не беспокоиться о координации работы нескольких обработчиков, выполняющихся конкурентно. Каждый действует так, будто находится в своем маленьком мире, добавляя и удаляя данные из очередей, не беспокоясь о том, откуда эти данные пришли и куда они уходят. Брокер сообщений гарантирует, что очередь будет доступна произвольному числу обработчиков.

Да, брокеры сообщений не являются панацеей от всех бед, но они являются отличным инструментом поддержки масштабирования и параллелизма в неконкурентных окружениях выполнения.

РАСПРЕДЕЛЕННЫЕ МНОГОУРОВНЕВЫЕ СИСТЕМЫ В TWISTED

Django Channels – это не только полезный инструмент для развертывания горизонтально масштабируемых веб-приложений, но и наглядный пример организации распределенной многоуровневой программной системы.

Распределенная многоуровневая программная система – это система, построенная путем разделения функций системы на «уровни», взаимодействующие друг с другом с использованием некоторой шины сообщений. В случае приложения, использующего Django Channels, такая система обычно имеет три уровня: Daphne, Django и любую базу данных для хранения моделей Django (например, MySQL или PostgreSQL). Но вообще идея многоуровневой архитектуры имеет намного более общий характер.

Асинхронные сетевые фреймворки, такие как Twisted, часто являются ключевыми компонентами многоуровневых систем. Во многом это связано с тем, что многоуровневые системы неизбежно подвержены задержкам из-за использования формализованных или специальных механизмов RPC (вызова удаленных процедур). Поскольку каждый узел на данном уровне системы стремится использовать системные ресурсы максимально эффективно, многоуровневые системы, использующие методы асинхронного программирования, значительно более масштабируемы и эффективны.

Каноническая многоуровневая архитектура делит приложение на три уровня, каждый из которых отвечает за отдельный аспект приложения. Обычно один уровень занимается исключительно хранением данных (база данных),

другой отвечает за выполнение прикладной бизнес-логики, третий уровень – за представление. Этот шаблон широко распространен в практике разработки веб-приложений и в действительности является разновидностью шаблона «модель – представление – контроллер».

При создании многоуровневых приложений на основе Twisted необходимо определить механизм связи между уровнями. Однако в конечном итоге все сводится к созданию механизма RPC в том или ином виде, позволяющем отдельным уровням делегировать работу другим уровням. Учитывая, что приложениям в любом случае требуется уровень RPC, можно сэкономить много сил и времени, взяв за основу какой-то стандартный механизм RPC.

Наиболее распространенным выбором на роль механизма RPC является REST. Это отличный выбор, учитывая превосходную поддержку HTTP в Twisted, но в зависимости от конкретного приложения иногда предпочтительнее может оказаться любой другой механизм RPC. Ключом к успеху архитектуры такого типа является понимание, что по своей природе Twisted очень хорошо подходит для приложений на основе RPC: если ваше основное приложение предполагает асинхронное выполнение, добавление большего количества асинхронных уровней часто оказывается относительно простым делом. При тщательном подходе к выбору RPC и дизайна приложения можно обеспечить произвольное горизонтальное масштабирование этого приложения. Все крупнейшие в мире веб-проекты реализованы именно в таком стиле, и полезно знать, что Twisted предлагает массу инструментов, которые вы можете освоить самостоятельно.

ТЕКУЩЕЕ ПОЛОЖЕНИЕ ДЕЛ И ВОЗМОЖНОСТЬ РАСШИРЕНИЯ В БУДУЩЕМ

9 сентября 2016 года библиотека Channels официально стала одним из проектов Django. Это означает, что ее разработка ведется под эгидой проекта Django и Django Software Foundation, но не является частью основного репозитория Django. Проект находится в стадии активной разработки и может использоваться в промышленном окружении.

В настоящее время он поддерживает большинство основных функций. HTTP/1.1, HTTP/2 и WebSocket поддерживаются полностью, хотя, так же как в Twisted, некоторые возможности HTTP/2 пока не реализованы. Redis поддерживается как основной брокер сообщений, но для небольших приложений с не меньшим успехом можно использовать другие очереди сообщений, хранящие свои данные в памяти.

Будущие направления развития Django Channels многочисленны и разнообразны. Есть несколько направлений развития окружения для развертывания сложных конкурентных веб-приложений. Дополнительные брокеры сообщений, альтернативные серверы ASGI и даже уровни совместимости с различными веб-фреймворками: все это и многое другое можно и должно усовер-

шенствовать. Более широкая поддержка альтернативных протоколов также, вероятно, будет иметь некоторое значение для проекта.

Конечно, идеальным в долгосрочной перспективе было бы включение модели каналов в ядро Django. Это упростило бы разработку масштабируемых приложений в Django и помогло бы разработчикам писать свои приложения, изначально поддерживающие масштабируемость.

Итоги

В этой главе мы познакомились с Django Channels – библиотекой для разработки веб-приложений на основе Django с использованием модели конкурентного и асинхронного программирования. Мы обсудили базовую архитектуру Channels и рассмотрели основные технологии. Затем проанализировали, как эти технологии можно использовать для проектирования произвольных многоуровневых распределенных систем и как в таких системах максимально использовать возможности Twisted. Наконец, разобрали направления развития Channels в будущем.

Предметный указатель

Символы

`_populate_row()`, метод, 222

`.tar`-файлы, 210

А

А/В-тестирование, 195

Access Control List (ACL), списки управления доступом, 223

AccountFileChecker, 219

AccountURLChecker, 219

ACL (Access Control List), списки управления доступом, 223

aiohttp, 277

AMP (Asynchronous Messaging Protocol), протокол асинхронного обмена сообщениями, 195

команды, 199

плагин, 197

ASGI (Asynchronous Server Gateway Interface), 324

assertFailure, 249

assertNoResult, 250

async/await, конструкции, 258

Asynchronous Messaging Protocol (AMP), протокол асинхронного обмена сообщениями, 195

команды, 199

плагин, 197

asyncio и Twisted

PEP 3156, 271

механизм обещаний, 272

циклы событий, 273

`asyncio.get_event_loop`, 273

`asyncio.set_event_loop`, 273

`epoll`, Linux, 273

`kqueue`, macOS, 273

`twisted.internet.reactor`, 273

выбор реактора, 273

реактор, 273

стратегии, 273

asyncio, пакет, 271

`asyncio.Future`, класс, 272

`asyncio.get_event_loop_policy`, 273

`asyncio.set_event_loop_policy`, 273

Autobahn, 257

Autobahn, библиотека, 233

Automat, библиотека, 241

В

BuilBot

API баз данных, 300

Docker, протокол, 295

Trial, 299

автоматическое тестирование, 298

барьер конкуренции, 296

блокировки `Deferred`, 298

имитации компонентов, 300

конкурентный доступ к общим

ресурсам, 296

поток реактора, 296

тестирование отложенных

вычислений, 299

функции из пула потоков, 297

Buildbot

db, модуль, 292

`flushEventualQueue`, 291

Mozilla, 282

`requests`, 293

SQLAlchemy, 292

Tinderbox, 282

асинхронные инструменты, 288

асинхронные службы, 289

`addService`, 289

`AsyncMultiService`, 289

`ClientService`, класс, 290

`disownServiceParent`, 289

`IService`, 288

`IServiceCollection`, 288

`MessageQueueConnector`, 290

`setServiceParent`, 289

`startService`, 289

веб-интерфейс, 283

- дребезг, 288
- история, этапы асинхронной сборки, 287
- кеш LRU, 291
- определение, 282
- синхронные библиотеки, 292
- синхронные API, 286
- синхронный подход, 288
- цикл запрос/ответ, 287

Buildbot 0.9.0, 283

buildbot.test.fake.fakemaster.FakeMaster,
класс, 300

C

callLater, 313

CDN (Content Distribution Network), сеть
распространения контента, 182

CGI, стандарт

- environment, параметр, 169
- переменные окружения, 169

CGI (Common Gateway Interface),
обобщенный интерфейс шлюза, 166

CHK (Content-Hash Key), хеш-ключ
содержимого, 204

Clock(), объект, 253

Common Gateway Interface (CGI),
обобщенный интерфейс шлюза, 166

Content-Disposition, заголовок, 215

Content Distribution Network (CDN), сеть
распространения контента, 182

Content-Hash Key (CHK), хеш-ключ
содержимого, 204

convert_error, обработчик ошибок, 222

CORS, политика, 263

Crossbar.io, 264

D

Daphne, 324

defer.cancel(), 236

Deferred.addCallback, функция, 272

Deferred.fromFuture, метод класса, 276

DeferredLock, 298

DeferredQueue, 298

DeferredSemaphore, 298

Delegate API, 241

DelegatingResource, класс, 185

Diffie-Hellman, протокола обмена
ключами Диффи-Хеллмана, 229

dircaps (ссылки на каталоги), 203

Dispatcher, 219

Distributed Hash Table (DHT),
распределенная хеш-таблица, 233

Django Channels, 323

- компоненты, 325

- текущее положение дел и возможность
расширения в будущем, 328

Docker, 147

Python в

- варианты сборки, 160

- автоматизация с Dockerpy, 161

- копирование пакетов wheel между
этапами, 161

- развертывание виртуального
окружения, 158

- развертывание в формате Pex, 159

- развертывание полноценного
окружения, 153

runc и containerd, 149

Twisted в, 162

- пользовательские плагины, 162

- ENTRYPOINT и PID, 162

- NColon, 163

введение, 147

клиент, 149

контейнеры, 147

многоступенчатая сборка, 151

образы контейнеров, 148

реестр, 150

сборка образов, 150

F

failResultOf, 250

failureResultOf, 251

File, песчур, 183

filecaps (ссылки на файлы), 203

flushEventualQueue(), функция, 253

FTP, протокол передачи файлов, 202

FTPAvatarID, объект, 220

FTPFactory, 218

Future.add_done_callback, 272

G

get_size(), метод, 213

get_tor(), функция, 250

H

H2Stream, 309

Handler, 219

HAProxy, 191

HTTP/2, 301

- мультиплексирование, 309
- проблемы, 312
- приоритеты, 311
- противодавление, 315

HTTP, долгий опрос, 232

HTTP, заголовки, 169

HTTP, код ответа, 169

HTTP протокол, 214

HTTP AJAX-запросы, 255

HttpClientService, 294

HTTP GET запрос, 206

HTTP PUT и POST запросы, 206

I

IANA, стандарт, 171

IConsumer, 317

ICredentialsChecker, 219

IFTPShell, 219

inlineCallbacks

- requestFieldDeferred, 88
- requestFieldGenerator, 88
- отложенные вычисления, 87

IPushProducer, 317

IReactor, 313

IReactorCore.addSystemEventTrigger, 274

IReadable.read(), метод, 214

ITransport, 308

J

Jenkins, 298

K

Klein, 107, 112

- amount, аргумент, 113
- виртуальное окружение, 112
- декодирование сообщений, 113
- декоратор, 112
- и Deferred, 114
- механизм шаблонов Plating, 115

L

libp2p, протокол, 233

list(), метод, 221

M

- Magic Wormhole, инструмент
- диаграмма работы, 229
- код червоточины, 227

- командной строки на Python, 228

- makePromiseResolverPair(), функция, 244

- mkdir, команда, 204

- MySQL, 292

N

- NColony, 163

- Nevow, 213

P

- PEP 333, 323

- PEP 3153, 271

- PEP 3156, 271

- PEP (Python Enhancement Proposal),

- предложения по улучшению Python, 167

- PingPongProtocol, управление потоком данных, 42

- pip, инструмент, 108

- Portal, 219

- PostgreSQL, 292

- Promise/Future и Deferred, 244

- PUB/SUB – Publish/Subscribe

- (публикация/подписка), 257

- Pyramid, фреймворк, 173

- Python, 323

- варианты сборки, 160

- автоматизация с Dockerpy, 161

- копирование пакетов wheel между
- этапами, 161

- развертывание виртуального
- окружения, 158

- развертывание в формате Pex, 159

- развертывание полноценного
- окружения, 153

- Python 3, 209

- Python Enhancement Proposal (PEP),

- предложения по улучшению Python, 167

- PYTHONPATH, переменная

- окружения, 177

- python-requests, библиотека, 293

R

- Raspberry Pi, 256

- reactor.seconds(), 188

- readcap (ссылка для чтения), 203

- Request.write, 306

- RPC – Remote Procedure Calls (вызов
- удаленных процедур), 257

S

sans io-реализация HTTP/2, 305
 Secure Socket Layer (SSL), уровень защищенных сокетов, 179
 self.assertFailure(), проверка, 249
 Server Name Indication (SNI), индикация имени сервера, 180
 Server Sent Events (события, посылаемые сервером), 232
 SNI (Server Name Indication), индикация имени сервера, 180
 SPAKE2, протокол, диаграмма работы, 229
 SQLite, 292
 SSL (Secure Socket Layer), уровень защищенных сокетов, 179
 start_response, параметр, 169
 successResultOf, 250
 SynchronousTestCase, 299

T

Tahoe-LAFS
 FTP, 202
 архитектура, 206
 веб-интерфейс, 212
 использование Twisted, 208
 клиенты, 203
 посредники, 203
 распределенная система хранения, 202
 серверы, 203
 Task, класс, 275
 then(), метод, 245
 TLS. См. Transport Layer Security (TLS), защита транспортного уровня
 Transport Layer Security (TLS), защита транспортного уровня, 179
 аутентификация конечной точки, 179
 шифрование, 179
 transport.write, 306, 313
 Travis-CI, 298
 req, 107, 277
 Deferred, 110
 download, функция, 110
 feedparser, 111
 агрегирование каналов, 109
 библиотека, 294
 внедрение зависимостей, 110
 декоратор, 111
 инкапсуляция, 108

 клиентский API, 110
 Trial, 299
 Twisted
 API, 223
 несовместимости, 223
 разрешения для файлов, 224
 pipsi, утилита, 225
 tap-плагин, 175
 в Docker
 пользовательские плагины, 162
 ENTRYPOINT и PID, 162
 NColony, 163
 и реальный мир, 47
 система управления потоком
 активные производители, 59
 потребители, 62
 Twisted HTTP/2
 текущее положение дел и возможность расширения в будущем, 321
 противодавление, 315
 цели проектирования
 бесшовная интеграция, 303
 оптимизация поведения, 304
 повторное использование кода, 305
 twisted.internet.task.react, 273
 txrequests, библиотека, 294
 txtorcon, переменная, 250

V

verifyscp (ссылка для проверки), 204

W

WAPI (Web Application Programming Interface), программный интерфейс для веб-приложений, 212
 Web Application Programming Interface (WAPI), программный интерфейс для веб-приложений, 212
 WebOb, пакет, 172
 WebSocket
 из Python в JavaScript, 261
 разметка HTML, 262
 сервер на Python, 261
 из Python в Python
 self.sendMessage(), 261
 эхо-клиент, 259
 эхо-сервер, 258
 и Twisted, 256
 микросервисная архитектура, 256

- установка Autobahn, 257
- экосистема Autobahn, 257
- платформа аутентификации, 256
- процессы кеширования, 256
- распространение событий, 256
- рассылка уведомлений, 256
- WebSocket Application Messaging Protocol (WAMP), 263
 - win32api, 264
 - аутентификация, 263
 - клиент на JS, 266
 - клиент на Python, 267
 - клиентский код, 266
 - сериализация, 263
 - события, 263
 - совместимый маршрутизатор, 264
 - сопряжение сообщений, 263
 - широковещательная рассылка сообщений, 263
- WebSockets, 233
- Web Standard Gateway Interface (WSGI), стандартный интерфейс веб-шлюза, 166
- Web User Interface (WUI), веб-интерфейс пользователя, 212
- WGI-сервер, встроенное планирование задач, 186
- writesap (ссылка для записи), 203
- WSGI-сервер, 265
 - python -m twisted, 174
 - setup.py, 177
 - вызываемый объект Python, 171
 - демонстрационное веб-приложение, 174
 - для промышленной эксплуатации и разработки, 178
 - параметр --port, 174
 - плагин, 175
 - приложение, 168
 - путь поиска по умолчанию, 177
- wsgiref, модуль, 170
- WSGI (Web Server Gateway Interface), 323
- WSGI (Web Standard Gateway Interface), стандартный интерфейс веб-шлюза, 166
- WUI (Web User Interface), веб-интерфейс пользователя, 212

А

- Агрегатор каналов, первая версия klein.Plating, объект, 117

- Plating.CONTENT, слот, 118
- retrieveFeed, метод, 120
- SimpleFeedAggregation, класс, 119
- slowIncrement, страница, 118
- и Deferred, 120
- метод отображения, 120
- обработчики, 118
- реализация, 122
- Агрегирование каналов, 108
- Асинхронная обработка исключений
 - метод callback, 72
 - реализация в отложенных вычислениях, 77
- Асинхронное программирование, 69
 - обработка ошибок, 73
- Асинхронное тестирование с объектами Deferred, 248
- Асинхронный Python в Buildbot, 283
 - async/await, 286
 - asyncio, 286
 - Builder.startBuild, метод, 284
 - Deferred, 284
 - deferredGenerator, 285
 - inlineCallbacks, 285
 - NodeJS, 286
 - дополнительные возможности, 285
 - синхронная модель, 286

Б

- Балансировка нагрузки, 191
- Брокер сообщений/система очередей, 325

В

- Веб-интерфейс, 212
 - FileNodeHandler, 213
 - Range, заголовок, 215
 - URIHandler, ресурс, 212
 - заголовки Range, 215
 - каталоги, 214
 - коды ошибок HTTP, 216
 - корневой ресурс, 212
 - пользователя (Web User Interface, WUI), 212
 - сохранение на диск, 215
- Виртуальное окружение, 109

Г

- Генераторы
 - generatorFunction, 83

выражение `yield`, 83
 метод `send`, 84, 86
 метод `throw`, 86

Д

Деление веб-стека, 324
 Динамическая конфигурация, 195
 управляющая программа, 200
 Долгий опрос HTTP, 232

З

Загрузка зависимостей, 209
 Закон Деметры, 127
 Защита транспортного уровня (Transport Layer Security, TLS), 179
 аутентификация конечной точки, 179
 шифрование, 179

И

Индикация имени сервера (Server Name Indication, SNI), 180
 Инструменты композиции функций, 66
 Инструменты поддержки выполнения в режиме демона
 `.tar`-файлы, 210
 инструменты командной строки
 Tahoe-LAFS, 209
 Интеграционные тесты, 249
 Интерфейс
 FTP, 218
 аватар, 219
 конечная точка, 218
 область (`realm`), 219
 объект проверки, 219
 SFTP, 223
 полиморфизм, 304
 Интерфейсные протоколы, 211

К

Каноническая многоуровневая архитектура, 327
 Кеш LRU, 291
 Клиент Wormhole, архитектура, 239
 Конечные автоматы, 240
 Криптографическая хеш-функция, 215

М

Меркла хеш-дерево, 215
 Модель маршрутизации на основе сопоставления с шаблоном, 182

О

Обещаний механизм
 `Deferred.addCallback`, 272
 поведение `Deferred` и `Future`, 272
 циклы событий, 273
 Обобщенный интерфейс шлюза (Common Gateway Interface, CGI), 166
 Обработчики событий `requestField`, 66
 Одноразовые наблюдатели, 243
 Односторонние приложения (Single Page Applications, SPA), 119
 Отложенные вычисления в Twisted
 заполнители для будущих значений, 70, 72
 метод `addCallback`, 72, 74
 метод `errback`, 73, 76
 мультиплексирование, 98
 обработка ошибок, 73
 обработчики ошибок, 77
 обработчики результатов, 76
 обработчики событий
 `requestField`, 66
 композиция функций, 66
 программирование в стиле продолжений, 69
 тестирование, 102, 103
 Отложенные вычисления и конечные автоматы, 240
 Отменяемые отложенные операции, 236

П

Подсказки для подключения, 230
 Порты завершения ввода/вывода (I/O Completion Ports, IOCP), 70
 Поток реактора, 296
 Предвзятое подбрасывание монеты, 197
 Предложения по улучшению Python (Python Enhancement Proposal, PEP), 167
 Приложение Pyramid с поддержкой A/B-тестирования, 195
 Пример прокси с `aiohttp` и `treq`, 277
 `aiohttp.web.StreamResponse`, 280
 `asFuture`, вспомогательная функция, 280
 `Deferred.fromFuture`, 278
 `ensureDeferred`, 278
 `inlineCallbacks`, 278
 заголовки запроса, 280
 реактор `asyncio`, 278

реализация прокси-сервера, 278
 сервер aiohttp, 277
 сопрограмма, 278

Программирование в стиле продолжений, 69

Программный интерфейс для веб-приложений (Web Application Programming Interface, WAPI), 212

Протокол асинхронного обмена сообщениями (Asynchronous Messaging Protocol, AMP), 195
 команды, 199
 плагин, 197

Протокол обмена асинхронными сообщениями (Asynchronous Message Protocol, AMP), 123

Протокол обмена ключами

Диффи–Хеллмана (Diffie–Hellman), 229

Протокол передачи файлов (FTP), 202

Р

Разработка через тестирование, 123
 _service.py, 131
 FeedAggregation, экземпляр, 132
 resource(), метод, 126
 setUp, метод, 126
 XPath, 129
 веб-служб, 126
 интерфейс, 123
 классы Channel и Item, 131
 реализация, 129
 решение, 131
 тестирование Klein с помощью StubTreq, 126
 FeedAggregationTests, 126
 root, метод, 128
 отображение каналов в форматах HTML и JSON, 128, 129
 тестирование treq с помощью Klein, 133
 журналирование с twisted.logger, 136, 140
 запуск приложений с помощью twist, 141

Рандеву, сервер, 229

Распределенная хеш-таблица (Distributed Hash Table, DHT), 233

Распределенные многоуровневые системы, 327

Рекомендации, asyncio и Twisted
 asyncio.Future, 274
 asyncio.Task, 274
 Deferred, 274
 task.react, 276
 взаимодействия, 275
 сопрограммы asyncio, 274

С

Сервер рандеву, 229
 архитектура, 234
 база данных, 235

Сервер транзитной ретрансляции, 238

Сетевые протоколы и совместимость клиентов, 231

Сеть распространения контента (Content Distribution Network, CDN), 182

Синхронное тестирование
 и отсроченный вызов, 252
 отсроченный вызов, 247
 с объектами Deferred, 249

Слой шифрования защищенной командной оболочки SSH, 223

Событийно-ориентированное программирование, 69
 преимущества, 44, 47
 управлением потоком, 58

События
 и время
 LoopingCall, 53
 reactor.callLater, 51
 сложности, 53
 интерфейсы в zope.interface, 55, 56

Сокеты в режиме совместного использования, 192

Сопрограммы
 future-подобные объекты, 95
 в Python, yield from, генераторы, 90
 метод
 async, 91
 await, 91, 94
 ensureDeferred, 97
 send, 92
 throw, 92
 стек генераторов, 93

Списки управления доступом (Access Control List, ACL), 223

Стандартный интерфейс веб-шлюза (Web Standard Gateway Interface, WSGI), 166

Статические ресурсы, 183

Статические файлы

 postpath, 186

 prepath, 186

 setup.py, 183

 WSGI, 185

 конструктор суперкласса, 186

 листовые ресурсы, 183

 манифест, 183

Стратегии параллельного выполнения, 191

Стрелочные функции, 245

Структура каталога проекта, 124

Т

Транзитные соединения, 230

Транзитный клиент, 236

Транспорты и протоколы
и реакторы Twisted, 44
клиенты и серверы, 43

У

Упаковка/распространение, 209

Управление потоком

 в событийно-ориентированном
 программировании, 58

 в Twisted

 активные производители, 59

 потребители, 62

Управляющая программа, 200

Уровень защищенных сокетов (Secure Socket Layer, SSL), 179

Х

Хеш-дерево Меркла, 215

Хеш-ключ содержимого (Content-Hash Key, CHK), 204

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@aliants-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planet.ru**.

Моше Задка, Марк Уильямс, Кори Бенфилд, Брайан Уорнер,
Дастин Митчелл, Кевин Сэмюэл, Пьер Тарди

Twisted из первых рук

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод *Киселев А. Н.*

Корректоры *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура PT Serif. Печать офсетная.

Усл. печ. л. 27,46. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**