

Asynсіo и конкурентное программирование на Python



Мэттью Фаулер

 MANNING

 ДМК
издательство

Asynсio и конкурентное программирование на Python

Если типичную программу на стандартном Python подвергнуть слишком высокой нагрузке, то она будет работать с черепашьей скоростью. Для решения этой проблемы была разработана библиотека asynсio, которая позволяет разбить программу на более мелкие задачи и планировать их выполнение. В итоге получающиеся приложения работают молниеносно и допускают масштабирование.

В этой книге асинхронное, параллельное и конкурентное программирование рассматривается на конкретных примерах. Сложные для понимания вопросы иллюстрируются с помощью диаграмм, позволяющих наглядно представить, как работают задачи. Вы узнаете, как asynсio преодолевает ограничения Python и способствует ускорению медленных веб-серверов и микросервисов. Вы даже научитесь сочетать asynсio с традиционной многопроцессной обработкой, получив в награду резкий скачок производительности.

*Для программистов на Python среднего уровня.
Опыт работы с конкурентностью не требуется.*

Рассматриваемые темы:

- построение веб-API и конкурентная отправка веб-запросов с помощью библиотеки aiohttp;
- конкурентное выполнение тысяч SQL-запросов;
- создание задания MapReduce, способного конкурентно обрабатывать гигабайты данных;
- совместное использование многопоточности и asynсio для разрешения проблемы блокирующего кода.

За плечами Мэттью Фаулера 15 лет работы в области программной инженерии на разных должностях, от архитектора до руководителя отдела разработки.

«Ясное практическое руководство по применению asynсio в работе над реальными задачами».

Дидье Гарсиа, Software Sushi

«Хорошо организованный и написанный текст... знакомит читателей с различными способами применения конкурентности».

Дэн Шеих, BCG Digital Ventures

«Если вы хотите перейти на следующий уровень владения Python, то больше ничего искать не нужно. Эта книга для вас».

Эли Майост, IBM

«Детальное рассмотрение одной из самых сложных тем в программировании на Python — использования библиотеки asynсio».

*Питер Уайт,
университет Чарльза Стёрта*

ISBN 978-5-93700-166-5



9 785937 001665 >

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@aliens-kniga.ru

DMK
www.dmk.pф

Мэттью Фаулер

Asyncio и конкурентное программирование на Python

Python Concurrency with asyncio

MATTHEW FOWLER



MANNING
Shelter Island

Asynсio и конкурентное программирование на Python

МЭТТЮ ФАУЛЕР



Москва, 2023

УДК 004.42
ББК 32.372
Ф28

Фаулер М.

Ф28 Asyncio и конкурентное программирование на Python / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2022. – 398 с.: ил.

ISBN 978-5-93700-166-5

Из данной книги вы узнаете, как работает библиотека `asyncio`, как написать первое реальное приложение и как использовать функции веб-API для повышения производительности, пропускной способности и отзывчивости приложений на языке Python. Рассматривается широкий круг вопросов: от модели однопоточной конкурентности до многопроцессорной обработки.

Издание будет полезно не только Python-разработчикам, но и всем программистам, которые хотят лучше понимать общие проблемы конкурентности.

УДК 004.42
ББК 32.372

© DMK Press 2022. Authorized translation of the English edition © 2022 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of 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 978-1-6172-9866-0 (англ.)
ISBN 978-5-93700-166-5 (рус.)

© Manning Publications, 2022
© Перевод, оформление, издание, ДМК Пресс, 2022

*Посвящается моей любимой супруге Кэти.
Спасибо, что ты всегда рядом.*

Оглавление

1	■ Первое знакомство с asyncio	21
2	■ Основы asyncio	45
3	■ Первое приложение asyncio	74
4	■ Конкурентные веб-запросы	101
5	■ Неблокирующие драйверы баз данных	130
6	■ Счетные задачи	157
7	■ Решение проблем блокирования с помощью потоков	189
8	■ Потоки данных	223
9	■ Веб-приложения	251
10	■ Микросервисы	279
11	■ Синхронизация	303
12	■ Асинхронные очереди	327
13	■ Управление подпроцессами	350
14	■ Продвинутое использование asyncio	365

Содержание

Оглавление	6
Предисловие	12
Благодарности	14
Об этой книге	15
Об авторе	19
Об иллюстрации на обложке	20

1	Первое знакомство с <i>asuncio</i>	21
1.1	Что такое <i>asuncio</i> ?	22
1.2	Что такое ограниченность производительностью ввода-вывода и ограниченность быстродействием процессора	24
1.3	Конкурентность, параллелизм и многозадачность	25
1.3.1	Конкурентность	25
1.3.2	Параллелизм	26
1.3.3	Различие между конкурентностью и параллелизмом	27
1.3.4	Что такое многозадачность	28
1.3.5	Преимущества кооперативной многозадачности	28
1.4	Процессы, потоки, многопоточность и многопроцессность	29
1.4.1	Процесс	29
1.4.2	Поток	29
1.5	Глобальная блокировка интерпретатора	33
1.5.1	Освобождается ли когда-нибудь GIL?	37
1.5.2	<i>Asuncio</i> и GIL	39
1.6	Как работает однопоточная конкурентность	39
1.6.1	Что такое сокет?	39
1.7	Как работает цикл событий	41
	Резюме	44

2	Основы <i>asuncio</i>	45
2.1	Знакомство с сопрограммами	46
2.1.1	Создание сопрограмм с помощью ключевого слова <i>asunc</i>	46
2.1.2	Приостановка выполнения с помощью ключевого слова <i>await</i>	48
2.2	Моделирование длительных операций с помощью <i>sleep</i>	49
2.3	Конкурентное выполнение с помощью задач	52

2.3.1	Основы создания задач	52
2.3.2	Конкурентное выполнение нескольких задач	53
2.4	Снятие задач и задание тайм-аутов	56
2.4.1	Снятие задач	56
2.4.2	Задание тайм-аута и снятие с помощью <code>wait_for</code>	57
2.5	Задачи, сопрограммы, будущие объекты и объекты, допускающие ожидание	59
2.5.1	Введение в будущие объекты	59
2.5.2	Связь между будущими объектами, задачами и сопрограммами	61
2.6	Измерение времени выполнения сопрограммы с помощью декораторов	62
2.7	Ловушки сопрограмм и задач	65
2.7.1	Выполнение счетного кода	65
2.7.2	Выполнение блокирующих API	67
2.8	Ручное управление циклом событий	68
2.8.1	Создание цикла событий вручную	69
2.8.2	Получение доступа к циклу событий	69
2.9	Отладочный режим	70
2.9.1	Использование <code>asuncio.run</code>	70
2.9.2	Использование аргументов командной строки	71
2.9.3	Использование переменных окружения	71
	Резюме	72

3	Первое приложение <code>asuncio</code>	74
3.1	Работа с блокирующими сокетами	75
3.2	Подключение к серверу с помощью <code>telnet</code>	78
3.2.1	Чтение данных из сокета и запись данных в сокет	79
3.2.2	Разрешение нескольких подключений и опасности блокирования ...	80
3.3	Работа с неблокирующими сокетами	82
3.4	Использование модуля <code>selectors</code> для построения цикла событий сокетов	86
3.5	Эхо-сервер средствами цикла событий <code>asuncio</code>	89
3.5.1	Сопрограммы цикла событий для сокетов	89
3.5.2	Проектирование асинхронного эхо-сервера	90
3.5.3	Обработка ошибок в задачах	92
3.6	Корректная остановка	94
3.6.1	Прослушивание сигналов	95
3.6.2	Ожидание завершения начатых задач	96
	Резюме	99

4	Конкурентные веб-запросы	101
4.1	Введение в <code>aihttp</code>	102
4.2	Асинхронные контекстные менеджеры	103
4.2.1	Отправка веб-запроса с помощью <code>aihttp</code>	105
4.2.2	Задание тайм-аутов в <code>aihttp</code>	107
4.3	И снова о конкурентном выполнении задач	108
4.4	Конкурентное выполнение запросов с помощью <code>gather</code>	111
4.4.1	Обработка исключений при использовании <code>gather</code>	113
4.5	Обработка результатов по мере поступления	115
4.5.1	Тайм-ауты в сочетании с <code>as_completed</code>	117

4.6	Точный контроль с помощью wait	119
4.6.1	Ожидание завершения всех задач	119
4.6.2	Наблюдение за исключениями	122
4.6.3	Обработка результатов по мере завершения	123
4.6.4	Обработка тайм-аутов	126
4.6.5	Зачем оборачивать программы задачами?	127
Резюме	128

5	Неблокирующие драйверы баз данных	130
5.1	Введение в asynсpg	131
5.2	Подключение к базе данных Postgres	131
5.3	Определение схемы базы данных	133
5.4	Выполнение запросов с помощью asynсpg	135
5.5	Конкурентное выполнение запросов с помощью пулов подключений	138
5.5.1	Вставка случайных SKU в базу данных о товарах	138
5.5.2	Создание пула подключений для конкурентного выполнения запросов	142
5.6	Управление транзакциями в asynсpg	146
5.6.1	Вложенные транзакции	148
5.6.2	Ручное управление транзакциями	149
5.7	Асинхронные генераторы и потоковая обработка результатирующих наборов	151
5.7.1	Введение в асинхронные генераторы	151
5.7.2	Использование асинхронных генераторов и потокового курсора	153
Резюме	156

6	Счетные задачи	157
6.1	Введение в библиотеку multiprocessing	158
6.2	Использование пулов процессов	160
6.2.1	Асинхронное получение результатов	161
6.3	Использование исполнителей пула процессов в сочетании с asynсio	162
6.3.1	Введение в исполнители пула процессов	162
6.3.2	Исполнители пула процессов в сочетании с циклом событий	164
6.4	Решение задачи с помощью MapReduce и asynсio	166
6.4.1	Простой пример MapReduce	167
6.4.2	Набор данных Google Books Ngram	169
6.4.3	Применение asynсio для отображения и редукции	170
6.5	Разделяемые данные и блокировки	175
6.5.1	Разделение данных и состояние гонки	176
6.5.2	Синхронизация с помощью блокировок	179
6.5.3	Разделение данных в пулах процессов	181
6.6	Несколько процессов и несколько циклов событий	184
Резюме	188

7	Решение проблем блокирования с помощью потоков	189
7.1	Введение в модуль threading	190

7.2	Совместное использование потоков и <code>asyncio</code>	194
7.2.1	Введение в библиотеку <code>requests</code>	194
7.2.2	Знакомство с исполнителями пула потоков	195
7.2.3	Исполнители пула потоков и <code>asyncio</code>	197
7.2.4	Исполнители по умолчанию	198
7.3	Блокировки, разделяемые данные и взаимоблокировки	200
7.3.1	Реентерабельные блокировки	201
7.3.2	Взаимоблокировки	204
7.4	Циклы событий в отдельных потоках	206
7.4.1	Введение в <code>Tkinter</code>	207
7.4.2	Построение отзывчивого UI с помощью <code>asyncio</code> и потоков	209
7.5	Использование потоков для выполнения счетных задач	217
7.5.1	<code>hashlib</code> и многопоточность	217
7.5.2	Многопоточность и <code>NumPy</code>	220
	Резюме	222

8	Потоки данных	223
8.1	Введение в потоки данных	224
8.2	Транспортные механизмы и протоколы	224
8.3	Потоковые читатели и писатели	228
8.4	Неблокирующий ввод данных из командной строки	231
8.4.1	Режим терминала без обработки и сопрограмма <code>read</code>	235
8.5	Создание серверов	242
8.6	Создание чат-сервера и его клиента	244
	Резюме	249

9	Веб-приложения	251
9.1	Разработка REST API с помощью <code>aiohhttp</code>	252
9.1.1	Что такое REST?	252
9.1.2	Основы разработки серверов на базе <code>aiohhttp</code>	253
9.1.3	Подключение к базе данных и получение результатов	255
9.1.4	Сравнение <code>aiohhttp</code> и <code>Flask</code>	260
9.2	Асинхронный интерфейс серверного шлюза	263
9.2.1	Сравнение <code>ASGI</code> и <code>WSGI</code>	263
9.3	Реализация <code>ASGI</code> в <code>Starlette</code>	264
9.3.1	Оконечная REST-точка в <code>Starlette</code>	265
9.3.2	<code>WebSockets</code> и <code>Starlette</code>	266
9.4	Асинхронные представления <code>Django</code>	269
9.4.1	Выполнение блокирующих работ в асинхронном представлении	275
9.4.2	Использование асинхронного кода в синхронных представлениях	277
	Резюме	278

10	Микросервисы	279
10.1	Зачем нужны микросервисы?	280
10.1.1	Сложность кода	281
10.1.2	Масштабируемость	281
10.1.3	Независимость от команды и технологического стека	281
10.1.4	Чем может помочь <code>asyncio</code> ?	281
10.2	Введение в паттерн <code>backend-for-frontend</code>	282
10.3	Реализация API списка товаров	283

10.3.1	Сервис избранного.....	284
10.3.2	Реализация базовых сервисов	284
10.3.3	Реализация сервиса backend-for-frontend	289
10.3.4	Повтор неудачных запросов	294
10.3.5	Паттерн Прерыватель	297
	Резюме	302

11	Синхронизация	303
11.1	Природа ошибок в модели однопоточной конкурентности	304
11.2	Блокировки	309
11.3	Ограничение уровня конкурентности с помощью семафоров	312
11.3.1	Ограниченные семафоры	315
11.4	Уведомление задач с помощью событий	317
11.5	Условия	322
	Резюме	326

12	Асинхронные очереди	327
12.1	Основы асинхронных очередей	328
12.1.1	Очереди в веб-приложениях	335
12.1.2	Очередь в веб-роботе	338
12.2	Очереди с приоритетами	341
12.3	LIFO-очереди	347
	Резюме	349

13	Управление подпроцессами	350
13.1	Создание подпроцесса	351
13.1.1	Управление стандартным выводом	353
13.1.2	Конкурентное выполнение подпроцессов	357
13.2	Взаимодействие с подпроцессами	360
	Резюме	363

14	Продвинутое использование <i>asyncio</i>	365
14.1	API, допускающие сопрограммы и функции	366
14.2	Контекстные переменные	368
14.3	Принудительный запуск итерации цикла событий	370
14.4	Использование других реализаций цикла событий	371
14.5	Создание собственного цикла событий	373
14.5.1	Сопрограммы и генераторы	373
14.5.2	Использовать сопрограммы на основе генераторов не рекомендуется	374
14.5.3	Нестандартные объекты, допускающие ожидание	376
14.5.4	Сокеты и будущие объекты	378
14.5.5	Реализация задачи	381
14.5.6	Реализация цикла событий	382
14.5.7	Реализация сервера с использованием своего цикла событий	385
	Резюме	387

	Предметный указатель	388
--	----------------------------	-----

Предисловие

Почти 20 лет назад я начал профессиональную карьеру в области программной инженерии, написав приложение для управления масс-спектрометрами и другими лабораторными приборами и анализа поступающих от них данных. В приложении использовались языки Matlab, C++ и VB.net. Меня всегда охватывал восторг при виде того, как строчка кода заставляет машину двигаться по моему желанию, и с тех пор я понял, что хочу заниматься только программной инженерией. Шли годы, мои интересы постепенно сместились в сторону разработки API и распределенных систем, в основном на Java и Scala, но попутно я активно изучал Python.

Я впервые столкнулся с Python примерно в 2015 году, работа была связана главным образом с построением конвейера машинного обучения, который принимал данные от датчиков и на их основе делал предсказания относительно действий носителя датчика – например, отслеживание сна, подсчет числа шагов, переходы из положения сидя в положение стоя и т. д. Тогда этот конвейер был медленным настолько, что это стало проблемой для клиентов. Одним из способов, которые я применил для решения проблемы, стало использование конкурентности. Копаясь в доступной информации о конкурентном программировании на Python, я обнаружил, что разобраться в ней труднее, чем в привычном мне мире Java. Почему многопоточность работает не так, как в Java? Есть ли смысл использовать многопроцессную обработку? Что такое глобальная блокировка интерпретатора и зачем она нужна? На тему конкурентности в Python книг было немного, а знания в основном были разбросаны по документации и блогам разного качества. И сегодня ситуация мало чем отличается. Ресурсов стало больше, но ландшафт по-прежнему скудный, разъединенный и не такой дружелюбный к начинающим, каким должен, по идее, быть.

Разумеется, за последние несколько лет многое изменилось. Тогда пакет `asyncio` пребывал в младенчестве, а теперь стал важным модулем в Python. Ныне модели однопоточной конкурентности и сопрограммы являются одним из базовых компонентов Python в дополнение к многопоточности и многопроцессности. Это значит, что ландшафт конкурентности в Python стал шире и сложнее, но так и не обрел исчерпывающих ресурсов, к которым мог бы обратиться желающий изучить его.

Я написал эту книгу, чтобы заполнить этот пробел, конкретно в области `asyncio` и однопоточной конкурентности. Я хотел сделать сложную и плохо документированную тему однопоточной конкурентности более доступной разработчикам всех уровней. Кроме того, я хотел написать книгу, благодаря которой читатель стал бы лучше понимать общие проблемы конкурентности, выходящие за рамки Python. В таких каркасах, как Node.js, и таких языках, как Kotlin, имеются модели однопоточной конкурентности и сопрограммы, поэтому приведенные здесь сведения будут полезны и при работе с этими инструментами. Надеюсь, что книга окажется полезной читателям-разработчикам в повседневной работе, – и не только в Python, но и вообще в области конкурентного программирования.

Благодарности

Прежде всего хочу сказать спасибо своей жене, которая всегда была готова прийти на помощь в качестве корректора, когда я не был уверен, что делаю нечто разумное, и вообще всячески поддерживала меня на протяжении всего процесса. Второе спасибо – моему псу, Дагу, который всегда с готовностью ронял перед мной свой мячик, напоминая, что пора сделать перерыв и поиграть.

Затем я благодарю своего редактора, Дуга Раддера, и технического рецензента, Роберта Веннера. Ваши замечания здорово помогли мне написать книгу в срок и с высоким качеством, благодаря им код и пояснения получились осмысленными и понятными.

Спасибо всем рецензентам: Алексею Выскубову, Энди Майлсу, Чарльзу М. Шелтону, Крису Вайнеру, Кристоферу Коттмайеру, Клиффорду Тэрберу, Дэну Шейху, Дэвиду Кабреро, Дидье Гарсия, Димитриосу Кузис-Лукасу, Эли Майосту, Гэри Бэйку, Гонзало Габриэлю Хименесу Фуэнтесу, Грегори А. Люссьеру, Джеймсу Лю, Джереми Чену, Кенту Р. Спиллнеру, Дакшми Нарайянану Нарасимхану, Леонардо Таккари, Матиасу Бушу, Павлу Филатову, Филиппу Соренсену, Ричарду Вогану, Сандживу Киларапу, Симеону Лейзерзону, Симону Шёке, Симоне Сгуацца, Самиту К. Сингху, Вайрону Дадала, Уильяму Джамиру Силва и Зохебу Айнапору. Ваши предложения помогли сделать книгу лучше.

Наконец, я признателен бесчисленным учителям, коллегам и наставникам, с которыми общался на протяжении своей жизни. Я так много узнал от вас. Ваш совокупный опыт снабдил меня инструментами, позволившими издать эту работу и вообще добиться успеха в карьере. Без вас я не стал бы тем, кем стал. Спасибо вам!

Об этой книге

Эта книга написана для тех, кто хочет научиться использовать средства конкурентности в Python, чтобы повысить производительность, пропускную способность и отзывчивость приложений. Сначала мы рассмотрим базовые вопросы конкурентности, объясним, как работает модель однопоточной конкурентности в `asyncio`, а также расскажем о принципах работы сопрограмм и синтаксисе `async/await`. Затем перейдем к практическим приложениям конкурентности, например: как отправить несколько конкурентных веб-запросов или запросов к базе данных, как управлять потоками и процессами, строить веб-приложения и решать вопросы синхронизации.

Кому стоит прочитать эту книгу?

Книга адресована разработчикам средней и высокой квалификации, которые хотят разобраться в конкурентности и использовать ее в уже написанных или новых приложениях. Одна из целей книги – объяснить сложные вопросы простым и понятным языком. Предварительного знакомства с конкурентностью не требуется, хотя оно, конечно, не помешает. Мы рассмотрим широкий круг применений: от API на основе веба до командных приложений, так что книга будет полезна для решения многих задач, с которыми сталкиваются разработчики.

Структура книги

В этой книге 14 глав, в которых рассматриваются постепенно усложняющиеся темы. Последующие главы основаны на материале предыдущих.

- **Глава 1** посвящена базовым вопросам конкурентности в Python. Мы узнаем о задачах, ограниченных быстродействием процессора (счетных) и производительностью ввода-вывода, а также

начнем рассказ о том, как работает модель однопоточной конкурентности.

- **Глава 2** посвящена основам сопрограмм `asyncio` и использованию синтаксиса `async/await` в конкурентных приложениях.
- В **главе 3** рассматриваются неблокирующие сокеты и селекторы, а также описывается построение эхо-сервера с применением `asyncio`.
- **Глава 4** посвящена отправке нескольких конкурентных веб-запросов. Попутно мы больше узнаем о том, как использовать `asyncio API` для конкурентного выполнения сопрограмм.
- Тема **главы 5** – конкурентное выполнение запросов к базе данных с использованием пула подключений. Также мы узнаем об асинхронных контекстных менеджерах и асинхронных генераторах в контексте баз данных.
- В **главе 6** обсуждается многопроцессная обработка, в частности использование `asyncio` для выполнения счетных задач. Для демонстрации будет построено сообщение типа `map-reduce`.
- **Глава 7** посвящена многопоточной обработке, а особенно ее использованию в сочетании с `asyncio` для обработки блокирующего ввода-вывода. Это полезно при работе с библиотеками, в которые поддержка `asyncio` не встроена, что не мешает им получать выгоды от конкурентного выполнения.
- **Глава 8** посвящена потоковой обработке и протоколам. Мы напишем сервер и клиент чата, способные конкурентно обрабатывать несколько пользователей.
- В **главе 9** рассматриваются веб-приложения на основе `asyncio` и асинхронный интерфейс серверного шлюза, `ASGI`. Мы изучим несколько каркасов, поддерживающих `ASGI`, и обсудим, как строить в них веб-API. Также поговорим о технологии `WebSockets`.
- В **главе 10** описывается, как с помощью веб-API на основе `asyncio` построить гипотетическую микросервисную архитектуру.
- **Глава 11** посвящена проблемам синхронизации в модели однопоточной конкурентности и методам их решения. Мы рассмотрим блокировки, семафоры, события и условия.
- **Глава 12** посвящена асинхронным очередям. С их помощью мы построим веб-приложение, мгновенно отвечающее на клиентские запросы, хотя в фоновом режиме производятся длительные операции.
- В **главе 13** обсуждается создание подпроцессов и управление ими. Мы покажем, как читать и передавать данные подпроцессу.
- В **главе 14** рассматриваются дополнительные темы, в том числе принудительный переход к новой итерации цикла событий, контекстные переменные и создание собственного цикла событий. Эта информация полезна прежде всего проектировщикам `asyncio API` и читателям, интересующимся внутренним устройством цикла событий в `asyncio`.

Как минимум следует прочитать первые четыре главы, чтобы понять, как работает `asyncio`, как написать свое первое реальное приложение и как использовать базовые функции `asyncio` API для конкурентного выполнения сопрограмм (рассматриваются в главе 4). После этого вы вольны читать книгу в любом порядке, отвечающем вашим интересам.

О примерах кода

В этой книге много примеров кода как в пронумерованных листингах, так и в виде небольших фрагментов. Некоторые листинги используются повторно путем импорта в той же главе, а иногда и в нескольких главах. Предполагается, что код, используемый в нескольких главах, помещен в модуль `util`; мы создадим его в главе 2. Предполагается также, что для каждого отдельного листинга будет создан модуль с именем `chapter_{номер_главы}`, а код будет помещен в файл с именем `listing_{номер_главы}_{номер_листинга}.py`, принадлежащий этому модулю. Например, код из листинга 2.2 в главе 2 должен находиться в файле `listing_2_2.py`, принадлежащем модулю `chapter_2`.

В нескольких местах приводятся сведения о производительности, например время работы программы или количество веб-запросов в секунду. Примеры кода выполнялись на компьютере 2019 MacBook Pro с 8-ядерным процессором Intel Core i9 с тактовой частотой 2,4 ГГц и 32 Гб памяти DDR4, работающей на частоте 2667 МГц; использовалось также гигабитное беспроводное подключение к интернету. На вашей машине цифры могут получиться другими, как и коэффициенты ускорения или иного улучшения производительности.

Исполняемые фрагменты кода можно найти в онлайн-версии книги на сайте <https://livebook.manning.com/book/python-concurrency-with-asyncio>. Полный исходный код можно скачать бесплатно с сайта издательства Manning по адресу <https://www.manning.com/books/python-concurrency-with-asyncio>, также он доступен на Github по адресу <https://github.com/concurrency-in-python-with-asyncio>.

Форум на сайте liveBook

Приобретение этой книги открывает бесплатный доступ к платформе liveBook онлайн-чтения, созданной издательством Manning. Средства обсуждения на liveBook позволяют присоединять комментарии как к книге в целом, так и к отдельным разделам или абзацам. Совсем несложно добавить примечания для себя, задать или ответить на технический вопрос и получить помощь от автора и других пользователей. Для доступа к форуму перейдите по адресу <https://livebook.manning.com/#!/book/python-concurrency-with-asyncio/discussion>. Узнать о форумах Manning и правилах поведения на них можно по адресу <https://livebook.manning.com/#!/discussion>.

Издательство Manning обязуется предоставлять площадку для содержательного диалога между читателями, а также между читателями

и автором. Но это обязательство не подразумевает какого-то конкретного количества часов присутствия автора, участие которого в работе форума остается добровольным (и не оплачивается). Мы рекомендуем задавать автору трудные вопросы, чтобы его интерес не угасал! Форум и архивы прошлых обсуждений остаются доступны на сайте издательства до тех пор, пока книга продолжает допечатываться.

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 на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе



Мэттью Фаулер почти 20 лет занимается программной инженерией на различных должностях: от архитектора ПО до директора по инженерии. Он начинал с написания научных программ, затем перешел на сквозную разработку веб-приложений и распределенных систем и в итоге возглавил несколько команд разработчиков и менеджеров, которые занимались созданием интернет-магазина, обслуживающего десятки миллионов пользователей. Проживает в Лексингтоне, штат Массачусетс, с женой Кэти.

Об иллюстрации на обложке

На обложке книги изображена крестьянка из маркизата Баден. Рисунок взят из книги Жака Грассе де Сен-Совера, изданной в 1797 году. Все иллюстрации мастерски нарисованы и раскрашены вручную. В те времена легко было по одежде определить, где человек живет, чем занимается и каков его статус. Издательство Manning откликается на новации и инициативы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов быта в прошлых веках. Мы возвращаем его в том виде, в каком оно запечатлено на рисунках из таких собраний, как это.

Первое знакомство с *asuncio*

Краткое содержание главы

- Что такое библиотека *asuncio* и какие преимущества она дает.
- Конкурентность, параллелизм, потоки и процессы.
- Глобальная блокировка интерпретатора и создаваемые ей проблемы для конкурентности.
- Как неблокирующие сокеты позволяют добиться конкурентного выполнения в одном потоке.
- Начала конкурентности на основе цикла событий.

Многие программы, особенно в нынешнем мире, где правят бал веб-приложения, сильно зависят от операций ввода-вывода. Это скачивание веб-страницы из интернета, взаимодействие по сети с группой микросервисов или отправка нескольких запросов такой базе данных, как MySQL или Postgres. Выполнение веб-запроса или взаимодействие с микросервисом может занять несколько сотен миллисекунд или даже секунды, если сеть медленная. Запрос к базе данных может занимать много времени, особенно если нагрузка на базу высока или запрос сложный. Веб-серверу иногда приходится обрабатывать одновременно сотни или тысячи запросов.

Попытка выполнить эти запросы ввода-вывода сразу может привести к серьезным проблемам с производительностью. Если отправлять запросы один за другим, как в последовательном приложении, то за-

медление будет нарастать. Например, если мы пишем приложение, которое должно скачать 100 веб-страниц или выполнить 100 запросов к базе данных, и каждое взаимодействие продолжается 1 с, то для завершения приложения потребуется 100 с. Но если воспользоваться конкурентностью и начать все скачивания в одно и то же время, после чего дожидаться результатов, то теоретически все операции можно было бы завершить за 1 с.

Библиотека *asyncio* впервые появилась в версии Python 3.4 как еще один способ справляться с высокими конкурентными нагрузками, не прибегая к нескольким потокам или процессам. При правильном использовании эта библиотека может значительно повысить производительность и уменьшить потребление ресурсов в приложениях, выполняющих много операций ввода-вывода, поскольку позволяет запускать сразу много таких долго работающих задач.

В этой главе мы познакомимся с основами конкурентности, чтобы лучше понять, как она достигается в Python и библиотеке *asyncio*. Мы рассмотрим различия между задачами, ограниченными быстродействием процессора и производительностью ввода-вывода, чтобы вы могли понять, какая модель конкурентности лучше отвечает вашим потребностям. Мы также поговорим об основах процессов и потоков и о специфических проблемах, связанных с наличием в Python глобальной блокировки интерпретатора (GIL). Наконец, мы поймем, как использовать *неблокирующий ввод-вывод* совместно с циклом событий и таким образом добиться конкурентности всего в одном процессе и потоке. Это основная модель конкурентности в *asyncio*.

1.1 Что такое *asyncio*?

В синхронном приложении код выполняется последовательно. Следующая строка кода выполняется после завершения предыдущей, и в каждый момент времени происходит что-то одно. Эта модель хорошо работает для многих, если не для большинства приложений. Но что, если какая-то одна строка кода занимает слишком много времени? В таком случае весь последующий код должен будет замереть, пока эта строка не соблаговолит завершиться. Многие из нас раньше встречали плохо написанные пользовательские интерфейсы, в которых все поначалу шло хорошо, а потом приложение внезапно зависало, оставляя нас созерцать крутящееся колесико или ждать хоть какого-нибудь ответа. Такие блокировки в приложении оставляют тягостное впечатление у пользователя.

Любая достаточно длительная операция может блокировать приложение, но особенно часто это бывает, когда приложение ждет завершения ввода-вывода. Ввод-вывод выполняют такие устройства, как клавиатура, жесткий диск и, конечно же, сетевая карта. Такие операции ждут ввода от пользователя или получения содержимого

от веб-API. В синхронном приложении мы будем ждать завершения операции и до тех пор ничего не сможем делать. Это ведет к проблемам с производительностью и отзывчивостью, поскольку в каждый момент времени может выполняться только одна длительная операция, а она не дает приложению больше ничего делать.

Один из способов решения этой проблемы – ввести в программу конкурентность. Говоря по-простому, *конкурентность* позволяет одновременно выполнять более одной задачи. Примерами конкурентного ввода-вывода могут служить одновременная отправка нескольких веб-запросов или создание одновременных подключений к веб-серверу.

В Python есть несколько способов организовать такую конкурентность. Одним из последних добавлений в экосистему Python является библиотека асинхронного ввода-вывода *asyncio*. Она позволяет исполнять код в рамках модели асинхронного программирования, т. е. производить сразу несколько операций ввода-вывода, не жертвуя отзывчивостью приложения.

Так что же означают слова «асинхронное программирование»? Что длительную задачу можно выполнять в фоновом режиме отдельно от главного приложения. И система не блокируется в ожидании завершения этой задачи, а может заниматься другими вещами, не зависящими от ее исхода. Затем, по завершении задачи, мы получим уведомление о том, что она все сделала, и сможем обработать результат.

В Python версии 3.4, когда состоялся дебют библиотеки *asyncio*, она включала декораторы и синтаксис генератора `yield from` для определения сопрограмм. Сопрограмма – это метод, который можно приостановить, если имеется потенциально длительная задача, а затем возобновить, когда она завершится. В Python 3.5 в самом языке была реализована полноценная поддержка сопрограмм и асинхронного программирования, для чего были добавлены ключевые слова `async` и `await`. Этот синтаксис, общий с другими языками программирования, например C и JavaScript, позволяет писать асинхронный код так, что он выглядит как синхронный. Такой асинхронный код проще читать и понимать, поскольку он похож на последовательный код, с которым большинство программистов хорошо знакомо. Библиотека *asyncio* исполняет сопрограммы асинхронно, пользуясь моделью конкурентности, получившей название *однопоточный цикл событий*.

Название *asyncio* может навести на мысль, будто библиотека годится только для операций ввода-вывода. Но на самом деле она способна выполнять и операции других типов благодаря взаимодействию с механизмами многопроцессности и многопоточности. Поэтому синтаксис `async` и `await` можно использовать в сочетании с процессами и потоками, что делает соответствующий код понятнее. Следовательно, библиотека пригодна не только для организации конкурентного ввода-вывода, но и для выполнения счетных задач, активно использующих процессор. Чтобы лучше разобраться в том, с какими рабочими нагрузками позволяет справляться *asyncio* и какая модель

лучше всего подходит для каждого типа конкурентности, рассмотрим различия между операциями, ограниченными производительностью ввода-вывода и быстродействием процессора.

1.2 Что такое ограниченность производительностью ввода-вывода и ограниченность быстродействием процессора

Говоря, что операция ограничена производительностью ввода-вывода или быстродействием процессора, мы имеем в виду фактор, препятствующий более быстрой работе. Это значит, что если увеличить производительность того, что ограничивает операцию, то для ее завершения понадобится меньше времени.

Операция, ограниченная быстродействием процессора (счетная операция), завершилась бы быстрее, будь процессор более мощным, например с частотой 3, а не 2 ГГц. Операция, ограниченная производительностью ввода-вывода, работала бы быстрее, если бы устройство могло обработать больше данных за меньшее время. Для этого можно было бы увеличить пропускную способность сети, заплатив больше денег интернет-провайдеру или поставив более шустрю сетевую карту.

Счетные операции в Python – это обычно вычисления и обработка данных. Примерами могут служить вычисления цифр числа π или применение бизнес-логики к каждому элементу словаря. В случае операции, ограниченной вводом-выводом, мы тратим большую часть времени на ожидание ответа от сети или другого устройства. Примерами могут служить запрос к веб-серверу или чтение файла с жесткого диска.

Листинг 1.1 Операции, ограниченные производительностью ввода-вывода и быстродействием процессора

```
import requests
response = requests.get('https://www.example.com')
items = response.headers.items()
headers = [f'{key}: {header}' for key, header in items]
formatted_headers = '\n'.join(headers)
with open('headers.txt', 'w') as file:
    file.write(formatted_headers)
```

Веб-запрос ограничен
производительностью ввода-вывода

Обработка ответа
ограничена
быстродействием
процессора

Конкатенация строк ограничена
быстродействием процессора

Запись на диск ограничена производительностью ввода-вывода

Операции, ограниченные производительностью ввода-вывода и быстродействием процессора, обычно сосуществуют бок о бок. Сначала мы выполняем ограниченный производительностью ввода-вывода запрос, чтобы скачать содержимое страницы <https://www.example.com>. Получив ответ, мы выполняем ограниченный быстродействием процессора цикл форматирования заголовков ответа, в котором они преобразуются в строку и разделяются символами новой строки. Затем мы открываем файл и записываем в него строку – обе операции ограничены производительностью ввода-вывода.

Асинхронный ввод-вывод позволяет приостановить выполнение метода, встретив операцию ввода-вывода; ожидая завершения этой операции, работающей в фоновом режиме, мы можем выполнять какой-нибудь другой код. Это позволяет выполнять одновременно много операций ввода-вывода и тем самым ускорить работу приложения.

1.3 Конкурентность, параллелизм и многозадачность

Чтобы лучше понять, как конкурентность может повысить производительность приложения, важно для начала разобраться в терминологии конкурентного программирования. Мы узнаем, что означает слово «конкурентность» и как в `asynсio` используется концепция многозадачности для ее достижения. Конкурентность и параллелизм – два понятия, помогающие понять, как осуществляется планирование программы и какие задачи, методы и процедуры приводят весь механизм в действие.

1.3.1 Конкурентность

Говоря, что две задачи выполняются *конкурентно*, мы имеем в виду, что они работают в одно и то же время. Взять, к примеру, пекаря, выпекающего два разных торта. Чтобы их испечь, нужно сначала разогреть духовку. Разогрев может занимать десятки минут – это зависит от духовки и требуемой температуры, но нам необязательно ждать, пока духовка нагреется, поскольку можно в это время заняться другими делами, например смешать муку с сахаром и вбить в тесто яйца. Это можно делать, пока духовка звуковым сигналом не известит нас о том, что нагрелась.

Мы также не хотим связывать себя ограничением – приступать ко второму тарту только после готовности первого. Ничто не мешает замесить тесто для одного торта, положить его в миксер и начать готовить вторую порцию, пока первая взбивается. В этой модели мы переключаемся между разными задачами конкурентно. Такое переключение (делать что-то, пока духовка разогревается, переключаться с одного торта на другой) – пример *конкурентного* поведения.

1.3.2 Параллелизм

Хотя конкурентность подразумевает, что несколько задач выполняется одновременно, это еще не значит, что они работают параллельно. Говоря о *параллельной* работе, мы имеем в виду, что две задачи или более не просто чередуются, а выполняются строго в одно и то же время. Возвращаясь к примеру с тортами, представьте, что мы призвали на помощь второго пекаря. В этом случае мы можем трудиться над первым тортом, а помощник будет заниматься вторым. Два человека, готовящих тесто, работают параллельно, потому что одновременно делают два разных дела (рис. 1.1).

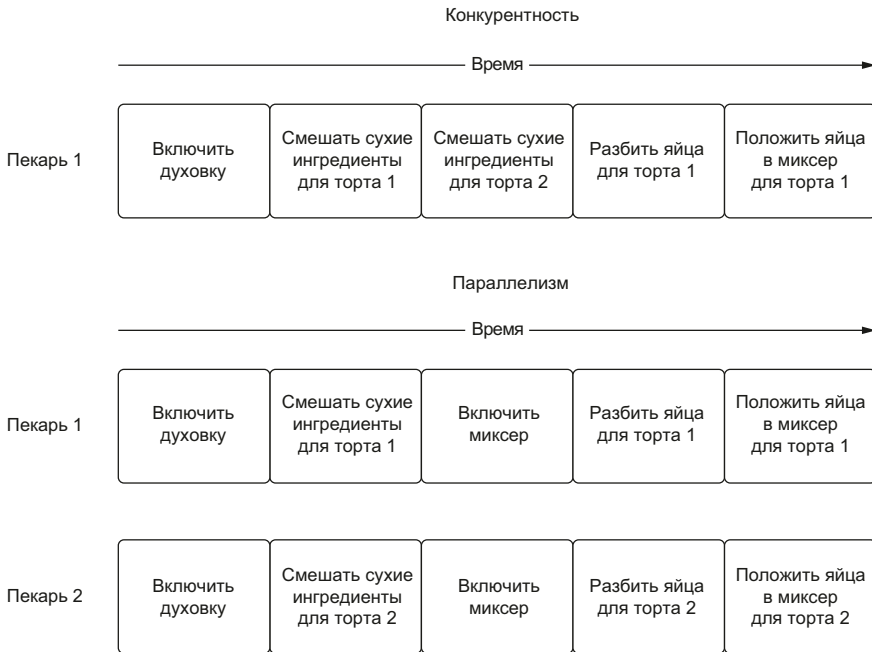


Рис. 1.1 В случае *конкурентности* несколько задач работает в течение одного промежутка времени, но только одна активна в каждый момент. В случае *параллелизма* несколько задач активно одновременно

Теперь переведем это на язык приложений, исполняемых операционной системой. Пусть работают два приложения. В конкурентной системе мы можем переключаться между ними, дав немного поработать сначала одному, а потом другому. Если делать это достаточно быстро, создается впечатление, что два дела делаются одновременно. В параллельной системе два приложения работают действительно одновременно, т. е. оба активны в одно и то же время.

Понятия конкурентности и параллелизма похожи (рис. 1.2), и различить их бывает затруднительно, но важно хорошо понимать, чем они отличаются.

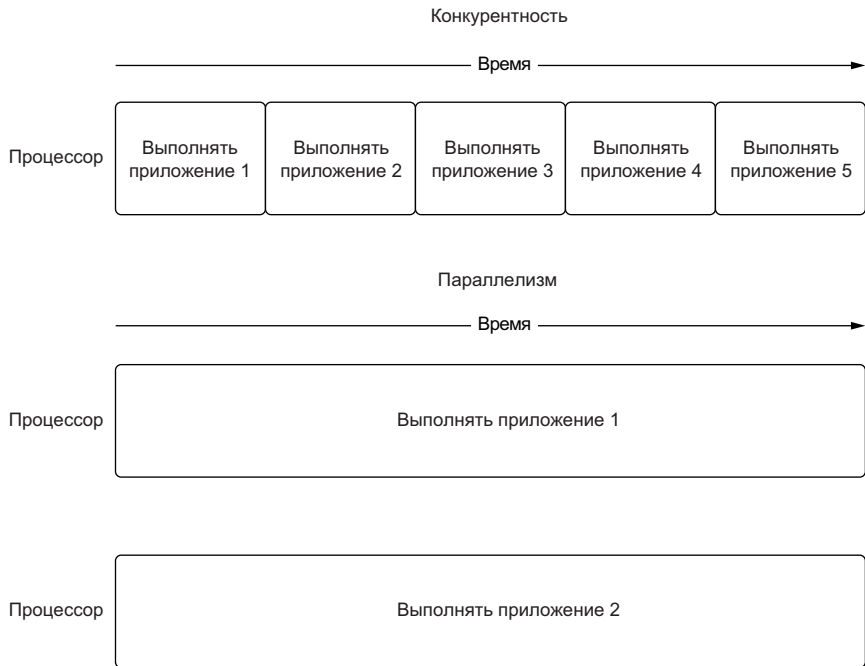


Рис. 1.2 В случае конкурентности мы переключаемся между двумя приложениями. В случае параллелизма мы активно выполняем два приложения одновременно

1.3.3 Различие между конкурентностью и параллелизмом

Конкурентность возможна, когда несколько задач может работать независимо друг от друга. Конкурентность можно организовать, имея процессор всего с одним ядром, применив *вытесняющую многозадачность* (определяется в следующем разделе) для переключения между задачами. С другой стороны, параллелизм означает, что мы должны выполнять две задачи или более строго одновременно. На машине с одним ядром это невозможно, необходимо иметь процессор с несколькими ядрами.

Параллелизм подразумевает конкурентность, но обратное верно не всегда. Многопоточное приложение, работающее на многоядерной машине, является и конкурентным, и параллельным. В этом случае имеется несколько задач, работающих одновременно, и два ядра, независимо исполняющих код этих задач. Но многозадачность допускает также наличие нескольких задач, работающих конкурентно, но так, что в каждый момент времени выполняется только одна.

1.3.4 Что такое многозадачность

В современном мире многозадачность встречается повсеместно. Мы готовим завтрак, а пока в чайнике закипает вода, кому-то звоним или отвечаем на текстовое сообщение. А пока электричка везет нас на работу, читаем книгу. В этом разделе мы обсудим два основных вида многозадачности: *вытесняющую* и *невытесняющую*, или *кооперативную*.

ВЫТЕСНЯЮЩАЯ МНОГОЗАДАЧНОСТЬ

В этой модели мы позволяем операционной системе решить, как переключаться между выполняемыми задачами с помощью процедуры *квантования времени*. Когда операционная система переключает задачи, мы говорим, что имеет место *вытеснение*.

Как устроен этот механизм, зависит от операционной системы. Обычно для этого используется либо несколько потоков, либо несколько процессов.

КООПЕРАТИВНАЯ МНОГОЗАДАЧНОСТЬ

В этой модели мы не полагаемся для переключения между задачами на операционную систему, а явно определяем в коде приложения точки, где можно уступить управление другой задаче. Исполняемые задачи *кооперируются*, т. е. говорят приложению: «Я сейчас на время приостановлюсь, а ты можешь пока выполнять другие задачи».

1.3.5 Преимущества кооперативной многозадачности

В *asynсio* для организации конкурентности используется кооперативная многозадачность. Когда приложение доходит до точки, в которой может подождать результата, мы явно помечаем это в коде. Поэтому другой код может работать, пока мы ждем получения результата, вычисляемого в фоновом режиме. Как только вычисление результата завершится, мы «просыпаемся» и возобновляем задачу. Это является формой конкурентности, потому что несколько задач может работать одновременно, но – и это очень важно – не параллельно, так как их выполнение чередуется.

У кооперативной многозадачности есть ряд преимуществ перед вытесняющей. Во-первых, кооперативная многозадачность потребляет меньше ресурсов. Когда операционной системе нужно переключиться между потоками или процессами, мы говорим, что имеет место *контекстное переключение*. Это трудоемкая операция, потому что операционная система должна сохранить всю информацию о работающем процессе или потоке, чтобы потом его можно было возобновить.

Второе преимущество – *гранулярность*. Операционная система приостанавливает поток или процесс в соответствии со своим алго-

ритмом планирования, но выбранный для этого момент не всегда оптимален. В случае кооперативной многозадачности мы явно помечаем точки, в которых приостановить задачу наиболее выгодно. Это дает выигрыш в эффективности, потому что мы переключаем задачи именно в тот момент, когда это нужно. А теперь, когда мы разобрались в понятиях конкурентности, параллелизма и многозадачности, посмотрим, как они реализуются в Python с помощью потоков и процессов.

1.4 Процессы, потоки, многопоточность и многопроцессность

Чтобы лучше понять, как работает конкурентность в Python, нужно сначала разобраться с потоками и процессами. Затем мы поговорим о том, как использовать их для организации конкурентной работы с помощью многопоточности и многозадачности. Начнем с определения процесса и потока.

1.4.1 Процесс

Процессом называется работающее приложение, которому выделена область памяти, недоступная другим приложениям. Пример создания Python-процесса – запуск простого приложения «hello world» или ввод слова `python` в командной строке для запуска цикла REPL (цикл чтения–вычисления–печати).

На одной машине может работать несколько процессов. Если машина оснащена процессором с несколькими ядрами, то несколько процессов могут работать одновременно. Если процессор имеет только одно ядро, то все равно можно выполнять несколько приложений конкурентно, но уже с применением квантования времени. При использовании квантования операционная система будет автоматически вытеснять работающий процесс по истечении некоторого промежутка времени и передавать процессор другому процессу. Алгоритмы, определяющие, в какой момент переключать процессы, зависят от операционной системы.

1.4.2 Поток

Потоки можно представлять себе как облегченные процессы. Кроме того, это наименьшая единица выполнения, которая может управляться операционной системой. У потоков нет своей памяти, они пользуются памятью создавшего их процесса. Потоки ассоциированы с процессом, создавшим их. С каждым процессом всегда ассоциирован по меньшей мере один поток, обычно называемый *главным*. Процесс может создавать дополнительные потоки, которые обычно

называются *рабочими* или *фоновыми*. Эти потоки могут конкурентно выполнять другую работу наряду с главным потоком. Потоки, как и процессы, могут работать параллельно на многоядерном процессе, и операционная система может переключаться между ними с помощью квантования времени. Обычное Python-приложение создает процесс и главный поток, который отвечает за его выполнение.

Листинг 1.2 Процессы и потоки в простом Python-приложении

```
import os
import threading

print(f'Исполняется Python-процесс с идентификатором: {os.getpid()}')

total_threads = threading.active_count()
thread_name = threading.current_thread().name

print(f'В данный момент Python исполняет {total_threads} поток(ов)')
print(f'Имя текущего потока {thread_name}')
```

На рис. 1.3 схематически изображен процесс, соответствующий листингу 1.2. Мы написали простое приложение, демонстрирующее работу главного потока. Сначала мы получаем уникальный идентификатор процесса и печатаем его, чтобы доказать, что действительно имеется работающий процесс. Затем получаем счетчик активных потоков и имя текущего потока, чтобы доказать, что действительно работает один поток – главный. При каждом выполнении этой программы идентификатор процесса будет разным, но всегда выводятся строки вида:

```
Исполняется Python-процесс с идентификатором: 98230
В данный момент Python исполняет 1 поток(ов)
Имя текущего потока MainThread
```

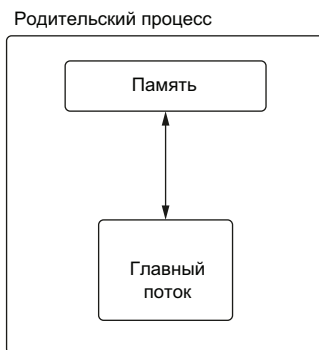


Рис. 1.3 Процесс с одним главным потоком, читающим данные из памяти

Процессы могут порождать дополнительные потоки, разделяющие память со своим процессом-родителем. Они могут конкурентно выполнять другую работу, это называется *многопоточностью*.

Листинг 1.3 Создание многопоточного Python-приложения

```
import threading

def hello_from_thread():
    print(f'Привет от потока {threading.current_thread()}!')

hello_thread = threading.Thread(target=hello_from_thread)
hello_thread.start()

total_threads = threading.active_count()
thread_name = threading.current_thread().name

print(f'В данный момент Python выполняет {total_threads} поток(ов)')
print(f'Имя текущего потока {thread_name}')
hello_thread.join()
```

На рис. 1.4 схематически изображен процесс и потоки, соответствующие листингу 1.3. Мы написали метод, печатающий имя текущего потока, а затем создали поток, исполняющий этот метод. После этого вызывается метод потока `start`, запускающий поток. А в конце вызывается метод `join`, который приостанавливает программу, до тех пор пока указанный поток не завершится. Этот код печатает такие сообщения:

```
Привет от потока <Thread(Thread-1, started 123145541312512)>!
В данный момент Python выполняет 2 поток(ов)
Имя текущего потока MainThread
```

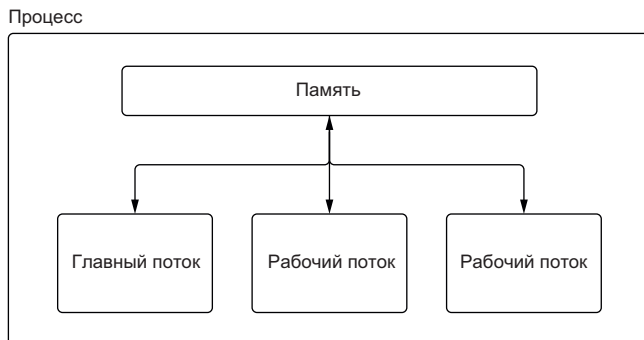


Рис. 1.4 Многопоточная программа с двумя рабочими и одним главным потоком. Все они разделяют общую память с процессом

Отметим, что при выполнении этой программы сообщения «Привет от потока» и «В данный момент Python выполняет 2 поток(ов)» могут быть напечатаны на одной строке. Это *состояние гонки*, мы будем говорить о нем в следующем разделе и подробнее в главах 6 и 7.

Многопоточные приложения – обычный способ реализации конкурентности во многих языках программирования. Но в Python есть несколько препятствий на пути организации конкурентности с по-

мощью потоков. Многопоточность полезна только для задач, ограниченных производительностью ввода-вывода, потому что нам мешает глобальная блокировка интерпретатора, о которой пойдет речь в главе 1.5.

Многопоточность не единственный способ добиться конкурентности. Можно вместо потоков создать несколько конкурентных процессов, это называется *многопроцессностью*. В таком случае родительский процесс создает один или более дочерних процессов, которыми управляет, а затем распределяет между ними работу.

В Python для этой цели имеется модуль `multiprocessing`. Его API похож на API модуля `threading`. Сначала создается процесс, при этом передается функция `target`. Затем вызывается метод `start`, чтобы начать выполнение процесса, и в конце – метод `join`, чтобы дождаться его завершения.

Листинг 1.4 Создание нескольких процессов

```
import multiprocessing
import os

def hello_from_process():
    print(f'Привет от дочернего процесса {os.getpid()}!')

if __name__ == '__main__':
    hello_process = multiprocessing.Process(target=hello_from_process)
    hello_process.start()

    print(f'Привет от родительского процесса {os.getpid()}')

    hello_process.join()
```

На рис. 1.5 схематически показан процесс и потоки для листинга 1.4. Мы создаем один дочерний процесс, который печатает свой идентификатор, а также печатаем идентификатор родительского процесса, чтобы доказать, что работают два разных процесса. Многопроцессность обычно предпочтительна, когда имеется счетная задача.

Многопоточность и многопроцессность могут показаться волшебной палочкой, обеспечивающей конкурентность в Python. Однако развернуться во всю мощь этим моделям мешает одна деталь реализации Python – глобальная блокировка интерпретатора.

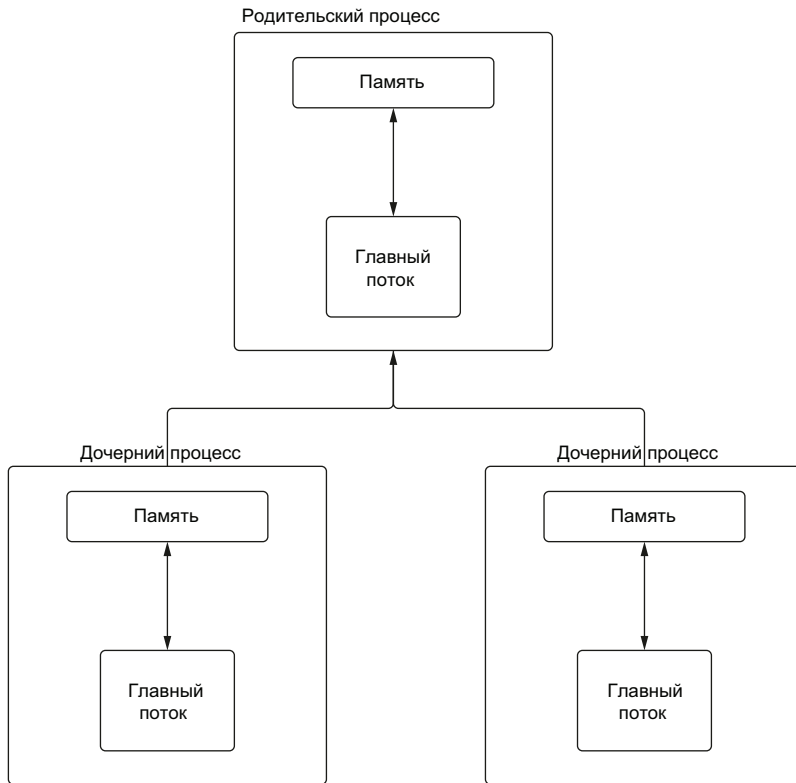


Рис. 1.5 Приложение, в котором имеется один родительский процесс и два дочерних

1.5 Глобальная блокировка интерпретатора

Глобальная блокировка интерпретатора (*global interpreter lock* – GIL) – тема, вызывающая споры в сообществе Python. Говоря кратко, GIL не дает Python-процессу исполнять более одной команды байт-кода в каждый момент времени. Это означает, что, даже если имеется несколько потоков на многоядерной машине, интерпретатор сможет в каждый момент исполнять только один поток, содержащий написанный на Python код. В мире, где процессоры имеют несколько ядер, это создает серьезную проблему для разработчиков на Python, желающих воспользоваться многопоточностью для повышения производительности приложений.

ПРИМЕЧАНИЕ Многопроцессные приложения могут конкурентно выполнять несколько команд байт-кода, потому что у каждого Python-процесса своя собственная GIL.

Так для чего же нужна GIL? Ответ кроется в том, как CPython управляет памятью. В CPython память управляется в основном с помощью подсчета ссылок. То есть для каждого объекта Python, например целого числа, словаря или списка, подсчитывается, сколько объектов в данный момент используют его. Когда объект перестает быть нужным кому-то, счетчик ссылок на него уменьшается, а когда кто-то новый обращается к нему, счетчик ссылок увеличивается. Если счетчик ссылок обратился в нуль, значит, на объект никто не ссылается, поэтому его можно удалить из памяти.

Что такое CPython?

CPython – это *эталонная реализация* Python, т. е. стандартная реализация языка, которая используется как *эталон* правильного поведения. Существуют и другие реализации, например Jython, работающая под управлением виртуальной машины Java, или IronPython, предназначенная для .NET Framework.

Конфликт потоков возникает из-за того, что интерпретатор CPython не является *потокобезопасным*. Это означает, что если два или более потоков модифицируют разделяемую переменную, то ее конечное состояние может оказаться неожиданным, поскольку зависит от порядка доступа к переменной со стороны потоков. Эта ситуация называется *состоянием гонки*. Состояния гонки могут возникать, когда два потока одновременно обращаются к одному объекту Python.

Как показано на рис. 1.6, если два потока одновременно увеличивают счетчик ссылок, то может случиться, что счетчик обнулится, хотя объект еще используется. Результатом станет крах приложения при попытке прочитать данные из освобожденной памяти.

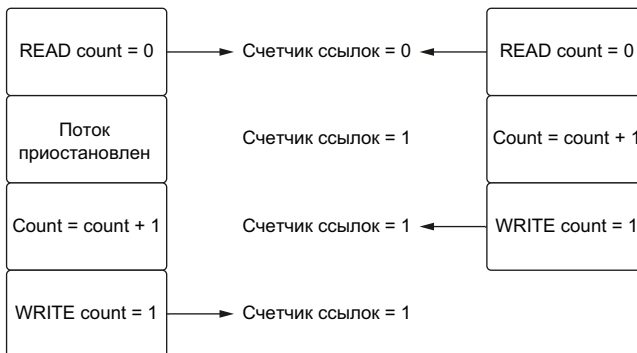


Рис. 1.6 Состояние гонки, возникающее, когда два потока одновременно пытаются увеличить счетчик ссылок. Вместо ожидаемого значения счетчика 2 мы получили значение 1

Для демонстрации влияния GIL на многопоточное программирование рассмотрим счетную задачу вычисления n -го числа Фибоначчи. Мы будем использовать довольно медленный алгоритм, чтобы продемонстрировать операции, занимающие много времени. В правильном решении для повышения производительности следовало бы использовать технику запоминания или другой математический метод.

Листинг 1.5 Генерирование последовательности Фибоначчи и его хронометраж

```
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

    print(f'fib({number}) равно {fib(number)}')
```

```
def fibs_no_threading():
    print_fib(40)
    print_fib(41)

start = time.time()

fibs_no_threading()

end = time.time()

print(f'Время работы {end - start:.4f} с.')
```

В этой реализации используется рекурсия, поэтому алгоритм получился медленным, он занимает время $O(2^N)$. Если нужно напечатать два числа Фибоначчи, то можно просто вычислить их синхронно и замерить время вычисления, как показано в листинге выше.

В зависимости от быстродействия процессора результаты хронометража могут быть различны, но результат будет примерно таким, как показано ниже.

```
fib(40) равно 63245986
fib(41) равно 102334155
Время работы 65.1516 с.
```

Вычисление долгое, но обращения к функции `print_fib` не зависят друг от друга. Это значит, что их можно поместить в разные потоки, которые теоретически могли бы исполняться разными процессорными ядрами, что ускорило бы работу приложения.

Листинг 1.6 Многопоточное вычисление последовательности чисел Фибоначчи

```
import threading
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

def fibs_with_threads():
    fortieth_thread = threading.Thread(target=print_fib, args=(40,))
    forty_first_thread = threading.Thread(target=print_fib, args=(41,))

    fortieth_thread.start()
    forty_first_thread.start()

    fortieth_thread.join()
    forty_first_thread.join()

start_threads = time.time()

fibs_with_threads()

end_threads = time.time()

print(f'Многопоточное вычисление заняло {end_threads - start_threads:.4f} с.')
```

Здесь мы создали два потока, один из которых вычисляет `fib(40)`, а другой `fib(41)`, и запустили их конкурентно, вызвав метод `start()` для каждого. Затем мы дважды вызвали метод `join()`, благодаря чему главная программа будет ждать завершения потоков. Учитывая, что вычисления `fib(40)` и `fib(41)` начались одновременно и выполняются конкурентно, можно было бы ожидать разумного ускорения работы, однако даже на многоядерной машине мы видим такой результат:

```
fib(40) равно 63245986
fib(41) равно 102334155
Многопоточное вычисление заняло 66.1059 с.
```

Многопоточная версия работала почти столько же времени. На самом деле даже чуть дольше! Это все из-за GIL и накладных расходов на создание и управление потоками. Да, потоки выполняются конкурентно, но в каждый момент времени только одному из них разрешено выполнять Python-код. А второй поток вынужден ждать завершения первого, что сводит на нет весь смысл нескольких потоков.

1.5.1 Освобождается ли когда-нибудь GIL?

Предыдущий пример поднимает вопрос, возможна ли вообще многопоточная конкурентность в Python, коль скоро GIL запрещает одновременное выполнение двух строк Python-кода? Но на наше счастье GIL не удерживается постоянно, что открывает возможность использования нескольких потоков.

Глобальная блокировка интерпретатора освобождается на время выполнения операций ввода-вывода. Это позволяет использовать потоки для конкурентного выполнения ввода-вывода, но не для выполнения счетного кода, написанного на Python (есть исключения, когда GIL все же освобождается на время выполнения счетных задач, и мы обсудим их в следующей главе). Для иллюстрации рассмотрим чтение кода состояния веб-страницы.

Листинг 1.7 Синхронное чтение кода состояния

```
import time
import requests

def read_example() -> None:
    response = requests.get('https://www.example.com')
    print(response.status_code)

sync_start = time.time()

read_example()
read_example()

sync_end = time.time()

print(f'Синхронное выполнение заняло{sync_end - sync_start:.4f} с.')
```

Здесь мы дважды скачиваем содержимое страницы *example.com* и печатаем код состояния. Результат зависит от скорости сетевого подключения и нашего местоположения, но в целом будет примерно таким:

```
200
200
Синхронное выполнение заняло 0.2306 с.
```

Теперь у нас есть эталон для сравнения – время работы синхронной версии, – и можно написать многопоточную версию. В ней мы создадим по одному потоку для каждого запроса к *example.com* и запустим их конкурентно.

```
import time
import threading
import requests
```

```
def read_example() -> None:
    response = requests.get('https://www.example.com')
    print(response.status_code)

thread_1 = threading.Thread(target=read_example)
thread_2 = threading.Thread(target=read_example)

thread_start = time.time()

thread_1.start()
thread_2.start()

print('Все потоки работают!')

thread_1.join()
thread_2.join()

thread_end = time.time()

print(f'Многопоточное выполнение заняло {thread_end - thread_start:.4f} с.')
```

В результате выполнения этой программы мы увидим следующие строки, где время, как и раньше, зависит от сетевого подключения и местоположения:

```
Все потоки работают!
200
200
Многопоточное выполнение заняло 0.0977 с.
```

Это примерно в два раза быстрее первоначальной версии, в которой потоки не использовались, поскольку теперь два запроса выполняются приблизительно в одно и то же время! Разумеется, результат зависит от скорости подключения к интернету и от характеристик машины, но порядок цифр именно такой.

Так почему же GIL освобождается при вводе-выводе, но не освобождается для счетных задач? Все дело в системных вызовах, которые выполняются за кулисами. В случае ввода-вывода низкоуровневые системные вызовы работают за пределами среды выполнения Python. Это позволяет освободить GIL, потому что код операционной системы не взаимодействует напрямую с объектами Python. GIL захватывается снова, только когда полученные данные переносятся в объект Python. Стало быть, на уровне ОС операции ввода-вывода выполняются конкурентно. Эта модель обеспечивает конкурентность, но не параллелизм. В других языках, например Java или C++, на многоядерных машинах можно организовать истинный параллелизм, потому что никакой GIL нет и код может выполняться строго одновременно. Но в Python лучшее, на что можно рассчитывать, – конкурентность операций ввода-вывода, поскольку в любой момент может выполняться только один кусок написанного на Python кода.

1.5.2 *Asynсio и GIL*

В `asynсio` используется тот факт, что операции ввода-вывода освобождают GIL, что позволяет реализовать конкурентность даже в одном потоке. При работе с `asynсio` мы создаем объекты *сопрограмм*. Сопрограмму можно представлять себе как облегченный поток. Точно так же, как может быть несколько потоков, работающих одновременно и исполняющих разные операции ввода-вывода, так может существовать много *сопрограмм*, работающих бок о бок. Ожидая завершения *сопрограмм*, занимающихся вводом-выводом, мы можем выполнять другой Python-код, получая таким образом конкурентность. Важно отметить, что `asynсio` не обходит GIL, мы по-прежнему ограничены ей. Если имеется счетная задача, то для ее конкурентного выполнения все равно нужно заводить отдельный процесс (и в `asynсio` есть для этого средства), иначе производительность снизится. Теперь, когда мы знаем, как организовать конкурентный ввод-вывод в одном потоке, посмотрим, как это работает на примере неблокирующих сокетов.

1.6 Как работает однопоточная конкурентность

В предыдущем разделе мы рассмотрели многопоточность как механизм конкурентного выполнения операций ввода-вывода. Однако для достижения такого вида конкурентности заводить несколько потоков ни к чему. Все это можно сделать в рамках одного процесса и одного потока. При этом мы воспользуемся тем фактом, что на уровне ОС операции ввода-вывода могут завершаться конкурентно. Чтобы лучше понять, как это возможно, рассмотрим подробнее, как работают сокеты и конкретно неблокирующие сокеты.

1.6.1 Что такое сокет?

Сокет – это низкоуровневая абстракция отправки и получения данных по сети. Именно с ее помощью производится обмен данными между клиентами и серверами. Сокеты поддерживают две основные операции: отправку и получение байтов. Мы записываем байты в сокет, затем они передаются по адресу назначения, чаще всего на какой-то сервер. Отправив байты, мы ждем, пока сервер пришлет ответ в наш сокет. Когда байты окажутся в сокете, мы сможем прочитать результат.

Низкоуровневую концепцию сокетов проще понять, если представлять их как почтовые ящики. Вы можете положить письмо в почтовый ящик, почтальон заберет его и доставит в почтовый ящик получателя. Получатель откроет его и достанет ваше письмо. В зависимости от содержания письма получатель может отправить письмо

в ответ. Здесь письмо является аналогом данных или байтов, которые мы хотим отправить. Можно рассматривать помещение письма в почтовый ящик как запись байтов в сокет, а извлечение его из ящика – как чтение байтов из сокета. Почтальон тогда является аналогом механизма передачи через интернет, который маршрутизирует данные до адреса назначения.

Если нужно получить содержимое страницы *example.com*, то мы открываем сокет, подключенный к серверу *example.com*. Затем записываем в сокет запрос и ждем ответа от сервера, в данном случае HTML-кода веб-страницы. На рис. 1.7 показан поток байтов между клиентом и сервером.

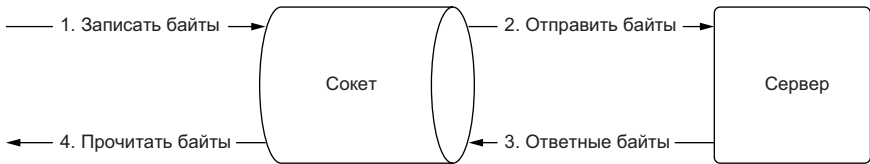


Рис. 1.7 Запись байтов в сокет и чтение байтов из сокета

По умолчанию сокеты *блокирующие*. Это значит, что на все время ожидания ответа от сервера приложение приостанавливается или *блокируется*. Следовательно, оно не может ничего делать, пока не придут данные от сервера или произойдет ошибка или случится тайм-аут.

На уровне операционной системы эта блокировка ни к чему. Сокеты могут работать в *неблокирующем режиме*, когда мы просто начинаем операцию чтения или записи и забываем о ней, а сами занимаемся другими делами. Но позже операционная система уведомляет нас о том, что байты получены, и в этот момент мы можем уделить им внимание. Это позволяет приложению не ждать прихода байтов, а делать что-то полезное. Для реализации такой схемы применяются различные системы уведомления с помощью событий, разные в разных ОС. Библиотека *asyncio* абстрагирует различия между системами уведомления, а именно:

- *kqueue* – FreeBSD и MacOS;
- *epoll* – Linux;
- *IOCP* (порт завершения ввода-вывода) – Windows.

Эти системы наблюдают за неблокирующими сокетами и уведомляют нас, когда с сокетом можно начинать работу. Именно они лежат в основе модели конкурентности в *asyncio*. В этой модели имеется только один поток, исполняющий Python-код. Встретив операцию ввода-вывода, интерпретатор передает ее на попечение системы уведомления, входящей в состав ОС. Совершив этот акт, поток Python волен исполнять другой код или открыть другие неблокирующие сокеты, о которых позаботится ОС. По завершении операции система «пробуждает» задачу, ожидающую результата, после чего выполня-

ется код, следующий за этой операцией. Эта схема изображена на рис. 1.8, где имеется несколько операций с разными сокетами.

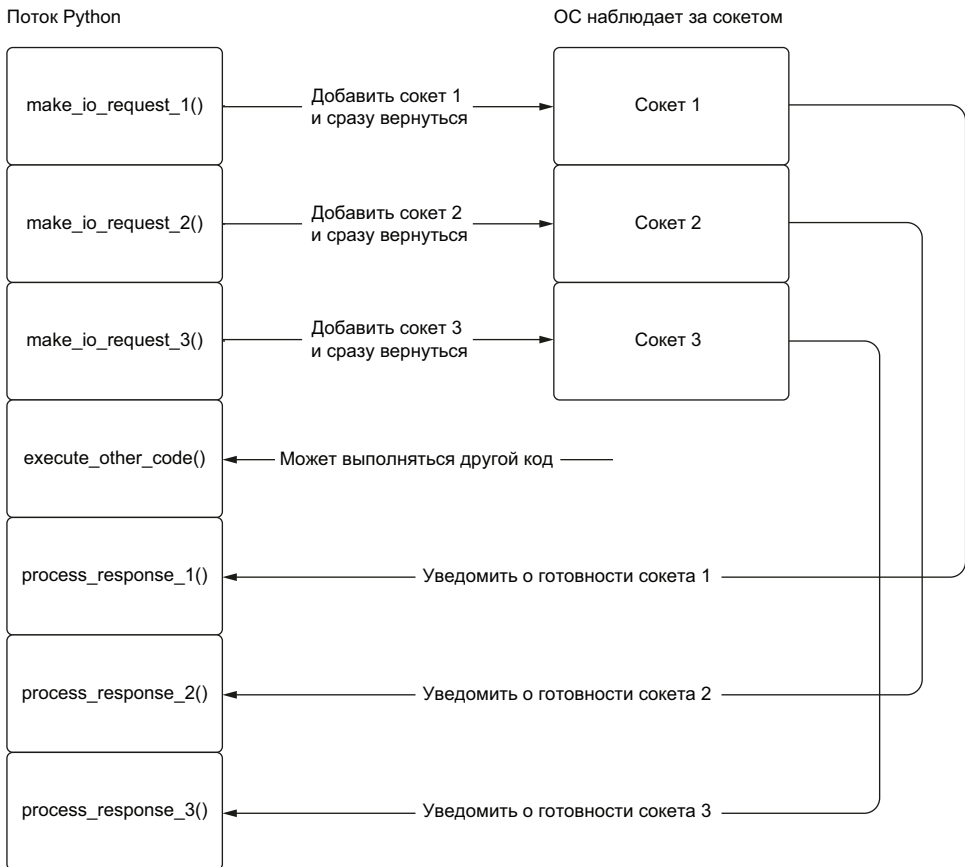


Рис. 1.8 После записи запроса в неблокирующий сокет управление возвращается немедленно, а ОС начинает наблюдать за данными в сокете. Поэтому функция `execute_other_code()` начинает выполняться сразу, не дожидаясь завершения операции ввода-вывода. Когда операция завершится, мы получим уведомление и сможем обработать ответ

Но как теперь отличить задачи, ожидающие завершения ввода-вывода, от тех, которые просто выполняют Python-код и ничего не ждут? Ответ дает конструкция под названием «цикл событий».

1.7 Как работает цикл событий

Цикл событий – сердце любого приложения `asyncio`. Этот паттерн проектирования встречается во многих системах и был придуман уже довольно давно. Используя в браузере JavaScript для отправки асин-

хронного запроса, вы создаете задачу, управляемую циклом событий. В GUI-приложениях Windows за кулисами используются так называемые циклы обработки сообщений; это основной механизм обработки таких событий, как нажатие клавиш, он позволяет одновременно отрисовывать интерфейс и реагировать на действия пользователя.

По сути своей цикл событий очень прост. Мы создаем очередь, в которой хранится список событий или сообщений, а затем входим в бесконечный цикл, где обрабатываем сообщения по мере их поступления. В Python базовый цикл событий мог бы выглядеть следующим образом:

```
from collections import deque

messages = deque()

while True:
    if messages:
        message = messages.pop()
        process_message(message)
```

В *asyncio* цикл событий управляет очередью задач, а не сообщений. Задача – это обертка вокруг сопрограммы. Сопрограмма может приостановить выполнение, встретив операцию ввода-вывода, и дать циклу событий возможность выполнить другие задачи, которые не ждут завершения ввода-вывода.

Создавая цикл событий, мы создаем пустую очередь задач. Затем добавляем в эту очередь задачи для выполнения. На каждой итерации цикла проверяется, есть ли в очереди готовая задача, и если да, то она выполняется, пока не встретит операцию ввода-вывода. В этот момент задача приостанавливается, и мы просим операционную систему наблюдать за ее сокетами. А сами тем временем переходим к следующей готовой задаче. На каждой итерации проверяется, завершилась ли какая-нибудь операция ввода-вывода; если да, то ожидавшие ее завершения задачи пробуждаются и им предоставляется возможность продолжить работу. Эта идея иллюстрируется на рис. 1.9: главный поток поставляет задачи циклу событий, а тот их выполняет.

Для конкретики представим, что имеется три задачи, каждая из которых отправляет асинхронный веб-запрос. Допустим, что на этапе инициализации они выполняют некоторый счетный код, затем посылают веб-запрос, а по его завершении обрабатывают результат, что снова требуется счет. На псевдокоде это выглядит так:

```
def make_request():
    cpu_bound_setup()
    io_bound_web_request()
```

```

cpu_bound_postprocess()

task_one = make_request()
task_two = make_request()
task_three = make_request()

```

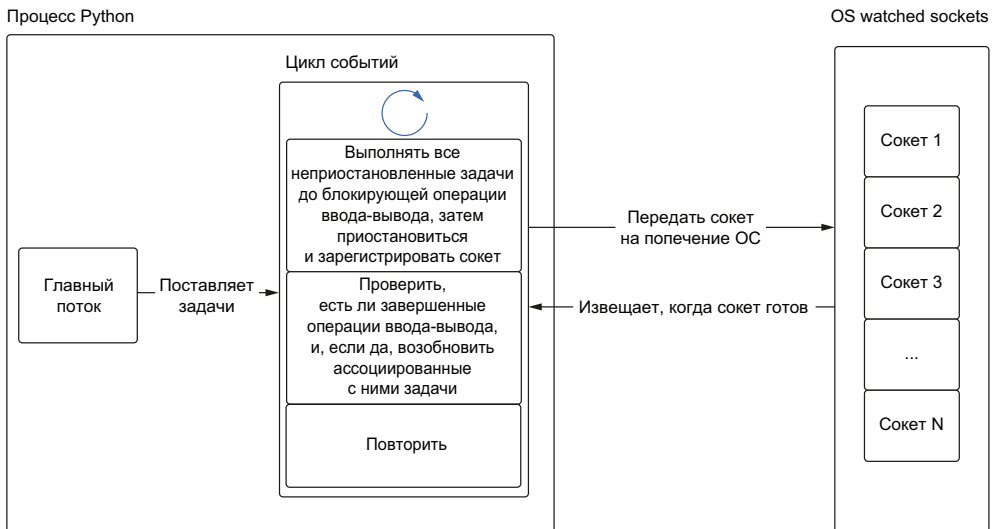


Рис. 1.9 Пример потока, поставляющего задачи циклу событий

Все три задачи вначале выполняют счетные операции, а поскольку поток всего один, то первая задача начинает работать, а остальные две ждут. Закончив счетную часть, задача 1 встречает операцию ввода-вывода и приостанавливается со словами: «Я жду завершения ввода-вывода, пусть поработают другие». После этого начинает работать задача 2 и, встретив операцию ввода-вывода, тоже приостанавливается. В этот момент обе задачи, 1 и 2, ждут завершения ввода-вывода, так что может приступить к работе задача 3.

Теперь допустим, что пока задача 3 ждет завершения своей операции ввода-вывода, пришел ответ на веб-запрос от задачи 1. Операционная система уведомляет нас о том, что ввод-вывод завершен. Мы можем возобновить задачу 1, пока задачи 2 и 3 ждут.

На рис. 1.10 показан поток выполнения, соответствующий только что описанному псевдокоду. Глядя на диаграмму слева направо, мы видим, что в каждый момент времени работает только один счетный кусок кода, но при этом конкурентно выполняются одна или две операции ввода-вывода. Именно из-за такого перекрытия ожидания ввода-вывода `asyncio` и достигает экономии во времени.

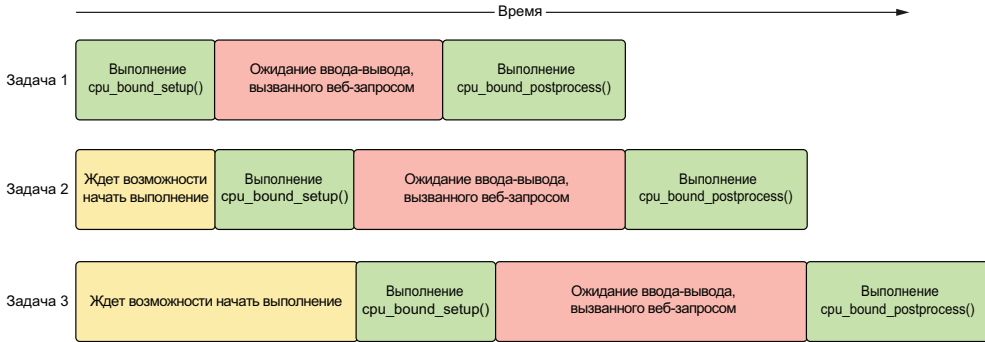


Рис. 1.10 Конкурентное выполнение нескольких задач с операциями ввода-вывода

Резюме

- Ограниченной быстродействием процессора (счетной) называется работа, потребляющая в основном ресурсы процессора, а ограниченной производительностью ввода-вывода – работа, потребляющая в основном ресурсы сети или других устройств ввода-вывода. Главная задача библиотеки *asyncio* – обеспечить конкурентность задач, ограниченных производительностью ввода-вывода, однако она также предлагает API для организации конкурентности счетных задач.
- Процессы и потоки – основные единицы конкурентности на уровне операционной системы. Процессы можно использовать для рабочих нагрузок, ограниченных как производительностью ввода-вывода, так и быстродействием процессора, а потоки в Python (обычно) – только для эффективного управления задачами, ограниченными производительностью ввода-вывода, потому что GIL не дает выполнять код параллельно.
- Мы видели, что в случае неблокирующих сокетов можно не приостанавливать приложение на время ожидания данных, а попросить операционную систему уведомить нас, когда данные поступят. Именно это позволяет *asyncio* организовать конкурентность в одном потоке.
- Мы познакомились с циклом событий, лежащим в основе приложения *asyncio*. В бесконечном цикле событий исполняются счетные задачи, а задачи, ожидающие ввода-вывода, приостанавливаются.

2

Основы *asyncio*

Краткое содержание главы

- Основы синтаксиса `async await` и сопрограмм.
- Конкурентное выполнение сопрограмм с помощью задач.
- Снятие задач.
- Создание цикла событий вручную.
- Измерение времени выполнения сопрограммы.
- Наблюдение за проблемами при выполнении сопрограмм.

В главе 1 мы видели, как организуется конкурентность с помощью процессов и потоков. Мы также рассмотрели возможность использования неблокирующего ввода-вывода и цикла событий для организации конкурентности в одном потоке. В этой главе мы поговорим о том, как писать однопоточные конкурентные программы с применением *asyncio*. Рассмотренные приемы позволят вам одновременно запускать длительные операции, например веб-запросы, запросы к базе данных и установление сетевых подключений.

Мы узнаем о *сопрограммах* и применении синтаксиса `async await` для их определения и выполнения. Мы также научимся выполнять сопрограммы конкурентно с помощью задач и посмотрим, какой можно достичь экономии времени, для чего создадим повторно используемый таймер. Наконец, рассмотрим типичные ошибки при работе с *asyncio* и научимся находить их в режиме отладки.

2.1 Знакомство с сопрограммами

Сопрограмму можно рассматривать как обычную функцию Python, наделенную сверхспособностью: приостанавливаться, встретив операцию, для выполнения которой нужно заметное время. По завершении такой длительной операции сопрограмму можно «пробудить», после чего она продолжит выполнение. Пока приостановленная сопрограмма ждет завершения операции, мы можем выполнять другой код. Такое выполнение другого кода во время ожидания и обеспечивает конкурентность внутри приложения. Можно также одновременно выполнять несколько длительных операций, что еще больше повышает производительность приложения.

Для создания и приостановки сопрограммы нам придется использовать ключевые слова Python `async` и `await`. Слово `async` определяет сопрограмму, а слово `await` приостанавливает ее на время выполнения длительной операции.

2.1.1 Создание сопрограмм с помощью ключевого слова *async*

Создать сопрограмму так же просто, как обычную функцию Python, только вместо ключевого слова `def` нужно использовать `async def`. Ключевое слово `async` говорит, что это сопрограмма, а не обычная функция.

Листинг 2.1 Использование ключевого слова *async*

```
async def my_coroutine() -> None:
    print('Hello world!')
```

Здесь сопрограмма всего лишь печатает сообщение «Hello world!». Отметим, что она не выполняет никаких длительных операций, просто печатает и возвращает управление. Это значит, что после передачи циклу событий эта сопрограмма будет выполнена немедленно, поскольку никакого блокирующего ввода-вывода нет и ничто не вынуждает ее приостановиться.

Несмотря на простоту синтаксиса, мы создали нечто, весьма отличное от обычной функции Python. Для иллюстрации напомним функцию, которая прибавляет единицу к целому числу, и сопрограмму, делающую то же самое, и сравним результаты. Также мы воспользуемся вспомогательной функцией `type`, чтобы узнать типы значений, возвращаемых сопрограммой и обычной функцией.

Листинг 2.2 Сравнение сопрограмм с обычными функциями

```
async def coroutine_add_one(number: int) -> int:
    return number + 1

def add_one(number: int) -> int:
```



```
    return number + 1

function_result = add_one(1)
coroutine_result = coroutine_add_one(1)

print(f'Результат функции равен {function_result}, а его тип равен {type(function_result)}')
print(f'Результат сопрограммы равен {coroutine_result}, а его тип равен {type(coroutine_result)}')
```

Этот код печатает следующие строки:

```
Результат функции равен 2, а его тип равен <class 'int'>
Результат сопрограммы равен <coroutine object coroutine_add_one at 0x1071d6040>,
а его тип равен <class 'coroutine'>
```

Обратите внимание, что обычная функция `add_one` выполняется и возвращает управление сразу, а результат вполне ожидаемый – целое число. Однако код сопрограммы `coroutine_add_one` вообще не выполняется, а получаем мы *объект сопрограммы*.

Это важный момент – сопрограммы не выполняются, если их вызывать напрямую. Вместо этого возвращается объект сопрограммы, который будет выполнен позже. Чтобы выполнить сопрограмму, мы должны явно передать ее циклу событий. И как же создать цикл событий и выполнить в нем нашу сопрограмму?

В версиях Python, предшествующих 3.7, цикл событий нужно было создавать вручную, если его еще не было. Но затем в `asyncio` было добавлено несколько функций, абстрагирующих управление циклом событий. Одна из них – вспомогательная функция `asyncio.run`, которую можно использовать для запуска нашей сопрограммы. Это показано в следующем листинге.

Листинг 2.3 Выполнение сопрограммы

```
import asyncio

async def coroutine_add_one(number: int) -> int:
    return number + 1

result = asyncio.run(coroutine_add_one(1))

print(result)
```

При выполнении этого кода печатается «2», как и следовало ожидать. Мы подали сопрограмму циклу событий и выполнили ее!

В этом случае `asyncio.run` делает несколько важных вещей. Во-первых, она создает новое событие. Потом она выполняет код переданной нами сопрограммы до конца и возвращает результат. Эта функция также подчищает все то, что могло остаться после завершения сопрограммы. И в конце она останавливает и закрывает цикл событий.

Но, возможно, самое главное в `asyncio.run` – то, что она задумана как главная точка входа в созданное нами приложение `asyncio`. Она выполняет только одну сопрограмму, и эта сопрограмма должна позаботиться обо всех остальных аспектах приложения. Далее мы будем использовать эту функцию почти во всех наших приложениях. Сопрограмма, которую выполняет `asyncio.run`, должна создать и запустить все прочие сопрограммы, это позволит нам обратить себе на пользу конкурентную природу `asyncio`.

2.1.2 Приостановка выполнения с помощью ключевого слова `await`

Пример, который мы видели в листинге, не имело смысла оформлять в виде сопрограммы, потому что в нем выполняется только неблокирующий Python-код. Истинное достоинство `asyncio` – возможность приостановить выполнение и дать циклу событий возможность выполнить другие задачи, пока длительная операция делает свое дело. Для приостановки выполнения служит ключевое слово `await`, за ним обычно следует обращение к сопрограмме (точнее, к объекту, *допускающему ожидание*, который необязательно является сопрограммой; мы вернемся к этому вопросу ниже в этой главе).

Использование ключевого слова `await` приводит к выполнению следующей за ним сопрограммы, а не просто к возврату объекту сопрограммы, как при прямом вызове. Кроме того, выражение `await` приостанавливает объемлющую сопрограмму до того момента, как сопрограмма, которую мы ждем, завершится и вернет результат. А после этого мы получим доступ к возвращенному результату, а объемлющая сопрограмма пробудится и обработает результат.

Ключевое слово `await` следует поместить перед вызовом сопрограммы. Продолжая предыдущую программу, мы можем написать код, который вызывает функцию `add_one` из асинхронной функции `main` и получает результат.

Листинг 2.4 Использование `await` для ожидания результата сопрограммы

```
import asyncio

async def add_one(number: int) -> int:
    return number + 1

async def main() -> None:
    one_plus_one = await add_one(1)
    two_plus_one = await add_one(2)
    print(one_plus_one)
    print(two_plus_one)

asyncio.run(main())
```

Приостановиться и ждать результата `add_one(1)`

Приостановиться и ждать результата `add_one(2)`

В листинге 2.4 мы приостанавливаем выполнение дважды. Сначала ждем завершения `add_one(1)`. После получения результата выполнение функции `main` возобновляется, и мы присваиваем значение, возвращенное `add_one(1)`, переменной `one_plus_one`, в данном случае она станет равна 2. Затем то же самое мы проделываем с `add_one(2)`, после чего печатаем результаты. Поток выполнения изображен на рис. 2.1. В блоках показано, что происходит в одной или нескольких строках кода.

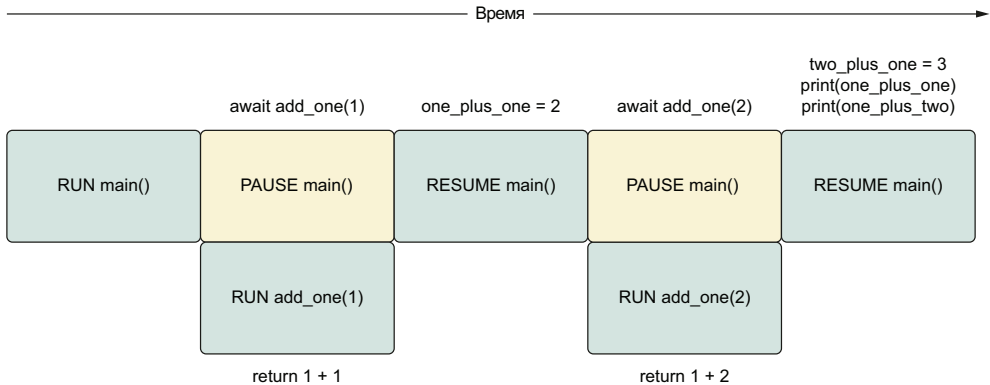


Рис. 2.1 Встретив выражение `await`, интерпретатор приостанавливает родительскую сопрограмму и выполняет сопрограмму в выражении `await`. По ее завершении родительская сопрограмма приостанавливается и возвращенное ей значение присваивается переменной

Этот код работает так же, как обычный последовательный код. По существу, мы имитируем обычный стек вызовов. А теперь рассмотрим простой пример, показывающий, как во время ожидания выполнить другой код, для чего введем в рассмотрение фиктивную операцию засыпания.

2.2 Моделирование длительных операций с помощью *sleep*

В предыдущих примерах не было медленных операций, мы просто изучили базовый синтаксис сопрограмм. Чтобы в полной мере почувствовать все преимущества и показать, как обрабатывать несколько событий одновременно, нам нужны какие-то длительные операции. Но, вместо того чтобы сразу отправлять недетерминированные веб-запросы или запросы к базе данных, мы смоделируем длительную операцию, указав, сколько времени ждать завершения. Для этого воспользуемся функцией `asyncio.sleep`.

Функция `asyncio.sleep` заставляет сопрограмму «заснуть» на заданное число секунд, т. е. приостанавливает ее на заданное время.

Это позволяет смоделировать, что происходит при обращении к базе данных или веб-ресурсу.

`asyncio.sleep` сама является сопрограммой, поэтому вызывать ее следует с помощью `await`. Вызвав ее напрямую, мы получим просто объект сопрограммы. Раз `asyncio.sleep` – сопрограмма, то, пока мы ее ждем, может выполняться другой код.

В листинге ниже показан простой пример – программа засыпает на 1 с, а затем печатает сообщение «Hello World!».

Листинг 2.5 Первое применение `sleep`

```
import asyncio

async def hello_world_message() -> str:
    await asyncio.sleep(1)
    return 'Hello World!'

async def main() -> None:
    message = await hello_world_message()
    print(message)

asyncio.run(main())
```

Приостановить `hello_world_message` на 1 с

Приостановить `main` до завершения `hello_world_message`

Эта программа ждет 1 с, а затем печатает сообщение «Hello World!». Поскольку `hello_world_message` – сопрограмма, а мы приостановили ее на 1 с, у нас появилась одна секунда, в течение которой мог бы конкурентно работать другой код.

В последующих примерах мы часто будем использовать `sleep`, поэтому потратим немного времени на создание повторно используемой сопрограммы, которая спит заданное время, а затем печатает полезную информацию. Назовем ее `delay`. Код приведен ниже.

Листинг 2.6 Повторно используемая сопрограмма `delay`

```
import asyncio

async def delay(delay_seconds: int) -> int:
    print(f'засыпаю на {delay_seconds} с')
    await asyncio.sleep(delay_seconds)
    print(f'сон в течение {delay_seconds} с закончился')
    return delay_seconds
```

Сопрограмма `delay` принимает целое число – продолжительность сна в секундах – и возвращает это же число вызывающей стороне по завершении сна. Мы также печатаем сообщения, когда сон начался и закончился. Это позволит увидеть, как конкурентно работает другой код, пока наша сопрограмма приостановлена.

Чтобы было проще обращаться к этой функции, создадим модуль, который будем при необходимости импортировать. В этот модуль мы будем добавлять и другие написанные нами повторно используемые

функции. Назовем его `util` и поместим нашу функцию `delay` в файл `delay_functions.py`. Также добавим файл `__init__.py`, содержащий следующую строку, чтобы было удобнее импортировать таймер:

```
from util.delay_functions import delay
```

Начиная с этого момента, мы будем применять предложение `from util import delay` всякий раз, как нужно использовать функцию `delay`. Теперь, когда у нас есть повторно используемая сопрограмма задержки, объединим ее с написанной ранее сопрограммой `add_one` и попробуем заставить простое сложение работать конкурентно, пока `hello_world_message` приостановлена.

Листинг 2.7 Выполнение двух сопрограмм

```
import asyncio
from util import delay

async def add_one(number: int) -> int:
    return number + 1

async def hello_world_message() -> str:
    await delay(1)
    return 'Hello World!'

async def main() -> None:
    message = await hello_world_message()
    one_plus_one = await add_one(1)
    print(one_plus_one)
    print(message)

asyncio.run(main())
```

При выполнении этого кода проходит 1 с, прежде чем будут напечатаны результаты обеих функций. Мы же хотели, чтобы значение `add_one(1)` было напечатано немедленно, пока `hello_world_message()` работает конкурентно. И что же не так с этим кодом? Дело в том, что `await` приостанавливает текущую сопрограмму и код внутри нее не выполняется, пока выражение `await` не вернет значение. Поскольку `hello_world_message` вернет значение только через секунду, сопрограмма `main` на эту секунду и приостанавливается. В данном случае код ведет себя как последовательный. Это поведение показано на рис. 2.2.

И `main`, и `hello_world` приостановлены в ожидании завершения `delay(1)`. А когда это случится, `main` возобновляется и может выполнить `add_one`.

Нам хотелось бы уйти от этой последовательной модели и выполнять `add_one` конкурентно с `hello_world`. Для этого введем в рассмотрение концепцию *задачи*.

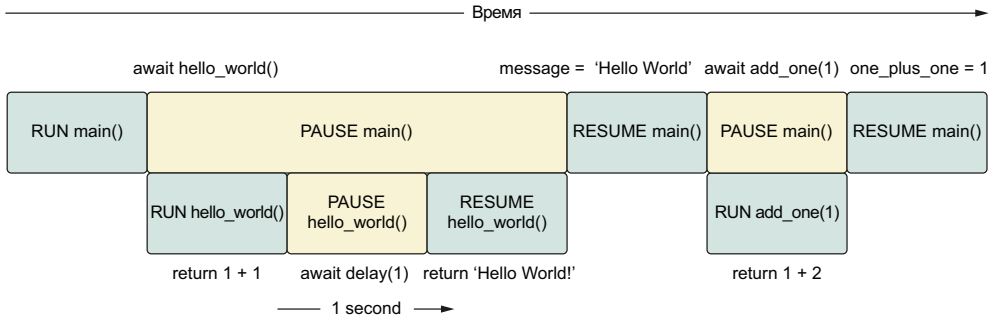


Рис. 2.2 Поток выполнения кода в листинге 2.7

2.3 Конкурентное выполнение с помощью задач

Ранее мы видели, что непосредственный вызов сопрограммы не передает ее циклу событий для выполнения. Вместо этого мы получаем объект сопрограммы, который нужно затем использовать совместно с ключевым словом `await` или передать функции `asyncio.run`, чтобы получить возвращенное значение. Располагая только этими инструментами, мы можем написать асинхронный код, но не можем выполнить его конкурентно. А чтобы это сделать, нужны *задачи*.

Задача – это обертка вокруг сопрограммы, которая планирует выполнение последней в цикле событий как можно раньше. И планирование, и выполнение происходят в неблокирующем режиме, т. е., создав задачу, мы можем сразу приступить к выполнению другого кода, пока эта задача работает в фоне. Сравните с ключевым словом `await`, которое блокирует выполнение, т. е. мы приостанавливаем всю сопрограмму на время, пока выражение `await` не вернет управление.

Способность создавать задачи и планировать их для немедленного выполнения в цикле событий означает, что несколько задач может работать приблизительно в одно и то же время. Пока одна задача выполняет длительную операцию, другие могут работать конкурентно. Для иллюстрации создадим две задачи и попробуем выполнить их одновременно.

2.3.1 Основы создания задач

Для создания задачи служит функция `asyncio.create_task`. Ей передается подлежащая выполнению сопрограмма, а в ответ она немедленно возвращает объект задачи. Этот объект можно включить в выражение `await`, которое извлечет возвращенное значение по завершении задачи.

Листинг 2.8 Создание задачи

```
import asyncio
from util import delay

async def main():
    sleep_for_three = asyncio.create_task(delay(3))
    print(type(sleep_for_three))
    result = await sleep_for_three
    print(result)

asyncio.run(main())
```

Здесь мы создали задачу, которой для выполнения нужно 3 с. Кроме того, мы печатаем тип задачи, в данном случае `<class 'asyncio.Task'>`, чтобы показать, что это не сопрограмма.

Следует также отметить, что предложение печати выполняется сразу после запуска задачи. Если бы мы просто использовали `await` для сопрограммы `delay`, то увидели бы сообщение только через 3 с.

Напечатав сообщение, мы применяем `await` к задаче `sleep_for_three`. Это приостанавливает сопрограмму `main` до получения результата от задачи.

Важно отметить, что обычно в каком-то месте приложения нужно использовать `await` для задачи. В листинге 2.8 если бы мы не включили `await`, то задача была бы запланирована, но почти сразу остановлена, после чего интерпретатор «прибрал» бы за ней, когда `asyncio.run` завершит цикл событий. Использование `await` применительно к задачам влияет также на обработку исключений, о чем речь пойдет в главе 3. Познакомившись с тем, как создавать задачи и конкурентно запускать другой код, посмотрим, как можно одновременно выполнять несколько длительных операций.

2.3.2 Конкурентное выполнение нескольких задач

Коль скоро задачи создаются мгновенно и планируются для выполнения как можно раньше, мы получаем возможность конкурентно выполнять несколько длительных задач. Для этого нужно последовательно запустить несколько задач в одной долго работающей сопрограмме.

Листинг 2.9 Конкурентное выполнение нескольких задач

```
import asyncio
from util import delay

async def main():
    sleep_for_three = asyncio.create_task(delay(3))
    sleep_again = asyncio.create_task(delay(3))
    sleep_once_more = asyncio.create_task(delay(3))

    await sleep_for_three
```

```

    await sleep_again
    await sleep_once_more

asyncio.run(main())

```

Здесь мы запустили три задачи, каждой из которых для завершения нужно 3 с. Каждое обращение к `create_task` возвращает управление немедленно, поэтому до предложения `await sleep_for_three` мы доходим сразу же. Ранее мы отмечали, что выполнение задач планируется «как можно раньше». На практике это означает, что в точке, где встречается первое после создания задачи предложение `await`, все ожидающие задачи начинают выполняться, так как `await` запускает очередную итерацию цикла событий.

Поскольку первым мы встречаем предложение `await sleep_for_three`, все три задачи начинают выполняться и засыпают одновременно. Значит, программа в листинге 2.9 завершится примерно через 3 с. Эта конкурентность показана на рис. 2.3 – обратите внимание, что все три задачи исполняют свои сопрограммы `sleep` в одно и то же время.

Заметим, что на рис. 2.3 код в задачах, помеченных `RUN delay(3)` (в данном случае некоторые предложения `print`), не работает конкурентно с другими задачами; конкурентно задачи только спят. Если бы мы выполняли операции задержки последовательно, то программа работала бы дольше 9 с. А конкурентность позволила уменьшить время работы в три раза!

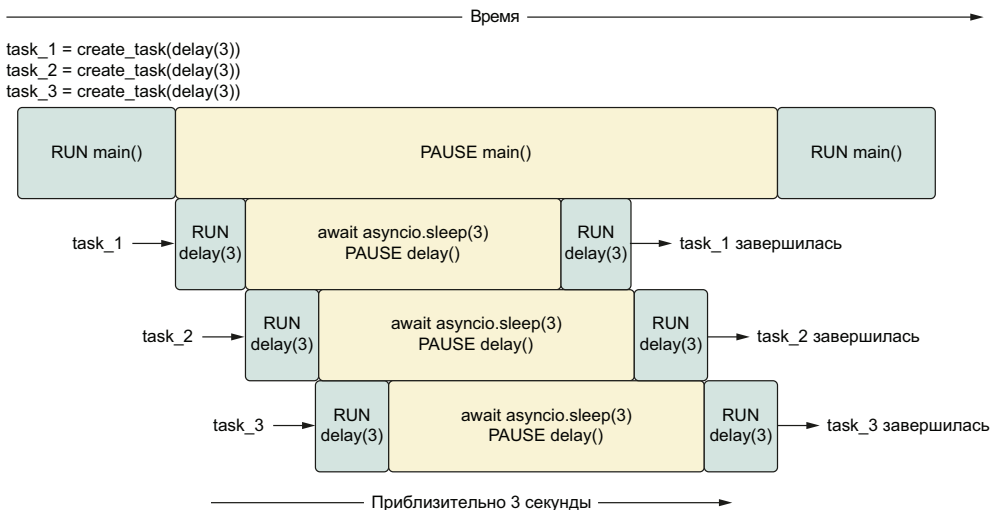


Рис. 2.3 Поток выполнения программы в листинге 2.9

ПРИМЕЧАНИЕ Это превосходство растет по мере увеличения числа задач; если бы мы запустили 10 таких задач, то программа работала бы те же 3 с, что дало бы 10-кратное ускорение.

Конкурентное выполнение таких длительных операций – как раз та область, где `asyncio` блистает и резко улучшает производительность приложения, но на этом преимущества не заканчиваются. Приложение в листинге 2.9 активно било баклуши, в течение трех секунд ожидая истечения времени задержки. Но пока один код ожидает, можно было бы выполнить другой. Допустим, мы хотим один раз в секунду печатать сообщения о состоянии, пока какие-то длительные задачи работают.

Листинг 2.10 Выполнение кода, пока другие операции работают в фоне

```
import asyncio
from util import delay

async def hello_every_second():
    for i in range(2):
        await asyncio.sleep(1)
        print("пока я жду, выполняется другой код!")

async def main():
    first_delay = asyncio.create_task(delay(3))
    second_delay = asyncio.create_task(delay(3))
    await hello_every_second()
    await first_delay
    await second_delay
```

Здесь мы создаем две задачи, работающие по 3 с. Пока эти задачи ждут, приложение простаивает, что дает нам возможность занять его другим кодом. В этом примере выполняется сопрограмма `hello_every_second`, которая дважды печатает сообщение с интервалом в одну секунду. Пока две задачи что-то делают, мы видим, как печатаются сообщения:

```
засыпаю на 3 с
засыпаю на 3 с
пока я жду, выполняется другой код!
пока я жду, выполняется другой код!
сон в течение 3 с закончился
сон в течение 3 с закончился
```

Поток выполнения показан на рис. 2.4.

Сначала мы запускаем две задачи, которые спят в течение 3 с; пока эти задачи простаивают, мы видим, как каждую секунду печатается сообщение «пока я жду, выполняется другой код!». Это означает, что даже во время выполнения длительных операций наше приложение может выполнять другие задачи.

Потенциальная проблема заключается в том, что задача может работать неопределенно долго. Быть может, нам захочется остановить задачу, если она никак не кончается сама. Такая возможность поддерживается и называется снятием.

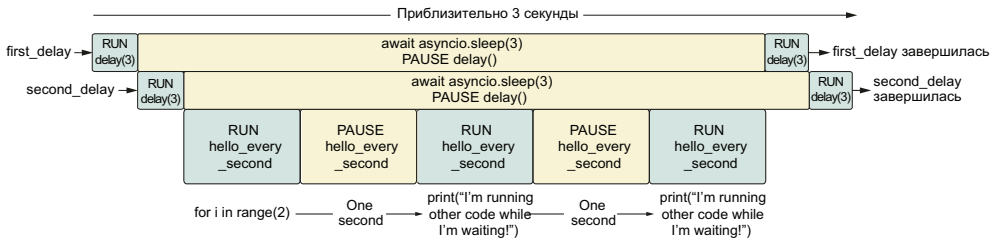


Рис. 2.4 Поток выполнения программы в листинге 2.10

2.4 Снятие задач и задание тайм-аутов

Сетевые подключения ненадежны. Установленное пользователем подключение может быть разорвано из-за медленной сети, или веб-сервер может «упасть» и оставить существующие запросы в подвешенном состоянии. Поэтому, отправляя запросы, мы должны внимательно следить за ними, чтобы не ждать слишком долго. Иначе приложение может зависнуть, ожидая результата, который никогда не придет. Наш пользователь останется недоволен – маловероятно, что он захочет вечно ждать ответа на свой запрос. Кроме того, иногда имеет смысл предоставить пользователю выбор, если задача работает слишком долго. Пользователь может согласиться с тем, что его запрос требует много времени, или остановить задачу, запущенную по ошибке.

В рассмотренных выше примерах если бы задачи работали вечно, то мы бы застряли в предложении `await` без всякой обратной связи. И остановить бы программу не смогли, даже если бы захотели. В библиотеке *asynсio* предусмотрены обе ситуации – мы можем снять задачу или задать тайм-аут.

2.4.1 Снятие задач

Снять задачу просто. У каждого объекта задачи есть метод `cancel`, который можно вызвать, если требуется остановить задачу. В результате снятия задача возбудит исключение `CanceledError`, когда мы ждем ее с помощью `await`. Это исключение можно обработать, как того требует ситуация.

Для иллюстрации предположим, что мы запустили задачу, которая не должна работать дольше 5 с. Если за это время задача не завершилась, то мы хотим ее снять, сообщив пользователю, что задача работает слишком долго и будет остановлена. Мы также хотим каждую секунду печатать сообщение о состоянии, чтобы держать пользователя в курсе, а не оставлять в неведении на протяжении нескольких секунд.

Листинг 2.11 Снятие задачи

```
import asyncio
from asyncio import CancelledError
from util import delay

async def main():
    long_task = asyncio.create_task(delay(10))

    seconds_elapsed = 0

    while not long_task.done():
        print('Задача не закончилась, следующая проверка через секунду.')
        await asyncio.sleep(1)
        seconds_elapsed = seconds_elapsed + 1
        if seconds_elapsed == 5:
            long_task.cancel()

    try:
        await long_task
    except CancelledError:
        print('Наша задача была снята')

asyncio.run(main())
```

Здесь мы создаем задачу, работающую 10 с. Затем в цикле `while` проверяем состояние задачи. Метод задачи `done` возвращает `True`, если задача завершилась, и `False` в противном случае. Каждую секунду мы проверяем, завершилась ли задача, и запоминаем, сколько секунд уже прошло. Если задача работает дольше 5 с, то мы ее снимаем. Далее задача запускается в предложении `await long_task` и, если возникло исключение `CancelledError`, печатается сообщение «Наша задача была снята».

Важно отметить, что исключение `CancelledError` может быть возбуждено только внутри предложения `await`. То есть, если вызвать метод `cancel`, когда задача исполняет Python-код, этот код будет продолжать работать, пока не встретится следующее предложение `await` (если встретится), и только тогда будет возбуждено исключение `CancelledError`. Вызов `cancel` не прерывает задачу, делающую свое дело; он снимает ее, только если она уже находится в точке ожидания или когда дойдет до следующей такой точки.

2.4.2 Задание тайм-аута и снятие с помощью `wait_for`

Проверять состояние каждую секунду или с другим интервалом, как в предыдущем примере, – не самый простой способ реализации тайм-аута. В идеале хотелось бы иметь вспомогательную функцию, которая позволяла бы задать тайм-аут и снять задачу по его истечении.

В `asyncio` есть такая возможность в виде функции `asyncio.wait_for`. Она принимает объект сопрограммы или задачи и тайм-аут в секун-

дах и возвращает сопрограмму, к которой можно применить `await`. Если задача не завершилась в отведенное время, то возбуждается исключение `TimeoutError` и задача автоматически снимается.

Для иллюстрации работы `wait_for` мы рассмотрим случай, когда задаче требуется две секунды, но мы даем ей только одну. Мы перехватываем исключение `TimeoutError` и смотрим, была ли задача снята.

Листинг 2.12 Задание тайм-аута для задачи с помощью `wait_for`

```
import asyncio
from util import delay

async def main():
    delay_task = asyncio.create_task(delay(2))
    try:
        result = await asyncio.wait_for(delay_task, timeout=1)
        print(result)
    except asyncio.exceptions.TimeoutError:
        print('Тайм-аут!')
        print(f'Задача была снята? {delay_task.cancelled()}')

asyncio.run(main())
```

Эта программа завершается примерно через 1 с. По истечении 1 с предложение `wait_for` возбуждает исключение `TimeoutError`, которое мы обрабатываем, а именно смотрим, была ли снята задача `delay`, и печатаем следующие сообщения:

```
засыпаю на 2 с
Тайм-аут!
Задача была снята? True
```

Автоматическое снятие задачи, работающей дольше, чем ожидается, обычно является разумной практикой. В противном случае сопрограмма могла бы ждать неопределенно долго, занимая ресурсы, которые никогда не будут освобождены. Но в некоторых случаях желательно дать сопрограмме поработать. Например, по прошествии некоторого времени мы можем проинформировать пользователя о том, что работа занимает дольше, чем ожидалось, но не снимать ее, когда тайм-аут истечет.

Для этого обернем нашу задачу функцией `asyncio.shield`. Эта функция предотвращает снятие сопрограммы, снабжая ее «щитом», позволяющим игнорировать запросы на снятие.

Листинг 2.13 Защита задачи от снятия

```
import asyncio
from util import delay

async def main():
    task = asyncio.create_task(delay(10))

    try:
```

```
        result = await asyncio.wait_for(asyncio.shield(task), 5)
        print(result)
    except TimeoutError:
        print("Задача заняла более 5 с, скоро она закончится!")
        result = await task
        print(result)

asyncio.run(main())
```

Здесь мы сначала создаем задачу, обертывающую сопрограмму. В этом состоит отличие от нашего первого примера снятия, потому что нам необходим доступ к задаче в блоке `except`. Если бы мы передали сопрограмму, то `wait_for` обернула бы ее задачей, но сослаться на эту задачу мы бы не смогли, потому что она внутренняя.

Затем внутри блока `try` мы вызываем `wait_for`, обернув предварительно задачу функцией `shield`, чтобы она не была снята. В блоке обработки исключения мы печатаем полезное сообщение пользователю, в котором говорим, что задача еще работает, после чего ждем ее завершения с помощью `await`. Это позволит задаче доработать до конца, а пользователь увидит следующие сообщения:

```
засыпаю на 10 с
Задача заняла более 5 с, скоро она закончится!
сон в течение 10 с закончился
<function delay at 0x10e8cf820> завершилась за 10 с
```

Снятие и защита от снятия – довольно хитрые вещи, у которых есть несколько достойных внимания сценариев применения. Мы познакомились с основами, но по мере усложнения примеров более глубоко изучим, как работает механизм снятия.

Мы рассмотрели основы задач и сопрограмм. Эти понятия переплетаются. В следующем разделе мы поговорим о том, как они связаны между собой, и увидим, как устроена библиотека `asyncio`.

2.5 *Задачи, сопрограммы, будущие объекты и объекты, допускающие ожидание*

И сопрограммы, и задачи можно использовать в выражениях `await`. Так что же между ними общего? Чтобы ответить на этот вопрос, нужно знать о типах `future` и `awaitable`. На практике будущие объекты (`future`) бывают нужны редко, но понимать их необходимо, если мы хотим уяснить, как работает `asyncio`. Далее в книге мы будем специально отмечать API, возвращающие будущие объекты.

2.5.1 *Введение в будущие объекты*

Объект `future` в Python содержит одно значение, которое мы ожидаем получить в будущем, но пока еще, возможно, не получили. Обычно

в момент создания `future` не обертывает никакого значения, потому что его еще не существует. Объект в таком состоянии называется неполным, неразрешенным или просто неготовым. И только получив результат, мы можем установить значение объекта `future`, в результате чего он становится полным и из него можно извлечь результат. Чтобы лучше разобраться со всем этим, создадим будущий объект, установим его значение и затем извлечем его.

Листинг 2.14 Основы будущих объектов

```
from asyncio import Future

my_future = Future()

print(f'my_future готов? {my_future.done()}')

my_future.set_result(42)

print(f'my_future готов? {my_future.done()}')

print(f'Какой результат хранится в my_future? {my_future.result()}')
```

Для создания объекта `future` нужно вызвать его конструктор. В этот момент во `future` нет никакого результата, поэтому вызов метода `done` возвращает `False`. Затем мы устанавливаем значение `future` методом `set_result`, который помечает его как готовый. Если бы вместо этого мы хотели записать во `future` исключение, то вызвали бы метод `set_exception`.

ПРИМЕЧАНИЕ Мы не вызываем метод `result`, прежде чем результат установлен, потому что тогда он возбудил бы исключение `InvalidState`.

Будущие объекты также можно использовать в выражениях `await`. Это означает «я посплю, пока в будущем объекте не будет установлено значение, с которым я могу работать, а когда оно появится, разбуди меня и дай возможность его обработать».

Рассмотрим пример – отправка веб-запроса возвращает объект `future`. В этом случае `future` возвращается немедленно, но, поскольку запрос занимает некоторое время, значение `future` еще не определено. Позже, когда запрос завершится, результат будет установлен, и мы сможем его получить. Те, кто знаком с JavaScript, легко увидят аналогию с *обещаниями* (`promise`). В Java похожая концепция называется *дополняемыми будущими объектами* (`completable future`).

Листинг 2.15 Ожидание будущего объекта

```
from asyncio import Future
import asyncio

def make_request() -> Future:
```

```

future = Future()
asyncio.create_task(set_future_value(future))
return future

async def set_future_value(future) -> None:
    await asyncio.sleep(1)
    future.set_result(42)

async def main():
    future = make_request()
    print(f'Будущий объект готов? {future.done()}')
    value = await future
    print(f'Будущий объект готов? {future.done()}')
    print(value)

asyncio.run(main())

```

Создать задачу, которая асинхронно установит значение future

Ждать 1 с, прежде чем установить значение

Приостановить main, пока значение future не установлено

Здесь мы определяем функцию `make_request`. В этой функции создается объект `future` и создается задача, которая асинхронно установит результат `future` через 1 с. Затем в функции `main` мы вызываем `make_request`. Она сразу же возвращает неготовый будущий объект, не содержащий результата, после чего мы применяем к нему `await`. Ожидание готовности этого объекта приостанавливает `main` на 1 с. Когда ожидание завершится, `value` будет равно 42 и объект `future` станет готовым.

В `asyncio` редко приходится иметь дело с будущими объектами. Тем не менее встречаются API, возвращающие будущие объекты, а иногда возникает необходимость писать код обратных вызовов, что требует будущих объектов. Кроме того, возможно, вам доведется читать или отлаживать код каких-то API `asyncio` самостоятельно. Реализация таких API опирается на будущие объекты, так что неплохо бы хотя бы в общих чертах понимать, как они работают.

2.5.2 Связь между будущими объектами, задачами и сопрограммами

Между задачами и будущими объектами существует тесная связь. На самом деле `task` напрямую наследует `future`. Можно считать, что объект `future` представляет значение, которое появится только в будущем. А `task` является комбинацией сопрограммы и `future`. Создавая задачу, мы создаем пустой объект `future` и запускаем сопрограмму. А когда сопрограмма завершится с результатом или вследствие исключения, мы записываем этот результат или объект-исключение во `future`.

А существует ли аналогичная связь между задачами и сопрограммами? Ведь все эти типы можно использовать в выражениях `await`.

Связующим звеном между ними является абстрактный базовый класс `Awaitable`. В нем определен единственный абстрактный метод `__await__`. Мы не будем вдаваться в детали того, как создавать соб-

ственные объекты, допускающие ожидание, а просто скажем, что любой объект, который реализует метод `__await__`, можно использовать в выражении `await`. Сопрограммы, как и будущие объекты, наследуют `Awaitable` напрямую. Задачи же расширяют будущие объекты, так что мы имеем диаграмму наследования, показанную на рис. 2.5.

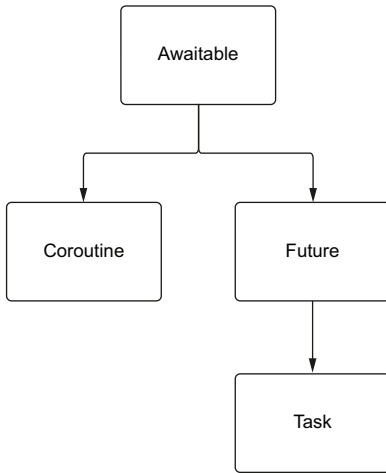


Рис. 2.5 Иерархия наследования класса `Awaitable`

Далее будем называть объекты, которые можно использовать в выражениях `await`, объектами, *допускающими ожидание* (*awaitable*). Этот термин часто встречается в документации по *asyncio*, поскольку многие методы API готовы принимать и сопрограммы, и задачи, и будущие объекты.

Итак, с основами сопрограмм, задач и будущих объектов мы разобрались. А как оценить их производительность? До сих пор мы только теоретизировали на тему времени их работы. Чтобы сделать рассуждения более строгими, добавим средства измерения времени.

2.6 Измерение времени выполнения сопрограммы с помощью декораторов

До сих пор мы лишь приблизительно говорили о том, сколько времени работают наши приложения, не измеряя его явно. Чтобы построить настоящий профиль, нужно написать код хронометража.

В качестве первой попытки можно было бы обернуть каждое предположение `await`, запоминая время начала и завершения сопрограммы:

```
import asyncio
import time
```



```
async def main():
    start = time.time()
    await asyncio.sleep(1)
    end = time.time()
    print(f'Сон занял {end - start} с)

asyncio.run(main())
```

Однако если предложений `await` и задач много, то это быстро надоест. Хорошо бы придумать допускающий повторное использование способ замерять время работы любой сопрограммы. Это можно сделать с помощью декоратора, который будет выполнять предложение `await` за нас (листинг 2.16). Назовем этот декоратор `async_timed`.

Что такое декоратор?

Декоратор в Python – это паттерн, который позволяет расширять функциональность существующей функции, не изменяя ее код. Мы можем «перехватить» вызываемую функцию и применить декоратор до или после вызова. Декоратор – один из способов реализации сквозной функциональности. В следующем листинге приведен пример декоратора.

Листинг 2.16 Декоратор для хронометража сопрограмм

```
import functools
import time
from typing import Callable, Any

def async_timed():
    def wrapper(func: Callable) -> Callable:
        @functools.wraps(func)
        async def wrapped(*args, **kwargs) -> Any:
            print(f'выполняется {func} с аргументами {args} {kwargs}')
            start = time.time()
            try:
                return await func(*args, **kwargs)
            finally:
                end = time.time()
                total = end - start
                print(f'{func} завершилась за {total:.4f} с')
        return wrapped

    return wrapper
```

Здесь мы создаем новую сопрограмму `wrapped`. Это обертка вокруг исходной сопрограммы, которая принимает ее аргументы, `*args` и `**kwargs`, выполняет предложение `await` и возвращает результат. Мы окружили это предложение двумя сообщениями: одно печатается в начале выполнения функции, а второе в конце. В них отображаются время начала и завершения точно так же, как мы делали в пре-

дыдущих примерах. Теперь можно добавить к любой сопрограмме этот декоратор в виде аннотации и увидеть, сколько времени она работала.

Листинг 2.17 Хронометраж двух конкурентных задач с помощью декоратора

```
import asyncio

@async_timed()
async def delay(delay_seconds: int) -> int:
    print(f'засыпаю на {delay_seconds} с')
    await asyncio.sleep(delay_seconds)
    print(f'сон в течение {delay_seconds} с закончился')
    return delay_seconds

@async_timed()
async def main():
    task_one = asyncio.create_task(delay(2))
    task_two = asyncio.create_task(delay(3))

    await task_one
    await task_two

asyncio.run(main())
```

Эта программа печатает примерно такие сообщения:

```
выполняется <function main at 0x109111ee0> с аргументами () {}
выполняется <function delay at 0x1090dc700> с аргументами (2,) {}
выполняется <function delay at 0x1090dc700> с аргументами (3,) {}
<function delay at 0x1090dc700> завершилась за 2.0032 с
<function delay at 0x1090dc700> завершилась за 3.0003 с
<function main at 0x109111ee0> завершилась за 3.0004 с
```

Как видим, два вызова `delay` работали примерно 2 и 3 с соответственно, что в сумме составляет 5 с. Заметим, однако, что сопрограмма `main` работала всего 3 с, поскольку ожидание производилось конкурентно.

Мы будем пользоваться этим декоратором и сведениями, которые он выводит, в следующих нескольких главах, чтобы показать, сколько времени требуется для выполнения нашим сопрограммам и когда они начинаются и завершаются. Это поможет составить четкое представление о том, какой выигрыш в производительности дает конкурентность.

Чтобы на этот декоратор было проще ссылаться в будущем, включим его в наш модуль `util` и запишем код в файл `async_timer.py`. Также добавим в файл модуля `__init__.py` следующую строку, позволяющую удобно импортировать таймер:

```
from util.async_timer import async_timed
```

Далее в этой книге мы будем писать `from util import async_timed`, когда понадобится таймер.

Теперь, когда можем использовать декоратор для измерения выигрыша, который дает `asyncio` при конкурентном выполнении задач, попробуем применить `asyncio` к существующим приложениям. Это возможно, но нужно внимательно следить за тем, чтобы не попасть в одну из типичных ловушек, которые могут не повысить, а снизить производительность приложения.

2.7 Ловушки сопрограмм и задач

Увидев, какой выигрыш может дать конкурентное выполнение длительных задач, мы можем поддасться искушению использовать сопрограммы и задачи всюду и везде. Но просто снабдить функции ключевым словом `async` и обернуть их задачами может оказаться недостаточно для повышения производительности. А в некоторых случаях производительность может даже упасть.

Есть две основные ошибки на пути преобразования приложения в асинхронное. Первая – попытка выполнить счетный код в задачах или сопрограммах, не прибегая к многопроцессности, вторая – использовать блокирующие API ввода-вывода, пренебрегая многопоточностью.

2.7.1 Выполнение счетного кода

В программе могут быть функции, выполняющие длительные вычисления, например обход большого словаря или математические расчеты. Если есть возможность выполнять эти функции конкурентно, то может возникнуть идея поместить их в отдельные задачи. В принципе, ничего плохого в этом нет, но нужно помнить, что модель конкурентности в `asyncio` однопоточная. Это значит, что действуют все ограничения одного потока и глобальной блокировки интерпретатора.

Чтобы убедиться в этом, попробуем запустить счетные функции конкурентно.

Листинг 2.18 Попытка конкурентного выполнения счетного кода

```
import asyncio
from util import delay

@async_timed()
async def cpu_bound_work() -> int:
    counter = 0
    for i in range(100000000):
        counter = counter + 1
    return counter
```

```
@async_timed()
async def main():
    task_one = asyncio.create_task(cpu_bound_work())
    task_two = asyncio.create_task(cpu_bound_work())
    await task_one
    await task_two

asyncio.run(main())
```

Выполнив этот код, мы увидим, что, несмотря на создание двух задач, он по-прежнему работает последовательно: сначала задача 1, потом – 2, и, значит, общее время работы складывается из двух обращений к `cpu_bound_work`:

```
выполняется <function main at 0x10a8f6c10> с аргументами () {}
выполняется <function cpu_bound_work at 0x10a8c0430> с аргументами () {}
<function cpu_bound_work at 0x10a8c0430> завершилась за 4.6750 с
выполняется <function cpu_bound_work at 0x10a8c0430> с аргументами () {}
<function cpu_bound_work at 0x10a8c0430> завершилась за 4.6680 с
<function main at 0x10a8f6c10> завершилась за 9.3434 с
```

Глядя на этот результат, мы можем подумать, что нет ничего плохого в том, чтобы повсюду расставить `async` и `await`. В конце концов, времени-то уходит столько же, сколько при последовательном выполнении. Однако при таком подходе мы можем оказаться в ситуации, когда производительность приложения падает. Особенно если в программе есть другие сопрограммы или задачи, в которых встречаются выражения `await`. Рассмотрим создание двух счетных задач наряду с длительной задачей, например нашей сопрограммой `delay`.

Листинг 2.19 Счетный код и длительная задача

```
import asyncio
from util import async_timed, delay

@async_timed()
async def cpu_bound_work() -> int:
    counter = 0
    for i in range(100000000):
        counter = counter + 1
    return counter

@async_timed()
async def main():
    task_one = asyncio.create_task(cpu_bound_work())
    task_two = asyncio.create_task(cpu_bound_work())
    delay_task = asyncio.create_task(delay(4))
    await task_one
    await task_two
    await delay_task

asyncio.run(main())
```

Кажется, что эта программа должна работать столько же, сколько программа в листинге 2.18. Разве `delay_task` не будет выполняться конкурентно со счетными задачами? Нет, не будет, потому что мы сначала создали две счетные задачи и тем самым не даем циклу событий выполнить что-то еще. Следовательно, время работы приложения будет равно сумме времен работы задач `cpu_bound_work` плюс 4 с, которые займет задача `delay`.

Если требуется выполнить счетную работу и все-таки использовать `async / await`, то это можно сделать. Но придется воспользоваться многопроцессностью и попросить `asyncio` выполнять наши задачи в *пуле процессов*. Как это сделать, мы узнаем в главе 6.

2.7.2 Выполнение блокирующих API

Может возникнуть соблазн использовать существующие библиотеки ввода-вывода, обернув их сопрограммами. Однако при этом возникнут те же проблемы, что для счетных операций. Эти API будут блокировать главный поток. Поэтому, попытавшись выполнить блокирующий вызов API в сопрограмме, мы заблокируем сам поток цикла событий, а значит, воспрепятствуем выполнению всех остальных сопрограмм и задач. Примерами блокирующих API является библиотека `requests` или функция `time.sleep`. Вообще, любая функция, которая выполняет ввод-вывод, не являясь сопрограммой, или занимает процессор длительными операциями, может считаться блокирующей.

В качестве примера попробуем трижды конкурентно получить код состояния страницы `www.example.com` с помощью библиотеки `requests`. Поскольку задачи запускаются конкурентно, можно ожидать, что на все про все уйдет столько же времени, сколько на однократное получение кода состояния.

Листинг 2.20 Неправильное использование блокирующего API как сопрограммы

```
import asyncio
import requests
from util import async_timed

@async_timed()
async def get_example_status() -> int:
    return requests.get('http://www.example.com').status_code

@async_timed()
async def main():
    task_1 = asyncio.create_task(get_example_status())
    task_2 = asyncio.create_task(get_example_status())
    task_3 = asyncio.create_task(get_example_status())
    await task_1
    await task_2
    await task_3

asyncio.run(main())
```

Результат выполнения этой программы показан ниже. Обратите внимание, что полное время работы сопрограмы `main` приблизительно равно сумме времен запущенных задач получения кода состояния, т. е. мы не получили никакого выигрыша от конкурентности.

```
выполняется <function main at 0x1102e6820> с аргументами () {}
выполняется <function get_example_status at 0x1102e6700> с аргументами () {}
<function get_example_status at 0x1102e6700> завершилась за 0.0839 с
выполняется <function get_example_status at 0x1102e6700> с аргументами () {}
<function get_example_status at 0x1102e6700> завершилась за 0.0441 с
выполняется <function get_example_status at 0x1102e6700> с аргументами () {}
<function get_example_status at 0x1102e6700> завершилась за 0.0419 с
<function main at 0x1102e6820> завершилась за 0.1702 с
```

И снова причина в том, что библиотека `requests` блокирующая, т. е. блокирует поток, в котором выполняется. Поскольку *asynсio* однопоточная, библиотека `requests` блокирует цикл событий и не дает ничему выполняться конкурентно.

Большинство API, с которыми мы обычно работаем, в настоящее время являются блокирующими и без доработок работать с *asynсio* не будут. Нужно использовать библиотеку, которая поддерживает сопрограмы и неблокирующие сокеты. А это значит, что если используемая вами библиотека не возвращает сопрограмы и вы не употребляете `await` в собственных сопрограмах, то, вероятно, совершаете блокирующий вызов.

В примере выше мы могли бы использовать библиотеку `aiohttp`, в которой используются неблокирующие сокеты и которая возвращает сопрограмы, тогда с конкурентностью все было бы нормально. Мы познакомимся с этой библиотекой в главе 4.

Если вы все-таки хотите использовать библиотеку `requests`, то синтаксис *asynс* применить можно, но нужно явно попросить *asynсio* действовать многопоточность с помощью *исполнителя пула потоков*. Как это делается, узнаем в главе 7.

Мы рассказали, на что обращать внимание при работе с *asynсio*, и написали несколько простых приложений. До сих пор мы не создавали и не конфигурировали цикл событий самостоятельно, а полагались на уже готовые методы. В следующем разделе научимся создавать цикл событий, что позволит нам получить доступ к низкоуровневой функциональности *asynсio* и конфигурационным свойствам цикла событий.

2.8 Ручное управление циклом событий

До сих пор мы пользовались вспомогательной функцией `asynсio.run`, которая за кулисами создавала цикл событий и запускала приложение. В силу своей простоты это предпочтительный метод создания цикла событий. Но бывает, что функциональность, предлагаемая *asynсio*.

run, нас не устраивает. Например, что, если мы хотим реализовать специальную логику остановки задач, например дать оставшимся задачам завершиться, а не останавливать их, как делает `asyncio.run`?

Кроме того, могут понадобиться методы самого цикла событий. Как правило, они низкоуровневые, так что злоупотреблять ими не стоит. Однако если нужно сделать что-то нестандартное, например работать с сокетами напрямую или запланировать задачу на конкретный момент в будущем, то доступ к циклу событий необходим. Мы, конечно, не хотим и не должны увлекаться ручным управлением циклом событий, но временами от этого никуда не деться.

2.8.1 Создание цикла событий вручную

Мы можем создать цикл событий, воспользовавшись методом `asyncio.new_event_loop`. Он возвращает экземпляр цикла событий, который дает доступ ко всем низкоуровневым методам, в частности методу `run_until_complete`, который принимает сопрограмму и исполняет ее до завершения. Закончив работу с циклом событий, мы должны закрыть его, чтобы освободить занятые ресурсы. Обычно это делается в блоке `finally`, чтобы цикл был закрыт даже в случае исключения. Ниже показано, как создать цикл событий и запустить в нем приложение `asyncio`.

Листинг 2.21 Создание цикла событий вручную

```
import asyncio

async def main():
    await asyncio.sleep(1)

loop = asyncio.new_event_loop()

try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

Это похоже на то, что происходит при вызове `asyncio.run`, с той разницей, что оставшиеся задачи не отменяются. Если нам нужна специальная логика очистки, то ее следует реализовать в предложении `finally`.

2.8.2 Получение доступа к циклу событий

Иногда бывает необходим доступ к текущему циклу событий. Библиотека `asyncio` предоставляет для этой цели функцию `asyncio.get_running_loop`. В качестве примера рассмотрим метод `call_soon`, который планирует выполнение функции на следующей итерации цикла событий.

Листинг 2.22 Получение доступа к циклу событий

```
import asyncio

def call_later():
    print("Меня вызовут в ближайшем будущем!")

async def main():
    loop = asyncio.get_running_loop()
    loop.call_soon(call_later)
    await delay(1)

asyncio.run(main())
```

Здесь сопрограмма `main` получает цикл событий от функции `asyncio.get_running_loop`, и вызывает его метод `call_later`, который принимает функцию и выполняет ее на следующей итерации цикла. Существует еще функция `asyncio.get_event_loop`, также позволяющая получить доступ к циклу событий. Эта функция может создать новый цикл событий, если его еще не существует в момент вызова, что ведет к странному поведению. Рекомендуется использовать `get_running_loop`, поскольку она возбуждает исключение, если цикл событий не запущен, что позволяет избежать сюрпризов.

Хотя явно использовать цикл событий в приложениях слишком часто не стоит, бывает, что нужно настроить его параметры или воспользоваться низкоуровневыми функциями. В следующем разделе мы рассмотрим пример перевода цикла событий в *отладочный режим*.

2.9 Отладочный режим

В предыдущих разделах мы говорили, что любую сопрограмму всегда следует ожидать в какой-то точке приложения. Мы также видели недостатки выполнения счетного и иного блокирующего кода в сопрограммах и задачах. Но бывает трудно сказать, то ли сопрограмма потребляет слишком много процессорного времени, то ли мы по ошибке забыли где-то поставить `await`. По счастью, `asyncio` предоставляет отладочный режим для диагностики таких ситуаций.

При работе в отладочном режиме печатаются полезные сообщения, когда сопрограмма или задача работают больше 100 мс. Кроме того, если для некоторой сопрограммы отсутствует `await`, то возбуждается исключение, показывающее, в каком месте следовало бы добавить `await`. Есть несколько способов войти в отладочный режим.

2.9.1 Использование *asyncio.run*

Функция `asyncio.run`, которой мы пользовались для выполнения сопрограмм, имеет именованный параметр `debug`. По умолчанию он

равен `False`, но если присвоить ему значение `True`, то активируется отладочный режим:

```
asyncio.run(coroutine(), debug=True)
```

2.9.2 Использование аргументов командной строки

Включить отладочный режим можно, передав аргумент `-X dev` в командной строке, запускающей Python-приложение:

```
python3 -X dev program.py
```

2.9.3 Использование переменных окружения

Включить отладочный режим можно также, присвоив значение `1` переменной окружения `PYTHONASYNCIODEBUG`:

```
PYTHONASYNCIODEBUG=1 python3 program.py
```

ПРИМЕЧАНИЕ В версиях Python младше 3.9 в отладочном режиме имеется ошибка. При использовании `asyncio.run` работает только булев параметр `debug`. Задание параметра в командной строке или переменной окружения работает, только если цикл событий управляется вручную.

В отладочном режиме мы будем видеть информационные сообщения, если сопрограмма работает слишком долго. Чтобы проверить это, попробуем запустить следующий счетный код в задаче и посмотрим, будет ли напечатано предупреждение.

Листинг 2.23 Выполнение счетного кода в отладочном режиме

```
import asyncio
from util import async_timed

@async_timed()
async def cpu_bound_work() -> int:
    counter = 0
    for i in range(100000000):
        counter = counter + 1
    return counter

async def main() -> None:
    task_one = asyncio.create_task(cpu_bound_work())
    await task_one

asyncio.run(main(), debug=True)
```

При запуске этой программы мы увидим сообщение о том, что задача `task_one` работает слишком долго и, следовательно, блокирует цикл событий, не давая ему выполнять другие задачи:

```
Executing <Task finished name='Task-2' coro=<cpu_bound_work() done, defined  
at listing_2_9.py:5> result=1000000000 created at tasks.py:382> took 4.829  
seconds
```

Это может быть полезно для отладки ошибок, связанных со случайным выполнением блокирующего вызова. По умолчанию параметры заданы так, что предупреждение выдается, если сопрограмма работает дольше 100 мс, но, возможно, для вас это слишком мало или слишком много. Чтобы изменить значение, нужно получить объект цикла событий и задать в нем параметр `slow_callback_duration`, как показано в листинге 2.24. Это число с плавающей точкой равно количеству секунд, при превышении которого обратный вызов считается медленным.

Листинг 2.24 Изменение продолжительности медленного обратного вызова

```
import asyncio

async def main():
    loop = asyncio.get_event_loop()
    loop.slow_callback_duration = .250

asyncio.run(main(), debug=True)
```

Здесь продолжительность медленного обратного вызова установлена равной 250 мс, т. е. сообщение печатается, если сопрограмма работает дольше 250 мс.

Резюме

- Мы научились создавать сопрограммы с помощью ключевого слова `async`. Сопрограмма может приостанавливать себя, встретив блокирующую операцию. Это дает возможность поработать другим сопрограммам. После завершения операции, на которой сопрограмма приостановилась, она пробуждается и продолжает работу с прерванного места.
- Мы узнали, что вызову сопрограммы должно предшествовать ключевое слово `await`, означающее, что нужно дождаться возврата значения. При этом сопрограмма, внутри которой встретилось слово `await`, приостанавливается в ожидании результата. В это время могут работать другие сопрограммы.
- Мы узнали, как использовать функцию `asyncio.run` для выполнения одной сопрограммы, являющейся точкой входа в приложение.
- Мы научились использовать задачи для конкурентного выполнения нескольких длительных операций. Задачи – это обертки вокруг сопрограмм, которые исполняются в цикле событий. Созданная задача планируется для выполнения как можно раньше.

- Мы узнали, как снимать задачу, когда нужно остановить ее, и как задать тайм-аут, чтобы задача не работала бесконечно долго. После снятия задачи возбуждается исключение `CancelledError`, когда мы ожидаем результат. Если имеются ограничения на время работы задачи, то можно задать тайм-аут в методе `asyncio.wait_for`.
- Мы научились избегать типичных ошибок, которые допускают начинающие изучать `asyncio`. Первая – выполнение счетного кода в сопрограмах. Счетный код блокирует цикл событий и не дает выполняться другим сопрограммам, потому что модель конкурентности однопоточная. Вторая – блокирующий ввод-вывод, потому что обычные библиотеки нельзя использовать совместно с `asyncio`, а нужно работать с заточенными под `asyncio` и возвращающими сопрограммы. Если внутри вашей сопрограммы не встречается `await`, это должно вызвать подозрения. Тем не менее существуют способы исполнять счетный код и блокирующий ввод-вывод совместно с `asyncio`, мы рассмотрим их в главах 6 и 7.
- Мы узнали об отладочном режиме. Он помогает диагностировать типичные проблемы в коде на основе `asyncio`, например выполнение счетного кода в сопрограмме.

3

Первое приложение asyncio

Краткое содержание главы

- Применение сокетов для передачи данных по сети.
- Применение telnet для взаимодействия с приложением на основе сокетов.
- Использование селекторов для построения простого цикла событий для неблокирующих сокетов.
- Создание неблокирующего эхо-сервера, допускающего несколько подключений.
- Обработка исключений в задачах.
- Включение специальной логики остановки в приложение asyncio.

В главах 1 и 2 мы познакомились с сопрограммами, задачами и циклом событий, а также узнали, как выполнять длительные операции конкурентно, и изучили некоторые API `asyncio`, предназначенные для этой цели. Но до сих пор мы лишь эмулировали длительные операции с помощью функции `sleep`.

Не желая ограничиваться одними демонстрационными приложениями, воспользуемся некоторыми реальными блокирующими операциями, чтобы показать, как можно создать сервер, способный обрабатывать несколько пользователей одновременно. Мы сделаем это, используя только один поток, в результате чего приложение окажет-

ся проще и менее требовательным к ресурсам, чем альтернативные решения с несколькими потоками и процессами. Мы применим все, что узнали о сопрограммах, задачах и API `asyncio`, для написания командного эхо-сервера, в котором будут использоваться сокеты. К концу главы вы сможете сами создавать сетевые приложения на основе `asyncio`, умеющие одновременно обслуживать несколько пользователей в одном потоке.

Сначала рассмотрим, как отправлять и принимать данные с помощью блокирующих сокетов. Мы попробуем с их помощью построить эхо-сервер, обслуживающий несколько клиентов. И убедимся, что хорошо сделать это в одном потоке невозможно. Затем поговорим о том, как разрешить возникшие проблемы, сделав сокеты неблокирующими и воспользовавшись системой уведомлений, входящей в состав операционной системы. Это поможет понять, как работает механизм, лежащий в основе цикла событий. После этого мы применим сопрограммы с неблокирующими сокетами `asyncio`, чтобы правильно реализовать обслуживание нескольких пользователей, которые одновременно отправляют и принимают сообщения. Наконец, мы добавим специальную логику остановки, оставляющую некоторое время для завершения уже начатой работы.

3.1 Работа с блокирующими сокетами

В главе 1 мы познакомились с сокетами. Напомним, что сокет – это способ читать и записывать данные по сети. Можно считать, что сокет – своего рода почтовый ящик: мы кладем в него письмо, а оно затем доставляется по адресу получателя. Получатель сможет прочесть сообщение и, возможно, отправит ответ.

Прежде всего создадим главный сокет, называемый серверным. Он будет принимать сообщения от клиентов, желающих установить с нами соединение. После того как серверный сокет подтвердит запрос на подключение, мы создаем сокет, предназначенный для взаимодействия с клиентом. Таким образом, сервер становится похож на почтовое отделение не с одним, а с несколькими почтовыми ящиками. Что касается клиента, можно по-прежнему считать, что он владеет только одним почтовым ящиком, поскольку открывает для взаимодействия с нами один сокет. Когда клиент подключается к серверу, мы предоставляем ему почтовый ящик, а затем используем его для получения и отправки сообщений (рис. 3.1).

Такой сервер можно создать с помощью встроенного в Python модуля `socket`, предоставляющего средства для чтения, записи и управления сокетом. Для начала напишем простой сервер, который прослушивает порт, куда поступают запросы на подключение от клиентов, и печатает сообщения об успешном подключении. С этим сокетом будут ассоциированы имя хоста и порт, он станет главным «серверным сокетом», с которым могут взаимодействовать клиенты.

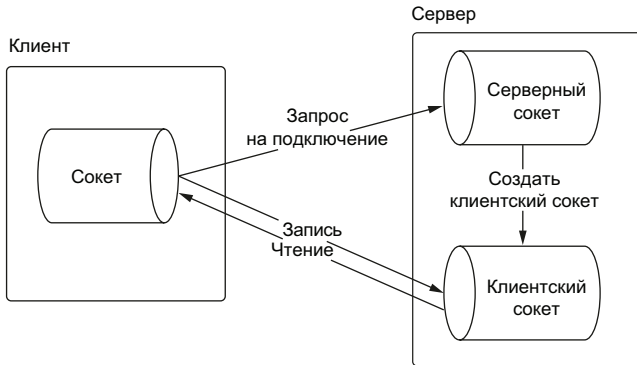


Рис. 3.1 Клиент подключается к серверному сокету. Затем сервер создает новый сокет для взаимодействия с клиентом

Для создания сокета нужно выполнить несколько шагов. Сначала с помощью функции `socket` создается сокет:

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Функция `socket` принимает два параметра. Первый, `socket.AF_INET`, – тип адреса, в данном случае адрес будет содержать имя хоста и номер порта. Второй, `socket.SOCK_STREAM`, означает, что для взаимодействия будет использоваться протокол TCP.

Что такое протокол TCP?

Протокол TCP (transmission control protocol – протокол управления передачей) предназначен для передачи данных между приложениями по сети. При его проектировании на первое место ставилась надежность. Он проверяет ошибки, гарантирует доставку данных по порядку и в случае необходимости производит повторную передачу. Но надежность сопровождается высокими накладными расходами. Большая часть веба функционирует на базе TCP. Противоположностью TCP является протокол передачи данных UDP, менее надежный, но и характеризующийся гораздо меньшими накладными расходами, чем TCP, а потому более производительный. В этой книге нас будут интересовать только TCP-сокеты.

Мы также вызываем функцию `setsockopt`, чтобы установить флаг `SO_REUSEADDR` в 1. Это позволит повторно использовать номер порта, после того как мы остановим и заново запустим приложение, избегнув тем самым ошибки «Адрес уже используется». Если этого не сделать, то операционной системе потребуется некоторое время, чтобы освободить порт, после чего приложение сможет запуститься без ошибок.

Функция `socket.socket` создает сокет, но начать взаимодействие по нему мы еще не можем, потому что сокет не привязан к адресу, по которому могут обращаться клиенты (у почтового отделения должен быть адрес!). В данном случае мы привяжем сокет к адресу своего собственного компьютера `127.0.0.1` и выберем произвольный порт `8000`:

```
address = (127.0.0.1, 8000)
server_socket.bind(server_address)
```

Теперь у сокета есть адрес – `127.0.0.1:8000`. Клиенты смогут использовать этот адрес для отправки данных нашему серверу, а если мы отправим данные клиенту, то клиент увидит адрес, с которого они пришли.

Далее мы должны активно прослушивать запросы от клиентов, желающих подключиться к нашему серверу. Для этого вызывается метод сокета `listen`. Затем мы ждем запроса на подключение с помощью метода `accept`. Этот метод блокирует программу до получения запроса, после чего возвращает объект подключения и адрес подключившегося клиента. Объект подключения – это еще один сокет, который можно использовать для чтения данных от клиента и записи адресованных ему данных.

```
server_socket.listen()
connection, client_address = server_socket.accept()
```

Теперь у нас есть все необходимое для создания серверного приложения на основе сокетов, которое будет ждать запросов на подключение и печатать сообщение, когда запрос придет.

Листинг 3.1 Запуск сервера и прослушивание порта для подключения

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
server_socket.listen()

connection, client_address = server_socket.accept()
print(f'Получен запрос на подключение от {client_address}!')
```

Создать TCP-сервер

Задать адрес сокета, 127.0.0.1:8000

Прослушивать запросы на подключение, или «открыть почтовое отделение»

Дождаться подключения и выделить клиенту почтовый ящик

Когда клиент подключится, мы получим клиентский сокет и адрес клиента, о чем не замедлим сообщить.

Итак, приложение готово, но как к нему подключиться и протестировать? Для этой цели существует немало инструментов, но в этой главе мы будем использовать командное приложение `telnet`.

3.2 Подключение к серверу с помощью telnet

В нашем простом примере нет никакого способа подключиться. Для чтения и записи данных на сервер имеется много командных приложений, в частности telnet – популярное и существующее уже давно.

Программа telnet была разработана в 1969 году, название является сокращением от *teletype network* (телетайпная сеть). Telnet устанавливает TCP-подключение к серверу, расположенному на указанном хосте. Затем создается терминал, который можно использовать для отправки и приема данных, отображаемых на экране терминала.

В Mac OS установить telnet можно с помощью программы Homebrew, набрав команду `brew install telnet` (для установки Homebrew зайдите на сайт <https://brew.sh/>). В дистрибутивах Linux нужно будет использовать системный менеджер пакетов (`apt-get install telnet` или что-то в этом роде). В Windows лучше всего использовать программу PuTTY, которую можно скачать с сайта <https://putty.org>.

ПРИМЕЧАНИЕ В PuTTY нужно будет включить режим локального редактирования строки, чтобы примеры из этой книги работали. Для этого перейдите в раздел *Terminal* в левой части окна настроек PuTTY и установите переключатель *Local line editing* в режим *Force on*.

Чтобы подключиться к серверу в листинге 3.1, мы можем выполнить команду telnet, указав, что хотим подключиться к порту 8000 хоста localhost:

```
telnet localhost 8000
```

В ответ увидим сообщение об успешном подключении. Затем telnet отображает курсор, что позволит нам печатать данные и нажатием клавиши **Enter** отправить их на сервер.

```
telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^']'.
```

На консоли серверного приложения мы должны увидеть сообщение о подключении со стороны telnet-клиента:

```
Получен запрос на подключение от ('127.0.0.1', 56526)!
```

А когда серверный код завершится, мы увидим сообщение *Connection closed by foreign host*, означающее, что сервер разорвал соединение с клиентом. Теперь у нас есть способ подключиться к серверу, а также читать данные от него и передавать ему данные, но сам сервер пока не умеет ни читать, ни записывать данные. Исправим это, воспользовавшись методами `sendall` и `recv` клиентского сокета.

3.2.1 Чтение данных из сокета и запись данных в сокет

Теперь, когда сервер, способный принимать запросы на подключение, создан, посмотрим, как читать посылаемые ему данные. В классе `socket` имеется метод `recv`, который позволяет получать данные из сокета. Метод принимает целое число, показывающее, сколько байтов мы хотим прочитать. Это важно, потому что мы не можем прочитать из сокета сразу все данные, а должны сохранять их в буфере, пока не дойдем до конца.

В данном случае концом считается пара символов: возврат каретки, перевод строки, или `'\r\n'`. Именно эта пара добавляется в конец строки, когда пользователь нажимает клавишу **Enter** в telnet. Чтобы продемонстрировать, как работает буферизация небольших сообщений, зададим размер буфера заведомо малым. В реальных приложениях нужен буфер побольше, например на 1024 байта. Большой буфер позволит воспользоваться механизмом буферизации на уровне операционной системы, это эффективнее, чем буферизация в приложении.

Листинг 3.2 Чтение данных из сокета

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
server_socket.listen()

try:
    connection, client_address = server_socket.accept()
    print(f'Получен запрос на подключение от {client_address}!')

    buffer = b''

    while buffer[-2:] != b'\r\n':
        data = connection.recv(2)
        if not data:
            break
        else:
            print(f'Получены данные: {data}!')
            buffer = buffer + data

    print(f'Все данные: {buffer}')
finally:
    server_socket.close()
```

Здесь мы ждем запроса на подключение в функции `server_socket.accept`, как и раньше. Получив запрос, мы пытаемся принять два байта и сохранить их в буфере. Затем входим в цикл и на каждой итерации проверяем, заканчивается ли буфер символами возврата каретки и перевода строки. Если нет, то получаем еще два байта, печатаем их

и добавляем в буфер. После получения `'\r\n'` мы выходим из цикла и печатаем все сообщение, полученное от клиента, а затем закрываем серверный сокет в блоке `finally`. Таким образом, соединение будет гарантированно закрыто, даже если при чтении данных возникнет исключение. Если подключиться к этому приложению из `telnet` и отправить сообщение `'testing123'`, то мы увидим такую картину:

```
Получен запрос на подключение от ('127.0.0.1', 49721)!
Получены данные: b'te'!
Получены данные: b'st'!
Получены данные: b'in'!
Получены данные: b'g1'!
Получены данные: b'23'!
Получены данные: b'\r\n'!
Все данные: b'testing123\r\n'
```

Итак, мы умеем читать данные из сокета, но как отправить данные клиенту? У сокетов имеется метод `sendall`, который принимает сообщение и отправляет его клиенту. Код в листинге 3.2 можно дополнить, так чтобы он отправлял клиенту копию полученного сообщения. Для этого нужно вызывать метод `connection.sendall`, передав ему заполненный буфер.

```
while buffer[-2:] != b'\r\n':
    data = connection.recv(2)
    if not data:
        break
    else:
        print(f'Получены данные: {data}!')
        buffer = buffer + data
print(f"Все данные: {buffer}")
connection.sendall(buffer)
```

Теперь после подключения к серверу с помощью `telnet` и отправки сообщения мы должны увидеть на терминале это же сообщение. Мы написали очень простой эхо-сервер!

Сейчас это приложение в каждый момент времени обслуживает только одного клиента, но подключиться к одному серверному сокету может несколько клиентов. Изменим код, разрешив одновременное подключение нескольких клиентов. Попутно продемонстрируем, что нормально поддерживать несколько клиентов с помощью блокирующих сокетов не получается.

3.2.2 Разрешение нескольких подключений и опасности блокирования

Сокет, находящийся в режиме прослушивания, допускает одновременное подключение нескольких клиентов. Это значит, что при повторном вызове `socket.accept` мы каждый раз будем получать новый

клиентский сокет для чтения и записи данных. Зная это, мы можем без труда изменить предыдущий пример, так чтобы сервер обслуживал несколько клиентов. Будем в бесконечном цикле вызывать `socket.accept()` для прослушивания новых подключений. Приняв подключение, добавим его в конец списка имеющихся подключений. Затем в цикле обойдем все подключения, примем из каждого данные и запишем их обратно в сокет, чтобы передать клиенту.

Листинг 3.3 Подключение нескольких клиентов

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
server_socket.listen()

connections = []

try:
    while True:
        connection, client_address = server_socket.accept()
        print(f'Получен запрос на подключение от {client_address}!')
        connections.append(connection)

        for connection in connections:
            buffer = b''

            while buffer[-2:] != b'\r\n':
                data = connection.recv(2)
                if not data:
                    break
                else:
                    print(f'Получены данные: {data}!')
                    buffer = buffer + data

            print(f"Все данные: {buffer}")

            connection.send(buffer)

finally:
    server_socket.close()
```

Можем проверить эту версию. Подключимся с помощью telnet и введем сообщение. Затем можно подключиться из второго telnet-клиента и отправить другое сообщение. И тут же наткнемся на проблему. Первый клиент работает и получает копии своих сообщений, как и положено, а вот второй не получает ничего. Связано это с тем, что по умолчанию сокеты блокирующие. Методы `accept` и `recv` блокируют выполнение программы, пока не получат данные. А значит, после того как первый клиент подключился, мы будем ждать, когда

он отправит свое первое сообщение. А остальные клиенты в это время зависнут в ожидании следующей итерации цикла, которая не произойдет, пока не придут данные от первого клиента (рис. 3.2).

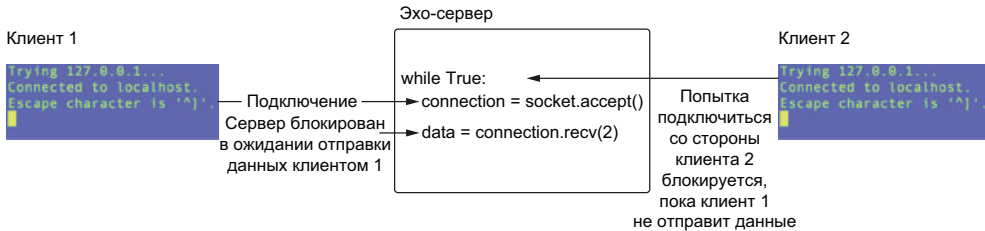


Рис. 3.2 При использовании блокирующих сокетов клиент 1 подключается, но клиент 2 заблокирован, пока первый клиент не отправит данные

Очевидно, что пользователям это не понравится; мы написали программу, которая не масштабируется на случай, когда клиентов больше одного. Эту проблему можно решить, переведя сокет в неблокирующий режим. Если сокет помечен как неблокирующий, его методы не будут блокировать выполнение программы в ожидании поступления данных.

3.3 Работа с неблокирующими сокетами

Предыдущая версия эхо-сервера допускала подключение нескольких клиентов, но при наличии более одного подключения возникали проблемы: один клиент может заставить всех остальных ждать, пока он отправит данные. Эту проблему можно решить, переведя сокет в неблокирующий режим. Тогда вызов любого метода, который мог бы блокировать выполнение, например `recv`, будет возвращать управление немедленно. Если в соquete есть доступные для обработки данные, то они будут возвращены, как в случае блокирующего сокета. А если нет, то сокет сразу даст нам знать, что данные не готовы, и мы сможем перейти к выполнению другого кода.

Листинг 3.4 Создание неблокирующего сокета

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind(('127.0.0.1', 8000))
server_socket.listen()
server_socket.setblocking(False)
```

По сути дела, создание неблокирующего сокета отличается от создания блокирующего только вызовом метода `setblocking` с параметром `False`.

тром `False`. По умолчанию это значение равно `True`, т. е. сокет блокирующий. Теперь посмотрим, что произойдет в исходном приложении, если это сделать. Пропадет ли ошибка?

Листинг 3.5 Первая попытка создать неблокирующий сервер

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
server_socket.listen()
server_socket.setblocking(False) ← Пометить серверный сокет
                                   как неблокирующий
connections = []

try:
    while True:
        connection, client_address = server_socket.accept()
        connection.setblocking(False) ← Пометить клиентский сокет
                                       как неблокирующий
        print(f'Получен запрос на подключение от {client_address}!')
        connections.append(connection)

        for connection in connections:
            buffer = b''

            while buffer[-2:] != b'\r\n':
                data = connection.recv(2)
                if not data:
                    break
                else:
                    print(f'Получены данные: {data}!')
                    buffer = buffer + data

            print(f'Все данные: {buffer}')

            connection.send(buffer)

finally:
    server_socket.close()
```

При выполнении этой программы одно отличие обнаруживается немедленно – программа падает сразу после запуска! Возникает исключение `BlockingIOError`, потому что к серверному сокету еще никто не подключился, поэтому нет данных для обработки:

```
Traceback (most recent call last):
  File "echo_server.py", line 14, in <module>
    connection, client_address = server_socket.accept()
  File "python3.8/socket.py", line 292, in accept
    fd, addr = self._accept()
BlockingIOError: [Errno 35] Resource temporarily unavailable
```

Так сокет несколько неожиданно говорит нам: «У меня нет данных, попробуйте вызвать меня еще раз позже». Не существует простого способа узнать, есть ли в сокете данные, поэтому возможное решение – перехватить исключение, проигнорировать его и продолжать цикл, пока в сокете не появятся данные. При таком подходе мы будем постоянно проверять наличие новых подключений с максимальной скоростью. Это должно решить проблему, с которой столкнулся наш эхо-сервер с блокирующим сокетом.

Листинг 3.6 Перехват и игнорирование ошибок блокирующего ввода-вывода

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
server_socket.listen()
server_socket.setblocking(False)

connections = []

try:
    while True:
        try:
            connection, client_address = server_socket.accept()
            connection.setblocking(False)
            print(f'Получен запрос на подключение от {client_address}!')
            connections.append(connection)
        except BlockingIOError:
            pass

    for connection in connections:
        try:
            buffer = b''

            while buffer[-2:] != b'\r\n':
                data = connection.recv(2)
                if not data:
                    break
                else:
                    print(f'Получены данные: {data}!')
                    buffer = buffer + data

            print(f'Все данные: {buffer}')
            connection.send(buffer)
        except BlockingIOError:
            pass

finally:
    server_socket.close()
```

На каждой итерации бесконечного цикла ни один из вызовов `assert` и `recv` не блокирует выполнение, и мы либо сразу же получаем исключение, которое игнорируем, либо данные, которые обрабатываем. Итерации выполняются быстро, и поведение программы не зависит от того, посылает кто-нибудь данные или нет. Так что проблема блокирующего сервера решена, и несколько клиентов могут подключаться и отправлять данные одновременно.

Описанный подход работает, но обходится дорого. Первый недостаток – качество кода. Перехват исключений всякий раз, как не оказывается данных, приводит к многословному и чреватому ошибками коду. Второй – потребление ресурсов. Запустив эту программу на ноутбуке, вы уже через несколько секунд услышите, что вентилятор начал работать громче. Это приложение постоянно потребляет почти 100 % процессорного времени (рис. 3.3), поскольку мы выполняем итерации цикла и получаем исключения настолько быстро, насколько позволяет операционная система. В результате в рабочей нагрузке преобладает потребление процессора.

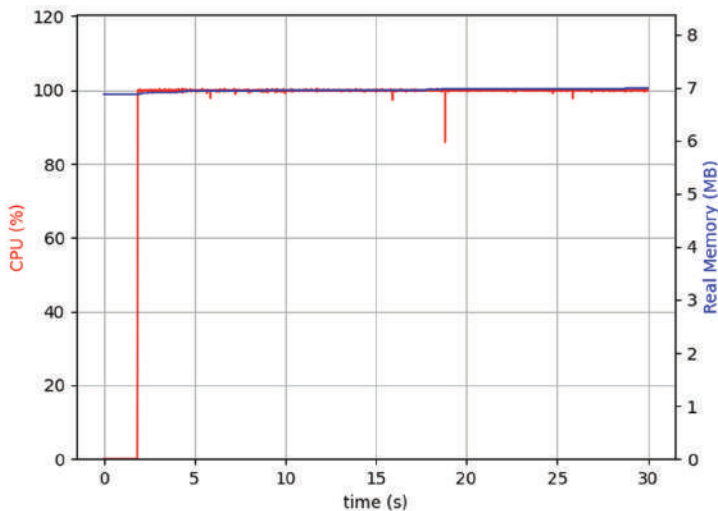


Рис. 3.3 Из-за перехвата исключений в бесконечном цикле потребление процессора быстро достигает до 100 % и на этом уровне остается

Выше мы уже упоминали о системе уведомления о событиях в операционной системе, которая может уведомить нас о том, что в соquete появились данные. Эта система опирается на аппаратные прерывания и не подразумевает опроса в цикле `while`, как в показанной выше программе. В Python имеется библиотека для использования такой системы уведомления. В следующем разделе мы воспользуемся ей для решения проблемы потребления ресурсов. Попутно построим примитивный цикл событий, работающий только для сокетов.

3.4 Использование модуля *selectors* для построения цикла событий сокетов

У операционной системы есть эффективные API, позволяющие наблюдать за появлением данных в сокетах и за другими событиями. Конкретный API зависит от системы (*kqueue*, *epoll*, *IOCP* – самые известные), но все системы уведомления работают по одному принципу. Мы передаем системе список сокетов, за событиями которых хотим наблюдать, а она сообщает, когда в одном из них появляются данные.

Поскольку все это реализовано на аппаратном уровне, процессор в мониторинге почти не участвует, так что потребление ресурсов невелико. Эти системы уведомления и лежат в основе механизма конкурентности в *asynсio*. Поняв, как они устроены, мы сможем лучше понять, как работает *asynсio*.

Система уведомления о событиях зависит от операционной системы. Но модуль Python *selectors* абстрагирует эту зависимость, так что мы получаем правильное событие, в какой бы системе наш код ни работал. То есть код оказывается переносимым.

Эта библиотека предоставляет абстрактный базовый класс *BaseSelector*, имеющий реализации для каждой системы уведомления. Кроме того, имеется класс *DefaultSelector*, который автоматически выбирает реализацию, подходящую для конкретной системы.

У класса *BaseSelector* есть несколько важных концепций. Первая из них – *регистрация*. Если мы хотим получать уведомления для какого-то сокета, то регистрируем его, сообщая, какие именно события нас интересуют, например чтение и запись. Наоборот, если сокет нас больше не интересует, то его регистрацию можно отменить.

Вторая концепция – *селекция*. Функция *select* блокирует выполнение, пока не произойдет какое-то событие, после чего возвращает список сокетов, готовых для обработки, а также событие, которое произошло с каждым сокетом. Поддерживается также тайм-аут – если в течение указанного времени ничего не произошло, то возвращается пустой список сокетов.

Имея такие строительные блоки, мы можем написать неблокирующий эхо-сервер, не нагружающий процессор. После создания серверного сокета мы регистрируем его в селекторе по умолчанию, который будет прослушивать запросы на подключение от клиентов. Как только кто-то подключится к серверному сокету, мы зарегистрируем клиентский сокет, чтобы селектор наблюдал за отправленными в него данными. Получив данные, мы отправим их назад клиенту. Кроме того, добавим тайм-аут, чтобы продемонстрировать возможность выполнять другой код, пока мы ждем наступления событий.

Листинг 3.7 Использование селектора для построения неблокирующего сервера

```

import selectors
import socket
from selectors import SelectorKey
from typing import List, Tuple

selector = selectors.DefaultSelector()

server_socket = socket.socket()
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

server_address = ('127.0.0.1', 8000)
server_socket.setblocking(False)
server_socket.bind(server_address)
server_socket.listen()

selector.register(server_socket, selectors.EVENT_READ)

while True:
    events: List[Tuple[SelectorKey, int]] = selector.select(timeout=1)

    if len(events) == 0:
        print('Событий нет, подожду еще!')

    for event, _ in events:
        event_socket = event.fileobj

        if event_socket == server_socket:
            connection, address = server_socket.accept()
            connection.setblocking(False)
            print(f"Получен запрос на подключение от {address}")
            selector.register(connection, selectors.EVENT_READ)

        else:
            data = event_socket.recv(1024)
            print(f"Получены данные: {data}")
            event_socket.send(data)

```

Создать селектор с тайм-аутом 1 с

Если ничего не произошло, сообщить об этом. Такое возможно в случае тайм-аута

Получить сокет, для которого произошло событие, он хранится в поле `fileobj`

Если событие произошло с серверным сокетом, значит, была попытка подключения

Зарегистрировать клиент, подключившийся к сокету

Если событие произошло не с серверным сокетом, получить данные от клиента и отправить их обратно

Эта программа печатает сообщение «Событий нет, подожду еще!» примерно каждую секунду, пока не будет получено событие подключения. После этого мы регистрируем сокет для прослушивания событий чтения. Теперь, как только клиент отправит нам данные, селектор вернет событие готовности данных и мы сможем прочитать их с помощью функции `socket.recv`. Мы написали полнофункциональный эхо-сервер, поддерживающий нескольких клиентов. У него нет проблем с блокированием, поскольку чтение или запись производятся только тогда, когда имеются данные. Он почти не потребляет процессорного времени, так как мы пользуемся эффективной системой уведомления о событиях, которая реализована внутри операционной системы (рис. 3.4).

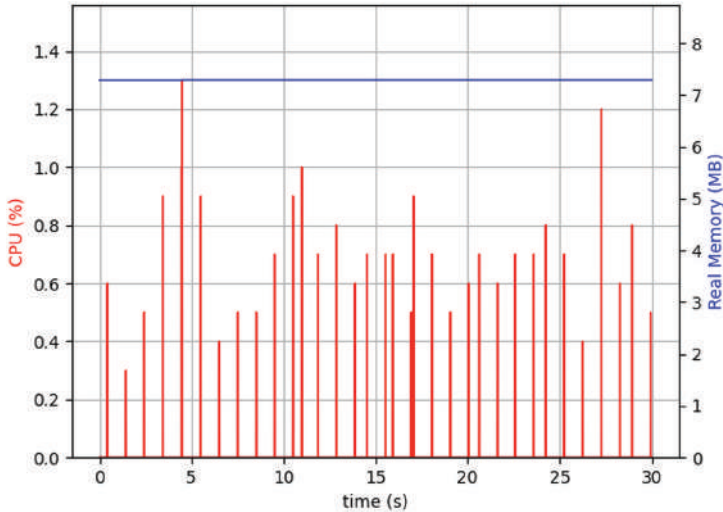


Рис. 3.4 График потребления процессорного времени эхо-сервером с селекторами. При таком методе потребление колеблется между 0 и 1 %

То, что мы сделали, во многом напоминает действия, выполняемые циклом событий под капотом. В данном случае события – получение данных через сокет. Каждая итерация нашего цикла событий и цикла событий *asynсio* запускается либо событием сокета, либо срабатыванием таймера. В цикле событий *asynсio* в любом из этих случаев активируется ожидающая сопрограмма и выполняется до конца или до следующего предложения *await*. Если *await* встречается в сопрограмме, где используется неблокирующий сокет, то она регистрирует этот сокет в системном селекторе и запоминает, что эта сопрограмма приостановлена в ожидании результата. Эту логику можно записать на псевдокоде, демонстрирующем идею:

```
paused = []
ready = []

while True:
    paused, new_sockets = run_ready_tasks(ready)
    selector.register(new_sockets)
    timeout = calculate_timeout()
    events = selector.select(timeout)
    ready = process_events(events)
```

Мы выполняем готовые к работе сопрограммы до тех пор, пока они не будут приостановлены в предложении *await*, и сохраняем их в массиве *paused*. Мы также отслеживаем новые сокеты, за которыми нужно наблюдать, и регистрируем их в селекторе. Затем вычисляем тайм-аут, указываемый при вызове *select*. Вычисление тайм-аута не вполне тривиально, обычно при этом учитывается, что запланировано для выполнения в конкретный момент времени или в течение

конкретного интервала. Примером может служить `asyncio.sleep`. Затем мы вызываем `select` и ждем возникновения события сокета или истечения тайм-аута. Как только произойдет то или другое, мы обрабатываем события и преобразуем их в список сопрограмм, готовых к выполнению.

Хотя построенный нами цикл событий пригоден только для событий сокетов, он демонстрирует идею использования селектора для регистрации интересующих нас сокетов, когда программа пробуждается только при возникновении чего-то, что мы хотим обработать. Более глубоко мы рассмотрим эту тему, когда будем строить собственный цикл событий в конце книги.

Теперь мы понимаем большую часть механизма, приводящего в движение *asyncio*. Но если бы мы ограничились селекторами для написания приложений, то пришлось бы реализовывать собственный цикл событий для получения той же функциональности, которую дает *asyncio*. Чтобы разобраться в том, как сделать это с помощью *asyncio*, возьмем все сделанное и переведем это на язык `async` / `await` и уже реализованного для нас цикла событий.

3.5 Эхо-сервер средствами цикла событий *asyncio*

Уровень `select` для большинства приложений является слишком низким. Возможно, мы хотим выполнять свой код в фоновом режиме, пока программа ждет поступления данных в сокет, или запускать фоновые задачи по расписанию. Попытавшись сделать это с помощью одних лишь селекторов, мы в итоге написали бы собственный цикл событий, в то время как *asyncio* предлагает уже готовый. Кроме того, сопрограммы и задачи предоставляют абстракции поверх селекторов, благодаря чему код становится проще писать и сопровождать, поскольку думать о селекторах вообще не нужно.

Теперь, когда мы лучше понимаем, как устроен цикл событий в *asyncio*, еще раз переделаем наш эхо-сервер, используя сопрограммы и задачи. Мы по-прежнему будем работать с низкоуровневыми сокетами, но на этот раз для управления ими воспользуемся API *asyncio*, которые возвращают сопрограммы. И немного расширим функциональность, чтобы продемонстрировать несколько ключевых концепций *asyncio*.

3.5.1 Сопрограммы цикла событий для сокетов

Поскольку сокеты – сравнительно низкоуровневое понятие, методы для работы с ними принадлежат самому циклу событий. Нам предстоит работать с тремя основными сопрограммами: `sock_accept`, `sock_recv` и `sock_sendall`. Это аналоги рассмотренных выше мето-

дов класса `socket`, только принимают они сокет в качестве аргумента и возвращают сопрограммы, которые могут ждать поступления данных с помощью `await`.

Начнем с `sock_accept`. Эта сопрограмма аналогична методу `socket.accept`, который мы видели в самой первой реализации. Она возвращает кортеж (структуру данных, в которой хранится упорядоченная последовательность значений), состоящий из сокета и адреса клиента. Мы передаем серверный сокет и ждем, когда сопрограмма что-то вернет. После этого мы будем иметь подключенный сокет и адрес. Сокет должен быть неблокирующим и уже привязанным к порту:

```
connection, address = await loop.sock_accept(socket)
```

Методы `sock_recv` и `sock_sendall` названы по аналогии с `sock_accept`. Они принимают сокет и ждут результата. `sock_recv` ждет поступления байтов в сокет. `sock_sendall` принимает сокет и данные, которые нужно отправить, после чего ждет, пока все данные будут отправлены. В случае успеха `sock_sendall` возвращает `None`.

```
data = await loop.sock_recv(socket)
success = await loop.sock_sendall(socket, data)
```

Располагая этими строительными блоками, мы можем перевести наши предыдущие подходы на язык сопрограмм и задач.

3.5.2 Проектирование асинхронного эхо-сервера

В главе 2 мы познакомились с сопрограммами и задачами. Так когда же в нашем эхо-сервере следует использовать сопрограмму, а когда нужно обернуть ее задачей? Чтобы принять решение, подумаем, как должно вести себя приложение.

Прежде всего решим, как следует прослушивать порт на предмет подключений. Прослушивая порт, мы можем обработать только одно подключение за раз, потому что `socket.accept` возвращает только одно клиентское подключение. Если входящих подключений несколько, то за кулисами они хранятся в очереди, но сейчас нам этот механизм неинтересен.

Поскольку нам не нужно обрабатывать несколько подключений конкурентно, одной сопрограммы, исполняющейся в бесконечном цикле, достаточно. Тогда другой код сможет работать конкурентно, пока эта сопрограмма приостановлена в ожидании подключения. Назовем эту сопрограмму `listen_for_connections`:

```
async def listen_for_connections(server_socket: socket,
                                loop: AbstractEventLoop):
    while True:
        connection, address = await loop.sock_accept(server_socket)
        connection.setblocking(False)
        print(f"Получен запрос на подключение от {address}")
```

Итак, сопрограмма для прослушивания порта у нас есть, а как насчет чтения и записи данных подключившимся клиентам? Для этого нужна просто сопрограмма или сопрограмма, обернутая задачей? В данном случае мы имеем несколько подключений, по каждому из которых в любой момент могут отправляться данные. Мы не хотим, чтобы ожидание данных по одному подключению блокировало все остальные, желательно иметь возможность читать и записывать данные для нескольких клиентов конкурентно. Поскольку требуется обрабатывать несколько подключений одновременно, имеет смысл создать для каждого подключения задачу, которая будет отвечать за чтение и запись данных.

Создадим сопрограмму `echo`, отвечающую за обработку данных, связанных с одним подключением. Эта сопрограмма будет крутиться в бесконечном цикле, ожидая данных от клиента. Получив данные, она будет отправлять их копию обратно клиенту.

Таким образом, в сопрограмме `listen_for_connections` мы для каждого нового подключения будем создавать задачу, обернутую сопрограммой `echo`. Эти две сопрограммы дают все, что нужно для построения асинхронного эхо-сервера.

Листинг 3.8 Построение асинхронного эхо-сервера

```
import asyncio
import socket
from asyncio import AbstractEventLoop

async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    while data := await loop.sock_recv(connection, 1024):
        await loop.sock_sendall(connection, data)

async def listen_for_connection(server_socket: socket,
                               loop: AbstractEventLoop):
    while True:
        connection, address = await loop.sock_accept(server_socket)
        connection.setblocking(False)
        print(f"Получен запрос на подключение от {address}")
        asyncio.create_task(echo(connection, loop))

async def main():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    server_address = ('127.0.0.1', 8000)
    server_socket.setblocking(False)
    server_socket.bind(server_address)
    server_socket.listen()

    await listen_for_connection(server_socket, asyncio.get_event_loop())
    asyncio.run(main())
```

В бесконечном цикле
ожидаем данных от клиента

Получив данные, отправляем
их обратно клиенту

После получения запроса на подключение создаем
задачу `echo`, ожидающую данные от клиента

Запускаем сопрограмму прослушивания
порта на предмет подключений

Архитектура программы показана на рис. 3.5. Имеется одна сопрограмма, `listen_for_connection`, прослушивающая порт. Как только клиент подключился, она запускает задачу `echo` для этого клиента, которая ожидает поступления данных и отправляет их обратно клиенту.

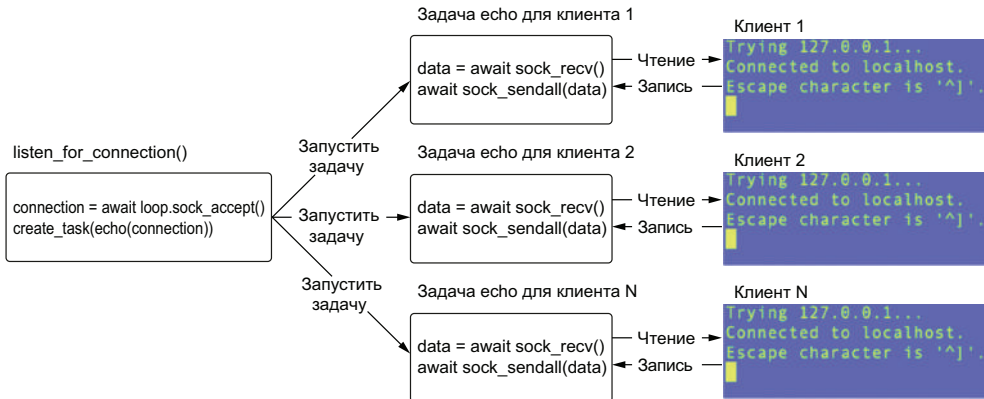


Рис. 3.5 Сопрограмма, прослушивающая порт, запускает по одной задаче для каждого нового подключения

Это приложение позволяет конкурентно прослушивать нескольких клиентов и отправлять им данные. Под капотом используются все те же селекторы, которые мы видели раньше, так что потребление процессора остается низким.

Мы написали полнофункциональный эхо-сервер, пользуясь только `asyncio`! Но нет ли в нашей реализации дефектов? Есть – мы не обрабатываем ошибки в задаче `echo`.

3.5.3 Обработка ошибок в задачах

Сетевые подключения часто ненадежны, поэтому в коде приложения могут возникать неожиданные исключения. Как должно вести себя приложение в случае ошибки чтения или записи данных? Для тестирования изменим нашу реализацию `echo` – будем возбуждать исключение, если клиент передает особое слово:

```
async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    while data := await loop.sock_recv(connection, 1024):
        if data == b'boom\r\n':
            raise Exception("Неожиданная ошибка сети")
        await loop.sock_sendall(connection, data)
```

Если теперь клиент пришлет слово «boom», то мы возбудим исключение и задача завершится аварийно. И что произойдет в этом случае? Мы увидим обратную трассу вызовов, содержащую предупреждение:

```

Task exception was never retrieved
future: <Task finished name='Task-2' coro=<echo() done, defined at
      asyncio_echo.py:5> exception=Exception('Неожиданная ошибка сети')>
Traceback (most recent call last):
  File "asyncio_echo.py", line 9, in echo
    raise Exception("Неожиданная ошибка сети")
Exception: Unexpected network error

```

Важна здесь фраза «Task exception was never retrieved» (Исключение задачи не извлекалось). Что она означает? Когда внутри задачи возникает исключение, считается, что задача завершена, а ее результатом является исключение. Это значит, что в стек вызовов исключение не попадает. И очистки здесь нет. Если это исключение возбуждается, то мы не можем отреагировать на ошибку в задаче, потому что не пытались его извлечь.

Чтобы исключение дошло до нас, задачу нужно вызывать в выражении `await`. В таком случае исключение будет возбуждено в точке `await`, и это отразится в трассе вызовов. Если не применить `await` к задаче в каком-то месте приложения, то мы рискуем никогда не увидеть возникшего исключения. Хотя в рассматриваемом примере мы видим сообщение, заданное в исключении, и можем подумать, что проблемы нет, это приложение можно немного изменить, так что сообщение будет от нас скрыто.

Для демонстрации предположим, что вместо игнорирования задач `echo`, созданных в `listen_for_connections`, мы сохраняем их в списке:

```

tasks = []

async def listen_for_connection(server_socket: socket,
                                loop: AbstractEventLoop):
    while True:
        connection, address = await loop.sock_accept(server_socket)
        connection.setblocking(False)
        print(f"Получено сообщение от {address}")
        tasks.append(asyncio.create_task(echo(connection, loop)))

```

На первый взгляд, все должно работать как прежде. Отправив сообщение «boom», мы должны увидеть исключение и предупреждение о том, что исключение не извлекалось. Однако это не так – мы ничего не увидим, пока не снимем приложение принудительно!

Все дело в том, что мы храним ссылку на задачу, а `asyncio` может напечатать сообщение и обратную трассу сбойной задачи, только когда она убирается в мусор. Это и понятно – ведь нет никакого способа узнать, не ожидают ли задачу в какой-то другой точке приложения, где она возбудила бы исключение. Из-за этих осложнений мы либо должны ждать задачи с помощью `await`, либо обрабатывать все исключения в самих задачах. Как поступить в нашем эхо-сервере?

Первое, что можно сделать, – обернуть код сопрограммы `echo` предположением `try/catch`, запротokolировать исключение и закрыть подключение:

```
import logging

async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    try:
        while data := await loop.sock_recv(connection, 1024):
            print('получены данные!')
            if data == b'boom\r\n':
                raise Exception("Неожиданная ошибка сети")
            await loop.sock_sendall(connection, data)
    except Exception as ex:
        logging.exception(ex)
    finally:
        connection.close()
```

Это решает неотложную проблему, из-за которой сервер ругается, что исключение не извлекалось, – ведь мы обработали его в самой сопрограмме. Кроме того, сокет корректно закрывается в блоке `finally`, так что в случае ошибки не остается висячего необработанного исключения.

Отметим, что в этой реализации все открытые подключения клиентов закрываются на этапе остановки приложения. Почему? В главе 2 мы говорили, что `asyncio.run` снимает все задачи, оставшиеся в момент остановки приложения. Мы также узнали, что, когда задача снимается, возбуждается исключение `CancelledError` в точке, где программа ждет ее с помощью `await`.

Здесь важно, где именно возбуждается исключение. Если наша задача ждет чего-то в предложении вида `await loop.sock_recv`, а мы эту задачу сняли, то `CancelledError` возбуждается в строке `await loop.sock_recv`. Это значит, что в рассматриваемом случае будет выполнен блок `finally`, так как исключение было возбуждено в выражении `await` при снятии задачи. Если изменить блок `except`, так чтобы он перехватывал и протоколировал эти исключения, то мы увидим по одному исключению `CancelledError` на каждую задачу.

Итак, мы решили проблему обработки ошибок при отказе задач. А что, если требуется произвести какую-то очистку после ошибок или что-то сделать с задачами, оставшимися в момент остановки приложения? Для этой цели предназначены обработчики сигналов `asyncio`.

3.6 Корректная остановка

Итак, мы написали эхо-сервер, который обрабатывает несколько конкурентных подключений, протоколирует ошибки и производит очистку в случае исключения. Но что произойдет, если нам понадобится остановить приложение? Наверное, стоило бы завершить обработку уже отправленных сообщений? Для этого можно добавить специальную логику остановки, которая дает несколько секунд на

завершение начатых задач, чтобы они отправили все сообщения, которые собирались. Хотя наша реализация не дотянет до промышленного качества, мы сможем изучить вещи, относящиеся к остановке приложений `asyncio` и отмене всех работающих задач.

Сигналы в Windows

Windows не поддерживает сигналы. Поэтому настоящий раздел относится только к системам на базе Unix. В Windows имеется другая система, которая на момент написания этой книги плохо работает с Python. Если хотите узнать, как сделать этот код кросс-платформенным, почитайте ответ на вопрос на сайте Stack Overflow по адресу <https://stackoverflow.com/questions/35772001>.

3.6.1 Прослушивание сигналов

Сигналы в Unix-системах – это механизм асинхронного уведомления процесса о событии, случившемся на уровне операционной системы. Несмотря на кажущуюся низкоуровневость сигналов, с некоторыми из них вы, безусловно, знакомы. Например, хорошо известен сигнал `SIGINT` (*сигнал прерывания*). Он посылается при нажатии **CTRL+C** для снятия командного приложения. В Python мы можем обработать его, перехватив исключение `KeyboardInterrupt`. Еще один известный сигнал – `SIGTERM` (*сигнал завершения*). Он посылается, когда мы выполним команду `kill` для снятия процесса.

Чтобы реализовать специальную логику остановки, мы включим в приложение обработчики сигналов `SIGINT` и `SIGTERM`, в которых дадим задачам *echo* несколько секунд для завершения.

Как прослушивать сигналы в приложении? Цикл событий `asyncio` позволяет прослушивать любой сигнал, указанный в методе `add_signal_handler`. Это не то же самое, что задание обработчиков сигналов с помощью функции `signal.signal` из модуля `signal`, так как `add_signal_handler` безопасно взаимодействует с циклом событий. Функция принимает номер сигнала и функцию, которая должна вызываться при получении этого сигнала. Для демонстрации добавим обработчик сигнала, который снимает все работающие задачи. В `asyncio` есть функция `asyncio.all_tasks`, возвращающая множество всех работающих задач.

Листинг 3.9 Добавление обработчика сигнала, снимающего все задачи

```
import asyncio, signal
from asyncio import AbstractEventLoop
from typing import Set

from util.delay_functions import delay
```

```
def cancel_tasks():
    print('Получен сигнал SIGINT!')
    tasks: Set[asyncio.Task] = asyncio.all_tasks()
    print(f'Снимается {len(tasks)} задач.')
    [task.cancel() for task in tasks]

async def main():
    loop: AbstractEventLoop = asyncio.get_running_loop()

    loop.add_signal_handler(signal.SIGINT, cancel_tasks)

    await delay(10)

asyncio.run(main())
```

Запустив это приложение, мы увидим, что сопрограмма `delay` начинает работать сразу и ждет 10 с. Если в течение этих 10 с нажать **CTRL+C**, то мы увидим сообщение «Получен сигнал SIGINT!», а вслед за ним сообщение о том, что снимаются все задачи. Кроме того, мы увидим, что в `asyncio.run(main())` возбуждено исключение `CancellationError`, поскольку мы сняли задачу.

3.6.2 Ожидание завершения начатых задач

В исходной постановке мы хотели, чтобы наш эхо-сервер давал задачам эхо несколько секунд на завершение перед остановкой. Это можно сделать, обернув задачи функцией `wait_for` и ожидая обернутые задачи с помощью `await`. По истечении тайм-аута эти задачи возбуждают исключение `TimeoutError`, и мы сможем завершить приложение.

Говоря об обработчике остановки, следует отметить, что это обычная функция Python, поэтому использовать внутри нее предложения `await` нельзя. Это проблема, так как предложенное решение подразумевает использование `await`. Возможное решение – создать сопрограмму, которая реализует логику остановки, и в обработчике остановки обернуть ее задачей:

```
async def await_all_tasks():
    tasks = asyncio.all_tasks()
    [await task for task in tasks]

async def main():
    loop = asyncio.get_event_loop()
    loop.add_signal_handler(signal.SIGINT,
                           lambda: asyncio.create_task(await_all_tasks()))
```

Такой подход будет работать, но у него есть недостаток: если внутри `await_all_tasks` возникнет исключение, то мы останемся с брошенной отказавшей задачей и, стало быть, с предупреждением «исключение не извлекалось». Может быть, есть способ лучше?

Можно решить проблему, возбудив специальное исключение, которое будет останавливать нашу сопрограмму `main`. Тогда мы сможем

перехватить его внутри `main` и выполнить логику остановки. Для этого придется создать собственный цикл событий взамен `asyncio.run`. Дело в том, что при возникновении исключения `asyncio.run` снимает все работающие задачи, а значит, мы не сможем обернуть задачи `echo` функцией `wait_for`.

```
class GracefulExit(SystemExit):
    pass

def shutdown():
    raise GracefulExit()

loop = asyncio.get_event_loop()

loop.add_signal_handler(signal.SIGINT, shutdown)

try:
    loop.run_until_complete(main())
except GracefulExit:
    loop.run_until_complete(close_echo_tasks(echo_tasks))
finally:
    loop.close()
```

Имея в виду этот подход, напомним логику остановки:

```
async def close_echo_tasks(echo_tasks: List[asyncio.Task]):
    waiters = [asyncio.wait_for(task, 2) for task in echo_tasks]
    for task in waiters:
        try:
            await task
        except asyncio.exceptions.TimeoutError:
            # Здесь мы ожидаем истечения тайм-аута
            pass
```

В сопрограме `close_echo_tasks` мы принимаем список задач `echo` и обортываем каждую из них задачей `wait_for` с двухсекундным тайм-аутом. Это означает, что любая задача `echo` будет иметь 2 с на завершение перед принудительным снятием. После этого в цикле ожидаем все обернутые задачи с помощью `await`. Мы перехватываем все исключения `TimeoutError`, поскольку ожидаем, что они будут возбуждены нашими задачами по истечении 2 с. Собирая все вместе, получаем такой код логики остановки эхо-сервера.

Листинг 3.10 Корректная остановка

```
import asyncio
from asyncio import AbstractEventLoop
import socket
import logging
import signal
from typing import List
```

```

async def echo(connection: socket,
               loop: AbstractEventLoop) -> None:
    try:
        while data := await loop.sock_recv(connection, 1024):
            print('got data!')
            if data == b'boom\r\n':
                raise Exception("Неожиданная ошибка сети")
            await loop.sock_sendall(connection, data)
    except Exception as ex:
        logging.exception(ex)
    finally:
        connection.close()

echo_tasks = []

async def connection_listener(server_socket, loop):
    while True:
        connection, address = await loop.sock_accept(server_socket)
        connection.setblocking(False)
        print(f"Получено сообщение от {address}")
        echo_task = asyncio.create_task(echo(connection, loop))
        echo_tasks.append(echo_task)

class GracefulExit(SystemExit):
    pass

def shutdown():
    raise GracefulExit()

async def close_echo_tasks(echo_tasks: List[asyncio.Task]):
    waiters = [asyncio.wait_for(task, 2) for task in echo_tasks]
    for task in waiters:
        try:
            await task
        except asyncio.exceptions.TimeoutError:
            # Здесь мы ожидаем истечения тайм-аута
            pass

async def main():
    server_socket = socket.socket()
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    server_address = ('127.0.0.1', 8000)
    server_socket.setblocking(False)
    server_socket.bind(server_address)
    server_socket.listen()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(getattr(signal, signame), shutdown)
    await connection_listener(server_socket, loop)

loop = asyncio.new_event_loop()

try:

```

```
loop.run_until_complete(main())
except GracefulExit:
    loop.run_until_complete(close_echo_tasks(echo_tasks))
finally:
    loop.close()
```

Если остановить приложение нажатием **CTRL+C** или командой **kill** в момент, когда подключен хотя бы один клиент, то будет выполнена логика остановки. Мы увидим, что приложение ждет 2 с, давая задачам `echo` возможность завершиться, а затем останавливается.

Есть две причины, по которым эта логика несовершенна. Во-первых, ожидая завершения задач `echo`, мы не останавливаем прослушиватель подключений. Это значит, что, пока мы ждем, может поступить новый запрос на подключение, для которого мы не сможем добавить двухсекундный тайм-аут. Во-вторых, мы ждем завершения всех задач `echo`, но перехватываем только исключения `TimeoutException`. Следовательно, если задача возбудит какое-то другое исключение, мы его запомним, а все последующие задачи, возбуждающие исключение, будут проигнорированы. В главе 4 мы рассмотрим методы `asyncio` для более правильной обработки исключений в группе допускающих ожидание объектов.

Хотя приложение несовершенно и является лишь модельным примером, мы все же построили полнофункциональный сервер с помощью библиотеки `asyncio`. Этот сервер может конкурентно обрабатывать много пользователей в одном потоке. Если бы мы использовали блокирующий подход, как в самом начале обсуждения, то должны были бы прибегнуть к многопоточности, из-за чего программа стала бы сложнее и потребляла больше ресурсов.

Резюме

В этой главе мы узнали о блокирующих и неблокирующих сокетах и более глубоко изучили работу цикла событий `asyncio`. Мы также написали свое первое приложение `asyncio`, эхо-сервер с высокой степенью конкурентности. Мы поняли, как обрабатывать ошибки в задачах и добавлять собственную логику остановки приложения.

- Мы узнали, как создать простое приложение с блокирующими сокетами. Блокирующий сокет останавливает весь поток, пока ожидает данных. Это препятствует организации конкурентности, потому что в каждый момент времени мы можем получать данные только от одного клиента.
- Мы узнали, как писать приложения с неблокирующими сокетами. Их методы возвращают управление немедленно – либо в результате получения данных, либо с исключением, если данные еще не пришли. Поэтому такие сокеты позволяют обеспечить конкурентность.

- Мы описали, как использовать модуль `selectors` для эффективного прослушивания событий в сокетах. Он позволяет зарегистрировать сокеты, за которыми мы хотим наблюдать, и сообщает, когда в неблокирующем сокее появляются данные.
- Поместив `select` в бесконечный цикл, мы заново реализуем основную часть функциональности цикла событий. Регистрируем интересные нас сокеты и в цикле выполняем свой код при появлении в сокее данных.
- Мы узнали, как использовать методы цикла событий *asynсio* для построения приложений с неблокирующими сокетами. Они принимают сокет и возвращают сопрограмму, которую можно затем использовать в выражении `await`. Это приостанавливает родительскую сопрограмму, до тех пор пока в сокее не появятся данные. Под капотом используется модуль `selectors`.
- Мы видели, как наши задачи реализуют конкурентность асинхронного эхо-сервера, способного отправлять и принимать данные одновременно от нескольких клиентов. Также рассмотрели, как обрабатывать ошибки в этих задачах.
- Мы узнали, как добавить в асинхронное приложение специальную логику остановки. В нашем конкретном примере мы решили дать начатым задачам несколько секунд на завершение отправки данных. Но никто не мешает сделать что-то другое.

Конкурентные веб-запросы

Краткое содержание главы

- Асинхронные контекстные менеджеры.
- Отправка совместимых с `asyncio` веб-запросов с помощью библиотеки `aiohttp`.
- Конкурентное выполнение веб-запросов с помощью `gather`.
- Обработка результатов по мере поступления с помощью `as_completed`.
- Отслеживание незавершенных запросов с помощью `wait`.
- Установка и обработка тайм-аутов для групп запросов и снятие запросов.

В главе 3 мы узнали о механизмах работы сокетов и написали простой эхо-сервер. Научившись проектировать базовое приложение, мы теперь применим эти знания к реализации конкурентных неблокирующих веб-запросов. Использование для этой цели `asyncio` позволит одновременно отправлять сотни запросов, резко сократив временные затраты по сравнению с синхронным подходом. Это полезно, когда нужно отправить несколько запросов различным функциям REST API, как часто бывает при работе с микросервисной архитектурой или при реализации веб-робота. Кроме того, при таком подходе мы сможем выполнять другой код, ожидая завершения потенциального долгого веб-запроса, так что приложение получится более отзывчивым.

В этой главе мы узнаем об асинхронной библиотеке *aiohttp*, которая позволяет сделать все вышеописанное. В ней для отправки веб-запросов используются неблокирующие сокеты, а результатом вызова являются сопрограммы, которые можно затем ждать с помощью `await`. Конкретно мы возьмем список, содержащий сотни URL, содержимое которых хотим получить, и запустим для них запросы одновременно. По ходу дела познакомимся с различными методами API `asyncio`, предназначенными для одновременного выполнения сопрограмм, и сможем выбрать между ожиданием завершения всех сопрограмм или обработкой результатов по мере поступления. Мы также посмотрим, как задавать тайм-ауты, – индивидуально и для целой группы запросов. Мы увидим, как снять множество исполняемых запросов в зависимости от результатов выполнения других запросов. Эти методы API полезны не только для веб-запросов, но и всегда, когда требуется конкурентно выполнить группу сопрограмм. Будем использовать эти функции на протяжении всей книги, а вы, как разработчик на платформе `asyncio`, часто будете применять их в собственной практике.

4.1 Введение в *aiohttp*

В главе 2 мы говорили, что начинающие изучать `asyncio` часто сталкиваются с проблемой: взять уже имеющийся код и понавтыкать в него `async` и `await` в надежде улучшить производительность. Чаще всего, в частности для веб-запросов, это работать не будет, поскольку большинство существующих библиотек – блокирующие.

Популярной библиотекой для работы с веб-запросами является `requests`. Она плохо совместима с `asyncio`, поскольку в ней используются блокирующие сокеты. Это значит, что отправка любого запроса блокирует поток, в котором запрос отправлен, а поскольку `asyncio` однопоточная, будет блокирован весь цикл событий, пока запрос не завершится.

Чтобы решить эту проблему и добиться конкурентности, нам нужна библиотека, которая не блокирует ничего вплоть до уровня сокетов. Одной из таких библиотек является *aiohttp* (асинхронный HTTP на стороне клиента и сервера для `asyncio` и Python), которая решает проблему с помощью неблокирующих сокетов.

Aiohttp – библиотека с открытым исходным кодом, часть проекта *aio-lib*s, в описании которого декларируется, что это «набор основанных на `asyncio` высококачественных библиотек» (см. <https://github.com/aio-lib>). Она представляет собой полнофункциональный веб-клиент и веб-сервер, т. е. умеет отправлять веб-запросы и может стать основой для разработки асинхронных веб-серверов. (Документация доступна по адресу <https://docs.aiohttp.org/>.) В этой главе мы сконцентрируемся на клиентской части *aiohttp*, но далее рассмотрим и написание веб-серверов с ее помощью.

Итак, с чего начать? Прежде всего нужно научиться отправлять HTTP-запрос. Сначала следует рассказать о новом синтаксисе асинхронных контекстных менеджеров, который позволяет корректно начинать и закрывать HTTP-сеансы. Разработчики на платформе `asyncio` часто используют этот синтаксис для асинхронного захвата ресурсов, например подключений к базе данных.

4.2 Асинхронные контекстные менеджеры

В любом языке программирования приходится работать с ресурсами, которые нужно открывать, а затем закрывать, например с файлами. При этом нужно помнить об исключениях – если исключение возникает, когда ресурс открыт, то может не представиться возможности для очистки, и в результате мы будем иметь утечку ресурсов. В Python эта проблема решается просто, с использованием блока `finally`. Следующий пример написан не вполне в духе Python, тем не менее файл всегда закрывается, даже если возникло исключение:

```
file = open('example.txt')

try:
    lines = file.readlines()
finally:
    file.close()
```

Здесь гарантируется, что описатель файла не останется открытым, если при выполнении функции `file.readlines` возникнет исключение. Недостаток в том, что нужно не забыть обернуть код конструкцией `try/finally` и вызвать правильный метод закрытия ресурса. Для файлов это нетрудно, нужно только вызывать для них метод `close`, но все же хотелось бы иметь что-то допускающее повторное использование, особенно с учетом того, что очистка может не сводиться к вызову одного метода. В Python для этой цели предусмотрены *контекстные менеджеры*. Они позволяют абстрагировать логику освобождения ресурса с помощью блока `try/finally`:

```
with open('example.txt') as file:
    lines = file.readlines()
```

Этот «питонический» способ управления файлами выглядит куда чище. Если внутри блока `with` возникнет исключение, файл автоматически будет закрыт. Это работает для синхронных ресурсов, но что делать, если мы хотим применить этот механизм асинхронно? В таком случае контекстный менеджер не подходит, потому что предназначен только для работы с синхронным Python-кодом, а не с сопрограммами и задачами. По этой причине в язык было включено новое средство: *асинхронные контекстные менеджеры*. Синтаксис почти такой же, только вместо `with` нужно писать `async with`.

Асинхронный контекстный менеджер – это класс, реализующий два специальных метода-сопрограммы: `__aenter__`, который асинхронно захватывает ресурс, и `__aexit__`, который закрывает ресурс. Сопрограмма `__aexit__` принимает несколько аргументов, относящихся к обработке исключений, их мы в этой главе рассматривать не будем.

Чтобы лучше разобраться, реализуем асинхронный контекстный менеджер для сокетов, с которыми познакомились в главе 3. Подключенный клиентский сокет можно рассматривать как управляемый ресурс. Когда клиент подключается, мы захватываем клиентский сокет. А закончив работу с ним, очищаем и закрываем соединение. В главе 3 мы обернули все это блоком `try/finally`, но могли бы вместо этого реализовать асинхронный контекстный менеджер.

Листинг 4.1 Асинхронный контекстный менеджер, ожидающий подключения клиента

```
import asyncio
import socket
from types import TracebackType
from typing import Optional, Type

class ConnectedSocket:

    def __init__(self, server_socket):
        self._connection = None
        self._server_socket = server_socket

    async def __aenter__(self):
        print('Вход в контекстный менеджер, ожидание подключения')
        loop = asyncio.get_event_loop()
        connection, address = await loop.sock_accept(self._server_socket)
        self._connection = connection
        print('Подключение подтверждено')
        return self._connection

    async def __aexit__(self,
                       exc_type: Optional[Type[BaseException]],
                       exc_val: Optional[BaseException],
                       exc_tb: Optional[TracebackType]):
        print('Выход из контекстного менеджера')
        self._connection.close()
        print('Подключение закрыто')

    async def main():
        loop = asyncio.get_event_loop()

        server_socket = socket.socket()
        server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        server_address = ('127.0.0.1', 8000)
        server_socket.setblocking(False)
```

Эта сопрограмма вызывается при входе в блок `with`. Она ждет подключения клиента и возвращает это подключение

Эта сопрограмма вызывается при выходе из блока `with`. В ней мы производим очистку ресурса, в данном случае закрывается подключение

```

server_socket.bind(server_address)
server_socket.listen()

async with ConnectedSocket(server_socket) as connection:
    data = await loop.sock_recv(connection, 1024)
    print(data)
asyncio.run(main())

```

Здесь вызывается `__aenter__` и начинается ожидание подключения

После этого предложения вызывается `__aexit__` и подключение закрывается

Здесь мы создали асинхронный контекстный менеджер `ConnectedSocket`. Этот класс принимает серверный сокет, и в сопрограмме `__aenter__` мы ждем подключения клиента. Как только клиент подключился, возвращается соответствующее ему клиентское подключение. Это дает нам доступ к подключению в части `as` предложения `async with`. Затем в блоке `async with` мы используем это подключение для ожидания данных от клиента. Когда выполнение этого блока завершается, вызывается сопрограмма `__aexit__`, которая закрывает подключение. Если клиент подключился с помощью telnet и отправил какие-то тестовые данные, то при работе этой программы мы увидим такие сообщения:

```

Вход в контекстный менеджер, ожидание подключения
Подключение подтверждено
b'test\r\n'
Выход из контекстного менеджера
Подключение закрыто

```

В `aiohttp` асинхронные контекстные менеджеры всю используют для создания HTTP-сеансов и подключений. Мы также будем использовать их в главе 5 при работе с асинхронными подключениями к базе данных и транзакциями. Обычно писать асинхронные контекстные менеджеры самостоятельно не возникает необходимости, но полезно понимать, как они работают и чем отличаются от обычных контекстных менеджеров. Познакомившись с контекстными менеджерами, воспользуемся ими в сочетании с `aiohttp`, чтобы отправить асинхронный веб-запрос.

4.2.1 Отправка веб-запроса с помощью `aiohttp`

Сначала нужно установить библиотеку `aiohttp`. Это можно сделать с помощью программы `pip`:

```
pip install -Iv aiohttp==3.8.1
```

Будет установлена последняя версия `aiohttp` (на момент написания книги 3.8.1). После этого можно приступить к отправке запросов.

В библиотеке `aiohttp` и вообще при работе с веб-запросами используется понятие *сеанса*. Сеанс можно рассматривать как создание нового окна браузера. В этом окне можно открывать разные веб-страницы, которые могут посылать куки, сохраняемые браузером.

Внутри сеанса хранится много открытых подключений, их можно при необходимости использовать повторно. Это называется *пулом подключений* и играет важную роль в производительности приложений на базе aiohttp. Поскольку создание подключения – дорогостоящее действие, наличие пула повторно используемых подключений сокращает затраты на выделение и освобождение ресурсов. Сеанс также самостоятельно сохраняет все полученные куки.

Как правило, мы хотим пользоваться преимуществами пула подключений, поэтому в большинстве приложений на базе aiohttp создается один сеанс для всего приложения. Затем объект сеанса передается методам. У объекта сеанса имеются методы для отправки веб-запросов, в том числе GET, PUT и POST. Для создания сеанса используется синтаксис `async with` и асинхронный контекстный менеджер `aiohttp.ClientSession`.

Листинг 4.2 Отправка веб-запроса с помощью aiohttp

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from util import async_timed

@async_timed()
async def fetch_status(session: ClientSession, url: str) -> int:
    async with session.get(url) as result:
        return result.status

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https://www.example.com'
        status = await fetch_status(session, url)
        print(f'Состояние для {url} было равно {status}')

asyncio.run(main())
```

При выполнении этой программы мы увидим сообщение «Состояние для `http://www.example.com` было равно 200». Сначала мы создали клиентский сеанс в блоке `async with`, вызвав функцию `aiohttp.ClientSession()`. Имея сеанс, можно отправить любой веб-запрос. В данном случае мы написали вспомогательную функцию `fetch_status_code`, которая получает сеанс и URL-адрес, а возвращает состояние для этого адреса. В этой функции имеется еще один блок `async with`, а в сеансе выполняется GET-запрос с указанным URL-адресом. Результат можно обработать в том же блоке `with`. В данном случае мы просто извлекаем и возвращаем код состояния.

Отметим, что по умолчанию в сеансе `ClientSession` можно создать не более 100 подключений, что дает неявную верхнюю границу количества конкурентных веб-запросов. Чтобы изменить этот предел, мож-

но создать экземпляр класса `TCPConnector`, входящего в состав `aiohttp`, указав максимальное число подключений, и передать его конструктору `ClientSession`. Подробнее см. документацию по `aiohttp` по адресу https://docs.aiohttp.org/en/stable/client_advanced.html#connectors.

Функция `fetch_status` еще не раз понадобится нам в этой главе, поэтому сделаем ее повторно используемой. Создайте модуль Python `chapter_04` и поместите в его файл `__init__.py` эту функцию. В последующих примерах мы будем импортировать ее в предложении `from chapter_04 import fetch_status`.

Замечание для пользователей Windows

В настоящее время в версии `aiohttp` для Windows есть ошибка, из-за которой иногда печатается сообщение «`RuntimeError: Event loop is closed`», хотя приложение работает нормально. Подробнее об этой ошибке можно прочитать по адресу <https://github.com/aio-libs/aiohttp/issues/4324> и <https://bugs.python.org/issue39232>. Чтобы обойти ее, нужно либо управлять циклом событий вручную с помощью функции `asyncio.get_event_loop().run_until_complete(main())`, как описано в главе 2, либо изменить политику цикла событий, вызвав функцию `asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())` до вызова `asyncio.run(main())`.

4.2.2 Задание тайм-аутов в `aiohttp`

Выше мы видели, как задать тайм-аут для объекта, допускающего ожидание, с помощью функции `asyncio.wait_for`. Так можно задать тайм-аут и для запроса в `aiohttp`, но есть более чистый способ – воспользоваться функциональностью, изначально присутствующей в `aiohttp`. По умолчанию тайм-аут равен 5 мин., т. е. никакая операция не будет выполняться дольше. Это большой тайм-аут, и многие разработчики предпочитают его уменьшить. Тайм-аут можно задавать на уровне сеанса, тогда он будет применяться к каждой операции, или на уровне запроса, если требуется более точное управление.

Тайм-аут задается с помощью структуры данных `aiohttp ClientTimeout`. Она позволяет установить не только общий тайм-аут в секундах для всего запроса, но также отдельные тайм-ауты для установления соединения или чтения данных. Рассмотрим, как задать тайм-аут для сеанса и для одного запроса.

Листинг 4.3 Задание тайм-аутов в `aiohttp`

```
import asyncio
import aiohttp
from aiohttp import ClientSession

async def fetch_status(session: ClientSession,
                       url: str) -> int:
```

```

ten_millis = aiohttp.ClientTimeout(total=.01)
async with session.get(url, timeout=ten_millis) as result:
    return result.status

async def main():
    session_timeout = aiohttp.ClientTimeout(total=1, connect=.1)
    async with aiohttp.ClientSession(timeout=session_timeout) as session:
        await fetch_status(session, 'https://example.com')

asyncio.run(main())

```

Здесь задается два тайм-аута. Первый – на уровне клиентского сеанса, полный тайм-аут равен 1 с, а тайм-аут для установления соединения – 100 мс. Затем в функции `fetch_status` мы переопределяем этот тайм-аут для нашего GET-запроса, задавая его равным 10 мс. Если запрос к `example.com` займет более 10 мс, то будет возбуждено исключение `asyncio.TimeoutError` в точке `await fetch_status`. В этом примере 10 мс должно хватить для завершения запроса к example.com, так что исключения мы не увидим. Чтобы протестировать исключение, измените URL-адрес, выбрав страницу, скачивание которой занимает больше 10 мс.

Эти примеры демонстрируют основы `aiohttp`. Однако наше приложение ничего не выиграет от выполнения одного запроса с помощью `asyncio`. Выигрыш мы получим, только если будем выполнять несколько веб-запросов конкурентно.

4.3 И снова о конкурентном выполнении задач

В первых главах книги мы научились создавать несколько задач для конкурентного выполнения сопрограмм. Для этого мы вызывали функцию `asyncio.create_task`, а затем ждали завершения задачи, как показано ниже:

```

import asyncio

async def main() -> None:
    task_one = asyncio.create_task(delay(1))
    task_two = asyncio.create_task(delay(2))

    await task_one
    await task_two

```

Такая тактика работает в простых случаях, когда имеется всего одна или две конкурентно запускаемые сопрограммы. Но если мы хотим конкурентно выполнить сотни, тысячи, а то и больше веб-запросов, то код становится слишком многословным.

Возникает соблазн воспользоваться циклом `for` или списковым включением, чтобы сделать код более лаконичным, как показано

в листинге ниже. Но такой подход может оказаться проблематичным, если реализовать его неправильно.

Листинг 4.4 Неправильное использование спискового включения для создания и ожидания задач

```
import asyncio
from util import async_timed, delay

@async_timed()
async def main() -> None:
    delay_times = [3, 3, 3]
    [await asyncio.create_task(delay(seconds)) for seconds in delay_times]
    asyncio.run(main())
```

Поскольку в идеале мы хотим, чтобы все задачи `delay` выполнялись конкурентно, ожидается, что метод `main` будет работать примерно 3 с. Однако же он работает 9 с, так как все выполняется последовательно.

```
выполняется <function main at 0x10f14a550> с аргументами () {}
выполняется <function delay at 0x10f7684c0> с аргументами (3,) {}
засыпаю на 3 с
сон в течение 3 с закончился
<function delay at 0x10f7684c0> завершилась за 3.0008 с
выполняется <function delay at 0x10f7684c0> с аргументами (3,) {}
засыпаю на 3 с
сон в течение 3 с закончился
<function delay at 0x10f7684c0> завершилась за 3.0009 с
выполняется <function delay at 0x10f7684c0> с аргументами (3,) {}
засыпаю на 3 с
сон в течение 3 с закончился
<function delay at 0x10f7684c0> завершилась за 3.0020 с
<function main at 0x10f14a550> завершилась за 9.0044 с
```

Здесь имеется тонкая ошибка. Все дело в том, что мы применяем `await` сразу же после создания задачи. Это значит, что мы приостанавливаем списковое включение и сопрограмму `main` для каждой созданной задачи `delay` до момента, когда она завершится. В данном случае в каждый момент времени будет работать только одна задача, а не все конкурентно. Исправить это легко, хотя немного муторно. Мы можем создавать задачи в одном списковом включении, а ждать в другом. Тогда все будет работать конкурентно.

Листинг 4.5 Использование спискового включения для конкурентного выполнения задач

```
import asyncio
from util import async_timed, delay

@async_timed()
async def main() -> None:
```

```

delay_times = [3, 3, 3]
tasks = [asyncio.create_task(delay(seconds)) for seconds in delay_times]
[await task for task in tasks]

asyncio.run(main())

```

Здесь создается сразу несколько задач, которые запоминаются в списке `tasks`. Создав все задачи, мы ждем их завершения в другом списковом включении. Это работает, потому что функция `create_task` возвращает управление немедленно, и мы ничего не ждем, до тех пор пока все задачи не будут созданы. Таким образом, будет затрачено время, равное максимальной задержке в списке `delay_times`, т. е. общее время работы составит приблизительно 3 с.

```

выполняется <function main at 0x10d4e1550> с аргументами () {}
выполняется <function delay at 0x10daff4c0> с аргументами (3,) {}
засыпаю на 3 с
выполняется <function delay at 0x10daff4c0> с аргументами (3,) {}
засыпаю на 3 с
выполняется <function delay at 0x10daff4c0> с аргументами (3,) {}
засыпаю на 3 с
сон в течение 3 с закончился
<function delay at 0x10daff4c0> завершилась за 3.0029 с
сон в течение 3 с закончился
<function delay at 0x10daff4c0> завершилась за 3.0029 с
сон в течение 3 с закончился
<function delay at 0x10daff4c0> завершилась за 3.0029 с
<function main at 0x10d4e1550> завершилась за 3.0031 с

```

Это код делает то, что нам нужно, но недостатки еще имеются. И первый из них заключается в том, что он состоит из нескольких строк, и мы должны не забыть отделить создание задач от ожидания. Второй – негибкость, поскольку если одна из наших сопрограмм завершается значительно раньше прочих, то мы застрянем во втором списковом включении, ожидая завершения всех остальных сопрограмм. Иногда это приемлемо, но чаще мы стремимся к большей отзывчивости и хотим обрабатывать результаты сразу после поступления. Третий и, пожалуй, самый крупный недостаток связан с обработкой исключений. Если в какой-то сопрограмме возникает исключение, то оно будет возбуждено в момент ожидания сбоями задачи. Это значит, что мы не сможем обработать успешно завершившиеся задачи, потому что одно-единственное исключение останавливает всю работу.

В `asyncio` есть функции для таких ситуаций и не только. Их рекомендуется использовать при конкурентном выполнении нескольких задач. В следующих разделах мы рассмотрим некоторые из них и покажем, как с ними работать в контексте конкурентной отправки нескольких веб-запросов.

4.4 Конкурентное выполнение запросов с помощью `gather`

Для конкурентного выполнения допускающих ожидание объектов широко используется функция `asyncio.gather`. Она принимает последовательность допускающих ожидание объектов и запускает их конкурентно всего в одной строке кода. Если среди объектов есть сопрограмма, то `gather` автоматически обортывает ее задачей, чтобы гарантировать конкурентное выполнение. Это значит, что не нужно отдельно обортывать все сопрограммы по отдельности с помощью функции `asyncio.create_task`, как мы делали раньше.

`asyncio.gather` возвращает объект, допускающий ожидание. Если использовать его в выражении `await`, то выполнение будет приостановлено, пока не завершатся все переданные объекты. А когда это произойдет, `asyncio.gather` вернет список результатов работы.

Мы можем воспользоваться этой функцией, чтобы конкурентно отправить любое число веб-запросов. Рассмотрим пример, где имеется 1000 запросов и мы хотим получить все коды состояния. Снабдим сопрограмму `main` декоратором `@async_timed`, чтобы знать, сколько прошло времени.

Листинг 4.6 Конкурентное выполнение запросов с помощью `gather`

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from chapter_04 import fetch_status
from util import async_timed

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com' for _ in range(1000)]
        requests = [fetch_status(session, url) for url in urls]
        status_codes = await asyncio.gather(*requests)
        print(status_codes)

asyncio.run(main())
```

Сгенерировать список сопрограмм
для каждого запроса, который мы
хотим отправить

Ждать завершения
всех запросов

Здесь мы сначала генерируем список URL-адресов, для которых хотим получить код состояния; для простоты все адреса равны `example.com`. Затем берем этот список и вызываем `fetch_status_code`, чтобы получить список сопрограмм, который передадим `gather`. При этом все сопрограммы обортываются задачами и запускаются конкурентно. Во время выполнения на стандартном выводе будет напечатано 1000 сообщений, показывающих, что сопрограммы `fetch_status_code`

запущены последовательно, но запросы выполняются конкурентно. Когда придут все результаты, мы увидим сообщение вида «<function fetch_status_code at 0x10f3fe3a0> завершилась за 0.5453 с». После того как содержимое всех запрошенных URL-адресов будет получено, начнут печататься коды состояния. Продолжительность всего процесса зависит от скорости интернет-подключения и быстродействия компьютера, обычно скрипт завершается за 500–600 мс.

А если сравнить с синхронным выполнением? Функцию `main` легко модифицировать, так чтобы она ждала `fetch_status_code` с помощью `await`, т. е. блокировала выполнение при каждом запросе. Так мы заставим работать ее синхронно.

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com' for _ in range(1000)]
        status_codes = [await fetch_status_code(session, url) for url in urls]
        print(status_codes)
```

Теперь программа будет работать гораздо дольше. Кроме того, вместо 1000 сообщений «выполняется `function fetch_status_code`», за которыми следует 1000 сообщений «`function fetch_status_code` завершилась», мы увидим следующую пару строк для каждого запроса:

```
выполняется <function delay at 0x10d95b310> с аргументами (3,) {}
<function delay at 0x10d95b310> завершилась за 0.01884 с
```

Это означает, что запросы следуют друг за другом и каждый начинается только после того, как предыдущий вызов `fetch_status_code` завершился. Так насколько же это медленнее асинхронной версии? Зависит от интернет-подключения и машины, на которой запущена программа, но приблизительно выполнение занимает 18 с. То есть асинхронная версия работает в 33 раза быстрее. Впечатляет!

Стоит отметить, что порядок поступления результатов для переданных объектов, допускающих ожидание, не детерминирован. Например, если передать `gather` сопрограммы `a` и `b` именно в таком порядке, то `b` может завершиться раньше, чем `a`. Но приятная особенность `gather` заключается в том, что, независимо от порядка завершения допускающих ожидание объектов, результаты гарантированно будут возвращены в том порядке, в каком объекты передавались. Продemonстрируем это в описанном ранее сценарии использования функции `delay`.

Листинг 4.7 Завершение допускающих ожидание объектов не по порядку

```
import asyncio
from util import delay

async def main():
```

```
results = await asyncio.gather(delay(3), delay(1))
print(results)

asyncio.run(main())
```

Здесь мы передали *gather* две сопрограммы. Первой для завершения требуется 3 с, второй – одна. Можно ожидать, что результатом будет список [1, 3], потому что односекундная сопрограмма завершается раньше трехсекундной. Но на самом деле возвращается [3, 1] – в том порядке, в каком сопрограммы передавались. Функция *gather* гарантирует детерминированный порядок результатов, несмотря на недетерминированность их получения. На внутреннем уровне *gather* использует для этой цели специальную реализацию *future*. Интересующемуся читателю предлагается заглянуть в исходный код *gather*, этот поучительный опыт поможет осознать, как много API *asyncio* построено с использованием будущих объектов.

В примерах выше предполагается, что все запросы завершаются успешно и не возбуждают исключений. Но что, если в каком-то запросе произойдет ошибка?

4.4.1 *Обработка исключений при использовании gather*

Разумеется, ответ на веб-запрос приходит не всегда, иногда возникает исключение. Поскольку сети ненадежны, возможны различные сценарии отказов. Например, переданный адрес может не существовать или оказаться временно недоступным из-за остановки сайта. Сервер, к которому мы пытаемся подключиться, тоже может быть остановлен или отказать в подключении.

asyncio.gather принимает необязательный параметр, *return_exceptions*, который позволяет указать, как мы хотим обрабатывать исключения от допускающих ожидание объектов. Это булево значение, поэтому возможно два варианта:

- *return_exceptions=False* – это режим по умолчанию. Если хотя бы одна сопрограмма возбуждает исключение, то *gather* возбуждает то же исключение в точке *await*. Но, даже если какая-то сопрограмма откажет, остальные не снимаются и продолжают работать при условии, что мы обработаем исключение и оно не приведет к остановке цикла событий и снятию задач;
- *return_exceptions=True* – в этом случае исключения возвращаются в том же списке, что результаты. Сам по себе вызов *gather* не возбуждает исключений, и мы можем обработать исключения, как нам удобно.

Для иллюстрации изменим список URL-адресов, включив в него недопустимый адрес. Тогда *aiohttp* возбудит исключение при попытке выполнить запрос. Передадим этот список *gather* и посмотрим, как она будет себя вести при разных значениях *return_exceptions*:

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com', 'python://example.com']
        tasks = [fetch_status_code(session, url) for url in urls]
        status_codes = await asyncio.gather(*tasks)
        print(status_codes)
```

Запрос к адресу 'python://example.com' завершается ошибкой, потому что такой URL недопустим. Поэтому наша сопрограмма `fetch_status_code` возбуждает исключение `AssertionError`, так как схему `python://` нельзя преобразовать в номер порта. Это исключение возникает в точке, где мы ждем `gather` с помощью `await`. Выполнив этот код и взглянув на результат, мы увидим, что исключение было возбуждено, но выполнение другого запроса продолжилось (для краткости мы сократили длинную трассу вызовов):

```
выполняется <function main at 0x107f4a4c0> с аргументами () {}
выполняется <function fetch_status_code at 0x107f4a3a0>
выполняется <function fetch_status_code at 0x107f4a3a0>
<function fetch_status_code at 0x107f4a3a0> завершилась за 0.0004 с
<function main at 0x107f4a4c0> завершилась за 0.0203 с
<function fetch_status_code at 0x107f4a3a0> завершилась за 0.0198 с
Traceback (most recent call last):
  File "gather_exception.py", line 22, in <module>
    asyncio.run(main())
AssertionError

Process finished with exit code 1
```

`asyncio.gather` не снимает другие работающие задачи из-за отказа. Во многих случаях это приемлемо, но вообще является одним из недостатков `gather`. Ниже в этой главе мы покажем, как снять конкурентные задачи.

Еще одна потенциальная проблема приведенного выше кода заключается в том, что если произойдет несколько исключений, то в точке `await gather` мы увидим только первое. Это можно исправить, задав параметр `return_exceptions=True`; тогда будут возвращены все исключения, случившиеся во время выполнения сопрограмм. После этого можно будет отделить исключения от результатов и обработать их. Рассмотрим тот же пример с недопустимым URL-адресом:

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com', 'python://example.com']
        tasks = [fetch_status_code(session, url) for url in urls]
        results = await asyncio.gather(*tasks, return_exceptions=True)

    exceptions = [res for res in results if isinstance(res, Exception)]
    successful_results = [res for res in results if not isinstance(res, Exception)]
```

```
print(f'Все результаты: {results}')  
print(f'Завершились успешно: {successful_results}')  
print(f'Завершились с исключением: {exceptions}')
```

При выполнении этой программы исключения не возбуждаются, а возвращаются в одном списке с результатами. Затем мы отфильтровываем все исключения, оставляя только результаты успешно завершившихся запросов. В итоге получаем:

```
Все результаты: [200, AssertionError()]  
Завершились успешно: [200]  
Завершились с исключением: [AssertionError()]
```

Теперь мы видим все исключения, возбужденные сопрограммами, так что эта проблема решена. Хорошо еще и то, что не нужно явно обрабатывать исключения в блоке `try/catch`, потому что в точке `await` исключение не возбуждается. Немного раздражает необходимость отделять зерна от плевел, но и вообще API несовершенен.

Функция `gather` имеет несколько недостатков. Первый мы уже упоминали – не так просто отменить задачи, если одна из них возбудила исключение. Представьте, что мы отправляем запросы одному серверу, и если хотя бы один завершится неудачно, например из-за превышения лимита на частоту запросов, то остальные постигнет та же участь. В таком случае хотелось бы отменить запросы, чтобы освободить ресурсы, но это нелегко, потому что наши сопрограммы обернуты задачами и работают в фоновом режиме.

Второй недостаток – необходимость дожидаться завершения всех сопрограмм, прежде чем можно будет приступить к обработке результатов. Если мы хотим обрабатывать результаты по мере поступления, то возникает проблема. Например, если один запрос выполняется 100 мс, а другой 20 с, то придется ждать, ничего не делая, 20 с, прежде чем мы сможем обработать результаты первого запроса.

`Asyncio` предлагает API, позволяющие решить обе проблемы. Сначала посмотрим, как обрабатывать результаты по мере поступления.

4.5 Обработка результатов по мере поступления

Хотя во многих случаях функция `asyncio.gather` нас устраивает, у нее есть недостаток – необходимость дожидаться завершения всех допускающих ожидания объектов, прежде чем станет возможен доступ к результатам. Это проблема, если требуется обрабатывать результаты в темпе их получения. А также в случае, когда одни объекты завершаются быстро, а другие медленно, потому что `gather` будет ждать завершения всех. В результате приложение может стать неотзывчивым. Представьте, что пользователь отправил 100 запросов, из которых два

медленные, а остальные быстрые. Было бы хорошо, если бы мы могли что-то сообщить пользователю, как только начинают поступать ответы.

Для решения этой проблемы `asyncio` предлагает функцию `as_completed`. Она принимает список допускающих ожидание объектов и возвращает итератор по будущим объектам. Эти объекты можно перебирать, применяя к каждому `await`. Когда выражение `await` вернет управление, мы получим результат первой завершившейся сопрограммы. Это значит, что мы сможем обрабатывать результаты по мере их доступности, но теперь порядок результатов не детерминирован, поскольку неизвестно, какой объект завершится первым.

Для иллюстрации смоделируем ситуацию, когда один запрос завершается быстро, а другой медленно. Добавим в функцию `fetch_status` параметр `delay` и будем вызывать `asyncio.sleep` для имитации медленного запроса:

```
async def fetch_status(session: ClientSession,
                      url: str,
                      delay: int = 0) -> int:
    await asyncio.sleep(delay)
    async with session.get(url) as result:
        return result.status
```

Затем в цикле `for` обойдем итератор, возвращенный функцией `as_completed`.

Листинг 4.8 Использование `as_completed`

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from util import async_timed
from chapter_04 import fetch_status

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = [fetch_status(session, 'https://www.example.com', 1),
                    fetch_status(session, 'https://www.example.com', 1),
                    fetch_status(session, 'https://www.example.com', 10)]
        for finished_task in asyncio.as_completed(fetchers):
            print(await finished_task)

asyncio.run(main())
```

Здесь мы создаем три сопрограммы – две из них завершаются примерно через 1 с, а третья через 10 с. Эти сопрограммы передаются функции `as_completed`. Под капотом каждая сопрограмма оборачивается задачей и начинает выполняться конкурентно. Функция немедленно возвращает итератор, который мы начинаем обходить. Войдя в цикл `for`, мы сразу натываемся на `await finished_task`. Здесь выполнение приостанавливается до момента поступления первого ре-

зультата. В данном случае первый результат поступит через 1 с, и мы напечатаем код состояния. Затем снова дойдем до `await finished_task`, и, так как запросы выполняются конкурентно, второй результат станет доступен почти мгновенно. Наконец, через 10 с завершится третий запрос, а вместе с ним и цикл. Будут напечатаны следующие сообщения:

```
выполняется <function fetch_status at 0x10dbed4c0>
выполняется <function fetch_status at 0x10dbed4c0>
выполняется <function fetch_status at 0x10dbed4c0>
<function fetch_status at 0x10dbed4c0> завершилась за 1.1269 с
200
<function fetch_status at 0x10dbed4c0> завершилась за 1.1294 с
200
<function fetch_status at 0x10dbed4c0> завершилась за 10.0345 с
200
<function main at 0x10dbed5e0> завершилась за 10.0353 с
```

Полное время обхода `result_iterator` по-прежнему составляет 10 с, как было бы при использовании `asyncio.gather`, однако теперь результат первого запроса печатается сразу после получения. Это дает нам дополнительное время для обработки результата первой успешно завершившейся сопрограммы, пока остальные еще выполняются, поэтому приложение оказывается более отзывчивым.

Эта функция также дает больше контроля над обработкой исключений. Если задача возбудит исключение, то мы сможем обработать ее сразу же, поскольку оно возникает в точке, где мы ожидаем будущего объекта с помощью `await`.

4.5.1 Тайм-ауты в сочетании с `as_completed`

Любой веб-запрос может занять много времени. Возможно, сервер испытывает высокую нагрузку, а возможно, сеть медленная. Выше мы видели, как задать тайм-аут для отдельного запроса, но что, если это нужно сделать для группы запросов? Функция `as_completed` предоставляет такую возможность с помощью необязательного параметра `timeout`, равного величине тайм-аута в секундах. Он управляет временем работы `as_completed`; если потребуется больше, то каждый допускающий ожидание объект в итераторе возбудит исключение `TimeoutException` в точке ожидания с помощью `await`.

Для иллюстрации возьмем предыдущий пример и создадим два запроса по 10 с и один запрос, занимающий 1 с. И зададим тайм-аут 2 с при вызове `as_completed`. По завершении цикла напечатаем все выполняемые в данный момент задачи.

Листинг 4.9 Задание тайм-аута для `as_completed`

```
import asyncio
import aiohttp
```

```

from aiohttp import ClientSession
from util import async_timed
from chapter_04 import fetch_status

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = [fetch_status(session, 'https://example.com', 1),
                    fetch_status(session, 'https://example.com', 10),
                    fetch_status(session, 'https://example.com', 10)]

        for done_task in asyncio.as_completed(fetchers, timeout=2):
            try:
                result = await done_task
                print(result)
            except asyncio.TimeoutError:
                print('Произошел тайм-аут!')

        for task in asyncio.tasks.all_tasks():
            print(task)

asyncio.run(main())

```

При выполнении этой программы мы увидим результат первого вызова `fetch_status`, а спустя 2 с два тайм-аута. Также мы увидим, что два вызова `fetch_status` еще работают:

```

выполняется <function main at 0x109c7c430> с аргументами () {}
200
Произошел тайм-аут!
Произошел тайм-аут!
<function main at 0x109c7c430> завершилась за 2.0055 с
<Task pending name='Task-2' coro=<fetch_status_code()>>
<Task pending name='Task-1' coro=<main>>
<Task pending name='Task-4' coro=<fetch_status_code()>>

```

`as_completed` справляется со своей задачей – возвращать результат по мере поступления, но она не лишена недостатков. Первый заключается в том, что хотя мы и получаем результаты в темпе их поступления, но невозможно сказать, какую сопрограмму или задачу мы ждем, поскольку порядок абсолютно не детерминирован. Если порядок нас не волнует, то и ладно, но если требуется ассоциировать результаты с запросами, то возникает проблема.

Второй недостаток в том, что, хотя исключения по истечении тайм-аута возбуждаются как положено, все созданные задачи продолжают работать в фоновом режиме. А если мы захотим их снять, то будет трудно понять, какие задачи еще работают. Вот вам и еще одна проблема! Если эти проблемы требуется решить, то нужно точно знать, какие допускающие ожидание объекты уже завершились, а какие еще нет. Поэтому `asyncio` предоставляет функцию `wait`.

4.6 Точный контроль с помощью `wait`

Один из недостатков обеих функций `gather` и `as_completed` – сложности со снятием задач, работавших в момент исключения. Во многих случаях в этом нет ничего страшного, но представьте ситуацию, когда мы делаем несколько вызовов сопрограммы, и если один из них завершается неудачно, то так же завершатся и все остальные. Примером может служить задание недопустимого параметра веб-запроса или достижение лимита на частоту запросов. Это может привести к падению производительности, потому что мы потребляем больше ресурсов, так как запустили больше задач, чем необходимо. Другой недостаток, свойственный `as_completed`, – недетерминированный порядок получения результатов, из-за чего трудно понять, какая именно задача завершилась.

Функция `wait` в `asyncio` похожа на `gather`, но дает более точный контроль над ситуацией. У нее есть несколько параметров, позволяющих решить, когда мы хотим получить результаты. Кроме того, она возвращает два множества: задачи, завершившиеся успешно или в результате исключения, а также задачи, которые продолжают выполняться. Еще эта функция позволяет задать тайм-аут, который, однако, ведет себя не так, как в других функциях API: он не возбуждает исключений. В тех случаях, когда необходимо, эта функция позволяет решить некоторые отмеченные выше проблемы, присущие другим функциям `asyncio`.

Базовая сигнатура `wait` – список допускающих ожидание объектов, за которым следует факультативный тайм-аут и факультативный параметр `return_when`, который может принимать значения `ALL_COMPLETED`, `FIRST_EXCEPTION` и `FIRST_COMPLETED`, а по умолчанию равен `ALL_COMPLETED`. Хотя на момент написания книги `wait` принимает список допускающих ожидание объектов, в будущих версиях Python она будет принимать только объекты `task`. Почему так, мы объясним в конце раздела, а в примерах кода будем придерживаться рекомендованной практики и обертывать все сопрограммы задачами.

4.6.1 Ожидание завершения всех задач

Этот режим подразумевается по умолчанию, если параметр `return_when` не задан явно. Он ближе всего к функции `asyncio.gather`, хотя несколько отличий все же есть. Как следует из названия, в этом режиме функция ждет завершения всех задач и только потом возвращает управление. Применим ее в нашем примере конкурентного выполнения нескольких веб-запросов.

Листинг 4.10 Изучение поведения `wait` по умолчанию

```
import asyncio
import aiohttp
```

```

from aiohttp import ClientSession
from util import async_timed
from chapter_04 import fetch_status

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = \
            [asyncio.create_task(fetch_status(session, 'https://example.com')),
             asyncio.create_task(fetch_status(session, 'https://example.com'))]
        done, pending = await asyncio.wait(fetchers)

        print(f'Число завершившихся задач: {len(done)}')
        print(f'Число ожидающих задач: {len(pending)}')

        for done_task in done:
            result = await done_task
            print(result)

asyncio.run(main())

```

Здесь мы конкурентно выполняем два веб-запроса, передавая `wait` список задач. Предложение `await wait` вернет управление, когда все запросы завершатся, и мы получим два множества: завершившиеся задачи и еще работающие задачи. Множество `done` содержит все задачи, которые завершились успешно или в результате исключения, а множество `pending` – еще не завершившиеся задачи. В данном случае мы задали режим `ALL_COMPLETED`, поэтому множество `pending` будет пустым, так как `asyncio.wait` не вернется, пока все не завершится. В итоге мы увидим такие сообщения:

```

выполняется <function main at 0x10124b160> с аргументами () {}
Число завершившихся задач: 2
Число ожидающих задач: 0
200
200
<function main at 0x10124b160> завершилась за 0.4642 с

```

Если в одном из запросов возникнет исключение, то `asyncio.wait` не возбудит его, как `asyncio.gather`. В этом случае мы, как и раньше, получим оба множества `done` и `pending`, но не увидим исключения, пока не применим `await` к той задаче из `done`, где имела место ошибка.

Таким образом, у нас на выбор есть несколько способов обработать исключения. Можно выполнить `await` и дать возможность исключению распространиться выше, можно выполнить `await`, обернуть его в блок `try/except`, чтобы обработать исключение, или воспользоваться методами `task.result()` и `task.exception()`. Вызывать эти методы безопасно, поскольку в множестве `done` гарантированно находятся завершенные задачи; иначе их вызов привел бы к исключению.

Предположим, что мы не хотим возбуждать исключение и обрушивать свое приложение. Вместо этого мы хотим напечатать результат

задачи, если он получен, или запротоколировать ошибку в случае исключения. В таком случае вполне подойдет использование методов объекта Task. Посмотрим, как их использовать для обработки исключений.

Листинг 4.11 Обработка исключений при использовании wait

```
import asyncio
import logging

@async_typed()
async def main():
    async with aiohttp.ClientSession() as session:
        good_request = fetch_status(session, 'https://www.example.com')
        bad_request = fetch_status(session, 'python://bad')

        fetchers = [asyncio.create_task(good_request),
                    asyncio.create_task(bad_request)]

        done, pending = await asyncio.wait(fetchers)

        print(f'Число завершившихся задач: {len(done)}')
        print(f'Число ожидающих задач: {len(pending)}')

        for done_task in done:
            # result = await done_task вызовет исключение
            if done_task.exception() is None:
                print(done_task.result())
            else:
                logging.error("При выполнении запроса возникло исключение",
                              exc_info=done_task.exception())

asyncio.run(main())
```

Функция `done_task.exception()` проверяет, имело ли место исключение. Если нет, то можно получить результат из `done_task` методом `result`. Здесь также было бы безопасно написать `result = await done_task`, хотя при этом может возникнуть исключение, чего мы, возможно, не желаем. Если результат `exception()` не равен `None`, то в допускающем ожидание объекте возникло исключение и его можно обработать, как нам угодно. В данном случае просто печатаем трассу стека в момент исключения. При выполнении этой программы мы увидим такую картину (для краткости трасса вызовов сокращена):

```
выполняется <function main at 0x10401f1f0> с аргументами () {}
Число завершившихся задач: 2
Число ожидающих задач: 0
200
<function main at 0x10401f1f0> завершилась за 0.12386679649353027 с
ERROR:root:Request got an exception
Traceback (most recent call last):
AssertionError
```

4.6.2 Наблюдение за исключениями

Режим `ALL_COMPLETED` страдает всеми теми же недостатками, что и `gather`. Пока мы ждем завершения сопрограмм, может возникнуть сколько угодно исключений, но мы их не увидим, пока все задачи не завершатся. Это может стать проблемой, если после первого же исключения следует снять все остальные выполняющиеся запросы. Кроме того, немедленная обработка ошибок желательна для повышения отзывчивости приложения.

Чтобы поддержать эти сценарии, `wait` имеет режим `FIRST_EXCEPTION`. В этом случае мы получаем два разных поведения в зависимости от того, возникает в какой-то задаче исключение или нет.

Ни один допускающий ожидание объект не возбудил исключения

Если ни в одной задаче не было исключений, то этот режим эквивалентен `ALL_COMPLETED`. Мы дождемся завершения всех задач, после чего множество `done` будет содержать все задачи, а множество `pending` останется пустым.

В одной или нескольких задачах возникло исключение

Если хотя бы в одной задаче возникло исключение, то `wait` немедленно возвращается. Множество `done` будет содержать как задачи, завершившиеся успешно, так и те, в которых имело место исключение. Гарантируется, что `done` будет содержать как минимум одну задачу – завершившуюся ошибкой, но может содержать и какие-то успешно завершившиеся задачи. Множество `pending` может быть пустым, а может содержать задачи, которые продолжают выполняться. Мы можем использовать его для управления выполняемыми задачами по своему усмотрению.

Для иллюстрации поведения `wait` в этих случаях посмотрим, что происходит, когда есть два долгих веб-запроса и мы хотели бы сразу же снять другой, если при выполнении одного возникло исключение.

Листинг 4.12 Отмена работающих запросов при возникновении исключения

```
import aiohttp
import asyncio
import logging
from chapter_04 import fetch_status
from util import async_timed

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
```

```

fetchers = \
    [asyncio.create_task(fetch_status(session, 'python://bad.com')),
      asyncio.create_task(fetch_status(session, 'https://www.example.com', delay=3)),
      asyncio.create_task(fetch_status(session, 'https://www.example.com', delay=3))]

done, pending = await asyncio.wait(fetchers,
                                    return_when=asyncio.FIRST_EXCEPTION)

print(f'Число завершившихся задач: {len(done)}')
print(f'Число ожидающих задач: {len(pending)}')

for done_task in done:
    if done_task.exception() is None:
        print(done_task.result())
    else:
        logging.error("При выполнении запроса возникло исключение",
                      exc_info=done_task.exception())

for pending_task in pending:
    pending_task.cancel()

asyncio.run(main())

```

Здесь мы отправляем один плохой запрос и два хороших, по 3 с каждый. Ожидание завершения wait прекращается почти сразу, потому что плохой запрос тут же завершается с ошибкой. Затем мы в цикле обходим множество done. В данном случае оно будет содержать всего одну задачу, потому что первый запрос немедленно завершился в результате исключения. Для нее мы идем по ветви, печатающей сведения об исключении.

Множество pending содержит два элемента, поскольку у нас есть два запроса, исполняющихся примерно по 3 с, а первый запрос закончился почти сразу. Мы не хотим, чтобы оставшиеся запросы продолжали выполняться, поэтому прерываем их методом cancel. В итоге получается такая картина:

```

выполняется <function main at 0x105cfd280> с аргументами () {}
Число завершившихся задач: 1
Число ожидающих задач: 2
<function main at 0x105cfd280> завершилась за 0.0044 с
ERROR:root:Request got an exception

```

ПРИМЕЧАНИЕ Приложение не заняло почти никакого времени, потому что мы быстро отреагировали на то, что один из запросов возбудил исключение; прелесть этого режима в том, что реализуется тактика быстрого отказа, т. е. быстрой реакции на возникающие проблемы.

4.6.3 Обработка результатов по мере завершения

У режимов ALL_COMPLETED и FIRST_EXCEPTION есть недостаток: если за-

дачи не возбуждают исключений, то мы должны ждать, пока все они завершатся. Иногда это приемлемо, но если мы должны отреагировать на успешное завершение задачи немедленно, то ничего не получится.

В случае, когда требуется немедленная реакция, можно было бы воспользоваться функцией `as_completed`, однако у нее есть проблема: непонятно, какие задачи еще работают, а какие уж завершились. Получать задачи можно только по одной, от итератора.

Но есть и хорошая новость – параметр `return_when` может принимать значение `FIRST_COMPLETED`. В этом режиме `wait` возвращает управление, как только получен хотя бы один результат. Это может быть как успешно завершившаяся задача, так и задача, в которой возникло исключение. Остальные задачи можно либо снять, либо дать им возможность продолжать работу. Продемонстрируем этот режим, для чего отправим несколько веб-запросов и обработаем тот, что закончится первым.

Листинг 4.13 Обработка запросов по мере завершения

```
import asyncio
import aiohttp
from util import async_timed
from chapter_04 import fetch_status

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https://www.example.com'
        fetchers = [asyncio.create_task(fetch_status(session, url)),
                     asyncio.create_task(fetch_status(session, url)),
                     asyncio.create_task(fetch_status(session, url))]
        done, pending = await asyncio.wait(fetchers,
                                           return_when=asyncio.FIRST_COMPLETED)

        print(f'Число завершившихся задач: {len(done)}')
        print(f'Число ожидающих задач: {len(pending)}')

        for done_task in done:
            print(await done_task)

asyncio.run(main())
```

Здесь мы конкурентно отправляем три запроса. Сопрограмма `wait` вернет управление, как только завершится любой из них. Следовательно, в `done` будет один завершившийся запрос, а в `pending` – еще работающие, и мы увидим следующую картину:

```
выполняется <function main at 0x10222f1f0> с аргументами () {}
Число завершившихся задач: 1
Число ожидающих задач: 2
200
<function main at 0x10222f1f0> завершилась за 0.1138 с
```

Эти запросы могут завершиться примерно в одно время, поэтому не исключено, что в `done` окажется две или три задачи. Попробуйте выполнить эту программу несколько раз и посмотрите, как будут меняться результаты.

Описанный подход позволяет реагировать, как только завершится первая задача. Но что, если мы хотим обработать и остальные результаты по мере поступления, как при использовании `as_completed`? Предыдущий пример легко можно модифицировать, так чтобы задачи из множества `pending` обрабатывались в цикле, пока там ничего не останется. Тогда мы получим поведение, аналогичное `as_completed`, с тем дополнительным преимуществом, что на каждом шаге точно знаем, какие задачи завершились, а какие еще работают.

Листинг 4.14 Обработка всех результатов по мере поступления

```
import aiohttp
from chapter_04 import fetch_status
from util import async_timed

@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https://www.example.com'
        pending = [asyncio.create_task(fetch_status(session, url)),
                   asyncio.create_task(fetch_status(session, url)),
                   asyncio.create_task(fetch_status(session, url))]

        while pending:
            done, pending = await asyncio.wait(pending,
                                              return_when=asyncio.FIRST_COMPLETED)

            print(f'Число завершившихся задач: {len(done)}')
            print(f'Число ожидающих задач: {len(pending)}')

            for done_task in done:
                print(await done_task)

asyncio.run(main())
```

Здесь мы создаем множество `pending` и инициализируем его задачами, которые хотим выполнить. Цикл `while` выполняется, пока в `pending` остаются элементы, и на каждой итерации мы вызываем `wait` для этого множества. Получив результат от `wait`, мы обновляем множества `done` и `pending`, а затем печатаем завершившиеся задачи. Получается поведение, похожее на `as_completed`, с тем отличием, что теперь мы лучше знаем, какие задачи завершились, а какие продолжают работать. При выполнении этой программы мы увидим такую картину:

```
выполняется <function main at 0x10d1671f0> с аргументами () {}
Число завершившихся задач: 1
```

```

Число ожидающих задач: 2
200
Число завершившихся задач: 1
Число ожидающих задач: 1
200
Число завершившихся задач: 1
Число ожидающих задач: 0
200
<function main at 0x10d1671f0> завершилась за 0.1153 с

```

Поскольку запрос выполняется быстро, все запросы могли бы завершиться одновременно, так что вполне возможна и такая картина:

```

выполняется <function main at 0x1100f11f0> с аргументами () {}
Число завершившихся задач: 3
Число ожидающих задач: 0
200
<function main at 0x1100f11f0> завершилась за 0.1304 с

```

4.6.4 Обработка тайм-аутов

Функция `wait` позволяет не только точнее контролировать порядок ожидания задач, но и задавать время, в течение которого все допускающие ожидание объекты должны завершиться. Для этого нужно задать параметр `timeout`, указав в нем максимальное время работы в секундах. Между обработкой тайм-аутов `wait` и ранее рассмотренными `wait_for` и `as_completed` есть два отличия.

Сопрограммы не снимаются

Если при выполнении сопрограммы, запущенной с помощью `wait_for`, случался тайм-аут, то она автоматически снималась. В случае `wait` это не так: поведение ближе к `gather` и `as_completed`. Если мы хотим снять сопрограммы из-за тайм-аута, то должны явно обойти их и снять каждую.

Исключения не возбуждаются

`wait` не возбуждает исключения в случае тайм-аута, в отличие от `wait_for` и `as_completed`. Когда случается тайм-аут, `wait` возвращает все завершившиеся задачи, а также те, что еще не завершились в момент тайм-аута.

Например, рассмотрим случай, когда два запроса завершаются быстро, а третий занимает несколько секунд. Мы зададим в `wait` тайм-аут 1 с, чтобы понять, что происходит, когда задачи не успевают завершиться. Параметр `return_when` пусть принимает значение по умолчанию `ALL_COMPLETED`.

Листинг 4.15 Использование тайм-аутов в wait

```
@async_timed()
async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https://example.com'
        fetchers = [asyncio.create_task(fetch_status(session, url),
                                         asyncio.create_task(fetch_status(session, url),
                                         asyncio.create_task(fetch_status(session, url, delay=3)))]

        done, pending = await asyncio.wait(fetchers, timeout=1)

        print(f'Число завершившихся задач: {len(done)}')
        print(f'Число ожидающих задач: {len(pending)}')

        for done_task in done:
            result = await done_task
            print(result)

asyncio.run(main())
```

Здесь wait вернет множества done и pending через 1 с. В множестве done будет два быстрых запроса, поскольку они успевают завершиться за это время. А медленный запрос еще работает, поэтому окажется в множестве pending. Затем мы ждем задачи из done с помощью await и получаем возвращенные ими значения. При желании можно было бы снять задачу, находящуюся в pending. При выполнении этого кода мы увидим следующую картину:

```
выполняется <function main at 0x11c68dd30> с аргументами () {}
Число завершившихся задач: 2
Число ожидающих задач: 1
200
200
<function main at 0x11c68dd30> завершилась за 1.0022 с
```

Заметим, что, как и раньше, задачи в множестве pending не сняты и продолжают работать, несмотря на тайм-аут. Если в конкретной ситуации требуется завершить еще выполняющиеся задачи, то следовало бы явно обойти множество pending и вызвать для каждой задачи cancel.

4.6.5 Зачем оборачивать сопрограммы задачами?

В начале этой главы мы упомянули, что сопрограммы, передаваемые wait, рекомендуется оборачивать задачами. Почему? Вернемся к предыдущему примеру, иллюстрирующему тайм-аут, и немного изменим его. Пусть имеется два запроса к разным API, которые назовем API A и API B. Оба могут тормозить, но наше приложение может продолжить работу, не получив результата от API B, просто было бы хорошо его иметь. Мы хотим, чтобы приложение было отзывчивым,

поэтому задаем для запросов тайм-аут 1 с. Если по истечении тайм-аута запрос к API В еще работает, то мы отменяем его. Посмотрим, что произойдет, если реализовать эту идею, не обертывая сопрограммы задачами.

Листинг 4.16 Отмена медленного запроса

```
import asyncio
import aiohttp
from chapter_04 import fetch_status

async def main():
    async with aiohttp.ClientSession() as session:
        api_a = fetch_status(session, 'https://www.example.com')
        api_b = fetch_status(session, 'https://www.example.com', delay=2)

        done, pending = await asyncio.wait([api_a, api_b], timeout=1)

        for task in pending:
            if task is api_b:
                print('API В слишком медленный, отмена')
                task.cancel()

asyncio.run(main())
```

Мы ожидаем, что этот код напечатает «API В слишком медленный, отмена», но на самом деле мы его вообще не увидим. Такое может случиться, потому что, когда `wait` передаются просто сопрограммы, они автоматически обертываются задачами, а возвращенные множества `done` и `pending` будут содержать эти задачи. Это значит, что сравнения на предмет присутствия задачи в множестве `pending`, как в предложении `if task is api_b`, неправомерны, потому что мы сравниваем разные объекты: сопрограмму и задачу. Но если обернуть `fetch_status` задачей, то новые объекты не создаются и сравнение `if task is api_b` будет работать, как мы и ожидаем.

Резюме

- Мы научились использовать и создавать асинхронные контекстные менеджеры. Это специальные классы, которые позволяют асинхронно захватывать ресурсы, а затем освобождать их даже при наличии исключения. Их задача – очистить захваченные ресурсы без лишнего кода, особенно они полезны при работе с HTTP-сеансами и подключениями к базе данных. Для использования асинхронных контекстных менеджеров предназначена синтаксическая конструкция `async with`.
- Мы можем использовать библиотеку `aiohttp` для отправки асинхронных веб-запросов. Эта библиотека позволяет создавать клиен-

ты и серверы с неблокирующими сокетами. В случае веб-клиента мы можем конкурентно выполнять несколько запросов, не блокируя цикл событий.

- Функция `asyncio.gather` позволяет конкурентно запустить несколько сопрограмм и ждать их завершения. Она возвращает управление, когда завершатся все переданные ей объекты, допускающие ожидание. Если нужно отслеживать возникающие ошибки, то можно задать параметр `return_exceptions` равным `True`. Тогда будут возвращаться как результаты успешно завершившихся объектов, так и возникшие исключения.
- Функция `as_completed` позволяет обрабатывать результаты списка допускающих ожидание объектов по мере их завершения. Она возвращает итератор по будущим объектам, который можно обойти. Как только сопрограмма или задача завершается, итератор отдает ее результат.
- Если мы хотим выполнить несколько задач конкурентно, но при этом понимать, какие задачи уже завершились, а какие еще работают, то можем использовать функцию `wait`. Она также дает более точный контроль над моментом возврата результатов. После возврата мы получаем множество завершившихся задач и множество еще работающих. Затем можем отменить какие-то задачи или снова ждать их завершения.

Неблокирующие драйверы баз данных

Краткое содержание главы

- Выполнение совместимых с `asyncio` запросов к базе данных с помощью `asynpg`.
- Создание пула подключений к базе данных для конкурентного выполнения нескольких SQL-запросов.
- Управление асинхронными транзакциями базы данных.
- Использование асинхронных генераторов для потоковой обработки результатов запроса.

В главе 4 мы изучали отправку неблокирующих веб-запросов с помощью библиотеки `aiohttp`, а заодно рассмотрели несколько функций `asyncio` API для конкурентного выполнения этих запросов. Сочетание `asyncio` API с библиотекой `aiohttp` позволяет конкурентно выполнять долгие веб-запросы и тем самым повысить производительность приложения. Рассмотренные в главе 4 идеи применимы не только к веб-запросам, а также к SQL-запросам, что позволяет писать более быстрые приложения для работы с базой данных.

Как и в случае веб-запросов, нам понадобится совместимая с `asyncio` библиотека, потому что большинство стандартных библиотек для работы с SQL блокируют главный поток, а значит, и цикл событий до получения результата. В этой главе мы подробнее поговорим об асинхронном доступе к базам данных на примере библиотеки `asynpg`.

Сначала создадим простую схему, в которой будем хранить товары, продаваемые через интернет-магазин, а затем покажем, как предъявлять к ней запросы асинхронно. После этого мы рассмотрим управление транзакциями в нашей базе данных, а также вопрос о пуле подключений.

5.1 Введение в *asynpcpg*

Выше мы уже говорили, что существующие блокирующие библиотеки плохо сочетаются с сопрограммами. Чтобы конкурентно выполнять запросы к базе данных, нам необходима совместимая с *asyncio* библиотека, в которой используются неблокирующие сокеты. Примером такой библиотеки является *asynpcpg*, позволяющая асинхронно подключаться к базе данных Postgres и предъявлять к ней запросы.

В этой главе мы будем говорить только о СУБД Postgres, но все изложенное применимо также к MySQL и другим базам данных. Авторы *aiohttp* создали также библиотеку *aiomysql*, которая умеет подключаться к СУБД MySQL и выполнять в ней запросы.

Несмотря на некоторые различия, API похожи и знания, полученные для одной, переносятся на другую. Отметим, что библиотека *asynpcpg* не реализует спецификации API работы с базами данных для Python, описанные в документе PEP-249 (<https://www.python.org/dev/peps/pep-0249>). Это осознанный выбор со стороны авторов библиотеки, поскольку асинхронная реализация принципиально отлична от синхронной. Однако создатели *aiomysql* пошли по другому пути и все-таки реализовали PEP-249, так что эта библиотека покажется более привычной тем, кто работал с синхронными драйверами баз данных в Python.

Текущая документация по *asynpcpg* доступна по адресу <https://magic-stack.github.io/asynpcpg/current/>. Определившись с драйвером, попробуем подключиться к базе данных.

5.2 Подключение к базе данных Postgres

Для работы с *asynpcpg* мы возьмем реалистичный пример базы данных с информацией о товарах для интернет-магазина. На этом сквозном примере мы продемонстрируем проблемы, которые необходимо решить.

Прежде чем создавать базу данных и выполнять запросы, необходимо подключиться к базе данных. В этой главе мы будем предполагать, что на локальной машине установлена СУБД Postgres, которая прослушивает порт по умолчанию 5432, и что пользователь по умолчанию *postgres* имеет пароль *'password'*.

ПРЕДУПРЕЖДЕНИЕ Мы зашиваем пароль в код примеров только для простоты демонстрации. В реальном коде паролей не должно быть *никогда*, потому что это нарушает базовые принципы безопасности. Всегда храните пароли в переменных окружения или пользуйтесь каким-то другим механизмом конфигурирования.

Скачать и установить Postgres можно со страницы <https://www.postgresql.org/download/>; выберите только подходящую операционную систему. Можно также остановиться на Docker-образе Postgres; дополнительные сведения имеются по адресу https://hub.docker.com/_/postgres/.

После базы данных нужно установить библиотеку `asyncpg`. Для этого мы воспользуемся программой `pip3` и установим последнюю на момент написания книги версию 0.0.23:

```
pip3 install -Iv asyncpg==0.0.23
```

Затем библиотеку нужно импортировать и подключиться к базе данных. Для этой цели `asyncpg` предлагает функцию `asyncpg.connect`. Воспользуемся ей, чтобы подключиться к базе, и напечатаем номер версии:

Листинг 5.1 Подключение к базе данных Postgres от имени пользователя по умолчанию

```
import asyncpg
import asyncio

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='postgres',
                                       password='password')

    version = connection.get_server_version()
    print(f'Подключено! Версия Postgres равна {version}')
    await connection.close()

asyncio.run(main())
```

Здесь мы создаем подключение к базе данных `postgres` от имени пользователя по умолчанию `postgres`. В предположении, что экземпляр `Postgres` работает, мы должны увидеть на консоли сообщение вида «Подключено! Версия `Postgres` равна `ServerVersion(major=12, minor=0, micro=3, releaselevel='final' serial=0)`», означающее, что мы успешно подключились к базе данных. И в конце закрываем подключение в предложении `await connection.close()`.

Мы подключились, но пока в нашей базе ничего нет. Следующий шаг – создать схему, с которой можно будет работать. Попутно мы узнаем, как выполнять простые запросы с помощью `asyncpg`.

5.3 Определение схемы базы данных

Чтобы начать выполнять запросы, нужна схема базы данных. Мы создадим простую схему `products`, моделирующую товары на складе интернет-магазина. Определим несколько сущностей, которые затем преобразуем в таблицы базы данных.

Марка

Под *маркой* (brand) понимается производитель многих различных товаров. Например, Ford – марка, под которой производятся различные модели автомобилей (Ford F150, Ford Fiesta и т. д.).

Товар

Товар (product) ассоциирован с одной маркой, существует связь один-ко-многим между марками и товарами. Для простоты в нашей базе данных у товара будет только название. В примере с Ford товаром будет компактный автомобиль *Fiesta*; его марка *Ford*. Кроме того, с каждым товаром может быть связано несколько размеров и цветов. Совокупность размера и цвета мы определим как SKU.

SKU

SKU расшифровывается как *stock keeping unit* (складская единица хранения). SKU представляет конкретный предмет, выставленный на продажу. Например, для товара *джинсы* SKU может иметь вид: *джинсы, размер: M, цвет: синий* или *джинсы, размер: S, цвет: черный*. Существует связь один-ко-многим между товаром и SKU.

Размер товара

Товар может иметь несколько размеров (product size). В этом примере будем предполагать, что всего есть три размера: малый (S), средний (M) и большой (L). С каждым SKU ассоциирован один размер, поэтому существует связь один-ко-многим между размером товара и SKU.

Цвет товара

Товар может иметь несколько цветов (product color). В этом примере будем предполагать, что цветов всего два: черный и синий. Существует связь один-ко-многим между цветом товара и SKU.

В итоге получается модель базы данных, изображенная на рис. 5.1.

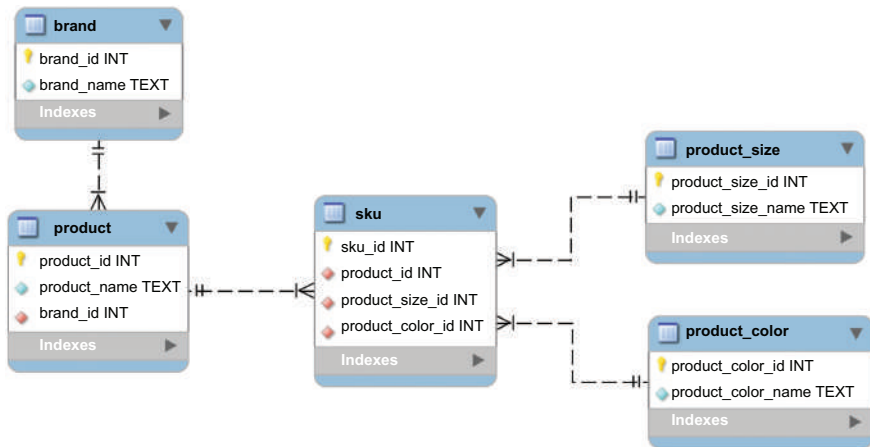


Рис. 5.1 Диаграмма сущность–связь для базы данных о товарах

Теперь определим переменные, нужные для создания этой схемы. С помощью `asynsrg` мы выполним соответствующие команды для создания базы данных о товарах. Поскольку размеры и цвета известны заранее, вставим несколько записей в таблицы `product_size` и `product_color`. Мы будем ссылаться на эти переменные в последующих листингах, чтобы не повторять длинные команды SQL.

Листинг 5.2 Команды создания таблиц в схеме базы данных о товарах

```

CREATE_BRAND_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS brand(
        brand_id SERIAL PRIMARY KEY,
        brand_name TEXT NOT NULL
    );"""

CREATE_PRODUCT_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS product(
        product_id SERIAL PRIMARY KEY,
        product_name TEXT NOT NULL,
        brand_id INT NOT NULL,
        FOREIGN KEY (brand_id) REFERENCES brand(brand_id)
    );"""

CREATE_PRODUCT_COLOR_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS product_color(
        product_color_id SERIAL PRIMARY KEY,

```



```

        product_color_name TEXT NOT NULL
    );"""

CREATE_PRODUCT_SIZE_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS product_size(
        product_size_id SERIAL PRIMARY KEY,
        product_size_name TEXT NOT NULL
    );"""

CREATE_SKU_TABLE = \
    """
    CREATE TABLE IF NOT EXISTS sku(
        sku_id SERIAL PRIMARY KEY,
        product_id INT NOT NULL,
        product_size_id INT NOT NULL,
        product_color_id INT NOT NULL,
        FOREIGN KEY (product_id)
        REFERENCES product(product_id),
        FOREIGN KEY (product_size_id)
        REFERENCES product_size(product_size_id),
        FOREIGN KEY (product_color_id)
        REFERENCES product_color(product_color_id)
    );"""

COLOR_INSERT = \
    """
    INSERT INTO product_color VALUES(1, 'Blue');
    INSERT INTO product_color VALUES(2, 'Black');
    """

SIZE_INSERT = \
    """
    INSERT INTO product_size VALUES(1, 'Small');
    INSERT INTO product_size VALUES(2, 'Medium');
    INSERT INTO product_size VALUES(3, 'Large');
    """

```

Итак, команды создания таблиц и вставки записей о размерах и цветах у нас есть, теперь нужно как-то выполнить соответствующие запросы.

5.4 *Выполнение запросов с помощью asynpcpg*

Чтобы выполнить запрос к базе данных, нужно сначала подключиться к экземпляру Postgres и создать базу данных вне Python. Для создания базы достаточно одной SQL-команды (после подключения от имени пользователя по умолчанию postgres):

```
CREATE DATABASE products;
```

Для этого выполните в командной строке команду `sudo -u postgres psql -c "CREATE TABLE products;"`. В следующих примерах предполагается, что она уже выполнена и мы можем подключаться к базе данных `products` непосредственно.

Создав базу данных, подключимся к ней и выполним команды `create`. В классе `connection` имеется сопрограмма `execute`, которая позволяет выполнить команды создания одну за другой. Эта сопрограмма возвращает полученную от `Postgres` строку, представляющую состояние запроса. Давайте выполним показанные в предыдущем разделе команды.

Листинг 5.3 Использование сопрограммы `execute` для выполнения команд `create`

```
import asyncio

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    statements = [CREATE_BRAND_TABLE,
                  CREATE_PRODUCT_TABLE,
                  CREATE_PRODUCT_COLOR_TABLE,
                  CREATE_PRODUCT_SIZE_TABLE,
                  CREATE_SKU_TABLE,
                  SIZE_INSERT,
                  COLOR_INSERT]

    print('Создается база данных product...')
    for statement in statements:
        status = await connection.execute(statement)
        print(status)
    print('База данных product создана!')
    await connection.close()

asyncio.run(main())
```

Сначала создается подключение к базе данных `products` так же, как в нашем первом примере, с той разницей, что теперь мы подключаемся к другой базе. Затем выполняем команды `CREATE TABLE` по одной с помощью функции `connection.execute()`. Отметим, что `execute()` – сопрограмма, поэтому завершения SQL-команд следует ожидать с помощью `await`. Если все работает правильно, то состоянием каждой команды `create` должна быть строка `CREATE TABLE`, а каждой команды `insert` – строка `INSERT 0 1`. В конце мы закрываем подключение к базе данных. В этом примере мы ожидаем завершения каждой SQL-команды с помощью `await` в цикле `for`, поэтому команды `INSERT` будут выполнены синхронно. Поскольку одни таблицы зависят от других, мы не можем выполнять эти команды конкурентно.

С этими командами не связаны результаты, поэтому давайте вставим несколько записей и выполним простые запросы. Сначала вставим несколько марок, а затем выполним запрос и убедимся, что вставка прошла нормально. Для вставки данных можно использовать все ту же сопрограмму `execute`, а для выборки – сопрограмму `fetch`.

Листинг 5.4 Вставка и выборка марок

```
import asyncpg
import asyncio
from asyncpg import Record
from typing import List

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')
    await connection.execute("INSERT INTO brand VALUES(DEFAULT, 'Levis')")
    await connection.execute("INSERT INTO brand VALUES(DEFAULT, 'Seven')")

    brand_query = 'SELECT brand_id, brand_name FROM brand'
    results: List[Record] = await connection.fetch(brand_query)

    for brand in results:
        print(f'id: {brand["brand_id"]}, name: {brand["brand_name"]}')

    await connection.close()

asyncio.run(main())
```

Сначала вставим в таблицу `brand` две марки. Затем с помощью вызова `connection.fetch` выберем все марки из таблицы `brand`. По завершении запроса результаты будут находиться в памяти, в переменной `results`. Каждый результат представлен объектом `asyncpg.Record`. Эти объекты похожи на словари: они позволяют обращаться к данным, передавая имя столбца в качестве индекса. При выполнении показанной выше программы мы увидим такие строки:

```
id: 1, name: Levis
id: 2, name: Seven
```

В этом примере все возвращенные запросом данные помещены в список. Если бы мы хотели выбрать одну запись, то вызвали бы функцию `connection.fetchrow()`. По умолчанию все результаты запроса загружаются в память, поэтому пока нет разницы в производительности между `fetchrow` и `fetch`. Ниже в этой главе мы узнаем, как обрабатывать результирующие наборы потоком с помощью курсоров. Тогда в память загружается только небольшое подмножество результатов; это полезно, если запрос может возвращать очень много данных.

В этом примере запросы выполняются один за другим, поэтому синхронный драйвер базы данных показал бы такую же производительность. Но поскольку теперь мы возвращаем сопрограммы, ничто не мешает использовать изученные в главе 4 методы `asyncio API` для конкурентного выполнения запросов.

5.5 Конкурентное выполнение запросов с помощью пулов подключений

По-настоящему преимущества `asyncio` для операций ввода-вывода проявляются, когда несколько задач выполняется конкурентно. Не зависящие друг от друга запросы, которые повторяются многократно, – хорошие примеры применения конкурентности для повышения производительности приложения. Для демонстрации предположим, что мы управляем успешным интернет-магазином. Наша компания продает 100 000 SKU от 1000 различных марок.

Предположим также, что мы продаем товары через партнеров, которые запрашивают сразу тысячи товаров, пользуясь организованным нами пакетным процессом. Запускать все запросы последовательно было бы медленно, поэтому хотелось бы создать приложение, которое будет выполнять их конкурентно, повысив тем самым скорость. Поскольку это всего лишь пример и 100 000 SKU у нас нет, начнем с создания записей о фиктивных товарах и SKU в базе данных. Случайным образом сгенерируем 100 000 SKU для случайных марок и товаров, а затем используем этот набор данных как основу для запросов.

5.5.1 Вставка случайных SKU в базу данных о товарах

Поскольку мы не хотим вводить марки, товары и SKU самостоятельно, сгенерируем их случайным образом. Выберем случайные названия из списка 1000 самых часто встречающихся английских слов. Эти слова находятся в текстовом файле `common_words.txt`, который можно скачать из репозитория книги на GitHub по адресу <https://github.com/concurrency-in-python-withasyncio/data>.

Первым делом вставим марки, потому что в таблице товаров `brand_id` является внешним ключом. Воспользуемся для этой цели сопрограммой `connection.executemany`, которая выполняет параметризованную SQL-команду и позволит нам написать один запрос и передать ему список вставляемых записей в виде параметров, а не создавать по одной команде `INSERT` для каждой марки.

Сопрограмма `executemany` принимает одну SQL-команду и список кортежей, содержащих вставляемые значения. Параметры представлены маркерами `$1`, `$2` ... `$N`. Число после знака доллара равно индексу элемента кортежа. Например, если имеется запрос `"INSERT INTO table`

VALUES(\$1, \$2)" и список кортежей [('a', 'b'), ('c', 'd')], то будут выполнены две команды вставки:

```
INSERT INTO table ('a', 'b')
INSERT INTO table ('c', 'd')
```

Сначала сгенерируем список 100 случайных марок из списка частых слов и вернем его в виде списка кортежей по одному значению в каждом, чтобы его можно было сразу передать сопрограмме `executemany` и таким образом выполнить команды `INSERT`.

Листинг 5.5 Вставка случайных марок

```
import asyncpg
import asyncio
from typing import List, Tuple, Union
from random import sample

def load_common_words() -> List[str]:
    with open('common_words.txt') as common_words:
        return common_words.readlines()

def generate_brand_names(words: List[str]) -> List[Tuple[Union[str, ]]]:
    return [(words[index],) for index in sample(range(100), 100)]

async def insert_brands(common_words, connection) -> int:
    brands = generate_brand_names(common_words)
    insert_brands = "INSERT INTO brand VALUES(DEFAULT, $1)"
    return await connection.executemany(insert_brands, brands)

async def main():
    common_words = load_common_words()
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    await insert_brands(common_words, connection)

asyncio.run(main())
```

За кулисами `executemany` в цикле обходит список марок и генерирует по одной команде `INSERT` для каждой марки. Затем она выполняет сразу все эти команды. Заодно этот метод параметризации предохраняет нас от атак внедрением SQL, поскольку входные данные надлежащим образом экранируются. По завершении мы будем иметь в системе 100 марок со случайными названиями.

Разобравшись со вставкой случайных марок, воспользуемся той же техникой для вставки товаров и SKU. Для товаров создадим описание, составленное из 10 случайных слов и случайного идентификатора марки. Для SKU случайно выберем размер, цвет и товар. Будем предполагать, что идентификаторы марок принимают значения от 1 до 100.

Листинг 5.6 Вставка случайных товаров и SKU

```

import asyncio
import asyncpg
from random import randint, sample
from typing import List, Tuple
from chapter_05.listing_5_5 import load_common_words

def gen_products(common_words: List[str],
                 brand_id_start: int,
                 brand_id_end: int,
                 products_to_create: int) -> List[Tuple[str, int]]:
    products = []
    for _ in range(products_to_create):
        description = [common_words[index] for index in sample(range(1000), 10)]
        brand_id = randint(brand_id_start, brand_id_end)
        products.append((" ".join(description), brand_id))
    return products

def gen_skus(product_id_start: int,
             product_id_end: int,
             skus_to_create: int) -> List[Tuple[int, int, int]]:
    skus = []
    for _ in range(skus_to_create):
        product_id = randint(product_id_start, product_id_end)
        size_id = randint(1, 3)
        color_id = randint(1, 2)
        skus.append((product_id, size_id, color_id))
    return skus

async def main():
    common_words = load_common_words()
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    product_tuples = gen_products(common_words,
                                   brand_id_start=1,
                                   brand_id_end=100,
                                   products_to_create=1000)
    await connection.executemany("INSERT INTO product VALUES(DEFAULT, $1, $2)",
                                  product_tuples)
    sku_tuples = gen_skus(product_id_start=1,
                           product_id_end=1000,
                           skus_to_create=100000)
    await connection.executemany("INSERT INTO sku VALUES(DEFAULT, $1, $2, $3)",
                                  sku_tuples)
    await connection.close()

asyncio.run(main())

```

В результате выполнения этой программы мы получим базу данных, содержащую 1000 товаров и 100 000 SKU. Вся процедура может занять несколько секунд, точное время зависит от машины. Теперь, написав запрос с несколькими соединениями, мы сможем запросить все имеющиеся SKU для конкретного товара. Для product id 100 этот запрос выглядит так:

```
product_query = \
"""
SELECT
p.product_id,
p.product_name,
p.brand_id,
s.sku_id,
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_color_id
JOIN product_size as ps on ps.product_size_id = s.product_size_id
WHERE p.product_id = 100"""
```

В результате выполнения этого запроса мы получим по одной строке для каждого SKU, связанного с товаром, а также английские названия цвета и размера вместо идентификаторов. В предположении, что в каждый момент времени запрашивается информация о многих товарах, этот запрос является прекрасным кандидатом для конкурентного выполнения. Можно наивно применить функцию `asyncio.gather`, указав одно подключение:

```
async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    print('Creating the product database...')
    queries = [connection.execute(product_query),
               connection.execute(product_query)]
    results = await asyncio.gather(*queries)
```

Но при попытке выполнить это мы получим сообщение об ошибке:

```
RuntimeError: readexactly() called while another coroutine is already waiting
for incoming data
```

Что это значит? В SQL одному подключению к базе соответствует один сокет. Поскольку подключение всего одно, а мы пытаемся прочитать одновременно результаты нескольких запросов, естественно, возникает ошибка. Проблема можно решить, создав несколько подключений к базе, по одному на каждый запрос. Но создание подклю-

чений обходится дорого, поэтому имеет смысл кешировать их и получать из кеша по мере необходимости. Обычно такой кеш называется *пулом подключений*.

5.5.2 Создание пула подключений для конкурентного выполнения запросов

Поскольку в каждый момент времени на одном подключении можно выполнять только один запрос, нам необходим механизм для создания нескольких подключений и управления ими. Именно таким механизмом и является пул подключений. Мы можем рассматривать его как кеш существующих подключений к экземпляру сервера базы данных. В пуле находится конечное число подключений, которые можно *захватывать* по мере необходимости.

Захват подключения означает, что мы спрашиваем у пула: «У тебя есть сейчас свободные подключения? Если да, дай мне одно, чтобы я мог выполнить запрос». Пулы обеспечивают повторное использование подключений. Иными словами, если мы захватываем подключение из пула для выполнения запроса, то после завершения запроса подключение «освобождается», т. е. возвращается в пул. Это важно, потому что создание подключения к базе данных – процесс небыстрый. Если бы нужно было создавать подключения для каждого запроса, то производительность приложения резко снизилась бы.

Поскольку число подключений в пуле конечно, возможно, придется немного подождать освобождения подключения, если все они заняты. Поэтому операция захвата подключения может занять некоторое время. Если в пуле всего 10 подключений и все они используются, то придется подождать, пока одно освободится для выполнения нашего запроса.

Для иллюстрации того, как это работает в терминах *asyncio*, допустим, что имеется пул с двумя подключениями. Допустим также, что имеется три сопрограммы, каждая из которых выполняет запрос. Будем запускать их конкурентно, как задачи. При таких настройках пула первые две сопрограммы успешно захватят оба имеющихся подключения и начнут выполнять свои запросы. А третья сопрограмма в это время будет ждать освобождения подключения. Когда какая-нибудь из первых двух сопрограмм закончит выполнять запрос, она освободит свое подключение и вернет его в пул. В этот момент третья сопрограмма захватит подключение и начнет выполнять запрос (рис. 5.2).

В этой модели одновременно может выполняться не более двух запросов. Обычно пул подключений делают гораздо больше, чтобы увеличить степень конкурентности. В наших примерах пул будет содержать 6 подключений, но вообще их число зависит от оборудования, на котором работают база данных и приложение. Следует провести тестирование производительности и найти оптимальный размер

пула. Помните, что больше не всегда лучше, но это отдельная и очень обширная тема.

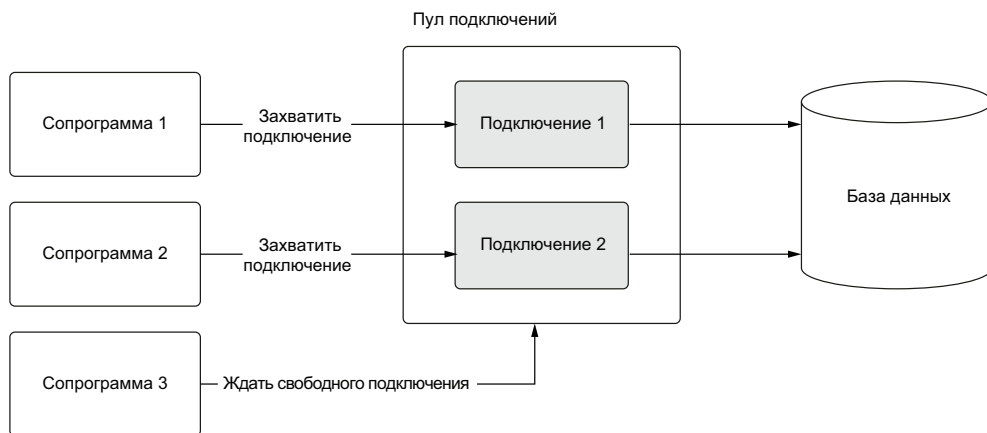


Рис. 5.2 Сопрограммы 1 и 2 захватывают подключения, чтобы выполнить свои запросы, а сопрограмма 3 в это время ждет подключения. Как только сопрограмма 1 или 2 завершится, сопрограмма 3 сможет воспользоваться освободившимся подключением и выполнить свой запрос

Итак, вы теперь понимаете, как работает пул подключений. Но как создать его в `asyncpg`? Для этой цели `asyncpg` предлагает сопрограмму `create_pool`. Мы воспользуемся ей вместо функции `connect`, которую раньше использовали для подключения к базе данных. При вызове `create_pool` будем задавать желаемое количество подключений в пуле с помощью параметров `min_size` и `max_size`. Параметр `min_size` определяет минимальное число подключений, т. е. гарантируется, что столько подключений в пуле всегда будет. Параметр `max_size` определяет максимальное число подключений. Если подключений недостаточно, то пул попытается создать еще одно при условии, что число подключений не превысит `max_size`. В первом примере мы зададим оба значения равными 6, т. е. пул будет гарантированно содержать шесть подключений.

Пулы `asyncpg` являются *асинхронными контекстными менеджерами*, т. е. для создания пула нужно использовать конструкцию `async with`. После того как пул создан, мы можем захватывать соединения с помощью сопрограммы `acquire`. Она приостанавливается, до тех пор пока в пуле не появится свободное подключение. Затем это подключение можно использовать для выполнения SQL-запроса. Захват соединения также является асинхронным контекстным менеджером, который возвращает соединение в пул после использования, так что и в этом случае нужна конструкция `async with`. Теперь можно переписать код, так чтобы несколько запросов выполнялось конкурентно.

Листинг 5.7 Создание пула подключений и конкурентное выполнение запросов

```
import asyncio
import asyncpg

product_query = \
    """
SELECT
p.product_id,
p.product_name,
p.brand_id,
s.sku_id,
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_color_id
JOIN product_size as ps on ps.product_size_id = s.product_size_id
WHERE p.product_id = 100"""

async def query_product(pool):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)

async def main():
    async with asyncpg.create_pool(host='127.0.0.1',
                                   port=5432,
                                   user='postgres',
                                   password='password',
                                   database='products',
                                   min_size=6,
                                   max_size=6) as pool:
        await asyncio.gather(query_product(pool),
                              query_product(pool))

asyncio.run(main())
```

Создать пул с шестью подключениями

Конкурентно выполнить два запроса

Здесь мы сначала создаем пул с шестью подключениями. Затем создается два объекта сопрограмм выполнения запросов, конкурентная работа которых планируется с помощью `asyncio.gather`. Сопрограмма `query_product` сначала захватывает подключение из пула, вызывая метод `pool.acquire()`. Затем она приостанавливается, до тех пор пока не освободится подключение. Это делается в блоке `async with`; тем самым гарантируется, что по выходе из блока подключение будет возвращено в пул. Это важно, потому что в противном случае подключения быстро закончились бы и приложение зависло бы в ожидании подключения, которого никогда не получит. Захватив подключение, мы можем выполнить запрос, как в предыдущих примерах.

Можно увеличить количество запросов в этом примере до 10 000, создав 10 000 объектов сопрограмм. Чтобы было интереснее, на-

пишем версию, которая выполняет эти запросы синхронно, и сравним время.

Листинг 5.8 Синхронное и конкурентное выполнение запросов

```
import asyncio
import asyncpg
from util import async_timed

product_query = \
    """
SELECT
p.product_id,
p.product_name,
p.brand_id,
s.sku_id,
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_color_id
JOIN product_size as ps on ps.product_size_id = s.product_size_id
WHERE p.product_id = 100"""

async def query_product(pool):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)

@async_timed()
async def query_products_synchronously(pool, queries):
    return [await query_product(pool) for _ in range(queries)]

@async_timed()
async def query_products_concurrently(pool, queries):
    queries = [query_product(pool) for _ in range(queries)]
    return await asyncio.gather(*queries)

async def main():
    async with asyncpg.create_pool(host='127.0.0.1',
                                   port=5432,
                                   user='postgres',
                                   password='password',
                                   database='products',
                                   min_size=6,
                                   max_size=6) as pool:
        await query_products_synchronously(pool, 10000)
        await query_products_concurrently(pool, 10000)

asyncio.run(main())
```

В сопрограмме `query_products_synchronously` мы поместили `await` в списковое включение, благодаря чему все вызовы `query_product` выполняются последовательно. А в сопрограмме `query_products_con-`

currently создается список сопрограмм, подлежащих выполнению, после чего все они запускаются конкурентно с помощью gather. В сопрограмме main мы выполняем синхронную и конкурентную версии с 10 000 запросами. Точные результаты могут варьироваться в зависимости от оборудования, но в общем конкурентная версия примерно в пять раз быстрее последовательной:

```
выполняется <function query_products_synchronously at 0x1219ea1f0> с аргументами
(<asynpcrg.pool.Pool object at 0x12164a400>, 10000) {}
<function query_products_synchronously at 0x1219ea1f0> завершилась за 21.8274 с
выполняется <function query_products_concurrently at 0x1219ea310> с аргументами
(<asynpcrg.pool.Pool object at 0x12164a400>, 10000) {}
<function query_products_concurrently at 0x1219ea310> завершилась за 4.8464 с
```

Это значительное улучшение, но можно добиться большего. Поскольку наш запрос относительно быстрый, этот код является смесью счетного и зависящего от производительности ввода-вывода. В главе 6 мы увидим, как в этой ситуации выжать дополнительную производительность.

До сих пор мы рассматривали вставку данных в базу при условии отсутствия ошибок. Но что, если на середине процесса вставки товаров произойдет ошибка? Мы не хотим получить несогласованное состояния базы данных, поэтому самое время поговорить о транзакциях. Далее мы увидим, как использовать асинхронные контекстные менеджеры, чтобы начать транзакцию и управлять ей.

5.6 Управление транзакциями в asynpcrg

Транзакции – ключевая концепция во многих базах данных, обладающих свойствами ACID (атомарность, согласованность, изолированность, долговечность). *Транзакция* включает одну или несколько SQL-команд, выполняемых как неделимое целое. Если при выполнении этих команд не возникло ошибки, то транзакция *фиксируется* в базе данных, так что изменения становятся постоянными. Если же ошибки были, то транзакция *откатывается* и база данных выглядит так, будто ни одна из команд не выполнялась. В случае нашей базы данных о товарах необходимость в откате набора обновлений может возникнуть, если мы попытаемся вставить дубликат марки или нарушим установленное ограничение целостности.

В asynpcrg для работы с транзакциями проще всего воспользоваться асинхронным контекстным менеджером `connection.transaction`, который начинает транзакцию, а затем, если в блоке `async with` произошло исключение, автоматически откатывает ее. Если же все команды выполнены успешно, то транзакция автоматически фиксируется. Рассмотрим, как создать транзакцию и выполнить две простые команды `insert` для добавления двух марок.

Листинг 5.9 Создание транзакции

```

import asyncio
import asyncpg

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    async with connection.transaction():
        await connection.execute("INSERT INTO brand "
                                "VALUES(DEFAULT, 'brand_1')")
        await connection.execute("INSERT INTO brand "
                                "VALUES(DEFAULT, 'brand_2')")

    query = """SELECT brand_name FROM brand
               WHERE brand_name LIKE 'brand%'"""
    brands = await connection.fetch(query)
    print(brands)

    await connection.close()

asyncio.run(main())

```

Начать транзакцию
базы данных

Выбрать марки и убедиться,
что транзакция была зафиксирована

В предположении, что транзакция успешно зафиксирована, мы должны увидеть на консоли сообщение [`<Record brand_name='brand_1'>`, `<Record brand_name='brand_2'>`]. Для демонстрации того, что происходит при откате, сделаем в SQL-коде ошибку, а именно попытаемся вставить две марки с одинаковым первичным ключом `id`. Первая команда вставки выполнится успешно, но вторая возбудит исключение из-за дубликата ключа.

Листинг 5.10 Обработка ошибки в транзакции

```

import asyncio
import logging
import asyncpg

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    try:
        async with connection.transaction():
            insert_brand = "INSERT INTO brand VALUES(9999, 'big_brand')"
            await connection.execute(insert_brand)
            await connection.execute(insert_brand)
    except Exception:

```

Команда insert завершится
неудачно из-за дубликата
первичного ключа

```

        logging.exception('Ошибка при выполнении транзакции')
    finally:
        query = """SELECT brand_name FROM brand
                    WHERE brand_name LIKE 'big_%'"""
        brands = await connection.fetch(query)
        print(f'Результат запроса: {brands}')

        await connection.close()

asyncio.run(main())

```

Если было исключение, протоколировать ошибку

Выбрать марки и убедиться, что ничего не вставлено

Вторая команда возбудила исключение, и вот что мы увидим:

```

ERROR:root:Ошибка при выполнении транзакции
Traceback (most recent call last):
  File "listing_5_10.py", line 16, in main
    await connection.execute("INSERT INTO brand "
  File "asynpg/connection.py", line 272, in execute
    return await self._protocol.query(query, timeout)
  File "asynpg/protocol/protocol.pyx", line 316, in query
    asynpg.exceptions.UniqueViolationError: duplicate key value violates unique
    constraint "brand_pkey"
DETAIL: Key (brand_id)=(9999) already exists.
Query result was: []

```

Сначала возникло исключение, потому что мы пытались вставить дубликат ключа, а затем мы видим, что результат команды `select` пуст, т. е. мы успешно откатали транзакцию.

5.6.1 Вложенные транзакции

Asynpg поддерживает также *вложенные транзакции* благодаря имеющемуся в Postgres механизму *точек сохранения*, которые определяются командой `SAVEPOINT`. Если определена точка сохранения, то мы можем откатиться к ней, т. е. все запросы, выполненные после точки сохранения, откатываются, а те, что были до нее, – нет.

В asynpg для создания точки сохранения вызывается контекстный менеджер `connection.transaction`, которому передается существующая транзакция. Затем если во внутренней транзакции произошла ошибка, то она откатывается, но внешняя транзакция при этом не затрагивается. Посмотрим, как это работает, для чего вставим внутри транзакции марку, а затем во вложенной транзакции попытаемся вставить уже существующий в базе цвет.

Листинг 5.11 Вложенная транзакция

```

import asyncio
import asynpg
import logging

async def main():

```

```
connection = await asyncpg.connect(host='127.0.0.1',
                                    port=5432,
                                    user='postgres',
                                    database='products',
                                    password='password')

async with connection.transaction():
    await connection.execute("INSERT INTO brand VALUES(DEFAULT, 'my_new_brand')")

    try:
        async with connection.transaction():
            await connection.execute("INSERT INTO product_color VALUES(1, 'black')")
    except Exception as ex:
        logging.warning('Ошибка при вставке цвета товара игнорируется', exc_info=ex)

await connection.close()

asyncio.run(main())
```

Здесь первая команда INSERT выполнена успешно, потому что такой марки в базе еще нет. А при выполнении второй команды INSERT возникает ошибка дубликата ключа. Поскольку вторая команда находится внутри транзакции и мы перехватили и запротоколировали исключение, то, несмотря на ошибку, внешняя транзакция не откатывается и новая марка вставляется в базу. Не будь вложенной транзакции, ошибка во второй команде вставки привела бы к откату вставки марки.

5.6.2 Ручное управление транзакциями

До сих пор для фиксации и отката транзакций мы использовали асинхронные контекстные менеджеры. Поскольку это короче, чем управлять самостоятельно, то такой подход обычно рекомендуется. Но иногда бывают ситуации, когда транзакцией приходится управлять вручную. Например, если мы хотим выполнить специальный код при откате или произвести откат по условию, отличному от исключения.

Для ручного управления транзакцией мы можем воспользоваться менеджером транзакций, возвращенным методом `connection.transaction`, вне контекстного менеджера. При этом нужно вручную вызывать его метод `start`, чтобы начать транзакцию, а затем метод `commit` в случае успешного завершения или `rollback` в случае ошибки. Продемонстрируем это, для чего перепишем наш первый пример.

Листинг 5.12 Ручное управление транзакцией

```
import asyncio
import asyncpg
from asyncpg.transaction import Transaction

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
```

```

port=5432,
user='postgres',
database='products',
password='password')

```

Начать транзакцию →

```

transaction: Transaction = connection.transaction()
await transaction.start()
try:
    await connection.execute("INSERT INTO brand "
                              "VALUES(DEFAULT, 'brand_1')")
    await connection.execute("INSERT INTO brand "
                              "VALUES(DEFAULT, 'brand_2')")
except asyncpg.PostgresError:
    print('Ошибка, транзакция откатывается!')
    await transaction.rollback()
else:
    print('Ошибки нет, транзакция фиксируется!')
    await transaction.commit()

```

Создать экземпляр транзакции ←

Если было исключение, откатить ←

Если исключения не было, зафиксировать ←

```

query = """SELECT brand_name FROM brand
            WHERE brand_name LIKE 'brand%'"""
brands = await connection.fetch(query)
print(brands)

await connection.close()

asyncio.run(main())

```

Мы начинаем с того, что создаем транзакцию тем же методом, который использовали при работе с асинхронным контекстным менеджером, но теперь сохраняем возвращенный экземпляр класса `Transaction`. Этот класс можно рассматривать как менеджер нашей транзакции, поскольку он умеет фиксировать и откатывать транзакцию по мере необходимости. Имея экземпляр транзакции, мы вызываем сопрограмму `start`. Она выполняет запрос к `Postgres`, необходимый, чтобы начать транзакцию. Затем в блоке `try` можем выполнять произвольные запросы. В данном случае мы вставляем две марки. Если хотя бы одна команда `INSERT` завершится ошибкой, то мы попадем в блок `except` и откатим транзакцию, вызвав сопрограмму `rollback`. Если же ошибок не было, то вызываем сопрограмму `commit`, которая завершает транзакцию и делает все изменения в базе данных постоянными.

До сих пор мы выполняли запросы так, что все результаты сразу загружались в память. Во многих приложениях это нормально, потому что запросы, как правило, возвращают небольшие результирующие наборы. Но бывает, что результирующий набор настолько велик, что целиком в память не помещается. В таком случае мы хотели бы обрабатывать результаты потоком, чтобы снизить нагрузку на оперативную память системы. Далее рассмотрим, как сделать это с помощью `asynpcrg`, а попутно познакомимся с асинхронными генераторами.

5.7 Асинхронные генераторы и потоковая обработка результирующих наборов

У реализации `fetch` по умолчанию в `asynprg` есть один недостаток: она загружает все возвращенные по запросу данные в память. Следовательно, если запрос возвращает миллионы строк, то мы попытаемся скопировать весь этот набор из базы данных на запросившую машину. Представьте, что наш бизнес оказался сверхуспешным и в базе данных хранятся миллиарды товаров. Весьма вероятно, что некоторые запросы будут возвращать очень большие результирующие наборы, что может снизить производительность.

Конечно, мы могли бы включить в запрос фразу `LIMIT` и разбить результаты на страницы, и для многих, а то и для большинства приложений это имеет смысл. Однако у этого подхода есть недостаток – мы отправляем один и тот же запрос несколько раз, что может создать излишнюю нагрузку на базу данных. Если в нашем приложении это действительно проблема, то стоит обрабатывать результаты запросов потоком. Это снизит как потребление памяти на уровне приложения, так и нагрузку на базу данных. Однако расплачиваться приходится дополнительными раундами обращения к базе по сети.

Postgres поддерживает потоковую обработку результатов запроса с помощью курсоров. Курсор можно рассматривать как указатель на текущую позицию в результирующем наборе. Получая один результат из потокового запроса, мы продвигаем курсор на следующую позицию и т. д., пока результаты не будут исчерпаны.

В `asynprg` получить курсор можно непосредственно от подключения, а затем использовать для потоковой обработки запроса. В реализации курсоров используется средство `asynprg`, которое нам еще не встречалось, – *асинхронные генераторы*. Асинхронные генераторы порождают результаты асинхронно по одному, как и обычные генераторы в Python. Они также позволяют использовать специальный синтаксис цикла `for` для обхода результатов. Чтобы лучше разобраться в том, как это работает, давайте сначала познакомимся с асинхронными генераторами и с синтаксической конструкцией `asynprg for` для их обхода.

5.7.1 Введение в асинхронные генераторы

Многие разработчики знакомы с генераторами по синхронному коду на Python. Генераторы – это реализация паттерна проектирования Итератор, получившего известность благодаря книге «банды четырех» «Паттерны проектирования» (Addison-Wesley Professional, 1994). Этот паттерн позволяет «лениво» определять последовательности данных и обходить их поэлементно. Это полезно, когда последовательность потенциально велика и сохранить ее в памяти целиком невозможно. Простой синхронный генератор – это обычная функция

Python, содержащая предложение `yield` вместо `return`. Например, вот как можно создать и использовать генератор, возвращающий положительные целые числа, начиная с нуля и до заданной границы.

Листинг 5.13 Синхронный генератор

```
def positive_integers(until: int):
    for integer in range(until):
        yield integer

positive_iterator = positive_integers(2)

print(next(positive_iterator))
print(next(positive_iterator))
```

Здесь функция `positive_integers` принимает целое число, до которого нужно посчитать. В ней мы входим в цикл, продолжающийся, пока не будет достигнуто заданное число. На каждой итерации мы отдаем следующее целое число в предложении `yield`. При вызове `positive_integers(2)` мы не возвращаем список целиком и даже не выполняем цикл. Запросив тип `positive_iterator`, мы получим в ответ `<class 'generator'>`.

Затем используем функцию `next` для обхода генератора. При каждом ее вызове выполняется одна итерация цикла `for` в `positive_integers`, и мы получаем результат предложения `yield`. Таким образом, код в листинге 5.13 напечатает на консоли 0 и 1. Вместо `next` мы могли использовать совместно с генератором цикл `for`, чтобы перебрать все возвращаемые генератором значения.

Для синхронных методов все это работает, но как быть, если мы хотим использовать сопрограммы для асинхронного порождения последовательности значений? В нашем примере с базой данных как можно «лениво» получить данные из базы? Для этого в Python имеются асинхронные генераторы и специальная конструкция `async for`. Для демонстрации простого асинхронного генератора реализуем тот же пример порождения целых чисел, но включим в него сопрограмму, которая работает несколько секунд. Для этого воспользуемся функцией `delay` из главы 2.

Листинг 5.14 Простой асинхронный генератор

```
import asyncio
from util import delay, async_timed

async def positive_integers_async(until: int):
    for integer in range(1, until):
        await delay(integer)
        yield integer

@async_timed()
async def main():
    async_generator = positive_integers_async(3)
```

```
print(type(async_generator))
async for number in async_generator:
    print(f'Получено число {number}')

asyncio.run(main())
```

Как видим, это не обычный генератор, а объект типа `<class 'async_generator'>`. Асинхронный генератор отличается от обычного тем, что отдает не объекты Python, а генерирует сопрограммы, которые могут ждать получения результата с помощью `await`. Поэтому обычные циклы `for` и функция `next` с такими генераторами работать не будут. А вместо них предложена специальная синтаксическая конструкция `async for`. В данном примере мы использовали ее, чтобы обойти целые числа в сопрограмме `positive_integers_async`.

Этот код напечатает числа 1 и 2, но будет ждать 1 с перед возвратом первого числа и 2 с перед возвратом второго. Отметим, что генератор не выполняет порожденные сопрограммы конкурентно, а порождает и ждет их одну за другой.

5.7.2 Использование асинхронных генераторов и потокового курсора

Понятие асинхронного генератора прекрасно сочетается с понятием *потокового курсора базы данных*. С помощью таких генераторов мы можем получать по одной строке за раз в простом цикле, похожем на `for`. В `asyncpg` для выполнения потоковой обработки нужно сначала открыть транзакцию, поскольку таково требование Postgres. Затем мы можем вызвать метод `cursor` класса `Connection` и получить курсор. Методу `cursor` передается запрос, а возвращает он асинхронный генератор для потоковой обработки результатов. Для примера выполним запрос, который получает на курсоре все товары, имеющиеся в базе данных. И будем выбирать элементы из результирующего набора по одному в цикле `async for`.

Листинг 5.15 Потоковая обработка результатов

```
import asyncpg
import asyncio
import asyncpg

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    query = 'SELECT product_id, product_name FROM product'
    async with connection.transaction():
        async for product in connection.cursor(query):
```

```

        print(product)

    await connection.close()

asyncio.run(main())

```

Здесь распечатываются все имеющиеся товары. Хотя в таблице хранится 1000 товаров, в память загружается лишь небольшая порция. На момент написания книги объем предвыборки по умолчанию был равен 50 записей, чтобы уменьшить затраты на сетевой трафик. Это значение можно изменить, задав параметр `prefetch`.

Курсор можно использовать и для того, чтобы извлечь произвольное число записей из середины результирующего набора, как показано в листинге ниже.

Листинг 5.16 Перемещение по курсору и выборка записей

```

import asyncpg
import asyncio

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    async with connection.transaction():
        query = 'SELECT product_id, product_name from product'
        cursor = await connection.cursor(query)
        await cursor.forward(500)
        products = await cursor.fetch(100)
        for product in products:
            print(product)

    await connection.close()

asyncio.run(main())

```

Создать курсор для запроса

Сдвинуть курсор вперед на 500 записей

Получить следующие 100 записей

Здесь мы сначала создаем курсор для запроса. Обратите внимание, что это делается в предложении `await`, как для сопрограммы, а не асинхронного генератора; это возможно, потому что в `asyncpg` курсор является одновременно асинхронным генератором и объектом, допускающим ожидание. В большинстве случаев оба способа похожи, но при таком создании курсора есть различия в поведении предвыборки – мы не можем задать количество выбираемых за одно обращение записей; попытка сделать это приведет к исключению `InterfaceError`.

Получив курсор, мы вызываем его метод-сопрограмму `forward`, чтобы сдвинуться вперед по результирующему набору. В результате мы пропустим первые 500 записей в таблице товаров. Затем выбираем следующие 100 товаров и печатаем их на консоли.

По умолчанию такие курсоры не допускают прокрутки, т. е. двигаться по результирующему набору можно только вперед. Если нужны прокручиваемые курсоры, допускающие смещение вперед и назад, то следует самостоятельно выполнить SQL-команду `DECLARE ... SCROLL CURSOR` (подробнее о том, как это делается, можно прочитать в документации Postgres по адресу <https://www.postgresql.org/docs/current/plpgsql-cursors.html>).

Оба метода полезны, когда результирующий набор велик и мы не хотим загружать его в память целиком. Циклы `async for` (листинг 5.16) хороши для обхода всего набора, тогда как создание курсора и метод `fetch` удобны для выборки порции записей или пропуска части результирующего набора.

Но что, если нам только и нужно, что выбрать фиксированное число элементов с предвыборкой, но при этом использовать цикл `async for`? Можно было бы добавить в цикл `async for` счетчик и выходить из цикла после получения некоторого числа элементов, но такой подход не приспособлен для повторного использования. Если такие действия производятся в программе часто, то лучше написать свой собственный асинхронный генератор, который мы назовем `take`. Он будет принимать асинхронный генератор и число элементов. Покажем, как это делается, на примере выборки первых пяти элементов из результирующего набора.

Листинг 5.17 Получение заданного числа элементов с помощью асинхронного генератора

```
import asyncpg
import asyncio

async def take(generator, to_take: int):
    item_count = 0
    async for item in generator:
        if item_count > to_take - 1:
            return
        item_count = item_count + 1
        yield item

async def main():
    connection = await asyncpg.connect(host='127.0.0.1',
                                       port=5432,
                                       user='postgres',
                                       database='products',
                                       password='password')

    async with connection.transaction():
        query = 'SELECT product_id, product_name from product'
        product_generator = connection.cursor(query)

        async for product in take(product_generator, 5):
            print(product)
```

```
print('Получены первые пять товаров!')

await connection.close()

asyncio.run(main())
```

Наш асинхронный генератор `take` хранит число уже отданных элементов в переменной `item_count`. В цикле `async_for` он отдает нам записи с помощью предложения `yield` и, как только будет отдано `item_count` элементов, выполняет `return` и тем самым завершает работу. А в сопрограме `main` можно использовать `take` в цикле `async_for`, как обычно. В данном случае мы запрашиваем у курсора первые пять элементов и видим следующую картину:

```
<Record product_id=1 product_name='among paper foot see shoe ride age'>
<Record product_id=2 product_name='major wait half speech lake won't'>
<Record product_id=3 product_name='war area speak listen horse past edge'>
<Record product_id=4 product_name='smell proper force road house planet'>
<Record product_id=5 product_name='ship many dog fine surface truck'>
```

Получены первые пять товаров!

Мы написали этот код самостоятельно, но в библиотеке *aiostream* с открытым исходным кодом есть и эта функциональность, и многое другое для работы с асинхронными генераторами. Можете почитать документацию по адресу *aiostream.readthedocs.io*.

Резюме

В этой главе мы изучили основы создания и выборки записей в Postgres с применением асинхронного подключения к базе данных. Зная все это, вы теперь можете писать конкурентные клиенты базы данных.

- Мы узнали, как использовать `asynscrpg` для подключения к базе данных Postgres.
- Мы научились использовать `asynscrpg` для создания таблиц, вставки записей и исполнения отдельных запросов.
- Мы узнали, как создавать пул подключений в `asynscrpg`. Это позволяет конкурентно выполнять несколько запросов, используя такие методы, как `gather`, и тем самым повысить скорость работы приложения.
- Мы научились управлять транзакциями с помощью `asynscrpg`. Транзакции позволяют откатывать изменения в случае ошибки базы данных, поддерживая базу данных в согласованном состоянии.
- Мы научились создавать асинхронные генераторы и использовать их для потоковой обработки запросов. Сочетание того и другого позволяет работать с большими наборами данных, не помещающимися в память.

Счетные задачи

Краткое содержание главы

- Библиотека `multiprocessing`.
- Создание пулов процессов для счетных задач.
- Использование `async` и `await` для управления счетными задачами.
- Использование MapReduce для решения счетной задачи с помощью `asyncio`.
- Разделение данных между процессами с применением блокировок.
- Повышение скорости работы программ, содержащих операции, ограниченные быстродействием процессора и производительностью ввода-вывода.

До сих пор нас интересовало, какой выигрыш позволяет получить `asyncio` при конкурентном выполнении операций ввода-вывода. Ввод-вывод – то, для чего `asyncio` создавалась в первую очередь, а описанный выше способ написания кода требует внимательно следить за тем, чтобы в сопрограммах не было счетного кода. На первый взгляд, это серьезно ограничивает `asyncio`, но на самом деле библиотека более универсальна.

В `asyncio` имеется API для взаимодействия с библиотекой Python `multiprocessing`. Это позволяет использовать предложение `await`

и различные API `asyncio` при работе с несколькими процессами и таким образом получать выигрыш в производительности даже при выполнении счетного кода, например математических вычислений или обработки данных. При этом мы обходим глобальную блокировку интерпретатора и можем воспользоваться всеми преимуществами многоядерной машины.

В этой главе мы сначала рассмотрим модуль `multiprocessing` и познакомимся с идеей выполнения нескольких процессов. Затем поговорим об *исполнителях пула процессов* и о том, как связать их с `asyncio`. Полученными знаниями мы воспользуемся, чтобы решить счетную задачу с применением техники MapReduce. Также узнаем об управлении разделяемым между несколькими процессами состоянием и о том, как блокировки позволяют избежать ошибок конкурентного выполнения. Наконец, мы увидим, как многопроцессность может повысить производительность приложения из главы 5, в котором встречаются как счетные операции, так и операции ввода-вывода.

6.1 Введение в библиотеку `multiprocessing`

В главе 1 мы познакомились с глобальной блокировкой интерпретатора. Она препятствует параллельному выполнению нескольких участков байт-кода. Это означает, что для любых задач, кроме ввода-вывода и еще нескольких мелких исключений, многопоточность не дает никакого выигрыша в производительности, – в отличие от таких языков, как Java и C++. Кажется, что мы уперлись в тупик, и для распараллеливания счетных задач Python не предлагает никакого решения, но на самом деле решение есть – библиотека `multiprocessing`.

Вместо запуска потоков для распараллеливания работы родительский процесс будет запускать дочерние процессы. В каждом дочернем процессе работает отдельный интерпретатор Python со своей GIL. В предположении, что код выполняется на машине с несколькими процессорными ядрами, это означает, что можно эффективно распараллелить счетные задачи. И даже если процессов больше, чем ядер, механизм вытесняющей многозадачности, встроенный в операционную систему, позволит выполнять задачи конкурентно. Такая конфигурация является конкурентной и параллельной.

Начнем с параллельного выполнения двух функций. Функция будет совсем простой – подсчет от нуля до большого числа, – но позволит понять, как работает API и какого выигрыша можно достичь.

Листинг 6.1 Два параллельных процесса

```
import time
from multiprocessing import Process

def count(count_to: int) -> int:
    start = time.time()
```



```

counter = 0
while counter < count_to:
    counter = counter + 1
end = time.time()
print(f'Закончен подсчет до {count_to} за время {end-start}')
return counter

if __name__ == "__main__":
    start_time = time.time()

    to_one_hundred_million = Process(target=count, args=(100000000,))
    to_two_hundred_million = Process(target=count, args=(200000000,))

    to_one_hundred_million.start()
    to_two_hundred_million.start()

    to_one_hundred_million.join()
    to_two_hundred_million.join()

    end_time = time.time()
    print(f'Полное время работы {end_time-start_time}')
```

Создать процесс для выполнения функции count

Запустить процесс. Этот метод возвращает управление немедленно

Ждать завершения процесса. Этот метод блокирует выполнение, пока процесс не завершится

Здесь мы создаем простую функцию count, которая принимает целое число и в цикле увеличивает его на единицу, пока не дойдет до переданной верхней границы. Затем создаем два процесса – один считает до 100 000 000, другой до 200 000 000. Класс Process принимает два аргумента: target – имя подлежащей выполнению функции и args – кортеж передаваемых ей аргументов. Затем для каждого процесса вызывается метод start. Он сразу же возвращает управление и начинает выполнять процесс. В данном примере оба процесса запускаются один за другим, после чего для каждого вызывается метод join. В результате главный процесс блокируется до тех пор, пока оба дочерних не завершатся. Если бы мы этого не сделали, то программа завершилась бы практически мгновенно и «прикончила» бы оба дочерних процесса, потому что никто не ждет их завершения. Программа в листинге 6.1 выполняет обе функции count конкурентно; в предположении, что компьютер оснащен по меньшей мере двумя ядрами, мы должны заметить ускорение работы. На 8-ядерной машине с тактовой частотой 2,5 ГГц мы получили следующие результаты:

```

Закончен подсчет до 100000000 за время 5.3844
Закончен подсчет до 200000000 за время 10.6265
Полное время работы 10.8586
```

Обе функции count суммарно заняли чуть больше 16 с, но наше приложение завершилось меньше чем за 11 с. То есть, по сравнению с последовательной версией, мы выиграли примерно 5 с. Конечно, на вашей машине результаты могут немного отличаться, но общая картина именно такова.

Обратите внимание на предложение if __name__ == "__main__":, с которым мы раньше не встречались. Это особенность библиотеки

multiprocessing; если этого не сделать, то возможно появление ошибки «An attempt has been made to start a new process before the current process has finished its bootstrapping phase» (Попытка запустить новый процесс до завершения фазы инициализации текущего процесса). Нужно это для того, чтобы помешать другим программам, импортирующим ваш код, случайно запустить несколько процессов.

Это дает неплохой выигрыш в производительности, но выглядит неэлегантно, потому что мы должны вызывать `start` и `join` для каждого запущенного процесса. Кроме того, мы не знаем, какой процесс закончится первым; если мы собираемся использовать нечто вроде `asyncio.as_completed` и обрабатывать результаты по мере завершения процесса, то нас поджидает фиаско. Метод `join` не возвращает то значение, которое вернула выполненная функция; на самом деле вообще не существует способа получить возвращенное ей значение без использования разделяемой памяти!

Этот API годится для простых случаев, но, очевидно, не работает, если нужно получать возвращенное функцией значение или обрабатывать результаты по мере готовности. По счастью, проблему позволяют решить пулы процессов.

6.2 Использование пулов процессов

В предыдущем примере мы вручную создавали процессы и вызывали их методы `start` и `join` для запуска и ожидания завершения. У этого подхода обнаружилось несколько недостатков, от качества кода до невозможности получить результаты, возвращенные процессом. Модуль `multiprocessing` предлагает API, позволяющий справиться с этими проблемами, – *пулы процессов*.

Пул процессов напоминает пул подключений, который мы видели в главе 5. Разница в том, что мы создаем не набор подключений к базе данных, а набор Python-процессов, который можно использовать для параллельного выполнения функций. Если в процессе требуется выполнить какую-то счетную функцию, то мы просим пул потоков подействовать. За кулисами он выполняет функцию в доступном процессе и возвращает ее значение. Для демонстрации создадим простой пул процессов и выполним в нем несколько функций в духе «hello world».

Листинг 6.2 Создание пула процессов

```
from multiprocessing import Pool

def say_hello(name: str) -> str:
    return f'Привет, {name}'

if __name__ == "__main__":
    with Pool() as process_pool:  ← Создать пул процессов
```

```

hi_jeff = process_pool.apply(say_hello, args=('Jeff',))
hi_john = process_pool.apply(say_hello, args=('John',))
print(hi_jeff)
print(hi_john)

```

← Выполнить say_hello с аргументом 'Jeff' в отдельном процессе и получить результат

Здесь мы создаем пул процессов в предложении `with Pool() as process_pool`. Это контекстный менеджер, потому что после завершения работы с пулом нужно корректно остановить созданные Python-процессы. Если этого не сделать, то возникает риск утечки ценного ресурса – процессов. При создании этого пула автоматически создается столько процессов, сколько имеется процессорных ядер на данной машине. Количество ядер можно получить от функции `multiprocessing.cpu_count()`. Если это не годится, то можно передать функции `Pool()` произвольное целое число в аргументе `processes`. Но обычно значение по умолчанию является неплохой отправной точкой.

Далее мы применяем метод пула процессов `apply`, чтобы выполнить функцию `say_hello` в отдельном процессе. Это похоже на то, что мы делали в классе `Process`, когда передавали подлежащую выполнению функцию и кортеж аргументов. Разница в том, что не нужно ни запускать процесс, ни вызывать `join` самостоятельно. Кроме того, мы получаем значение, возвращенное функцией, чего в предыдущем примере сделать не могли. Программа должна напечатать:

```

Привет, Jeff
Привет, John

```

Работать-то работает, но есть проблема. Метод `apply` блокирует выполнение, пока функция не завершится. Это значит, что если бы каждое обращение к `say_hello` занимало 10 с, то программа работала бы примерно 20 с, потому что выполнение последовательное и все усилия сделать ее параллельной пошли насмарку. Эту проблему можно решить, воспользовавшись методом пула процессов `apply_async`.

6.2.1 Асинхронное получение результатов

В предыдущем примере каждое обращение к `apply` блокировало программу до завершения функции. Это не годится, если мы хотим построить по-настоящему параллельный технологический процесс. Чтобы обойти эту трудность, мы можем применить метод `apply_async`. Он сразу же возвращает `AsyncResult` и начинает выполнять процесс в фоновом режиме. Имея объект `AsyncResult`, мы можем вызвать его метод `get` и получить результат. Модифицируем пример `say_hello`, так чтобы получать и использовать результаты асинхронно.

Листинг 6.3 Асинхронное получение результатов от пула процессов

```

from multiprocessing import Pool

def say_hello(name: str) -> str:
    return f'Привет, {name}'

```

```
if __name__ == "__main__":  
    with Pool() as process_pool:  
        hi_jeff = process_pool.apply_async(say_hello, args=('Jeff',))  
        hi_john = process_pool.apply_async(say_hello, args=('John',))  
        print(hi_jeff.get())  
        print(hi_john.get())
```

В случае использования `apply_async` оба вызова `say_hello` начинают работать в отдельных процессах. Затем, когда мы вызываем метод `get`, родительский процесс блокируется, пока каждый дочерний не вернет значение. Конкурентное выполнение мы обеспечили, но что, если `hi_jeff` занимает 10 с, а `hi_john` только одну? В таком случае, поскольку мы вызвали метод `get` объекта `hi_jeff` первым, программа зависнет на 10 с, прежде чем напечатать приветствие Джону, хотя готово оно было уже через 1 с. Если мы хотим реагировать сразу после получения результата, то следует признать, что проблема осталась. На самом деле нам нужно что-то типа функции `as_completed` из библиотеки `asyncio`. Далее мы увидим, как использовать исполнителей пула процессов в сочетании с `asyncio`, чтобы решить эту проблему.

6.3 Использование исполнителей пула процессов в сочетании с `asyncio`

Мы видели, как использовать пулы процессов для конкурентного выполнения счетных операций. Такие пулы хороши для простых случаев, но Python предлагает абстракцию поверх пула процессов в модуле `concurrent.futures`. Это модуль содержит *исполнители* для процессов и потоков, которые можно использовать как самостоятельно, так и в сочетании с `asyncio`. Начнем с рассмотрения основ класса `ProcessPoolExecutor`, похожего на `ThreadPool`. Затем посмотрим, как связать его с `asyncio`, чтобы можно было использовать всю мощь функций API, в частности `gather`.

6.3.1 Введение в исполнители пула процессов

API пула процессов в Python тесно связан с процессами, но многопроцессность – один из двух способов реализовать вытесняющую многозадачность; другим является многопоточность. Что, если мы хотим легко переходить от процессов к потокам и обратно? В таком случае требуется построить абстракцию передачи работы пулу ресурсов, которой было бы все равно, что такое ресурс: процесс, поток или еще что-то.

Модуль `concurrent.futures` предоставляет как раз такую абстракцию в виде абстрактного класса `Executor`, в котором определены два метода для асинхронного выполнения работы. Первый из них, `submit`, принимает вызываемый объект и возвращает объект `Future` (отметим, что это не то же самое, что будущие объекты `asyncio`, а часть модуля

`concurrent.futures`) – это эквивалент метода `Pool.apply_async`, рассмотренного в предыдущем разделе. Второй метод называется `map`. Он принимает вызываемый объект и список аргументов, после чего асинхронно выполняет объект с каждым из этих аргументов. Возвращается итератор по результатам вызовов. Это аналогично функции `asyncio.as_completed` в том смысле, что результаты становятся доступны по мере поступления. У класса `Executor` есть две конкретные реализации: `ProcessPoolExecutor` и `ThreadPoolExecutor`. Поскольку мы используем несколько процессов для выполнения счетных задач, сосредоточимся на классе `ProcessPoolExecutor`. А в главе 7 поговорим о потоках и классе `ThreadPoolExecutor`. Для демонстрации работы `ProcessPoolExecutor` снова возьмем пример с подсчетом чисел и выполним его для нескольких малых и нескольких больших границ, чтобы посмотреть, как появляются результаты.

Листинг 6.4 Исполнители пула процессов

```
import time
from concurrent.futures import ProcessPoolExecutor

def count(count_to: int) -> int:
    start = time.time()
    counter = 0
    while counter < count_to:
        counter = counter + 1
    end = time.time()
    print(f'Завершен подсчет до {count_to} за время {end - start}')
    return counter

if __name__ == "__main__":
    with ProcessPoolExecutor() as process_pool:
        numbers = [1, 3, 5, 22, 100000000]
        for result in process_pool.map(count, numbers):
            print(result)
```

Как и раньше, объект `ProcessPoolExecutor` создан под управлением контекстного менеджера. Количество ресурсов по умолчанию тоже равно числу процессорных ядер, как и в случае пула процессов. После этого мы используем метод `process_pool.map` для выполнения функции `count`, указывая список верхних границ.

Запустив эту программу, мы увидим, что обращения к `count` с небольшими верхними границами заканчиваются быстро и печатаются почти мгновенно. Но обращение с границей `100000000` занимает куда больше времени и печатается после небольших чисел, так что результат получается таким:

```
Завершен подсчет до 1 за время 9.5367e-07
Завершен подсчет до 3 за время 9.5367e-07
Завершен подсчет до 5 за время 9.5367e-07
Завершен подсчет до 22 за время 3.0994e-06
1
```

```
3
5
22
Завершен подсчет до 100000000 за время 5.2097
100000000
```

Хотя кажется, что программа работает так же, как `asyncio.as_completed`, на самом деле порядок итераций детерминирован и определяется тем, в каком порядке следуют числа в списке `numbers`. Это значит, что если бы первым числом было `100000000`, то пришлось бы ждать завершения соответствующего вызова, и только потом появилась бы возможность напечатать другие результаты, хотя они и были вычислены раньше. То есть эта техника не такая отзывчивая, как функция `asyncio.as_completed`.

6.3.2 Исполнители пула процессов в сочетании с циклом событий

Познакомившись с тем, как работают исполнители пула процессов, посмотрим, как включить их в цикл событий `asyncio`. Это позволит нам использовать такие рассмотренные в главе 4 функции API, как `gather` и `as_completed`, для управления несколькими процессами.

Исполнитель пула процессов для работы с `asyncio` создается так же, как было описано выше, т. е. в контекстном менеджере. Имея пул, мы можем использовать специальный метод цикла событий `asyncio` — `run_in_executor`. Этот метод принимает выполняемый объект и исполнитель (пула процессов или пула потоков), после чего исполняет этот объект внутри пула и возвращает допускающий ожидание объект, который можно использовать в предложении `await` или передать какой-нибудь функции API, например `gather`.

Давайте еще раз реализуем предыдущий пример, теперь с исполнителем пула процессов. Мы подадим исполнителю несколько задач подсчета и будем ждать их завершения с помощью `gather`. Метод `run_in_executor` принимает только вызываемый объект и не позволяет задать аргументы функции. Эту трудность мы обойдем, воспользовавшись частичным применением функции, чтобы сконструировать обращения к `count`, не нуждающиеся в аргументах.

Что такое частичное применение функции?

Механизм частичного применения функции реализован в модуле `functools`. Идея в том, чтобы взять функцию, принимающую некоторые аргументы, и преобразовать ее в другую функцию, принимающую меньше аргументов. Некоторые аргументы при этом «замораживаются». Например, наша функция `count` принимает один аргумент. Мы можем преобразовать ее в функцию без аргументов, воспользовавшись функцией `functools.partial` и указав в качестве ее аргумента то значение, с которым мы

хотим вызвать `count`. Если требуется вызвать `count(42)`, но при этом не передавать никаких аргументов, то можно определить `call_with_42 = functools.partial(count, 42)`, а затем просто вызвать `call_with_42()`.

Листинг 6.5 Исполнитель пула процессов в сочетании с *asyncio*

```
import asyncio
from asyncio.events import AbstractEventLoop
from concurrent.futures import ProcessPoolExecutor
from functools import partial
from typing import List

def count(count_to: int) -> int:
    counter = 0
    while counter < count_to:
        counter = counter + 1
    return counter

async def main():
    with ProcessPoolExecutor() as process_pool:
        loop: AbstractEventLoop = asyncio.get_running_loop()
        nums = [1, 3, 5, 22, 100000000]
        calls: List[partial[int]] = [partial(count, num) for num in nums]
        call_coros = []

        for call in calls:
            call_coros.append(loop.run_in_executor(process_pool, call))

        results = await asyncio.gather(*call_coros)

        for result in results:
            print(result)

if __name__ == "__main__":
    asyncio.run(main())
```

Сформировать все обращения к пулу процессов, поместив их в список

Создать частично применяемую функцию `count` с фиксированным аргументом

←

←

←

Ждать получения результатов

Сначала мы, как и раньше, создаем исполнитель пула процессов. Затем получаем цикл событий *asyncio*, поскольку `run_in_executor` – метод класса `AbstractEventLoop`. Затем с помощью частичного применения функции вызываем `count` с каждым числом из списка `nums` в качестве аргумента, поскольку прямой вызов с аргументом невозможен. Сформированные вызовы функции `count` можно передать исполнителю. Мы обходим эти вызовы в цикле, вызывая `loop.run_in_executor` для каждого и сохраняя полученные в ответ объекты, допускающие ожидание, в списке `call_coros`. Затем передаем этот список функции `asyncio.gather` и ждем завершения всех вызовов.

При желании можно было бы также использовать функцию `asyncio.as_completed` для получения результатов дочерних процессов по мере их готовности. Тем самым мы решили бы проблему рассмотренного выше метода пула процессов `map` в случае, если бы какая-то задача занимала много времени.

Теперь мы знаем все, что необходимо для использования пула процессов совместно с `asyncio`. Далее рассмотрим, как повысить производительность реального приложения с помощью библиотек `multiprocessing` и `asyncio`.

6.4 Решение задачи с помощью *MapReduce* и *asyncio*

Чтобы понять, задачи какого типа можно решить применением техники *MapReduce*, рассмотрим гипотетическую проблему. А затем применим полученные знания к решению похожей задачи с большим набором данных, находящимся в свободном доступе.

Вернемся к примеру интернет-магазина из главы 5 и предположим, что наш сайт получает большой объем текстовых данных через поле «Вопросы и замечания» в форме на портале технической поддержки. Поскольку наш сайт пользуется успехом, объем этого набора данных, содержащего отзывы клиентов, уже измеряется терабайтами и с каждым днем растет.

Чтобы лучше понять типичные проблемы, с которыми сталкиваются пользователи, нас попросили найти самые часто встречающиеся в этом наборе слова. Простое решение – запустить один процесс и в цикле обработать все комментарии, запоминая, сколько раз встретилось каждое слово. Это будет работать, но, поскольку набор данных велик, его последовательный просмотр может занять много времени. Нельзя ли как-нибудь решить задачу побыстрее?

Именно для такого рода задач предназначена технология *MapReduce*. В модели программирования *MapReduce* большой набор данных сначала разбивается на меньшие части. Затем мы можем решить задачу для поднабора данных, а не для всего набора – это называется *отображением* (*mapping*), поскольку мы «отображаем» данные на частичный результат.

После того как задачи для всех поднаборов решены, мы можем объединить результаты в окончательный ответ. Этот шаг называется *редукцией* (*reducing*), потому что «редуцируем» (сводим) несколько ответов в один. Подсчет частоты вхождения слов в большой набор текстовых данных – каноническая задача *MapReduce*. Если набор данных достаточно велик, то его разбиение на меньшие части может дать выигрыш в производительности, поскольку все операции отображения можно выполнять параллельно, как показано на рис. 6.1.

Есть системы, например *Hadoop* и *Spark*, которые выполняют операции *MapReduce* в кластере компьютеров для по-настоящему больших наборов данных. Но наборы поменьше можно обработать и на одном компьютере с помощью библиотеки `multiprocessing`. В этом разделе мы покажем, как реализовать технологический процесс *Ma-*

рReduce с помощью multiprocessing и найти частоты различных слов в мировой литературе, начиная с 1500 года.

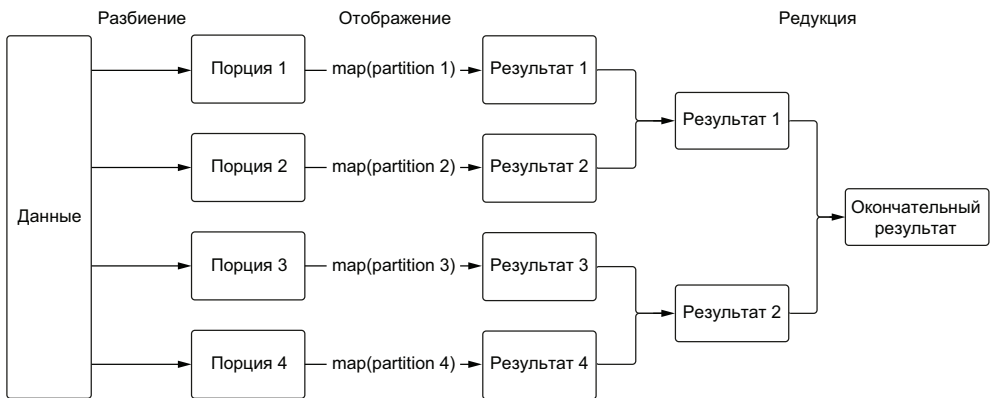


Рис. 6.1 Большой набор данных разбивается на разделы, после чего функция отображения порождает промежуточные результаты, которые затем объединяются в окончательный

6.4.1 Простой пример MapReduce

Чтобы лучше понять, как работает MapReduce, разберем конкретный пример. Предположим, что имеется файл, каждая строка которого содержит текстовые данные, скажем вот такие четыре строки:

```
I know what I know.  
I know that I know.  
I don't know that much.  
They don't know much.
```

Мы хотим подсчитать, сколько раз в этом наборе встречается каждое слово. Пример настолько мал, что хватило бы и простого цикла `for`, но давайте все же применим модель MapReduce.

Сначала нужно разбить данные на меньшие порции. Для простоты примем за порцию одну строку. Затем нужно определить операцию отображения. Поскольку нам нужны частоты слов, будем разбивать строку текста по пробелам. Это даст нам массив слов в строке. После этого его можно обойти в цикле, запоминая различные встретившиеся слова в словаре.

Наконец, нужно определить операцию редукции. Она принимает один или несколько результатов операций отображения и объединяет их в окончательный ответ. В этом примере нам нужно взять два построенных операцией отображения словаря и объединить их в один. Если слово существует в обоих словарях, счетчики его вхождений складываются; если нет, мы копируем счетчик вхождений в результирующий словарь. Определив операции, мы можем применить операцию `map` к каждой строке текста, а затем операцию `reduce` к парам

результатов отображения. Взглянем на код, который делает это для нашего текстового файла с четырьмя строками.

Листинг 6.6 Однопоточная модель MapReduce

```
import functools
from typing import Dict

def map_frequency(text: str) -> Dict[str, int]:
    words = text.split(' ')
    frequencies = {}
    for word in words:
        if word in frequencies:
            frequencies[word] = frequencies[word] + 1
        else:
            frequencies[word] = 1
    return frequencies

def merge_dictionaries(first: Dict[str, int],
                      second: Dict[str, int]) -> Dict[str, int]:
    merged = first
    for key in second:
        if key in merged:
            merged[key] = merged[key] + second[key]
        else:
            merged[key] = second[key]
    return merged

lines = ["I know what I know",
         "I know that I know",
         "I don't know much",
         "They don't know much"]

mapped_results = [map_frequency(line) for line in lines]

for result in mapped_results:
    print(result)

print(functools.reduce(merge_dictionaries, mapped_results))
```

Если слово уже есть в словаре частот, прибавить единицу к счетчику

Если слова еще нет в словаре частот, положить счетчик равным единице

Если слово встречается в обоих словарях, сложить счетчики

Если слово не встречается в обоих словарях, скопировать счетчик

Для каждой строки текста выполнить операцию map

Редуцировать все промежуточные счетчики в окончательный результат

К каждой строке текста мы применяем операцию map, что дает счетчики частот для каждой строки. Затем частичные результаты отображения можно объединить. Мы применяем нашу функцию слияния `merge_dictionaries` в сочетании с библиотечной функцией `functools.reduce`. В результате получается следующая картина:

```
Mapped results:
{'I': 2, 'know': 2, 'what': 1}
{'I': 2, 'know': 2, 'that': 1}
{'I': 1, "don't": 1, 'know': 1, 'much': 1}
{'They': 1, "don't": 1, 'know': 1, 'much': 1}
```

Final Result:

```
{'I': 5, 'know': 6, 'what': 1, 'that': 1, 'don't': 2, 'much': 2, 'They': 1}
```

Разобравшись с основами технологии MapReduce на модельной задаче, посмотрим, как применить ее к реальному набору данных, для которого библиотека multiprocessing может дать выигрыш в производительности.

6.4.2 Набор данных Google Books Ngram

Нам нужен достаточно большой набор данных, чтобы продемонстрировать все преимущества сочетания MapReduce с библиотекой multiprocessing. Если набор данных слишком мал, то мы, скорее всего, увидим не преимущества, а падение производительности из-за накладных расходов на управление процессами.

Набор данных Google Books Ngram достаточен для наших целей. Чтобы понять, что это такое, сначала определим понятие *n*-граммы.

n-граммой называется последовательность *N* слов заданного текста. Во фразе «the fast dog» есть шесть *n*-грамм. Три 1-граммы, или *униграммы*, содержащие по одному слову (*the*, *fast* и *dog*), две 2-граммы, или *диграммы* (*the fast* и *fast dog*), и одна 3-грамма, или *триграмма* (*the fast dog*).

Набор данных Google Books Ngram состоит из *n*-грамм, взятых из 8 000 000 книг, начиная с 1500 года. Это более шести процентов от всех изданных в мире книг. Подсчитано, сколько раз каждая уникальная *n*-грамма встречается в текстах, и результаты сгруппированы по годам. В этом наборе данных присутствуют *n*-граммы для *n* от 1 до 5, представленные в формате с табуляторами. Каждая строка набора содержит *n*-грамму, год ее появления, сколько раз она встречалась и в скольких книгах. Рассмотрим первые несколько строк набора униграмм для слова *aardvark*:

```
Aardvark 1822 2 1
Aardvark 1824 3 1
Aardvark 1827 10 7
```

Это означает, что в 1822 году слово *aardvark* (трубкозуб) дважды встретилось в одной книге. А в 1827 году оно встретилось десять раз в семи книгах. В наборе данных есть гораздо больше строк для слова *aardvark* (например, в 2007 году оно встретилось 1200 раз), что доказывает все более частое упоминание трубкозубов в литературе с течением времени.

В этом примере мы подсчитаем количество вхождений одиночных слов (униграмм), начинающихся с буквы *a*. Этот набор данных занимает приблизительно 1,8 Гб. Мы агрегируем данные по количеству упоминаний каждого слова в литературе, начиная с 1500 года. И воспользуемся этим для ответа на вопрос: «Сколько раз слово *aardvark* встречалось в литературе с 1500-го года?» Нужный нам файл можно

скачать по адресу <https://storage.googleapis.com/books/ngrams/books/googlebooks-eng-all-1gram-20120701-a.gz> или <https://mattfowler.io/data/googlebooks-eng-all-1gram-20120701-a.gz>. Любую другую часть этого набора данных можно скачать по адресу <http://storage.googleapis.com/books/ngrams/books/datasetsv2.html>.

6.4.3 Применение *asynсio* для отображения и редукции

Чтобы было с чем сравнивать, напишем сначала синхронную версию подсчета частот слов. А затем используем полученный словарь частот для ответа на вопрос: «Сколько раз слово *aardvark* встречалось в литературе, начиная с 1500 года?» Прежде всего нужно загрузить весь набор данных в память. А затем построить словарь, в котором будет храниться отображение слов на количество вхождений. Для каждой строки файла проверяется, имеется ли уже данное слово в словаре. Если да, то его счетчик в словаре увеличивается на единицу, а иначе слово добавляется в словарь со счетчиком, равным 1.

Листинг 6.7 Подсчет частот слов, начинающихся буквой *a*

```
import time

freqs = {}

with open('googlebooks-eng-all-1gram-20120701-a', encoding='utf-8') as f:
    lines = f.readlines()

    start = time.time()

    for line in lines:
        data = line.split('\t')
        word = data[0]
        count = int(data[2])
        if word in freqs:
            freqs[word] = freqs[word] + count
        else:
            freqs[word] = count

    end = time.time()
    print(f'{end-start:.4f}')
```

Во время, затраченное на счетную операцию, мы будем включать только время подсчета частот, не учитывая время загрузки файла. Чтобы библиотека *multiprocessing* дала заметный выигрыш, понадобится машина с достаточным для эффективного распараллеливания числом ядер – большим, чем у стандартных ноутбуков. Мы воспользовались для этой цели большим экземпляром Elastic Compute Cloud (EC2) в службе Amazon Web Servers (AWS).

AWS – это служба облачных вычислений, эксплуатируемая компанией Amazon. AWS представляет собой набор облачных служб, позволяющих решать различные задачи (от хранения файлов до круп-

номасштабного машинного обучения) и не заботиться при этом об администрировании собственных физических серверов. Одной из таких служб является EC2. Она дает возможность запускать нужное вам приложение на арендованной у AWS виртуальной машине с заданным числом процессорных ядер и объемом памяти. Подробнее об AWS и EC2 можно прочитать на сайте <https://aws.amazon.com/ec2>.

Мы будем тестировать на экземпляре c5ad.8xlarge. На момент написания книги это была машина с 32 ядрами, 65 Гб памяти и SSD-диском. На этом экземпляре скрипт в листинге 6.7 работал приблизительно 76 с. Посмотрим, удастся ли улучшить этот результат, воспользовавшись библиотеками multiprocessing и asyncio. При запуске на машине с меньшим числом ядер или других ресурсов результат может быть иным.

Первым делом разобьем наш набор данных на меньшие порции. Определим генератор разбиения, который принимает большой список данных и выделяет порции произвольного размера.

```
def partition(data: List,
              chunk_size: int) -> List:
    for i in range(0, len(data), chunk_size):
        yield data[i:i + chunk_size]
```

Этот генератор можно использовать для создания порций размера chunk_size. Нам он понадобится для создания порций, передаваемых функциям отображения, которые затем будут запущены параллельно. Далее определим функцию отображения. Она мало чем отличается от функции из предыдущего примера, разве что адаптирована к нашему набору данных.

```
def map_frequencies(chunk: List[str]) -> Dict[str, int]:
    counter = {}
    for line in chunk:
        word, _, count, _ = line.split('\t')
        if counter.get(word):
            counter[word] = counter[word] + int(count)
        else:
            counter[word] = int(count)
    return counter
```

Пока что оставим ту же операцию редукции, что в предыдущем примере. Теперь у нас есть все необходимое для распараллеливания операций отображения. Мы создадим пул процессов, разобьем данные на порции и для каждой порции выполним функцию map_frequencies, забрав ресурс («исполнитель») из пула. Остается только один вопрос: как выбрать размер порции?

Простого ответа на этот вопрос не существует. Есть эвристическое правило – сбалансированный подход¹: порция не должна быть ни

¹ Goldilocks approach – подход Златовласки (вариант нашей сказки про Машеньку и трех медведей). – Прим. перев.

слишком большой, ни слишком маленькой. Маленькой она не должна быть потому, что созданные порции сериализуются (в формате pickle) и раздаются исполнителям, после чего исполнители десериализуют их. Процедура сериализации и десериализации может занимать много времени, сводя на нет весь выигрыш, если производится слишком часто. Например, размер порции, равный 2, – заведомо неудачное решение, потому что потребовалось бы почти 1 000 000 операций сериализации и десериализации.

Но и слишком большая порция – тоже плохо, поскольку это не даст нам в полной мере задействовать возможности компьютера. Например, если имеется 10 ядер, но всего две порции, то мы ничем не загружаем восемь ядер, которые могли бы работать параллельно.

Для этого примера мы выбрали размер порции 60 000, поскольку он, похоже, дает разумную производительность на машине AWS, на которой производилось тестирование. Если вы будете применять такой подход в своих задачах обработки данных, то попробуйте разные размеры порций, пока не найдете подходящий для имеющейся машины и набора данных, или разработайте эвристический алгоритм определения правильного размера. Теперь объединим все эти части с пулом процессов и методом `run_in_executor` цикла событий для распараллеливания операций отображения.

Листинг 6.8 Распараллеливание с помощью MapReduce и пула процессов

```
import asyncio
import concurrent.futures
import functools
import time
from typing import Dict, List

def partition(data: List,
              chunk_size: int) -> List:
    for i in range(0, len(data), chunk_size):
        yield data[i:i + chunk_size]

def map_frequencies(chunk: List[str]) -> Dict[str, int]:
    counter = {}
    for line in chunk:
        word, _, count, _ = line.split('\t')
        if counter.get(word):
            counter[word] = counter[word] + int(count)
        else:
            counter[word] = int(count)
    return counter

def merge_dictionaries(first: Dict[str, int],
                       second: Dict[str, int]) -> Dict[str, int]:
    merged = first
    for key in second:
        if key in merged:
```

```

        merged[key] = merged[key] + second[key]
    else:
        merged[key] = second[key]
    return merged

async def main(partition_size: int):
    with open('googlebooks-eng-all-1gram-20120701-a', encoding='utf-8') as f:
        contents = f.readlines()
        loop = asyncio.get_running_loop()
        tasks = []
        start = time.time()
        with concurrent.futures.ProcessPoolExecutor() as pool:
            for chunk in partition(contents, partition_size):
                tasks.append(loop.run_in_executor(pool,
                    functools.partial(map_frequencies, chunk)))
        intermediate_results = await asyncio.gather(*tasks)
        final_result = functools.reduce(merge_dictionaries,
                                         intermediate_results)

        print(f"Aardvark встречается {final_result['Aardvark']} раз.")

        end = time.time()
        print(f'Время MapReduce: {(end - start):.4f} секунд')

if __name__ == "__main__":
    asyncio.run(main(partition_size=60000))

```

Для каждой порции выполнить операцию отображения в отдельном процессе

Ждать завершения всех операций отображения

Редуктировать промежуточные результаты в окончательный

В сопрограамме `main` мы создаем пул процессов и разбиваем данные на порции. Для каждой порции функция `map_frequencies` исполняет-ся в отдельном процессе. Затем с помощью `asyncio.gather` ждем завершения построения промежуточных словарей. Когда все операции отображения завершатся, мы выполняем операцию редукции для получения окончательного результата.

На описанном выше экземпляре AWS эта программа завершается примерно за 18 с, т. е. мы получили значительный выигрыш по сравнению с последовательной версией. Неплохо, учитывая, что дополнительного кода мы написали не так уж много! Можете поэкспериментировать с машинами, имеющими больше ядер, и посмотреть, нельзя ли еще улучшить производительность этого алгоритма.

Вы, наверное, обратили внимание, что в родительском процессе еще остался счетный код, допускающий распараллеливание. Операция редукции принимает тысячи словарей и объединяет их. Мы можем применить ту же логику разбиения, что для исходного набора данных, чтобы разбить множество словарей на порции и объединять их в нескольких процессах. Перепишем для этого функцию `reduce`. В ней мы разобьем список на части и для каждой части выполним редуктор в отдельном процессе. По завершении этой операции снова выполним разбиение и редукцию и будем поступать так, пока не останется один словарь. (В листинге ниже мы для краткости опустили функции разбиения, отображения и объединения.)

Листинг 6.9 Распараллеливание операции reduce

```

import asyncio
import concurrent.futures
import functools
import time
from typing import Dict, List
from chapter_06.listing_6_8 import partition, merge_dictionaries,
    map_frequencies

async def reduce(loop, pool, counters, chunk_size) -> Dict[str, int]:
    chunks: List[List[Dict]] = list(partition(counters, chunk_size))
    reducers = []
    while len(chunks[0]) > 1:
        for chunk in chunks:
            reducer = functools.partial(functools.reduce,
                                         merge_dictionaries, chunk)
            reducers.append(loop.run_in_executor(pool, reducer))
            Редуцировать каждую порцию в один словарь
        reducer_chunks = await asyncio.gather(*reducers)
        chunks = list(partition(reducer_chunks, chunk_size))
        reducers.clear()
    return chunks[0][0]
    Снова разбить результаты и выполнить еще одну итерацию цикла

Разбить словари на допускающие распараллеливание порции
Ждать завершения всех операций редукции

async def main(partition_size: int):
    with open('googlebooks-eng-all-1gram-20120701-a', encoding='utf-8') as f:
        contents = f.readlines()
        loop = asyncio.get_running_loop()
        tasks = []
        with concurrent.futures.ProcessPoolExecutor() as pool:
            start = time.time()

            for chunk in partition(contents, partition_size):
                tasks.append(loop.run_in_executor(pool,
                                                    functools.partial(map_frequencies, chunk)))

            intermediate_results = await asyncio.gather(*tasks)
            final_result = await reduce(loop, pool, intermediate_results, 500)

            print(f"Aardvark has appeared {final_result['Aardvark']} times.")

            end = time.time()
            print(f'Время MapReduce: {(end - start):.4f} секунд')

if __name__ == "__main__":
    asyncio.run(main(partition_size=60000))

```

Выполнение такой распараллеленной функции `reduce` дает лишь небольшой выигрыш в производительности или вообще ничего в зависимости от машины. Накладные расходы на сериализацию промежуточных словарей и раздачу их дочерним процессам съедают большую часть выигрыша от распараллеливания. Такая оптимизация принесла мало пользы, но если бы операция `reduce` потребляла больше процессорного времени или сам набор данных был бы больше, то этот подход мог бы оказаться выгодным.

Многопроцессность, очевидно, имеет преимущество перед синхронным подходом, но пока что непонятно, сколько операций отображения завершено в каждый момент времени. В синхронной версии нужно было бы всего лишь добавить счетчик, который увеличивается для каждой обработанной строки, – и так мы узнали бы, как далеко продвинулись. Но процессы по умолчанию не разделяют никакой общей памяти, и как в этих условиях создать счетчик для наблюдения за ходом выполнения?

6.5 Разделяемые данные и блокировки

В главе 1 мы говорили о том, что каждый процесс имеет собственную память, изолированную от памяти других процессов. Спрашивается, что делать, если нужно организовать общую память для хранения разделяемой несколькими процессами информации о состоянии?

Библиотека multiprocessing поддерживает так называемые *объекты разделяемой памяти*. Это блок памяти, выделенный так, что к нему могут обращаться разные процессы. Каждый процесс может читать и записывать в этот блок, как показано на рис. 6.2.

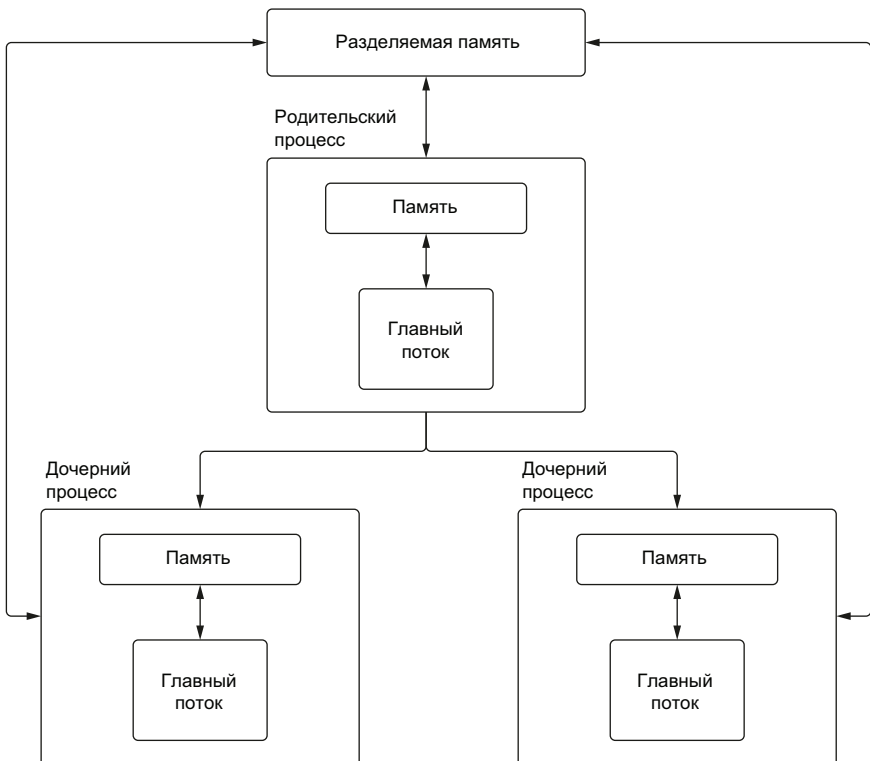


Рис. 6.2 Родительский и два дочерних процесса, разделяющих общую память

Если работа с разделяемым состоянием организована некорректно, то возможны ошибки, которые трудно воспроизвести. Вообще говоря, по возможности лучше избегать разделяемого состояния. Но иногда без него не обойтись. Примером может служить разделяемый счетчик.

Чтобы больше узнать о разделяемых данных, мы включим в рассмотренный выше пример счетчик завершенных операций отображения. И будем периодически выводить его текущее значение, чтобы показать пользователю, как далеко мы продвинулись.

6.5.1 Разделение данных и состояние гонки

Библиотека `multiprocessing` поддерживает два вида разделяемых данных: значения и массив. Под *значением* понимается одиночное значение, например целое число или число с плавающей точкой. А *массив* – это массив одиночных значений. В разделяемой памяти можно хранить только данные типов, определенных в модуле Python `array`, описанном в документации по адресу <https://docs.python.org/3/library/array.html#module-array>.

Чтобы создать значение или массив, нам нужен код типа из модуля `array`; это просто символ. Давайте создадим два разделяемых элемента данных: целое число и массив целых чисел. Затем создадим два процесса, которые будут параллельно инкрементировать эти элементы данных.

Листинг 6.10 Разделяемые значения и массивы

```
from multiprocessing import Process, Value, Array

def increment_value(shared_int: Value):
    shared_int.value = shared_int.value + 1

def increment_array(shared_array: Array):
    for index, integer in enumerate(shared_array):
        shared_array[index] = integer + 1

if __name__ == '__main__':
    integer = Value('i', 0)
    integer_array = Array('i', [0, 0])

    procs = [Process(target=increment_value, args=(integer,)),
             Process(target=increment_array, args=(integer_array,))]

    [p.start() for p in procs]
    [p.join() for p in procs]

    print(integer.value)
    print(integer_array[:])
```

Здесь мы создаем два процесса – один инкрементирует разделяемое целое число, а другой инкрементирует каждый элемент разделя-

емого массива. По завершении обоих процессов данные печатаются.

Поскольку к этим двум элементам никогда не обращаются разные процессы, все работает хорошо. Но будет ли этот код работать, когда несколько процессов модифицируют данные одновременно? Проверим, создав два процесса, которые параллельно инкрементируют разделяемое целое значение. Так как процессов два и каждый увеличивает разделяемый счетчик на единицу, то мы ожидаем, что после завершения процессов разделяемое значение будет равно 2.

Листинг 6.11 Параллельное инкрементирование разделяемого счетчика

```
from multiprocessing import Process, Value

def increment_value(shared_int: Value):
    shared_int.value = shared_int.value + 1

if __name__ == '__main__':
    for _ in range(100):
        integer = Value('i', 0)
        procs = [Process(target=increment_value, args=(integer,)),
                  Process(target=increment_value, args=(integer,))]

        [p.start() for p in procs]
        [p.join() for p in procs]
        print(integer.value)
        assert(integer.value == 2)
```

Проблема недетерминированная, поэтому результаты могут получаться разные, но в какой-то момент вы обязательно увидите, что результат не равен 2.

```
2
2
2
Traceback (most recent call last):
  File "listing_6_11.py", line 17, in <module>
    assert(integer.value == 2)
AssertionError
1
```

Иногда счетчик оказывается равен 1! Почему? То, с чем мы столкнулись, называется *состоянием гонки*. Оно возникает, если исход последовательности операций зависит от того, какая операция заканчивается первой. Можно рассматривать операции как гонку на опережение; если порядок прихода к финишу правильный, то все работает нормально, в противном случае возможно неожиданное поведение.

Так где же возникает гонка в нашем примере? Проблема в том, что для увеличения значения на единицу его нужно сначала прочитать, прибавить 1, а потом записать в память. Разделяемое значение, кото-

рое видит процесс, зависит от того, в какой момент он его читает. Если процессы работают, как показано на рис. 6.3, то все будет хорошо.

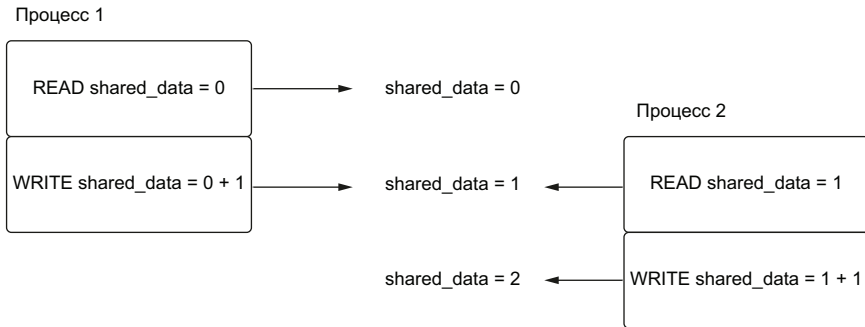


Рис. 6.3 Состояния гонки удалось избежать

Здесь Процесс 1 увеличивает значение, перед тем как Процесс 2 читает его, а потому выигрывает гонку. Поскольку Процесс 2 финиширует вторым, он видит правильное значение 1, прибавляет к нему единицу, и конечный результат оказывается верным.

Но что, если в нашей виртуальной гонке случилась ничья? Взгляните на рис. 6.4.

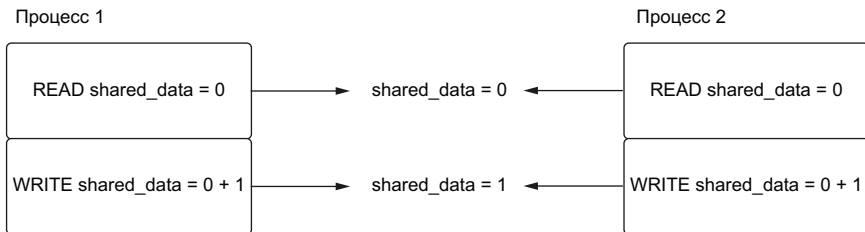


Рис. 6.4 Состояния гонки

В этом случае оба процесса 1 и 2 читают начальное значение, равное 0. Затем оба увеличивают его, получают 1 и одновременно записывают в память. Результат – неправильное значение! Может возникнуть вопрос: «Но ведь в нашем коде всего одна строка. Откуда же две операции?» Дело в том, что за кулисами инкрементирования значения состоит из двух машинных операций, что и является причиной проблемы, – операция оказывается *неатомарной* или *потоконебезопасной*. Предвидеть это не так-то просто. Подробное объяснение того, какие операции атомарны, а какие нет, можно найти по адресу <http://mng.bz/5Kj4>.

С такими ошибками трудно бороться, потому что трудно воспроизвести. В отличие от обычных ошибок, они зависят от порядка выполнения действий операционной системой, а мы его не контролируем. И как все-таки исправить эту досадную ошибку?

6.5.2 Синхронизация с помощью блокировок

Избежать гонки можно, *синхронизировав* доступ к тем разделяемым данным, которые мы собираемся модифицировать. Что под этим понимается? Это значит, что мы управляем доступом к разделяемым данным таким образом, чтобы все операции финишировали в осмысленном порядке. Если возможна ничья между какими-то двумя операциями, то мы явно блокируем вторую до завершения первой, гарантируя тем самым, что операции финишируют так, как нужно. Можно представлять себе, что на финишной прямой стоит судья, и если он видит, что надвигается ничья, то говорит бегунам «Стоять, проходить по одному!» и выбирает того, кто должен будет подождать, пока другой финиширует.

Один из механизмов для синхронизации доступа к разделяемым данным называется *блокировкой*, или *мьютексом* (от *mutual exclusion* – взаимное исключение). Он позволяет одному процессу заблокировать участок кода, т. е. запретить всем остальным его выполнение. Заблокированный участок обычно называют *критической секцией*. Если один процесс выполняет код в критической секции, а второй пытается выполнить тот же код, то второму придется подождать (арбитр не пускает его), пока первый закончит работу и выйдет из критической секции.

Блокировки поддерживают две основные операции: *захват* и *освобождение*. Гарантируется, что процесс, захвативший блокировку, – единственный, кто может выполнять код в критической секции. Закончив выполнение кода, требующего синхронизации доступа, мы освобождаем блокировку. Это дает возможность другим процессам захватить блокировку и выполнить код в критической секции. Если процесс попытается выполнить код в секции, заблокированной другим процессом, то будет приостановлен, пока этот другой процесс не освободит блокировку.

Вернемся к примеру, где возникло состояние гонки, и, глядя на рис. 6.5, попробуем понять, что происходит, когда два процесса пытаются захватить блокировку приблизительно одновременно. И разберемся, почему блокировка не дает счетчику принять неправильное значение.

На этом рисунке видно, что Процесс 1 первым успешно захватил блокировку, после чего прочитал и увеличил разделяемый счетчик. Второй процесс тоже пытался захватить блокировку, но потерпел неудачу и был заблокирован в ожидании освобождения блокировки первым процессом. После того как первый процесс освободил блокировку, второй успешно захватил ее и увеличил счетчик. Гонка предотвращена, потому что блокировка не дает сразу нескольким процессам одновременно читать и записывать разделяемые данные.

Как реализовать синхронизацию доступа к разделяемым данным? Разработчики библиотеки `multiprocessing` подумали об этом и любезно включили метод, который возвращает блокировку на значение или массив. Чтобы захватить блокировку, нужно вызвать `get_lock()`.

`acquire()`, а для ее освобождения – метод `get_lock().release()`. В листинге 6.12 эти методы использованы для предотвращения ошибки.

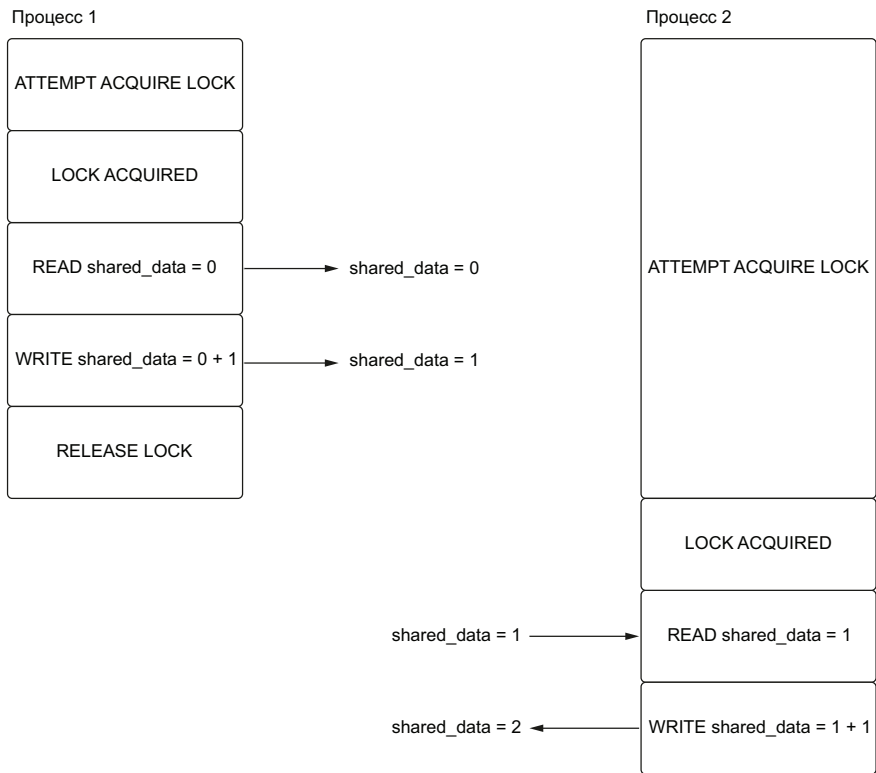


Рис. 6.5 Попытка Процесса 2 прочитать разделяемые данные блокируется, пока Процесс 1 не освободит блокировку

Листинг 6.12 Захват и освобождение блокировки

```
from multiprocessing import Process, Value

def increment_value(shared_int: Value):
    shared_int.get_lock().acquire()
    shared_int.value = shared_int.value + 1
    shared_int.get_lock().release()

if __name__ == '__main__':
    for _ in range(100):
        integer = Value('i', 0)
        procs = [Process(target=increment_value, args=(integer,)),
                  Process(target=increment_value, args=(integer,))]
    [p.start() for p in procs]
    [p.join() for p in procs]
    print(integer.value)
    assert (integer.value == 2)
```

При выполнении этой программы мы всегда будем получать значение 2. Гонка устранена! Заметим, что блокировки являются контекстными менеджерами, и, чтобы сделать код чище, мы могли бы использовать в функции `increment_value` блок `with`. Тогда захват и освобождение блокировки будут производиться автоматически:

```
def increment_value(shared_int: Value):  
    with shared_int.get_lock():  
        shared_int.value = shared_int.value + 1
```

Обратите внимание, что мы принудительно преобразовали конкурентный код в последовательный, сведя на нет преимущества распараллеливания. Это важное наблюдение, поскольку проливает свет на недостаток синхронизации и разделяемые данные вообще. Чтобы избежать гонки, код в критических секциях обязан выполняться последовательно. Это может отрицательно сказаться на производительности многопроцессного кода. Поэтому нужно внимательно следить за тем, чтобы защищать блокировкой только то, что абсолютно необходимо, и не мешать остальному коду выполняться конкурентно. Столкнувшись с состоянием гонки, не нужно идти по легкому пути и защищать блокировкой все вообще. Проблему-то вы решите, но, скорее всего, производительность сильно пострадает.

6.5.3 Разделение данных в пулах процессов

Только что мы видели, как разделить данные между двумя процессами, а как применить эти знания к пулу процессов? Процессы в пуле мы не создаем вручную, что вызывает проблемы при разделении данных. Почему?

Выполнение задачи, переданной пулу процессов, может начаться не сразу, потому что все процессы могут быть заняты. Как пул справляется с этим? За кулисами исполнители пула процессов поддерживают очередь задач. Когда мы передаем задачу пулу, ее аргументы сериализуются и помещаются в очередь. Когда процесс-исполнитель будет готов, он запросит задачу из очереди. Извлекая задачу из очереди, процесс-исполнитель десериализует аргументы и приступает к выполнению.

Разделяемые данные по определению разделяются между всеми процессами-исполнителями. Следовательно, сериализовывать и десериализовывать их для передачи от одного процесса другому не имеет смысла. На самом деле объекты `Value` и `Array` вообще не допускают сериализации, поэтому попытка передать разделяемые данные в аргументах, как мы делали раньше, приведет к ошибке с сообщением «`can't pickle Value objects`».

Чтобы решить проблему, мы должны поместить разделяемый счетчик в глобальную переменную и каким-то образом дать знать об этом процессам-исполнителям. Для этого предназначены *инициализаторы пула процессов* – специальные функции, которые вызываются в момент запуска каждого процесса в пуле. С их помощью мы можем

создать ссылку на разделяемую память, выделенную родительским процессом. Инициализатор можно передать пулу процессов в момент его создания. Продемонстрируем это на простом примере инкрементирования счетчика.

Листинг 6.13 Инициализация пула процессов

```
from concurrent.futures import ProcessPoolExecutor
import asyncio
from multiprocessing import Value

shared_counter: Value

def init(counter: Value):
    global shared_counter
    shared_counter = counter

def increment():
    with shared_counter.get_lock():
        shared_counter.value += 1

async def main():
    counter = Value('d', 0)
    with ProcessPoolExecutor(initializer=init,
                             initargs=(counter,)) as pool:
        await asyncio.get_running_loop().run_in_executor(pool, increment)
    print(counter.value)

if __name__ == "__main__":
    asyncio.run(main())
```

Говорим пулу, что он должен выполнять функцию `init` с аргументом `counter` для каждого процесса

Сначала мы определяем глобальную переменную `shared_counter`, которая будет содержать ссылку на созданный нами разделяемый объект `Value`. В функции `init` мы принимаем `Value` и инициализируем `shared_counter` этим значением. Затем в сопрограмме `main` создаем счетчик и инициализируем его нулем. Далее функция `init` и счетчик передаются в виде параметров `initializer` и `initargs` конструктору пула процессов. Функция `init` будет вызываться для каждого процесса, созданного в пуле, и правильно инициализировать переменную `shared_counter` значением, созданным в сопрограмме `main`. Вы спросите: «А к чему все эти хлопоты? Не проще ли инициализировать глобальную переменную `shared_counter: Value = Value('d', 0)`, а не оставлять ее пустой?» Нет, так нельзя, потому что при создании каждого процесса будет по новой выполняться создающий его скрипт. А значит, каждый процесс будет начинаться с выполнения предложения `shared_counter: Value = Value('d', 0)`, и если у нас есть 100 процессов, то мы получим 100 значений `shared_counter`, равных 0, что приведет к странному поведению.

Итак, теперь мы знаем, как правильно инициализировать разделяемые данные в пуле процессов. Посмотрим, как это применить к нашему приложению `MapReduce`. Создадим разделяемый счетчик, который

будет увеличиваться на 1 при завершении каждой операции отображения. Также создадим задачу `progress_reporter`, которая будет работать в фоновом режиме и каждую секунду выводить на консоль информацию о том, как далеко мы продвинулись. Чтобы не повторяться, импортируем код, касающийся разбиения на порции и редукции.

Листинг 6.14 Наблюдение за ходом отображения

```
from concurrent.futures import ProcessPoolExecutor
import functools
import asyncio
from multiprocessing import Value
from typing import List, Dict
from chapter_06.listing_6_8 import partition, merge_dictionaries

map_progress: Value

def init(progress: Value):
    global map_progress
    map_progress = progress

def map_frequencies(chunk: List[str]) -> Dict[str, int]:
    counter = {}
    for line in chunk:
        word, _, count, _ = line.split('\t')
        if counter.get(word):
            counter[word] = counter[word] + int(count)
        else:
            counter[word] = int(count)

    with map_progress.get_lock():
        map_progress.value += 1

    return counter

async def progress_reporter(total_partitions: int):
    while map_progress.value < total_partitions:
        print(f'Завершено операций отображения:
              {map_progress.value}/{total_partitions}')
        await asyncio.sleep(1)

async def main(partiton_size: int):
    global map_progress

    with open('googlebooks-eng-all-1gram-20120701-a', encoding='utf-8') as f:
        contents = f.readlines()
        loop = asyncio.get_running_loop()
        tasks = []
        map_progress = Value('i', 0)

        with ProcessPoolExecutor(initializer=init,
                                initargs=(map_progress,)) as pool:
            total_partitions = len(contents) // partiton_size
            reporter = asyncio.create_task(progress_reporter(total_partitions))
```

```

for chunk in partition(contents, partiton_size):
    tasks.append(loop.run_in_executor(pool,
                                     functools.partial(map_frequencies, chunk)))

counters = await asyncio.gather(*tasks)

await reporter

final_result = functools.reduce(merge_dictionaries, counters)

print(f"Aardvark встречается {final_result['Aardvark']} раз.")

if __name__ == "__main__":
    asyncio.run(main(partiton_size=60000))

```

Главное отличие этой реализации от первоначальной, если не считать инициализации разделяемого счетчика, находится внутри функции `map_frequencies`. Закончив подсчет всех слов в данной порции, мы захватываем блокировку на разделяемый счетчик и увеличиваем его на единицу. Мы также добавили сопрограмму `progress_reporter`, которая работает в фоновом режиме и каждую секунду сообщает, сколько заданий завершено. Во время работы программы мы будем видеть следующую картину:

```

Завершено операций отображения: 17/1443
Завершено операций отображения: 144/1443
Завершено операций отображения: 281/1443
Завершено операций отображения: 419/1443
Завершено операций отображения: 560/1443
Завершено операций отображения: 701/1443
Завершено операций отображения: 839/1443
Завершено операций отображения: 976/1443
Завершено операций отображения: 1099/1443
Завершено операций отображения: 1230/1443
Завершено операций отображения: 1353/1443
Aardvark встречается 15209 раз.

```

Теперь мы знаем, как совместно использовать библиотеки `multi-processing` и `asyncio` для повышения производительности при решении счетных задач. А что, если рабочая нагрузка представляет собой смесь счетных операций и операций ввода-вывода? Мы можем использовать многопроцессность, но существует ли способ объединить идеи многопроцессности и однопоточной конкурентности, чтобы добиться еще большей производительности?

6.6 Несколько процессов и несколько циклов событий

Многопроцессность полезна в основном для счетных задач, но может дать некоторые преимущества и для рабочих нагрузок с преоб-

ладанием операций ввода-вывода. Взять, к примеру, конкурентное выполнение нескольких SQL-запросов в листинге 5.8 из предыдущей главы. Может ли применение многопроцессности повысить его производительность? На рис. 6.6 показан график потребления процессорного времени при работе на одном ядре.

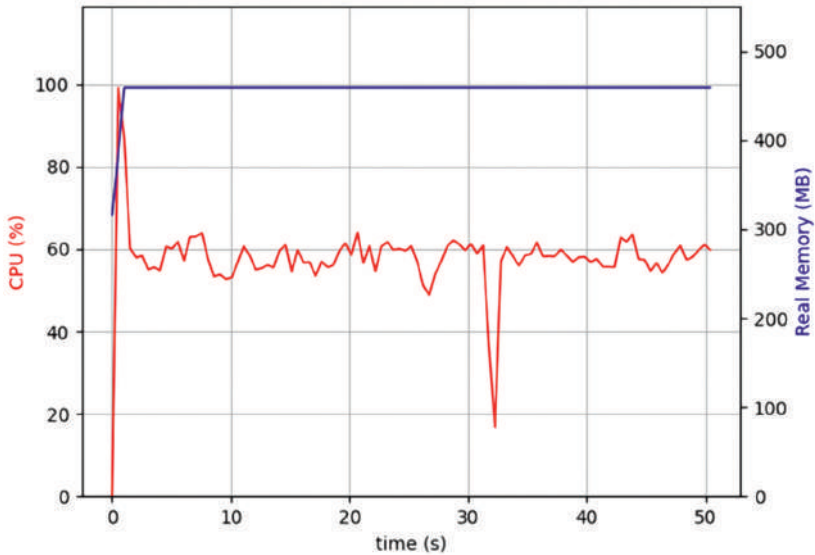


Рис. 6.6 График потребления процессорного времени программой в листинге 5.8

Хотя этот код занят в основном отправкой запросов базе данных, т. е. вводом-выводом, все равно он потребляет довольно много процессорного времени. Почему? В данном случае нужно обрабатывать результаты, полученные от Postgres, а для этого нужен процессор. Поскольку программа однопоточная, во время этой работы цикл событий не может обрабатывать результаты других запросов, а стало быть, может возникнуть проблема низкой пропускной способности. Если конкурентно отправлено 10 000 SQL-запросов, но обрабатывать можно только по одному результату в каждый момент времени, то будет копиться очередь необработанных результатов.

Нельзя ли как-то улучшить пропускную способность, воспользовавшись многопроцессностью? В этом случае каждый процесс имеет собственный поток и собственный интерпретатор Python. Это открывает возможность создать по одному циклу событий для каждого процесса в пуле, а значит, распределить запросы между несколькими процессами. На рис. 6.7 показано, что и нагрузка на процессор при этом распределяется между несколькими процессами.

Это не увеличит пропускную способность ввода-вывода, зато увеличит количество результатов, обрабатываемых одновременно, а стало быть, и общую пропускную способность приложения. В листинге 6.15

показано, как построить программу, реализующую эту архитектуру, на основе кода в листинге 5.7.

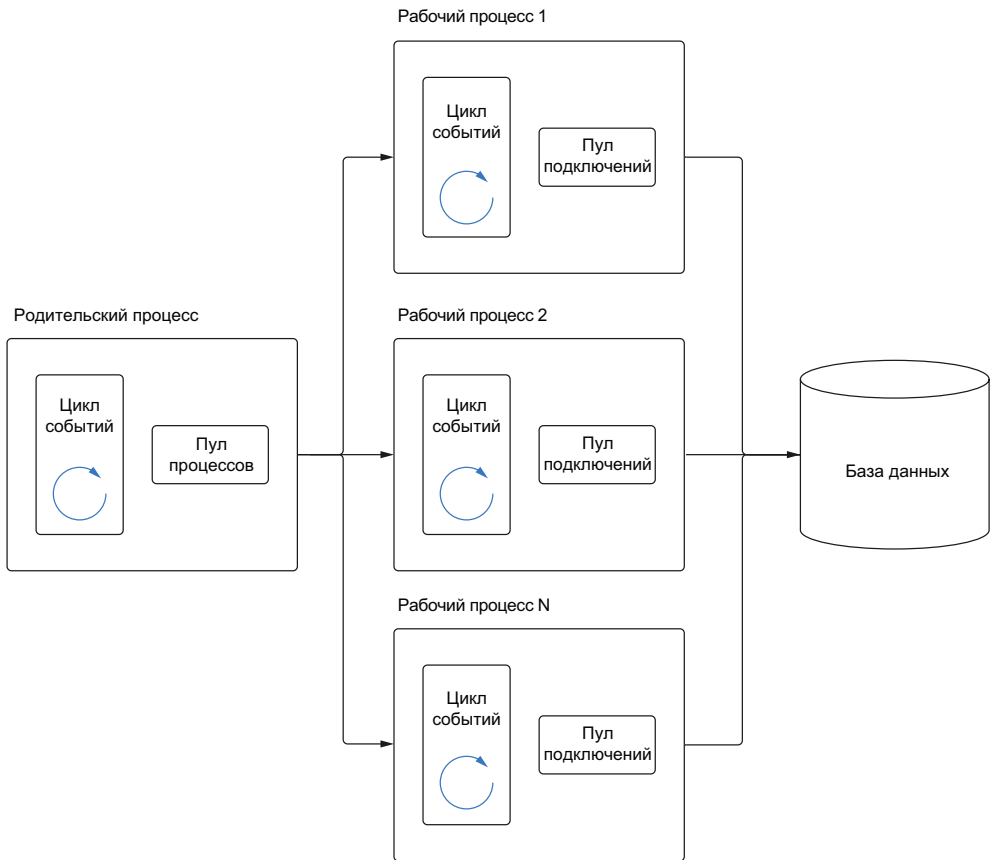


Рис. 6.7 Родительский процесс создает пул процессов, а затем исполнителей, каждый из которых имеет собственный цикл событий

Листинг 6.15 Цикл событий в каждом процессе

```
import asyncio
import asyncpg
from util import async_timed
from typing import List, Dict
from concurrent.futures.process import ProcessPoolExecutor

product_query = \
    """
SELECT
    p.product_id,
    p.product_name,
    p.brand_id,
    s.sku_id,
```

```
pc.product_color_name,
ps.product_size_name
FROM product as p
JOIN sku as s on s.product_id = p.product_id
JOIN product_color as pc on pc.product_color_id = s.product_color_id
JOIN product_size as ps on ps.product_size_id = s.product_size_id
WHERE p.product_id = 100"""
```

```
async def query_product(pool):
    async with pool.acquire() as connection:
        return await connection.fetchrow(product_query)

@async_timed()
async def query_products_concurrently(pool, queries):
    queries = [query_product(pool) for _ in range(queries)]
    return await asyncio.gather(*queries)

def run_in_new_loop(num_queries: int) -> List[Dict]:
    async def run_queries():
        async with asyncpg.create_pool(host='127.0.0.1',
                                        port=5432,
                                        user='postgres',
                                        password='password',
                                        database='products',
                                        min_size=6,
                                        max_size=6) as pool:
            return await query_products_concurrently(pool, num_queries)

    results = [dict(result) for result in asyncio.run(run_queries())]
    return results
```

Выполнять запросы
в новом цикле событий
и преобразовывать
их в словари

```
@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    pool = ProcessPoolExecutor()
    tasks = [loop.run_in_executor(pool, run_in_new_loop, 10000) for _ in
             range(5)]
    all_results = await asyncio.gather(*tasks)
    total_queries = sum([len(result) for result in all_results])
    print(f'Извлечено товаров из базы данных: {total_queries}.')

if __name__ == "__main__":
    asyncio.run(main())
```

Создать пять процессов, каждый
со своим циклом событий,
для выполнения запросов

Ждать получения
всех результатов

Мы написали новую функцию: `run_in_new_loop`. Внутри нее определена сопрограмма, `run_queries`, которая создает новый пул подключений и конкурентно выполняет заданное число запросов. Затем мы вызываем `run_queries` с помощью `asyncio.run`, которая исполняет ее в новом цикле событий.

Следует отметить, что мы преобразуем результаты в словари, потому что объекты записей `asyncpg` невозможно сериализовать в формате `pickle`. Преобразование в структуру данных, допускающую

сериализацию, гарантирует, что мы сможем отправить результат родительскому процессу.

В сопрограме `main` мы создаем пул процессов и пять раз вызываем функцию `gun_in_new_loop`. В результате будет конкурентно отправлено 50 000 запросов – по 10 000 на каждый из пяти процессов. Выполнив программу, вы увидите, что пять процессов запускаются быстро и заканчиваются примерно в одно и то же время. Время работы программы чуть больше, чем самого медленного процесса. На восьмиядерной машине скрипт работал примерно 13 с. Напомним, что в примере из главы 5 мы обработали 10 000 запросов примерно за 6 с. Значит, пропускная способность тогда составляла 1666 запросов в секунду. Сочетание многопроцессности и цикла событий позволило обработать 50 000 за 13 с, или около 3800 запросов в секунду. То есть пропускная способность увеличилась в два с лишним раза.

Резюме

- Мы узнали, как выполнять несколько функций Python параллельно с помощью пула процессов.
- Мы научились создавать исполнитель пула процессов и запускать функции Python параллельно. Исполнитель пула процессов позволяет использовать такие функции `asyncio API`, как `gather`, чтобы конкурентно выполнять несколько процессов и ждать результатов.
- Мы научились решать задачи с помощью модели программирования MapReduce с применением пула процессов и `asyncio`. Эта техника применима не только к MapReduce, но и вообще к любой счетной задаче, данные для которой можно разбить на меньшие порции.
- Мы узнали, как разделять между процессами информацию о состоянии. Это позволяет собирать данные о запущенных дочерних процессах, например счетчик чего-то.
- Мы научились избегать состояний гонки с помощью блокировок. Состояние гонки возникает, когда несколько процессов пытается обратиться к данным примерно в одно и то же время, и чревато трудно воспроизводимыми ошибками.
- Мы узнали, как использовать библиотеку `multiprocessing` для расширения возможностей `asyncio` путем создания отдельного цикла событий в каждом процессе. Потенциально это может повысить производительность для рабочих нагрузок, сочетающих счетные задачи и ввод-вывод.

Решение проблем блокирования с помощью потоков

Краткое содержание главы

- Введение в библиотеку `threading`.
- Создание пула потоков для обработки блокирующего ввода-вывода.
- Использование `async` и `await` для управления потоками.
- Использование пулов потоков для обработки блокирующего ввода-вывода.
- Разделяемые данные и блокировка на уровне потоков.
- Обработка счетных задач с помощью потоков.

При разработке с нуля нового приложения, ограниченного производительностью ввода-вывода, `asyncio` может стать естественным выбором. Мы с самого начала сможем использовать неблокирующие библиотеки, совместимые с `asyncio`, например `asyncpg` и `aiohttp`. Однако разработка с чистого листа, когда проект не обременен историческими ограничениями, – роскошь, которую могут позволить себе не многие разработчики. Значительная часть работы может приходиться на управление написанным ранее кодом, в котором используются блокирующие библиотеки ввода-вывода, например `requests` для отправки HTTP-запросов, `psycopg` – для работы с базами данных `Postgres` и т. д. Может быть и так, что совместимой с `asyncio` библиотеки еще не существует. И как тогда получить выигрыш от конкурентности, который сулит `asyncio`?

Ответ дает *многопоточность*. Поскольку блокирующие операции ввода-вывода освобождают глобальную блокировку интерпретатора, мы получаем возможность выполнять ввод-вывод конкурентно в разных потоках. Как и библиотека `multiprocessing`, `asyncio` позволяет использовать пулы потоков, т. е. мы можем получить все преимущества многопоточности, не отказываясь от таких API `asyncio`, как `gather` и `wait`.

В этой главе мы узнаем, как использовать многопоточность совместно с `asyncio`, чтобы выполнять блокирующие API, например `requests`, в потоках. Кроме того, мы научимся синхронизировать доступ к разделяемым данным, как в предыдущей главе, и рассмотрим такие продвинутые темы, как *реентерабельные блокировки* и *взаимоблокировки*. Также узнаем, как сочетать `asyncio` с синхронным кодом, для чего построим отзывчивый GUI для нагрузочного тестирования HTTP. Наконец, мы поговорим об исключительных ситуациях, в которых многопоточность можно использовать для решения счетных задач.

7.1 Введение в модуль *threading*

Для создания потоков и управления ими в Python применяется модуль `threading`. Он предлагает класс `Thread`, экземпляр которого принимает функцию, подлежащую выполнению в отдельном потоке. Интерпретатор Python однопоточный в том смысле, что в каждый момент времени может выполняться только один участок байт-кода, даже если в процессе работает несколько потоков. Глобальная блокировка интерпретатора не позволяет выполнять несколько потоков одновременно.

На первый взгляд кажется, что Python лишает нас возможности получить хоть какую-то выгоду от многопоточности, однако в некоторых случаях глобальная блокировка освобождается, и главный из них – операции ввода-вывода. Python может освободить GIL на время их выполнения, потому что для выполнения ввода-вывода вызывается низкоуровневая функция операционной системы. Эти функции работают за пределами интерпретатора, т. е. никак не могут повредить его внутренние структуры, от чего и призвана защитить GIL.

Чтобы лучше понять, как создавать и выполнять потоки в контексте блокирующего ввода-вывода, вернемся к примеру эхо-сервера из главы 3. Напомним, что для обработки нескольких подключений мы должны были перевести сокеты в неблокирующий режим и использовать модуль `select` для наблюдения за событиями сокетов. А что, если мы работаем с унаследованным кодом, который не дает возможности использовать неблокирующие сокеты? Можно ли тогда построить эхо-сервер, способный обрабатывать более одного клиента одновременно?

Поскольку методы сокета `recv` и `sendall` – это операции ввода-вывода и, стало быть, освобождают GIL, ничто не мешает выполнять их

в разных потоках конкурентно. А это значит, что мы можем создать по одному потоку для каждого подключившегося клиента и в этом потоке читать и записывать данные. Эта модель широко распространена; например, она используется в веб-сервере Apache. Чтобы проверить эту идею, будем ждать подключения в главном потоке, а затем создавать отдельный поток для эхо-копирования данных от клиента.

Листинг 7.1 Многопоточный эхо-сервер

```
from threading import Thread
import socket
```

```
def echo(client: socket):
    while True:
        data = client.recv(2048)
        print(f'Получно {data}, отправляю!')
        client.sendall(data)
```

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind(('127.0.0.1', 8000))
    server.listen()
    while True:
        connection, _ = server.accept()
        thread = Thread(target=echo, args=(connection,))
        thread.start()
```

Блокируется в ожидании
подключения клиентов

Начать выполнение потока

Как только клиент подключился,
создать поток для выполнения функции echo

Здесь мы в бесконечном цикле ждем подключений к серверному сокету. Когда клиент подключился, мы создаем новый поток для выполнения функции `echo`. В аргументе `target` потоку передается функция `echo`, а в аргументе `args` кортеж аргументов, которые должны быть переданы `echo`. Таким образом, в потоке будет вызвана `echo(connection)`. Затем мы запускаем поток и начинаем новую итерацию цикла, ожидая следующего подключения. Тем временем в созданном только что потоке выполняет цикл ожидания данных от клиента и их эхо-копирования.

Мы можем создать сколько угодно конкурентных telnet-клиентов, и все их сообщения будут корректно копироваться. Поскольку методы `recv` и `sendall` работают в отдельном потоке, они не блокируют друг друга; блокируется только тот поток, в котором они выполняются.

Это решает проблему невозможности одновременного подключения нескольких клиентов при использовании блокирующих сокетов, хотя у описанного подхода есть свои трудности, присущие только потокам. Что будет, если мы попытаемся снять процесс нажатием **CTRL+C** в момент, когда подключены клиенты? Будут ли созданные потоки корректно остановлены?

Оказывается, что нет. При снятии в методе `server.accept()` возникает исключение `KeyboardInterrupt`, но приложение не остановит-

ся, потому что этому помешают работающие фоновые потоки. Более того, подключенные клиенты продолжают отправлять и получать сообщения!

К сожалению, созданные пользователем потоки не получают исключения `KeyboardInterrupt`, оно возбуждается только в главном потоке. Поэтому наши потоки радостно продолжают читать данные от клиентов, не давая приложению завершиться.

Решить эту проблему можно двумя способами. Во-первых, можно использовать потоки-демоны, а во-вторых, придумать собственный способ снятия, или «прерывания», работающего потока. Демоны – это специальный вид потоков, предназначенный для выполнения длительных фоновых задач. Они не мешают приложению завершиться. На самом деле если работают только потоки-демоны, то приложение вообще завершается автоматически. Главный поток Python не является демоном, но если все потоки обслуживания подключений сделать демонами, то приложение завершится при возникновении `KeyboardInterrupt`. Перейти на потоки-демоны в листинге 7.1 просто – нужно лишь написать `thread.daemon = True` перед выполнением `thread.start()`. После этого изменения приложение будет правильно завершаться при нажатии **CTRL+C**.

Но у этого подхода есть проблема – потоки-демоны завершаются без уведомления, и мы не можем выполнить в этот момент никакой код очистки. Предположим, к примеру, что при завершении программы мы хотим отправить клиенту сообщение о том, что сервер будет остановлен. Можно ли как-то прервать наш поток и чисто остановить сокет? Если вызвать метод сокета `shutdown`, то все текущие вызовы `recv` вернут 0, а вызовы `sendall` возбудят исключение. Если вызвать `shutdown` из главного потока, то мы прервем потоки обслуживания клиентов, заблокированные в вызове `recv` или `sendall`. А затем можно будет обработать исключение в клиентском потоке и выполнить очистку.

Для этого будем создавать потоки немного иначе, чем раньше, – унаследуем их от самого класса `Thread`. Это позволит нам определить поток с методом `close`, в котором мы сможем остановить клиентский сокет. Тогда методы `recv` и `sendall` будут прерваны, и мы сможем выйти из цикла `while` и завершить поток.

В классе `Thread` имеется метод `run`, который можно переопределить. В подклассе `Thread` мы реализуем этот метод, поместив в него код, который должен выполнять поток после запуска. В нашем случае это цикл эхо-копирования, в котором вызываются методы `recv` и `sendall`.

Листинг 7.2 Создание подкласса `Thread` для чистой остановки

```
from threading import Thread
import socket
```

```
class ClientEchoThread(Thread):
```

```

def __init__(self, client):
    super().__init__()
    self.client = client

def run(self):
    try:
        while True:
            data = self.client.recv(2048)
            if not data:
                raise BrokenPipeError('Подключение закрыто!')
            print(f'Получено {data}, отправляю!')
            self.client.sendall(data)
    except OSError as e:
        print(f'Поток прерван исключением {e}, производится остановка!')

def close(self):
    if self.is_alive():
        self.client.sendall(bytes('Останавливаюсь!', encoding='utf-8'))
        self.client.shutdown(socket.SHUT_RDWR)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind(('127.0.0.1', 8000))
    server.listen()
    connection_threads = []
    try:
        while True:
            connection, addr = server.accept()
            thread = ClientEchoThread(connection)
            connection_threads.append(thread)
            thread.start()
    except KeyboardInterrupt:
        print('Останавливаюсь!')
        [thread.close() for thread in connection_threads]

```

Если нет данных, возбудить исключение. Это бывает, когда подключение было закрыто клиентом или остановлено сервером

В случае исключения выйти из метода `run`. При этом поток завершается

Разомкнуть подключение клиента, остановив чтение и запись

Разомкнуть подключение, если поток еще активен; поток может быть неактивен, если клиент закрыл подключение

Вызвать метод `close` созданных потоков, чтобы разомкнуть все клиентские подключения в случае прерывания с клавиатуры

Сначала мы создаем новый класс, `ClientEchoThread`, наследующий `Thread`. В нем переопределен метод `run`, теперь он включает код нашей функции `echo`, но с несколькими изменениями. Во-первых, весь код обернут блоком `try/catch`, в котором перехватываются исключения `OSError`. Такое исключение возбуждается методом `sendall` в случае закрытия клиентского сокета. Мы также проверяем, не вернул ли метод `recv` значение 0. Это бывает в двух случаях: если клиент закрыл подключение (например, кто-то завершил `telnet`) или если мы разомкнули клиентское подключение сами. В любом случае мы возбуждаем исключение `BrokenPipeError` (подкласс `OSError`), печатаем сообщение в блоке `except` и выходим из метода `run`, что приводит к остановке потока.

Также в классе `ClientEchoThread` определен метод `close`. Прежде чем разомкнуть клиентское подключение, он проверяет, активен ли поток. Что такое «активный» поток и зачем нам это делать? Поток называется активным, если выполняется его метод `run`; в данном случае

так обстоит дело, если наш метод `gun` не возбудил исключение. Эта проверка нужна, потому что сам клиент мог закрыть подключение, что привело бы к исключению `BrokenPipeError` в методе `gun` еще до вызова `close`. Но тогда вызов `sendall` возбудил бы исключение, поскольку подключение уже не функционально.

Наконец, в главном цикле мы прослушиваем порт в ожидании новых подключений и перехватываем исключение `KeyboardInterrupt`. Как только оно возникает, мы вызываем метод `close` каждого созданного потока. При этом клиенту отправляется сообщение, если подключение еще активно, после чего подключение размыкается.

Вообще говоря, снятие работающих потоков в Python и не только – непростая задача, решение которой зависит от конкретной ситуации. Нужно принимать специальные меры, чтобы потоки не помешали приложению завершиться, и решить, в каком месте размещать точки прерывания для выхода из потока.

Мы рассмотрели два способа ручного управления потоками: создание объекта потока с целевой функцией и порождение подкласса `Thread` с переопределением метода `run`. Разобравшись с основами многопоточности, посмотрим, как использовать ее совместно с `asyncio`, чтобы стало возможно работать с популярными блокирующими библиотеками.

7.2 Совместное использование потоков и `asyncio`

Теперь мы знаем, как создать несколько потоков и управлять ими для обработки блокирования. Недостаток в том, что потоки нужно создавать индивидуально и отслеживать их. А хотелось бы использовать API, основанные на `asyncio`, которые умеют дожидаться результата от потока, а не управлять потоками самостоятельно. Для этого можно использовать *пулы потоков*, похожие на пулы процессов из главы 6. В этом разделе мы познакомимся с популярной блокирующей клиентской библиотекой HTTP и посмотрим, как использовать библиотеки `threads` и `asyncio` для конкурентного выполнения веб-запросов.

7.2.1 Введение в библиотеку `requests`

Библиотека `requests` – популярная клиентская библиотека HTTP, которую автор описывает словами «HTTP для людей». Актуальная документация имеется по адресу <https://requests.readthedocs.io/en/master/>. Библиотека позволяет отправлять HTTP-запросы веб-серверу, как и `aiohttp`. Мы будем использовать последнюю версию (на момент написания книги – 2.24.0). Для установки библиотеки следует выполнить команду:

```
pip install -Iv requests==2.24.0
```

После установки все готово для отправки простых HTTP-запросов. Начнем с парочки запросов к сайту `example.com` с целью получить код состояния, как мы раньше проделывали с `aiohttp`.

Листинг 7.3 Базовое использование `requests`

```
import requests

def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code

url = 'https://www.example.com'
print(get_status_code(url))
print(get_status_code(url))
```

Здесь мы выполняем два запроса HTTP GET подряд. В ответ должны получить ответы 200. Мы не стали создавать HTTP-сеанс, как в случае `aiohttp`, хотя библиотека их поддерживает, чтобы сохранять куки между запросами.

Библиотека `requests` блокирующая, т. е. каждый запрос `requests.get` прекращает выполнение Python-кода в потоке до завершения запроса. Это влияет на способы ее использования совместно с `asyncio`. Если попытаться включить функции из нее в сопрограмму или в задачу, то весь цикл событий будет заблокирован до завершения запроса. Если запрос занимает 2 с, то в течение этого времени наше приложение ничего не сможет делать. Поэтому блокирующие операции следует выполнять в отдельном потоке.

7.2.2 Знакомство с исполнителями пула потоков

Как и для исполнителей пула процессов, библиотека `concurrent.futures` содержит реализацию абстрактного класса `Executor` для работы с потоками – `ThreadPoolExecutor`. Вместо пула рабочих процессов исполнитель пула потоков создает и обслуживает пул потоков, которому можно передавать задания.

Если пул процессов по умолчанию создает по одному рабочему процессу для каждого имеющегося процессорного ядра, то определение количества рабочих потоков несколько сложнее. По умолчанию оно равно `min(32, os.cpu_count() + 4)`. Это значит, что максимальное число потоков в пуле равно 32, а минимальное 5. Верхняя граница задается равной 32, чтобы избежать создания неожиданно большого числа потоков на машинах с большим количеством ядер (напомним, что создавать и обслуживать потоки дорого). А нижняя граница равна 5, потому что на машинах с одним-двумя ядрами запуск всего двух потоков вряд ли как-то улучшит производительность. Часто имеет смысл создать немного больше потоков, чем имеется ядер, если предполагается использовать их для ввода-вывода. Например, на 8-ядерной машине, согласно приведенной выше формуле, будет

создано 14 потоков. Конкурентно смогут работать только 8 потоков, а остальные будут ждать завершения ввода-вывода.

Модифицируем пример в листинге 7.3, так чтобы он конкурентно выполнял 1000 HTTP-запросов, пользуясь пулом потоков. И будем вести хронометраж, чтобы понять, получили ли мы какой-то выигрыш.

Листинг 7.4 Выполнение запросов с помощью пула потоков

```
import time
import requests
from concurrent.futures import ThreadPoolExecutor

def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code

start = time.time()

with ThreadPoolExecutor() as pool:
    urls = ['https:// www.example.com' for _ in range(1000)]
    results = pool.map(get_status_code, urls)
    for result in results:
        print(result)

end = time.time()

print(f'Выполнение запросов завершено за {end - start:.4f} с')
```

На 8-ядерной машине с быстрым интернетом этот код выполняется за 8–9 с, когда число потоков выбрано по умолчанию. Легко написать синхронную версию и выяснить, есть ли какой-то выигрыш от многопоточности.

```
start = time.time()

urls = ['https:// www.example.com' for _ in range(1000)]

for url in urls:
    print(get_status_code(url))

end = time.time()

print(f'Выполнение запросов завершено за {end - start:.4f} с')
```

Выполнение этого кода заняло 100 с! То есть многопоточный код оказался в 10 с лишним раз быстрее синхронного – неплохая прибавка к производительности!

Конечно, это улучшение, но в главе 4 было показано, что с `aiohttp` мы смогли выполнить 1000 запросов меньше чем за секунду. Почему же многопоточная версия оказалась настолько медленнее? Напомним, что максимальное число потоков ограничено числом 32. Это ограничение можно обойти, задав `max_workers=1000` при создании пула потоков:

```
with ThreadPoolExecutor(max_workers=1000) as pool:
    urls = ['https://www.example.com' for _ in range(1000)]
    results = pool.map(get_status_code, urls)
    for result in results:
        print(result)
```

Это может дать некоторое улучшение, потому что теперь потоков столько, сколько запросов. Но все равно мы даже не приблизимся к производительности кода, основанного на сопрограммах. Все дело в накладных расходах, связанных с потоками. Потоки создаются на уровне операционной системы и обходятся дороже сопрограмм. К тому же у контекстного переключения потоков на уровне ОС тоже есть цена. Сохранение и восстановление состояния потока при контекстном переключении съедает часть выигрыша, полученного от использования потоков.

Решая, сколько потоков нужно в приложении, лучше начать с малого (число ядер плюс еще немного – хорошая отправная точка), протестировать и постепенно увеличивать количество. Обычно удастся найти оптимальное число, после которого время работы выходит на плато и может даже уменьшиться, сколько бы потоков ни добавлять. Как правило, это число невелико по сравнению с числом запросов (короче – создавать 1000 потоков для отправки 1000 запросов, скорее всего, не стоит, только зря потратите ресурсы).

7.2.3 *Исполнители пула потоков и asyncio*

Использование исполнителей пула потоков в цикле событий *asyncio* мало чем отличается от использования класса *ProcessPoolExecutor*. В том-то и заключается прелесть абстрактного базового класса *Executor*, что мы можем выполнять один и тот же код с процессами или потоками, но только изменить одну строчку. Модифицируем приведенный выше пример выполнения 1000 запросов, заменив *pool.map* на *asyncio.gather*.

Листинг 7.5 Использование исполнителя пула потоков совместно с *asyncio*

```
import functools
import requests
import asyncio
from concurrent.futures import ThreadPoolExecutor
from util import async_timed

def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code

@async_timed()
async def main():
    loop = asyncio.get_running_loop()
```

```

with ThreadPoolExecutor() as pool:
    urls = ['https://www.example.com' for _ in range(1000)]
    tasks = [loop.run_in_executor(pool,
                                   functools.partial(get_status_code, url)) for url in urls]
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())

```

Мы создаем пул потоков, как и раньше, но вместо использования `map` строим список задач, вызывая функцию `get_status_code` из `loop.run_in_executor`. Получив список задач, мы можем ждать их завершения с помощью `asyncio.gather` или любой другой из уже знакомых нам функций `asyncio`.

Под капотом `loop.run_in_executor` вызывает метод `submit` исполнителя пула потоков. Это ставит все переданные задачи в очередь. Затем рабочие потоки в пуле могут выбирать задачи из очереди и выполнять их до завершения. Этот подход не дает никакого выигрыша по сравнению с использованием пула потоков без `asyncio`, но, пока мы ждем `await asyncio.gather`, может выполняться другой код.

7.2.4 Исполнители по умолчанию

В документации по `asyncio` сказано, что параметр `executor` метода `run_in_executor` может быть равен `None`. В этом случае используется *исполнитель по умолчанию*, ассоциированный с циклом событий. Можно считать, что это допускающий повторное использование синглтонный исполнитель для всего приложения. Исполнитель по умолчанию всегда имеет тип `ThreadPoolExecutor`, если с помощью метода `loop.set_default_executor` не было задано иное. Следовательно, мы можем упростить код в листинге 7.5, как показано ниже.

Листинг 7.6 Использование исполнителя по умолчанию

```

import functools
import requests
import asyncio
from util import async_timed

def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code

@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    urls = ['https://www.example.com' for _ in range(1000)]
    tasks = [loop.run_in_executor(None, functools.partial(get_status_code,
                                                            url)) for url in urls]
    results = await asyncio.gather(*tasks)

```



```
print(results)

asyncio.run(main())
```

Здесь мы не создаем собственный экземпляр `ThreadPoolExecutor` для использования в качестве контекстного менеджера, как раньше, а передаем в качестве исполнителя `None`. При первом вызове `run_in_executor` `asyncio` создает и кеширует исполнитель пула потоков по умолчанию. При последующих вызовах используется уже созданный исполнитель, т. е. он оказывается глобальным относительно цикла событий. Остановка такого пула также отличается от того, что мы видели раньше. Ранее созданный исполнитель пула потоков останавливался в момент выхода из блока `with`, управляемого контекстным менеджером. Что до исполнителя по умолчанию, то он существует до момента выхода из цикла событий, а это обычно происходит при завершении приложения. Использование исполнителя пула потоков по умолчанию упрощает программу, но нельзя ли пойти еще дальше?

В версии Python 3.9 появилась сопрограмма `asyncio.to_thread`, которая еще больше упрощает передачу работы исполнителю пула потоков по умолчанию. Она принимает функцию, подлежащую выполнению в потоке, и ее аргументы. Раньше для передачи аргументов нужно было использовать функцию `functools.partial`, так что теперь код стал немного чище. Затем эта функция выполняется с переданными аргументами в исполнителе по умолчанию и текущем цикле событий. Это позволяет еще упростить код. Сопрограмма `to_thread` устраняет необходимость в `functools.partial` и `asyncio.get_running_loop`, что уменьшает число строк кода.

Листинг 7.7 Использование сопрограммы `to_thread`

```
import requests
import asyncio
from util import async_timed

def get_status_code(url: str) -> int:
    response = requests.get(url)
    return response.status_code

@async_timed()
async def main():
    urls = ['https:// www.example.com' for _ in range(1000)]
    tasks = [asyncio.to_thread(get_status_code, url) for url in urls]
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())
```

До сих пор мы говорили только о том, как выполнять блокирующий код в потоках. Но сила сочетания потоков с `asyncio` еще и в том, что, пока мы ждем завершения потоков, можно выполнять другой код.

Чтобы увидеть, как это делается, вернемся к примеру из главы 6, в котором периодически печатали состояние долго работающей задачи.

7.3 Блокировки, разделяемые данные и взаимоблокировки

Многопоточный код, как и многопроцессный, подвержен состоянию гонки при использовании разделяемых данных, потому что управлять порядком выполнения потоков мы не можем. Всякий раз, как два потока или процесса потенциально могут изменить разделяемый элемент данных, следует использовать блокировку для синхронизации доступа. Концептуально это ничем не отличается от подхода, который мы применяли при работе с библиотекой `multiprocessing`, но из-за модели памяти в потоках детали немного изменяются.

Напомним, что создаваемые процессы по умолчанию не разделяют память. То есть мы должны были создавать и правильно инициализировать специальные объекты разделяемой памяти, чтобы все процессы могли читать и записывать их. Но потоки уже имеют доступ к памяти создавшего их процесса, поэтому ничего такого делать не нужно, а к разделяемым переменным возможен прямой доступ.

Это немного упрощает дело, но, так как мы теперь не работаем с разделяемыми объектами `Value`, в которые блокировки уже встроены, придется создавать их самим. Для этого нам понадобится реализация класса `Lock` в модуле `threading`, отличающаяся от реализации в модуле `multiprocessing`. Нужно лишь импортировать `Lock` из `threading` и окружить критические секции вызовами его методов `acquire` и `release` либо пометить их внутрь контекстного менеджера.

Работу с блокировками продемонстрируем на примере многопоточного варианта задачи из главы 6, отвечающей за отображение хода выполнения длительной работы. Возьмем задачу об отправке тысячи веб-запросов и с помощью разделяемого счетчика будем следить за тем, сколько запросов уже завершено.

Листинг 7.8 Печать информации о состоянии отправки запросов

```
import functools
import requests
import asyncio
from concurrent.futures import ThreadPoolExecutor
from threading import Lock
from util import async_timed

counter_lock = Lock()
counter: int = 0

def get_status_code(url: str) -> int:
    global counter
```

```
response = requests.get(url)
with counter_lock:
    counter = counter + 1
return response.status_code

async def reporter(request_count: int):
    while counter < request_count:
        print(f'Завершено запросов: {counter}/{request_count}')
        await asyncio.sleep(.5)

@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        request_count = 200
        urls = ['https://www.example.com' for _ in range(request_count)]
        reporter_task = asyncio.create_task(reporter(request_count))
        tasks = [loop.run_in_executor(pool,
            functools.partial(get_status_code, url)) for url in urls]
        results = await asyncio.gather(*tasks)
        await reporter_task
        print(results)

asyncio.run(main())
```

Этот код похож на тот, что был написан в главе 6 для печати информации о ходе выполнения операций `map`. Доступ к глобальной переменной `counter` нуждается в синхронизации. В функции `get_status_code` мы захватываем блокировку, перед тем как увеличить счетчик. Затем в сопрограмме `main` мы запускаем фоновый поток `reporter`, который каждые 500 мс выводит количество завершенных запросов. При выполнении программы мы увидим такую картину:

```
Завершено запросов: 0/200
Завершено запросов: 48/200
Завершено запросов: 97/200
Завершено запросов: 163/200
```

Мы уже многое знаем о блокировках в многопоточном и многопроцессном коде, но кое-что осталось за кадром. Рассмотрим понятие *реентерабельности*.

7.3.1 Реентерабельные блокировки

Простые блокировки годятся для координации доступа к разделяемой переменной со стороны нескольких потоков, но что, если поток попытается захватить блокировку, которую сам же и захватил ранее? Это вообще безопасно? Поскольку блокировку захватывает тот же поток, проблем вроде бы быть не должно, потому что он по определению однопоточный, а значит, операция потокобезопасна.

Так-то оно так, но если ограничиться блокировками, которые мы использовали до сих пор, то проблема все же возникнет. Чтобы по-

нять, в чем дело, допустим, что мы пишем рекурсивную функцию суммирования, которая принимает список целых чисел и возвращает их сумму. Список может модифицироваться несколькими потоками, поэтому нужно гарантировать, что никто не изменит его на протяжении суммирования. Попробуем реализовать это с помощью обычной блокировки и посмотрим, что получится. Включим также печать на консоль, чтобы понять, как продвигается выполнение функции.

Листинг 7.9 Блокировки и рекурсия

```
from threading import Lock, Thread
from typing import List

list_lock = Lock()

def sum_list(int_list: List[int]) -> int:
    print('Ожидание блокировки...')
    with list_lock:
        print('Блокировка захвачена.')
        if len(int_list) == 0:
            print('Суммирование завершено.')
            return 0
        else:
            head, *tail = int_list
            print('Суммируется остаток списка.')
            return head + sum_list(tail)

thread = Thread(target=sum_list, args=([1, 2, 3, 4],))
thread.start()
thread.join()
```

Эта программа печатает несколько сообщений, а потом зависает навечно:

```
Ожидание блокировки...
Блокировка захвачена.
Суммируется остаток списка.
Ожидание блокировки...
```

Что же тут происходит? В первый раз мы захватили блокировку `list_lock` успешно. Затем выделили первый элемент и остаток списка и рекурсивно вызвали `sum_list` для остатка. При этом мы попытались во второй раз захватить `list_lock`. Именно в этом месте программа и зависла, потому что блокировка уже захвачена и попытка захватить ее еще раз приводит к ожиданию освобождения. Но мы никогда не выйдем из блока `with` и не сможем освободить блокировку, т. е. мы ждем события, которое никогда не произойдет!

Но ведь рекурсия имеет место в одном потоке, а значит, многократный захват блокировки не должен вызывать проблем, потому что никакой гонки не возникает. Чтобы справиться с такими ситуация-

ми, библиотека `threading` предоставляет *реентерабельные* (повторно входимые) блокировки. Это специальный вид блокировки, который допускает неоднократный захват из одного потока, позволяя ему «повторно входить» в критические секции. Реентерабельные блокировки реализованы в классе `RLock`. Чтобы исправить код, нужно всего лишь изменить две строки: предложение `import` и создание `list_lock`, заменив их такими:

```
from threading import RLock

list_lock = RLock()
```

Теперь код будет работать правильно, и один поток сможет захватывать блокировки несколько раз. На внутреннем уровне реентерабельная блокировка хранит счетчик захватов. При каждом захвате блокировки потоком, захватившим ее впервые, счетчик увеличивается на 1, а при каждом освобождении счетчик уменьшается. Когда счетчик обратится в 0, блокировка освобождается и ее могут захватить другие потоки.

Рассмотрим более реалистичный сценарий, чтобы лучше понять концепцию рекурсии с блокировками. Пусть требуется построить потокобезопасный класс списка целых чисел, в котором имеется метод поиска всех элементов с указанным значением и замены его другим. Этот класс будет содержать обычный список Python и блокировку для предотвращения гонки. Предположим, что существующий класс уже содержит метод `indices_of(to_find: int)`, который принимает целое число и возвращает индексы всех элементов списка со значением `to_find`. Поскольку мы стремимся следовать принципу DRY («не повторяйся»), воспользуемся этим методом, чтобы определить метод поиска и замены (отметим, что технически это не самый эффективный способ решить задачу, но для иллюстрации идеи сойдет). Таким образом, наш класс и его метод будут выглядеть примерно так, как показано в листинге ниже.

Листинг 7.10 Класс потокобезопасного списка

```
from threading import Lock
from typing import List

class IntListThreadsafe:

    def __init__(self, wrapped_list: List[int]):
        self._lock = Lock()
        self._inner_list = wrapped_list

    def indices_of(self, to_find: int) -> List[int]:
        with self._lock:
            enumerator = enumerate(self._inner_list)
            return [index for index, value in enumerator if value == to_find]
```

```

def find_and_replace(self,
                      to_replace: int,
                      replace_with: int) -> None:
    with self._lock:
        indices = self.indices_of(to_replace)
        for index in indices:
            self._inner_list[index] = replace_with

threadsafe_list = IntListThreadsafe([1, 2, 1, 2, 1])
threadsafe_list.find_and_replace(1, 2)

```

Если другим потоком модифицируется список во время выполнения метода `indices_of`, то возможно получение неправильного значения, поэтому, перед тем как искать индексы, мы должны захватить блокировку. По той же причине должен захватить блокировку метод `find_and_replace`. Однако если бы блокировка была обычной, то мы навсегда зависли бы в `find_and_replace`, поскольку он сначала захватывает блокировку, а затем вызывает другой метод, пытающийся захватить ту же самую блокировку. В данном случае переход на `RLock` решает проблему, потому что `find_and_replace` дважды захватывает блокировку в одном потоке. Этот пример иллюстрирует общую ситуацию, когда бывают нужны реентерабельные блокировки. Если вы разрабатываете потокобезопасный класс с методом А, который захватывает блокировку, и методом В, который также захватывает блокировку и вызывает метод А, то, вероятно, нужна реентерабельная блокировка.

7.3.2 Взаимоблокировки

С термином *deadlock* (тупиковая ситуация) вы, возможно, встречались, читая в новостях о переговорах политических партий, когда одна сторона требует что-то от другой, а другая выдвигает встречное требование. Стороны не могут согласовать следующий шаг, и переговоры заходят в тупик. В информатике аналогичная ситуация возникает, когда имеет место неразрешимая конкуренция за разделяемый ресурс, в результате чего приложение зависает. Только здесь это называется взаимоблокировкой.

В предыдущем разделе мы видели пример взаимоблокировки, когда не реентерабельная блокировка может привести к зависанию. В этом случае в тупик зашли переговоры с самим собой, когда мы пытались захватить блокировку, которую сами же отказались освободить. Но такое возможно и тогда, когда два потока используют более одной блокировки. На рис. 7.1 поток А запрашивает блокировку, удерживаемую потоком В, а поток В – блокировку, удерживаемую потоком А. В итоге мы не можем никуда продвинуться – произошла взаимоблокировка. В таком случае реентерабельная блокировка не поможет, поскольку налицо несколько потоков, каждый из которых запрашивает ресурс, удерживаемый другим потоком.

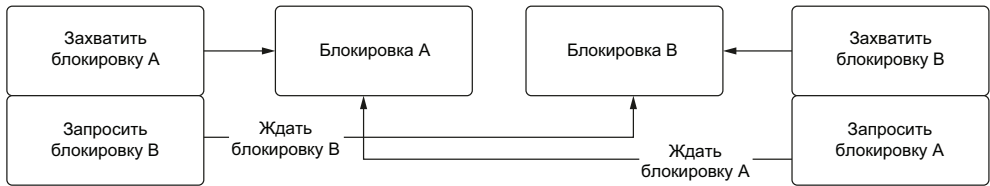


Рис. 7.1 Потоки А и В захватывают блокировки А и В примерно в одно и то же время. Затем поток А ждет блокировку В, которую удерживает поток В. Тем временем поток В ждет блокировку А, которую удерживает поток А. Из-за этой циклической зависимости возникает взаимоблокировка, и приложение зависает

Посмотрим, как возникает такая блокировка в программе. Мы создадим две блокировки, А и В, и два метода, пытающиеся захватить обе блокировки. Один метод сначала захватывает А, потом В, а другой – сначала В, потом А.

Листинг 7.11 Взаимоблокировка

```
from threading import Lock, Thread
import time

lock_a = Lock()
lock_b = Lock()

def a():
    with lock_a:  ← Захватить блокировку А
        print('Захвачена блокировка а из метода а!')
        time.sleep(1)  ← Ждать одну секунду; это создает
                        ← подходящие условия для
                        ← взаимоблокировки
        with lock_b:
            print('Захвачены обе блокировки из метода а!')

def b():
    with lock_b:
        print('Захвачена блокировка b из метода b!')
        with lock_a:
            print('Захвачены обе блокировки из метода b!')

thread_1 = Thread(target=a)
thread_2 = Thread(target=b)
thread_1.start()
thread_2.start()
thread_1.join()
thread_2.join()
```

Захватить блокировку В

Захватить блокировку А

При выполнении этой программы мы увидим следующие сообщения, после чего приложение зависнет навсегда:

```
Захвачена блокировка а из метода а!
Захвачена блокировка b из метода b!
```

Сначала мы вызываем метод А и захватываем блокировку А, затем вносим искусственную задержку, чтобы дать методу В возмож-

ность захватить блокировку В. И переходим в состояние, где метод А удерживает блокировку А, а метод В удерживает блокировку В. Затем метод А пытается захватить блокировку В, но ее удерживает метод В. Одновременно метод В пытается захватить блокировку А, которую удерживает метод А, ожидающий, пока В освободит свою блокировку. Каждый метод ждет, пока другой освободит ресурс, и мы не можем продвинуться ни на шаг.

Как выходить из этой ситуации? Одно из решений – «алгоритм страуса», названный так потому, что страус прячет голову в песок при виде опасности (хотя на самом деле страусы так себя не ведут). Заключается он в том, что мы игнорируем проблему и придумываем стратегию перезапуска приложения в случае ошибки. Идея такого подхода основана на предположении о том, что ошибка возникает настолько редко, что нет смысла ее исправлять. Если убрать из программы `sleep`, то взаимоблокировка будет встречаться очень редко, поскольку возникает лишь при очень специфической последовательности операций. Это, конечно, не исправление, и назвать такую стратегию идеальной язык не поворачивается, но она применяется, когда взаимоблокировки – событие редкое.

Но в нашей ситуации есть очень простое исправление – нужно только, чтобы блокировки в обоих методах захватывались в одном и том же порядке. Например, оба метода А и В могут сначала захватить блокировку А, а затем В. Это устраняет проблему, потому что мы никогда не захватываем блокировки в порядке, который мог бы привести к взаимоблокировке. Другое решение – перепроектировать программу, так чтобы использовать одну блокировку вместо двух. Если блокировка только одна, то взаимоблокировка невозможна в принципе (если не считать взаимоблокировку из-за нереентерабельности, которую мы видели выше). Вообще, столкнувшись с несколькими блокировками, следует спросить себя: «Всегда ли блокировки захватываются в согласованном порядке? Нельзя ли переделать код, так чтобы использовать только одну блокировку?»

Мы видели, как эффективно использовать потоки совместно с `asynсio` и изучили более сложные сценарии блокировки. Далее мы поговорим о том, как с помощью потоков интегрировать библиотеку `asynсio` с существующими приложениями, которые сами по себе, возможно, плохо совместимы с ней.

7.4 Циклы событий в отдельных потоках

Мы в основном интересовались приложениями, которые сверху до низу реализованы с применением сопрограмм и `asynсio`. Если какая-то работа не укладывалась в модель однопоточной конкурентности, то мы выполняли ее в потоках или процессах. Но не все приложения устроены согласно этой парадигме. Что, если мы имеем дело с уже

имеющимся синхронным приложением и хотим включить в него `asyncio`?

Например, в такой ситуации можно оказаться при построении пользовательских интерфейсов персональных приложений. В каркасах для разработки GUI обычно имеется собственный цикл событий, который блокирует главный поток. Это значит, что любая длительная операция может привести к заморозке пользовательского интерфейса. Кроме того, цикл событий UI мешает создать цикл событий `asyncio`. В этом разделе мы научимся использовать многопоточность для выполнения нескольких циклов событий одновременно на примере построения отзывчивого пользовательского интерфейса программы нагрузочного тестирования HTTP с помощью библиотеки Tkinter.

7.4.1 Введение в Tkinter

Tkinter – платформенно-независимая библиотека для построения графического интерфейса пользователя (GUI), включенная в стандартный дистрибутив Python. Название означает «Tk interface», т. е. это интерфейс к низкоуровневой библиотеке Tk, написанной на языке tcl. После создания библиотеки Tkinter для Python Tk обрела популярность среди Python-разработчиков, которые стали писать на ней пользовательские интерфейсы приложений для настольных ПК.

В Tkinter имеется набор «виджетов» – меток, текстовых полей, кнопок и т. д., которые можно размещать в окне на экране. При взаимодействии с виджетом, например при вводе текста и нажатии кнопки, активируется функция, выполняющая некий код. Код, исполняемый в ответ на действие пользователя, может быть совсем простым, скажем обновление другого виджета или инициирование другой операции.

Tkinter, как и многие другие библиотеки для построения GUI, отрисовывает виджеты и реализует взаимодействие с пользователем с помощью собственного цикла событий. В нем перерисовываются виджеты, обрабатываются события и проверяется, следует ли выполнить какой-то код в ответ на событие виджета. Чтобы познакомиться с Tkinter и ее циклом событий, напомним простое приложение типа `hello world`. В нем будет кнопка «say hello», при нажатии которой на консоль выводится сообщение «Привет!».

Листинг 7.12 Приложение «hello world» на Tkinter

```
import tkinter
from tkinter import ttk

window = tkinter.Tk()
window.title('Hello world app')
window.geometry('200x100')

def say_hello():
```

```
print('Привет!')

hello_button = ttk.Button(window, text='Say hello', command=say_hello)
hello_button.pack()

window.mainloop()
```

Эта программа сначала создает окно Tkinter (рис. 7.2) и задает название приложения и размер окна. Затем мы помещаем в окно кнопку и связываем с ней функцию `say_hello`. При нажатии кнопки эта функция выполнится и напечатает сообщение. После этого мы вызываем функцию `window.mainloop()`, запускающую цикл событий Tk, в котором исполняется приложение.

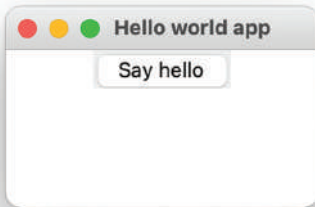


Рис. 7.2 Приложение «hello world» в листинге 7.12

Отметим, что приложение блокируется в функции `window.mainloop()`, которая исполняет цикл событий Tk. В этом бесконечном цикле проверяются события окна и перерисовываются виджеты; так продолжается до тех пор, пока мы не закроем окно. Между циклами событий Tk и `asyncio` есть интересные параллели. Например, что будет, если мы попытаемся выполнить блокирующую работу по нажатию кнопки? Добавив 10-секундную задержку в функцию `say_hello` с помощью вызова `time.sleep(10)`, мы тут же увидим проблему: приложение зависло на 10 с!

Как и `asyncio`, Tkinter выполняет *все* в своем цикле событий. А значит, если мы запустим длительную операцию, например отправку веб-запроса или загрузку большого файла, то заблокируем цикл событий Tk на все время ее выполнения. Итог – UI зависнет и перестанет отзываться на действия пользователя. Пользователь не сможет нажать на кнопку, виджеты, показывающие ход выполнения, перестанут обновляться, а операционная система, скорее всего, будет отображать индикатор занятости, как показано на рис. 7.3. Очевидно, что такое поведение нежелательно.

Это пример ситуации, в которой асинхронное программирование могло бы нас выручить. Если мы сумеем отправлять асинхронные запросы, не блокирующие цикл событий Tk, то избежим проблемы. Это труднее, чем кажется, потому что Tkinter ничего не знает об `asyncio`,

и задать в качестве обработчика команды сопрограмму невозможно. Можно было бы попытаться запустить в одном потоке сразу два цикла событий, но не получится. И Tkinter, и `asyncio` однопоточные, так что эта идея – то же самое, что пытаться одновременно выполнить два бесконечных цикла в одном потоке, – нонсенс! Если запустить цикл событий `asyncio` раньше цикла событий Tkinter, то первый блокирует второй. И наоборот. Так существует ли способ решить проблему?



Рис. 7.3 Зловещий «мячик судьбы» показывается в случае блокирования цикла событий на компьютерах Mac

На самом деле работающее приложение можно создать, если исполнять цикл событий `asyncio` в отдельном потоке. Посмотрим, как это сделать на примере приложения, которое будет информировать пользователя о состоянии длительной операции с помощью индикатора хода выполнения.

7.4.2 Построение отзывчивого UI с помощью `asyncio` и потоков

Сначала опишем приложение и его пользовательский интерфейс. Это будет программа для нагрузочного тестирования сайта. Оно принимает URL-адрес и количество запросов. При нажатии кнопки **Submit** мы будем с помощью `aiohhttp` отправлять запросы с максимальной скоростью, создавая заданную нагрузку на указанный веб-сервер. Поскольку это может занять много времени, добавим индикатор хода выполнения, чтобы видеть, далеко ли мы продвинулись. Индикатор будет обновляться после отправки 1 % запросов. Кроме того, мы предоставим пользователю возможность отменить запрос. UI будет содержать несколько виджетов: текстовые поля для ввода тестируемого URL и количества запросов, кнопку запуска и индикатор хода выполнения. Выглядеть он будет так, как показано на рис. 7.4.

С интерфейсом разобрались, теперь подумаем, как запустить одновременно два цикла событий. Идея в том, что цикл событий Tkinter должен работать в главном потоке, а цикл событий `asyncio` – в дополнительном. Когда пользователь нажимает кнопку **Submit**, мы передаем циклу событий `asyncio` сопрограмму, выполняющую нагрузочный

тест. Пока тест работает, мы можем отправлять команды обновления индикатора хода выполнения из цикла событий `asuncio` в цикл событий Tkinter. В итоге получается архитектура, показанная на рис. 7.5, которая включает взаимодействие между потоками. Но нужно внимательно следить за состояниями гонки, потому что цикл событий `asuncio` *не* является потокобезопасным! При проектировании Tkinter потокобезопасность учитывалась, так что вызывать ее цикл событий из другого потока можно (по крайней мере, в Python 3+; чуть ниже мы вернемся к этому вопросу).

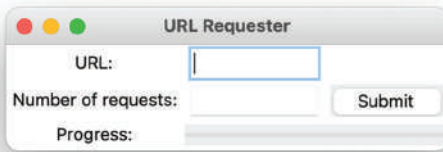


Рис. 7.4 Пользовательский интерфейс для задания URL-адреса

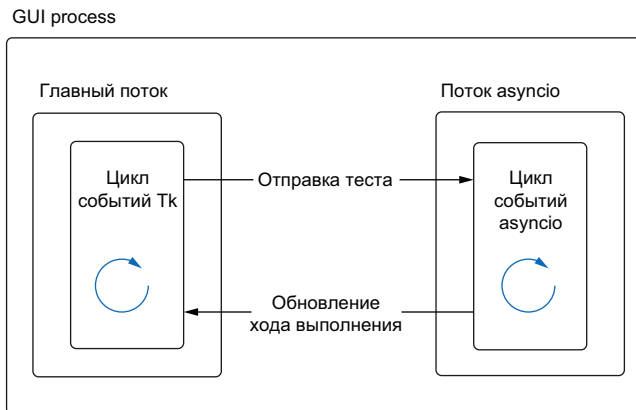


Рис. 7.5 Из цикла событий Tk задача передается циклу событий `asuncio`, который выполняется в отдельном потоке

Возникает искушение передавать сопрограммы из Tkinter с помощью `asuncio.run`, но эта функция блокирует выполнение, пока переданная сопрограмма не завершится, так что приложение Tkinter зависнет. Нужна функция, которая передает сопрограмму циклу событий без блокирования. Поэтому познакомимся еще с несколькими функциями `asuncio`, которые, с одной стороны, ничего не блокируют, а с другой, написаны с учетом потокобезопасности, поэтому могут использоваться для передачи задач. Первая из них – метод цикла событий `asuncio` `call_soon_threadsafe`. Он принимает функцию Python (не сопрограмму) и потокобезопасным образом планирует ее выполне-

ние на следующей итерации цикла событий. Вторая – `asyncio.run_coroutine_threadsafe`. Она принимает сопрограмму, потокобезопасным образом подает ее для выполнения и сразу же возвращает будущий объект, который позволит получить доступ к результату сопрограммы. Важно, что этот будущий объект является *не* будущим объектом `asyncio`, а экземпляром класса `future` из модуля `concurrent.futures`. Объясняется это тем, что будущие объекты `asyncio` потокобезопасны, в отличие от будущих объектов из `concurrent.futures`. Впрочем, этот класс `future` обладает той же функциональностью, что и будущие объекты из модуля `asyncio`.

Начнем с того, что реализуем несколько классов для нашего приложения на основе сказанного выше. Первым будем класс нагрузочного тестирования. Он отвечает за запуск и останов одного раунда тестирования и следит за тем, сколько запросов уже завершилось. Конструктор принимает URL-адрес, цикл событий `asyncio`, количество подлежащих выполнению запросов и функцию обратного вызова, обновляющую индикатор хода выполнения. Одним из членов класса является частота обновления, с которой вызывается эта функция. Мы вычисляем ее самостоятельно, так чтобы индикатор обновлялся периодически после отправки 1 % запросов.

Листинг 7.13 Класс нагрузочного тестирования

```
import asyncio
from concurrent.futures import Future
from asyncio import AbstractEventLoop
from typing import Callable, Optional
from aiohttp import ClientSession

class StressTest:

    def __init__(self,
                 loop: AbstractEventLoop,
                 url: str,
                 total_requests: int,
                 callback: Callable[[int, int], None]):
        self._completed_requests: int = 0
        self._load_test_future: Optional[Future] = None
        self._loop = loop
        self._url = url
        self._total_requests = total_requests
        self._callback = callback
        self._refresh_rate = total_requests // 100

    def start(self):
        future = asyncio.run_coroutine_threadsafe(self._make_requests(),
                                                  self._loop)
        self._load_test_future = future

    def cancel(self):
        if self._load_test_future:
```

Начать отправку запросов
и сохранить будущий объект,
чтобы впоследствии можно
было отменить тест

```

        self._loop.call_soon_threadsafe(self._load_test_future.cancel)

    async def _get_url(self, session: ClientSession, url: str):
        try:
            await session.get(url)
        except Exception as e:
            print(e)
        self._completed_requests = self._completed_requests + 1
        if self._completed_requests % self._refresh_rate == 0 \
            or self._completed_requests == self._total_requests:
            self._callback(self._completed_requests, self._total_requests)

    async def _make_requests(self):
        async with ClientSession() as session:
            reqs = [self._get_url(session, self._url) for _ in
                    range(self._total_requests)]
            await asyncio.gather(*reqs)

```

Чтобы отменить тест, нужно вызвать метод `cancel` объекта `_load_test_future`

После того как отправка 1 % запросов завершена, вызвать функцию обратного вызова, передав ей число завершенных запросов и общее число запросов

В методе `start` мы вызываем функцию `run_coroutine_threadsafe`, передавая ей функцию `_make_requests`. Она начинает отправлять запросы в цикле событий `asyncio`. Мы также запоминаем возвращенный будущий объект в переменной `_load_test_future`. Это позволит нам отменить нагрузочный тест в методе `cancel`. В методе `_make_requests` мы создаем список сопрограмм для всех веб-запросов и передаем его функции `asyncio.gather` для выполнения. Сопрограмма `_get_url` отправляет запрос, увеличивает счетчик `_completed_requests` и при необходимости вызывает функцию обратного вызова, передавая ей общее количество запросов и количество завершенных запросов. Для использования этого класса нужно просто создать его экземпляр и вызвать метод `start`. Если понадобится, позже можно будет вызвать метод `cancel`.

Интересно, что мы не ставили блокировку вокруг доступа к счетчику `_completed_requests`, хотя он и производится из нескольких сопрограмм. Напомним, что библиотека `asyncio` однопоточная и цикл событий `asyncio` в каждый момент времени исполняет только один кусок Python-кода. Поэтому увеличение счетчика оказывается атомарной операцией внутри `asyncio`, хотя при выполнении из разных потоков она была бы неатомарной. `asyncio` избавляет нас от многих видов состояния гонки, присущих многопоточности. От многих, но не от всех. Мы изучим этот вопрос в следующей главе.

Далее мы реализуем GUI для работы с классом нагрузочного тестирования. Чтобы сделать код чище, унаследуем напрямую от класса `TK` и инициализируем виджеты в конструкторе. При нажатии на кнопку

Submit мы создаем новый экземпляр класса `StressTest` и вызываем его метод `start`. Вопрос – что передать в качестве функции обратного вызова экземпляру `StressTest`? Тут потокобезопасность выступает на первый план, потому что эта функция будет вызываться из рабочего потока. Если она изменяет разделяемые данные, которые может модифицировать также главный поток, то возникает риск гонки. В нашем случае, поскольку Tkinter потокобезопасна, а мы всего лишь обновляем индикатор хода выполнения, все должно быть нормально. Но что, если нужно было бы сделать еще что-то с разделяемыми данными? Один из подходов – блокировка, но если бы мы могли выполнять функцию обратного вызова в главном потоке, то вообще избежали бы состояния гонки. Мы воспользуемся общим паттерном, чтобы продемонстрировать, как это делается, хотя можно было бы обновлять индикатор и напрямую.

Для реализации предложенной идеи можно использовать разделяемую потокобезопасную очередь из модуля `queue`. Наш поток `asuncio` может помещать запросы на обновление индикатора хода выполнения в эту очередь. Затем поток Tkinter может проверить, есть ли что-то в очереди, и обновить индикатор именно в том потоке, где это безопасно. Но мы должны сказать Tkinter, что нужно опрашивать очередь в главном потоке.

В Tkinter имеется метод `after`, позволяющий запланировать вызов функции с заданной периодичностью. Мы воспользуемся им для периодической проверки очереди обновлений индикатора (листинг 7.14). Если очередь не пуста, то мы можем безопасно обновить индикатор в главном потоке. Опрашивать очередь будем раз в 25 мс, чтобы задержка обновления была незаметна.

Действительно ли Tkinter потокобезопасна?

Поиск в сети по ключевым словам «Tkinter» и «thread safety», вы обнаружите массу противоречивой информации. Ситуация с потокобезопасностью в Tkinter сильно запутана. Отчасти это объясняется тем, что в течение нескольких лет в Tk и Tkinter не было надлежащей поддержки многопоточности. Даже после добавления модуля `threaded` осталось несколько ошибок, которые с тех пор были исправлены. Tk поддерживает два режима: с потоками и без. В режиме без потоков потокобезопасность вообще не гарантируется, а использование Tkinter в потоке, отличном от главного, неминуемо ведет к краху. В старых версиях Python режим потокобезопасности Tk не был включен, но, начиная с Python 3, он включен по умолчанию, так что потокобезопасность теперь гарантируется. В режиме с потоками если обновление инициируется из рабочего потока, то Tkinter захватывает мьютекс и помещает событие обновления в очередь для последующего выполнения из главного потока. В CPython соответствующий код находится в функции `Tkapp_Call` в файле `Modules/_tkinter.c`.

Листинг 7.14 TkInter GUI

```

from queue import Queue
from tkinter import Tk
from tkinter import Label
from tkinter import Entry
from tkinter import ttk
from typing import Optional
from chapter_07.listing_7_13 import StressTest

class LoadTester(Tk):
    def __init__(self, loop, *args, **kwargs):
        Tk.__init__(self, *args, **kwargs)
        self._queue = Queue()
        self._refresh_ms = 25

        self._loop = loop
        self._load_test: Optional[StressTest] = None
        self.title('URL Requester')

        self._url_label = Label(self, text="URL:")
        self._url_label.grid(column=0, row=0)

        self._url_field = Entry(self, width=10)
        self._url_field.grid(column=1, row=0)

        self._request_label = Label(self, text="Number of requests:")
        self._request_label.grid(column=0, row=1)

        self._request_field = Entry(self, width=10)
        self._request_field.grid(column=1, row=1)

        self._submit = ttk.Button(self, text="Submit", command=self._start)
        self._submit.grid(column=2, row=1)

        self._pb_label = Label(self, text="Progress:")
        self._pb_label.grid(column=0, row=3)

        self._pb = ttk.Progressbar(self, orient="horizontal", length=200,
                                   mode="determinate")
        self._pb.grid(column=1, row=3, columnspan=2)

        def _update_bar(self, pct: int):
            if pct == 100:
                self._load_test = None
                self._submit['text'] = 'Submit'
            else:
                self._pb['value'] = pct
                self.after(self._refresh_ms, self._poll_queue)

        def _queue_update(self, completed_requests: int, total_requests: int):
            self._queue.put(int(completed_requests / total_requests * 100))

        def _poll_queue(self):

```

В конструкторе инициализируем поля ввода, метки, кнопку отправки и индикатор хода выполнения

При нажатии на кнопку Submit вызывается метод `_start`

Метод `_update_bar` устанавливает процент заполненности индикатора хода выполнения от 0 до 100. Его следует вызывать только из главного потока

Извлечь обновление индикатора из очереди. Если получилось, обновить индикатор

Этот метод является обратным вызовом, который передается нагрузочному тесту; он добавляет обновление индикатора в очередь


```

if not self._queue.empty():
    percent_complete = self._queue.get()
    self._update_bar(percent_complete)
else:
    if self._load_test:
        self.after(self._refresh_ms, self._poll_queue)

def _start(self):
    if self._load_test is None:
        self._submit['text'] = 'Cancel'
        test = StressTest(self._loop,
                           self._url_field.get(),
                           int(self._request_field.get()),
                           self._queue_update)
        self.after(self._refresh_ms, self._poll_queue)
        test.start()
        self._load_test = test
    else:
        self._load_test.cancel()
        self._load_test = None
        self._submit['text'] = 'Submit'

```

Начать нагрузочное тестирование и каждые 25 мс опрашивать очередь обновлений

В конструкторе приложения мы создаем все виджеты пользовательского интерфейса, в том числе поля ввода для подлежащего тестированию URL-адреса и количества запросов, кнопку **Submit** и горизонтальный индикатор хода выполнения. Кроме того, с помощью метода `grid` мы аккуратно располагаем эти виджеты в окне. При создании кнопки в качестве команды задается метод `_start`. Он создает объект `StressTest` и запускает его, если тест еще не выполняется, а если выполняется, то останавливает его. Конструктору объекта `StressTest` в числе прочих параметров передается функция обратного вызова – метод `_queue_update`. Этот метод будет вызываться, когда потребуется обновить индикатор хода выполнения. Он вычисляет процент обновления и помещает его в очередь. Впоследствии мы используем метод `Tkinter after`, чтобы запланировать периодический вызов метода `_poll_queue` с интервалом 25 мс.

Применение очереди в качестве разделяемого механизма коммуникации вместо непосредственного вызова `_update_bar` гарантирует, что метод `_update_bar` будет вызван из потока цикла событий `Tkinter`. В противном случае индикатор обновлялся бы в цикле событий `asyncio`, потому что именно в этом потоке работает функция обратного вызова.

Реализовав пользовательский интерфейс, мы можем объединить все части в работающее приложение. Создадим новый поток для выполнения цикла событий в фоновом режиме, после чего запустим приложение `LoadTester`.

Листинг 7.15 Приложение для нагрузочного тестирования

```

import asyncio
from asyncio import AbstractEventLoop

```

```

from threading import Thread
from chapter_07.listing_7_14 import LoadTester

class ThreadedEventLoop(Thread):
    def __init__(self, loop: AbstractEventLoop):
        super().__init__()
        self._loop = loop
        self.daemon = True

    def run(self):
        self._loop.run_forever()

loop = asyncio.new_event_loop()

asyncio_thread = ThreadedEventLoop(loop)
asyncio_thread.start()

app = LoadTester(loop)
app.mainloop()

```

Создать новый класс потока, в котором будет крутиться цикл событий `asyncio`

Запустить новый поток, исполняющий цикл событий `asyncio` в фоновом режиме

Создать приложение `Tkinter` и запустить его главный цикл событий

Сначала мы определяем класс `ThreadedEventLoopClass`, наследующий `Thread`, для исполнения цикла событий. Конструктор класса принимает цикл событий и переводит поток в режим демона. Это делается, потому что цикл событий блокирующий и будет вечно крутиться в этом потоке. Такой тип бесконечного цикла не дал бы остановить GUI-приложение, если бы поток не был демоном. В методе потока `run` мы вызываем метод цикла событий `run_forever`. Название выбрано очень правильно, потому что запущенный цикл событий действительно будет работать вечно, блокируя поток, пока мы явно не остановим.

Далее мы создаем новый цикл событий `asyncio` методом `new_event_loop`. Затем создается и запускается экземпляр `ThreadedEventLoop`, которому передается цикл событий. В результате будет создан новый поток, в котором работает наш цикл событий. И наконец, мы создаем экземпляр приложения `LoadTester` и вызываем метод `mainloop`, который запускает цикл событий `Tkinter`.

Запустив приложение, мы увидим, что индикатор хода выполнения плавно обновляется, т. е. пользовательский интерфейс не заморожен. Приложение отзывается на действия пользователя, и мы можем в любой момент нажать кнопку **Cancel**, чтобы остановить нагрузочное тестирование. Такая техника выполнения цикла событий `asyncio` в отдельном потоке полезна для построения отзывчивых интерфейсов, да и вообще в любых синхронных унаследованных приложениях, плохо уживающихся с сопрограммами и `asyncio`.

Итак, мы видели, как использовать потоки для различных рабочих нагрузок, связанных с вводом-выводом, а как насчет счетных рабочих нагрузок? Напомним, что GIL не дает конкурентно выполнять байткод Python в потоках, но имеется несколько исключений из этого правила, благодаря которым определенные виды счетных задач можно запускать в потоках.

7.5 Использование потоков для выполнения счетных задач

Глобальная блокировка интерпретатора в Python – штука хитрая. Есть общее правило – многопоточность имеет смысл только для блокирующего ввода-вывода, потому что на время выполнения операций ввода-вывода GIL освобождается. Это справедливо в большинстве случаев, но не во всех. Чтобы можно было освободить GIL, не опасаясь ошибок из-за конкурентности, код не должен взаимодействовать ни с какими объектами Python (словарями, списками, целыми числами и т. д.). Такое возможно, когда значительная часть библиотеки реализована в виде низкоуровневого кода на C. И существуют хорошо известные библиотеки, например `hashlib` и `NumPy`, которые выполняют счетную работу на чистом C и освобождают GIL. Это позволяет использовать многопоточность, чтобы повысить производительность некоторых счетных рабочих нагрузок. Мы рассмотрим два примера: хеширование секретного текста в целях безопасности и анализ данных с применением `NumPy`.

7.5.1 *hashlib и многопоточность*

В современном мире безопасность превыше всего. Защита от чтения данных хакерами позволяет воспрепятствовать утечке конфиденциальной информации о клиентах (паролей и других данных) и избежать нанесения им ущерба.

Алгоритмы хеширования решают эту проблему, преобразуя входные данные так, что их невозможно ни прочитать, ни восстановить в исходном виде (если алгоритм безопасен). Например, пароль «password» может быть свернут в строку `'a12bc21df'`. Но хотя ни прочитать, ни восстановить исходный вид данных нельзя, мы все же можем проверить, что блок данных соответствует свертке. Это полезно, например, когда требуется проверить пароль пользователя или убедиться, что данные не были модифицированы.

Существует много алгоритмов хеширования, например `SHA512`, `BLAKE2` и `scrypt`, хотя `SHA` не считается оптимальным выбором для хранения паролей, потому что уязвим для атак полным перебором. Некоторые из этих алгоритмов реализованы в библиотеке `hashlib` для Python. Многие функции из этой библиотеки освобождают GIL, когда хешируют данные длиннее 2048 байт, поэтому многопоточность – одно из средств повысить производительность библиотеки. Кроме того, функция `scrypt`, применяемая для хеширования паролей, всегда освобождает GIL.

Рассмотрим гипотетический (надеемся) сценарий, демонстрирующий, когда многопоточность может оказаться полезной в сочетании с `hashlib`. Допустим, вы только что начали работать в новой должности главного архитектора программных систем в успешной

организации. Ваш начальник поручает вам исправить первую ошибку, чтобы вы поняли, как устроен процесс разработки в компании, – небольшой дефект в подсистеме входа. Вы начали с просмотра нескольких таблиц базы данных и к своему ужасу обнаружили, что все пароли клиентов хранятся в открытом виде! Значит, если база данных будет скомпрометирована, в руках злоумышленников окажутся пароли и они смогут входить от имени пользователей и получать конфиденциальные данные, например номера кредитных карт. Вы рассказываете о проблеме начальнику, и тот просит вас срочно найти решение.

Для хеширования паролей хорошо подходит алгоритм `scrypt`. Он безопасен, и исходный пароль не может быть восстановлен, поскольку используется *соль* – случайное число, гарантирующее, что свертка пароля уникальна. Для тестирования `scrypt` можно по-быстрому написать синхронный скрипт, который создает случайные пароли и хеширует. Это позволит понять, сколько времени займет вся процедура. Для примера будем тестировать на 10 000 случайных паролях.

Листинг 7.16 Хеширование паролей с помощью алгоритма `scrypt`

```
import hashlib
import os
import string
import time
import random

def random_password(length: int) -> bytes:
    ascii_lowercase = string.ascii_lowercase.encode()
    return b''.join(bytes(random.choice(ascii_lowercase)) for _ in
        range(length))

passwords = [random_password(10) for _ in range(10000)]

def hash(password: bytes) -> str:
    salt = os.urandom(16)
    return str(hashlib.scrypt(password, salt=salt, n=2048, p=1, r=8))

start = time.time()

for password in passwords:
    hash(password)

end = time.time()
print(end - start)
```

Функция `random_password` создает случайный пароль, состоящий из строчных букв. Мы используем ее для создания 10 000 случайных паролей по 10 символов каждый. Затем каждый пароль хешируется функцией `scrypt`. Опустим детали (параметры `n`, `p`, `r` функции `scrypt`), скажем только, что они позволяют настроить уровень безопасности и потребление памяти и процессора.

Выполнение этой программы на 8-ядерной машине с тактовой частотой процессора 2,4 ГГц занимает чуть больше 40 с, не так уж плохо. Беда в том, что база данных пользователей велика, и хешировать предстоит 1 000 000 000 паролей. Простой расчет показывает, что на это уйдет 40 с лишним дней! Можно было бы разбить набор данных на части и выполнить процедуру на нескольких машинах, но при такой скорости машин понадобится чересчур много. А нельзя ли воспользоваться многопоточностью и повысить скорость, а значит, уменьшить время и количество машин? Применим все, что мы знаем о многопоточности к проверке этой идеи. Создадим пул потоков и будем хешировать пароли в нескольких потоках.

Листинг 7.17 Хеширование с применением многопоточности и `asyncio`

```
import asyncio
import functools
import hashlib
import os
from concurrent.futures.thread import ThreadPoolExecutor
import random
import string

from util import async_timed

def random_password(length: int) -> bytes:
    ascii_lowercase = string.ascii_lowercase.encode()
    return b''.join(bytes(random.choice(ascii_lowercase)) for _ in
                      range(length))

passwords = [random_password(10) for _ in range(10000)]

def hash(password: bytes) -> str:
    salt = os.urandom(16)
    return str(hashlib.scrypt(password, salt=salt, n=2048, p=1, r=8))

@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    tasks = []

    with ThreadPoolExecutor() as pool:
        for password in passwords:
            tasks.append(loop.run_in_executor(pool, functools.partial(hash,
                                password)))

    await asyncio.gather(*tasks)

asyncio.run(main())
```

Этот подход подразумевает создание исполнителя пула потоков и задачи для каждого подлежащего хешированию пароля. Поскольку `hashlib` освобождает GIL, мы получаем вполне приличный выигрыш

в производительности. Этот код работает примерно 5 с вместо прежних 40. Мы уменьшили время работы с 47 дней до чуть больше пяти! На следующем шаге мы могли бы выполнить это приложение на нескольких машинах или взять машину с большим числом ядер и тем самым еще сократить время.

7.5.2 Многопоточность и NumPy

NumPy – чрезвычайно популярная Python-библиотека, она широко используется в науке о данных и машинном обучении. Она содержит множество математических функций для работы с массивами и матрицами, которые намного быстрее функций для работы с обычными массивами Python. Это превосходство объясняется тем, что на внутреннем уровне библиотека написана на низкоуровневых языках C и Fortran, превосходящих Python по производительности.

Поскольку многие операции библиотеки выполняются вне Python, открывается возможность освободить GIL и выполнять часть кода в нескольких потоках. Засада в том, что эта функциональность плохо документирована, но, вообще говоря, безопасно предполагать, что операции над матрицами допускают многопоточное выполнение. Тем не менее в зависимости от того, как реализована функция `numpy`, выигрыш может оказаться большим или малым. Если код напрямую вызывает функции, написанные на C, которые освобождают GIL, то выигрыш будет больше; если же низкоуровневые вызовы обернуты большим количеством кода на Python, то меньше. Учитывая недостаточность документации, можете поэкспериментировать и добавить многопоточность в критические участки приложения (какие участки нуждаются в ускорении, покажет профилирование), а затем проверить, получен ли какой-нибудь выигрыш. И решить, оправдывает ли потенциальный выигрыш дополнительную сложность.

Для демонстрации создадим большую матрицу, содержащую 4 000 000 000 элементов в 50 строках. Задача – вычислить среднее для каждой строки. В NumPy имеется для этой цели эффективная функция `mean`. Ее параметр `axis` позволяет вычислять все средние по одной оси без написания цикла. В нашем примере ось 1 означает, что среднее вычисляется по строкам.

Листинг 7.18 Вычисление средних в большой матрице с помощью NumPy

```
import numpy as np
import time

data_points = 4000000000
rows = 50
columns = int(data_points / rows)

matrix = np.arange(data_points).reshape(rows, columns)
```

```
s = time.time()

res = np.mean(matrix, axis=1)

e = time.time()
print(e - s)
```

Этот скрипт сначала создает массив с 4 000 000 000 целочисленных элементов в диапазоне 1 000 000 000–4 000 000 000 (отметим, что для этого необходимо много памяти; если приложение вылетает из-за нехватки памяти, уменьшите это число). Затем мы реорганизуем массив в матрицу с 50 строками. И наконец, вызываем функцию NumPy `mean` с параметром `axis`, равным 1, чтобы вычислить средние по строкам. На 8-ядерной машине с процессором 2,4 ГГц этот скрипт работает 25–30 с. Немного изменим код, сделав его многопоточным. Будем вычислять средние по каждой строке в отдельном потоке и с помощью `asyncio.gather` дождемся получения всех результатов.

Листинг 7.19 Многопоточность с NumPy

```
import functools
from concurrent.futures.thread import ThreadPoolExecutor
import numpy as np
import asyncio
from util import async_timed

def mean_for_row(arr, row):
    return np.mean(arr[row])

data_points = 4000000000
rows = 50
columns = int(data_points / rows)

matrix = np.arange(data_points).reshape(rows, columns)

@async_timed()
async def main():
    loop = asyncio.get_running_loop()
    with ThreadPoolExecutor() as pool:
        tasks = []
        for i in range(rows):
            mean = functools.partial(mean_for_row, matrix, i)
            tasks.append(loop.run_in_executor(pool, mean))

        results = asyncio.gather(*tasks)

asyncio.run(main())
```

Сначала мы создаем функцию `mean_for_row`, которая вычисляет среднее для одной строки. Поскольку планируем вычислять средние для каждой строки в отдельном потоке, уже нельзя использовать `mean` с параметром `axis`, как раньше. Затем в сопрограмме `main` мы создаем исполнитель пула потоков и задачи для вычисления среднего по

каждой строке. После чего с помощью `gather` ждем завершения всех вычислений.

На той же машине этот код работает 9-10 с, т. е. мы получили ускорение в 3 раза! Многопоточность может помочь в некоторых программах с применением NumPy, хотя на момент написания книги документация не сообщает, какие именно функции могут выиграть от наличия потоков. Если не уверены, стоит ли распараллеливать счетную рабочую нагрузку, тестируйте.

Кроме того, имейте в виду, что до экспериментов с многопоточностью или многопроцессностью код с применением NumPy должен быть максимально векторизован. Это означает, что нужно избегать циклов Python или функций типа `apply_along_axis`, которые просто скрывают цикл. Часто NumPy позволяет достичь гораздо большего, если перенести максимум вычислений на уровень библиотеки.

Резюме

- Мы узнали, как применять модуль `threading` для выполнения ввода-вывода.
- Мы научились корректно завершать потоки при остановке приложения.
- Мы научились использовать исполнитель пула потоков для распределения работы между потоками. Это позволяет использовать такие методы `asyncio` API, как `gather`, чтобы дождаться результатов всех потоков.
- Мы узнали, как выполнить блокирующие API ввода-вывода, например из библиотеки `requests`, в пулах потоков и воспользоваться `asyncio` для получения выигрыша в производительности.
- Мы научились избегать состояний гонки с помощью блокировок из модуля `threading`. Также узнали, как реентерабельные блокировки помогают избежать взаимоблокировок.
- Мы узнали, как выполнять цикл событий `asyncio` в отдельном потоке и передавать ему сопрограммы потокобезопасным способом. Это позволяет строить отзывчивые пользовательские интерфейсы с помощью таких библиотек, как `TkInter`.
- Мы научились применять многопоточность в сочетании с библиотеками `hashlib` и `numpy`. Низкоуровневые библиотеки иногда освобождают GIL, что позволяет использовать потоки даже для счетных задач.

Потоки данных

Краткое содержание главы

- Транспортные механизмы и протоколы.
- Использование потоков данных для организации сетевых подключений.
- Асинхронная обработка ввода из командной строки.
- Создание клиент-серверных приложений с помощью потоков данных.

В предыдущих главах при разработке сетевых приложений, в частности клиентов эхо-сервера, мы использовали библиотеку сокетов для чтения и записи данных клиентам. Работа с сокетами напрямую полезна при построении низкоуровневых библиотек, но вообще-то это довольно сложные структуры со своими нюансами, подробное рассмотрение которых выходит за рамки этой книги. Однако во многих случаях использование сокетов сводится к нескольким концептуально простым операциям, например: запуск сервера, ожидание запроса на подключение от клиентов и отправка данных клиентам. Проектировщики `asyncio` осознавали это и включили API сетевых потоков данных (`stream`), чтобы скрыть от нас нюансы сокетов. С этими высокоуровневыми API работать гораздо легче, чем с сокетами, поэтому клиент-серверные приложения и писать проще, и надежность их повышается. Потоки данных – рекомендуемый способ построения сетевых приложений в `asyncio`.

В этой главе мы сначала рассмотрим низкоуровневые API транспортных механизмов и протоколов на примере простого HTTP-клиента. Это даст нам основу для понимания устройства высокоуровневых API. Затем мы воспользуемся полученными знаниями и поговорим о потоковых читателях и писателях и их использовании для построения неблокирующего командного SQL-клиента. Это приложение будет асинхронно обрабатывать ввод данных пользователем, что позволит конкурентно выполнять несколько запросов из командной строки. Наконец, мы узнаем, как использовать серверный API `asynio` для создания клиентских и серверных приложений на примере работоспособного чат-сервера и его клиента.

8.1 *Введение в потоки данных*

В `asynio` *потоки данных*¹ представляют собой высокоуровневый набор классов и функций для создания и управления сетевыми подключениями и вообще потоками данных. С их помощью мы можем создавать клиентские подключения для чтения и записи данных на сервер. Более того, мы можем сами создавать серверы и управлять ими. Эти API абстрагируют обширные знания, необходимые для управления сокетами, например о работе SSL и о потерянных подключениях. Это немного упрощает жизнь разработчикам.

Потоковые API надстроены над низкоуровневыми API *транспорта и протоколов*. Они обертывают сокеты (и вообще любой поток данных), предоставляя чистый API для чтения и записи данных.

Устроены эти API несколько иначе, чем прочие; в них используются обратные вызовы. Вместо того чтобы активно ждать данные от сокета, как мы делали раньше, библиотека сама вызовет написанный нами метод класса в момент, когда данные будут доступны. Начнем рассмотрение с использования низкоуровневых API транспорта и протокола на примере простого HTTP-клиента.

8.2 *Транспортные механизмы и протоколы*

На верхнем уровне транспортный механизм – это абстракция взаимодействия с произвольным потоком данных. Взаимодействуя с сокетом или любым другим потоком данных, например стандартным вводом, мы имеем дело со знакомым набором операций. Мы читаем данные из источника или записываем в него данные, а по за-

¹ В русскоязычной литературе термин *поток* сильно перегружен. Этим словом переводится и *thread* (поток выполнения), и *stream* (поток данных), и *flow* (поток управления). Во избежание путаницы мы добавляем уточнения для всех терминов, кроме потока выполнения. – *Прим. перев.*

вершении работы закрываем его. Сокет очень хорошо ложится на эту абстракцию транспорта – чтение, запись, закрытие. Короче говоря, транспортный механизм определяет отправку данных источнику и получение данных от источника. Реализация транспортного механизма зависит от типа источника. Нас в основном интересуют классы `ReadTransport`, `WriteTransport` и `Transport`, хотя есть и другие для работы с UDP-подключениями и взаимодействия с дочерними процессами.

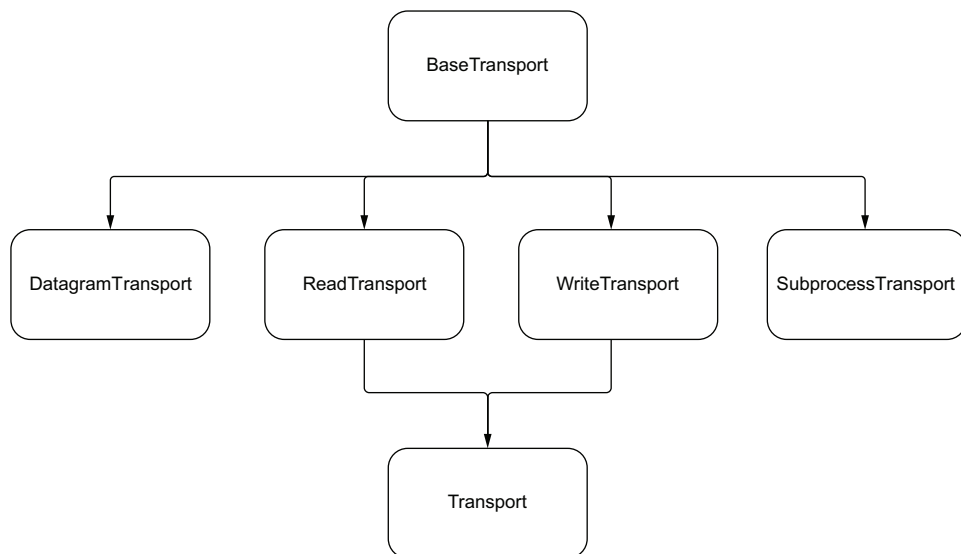


Рис. 8.1 Иерархия транспортных классов

Передача данных в сокеты и из него – только часть проблемы. А как насчет жизненного цикла сокета? Мы устанавливаем подключение, записываем в него данные и обрабатываем полученный ответ. Этот набор операций – принадлежность протокола. Отметим, что здесь под протоколом понимается просто некий класс Python, а не протокол типа HTTP или FTP. Транспортный механизм отвечает за передачу данных и вызывает методы протокола, когда происходят события, например установление подключения или готовность данных для обработки (рис. 8.2).

Чтобы понять, как транспортные механизмы и протоколы работают совместно, напишем простое приложение, которое будет отправлять один GET-запрос HTTP. Прежде всего следует определить класс, расширяющий `asyncio.Protocol`. Мы реализуем несколько методов базового класса: отправку запроса, полученных данных в ответ и обработку ошибок подключения.

Первым будет реализован метод `connection_made`. Транспорт вызывает этот метод, когда сокет успешно подключен к HTTP-серверу. Этот метод принимает в качестве аргумента экземпляр `Transport`, ко-

торый можно использовать для взаимодействия с сервером. В данном случае мы воспользуемся транспортом, чтобы передать HTTP-запрос.

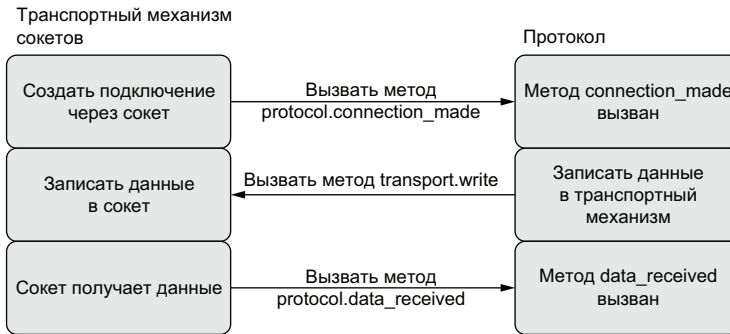


Рис. 8.2 Транспортный механизм вызывает методы протокола при возникновении событий. Протокол может записывать данные в транспортный механизм

Второй метод – `data_received`. Транспортный механизм вызывает его, когда приходят данные, и передает эти данные в виде массива байтов. Метод может быть вызван несколько раз, поэтому необходимо завести внутренний буфер для хранения данных. Возникает вопрос – как узнать, что ответ получен полностью? Для этого нужно реализовать метод `eof_received`. Он вызывается, когда получен *конец файла*; в случае сокета такое бывает, когда сервер закрывает подключение. Гарантируется, что, коль скоро этот метод был вызван, метод `data_received` уже никогда не будет вызван. Метод `eof_received` возвращает значение типа `Boolean`, определяющее, как останавливать транспорт (в данном примере закрыть клиентский сокет). Значение `False` гарантирует, что транспорт остановит себя сам, а значение `True` – что эта обязанность лежит на нашей реализации протокола. В данном случае нам не нужна специальная логика остановки, поэтому метод должен вернуть `False`, и мы не будем обрабатывать закрытие транспорта самостоятельно.

Осталось только решить, как хранить данные во внутреннем буфере. Каким же образом потребители нашего протокола получают результат после завершения запроса? Для этого мы создадим объект `Future`, где будет храниться полностью полученный результат. Затем в конце метода `eof_received` сделаем результат будущего объекта результатом HTTP-ответа. И определим сопрограмму `get_response`, которая будет ждать будущий объект с помощью `await`.

Теперь реализуем все вышеописанное в виде класса протокола. Назовем его `HTTPGetClientProtocol`.

Листинг 8.1 Выполнение HTTP-запроса с помощью транспортного механизма и протокола

```
import asyncio
from asyncio import Transport, Future, AbstractEventLoop
```

```

from typing import Optional

class HTTPGetClientProtocol(asyncio.Protocol):

    def __init__(self, host: str, loop: AbstractEventLoop):
        self._host: str = host
        self._future: Future = loop.create_future()
        self._transport: Optional[Transport] = None
        self._response_buffer: bytes = b''

    async def get_response(self):
        return await self._future

    def _get_request_bytes(self) -> bytes:
        request = f"GET / HTTP/1.1\r\n" \
            f"Connection: close\r\n" \
            f"Host: {self._host}\r\n\r\n"
        return request.encode()

    def connection_made(self, transport: Transport):
        print(f'Создано подключение к {self._host}')
        self._transport = transport
        self._transport.write(self._get_request_bytes())

    def data_received(self, data):
        print(f'Получены данные!')
        self._response_buffer = self._response_buffer + data

    def eof_received(self) -> Optional[bool]:
        self._future.set_result(self._response_buffer.decode())
        return False

    def connection_lost(self, exc: Optional[Exception]) -> None:
        if exc is None:
            print('Подключение закрыто без ошибок.')
        else:
            self._future.set_exception(exc)

```

Получив данные, сохранить их во внутреннем буфере

Ждать внутренний будущий объект, пока не будет получен ответ от сервера

Создать РЕЕ-запрос

После того как подключение установлено, использовать транспорт для отправки запроса

После закрытия подключения завершить будущий объект, скопировав в него данные из буфера

Если подключение было закрыто без ошибок, не делать ничего; иначе завершить будущий объект исключением

Итак, протокол реализован, воспользуемся им для отправки запроса. Для этого нам понадобится метод-сопрограмма `create_connection` цикла событий `asyncio`. Он создает подключение через сокет к указанному серверу и обортывает его подходящим транспортным механизмом. Помимо адреса сервера и номера порта, он принимает *фабрику протоколов*, т. е. функцию, которая создает экземпляры протоколов, – в нашем случае экземпляр только что созданного нами класса `HTTPGetClientProtocol`. Эта сопрограмма возвращает созданный ей транспорт и экземпляр протокола, созданный фабрикой.

Листинг 8.2 Использование протокола

```

import asyncio
from asyncio import AbstractEventLoop
from chapter_08.listing_8_1 import HTTPGetClientProtocol

```

```

async def make_request(host: str, port: int, loop: AbstractEventLoop) -> str:
    def protocol_factory():
        return HTTPGetClientProtocol(host, loop)

    _, protocol = await loop.create_connection(protocol_factory, host=host,
                                              port=port)

    return await protocol.get_response()

async def main():
    loop = asyncio.get_running_loop()
    result = await make_request('www.example.com', 80, loop)
    print(result)

asyncio.run(main())

```

Сначала определим метод `make_request`, который принимает адрес сервера и номер порта, куда мы хотим отправить запрос, и ждет ответа сервера. Внутри мы создаем внутренний метод, определяющий нашу фабрику протоколов, которая создает новый объект `HTTPGetClientProtocol`. Затем вызываем `create_connection`, передавая адрес сервера и номер порта, и получаем в ответ транспорт и протокол, созданный фабрикой. Транспорт нам ни к чему, поэтому игнорируем его, а протокол нужен, потому что мы собираемся использовать сопрограмму `get_response`; таким образом, мы сохраняем протокол в переменной `protocol`. Наконец, мы ждем сопрограмму `get_response` нашего протокола, которая ждет результата от HTTP-сервера. В сопрограмме `main` мы ждем `make_request` и печатаем результат. При выполнении этой программы мы увидим такой HTTP-ответ (для краткости тело HTML опущено):

```

Создано подключение к www.example.com
Получены данные!
HTTP/1.1 200 OK
Age: 193241
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Подключение закрыто без ошибок.

```

Мы научились работать с транспортными механизмами и протоколами. Это низкоуровневые API, поэтому так работать с потоками данных в `asyncio` не рекомендуется. Посмотрим, как использовать *потоки данных*, абстракцию более высокого уровня, надстроенную над транспортными механизмами и протоколами.

8.3 *Потоковые читатели и писатели*

Транспортные механизмы и протоколы – это низкоуровневые API, которые лучше использовать, когда нужен прямой контроль над тем, что происходит при отправке и получении данных. Например, ими

можно воспользоваться, если мы проектируем сетевую библиотеку или веб-каркас. Но в большинстве приложений такой уровень контроля не нужен, а использование транспортных механизмов и протоколов привело бы к написанию повторяющегося кода.

Проектировщики `asuncio` понимали это и создали высокоуровневый API *потоков данных*. Он инкапсулирует стандартные сценарии использования транспорта и протоколов в двух классах, которые проще понять и использовать: `StreamReader` и `StreamWriter`. Как легко догадаться, они отвечают за чтение и запись потоков данных соответственно. Именно их рекомендуется применять для разработки сетевых приложений на основе `asuncio`.

Для демонстрации транслируем пример отправки GET-запроса на язык потоков. Вместо непосредственного создания экземпляров `StreamReader` и `StreamWriter` `asuncio` предоставляет библиотечную сопрограмму `open_connection`, которая делает это за нас. Она принимает сервер и порт, к которым нужно подключиться, и возвращает кортеж, содержащий экземпляры `StreamReader` и `StreamWriter`. Наш план – использовать `StreamWriter` для отправки HTTP-запроса и `StreamReader` для чтения ответа. Методы `StreamReader` легко понять, один из них – сопрограмма `getline`, которая ждет получения полной строки данных. Или же можно было бы использовать сопрограмму `read`, которая ждет прихода заданного числа байтов.

Класс `StreamWriter` чуть сложнее. В нем, естественно, есть метод `write`, но это обычная функция, а не сопрограмма. Потоковый писатель старается сразу же записать в выходной буфер сокета, но этот буфер может быть заполнен. В таком случае данные сохраняются во внутренней очереди, откуда позднее копируются в буфер. И таким образом, возникает потенциальная проблема – вызов `write` необязательно отсылает данные немедленно. Это, в свою очередь, может вызвать переполнение памяти. Представьте, что сетевое соединение медленное и может передавать только 1 Кб в секунду, а наше приложение записывает данные со скоростью 1 Мб/с. Тогда буфер записи в приложении будет заполняться гораздо быстрее, чем опустошается выходной буфер сокета, и в конечном итоге мы потратим всю имеющуюся память, что неминуемо приведет к краху.

Как дождаться момента, когда все данные будут отправлены? Для этого имеется метод `drain`. Эта сопрограмма блокирует выполнение, пока все находящиеся в очереди данные не будут отправлены в сокет. И последовательность действий такова: после каждого вызова `write` использовать `await drain`. Строго говоря, вызывать `drain` после каждого `write` необязательно, но лучше это делать во избежание ошибок.

Листинг 8.3 Отправка HTTP-запроса с помощью потоковых писателей и читателей

```
import asuncio
from asuncio import StreamReader
from typing import AsyncGenerator
```

```

async def read_until_empty(stream_reader: StreamReader) ->
    AsyncGenerator[str, None]:
    while response := await stream_reader.readline():
        yield response.decode()
    async def main():
        host: str = 'www.example.com'
        request: str = f"GET / HTTP/1.1\r\n" \
            f"Connection: close\r\n" \
            f"Host: {host}\r\n\r\n"
        stream_reader, stream_writer = await
            asyncio.open_connection('www.example.com', 80)

        try:
            stream_writer.write(request.encode())
            await stream_writer.drain()

            responses = [response async for response in
                read_until_empty(stream_reader)]

            print(''.join(responses))
        finally:
            stream_writer.close()
            await stream_writer.wait_closed()

    asyncio.run(main())

```

Читать и декодировать строку,
пока не кончатся символы

Записать http-запрос
и опустошить буфер писателя

Читать строки
и сохранять их в списке

Закрывать писатель и ждать
завершения закрытия

Здесь мы сначала создаем асинхронный генератор, который читает все строки, возвращаемые `StreamReader`, и декодирует их, пока не кончатся символы. Затем в сопропрограмме `main` мы открываем подключение к `example.com` и попутно создаем экземпляры `StreamReader` и `StreamWriter`. После этого записываем запрос и опустошаем буфер писателя, используя методы `write` и `drain`. Когда запрос будет отправлен, мы с помощью асинхронного генератора получаем все строки ответа и сохраняем их в списке `responses`. И наконец, закрываем экземпляр `StreamWriter`, вызывая его метод `close`, и ждем завершения закрытия в сопропрограмме `wait_closed`. Зачем здесь нужно вызывать метод `и` сопропрограмму? Потому что в ходе вызова `close` выполняется несколько действий, в частности отмена регистрации сокета и вызов метода `connection_lost` нижележащего транспортного механизма. Все это происходит асинхронно на более поздней итерации цикла событий, а значит, сразу после вызова `close` наше подключение еще не закрыто, это случится немного позже. Если необходимо дождаться закрытия подключения перед дальнейшей обработкой или если важно знать об исключениях, случившихся во время закрытия, то лучше вызывать `wait_closed`.

Итак, мы познакомились с основами API потоков данных на примере отправки веб-запросов. Но полезность этих классов не ограничивается сетевыми приложениями. В следующем разделе мы увидим, как с их помощью создавать неблокирующие приложения командной строки.

8.4 Неблокирующий ввод данных из командной строки

Традиционно в Python для получения данных от пользователя применяется функция `input`. Она приостанавливает выполнение, пока пользователь не введет данные и не нажмет клавишу **Enter**. Но что, если мы хотим выполнять код в фоновом режиме и при этом отзываться на действия пользователя? Например, пользователь мог бы конкурентно запускать несколько длительных задач, скажем SQL-запросов. А в чате с командной строкой в качестве интерфейса мы хотим, чтобы пользователь мог набирать сообщение одновременно с получением сообщений от других участников.

Поскольку библиотека `asyncio` однопоточная, использование `input` означало бы, что мы блокируем цикл событий, а с ним и все приложение на время, пока пользователь вводит данные. Даже запуск задач для выполнения операции в фоновом режиме не спасет. Чтобы убедиться в этом, попробуем написать приложение, в котором пользователь вводит время, в течение которого приложение должно спать. Хотелось бы, чтобы приложение могло выполнять несколько таких операций засыпания конкурентно, но при этом принимать данные от пользователя. Поэтому мы будем в цикле запрашивать количество секунд и создавать задачу `delay`.

Листинг 8.4 Попытка выполнения задач в фоновом режиме

```
import asyncio
from util import delay

async def main():
    while True:
        delay_time = input('Введите время сна:')
        asyncio.create_task(delay(int(delay_time)))

asyncio.run(main())
```

Если бы этот код работал так, как мы хотели, то после ввода числа мы увидели бы сообщение «засыпаю на n с», а спустя n секунд сообщение «сон в течение n с закончился». Однако это не так – мы не видим ничего, кроме приглашения ввести время сна. Все дело в том, что в коде нет предложения `await`, поэтому наша задача не имеет возможности войти в цикл событий. Можно было бы схитрить и написать `await asyncio.sleep(0)` после строки `create_task`, запланировав тем самым задачу (этот прием, называемый «уступкой управления циклу событий», мы рассмотрим в главе 14). Но это не помогает, потому что вызов `input` блокирует весь поток и не дает фоновым задачам доработать до конца.

На самом деле нам нужно, чтобы функция `input` была сопрограммой, чтобы можно было написать нечто вроде `delay_time = await`

`input('Введите время сна: ')`). Тогда наша задача была бы корректно запланирована и продолжала бы работать, пока мы ждем ввода данных пользователем. К сожалению, сопрограммного варианта `input` не существует, нужно придумать что-то другое.

На помощь приходят протоколы и потоковые читатели. Напомним, что у потокового читателя есть метод-сопрограмма `readline`, а это как раз то, что мы ищем. Если бы мы смогли подключить потокового читателя к стандартному вводу, то получилось бы использовать сопрограмму для ввода данных.

У цикла событий `asyncio` имеется метод-сопрограмма `connect_read_pipe`, который подключает протокол к файлоподобному объекту, а это почти то, что нам нужно. Метод принимает *фабрику протоколов* и *канал*. Фабрика протоколов – это просто функция, создающая экземпляры протоколов, а канал – файлоподобный объект, имеющий методы `read` и `write`. Сопрограмма `connect_read_pipe` подключает канал к протоколу, созданному фабрикой, так что данные из канала передаются протоколу.

Стандартный консольный вход `sys.stdin` удовлетворяет требованиям, предъявляемым к файлоподобному объекту, поэтому мы можем передать его `connect_read_pipe`. Вызвав эту сопрограмму, мы получим кортеж, состоящий из протокола, созданного фабрикой, и транспорта `ReadTransport`. Вопрос: какой протокол должна создать фабрика и как соединить его с экземпляром `StreamReader`, у которого есть интересующая нас сопрограмма `readline`?

`Asyncio` предоставляет служебный класс `StreamReaderProtocol` для соединения потоковых читателей с протоколами. Конструктору этого класса передается экземпляр потокового читателя. Затем класс протокола делегирует работу созданному нами потоковому читателю, позволяя использовать его для чтения данных из стандартного ввода. Собирая все вместе, мы можем создать приложение командной строки, которое не блокирует цикл событий, ожидая, пока пользователь введет данные.

Для пользователей Windows

К сожалению, в Windows сопрограмма `connect_read_pipe` не работает с `sys.stdin`. Это связано с неисправленной ошибкой в реализации файловых дескрипторов в Windows. Чтобы описанный подход заработал в Windows, нужно вызывать `sys.stdin.readline()` в отдельном потоке, как было описано в главе 7. Дополнительные сведения имеются на странице <https://bugs.python.org/issue26832>.

Поскольку мы будем не раз использовать асинхронный читатель стандартного ввода в этой главе, оформим его в виде отдельного файла `listing_8_5.py`, который можно будет импортировать.

Листинг 8.5 Асинхронный читатель стандартного ввода

```
import asyncio
from asyncio import StreamReader
import sys

async def create_stdin_reader() -> StreamReader:
    stream_reader = asyncio.StreamReader()
    protocol = asyncio.StreamReaderProtocol(stream_reader)
    loop = asyncio.get_running_loop()
    await loop.connect_read_pipe(lambda: protocol, sys.stdin)
    return stream_reader
```

Повторно используемая сопрограмма `create_stdin_reader` создает объект `StreamReader`, который мы будем использовать для асинхронного чтения стандартного ввода. Сначала создается экземпляр потокового читателя и передается протоколу потокового читателя. Затем вызывается сопрограмма `connect_read_pipe`, ей передается фабрика протоколов, реализованная в виде лямбда-функции, которая возвращает созданный ранее протокол. Также ей передается `sys.stdin`, который соединяется с нашим протоколом потокового читателя. Поскольку транспорт и протокол, возвращаемые `connect_read_pipe`, нам не нужны, мы их игнорируем. Теперь эту функцию можно использовать для асинхронного чтения из стандартного ввода и наконец-то закончить наше приложение.

Листинг 8.6 Использование потоковых читателей для ввода данных

```
import asyncio
from chapter_08.listing_8_5 import create_stdin_reader
from util import delay

async def main():
    stdin_reader = await create_stdin_reader()
    while True:
        delay_time = await stdin_reader.readline()
        asyncio.create_task(delay(int(delay_time)))

asyncio.run(main())
```

В сопрограмме `main` мы вызываем `create_stdin_reader` и входим в бесконечный цикл, где ожидаем данных от пользователя в сопрограмме `readline`. Как только пользователь нажмет клавишу **Enter**, эта сопрограмма доставит введенный текст. Получив данные от пользователя, мы преобразуем их в целое число (заметим, что в реальном приложении нужно было бы обработать некорректный ввод, – сейчас это привело бы к краху) и создаем задачу `delay`. Теперь программа позволяет конкурентно выполнять несколько задач `delay`, не прерывая ввода данных пользователем. Например, если ввести задержки 5, 4 и 3, то мы увидим такую картину:

```

5
засыпаю на 5 с
4
засыпаю на 4 с
3
засыпаю на 3 с
сон в течение 5 с закончился
сон в течение 4 с закончился
сон в течение 3 с закончился

```

Работать-то работает, но у этого подхода есть серьезный изъян. Что будет, если сообщение появляется на экране, когда мы вводим время задержки? Чтобы узнать это, введите время задержки 3 с, а затем нажмите и удерживайте клавишу 1. Тогда картина будет такой:

```

3
засыпаю на 3 с
111111сон в течение 3 с закончился
11

```

Мы продолжаем вводить, а тем временем печатается сообщение от задачи `delay`, которое разрывает входную строку. Кроме того, входной буфер теперь содержит только 11, т. е. после нажатия **Enter** будет создана задача `delay` с задержкой 11 с, а первые цифры будут потеряны. Все дело в том, что по умолчанию терминал работает в *режиме с обработкой*, когда введенные пользователем данные копируются на стандартный вывод и обрабатываются специальные клавиши, в том числе **Enter** и **CTRL+C**. А проблема возникает из-за того, что сопрограмма `delay` пишет на стандартный вывод одновременно с эхо-копированием ввода, что приводит к состоянию гонки.

Кроме того, с продолжением стандартного вывода связана единственная позиция на экране. Она называется *курсором* и очень похожа на курсор в текстовом процессоре. Пока мы вводим данные, курсор находится в строке, в которой мы печатаем. А значит, все выходные сообщения от сопрограмм будут появляться в той же строке, что и вводимые нами данные, поскольку именно там находится курсор. Итог – нежелательное поведение.

Для решения этой проблемы нужно сделать две вещи. Во-первых, передать эхо-копирование ввода от терминала нашему приложению. Тем самым гарантируется, что во время эхо-копирования не будут выводиться сообщения от других сопрограмм, потому что программа однопоточная. Во-вторых, на время вывода сообщений нужно переместить курсор в другое место экрана, чтобы они не печатались в той же строке, где вводятся данные. То и другое делается с помощью изменения параметров терминала и использования управляющих последовательностей.

8.4.1 Режим терминала без обработки и сопрограмма *read*

Поскольку наш терминал функционирует в режиме с обработкой, он сам копирует введенные пользователем данные на стандартный вывод. Как можно перенести эту обработку в приложение и тем самым избежать описанного выше состояния гонки?

Нужно перевести терминал в режим *без обработки*. В этом режиме терминал не занимается буферизацией данных, их предобработкой и эхо-копированием, а просто передает каждое нажатие клавиши приложению. И мы сможем организовать копирование и предобработку, как пожелаем. Конечно, работы больше, но зато мы получаем точный контроль над записью на стандартный вывод, что позволит избежать состояния гонки.

Python позволяет перевести терминал не только в режим без обработки, но и в режим `cbreak`. Он отличается от режима без обработки тем, что нажатие клавиши **CTRL+C** все же интерпретируется, что избавляет нас от части работы. Войти в этот режим можно с помощью функции `setcbreak` из модуля `tty`:

```
import tty
import sys
tty.setcbreak(sys.stdin)
```

Теперь надо переосмыслить дизайн приложения. Сопрограмма `getline` больше работать не будет, потому что в режиме без обработки вводимые данные не копируются на стандартный вывод. Мы должны читать по одному символу и сохранять его в собственном внутреннем буфере, одновременно выполняя эхо-копирование. У потокового читателя стандартного ввода, созданного ранее, имеется метод `read`, который читает указанное число байтов из потока. Вызывая `read(1)`, мы будем читать по одному символу, который затем сохраним в буфере и скопируем на стандартный вывод.

Два кусочка головоломки у нас уже есть: вход в режим `cbreak` и чтение по одному символу с эхо-копированием. Теперь надо придумать, как отображать вывод сопрограмм `delay`, чтобы он не перемешивался с тем, что мы вводим.

Сформулируем требования к приложению, которые позволят решить проблему ввода и вывода в одной строке. И, руководствуясь ими, построим реализацию.

- 1 Поле ввода должно всегда находиться в последней строке экрана.
- 2 Сообщения сопрограммы должны начинаться с верхней строки.
- 3 Когда на экране не останется свободных строк для сообщений, уже выведенные сообщения должны прокручиваться вверх.

Как организовать вывод сообщений из сопрограммы `delay`, не нарушая этих требований? Поскольку мы хотим прокручивать сообщения вверх, когда сообщений больше, чем доступных строк, запись прямо на экран с помощью `print` не годится. Вместо этого мы будем хранить двустороннюю очередь (деку) сообщений, предназначенных для записи на стандартный вывод. Максимальное число элементов в деке сделаем равным числу строк на экране терминала. Это позволит реализовать желаемое поведение при заполнении очереди, поскольку элементы, оказавшиеся в конце, будут отбрасываться. Когда в очередь помещается новое сообщение, мы перерисовываем все сообщения, начиная с верхней строчки экрана. Так мы реализуем прокрутку, не храня информацию о состоянии стандартного вывода. Поток управления в приложении будет выглядеть, как показано на рис. 8.3.

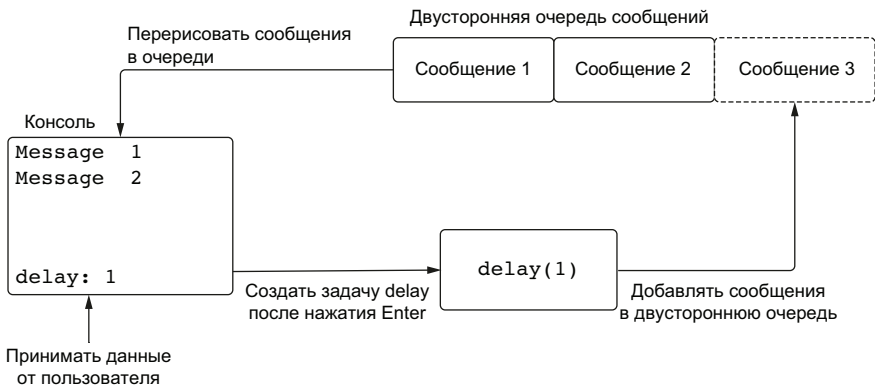


Рис. 8.3 Приложение вывода на консоль с задержкой

Наш план таков:

- 1) переместить курсор на последнюю строку экрана и, когда пользователь нажимает клавишу, добавлять ее во внутренний буфер и эхо-копировать на стандартный вывод;
- 2) когда пользователь нажимает **Enter**, создать задачу `delay`. Но не записывать выходные сообщения на стандартный вывод, а помещать их в очередь, максимальное число элементов в которой равно числу строк на экране;
- 3) после добавления сообщения в очередь перерисовать экран. Для этого сначала переместить курсор в левую верхнюю позицию экрана. Затем вывести все находящиеся в очереди сообщения. И после этого вернуть курсор в ту позицию, где он был до этого.

Для реализации этого плана нужно сначала научиться перемещать курсор по экрану. Это делается с помощью *управляющих кодов ANSI*, которые записываются на стандартный вывод и позволяют выполнять такие действия, как изменение цвета текста, перемещение курсора вверх или вниз или очистка строк. Первым символом управляющей последовательности должен быть управляющий код; в Python

для этого нужно вывести на консоль `\033`. Многие управляющие последовательности начинаются *инициатором* – парой символов `\033[`. Например, вот как переместить курсор на пять строк вниз:

```
sys.stdout.write('\033[5E')
```

В этой управляющей последовательности за инициатором следуют символы `5E`. Число `5` – это количество строк, а буква `E` – код, означающий «переместить курсор на указанное число строк вниз». Управляющие последовательности читать трудно. В листинге ниже мы написали несколько функций, названия которых показывают, что делает управляющий код. В следующих листингах мы будем их импортировать. Подробное описание управляющих последовательностей ANSI см. в статье «Википедии» по адресу https://en.wikipedia.org/wiki/ANSI_escape_code.

Подумаем, как нужно перемещать курсор по экрану, чтобы реализовать намеренное. Во-первых, нам понадобится перемещать курсор в нижнюю строку, где пользователь вводит данные. Далее, после нажатия **Enter** нужно будет стереть введенный текст. Сопрограмма выводит сообщения, начиная с верхней строки экрана, поэтому понадобится перемещать туда курсор. Кроме того, нужно будет сохранять и восстанавливать текущую позицию курсора, поскольку после печати сообщения сопрограммой мы должны будем вернуть курсор туда, где он был. Для всего этого используем следующие функции.

Листинг 8.7 Вспомогательные функции для вывода управляющих последовательностей

```
import sys
import shutil

def save_cursor_position():
    sys.stdout.write('\0337')

def restore_cursor_position():
    sys.stdout.write('\0338')

def move_to_top_of_screen():
    sys.stdout.write('\033[H')

def delete_line():
    sys.stdout.write('\033[2K')

def clear_line():
    sys.stdout.write('\033[2K\033[0G')

def move_back_one_char():
    sys.stdout.write('\033[1D')

def move_to_bottom_of_screen() -> int:
    _, total_rows = shutil.get_terminal_size()
```

```

input_row = total_rows - 1
sys.stdout.write(f'\033[{input_row}E')
return total_rows

```

Имея функции для перемещения курсора по экрану, реализуем допускающую повторное использование сопрограмму для чтения одного символа из стандартного ввода. Для этого воспользуемся сопрограммой `read`. Прочитав символ, мы записываем его на стандартный вывод и сохраняем во внутреннем буфере. Мы также хотим обрабатывать нажатие клавиши **Delete**, поэтому следим за ее появлением. После нажатия **Delete** мы удаляем символ из буфера и из стандартного вывода.

Листинг 8.8 Чтение из стандартного ввода по одному символу

```

import sys
from asyncio import StreamReader
from collections import deque
from chapter_08.listing_8_7 import move_back_one_char, clear_line

async def read_line(stdin_reader: StreamReader) -> str:
    def erase_last_char():
        move_back_one_char()
        sys.stdout.write(' ')
        move_back_one_char()
        # ← Функция для удаления предыдущего символа из стандартного вывода

    delete_char = b'\x7f'
    input_buffer = deque()
    while (input_char := await stdin_reader.read(1)) != b'\n':
        if input_char == delete_char:
            if len(input_buffer) > 0:
                input_buffer.pop()
                erase_last_char()
                sys.stdout.flush()
            # ← Если введен символ загла, то удалить предыдущий символ
        else:
            input_buffer.append(input_char)
            sys.stdout.write(input_char.decode())
            sys.stdout.flush()
            # ← Все символы, кроме загла, добавляются в конец буфера и эхо-копируются
    clear_line()
    return b''.join(input_buffer).decode()

```

Наша сопрограмма принимает потоковый читатель, соединенный со стандартным вводом. Затем мы определяем вспомогательную функцию, которая удаляет предыдущий символ из стандартного вывода, поскольку это нужно будет делать при нажатии **Delete**. Далее мы входим в цикл `while`, где читаем символы, пока пользователь не нажмет **Enter**. Если была нажата клавиша **Delete**, то мы удаляем последний символ из буфера и из стандартного вывода. В противном случае символ добавляется в буфер и эхо-копируется. После нажатия **Enter** мы очищаем строку ввода и возвращаем содержимое буфера.

Следующий наш шаг – реализовать очередь, в которой будут храниться сообщения для записи на стандартный вывод. Поскольку мы

хотим перерисовывать экран после добавления каждого сообщения, определим класс, который обертывает двустороннюю очередь и принимает допускающий ожидание объект обратного вызова. Этот объект будет отвечать за перерисовку экрана. Также добавим в класс метод-сопрограмму `append`, который будет добавлять элементы в конец двусторонней очереди и вызывать обратный вызов, передавая ему текущее содержимое очереди.

Листинг 8.9 Хранилище сообщений

```
from collections import deque
from typing import Callable, Deque, Awaitable

class MessageStore:
    def __init__(self, callback: Callable[[Deque], Awaitable[None]],
                 max_size: int):
        self._deque = deque(maxlen=max_size)
        self._callback = callback

    async def append(self, item):
        self._deque.append(item)
        await self._callback(self._deque)
```

Вот теперь у нас есть все необходимое для создания приложения. Перепишем сопрограмму `delay`, так чтобы она добавляла сообщения в хранилище. В сопрограмме `main` создадим вспомогательную сопрограмму, которая будет перерисовывать экран из очереди. Эту сопрограмму будем передавать нашему классу `MessageStore` в качестве обратного вызова. Затем воспользуемся написанной ранее сопрограммой `read_line`, которая принимает данные от пользователя и создает задачу `delay` при нажатии **Enter**.

Листинг 8.10 Приложение для асинхронной задержки

```
import asyncio
import os
import tty
from collections import deque
from chapter_08.listing_8_5 import create_stdin_reader
from chapter_08.listing_8_7 import *
from chapter_08.listing_8_8 import read_line
from chapter_08.listing_8_9 import MessageStore

async def sleep(delay: int, message_store: MessageStore):
    await message_store.append(f'Начало задержки {delay}')
    await asyncio.sleep(delay)
    await message_store.append(f'Конец задержки {delay}')

async def main():
    tty.setcbreak(sys.stdin)
    os.system('clear')
```

Добавить выходное сообщение в хранилище

```

rows = move_to_bottom_of_screen()

async def redraw_output(items: deque):
    save_cursor_position()
    move_to_top_of_screen()
    for item in items:
        delete_line()
        print(item)
    restore_cursor_position()

messages = MessageStore(redraw_output, rows - 1)

stdin_reader = await create_stdin_reader()

while True:
    line = await read_line(stdin_reader)
    delay_time = int(line)
    asyncio.create_task(sleep(delay_time, messages))

asyncio.run(main())

```

← Обратный вызов, который перемещает курсор в начало экрана, перерисовывает экран и возвращает курсор обратно

Теперь мы сможем создавать задержки и видеть, как сообщения выводятся на консоль, даже во время ввода данных. Это, конечно, сложнее, чем первая попытка, зато нет прежних проблем с записью на стандартный вывод.

Построенное приложение работает для сопрограммы `delay`, но как насчет чего-то более реалистичного? Написанные нами фрагменты достаточно надежны, так что их можно использовать и для создания более полезных приложений. Например, почему бы не задуматься о командном SQL-клиенте? Некоторые запросы выполняются долго, и мы хотели бы в это время запускать другие запросы или отменить уже работающий. Из разработанных компонентов можно создать клиент такого типа. Сделаем это для базы данных о товарах из главы 5, где хранятся торговые марки производителей одежды, товары и SKU. Для подключения к базе данных создадим пул подключений и повторно используем код из предыдущих примеров для ввода и выполнения запросов. Будем выводить основную информацию о запросах на консоль – пока только количество возвращенных строк.

Листинг 8.11 Асинхронный командный SQL-клиент

```

import asyncio
import asyncpg
import os
import tty
from collections import deque
from asyncpg.pool import Pool
from chapter_08.listing_8_5 import create_stdin_reader
from chapter_08.listing_8_7 import *
from chapter_08.listing_8_8 import read_line

```

```

from chapter_08.listing_8_9 import MessageStore

async def run_query(query: str, pool: Pool, message_store: MessageStore):
    async with pool.acquire() as connection:
        try:
            result = await connection.fetchrow(query)
            await message_store.append(f'Выбрано {len(result)} строк по запросу: {query}')
        except Exception as e:
            await message_store.append(f'Получено исключение {e} от: {query}')

async def main():
    tty.setcbreak(0)
    os.system('clear')
    rows = move_to_bottom_of_screen()

    async def redraw_output(items: deque):
        save_cursor_position()
        move_to_top_of_screen()
        for item in items:
            delete_line()
            print(item)
        restore_cursor_position()

    messages = MessageStore(redraw_output, rows - 1)

    stdin_reader = await create_stdin_reader()

    async with asyncpg.create_pool(host='127.0.0.1',
                                    port=5432,
                                    user='postgres',
                                    password='password',
                                    database='products',
                                    min_size=6,
                                    max_size=6) as pool:

        while True:
            query = await read_line(stdin_reader)
            asyncio.create_task(run_query(query, pool, messages))

    asyncio.run(main())

```

Код почти не изменился, только вместо сопрограммы `delay` мы создали сопрограмму `run_query`. Она не спит заданное время, а выполняет введенный пользователем запрос, который может занимать произвольное время. Это позволяет запускать новые запросы из командной строки, не дожидаясь завершения уже запущенных, и видеть результаты завершившихся запросов, не прерывая ввода новых.

Теперь мы знаем, как создавать командные клиенты, поддерживающие ввод данных в то время, как другой код работает и выводит что-то на консоль. Следующая наша цель – научиться писать серверы с помощью высокоуровневых API `asyncio`.

8.5 Создание серверов

При построении серверов, в частности эхо-сервера, мы создавали серверный сокет, привязывали его к порту и ждали запросов на подключение. Это работает, но `asyncio` позволяет создавать серверы на более высоком уровне абстракции, даже не думая об управлении сокетами. Это упрощает код, поэтому применение высокоуровневых API – рекомендуемый способ создания и управления серверами с помощью `asyncio`.

Для создания сервера служит сопрограмма `asyncio.start_server`. Она принимает несколько факультативных параметров, например для настройки SSL, но нас прежде всего интересуют параметры `host`, `port` и `client_connected_cb`. Первые два мы уже встречали: они задают адрес, который прослушивает сервер. Более интересен параметр `client_connected_cb`; это функция или сопрограмма обратного вызова, которая вызывается, когда к серверу подключился клиент. Она принимает объекты `StreamReader` и `StreamWriter`, позволяющие читать данные от клиента и записывать для него ответные данные.

При выполнении предложения `await start_server` мы получаем объект `AbstractServer`. Этому классу недостает многих нужных нам методов, но зато в нем есть метод `serve_forever`, который исполняет код сервера, пока мы не остановим его. Он также является контекстным менеджером, т. е. может употребляться в конструкции `async with`, гарантирующей корректный останов сервера при завершении программы.

Для демонстрации перепишем наш эхо-сервер и заодно немного расширим его функциональность. Он будет не только копировать вход в выход, но и отображать информацию о количестве подключившихся клиентов. Кроме того, мы будем выводить сообщение, когда клиент отключается от сервера. Для этого создадим класс `ServerState`, следящий за количеством подключенных клиентов. При подключении нового пользователя мы учтем его в состоянии сервера и уведомим о его появлении других клиентов.

Листинг 8.12 Создание эхо-сервера с помощью серверных объектов

```
import asyncio
import logging
from asyncio import StreamReader, StreamWriter

class ServerState:

    def __init__(self):
        self._writers = []
        Добавить клиента в состояние сервера  
и создать задачу эхо-копирования

    async def add_client(self, reader: StreamReader, writer: StreamWriter):
        self._writers.append(writer)
        await self._on_connect(writer)
        asyncio.create_task(self._echo(reader, writer))
```

```

        После подключения нового клиента сообщить ему, сколько клиентов
        подключено, и уведомить остальных о новом пользователе
    async def _on_connect(self, writer: StreamWriter):
        writer.write(f'Добро пожаловать! Число подключенных пользователей:
        {len(self._writers)}!\n'.encode()) =
        await writer.drain()
        await self._notify_all('Подключился новый пользователь!\n')

    async def _echo(self, reader: StreamReader, writer: StreamWriter):
        Обработать
        try:
        эхо-копирование ввода while (data := await reader.readline()) != b'':
        при отключении клиента writer.write(data)
        и уведомить остальных await writer.drain()
        пользователей об self._writers.remove(writer)
        отключении await self._notify_all(f'Клиент отключился. Осталось пользователей:
        {len(self._writers)}!\n')
        except Exception as e:
            logging.exception('Ошибка чтения данных от клиента.', exc_info=e)
            self._writers.remove(writer)

    async def _notify_all(self, message: str):
        for writer in self._writers:
            try:
                writer.write(message.encode())
                await writer.drain()
            except ConnectionError as e:
                logging.exception('Ошибка записи данных клиенту.', exc_info=e)
                self._writers.remove(writer)
        Вспомогательный метод для
        отправки сообщения всем
        остальным пользователям. Если
        отправить сообщение не удалось,
        удалить данного пользователя

    async def main():
        server_state = ServerState()
        При подключении нового клиента
        добавить его в состояние сервера

    async def client_connected(reader: StreamReader, writer:
    StreamWriter) -> None:
        await server_state.add_client(reader, writer)

    server = await asyncio.start_server(client_connected, '127.0.0.1', 8000)

    async with server:
        Запустить сервер и обслуживать
        запросы бесконечно
        await server.serve_forever()

    asyncio.run(main())

```

Когда пользователь подключается к серверу, вызывается сопрограмма `client_connected`, которой передаются читатель и писатель для этого пользователя, а та, в свою очередь, вызывает метод-сопрограмму состояния сервера `add_client`. В `add_client` мы сохраняем объект `StreamWriter`, чтобы можно было отправлять сообщения всем подключенным клиентам, а при отключении клиента удаляем этот объект. Далее вызывается метод `_on_connect`, который отправляет сообщение клиенту, информируя его о количестве подключенных пользователей. В `_on_connect` мы также уведомляем всех остальных клиентов о подключении нового пользователя.

Сопрограмма `_echo` похожа на то, что мы делали раньше, только теперь мы еще уведомляем всех пользователей о том, что какой-то клиент отключился. В результате мы получили эхо-сервер, уведомляющий всех своих клиентов о подключении и отключении пользователей.

Мы видели, как создать асинхронный сервер с расширенной функциональностью. На основе этих знаний далее разработаем чат-сервер и его клиента, т. е. еще более продвинутую программу.

8.6 *Создание чат-сервера и его клиента*

Мы уже умеем создавать серверы и асинхронно обрабатывать ввод данных в командной строке. Объединив то и другое, мы разработаем два новых приложения. Первое – чат-сервер, который может обслуживать несколько клиентов одновременно, второе – клиент, который подключается к этому серверу и отправляет и принимает сообщения чата.

Прежде чем приступить к проектированию приложения, сформулируем требования, которые помогут принять правильные проектные решения. Сначала для сервера.

- 1 Клиент чата должен иметь возможность подключиться к серверу, указав имя пользователя.
- 2 Подключившийся пользователь должен иметь возможность отправлять в чат сообщения, каждое из которых будет разослано всем остальным клиентам сервера.
- 3 Чтобы молчуны не занимали ресурсы, пользователь, не проявляющий активности дольше одной минуты, отключается.

Затем для клиента.

- 1 После запуска клиент должен запросить у пользователя имя и попытаться подключиться к серверу.
- 2 Подключившийся пользователь будет видеть, как сообщения от других пользователей прокручиваются сверху вниз.
- 3 Поле ввода должно располагаться в нижней строке экрана. После нажатия клавиши **Enter** текст в поле ввода должен быть отправлен серверу, а затем всем подключившимся клиентам.

Теперь подумаем, как должно быть организовано взаимодействие между клиентом и сервером. Прежде всего мы должны посылать сообщение с именем пользователя от клиента серверу. Надо отличать отправку имени пользователя от отправки сообщения, поэтому определим простой командный протокол, показывающий, что посылается имя пользователя. Не будем усложнять и просто передадим строку с именем команды `CONNECT`, за которой следует имя пользователя. Например, чтобы сервер подключил пользователя с именем «MissIslington», нужно отправить сообщение «`CONNECT MissIslington`».

После подключения просто отправляем сообщения серверу, который рассылает их всем подключенным клиентам (в том числе и нам;

если хотите, можете оптимизировать программу, убрав копирование отправителю). Для надежности можно было бы ввести команду, которую сервер отправляет клиенту, чтобы подтвердить получение сообщения, но для краткости мы это опустим.

Теперь у нас есть все необходимое для проектирования сервера. Создадим класс `ChatServerState` по аналогии с тем, что было сделано в предыдущем разделе. После подключения клиента мы ждем получения от него команды `CONNECT` с именем пользователя. Дождавшись, создаем задачу для прослушивания сообщений от клиента и рассылки их всем подключенным пользователям. Подключенные клиенты хранятся в словаре, отображающем имена на соответствующие экземпляры `StreamWriter`. Если подключенный клиент не проявляет активности дольше минуты, то мы отключаем его и удаляем запись о нем из словаря; при этом другим пользователям рассылается сообщение о том, что кто-то покинул чат.

Листинг 8.13 Чат-сервер

```
import asyncio
import logging
from asyncio import StreamReader, StreamWriter

class ChatServer:

    def __init__(self):
        self.username_to_writer = {}

    async def start_chat_server(self, host: str, port: int):
        server = await asyncio.start_server(self.client_connected, host, port)

        async with server:
            await server.serve_forever()

    async def client_connected(self, reader: StreamReader,
                               writer: StreamWriter):
        command = await reader.readline()
        print(f'CONNECTED {reader} {writer}')
        command, args = command.split(b' ')
        if command == b'CONNECT':
            username = args.replace(b'\n', b'').decode()
            self._add_user(username, reader, writer)
            await self._on_connect(username, writer)
        else:
            logging.error('Получена недопустимая команда от клиента, отключается.')
            writer.close()
            await writer.wait_closed()

    def _add_user(self, username: str, reader: StreamReader,
                  writer: StreamWriter):
        self.username_to_writer[username] = writer
        asyncio.create_task(self._listen_for_messages(username, reader))
```

Ждать, пока клиент отправит допустимое имя пользователя; в противном случае отключить его

Сохранить экземпляр потокового писателя для данного пользователя и создать задачу, прослушивающую его сообщения

```

async def _on_connect(self, username: str, writer: StreamWriter):
    writer.write(f'Добро пожаловать! Подключено пользователей:
{len(self._username_to_writer)}!\n'.encode())
    await writer.drain()
    await self._notify_all(f'Подключился {username}!\n')

async def _remove_user(self, username: str):
    writer = self._username_to_writer[username]
    del self._username_to_writer[username]
    try:
        writer.close()
        await writer.wait_closed()
    except Exception as e:
        logging.exception('Ошибка при закрытии клиентского писателя, игнорируется.',
                          exc_info=e)

async def _listen_for_messages(self,
                               username: str,
                               reader: StreamReader):
    try:
        while (data := await asyncio.wait_for(reader.readline(), 60)) != b'':
            await self._notify_all(f'{username}: {data.decode()}')
            await self._notify_all(f'{username} has left the chat\n')
    except Exception as e:
        logging.exception('Ошибка при чтении данных от клиента.', exc_info=e)
        await self._remove_user(username)

async def _notify_all(self, message: str):
    inactive_users = []
    for username, writer in self._username_to_writer.items():
        try:
            writer.write(message.encode())
            await writer.drain()
        except ConnectionError as e:
            logging.exception('Ошибка при записи данных клиенту.', exc_info=e)
            inactive_users.append(username)
    [await self._remove_user(username) for username in inactive_users]

async def main():
    chat_server = ChatServer()
    await chat_server.start_chat_server('127.0.0.1', 8000)

asyncio.run(main())

```

После подключения пользователя уведомить об этом всех остальных

Прослушивать сообщения от клиента и рассылать их всем остальным клиентам. Ждать сообщения не более минуты

Отправить сообщение всем подключенным клиентам и удалить отключившихся клиентов

Наш класс `ChatServer` инкапсулирует всю информацию о чат-сервере в одном чистом интерфейсе. Главной точкой входа является сопрограмма `start_chat_server`. Она запускает сервер с заданным адресом и номером порта и вызывает сопрограмму `serve_forever`. В качестве обратного вызова, информирующего о подключении клиента, мы указываем сопрограмму `client_connected`. Эта сопрограмма ждет первой строки данных от клиента, и если получена правильная команда `CONNECT`, то вызывает `_add_user`, а затем `_on_connect`; в противном случае соединение разрывается.

Функция `_add_user` сохраняет имя пользователя и связанный с ним потоковый писатель во внутреннем словаре, после чего создает задачу, которая прослушивает сообщения от этого пользователя. Сопрограмма `_on_connect` отправляет сообщение «Добро пожаловать» клиенту и уведомляет всех подключенных клиентов о новом пользователе.

В функции `_add_user` мы создали задачу для сопрограммы `_listen_for_messages`. Именно в ней сосредоточена основная функциональность приложения. В бесконечном цикле мы читаем сообщения от клиента, пока не увидим пустую строку, означающую, что клиент отключился. Получив сообщение, мы вызываем сопрограмму `_notify_all`, которая рассылает его всем подключенным клиентам. Чтобы удовлетворить требование об отключении клиентов, не проявляющих активности дольше минуты, мы обертываем сопрограмму `readline` сопрограммой `wait_for`. В результате будет возбуждено исключение `TimeoutError`, если клиент молчит. На этот случай мы перехватываем как `TimeoutError`, так и все остальные исключения. При возникновении любого исключения запись о клиенте удаляется из словаря `_user_name_to_writer`, поэтому никакие сообщения ему больше отправляться не будут.

Теперь у нас есть работающий сервер, но зачем нужен сервер, для которого нет клиентов? Реализуем клиент по аналогии с написанным ранее командным SQL-клиентом. Создадим сопрограмму, которая будет прослушивать сообщения от сервера, добавлять их в хранилище сообщений и перерисовывать экран при поступлении каждого нового сообщения. Также поместим поле ввода в последнюю строку экрана и при нажатии **Enter** будем отправлять сообщение чат-серверу.

Листинг 8.14 Клиент чат-сервера

```
import asyncio
import os
import logging
import tty
from asyncio import StreamReader, StreamWriter
from collections import deque
from chapter_08.listing_8_5 import create_stdin_reader
from chapter_08.listing_8_7 import *
from chapter_08.listing_8_8 import read_line
from chapter_08.listing_8_9 import MessageStore
```

```
async def send_message(message: str, writer: StreamWriter):
    writer.write((message + '\n').encode())
    await writer.drain()
```

```
async def listen_for_messages(reader: StreamReader,
                              message_store: MessageStore):
    while (message := await reader.readline()) != b'':
        await message_store.append(message.decode())
```

Прослушивать сообщения от сервера и добавлять их в хранилище сообщений

```

await message_store.append('Сервер закрыл соединение.')

async def read_and_send(stdin_reader: StreamReader,
                        writer: StreamWriter):
    while True:
        message = await read_line(stdin_reader)
        await send_message(message, writer)

async def main():
    async def redraw_output(items: deque):
        save_cursor_position()
        move_to_top_of_screen()
        for item in items:
            delete_line()
            sys.stdout.write(item)
        restore_cursor_position()

    tty.setcbreak(0)
    os.system('clear')
    rows = move_to_bottom_of_screen()

    messages = MessageStore(redraw_output, rows - 1)

    stdin_reader = await create_stdin_reader()
    sys.stdout.write('Введите имя пользователя: ')
    username = await read_line(stdin_reader)

    reader, writer = await asyncio.open_connection('127.0.0.1', 8000)

    writer.write(f'CONNECT {username}\n'.encode())
    await writer.drain()

    message_listener = asyncio.create_task(listen_for_messages(reader,
                                                                messages))
    input_listener = asyncio.create_task(read_and_send(stdin_reader, writer))

    try:
        await asyncio.wait([message_listener, input_listener],
                           return_when=asyncio.FIRST_COMPLETED)
    except Exception as e:
        logging.exception(e)
        writer.close()
        await writer.wait_closed()

asyncio.run(main())

```

Читать данные, введенные пользователем, и отправлять их серверу

Подключиться к серверу и отправить сообщение CONNECT с именем пользователя

Создать задачу для прослушивания сообщений и задачу для прослушивания ввода данных. Ждать, пока какая-то из них завершится

Сначала мы запрашиваем у пользователя имя, после чего отправляем сообщение «CONNECT» серверу. Затем создаем две задачи: одна прослушивает сообщения от сервера, а другая читает сообщения, которые вводит пользователь, и отправляет их серверу. Далее мы ждем завершения любой из этих задач, обернув их вызовом `asyncio.wait`. Так делается, потому что завершение клиента возможно по любой из двух причин: его может отключить сервер или прослушиватель ввода

возбудит исключение. Если бы мы ждали обе задачи независимо, то программа могла бы зависнуть. Например, если бы сначала мы ждали завершения прослушивателя ввода, а отключил бы нас сервер, то остановить прослушиватель ввода мы уже никак не смогли бы. Со-программа `wait` предотвращает эту проблему, поскольку какой бы из прослушивателей ни закончил работу первым, наше приложение все равно завершится. Если бы нам здесь требовалась большая точность управления, то можно было бы проверить множества `done` и `pending`, которые возвращает `wait`. Например, если прослушиватель ввода возбудил исключение, то мы могли бы снять прослушиватель сообщений.

Запустив чат-сервер, а затем два клиента, мы сможем отправлять и получать в клиенте сообщения, как в обычном чате. Например, два подключившихся к чату пользователя могли бы сгенерировать такой поток сообщений:

```
Добро пожаловать! Подключено пользователей: 1!  
Подключился MissIslington!  
Подключился SirBedevere!  
SirBedevere: Это твой носяра?  
MissIslington: Нет, накладной!
```

Мы написали чат-сервер, способный обслуживать одновременно несколько пользователей в одном потоке, и клиент для него. Это приложение можно было бы сделать более надежным. Например, можно повторно отправлять сообщения в случае ошибки или расширить протокол, подтверждая клиенту получение сообщения. Сделать приложение по-настоящему качественным довольно трудно, это выходит за рамки книги, но может оказаться интересным упражнением для читателя, поскольку нужно продумать много возможностей отказа. Применяя описанные идеи, вы сможете создавать надежные серверные и клиентские приложения, отвечающие вашим потребностям.

Резюме

- Мы научились использовать низкоуровневые API транспортных механизмов и протоколов для построения простого HTTP-клиента. Эти API являются фундаментом для высокоуровневых потоковых API `asynio`, использовать их в приложениях общего характера не рекомендуется.
- Мы научились использовать классы `StreamReader` и `StreamWriter` для написания сетевых приложений. Именно эти высокоуровневые API рекомендованы для работы с потоками данных в `asynio`.
- Мы узнали, как использовать потоки данных для создания неблокирующих приложений с интерфейсом командной строки, которые продолжают отзываться на действия пользователя, даже когда в фоне работают какие-то задачи.

- Мы научились создавать серверы с помощью сопрограммы `start_server`. Именно этот подход, а не работа с сокетами напрямую рекомендуется для создания серверов в `asynio`.
- Мы узнали, как создать отзывчивые клиентские и серверные приложения с помощью потоков данных. Располагая этими знаниями, мы сможете сами создавать сетевые клиент-серверные приложения.

Веб-приложения



Краткое содержание главы

- Создание веб-приложений с помощью aiohttp.
- Интерфейс асинхронного серверного шлюза (ASGI).
- Создание веб-приложений ASGI с помощью Starlette.
- Асинхронные представления Django.

Веб-приложения стоят за большинством сайтов, с которыми мы имеем дело в современном интернете. Если вы работали программистом в компании, присутствующей в интернете, то, вероятно, на каком-то этапе карьеры принимали участие в разработке веб-приложения. В мире синхронного Python это означает, что вы пользовались такими каркасами, как Flask, Bottle или сверхпопулярный Django. Если не считать последних версий Django, то все эти каркасы строились без учета `asyncio`. Поэтому, когда веб-приложению нужно сделать что-то, что можно распараллелить, например отправить запрос базе данных или обратиться к сторонним API, в нашем распоряжении имеется только многопоточность или многопроцессность. А значит, нужно изучить новые каркасы, совместимые с `asyncio`.

В этой главе мы поговорим о нескольких популярных веб-каркасах, совместимых с `asyncio`. Сначала рассмотрим каркас, с которым уже имели дело, aiohttp, и построим асинхронные REST-совместимые API. Затем познакомимся с интерфейсом асинхронного серверного шлю-

за, или ASGI, который стал асинхронной заменой интерфейсу шлюза веб-сервера (WSGI) и лежит в основе работы многих веб-приложений. Применяя реализацию ASGI в каркасе Starlette, мы построим простой REST-совместимый API с поддержкой технологии WebSocket. Мы также рассмотрим асинхронные представления Django. Производительность веб-приложений всегда следует учитывать, обдумывая масштабирование, поэтому мы также приведем некоторые показатели производительности, полученные с помощью инструмента нагрузочного тестирования.

9.1 Разработка REST API с помощью aiohttp

Ранее мы использовали библиотеку aiohttp для построения HTTP-клиента, способного отправлять тысячи конкурентных запросов веб-приложению. Но aiohttp поддерживает создание не только HTTP-клиентов, но и HTTP-серверов, совместимых с asyncio.

9.1.1 Что такое REST?

Аббревиатура REST означает *representational state transfer* (передача представимых состояний). Эта парадигма широко используется в современной разработке веб-приложений, особенно в сочетании с одностраничными приложениями на основе таких каркасов, как React и Vue. REST предлагает структурированный способ проектирования API без сохранения информации о состоянии, независимый от технологии, применяемой на стороне клиента. REST API пригоден для взаимодействия с любыми клиентами, от мобильного телефона до браузера, нужно лишь изменить представление данных на стороне клиента.

Основной концепцией в REST является *ресурс*. Как правило, ресурсом может быть все, что можно назвать именем существительным: заказчик, продукт, учетная запись и т. д. Перечисленные выше ресурсы были одиночными сущностями, но в роли ресурсов могут выступать и коллекции, например «заказчики» или «продукты», к элементам которых можно обратиться по уникальному идентификатору. У одиночных элементов могут быть также подресурсы. Например, с заказчиком может быть связан список его любимых продуктов. Чтобы лучше понять идею, рассмотрим несколько REST API:

```
customers
customers/{id}
customers/{id}/favorites
```

Здесь мы видим три оконечные точки REST API. Первая, `customers`, относится к коллекции заказчиков. Потребитель такого API ожидает, что будет возвращен список заказчиков (возможно, разбитый на

страницы, если он велик). Вторая оконечная точка относится к одному заказчику и принимает в качестве параметра идентификатор `id`. Если заказчик уникально идентифицируется числом, то обращение к `customers/1` должно вернуть данные для заказчика с идентификатором `id`, равным 1. И последняя оконечная точка – пример подресурса. У заказчика может быть список любимых продуктов, который является его подресурсом. Обращение к `customers/1/favorites` должно вернуть список любимых продуктов заказчика с идентификатором 1.

При проектировании REST API мы будем возвращать данные в формате JSON, поскольку так чаще всего и делают, но, в принципе, можно выбрать любой формат, отвечающий конкретным потребностям. REST API может даже поддерживать несколько представлений данных, выбираемых с помощью HTTP-заголовков.

Подробное рассмотрение всех деталей REST выходит за рамки этой книги, интересующимся порекомендуем диссертацию создателя REST по адресу <http://mng.bz/1jAg>.

9.1.2 Основы разработки серверов на базе aiohttp

Начнем с создания простого API типа «hello world» с помощью aiohttp. Для начала создадим оконечную точку GET, которая будет возвращать дату и время в формате JSON. Назовем эту оконечную точку `/time` и сделаем так, чтобы она возвращала месяц, день и текущее время.

Aiohttp предоставляет функциональность веб-сервера в модуле `web`. Импортировав его, мы можем определить оконечные точки (которые в aiohttp называются *маршрутами*) с помощью класса `RouteTableDef`, который предлагает декоратор, позволяющий задать тип запроса (GET, POST и т. д.) и строку с именем оконечной точки. Затем декоратором `RouteTableDef` можно снабдить сопрограммы, исполняемые при обращении к этой оконечной точке. Внутри этих декорированных сопрограмм можно выполнить произвольную логику приложения и вернуть данные клиенту.

Но одного лишь создания оконечных точек недостаточно, нужно еще запустить веб-приложение, которое будет обслуживать маршруты. Для этого создадим экземпляр класса `Application`, добавим в него маршруты из `RouteTableDef` и запустим приложение.

Листинг 9.1 Оконечная точка для возврата текущего времени

```
from aiohttp import web
from datetime import datetime
from aiohttp.web_request import Request
from aiohttp.web_response import Response
routes = web.RouteTableDef()
```

```
@routes.get('/time')
async def time(request: Request) -> Response:
    today = datetime.today()
```

Создать оконечную GET-точку `time`; когда клиент обратится к ней, будет вызвана сопрограмма `time`

```

result = {
    'month': today.month,
    'day': today.day,
    'time': str(today.time())
}

return web.json_response(result)

```

Взять словарь result и преобразовать его в формат JSON

```

app = web.Application()
app.add_routes(routes)
web.run_app(app)

```

Создать веб-приложение, зарегистрировать маршруты и запустить его

Здесь мы сначала создаем окончечную точку `time`. Запись `@routes.get('/time')` говорит, что декорированная сопрограмма будет вызвана, когда клиент выполнит GET-запрос к URI `/time`. В сопрограмме `time` мы получаем месяц, день и время и сохраняем их в словаре. Затем мы вызываем функцию `web.json_response`, которая преобразует словарь в формат JSON. Она же настраивает возвращаемый клиенту HTTP-ответ, в частности прописывает в нем код состояния 200 и тип содержимого `'application/json'`.

Затем создается и запускается веб-приложение. Сначала мы создаем экземпляр класса `Application` и вызываем его метод `add_routes`. Он регистрирует все декораторы, созданные в веб приложении. После этого мы вызываем функцию `run_app`, которая запускает веб-сервер. По умолчанию сервер будет прослушивать порт 8080 на локальном компьютере `localhost`.

Протестировать это приложение можно, обратившись к адресу `localhost:8080/time` либо в браузере, либо с помощью какой-нибудь командной утилиты, например `cURL` или `Wget`. Остановимся на `cURL` и выведем полный ответ, выполнив команду `curl -i localhost:8080/time`. В ответ мы увидим примерно такую картину:

```

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 51
Date: Mon, 23 Nov 2020 16:35:32 GMT
Server: Python/3.9 aiohttp/3.6.2

{"month": 11, "day": 23, "time": "11:35:32.033271"}

```

Итак, мы успешно создали свою первую окончечную точку с помощью `aiohttp`! Вы, наверное, обратили внимание, что у сопрограммы `time` есть параметр `request`. В этом примере мы его не использовали, но вскоре продемонстрируем его важность. В структуре данных `request` хранится информация о веб-запросе клиента: тело, строка запроса и т. д. Чтобы получить представление о заголовках запроса, добавьте предложение `print(request.headers)` куда-нибудь в сопрограмму `time` – будет напечатано что-то вроде

```

<CIMultiDictProxy('Host': 'localhost:8080', 'User-Agent': 'curl/7.64.1',
'Accept': '*//*')>

```


9.1.3 Подключение к базе данных и получение результатов

На примере конечной точки `time` мы продемонстрировали основные моменты, но в большинстве своем веб-приложения не так просты. Обычно нужно подключиться к базе данных, например PostgreSQL или Redis, и, возможно, взаимодействовать с другими REST API, например с API поставщика товаров для запроса или обновления.

Чтобы посмотреть, как это делается, построим REST API для доступа к базе данных интернет-магазина из главы 5. Именно, мы предоставим возможность получать информацию о существующих товарах и создавать новые.

Прежде всего нужно создать подключение к базе данных. Поскольку мы ожидаем, что у приложения будет много конкурентных пользователей, имеет смысл создать не одно подключение, а сразу пул. Вопрос: где должен находиться пул подключений, чтобы конечные точки могли легко им воспользоваться?

Для ответа на этот вопрос надо решить более общую проблему: где хранить разделяемые данные в приложениях на основе aiohttp. Тогда общим механизмом можно будет воспользоваться и для хранения ссылки на пул подключений.

Для хранения разделяемых данных класс `aiohttp Application` может выступать в роли словаря. Например, если бы мы хотели завести разделяемый словарь, к которому имеют доступ все маршруты, то могли бы сохранить его в приложении следующим образом:

```
app = web.Application()
app['shared_dict'] = {'key' : 'value'}
```

Теперь для обращения к разделяемому словарю нужно просто написать `app['shared_dict']`. Далее нужно решить, как обращаться к приложению из маршрута. Можно было бы сделать экземпляр приложения глобальным, но aiohttp предлагает лучший способ с помощью класса `Request`. Каждый запрос, передаваемый маршруту, содержит ссылку на экземпляр приложения в поле `app`, что позволяет легко обратиться к разделяемым данным. Например, чтобы получить разделяемый словарь и вернуть его в качестве ответа, можно было бы написать:

```
@routes.get('/')
async def get_data(request: Request) -> Response:
    shared_data = request.app['shared_dict']
    return web.json_response(shared_data)
```

Воспользуемся этой идеей, чтобы сохранить созданный пул подключений к базе данных. Но сначала решим, где лучше создать этот пул. Сделать это там, где создается экземпляр приложения, затруднительно, потому что это происходит вне сопрограммы, т. е. мы не сможем использовать выражения `await`.

Приложение aiohttp предоставляет обработчик сигнала `on_startup`, который можно использовать для инициализации. Можете считать,

что это список сопрограмм, выполняемых при запуске приложения. Для добавления сопрограммы нужно написать `app.on_startup.append(coroutine)`. Любой добавленной таким образом сопрограмме передается один параметр: экземпляр класса `Application`. Поэтому, создав пул подключений, мы сможем сохранить его в экземпляре приложения.

Нужно также подумать, что должно происходить при остановке приложения. Мы хотим явно закрывать и очищать подключения к базе данных, чтобы не оставалось висячих подключений, создающих ненужную нагрузку на базу данных. `aiohhttp` предоставляет еще один обработчик сигнала, `on_cleanup`. Зарегистрированные в нем сопрограммы будут вызываться при закрытии приложения, так что именно там проще всего остановить пул подключений. Для добавления сопрограмм тоже используется метод `append`.

Собирая все вместе, мы можем написать веб-приложение, которое создает пул подключений к базе данных о товарах. Чтобы протестировать его, заведем оконечную GET-точку, которая возвращает все имеющиеся в базе данных торговые марки. Назовем ее `/brands`.

Листинг 9.2 Подключение к базе данных о товарах

```
import asyncpg
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from asyncpg import Record
from asyncpg.pool import Pool
from typing import List, Dict

routes = web.RouteTableDef()
DB_KEY = 'database'

async def create_database_pool(app: Application):
    print('Создается пул подключений.')
    pool: Pool = await asyncpg.create_pool(host='127.0.0.1',
                                           port=5432,
                                           user='postgres',
                                           password='password',
                                           database='products',
                                           min_size=6,
                                           max_size=6)

    app[DB_KEY] = pool

async def destroy_database_pool(app: Application):
    print('Уничтожается пул подключений.')
    pool: Pool = app[DB_KEY]
    await pool.close()

@routes.get('/brands')
async def brands(request: Request) -> Response:
```

Создать пул подключений к базе данных и сохранить его в экземпляре приложения

Уничтожить пул в экземпляре приложения

Запросить все марки и вернуть результаты клиенту

```

connection: Pool = request.app[DB_KEY]
brand_query = 'SELECT brand_id, brand_name FROM brand'
results: List[Record] = await connection.fetch(brand_query)
result_as_dict: List[Dict] = [dict(brand) for brand in results]
return web.json_response(result_as_dict)

app = web.Application()
app.on_startup.append(create_database_pool)
app.on_cleanup.append(destroy_database_pool)
app.add_routes(routes)
web.run_app(app)

```

Добавить сопрограммы создания и уничтожения пула в обработчики инициализации и очистки

Сначала определяются две сопрограммы: для создания и уничтожения пула подключений. В сопрограмме `create_database_pool` мы создаем пул и сохраняем его в приложении под ключом `DB_KEY`. В сопрограмме `destroy_database_pool` мы получаем пул из экземпляра приложения и ждем его закрытия. На этапе запуска приложения эти сопрограммы добавляются в обработчики сигналов `on_startup` и `on_cleanup` соответственно.

Далее мы определяем маршрут `brands`. Сначала получаем пул подключений из веб-запроса и запрашиваем все имеющиеся в базе данных марки. Затем в цикле преобразуем данные о марках в словари. Это необходимо, потому что `aiohttp` не знает, как сериализовывать объекты `asyncpg Record`. После запуска приложения мы сможем перейти по адресу `localhost:8080/brands` в браузере и увидеть все марки в виде списка в формате JSON:

```
[{"brand_id": 1, "brand_name": "his"}, {"brand_id": 2, "brand_name": "he"},
 {"brand_id": 3, "brand_name": "at"}]
```

Итак, мы создали REST-совместимую оконечную точку для получения коллекции. Далее покажем оконечные точки для создания и обновления одиночных ресурсов. Мы реализуем две такие точки: типа `GET` для получения товара с указанным идентификатором и типа `POST` для создания нового товара.

Начнем с оконечной точки типа `GET` для получения товара. Она будет принимать целочисленный идентификатор, т. е. товару с идентификатором 1 будет соответствовать точка `/products/1`. Как создать маршрут с параметром? Библиотека `aiohttp` для этой цели предлагает заключать параметры в фигурные скобки, т. е. маршрут будет иметь вид `/products/{id}`. Соответствующие параметрам значения мы найдем в словаре `match_info`, хранящемся в запросе. В данном случае заданное пользователем значение параметра `id` можно будет получить в виде строки, написав `request.match_info['id']`.

Поскольку пользователь может передать недопустимый идентификатор, необходима обработка ошибок. Кроме того, клиент может запросить несуществующий идентификатор, поэтому мы должны правильно обрабатывать случай «не найдено». В обоих случаях мы бу-

дем возвращать код состояния HTTP 400, означающий, что клиент запросил несуществующий ресурс. Для представления ошибок в aiohttp имеется набор исключений, соответствующих кодам состояния HTTP. Стоит возбудить одно из них, и клиент получит отвечающий ему код состояния.

Листинг 9.3 Получение конкретного товара

```
import asyncpg
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from asyncpg import Record
from asyncpg.pool import Pool

routes = web.RouteTableDef()
DB_KEY = 'database'

@routes.get('/products/{id}')
async def get_product(request: Request) -> Response:
    try:
        str_id = request.match_info['id']
        product_id = int(str_id)

        query = \
            """
            SELECT
            product_id,
            product_name,
            brand_id
            FROM product
            WHERE product_id = $1
            """

        connection: Pool = request.app[DB_KEY]
        result: Record = await connection.fetchrow(query, product_id)

        if result is not None:
            return web.json_response(dict(result))
        else:
            raise web.HTTPNotFound()
    except ValueError:
        raise web.HTTPBadRequest()

async def create_database_pool(app: Application):
    print('Создается пул подключений.')
    pool: Pool = await asyncpg.create_pool(host='127.0.0.1',
                                           port=5432,
                                           user='postgres',
                                           password='password',
                                           database='products',
                                           min_size=6,
```

Получить параметр product_id из URL

Выполнить запрос для одного товара

Если получен результат, преобразовать его в формат JSON и отправить клиенту, в противном случае отправить сообщение "404 not found"

```

max_size=6)

app[DB_KEY] = pool

async def destroy_database_pool(app: Application):
    print('Уничтожается пул подключений.')
    pool: Pool = app[DB_KEY]
    await pool.close()

app = web.Application()
app.on_startup.append(create_database_pool)
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app)

```

Далее рассмотрим окончечную точку типа POST для создания нового товара в базе данных. Будем отправлять данные в теле запроса в виде JSON-строки, а затем транслировать ее в операцию вставки в базу данных. Нужно будет проверять, что данные в формате JSON корректны, и, если это не так, возвращать клиенту код состояния, указывающий на некорректность запроса.

Листинг 9.4 Оконечная точка для создания товара

```

import asyncpg
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from chapter_09.listing_9_2 import create_database_pool,
                                destroy_database_pool

routes = web.RouteTableDef()
DB_KEY = 'database'

@routes.post('/product')
async def create_product(request: Request) -> Response:
    PRODUCT_NAME = 'product_name'
    BRAND_ID = 'brand_id'

    if not request.can_read_body:
        raise web.HTTPBadRequest()

    body = await request.json()

    if PRODUCT_NAME in body and BRAND_ID in body:
        db = request.app[DB_KEY]
        await db.execute(''INSERT INTO product(product_id,
                                                product_name,
                                                brand_id)
                        VALUES(DEFAULT, $1, $2)''',
                        body[PRODUCT_NAME],
                        int(body[BRAND_ID]))

```

```

        return web.Response(status=201)
    else:
        raise web.HTTPBadRequest()

app = web.Application()
app.on_startup.append(create_database_pool)
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app)

```

Сначала мы с помощью функции `request.can_read_body` проверяем, не пусто ли тело, и, если пусто, сразу возвращаем код некорректного запроса. Затем получаем тело запроса в виде словаря с помощью сопрограммы `json`. Почему это сопрограмма, а не простая функция? Если запрос очень велик, то тело может быть буферизовано и его чтение займет некоторое время. Вместо того чтобы блокировать обработчик в ожидании поступления всех данных, мы в предложении `await` ждем, когда они будут готовы. Затем вставляем запись в таблицу товаров и возвращаем клиенту код состояния 201 Created.

Воспользовавшись `cURL`, мы получим показанный ниже результат, где видно, что после вставки товара в базу данных получен код состояния HTTP 201.

```

curl -i -d '{"product_name":"product_name", "brand_id":1}'
localhost:8080/product
HTTP/1.1 201 Created
Content-Length: 0
Content-Type: application/octet-stream
Date: Tue, 24 Nov 2020 13:27:44 GMT
Server: Python/3.9 aiohttp/3.6.2

```

Обработку ошибок следовало бы сделать более полной (например, что будет, если идентификатор марки не целое число или JSON-строка некорректна?), но процесс обработки POST-запроса для вставки записи в базу данных понятен.

9.1.4 Сравнение *aiohttp* и *Flask*

Благодаря библиотеке `aiohttp`, совместимой с `asyncio`, мы получаем возможность пользоваться такими асинхронными библиотеками, как `asynrcpg`. Но если не считать этого, то есть ли преимущества у использования библиотек типа `aiohttp` перед аналогичными синхронными каркасами, например `Flask`?

Конечно, все зависит от конфигурации сервера, оборудования сервера баз данных и других факторов, но в общем и целом приложения на основе `asyncio` позволяют добиться большей пропускной способности при меньшем потреблении ресурсов. В синхронном каркасе каждый обработчик запроса работает от начала до конца, не прерываясь. В асинхронном каркасе выражения `await` приостанавливают вы-

полнение и дают каркасу возможность заняться другой работой, что повышает эффективность.

Для проверки этого утверждения напишем эквивалент окончной точки `brands` в каркасе `Flask`. Предполагается знакомство с основами `Flask` и синхронных драйверов баз данных, хотя, даже если вы ничего об этом не знаете, понять код сможете. Начнем с установки `Flask` и `psycopg2`, синхронного драйвера `Postgres`:

```
pip install -Iv flask==2.0.1
pip install -Iv psycopg2==2.9.1
```

При установке `psycopg` возможны ошибки компиляции. В таком случае нужно будет установить инструменты `Postgres`, `OpenSSL` или еще какую-то библиотеку. Поиск в вебе информации об ошибке должен дать ответ. А теперь реализуем окончную точку. Сначала создадим подключение к базе данных. В обработчике запроса используем тот же запрос, что в предыдущем примере, и вернем результаты в виде `JSON`-массива.

Листинг 9.5 Приложение `Flask` для выборки торговых марок

```
from flask import Flask, jsonify
import psycopg2

app = Flask(__name__)
conn_info = "dbname=products user=postgres password=password host=127.0.0.1"
db = psycopg2.connect(conn_info)

@app.route('/brands')
def brands():
    cur = db.cursor()
    cur.execute('SELECT brand_id, brand_name FROM brand')
    rows = cur.fetchall()
    cur.close()
    return jsonify([{'brand_id': row[0], 'brand_name': row[1]} for row in rows])
```

Теперь нужно приложение запустить. В состав `Flask` входит сервер для разработки, но его качество далеко от производственного, поэтому сравнение будет несправедливым, тем более что он запускает только один процесс, поэтому в каждый момент времени способен обрабатывать лишь один запрос. Для тестирования нам нужен полнофункциональный `WSGI`-сервер. Воспользуемся сервером `Gunicorn`, хотя выбор велик. Установите `Gunicorn` следующей командой:

```
pip install -Iv gunicorn==20.1.0
```

Мы будем тестировать на 8-ядерной машине, поэтому запустим `Gunicorn` с восемью исполнителями. Выполнив команду `gunicorn -w 8 chapter_09.listing_9_5:app`, вы увидите, что работает восемь исполнителей:

```
[2020-11-24 09:53:39 -0500] [16454] [INFO] Starting gunicorn 20.0.4
[2020-11-24 09:53:39 -0500] [16454] [INFO] Listening at:
    http://127.0.0.1:8000 (16454)
[2020-11-24 09:53:39 -0500] [16454] [INFO] Using worker: sync
[2020-11-24 09:53:39 -0500] [16458] [INFO] Booting worker with pid: 16458
[2020-11-24 09:53:39 -0500] [16459] [INFO] Booting worker with pid: 16459
[2020-11-24 09:53:39 -0500] [16460] [INFO] Booting worker with pid: 16460
[2020-11-24 09:53:39 -0500] [16461] [INFO] Booting worker with pid: 16461
[2020-11-24 09:53:40 -0500] [16463] [INFO] Booting worker with pid: 16463
[2020-11-24 09:53:40 -0500] [16464] [INFO] Booting worker with pid: 16464
[2020-11-24 09:53:40 -0500] [16465] [INFO] Booting worker with pid: 16465
[2020-11-24 09:53:40 -0500] [16468] [INFO] Booting worker with pid: 16468
```

Это значит, что мы создали восемь подключений к базе данных и можем одновременно обслуживать восемь запросов. Теперь нужен инструмент для сравнения производительности Flask и aiohttp. Подойдет командная программа нагрузочного тестирования. Картина, конечно, будет не очень точная, но общее представление мы получим. Воспользуемся программой wrk, хотя можно было бы взять любую другую, например Apache Bench или Hey. Инструкции по установке имеются по адресу <https://github.com/wg/wrk>.

Начнем с 30-секундного нагрузочного теста сервера Flask. Создадим один поток и 200 подключений, моделируя 200 пользователей, одновременно обращающихся к нашему приложению с максимальной скоростью. На 8-ядерной машине с тактовой частотой 2,4 ГГц результаты получились такие:

```
Running 30s test @ http://localhost:8000/brands
  1 threads and 200 connections
16534 requests in 30.02s, 61.32MB read
Socket errors: connect 0, read 1533, write 276, timeout 0
Requests/sec:  550.82
Transfer/sec:   2.04MB
```

Мы обслужили примерно 550 запросов в секунду – неплохой результат. Теперь запустим вариант с aiohttp и сравним:

```
Running 30s test @ http://localhost:8080/brands
 1 threads and 200 connections
46774 requests in 30.01s, 191.45MB read
Requests/sec: 1558.46
Transfer/sec:  6.38MB
```

Благодаря aiohttp мы смогли обслужить более 1500 запросов в секунду, т. е. в три раза больше, чем Flask. Но еще важнее, что для этого понадобился всего один процесс, тогда как Flask с помощью *восьми процессов* сумел обслужить только *треть* этого числа! Производительность aiohttp можно еще увеличить, поместив перед сервером NGINX и запустив больше рабочих процессов.

Теперь мы умеем использовать aiohttp для построения веб-приложения, работающего с базой данных. В мире веб-приложений aiohttp

отличается тем, что сама она является веб-сервером и, не отвечая требованиям к WSGI, может работать автономно. Как мы видели на примере Flask, обычно бывает не так. Далее мы поговорим о том, что такое интерфейс ASGI, и увидим, как им пользоваться на примере ASGI-совместимого каркаса Starlette.

9.2 Асинхронный интерфейс серверного шлюза

В предыдущем примере мы использовали Flask совместно с WSGI-сервером Gunicorn. WSGI – стандартизованный способ передачи запросов веб-каркасу, например Flask или Django. Различных WSGI-серверов много, но при их проектировании поддержка асинхронных рабочих нагрузок не закладывалась, поскольку спецификация WSGI появилась намного раньше `asyncio`. По мере того как асинхронные веб-приложения получали все большее распространение, возникла необходимость абстрагировать каркасы от серверов. Так появилась *асинхронный интерфейс серверного шлюза*, или ASGI. Несмотря на сравнительно юный возраст, ASGI уже имеет несколько популярных реализаций и его поддерживает несколько каркасов, включая Django.

9.2.1 Сравнение ASGI и WSGI

Стандарт WSGI вырос на фрагментированном поле каркасов веб-приложений. Раньше выбор каркаса ограничивал множество пригодных для его использования веб-серверов, поскольку теми и другими не было стандартизованного интерфейса. WSGI решил эту проблему, специфицировав простой API для взаимодействия веб-серверов и написанных на Python каркасов. WSGI формально прописался в экосистеме Python в 2004 году после утверждения документа PEP-333 (предложение по улучшению Python; <https://www.python.org/dev/peps/pep-0333/>) и теперь является стандартом де факто для разработки веб-приложений.

Однако с асинхронными рабочими нагрузками WSGI не работает. В основе спецификации WSGI лежит простая Python-функция. Например, рассмотрим простейшее WSGI-приложение.

Листинг 9.6 WSGI-приложение

```
def application(env, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b"WSGI hello!"]
```

Мы можем запустить его на сервере Gunicorn, выполнив команду `gunicorn chapter_09.listing_9_6`, и протестировать с помощью `curl http://127.0.0.1:8000`. Как видите, для `await` здесь нет места. Кроме того, WSGI поддерживает только жизненные циклы запрос–ответ, т. е.

не будет работать с протоколами долговременных подключений, например WebSockets. ASGI исправляет ситуацию путем перепроектирования API под использование сопрограмм. Переведем этот пример с WSGI на ASGI.

Листинг 9.7 Простое ASGI-приложение

```
async def application(scope, receive, send):
    await send({
        'type': 'http.response.start',
        'status': 200,
        'headers': [[b'content-type', b'text/html']]
    })
    await send({'type': 'http.response.body', 'body': b'ASGI hello!'})
```

ASGI-функция `application` принимает три параметра: словарь `scope`, сопрограмму `receive` и сопрограмму `send`, которые служат для отправки и приема данных соответственно. В нашем примере мы сначала отправляем заголовки HTTP-ответа, а вслед за ними тело.

А как запустить это приложение? Есть несколько реализаций ASGI, мы воспользуемся одной из самых популярных, Uvicorn (<https://www.uvicorn.org/>). Сервер Uvicorn построен поверх uvloop и httptools – написанных на С быстрых реализаций цикла событий asyncio (на самом деле мы не привязаны к циклу событий, входящему в состав asyncio, но об этом поговорим в главе 14) и разбора HTTP. Для установки Uvicorn выполните команду:

```
pip install -Iv uvicorn==0.14.0
Теперь можно запустить приложение:
uvicorn chapter_09.listing_9_7:application
```

Перейдя по адресу `http://localhost:8000`, мы увидим сообщение «hello». Здесь мы для тестирования использовали Uvicorn напрямую, но рекомендуется запускать Uvicorn с помощью Gunicorn, потому что в Gunicorn встроена логика автоматического перезапуска рабочих процессов в случае аварии. Как это делается для Django, увидим в разделе 9.4.

Следует иметь в виду, что ASGI, в отличие от WSGI, еще не утвержден каким-нибудь PEP и на момент написания книги считается сравнительно новым стандартом. Вероятно, некоторые особенности работы ASGI будут со временем меняться.

Итак, мы уяснили различия между ASGI и WSGI. Но пока это все это на очень низком уровне, а хотим мы иметь каркас, который реализует ASGI за нас! Таких каркасов несколько, рассмотрим один из наиболее популярных.

9.3 Реализация ASGI в Starlette

Starlette – небольшой ASGI-совместимый каркас, созданный компанией Encode, разработавшей Uvicorn и другие популярные библиотеки,

например реализацию REST для Django. Он может похвастаться впечатляющей производительностью, поддержкой WebSocket и другими возможностями. Документация выложена на сайте <https://www.starlette.io/>. Посмотрим, как реализовать с его помощью простые оконечные точки типа REST и WebSocket. Но сначала установите его командой:

```
pip install -Iv starlette==0.15.0
```

9.3.1 Оконечная REST-точка в Starlette

Начнем изучение Starlette с реализации оконечной точки `brands` из предыдущих разделов. Для создания приложения нужно создать экземпляр класса `Starlette`. Его конструктор принимает несколько параметров, в том числе: список объектов `Route` и список сопрограмм, исполняемых на этапах инициализации и остановки. Объекты `Route` представляют собой отображения строкового пути (в нашем случае `brands`) на сопрограмму или другой вызываемый объект. Как и в `aihttp`, эти сопрограммы принимают один параметр, представляющий запрос, и возвращают ответ, поэтому обработчик маршрута очень похож на версию для `aihttp`. Отличаются они тем, как мы работаем с пулом подключений к базе данных. Он, как и раньше, хранится в экземпляре приложения `Starlette`, но не в нем самом, а во внутреннем объекте состояния.

Листинг 9.8 Оконечная точка `brands` в приложении `Starlette`

```
import asyncpg
from asyncpg import Record
from asyncpg.pool import Pool
from starlette.applications import Starlette
from starlette.requests import Request
from starlette.responses import JSONResponse, Response
from starlette.routing import Route
from typing import List, Dict

async def create_database_pool():
    pool: Pool = await asyncpg.create_pool(host='127.0.0.1',
                                           port=5432,
                                           user='postgres',
                                           password='password',
                                           database='products',
                                           min_size=6,
                                           max_size=6)

    app.state.DB = pool

async def destroy_database_pool():
    pool = app.state.DB
    await pool.close()

async def brands(request: Request) -> Response:
    connection: Pool = request.app.state.DB
```

```

brand_query = 'SELECT brand_id, brand_name FROM brand'
results: List[Record] = await connection.fetch(brand_query)
result_as_dict: List[Dict] = [dict(brand) for brand in results]
return JSONResponse(result_as_dict)

app = Starlette(routes=[Route('/brands', brands)],
                on_startup=[create_database_pool],
                on_shutdown=[destroy_database_pool])

```

Итак, оконечная точка `brands` у нас есть, теперь запустим ее с помощью `Uvicorn`. Как и раньше, создадим восемь исполнителей следующей командой:

```
uvicorn --workers 8 --log-level error chapter_09.listing_9_8:app
```

Обратившись к этой точке по адресу `localhost:8000/brands`, мы, как и раньше, увидим содержимое `brand`. Теперь выполним тест производительности, чтобы сравнить с `aiohhttp` и `Flask`. Будем пользоваться той же командой `wrk`, открыв 200 подключений. Тест будет продолжаться 30 с.

```

Running 30s test @ http://localhost:8000/brands
 1 threads and 200 connections
Requests/sec: 4365.37
Transfer/sec: 16.07MB

```

Мы смогли обслужить 4000 запросов в секунду, намного опередив `Flask` и даже `aiohhttp`! Правда, при тестировании `aiohhttp` мы использовали один рабочий процесс, так что сравнение не вполне честное (запустив восемь экземпляров `aiohhttp` за `NGINX`, мы получим сходные результаты), но демонстрация пропускной способности асинхронных каркасов получилась убедительной.

9.3.2 *WebSockets u Starlette*

Традиционно клиент посылает серверу HTTP-запрос, сервер возвращает ответ, и на этом транзакция заканчивается. Но что, если мы хотим построить веб-страницу, которая обновляется сама, без участия пользователя? Например, показывать актуальный счетчик посетителей, присутствующих в данный момент на сайте. Это можно сделать с помощью JavaScript-кода, который опрашивает некую оконечную точку, например раз в несколько секунд, и выводит на странице последний результат.

Этот подход работает, но не лишен недостатков. Главный из них – дополнительная нагрузка на веб-сервер, поскольку каждый цикл запрос–ответ потребляет время и ресурсы. Особенно обидно, что счетчик пользователей мог не измениться, так что мы только зря нагружаем систему, не получая никакой новой информации (проблему можно было бы смягчить за счет кеширования, но принципиальный вопрос остается, а кеширование увеличивает сложность и накладные

расходы). Опрос по протоколу HTTP можно уподобить ребенку, который расположился на заднем сиденье и то и дело спрашивает: «Мы уже приехали?»

Веб-сокеты предлагают альтернативу HTTP-опросу. Вместо цикла запрос–ответ мы создаем один постоянный сокет. А затем просто пересылаем данные через этот сокет. Сокет двунаправленный, т. е. передачу может инициировать как клиент, так и сервер, не прибегая каждый раз к жизненному циклу запрос–ответ. Применительно к отображению актуального счетчика посетителей сервер должен просто *сообщить* нам о том, что появился новый пользователь. Как показано на рис. 9.1, нам нет нужды повторно задавать вопрос, создавая тем самым дополнительную нагрузку только для того, чтобы получить те же самые данные.

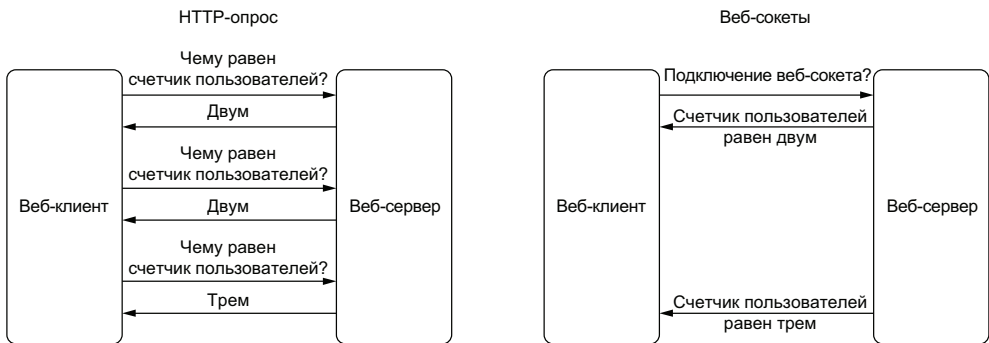


Рис. 9.1 Сравнение HTTP-опроса и веб-сокетов

Starlette предлагает встроенную поддержку веб-сокетов с помощью понятного интерфейса. Чтобы увидеть, как это работает, построим оконечную точку типа WebSocket, которая будет сообщать, сколько пользователей подключено к ней одновременно. Для начала установим поддержку веб-сокетов:

```
pip install -Iv websockets==9.1
```

Затем реализуем оконечную точку типа WebSocket. План состоит в том, чтобы хранить в памяти список всех подключенных к веб-сокету клиентов. Когда подключается новый клиент, мы добавляем его в список и рассылаем новый счетчик пользователей всем находящимся в списке клиентам. Когда клиент отключается, мы удаляем его из списка и сообщаем об этом изменении всем клиентам. Также добавим простую обработку ошибок. Если отправка сообщения приводит к исключению, то мы удаляем соответствующего клиента из списка.

В Starlette для создания оконечной точки типа WebSocket нужно унаследовать классу `WebSocketEndpoint` и реализовать в подклассе несколько сопрограмм. Первая, `on_connect`, вызывается, когда клиент

подключается к сокету. В ней мы будем сохранять веб-сокет клиента в списке и отправлять длину списка всем остальным сокетам. Вторая сопрограмма, `on_receive`, вызывается, когда по клиентскому подключению приходит сообщение серверу. Нам ее реализовывать не нужно, потому что мы не ожидаем, что клиенты будут отправлять данные. И последняя сопрограмма, `on_disconnect`, вызывается, когда клиент отключается. В этом случае мы удаляем клиента из списка подключенных веб-сокетов и сообщаем оставшимся клиентам новое значение счетчика.

Листинг 9.9 Оконечная точка типа WebSocket в Starlette

```
import asyncio
from starlette.applications import Starlette
from starlette.endpoints import WebSocketEndpoint
from starlette.routing import WebSocketRoute

class UserCounter(WebSocketEndpoint):
    encoding = 'text'
    sockets = []

    async def on_connect(self, websocket):
        await websocket.accept()
        UserCounter.sockets.append(websocket)
        await self._send_count()

    async def on_disconnect(self, websocket, close_code):
        UserCounter.sockets.remove(websocket)
        await self._send_count()

    async def on_receive(self, websocket, data):
        pass

    async def _send_count(self):
        if len(UserCounter.sockets) > 0:
            count_str = str(len(UserCounter.sockets))
            task_to_socket = {
                asyncio.create_task(websocket.send_text(count_str)): websocket
                for websocket in UserCounter.sockets
            }

            done, pending = await asyncio.wait(task_to_socket)
            for task in done:
                if task.exception() is not None:
                    if task_to_socket[task] in UserCounter.sockets:
                        UserCounter.sockets.remove(task_to_socket[task])

app = Starlette(routes=[WebSocketRoute('/counter', UserCounter)])
```

Когда клиент подключается, добавить его список сокетов и сообщить остальным новое значение счетчика

Когда клиент отключается, удалить его из списка и сообщить остальным новое значение счетчика

Сообщить всем клиентам, сколько пользователей сейчас подключено. Если во время отправки сообщения возникнет исключение, удалить соответствующий клиент из списка

Теперь определим страницу для взаимодействия с веб-сокетом. Мы напишем простой скрипт для подключения к конечной точке типа WebSocket. При получении сообщения будем обновлять счетчик на странице.

Листинг 9.10 Использование оконечной точки типа WebSocket

```
<!DOCTYPE html>
<html lang="">
<head>
  <title>Starlette Web Sockets</title>
  <script>
    document.addEventListener("DOMContentLoaded", () => {
      let socket = new WebSocket("ws://localhost:8000/counter");

      socket.onmessage = (event) => {
        const counter = document.querySelector("#counter");
        counter.textContent = event.data;
      };
    });
  </script>
</head>
<body>
  <span>Подключено пользователей: </span>
  <span id="counter"></span>
</body>
</html>
```

Здесь большую часть работы делает скрипт. Сначала мы подключаемся к оконечной точке, а затем определяем функцию обратного вызова `onmessage`. Она вызывается, когда сервер посылает нам данные. В ответ находим указанный элемент в DOM и записываем в него полученные данные. Отметим, что этот код выполняется после события `DOMContentLoaded`, иначе могло бы случиться, что элемент, содержащий счетчик, еще не существует в момент выполнения скрипта.

Запустив сервер командой `uvicorn --workers 1 chapter_09.listing_9_9:app` и открыв веб-страницу, вы увидите, что счетчик равен 1. Если открыть страницу несколько раз в разных вкладках, то счетчик во всех вкладках будет увеличиваться. После закрытия вкладки все счетчики уменьшатся. Отметим, что в данном примере рабочий процесс только один, поскольку мы храним в памяти разделяемое состояние (список сокетов); если бы рабочих процессов было несколько, то у каждого был бы свой список сокетов. В таком случае нужно было бы использовать какое-то постоянное хранилище, например базу данных.

Теперь мы умеем совместно использовать `aiohttp` и `Starlette` для создания совместимых с `asyncio` веб-приложений с оконечными точками типа REST и WebSocket. Эти каркасы популярны, но даже близко не могут сравниться с Django, королем всех веб-каркасов на Python.

9.4 Асинхронные представления Django

Django – один из самых популярных каркасов на Python. В него уже встроена обширнейшая функциональность, от ORM (объектно-реля-

ционных отображений) для работы с базами данных до настраиваемой административной консоли. До версии 3.0 приложения Django допускали только развертывание приложений на WSGI-серверах и почти не поддерживали `asynсio`, если не считать библиотеки `channels`. В версии 3.0 появилась поддержка ASGI, и начался процесс полного перевода Django на асинхронные рельсы. В версии 3.1 стали поддерживаться асинхронные представления, что позволяет напрямую использовать библиотеки `asynсio` в представлениях Django. На момент написания книги поддержка асинхронности в Django еще считается новшеством, а функциональность неполна (например, подсистема ORM полностью синхронна, но поддержка `asynсio` планируется). Следует ожидать, что поддержка асинхронности будет и дальше расширяться.

Использование асинхронных представлений мы рассмотрим на примере небольшого приложения, использующего `aihttp` в представлении. Допустим, что мы собираемся интегрироваться с внешним REST API и хотим написать утилиту, которая будет конкурентно выполнять несколько запросов, чтобы оценить время ответа, длину тела и количество ошибок (исключений). Мы построим представление, которое принимает в качестве параметров URL-адрес и счетчик запросов, обращается по этому адресу и агрегирует результаты, возвращая их в табличной форме.

Начнем с установки нужной версии Django:

```
pip install -Iv django==3.2.8
```

Теперь попросим административный инструмент Django создать заготовку приложения. Назовем проект `asynс_views`:

```
django-admin startproject asynс_views
```

После выполнения этой команды будет создан каталог `asynс_views` со следующей структурой:

```
asynс_views/  
  manage.py  
  asynс_views/  
    __init__.py  
    settings.py  
    urls.py  
    asgi.py  
    wsgi.py
```

Обратите внимание, что присутствуют оба файла – `wsgi.py` и `asgi.py`, т. е. мы сможем развернуть приложение на серверах обоих типов. Теперь можно воспользоваться Uvicorn и запросить простую страницу `hello world` в варианте для Django. Выполните следующую команду, находясь в каталоге верхнего уровня `asynс_views`:

```
gunicorn asynс_views.asgi:application -k uvicorn.workers.UvicornWorker
```


Перейдя по адресу `localhost:8000`, вы увидите приветственную страницу Django (рис. 9.2).

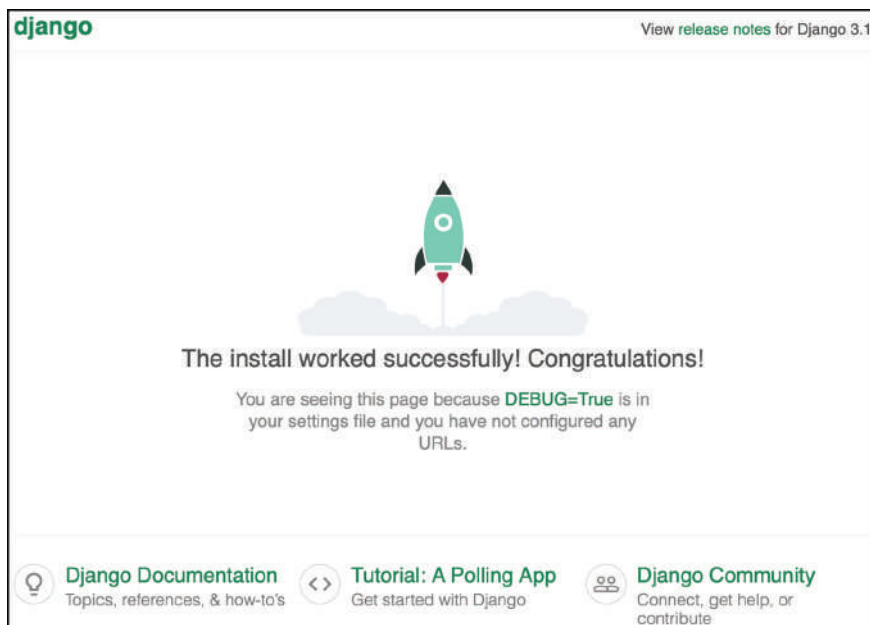


Рис. 9.2 Приветственная страница Django

Далее нужно создать приложение, которое мы назовем `async_api`. Находясь в каталоге `async_views`, выполните команду `python manage.py startapp async_api`. В результате будут построены файлы `model`, `view` и другие для приложения `async_api`.

Теперь все готово к созданию нашего первого асинхронного представления. В каталоге `async_api` должен находиться файл `views.py`. В нем мы можем сделать представление асинхронным, просто объявив его как сопрограмму. Мы реализуем асинхронное представление, которое будет конкурентно отправлять HTTP-запросы и отображать их коды состояния и другие данные в виде HTML-таблицы.

Листинг 9.11 Асинхронное представление Django

```
import asyncio
from datetime import datetime
from aiohttp import ClientSession
from django.shortcuts import render
import aiohttp

async def get_url_details(session: ClientSession, url: str):
    start_time = datetime.now()
    response = await session.get(url)
    response_body = await response.text()
```

```

end_time = datetime.now()
return {'status': response.status,
        'time': (end_time - start_time).microseconds,
        'body_length': len(response_body)}

async def make_requests(url: str, request_num: int):
    async with aiohttp.ClientSession() as session:
        requests = [get_url_details(session, url) for _ in range(request_num)]
        results = await asyncio.gather(*requests, return_exceptions=True)
        failed_results = [str(result) for result in results if
                           isinstance(result, Exception)]
        successful_results = [result for result in results if not
                              isinstance(result, Exception)]
        return {'failed_results': failed_results, 'successful_results':
                successful_results}

async def requests_view(request):
    url: str = request.GET['url']
    request_num: int = int(request.GET['request_num'])
    context = await make_requests(url, request_num)
    return render(request, 'async_api/requests.html', context)

```

Здесь мы сначала создаем сопрограмму, которая отправляет запрос и возвращает словарь, содержащий код состояния, общее время выполнения запроса и длину тела ответа. Далее определяем сопрограмму асинхронного представления `requests_view`. Она получает URL-адрес и счетчик запросов из параметров, конкурентно отправляет запросы с помощью `get_url_details` и собирает их результаты с помощью `gather`. Наконец, мы разделяем запросы на успешные и неудачные и помещаем результаты в словарь `context`, который затем передаем `render` для конструирования ответа. Заметим, что мы еще не написали шаблон ответа, поэтому пока передаем файл `async_views/requests.html`. Следующим шагом построим шаблон для представления результатов.

Сначала нужно создать подкаталог `templates` в каталоге `async_api`, а затем в нем создать папку `async_api`. Подготовив такую структуру каталогов, мы можем добавить представление в `async_api/templates/async_api`. Назовем его `requests.html` и обойдем словарь `context`, строя таблицу результатов.

Листинг 9.12 Представление `requests`

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Request Summary</title>
</head>
<body>
    <h1>Summary of requests:</h1>
    <h2>Failures:</h2>

```

```

<table>
    {% for failure in failed_results %}
    <tr>
        <td>{{failure}}</td>
    </tr>
    {% endfor %}
</table>
<h2>Successful Results:</h2>
<table>
    <tr>
        <td>Status code</td>
        <td>Response time (microseconds)</td>
        <td>Response size</td>
    </tr>
    {% for result in successful_results %}
    <tr>
        <td>{{result.status}}</td>
        <td>{{result.time}}</td>
        <td>{{result.body_length}}</td>
    </tr>
    {% endfor %}
</table>
</body>
</html>

```

В нашем представлении две таблицы: одна для отображения встретившихся исключений, а другая для отображения успешных результатов. Хотя это не самая красивая веб-страница, вся необходимая информация в ней есть. Далее мы должны связать шаблон и представление с URL-адресом, чтобы они выполнялись при отправке запроса из браузера. В папке `async_api` создайте такой файл `url.py`:

Листинг 9.13 Файл `async_api/url.py`

```

from django.urls import path
from . import views

app_name = 'async_api'

urlpatterns = [
    path('', views.requests_view, name='requests'),
]

```

Теперь нужно включить URL-адреса приложения `async_api` в приложение Django. В каталоге `async_views/async_views` уже должен присутствовать файл `urls.py`. В нем нужно изменить список `urlpatterns`, добавив в него ссылку на `async_api`. В итоге он будет выглядеть следующим образом:

```

from django.contrib import admin
from django.urls import path, include

```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('requests/', include('async_api.urls'))
]
```

Наконец, нужно добавить `async_views` в список установленных приложений. В файле `async_views/async_views/settings.py` измените список `INSTALLED_APPS`, включив в него `async_api`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'async_api'
]
```

Вот теперь есть все необходимое для запуска приложения. Запустить его можно той же командой `gunicorn`, которой мы пользовались при создании первого приложения Django. После этого можно перейти к нашей оконечной точке и отправлять запросы. Например, чтоб конкурентно выполнить 10 запросов к `example.com` и получить результаты, выполните команду:

```
http://localhost:8000/requests/?url=http://example.com&request_num=10
```

На вашей машине цифры могут быть другими, но вы увидите страницу типа показанной на рис. 9.3.

Итак, мы написали представление Django, которое может конкурентно отправлять произвольное число HTTP-запросов, для чего разместили его в ASGI-сервере; но что, если ASGI не вариант, например если вы имеете дело со старым приложением? Можно ли в таком случае разместить асинхронное представление? Попробуем сделать это, запустив под управлением Gunicorn WSGI-приложение из файла `wsgi.py` с синхронным исполнителем:

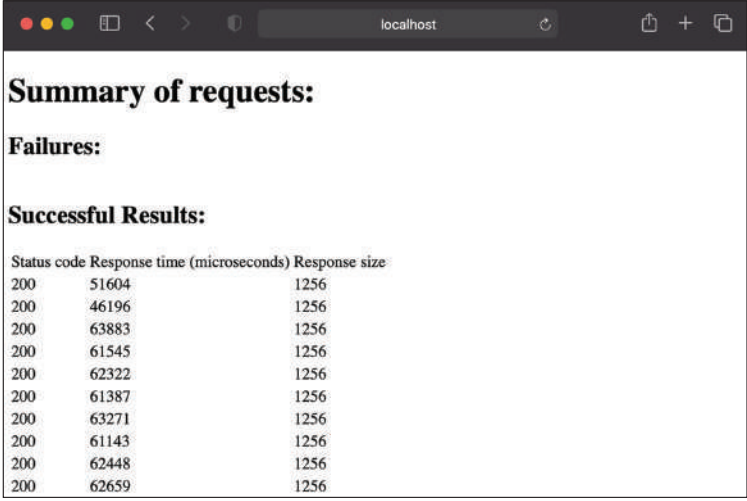
```
gunicorn async_views.wsgi:application
```

Мы по-прежнему можем обратиться к оконечной точке `requests`, и все будет работать нормально. А как же это работает? При выполнении WSGI-приложения каждое обращение к асинхронному представлению приводит к созданию нового цикла событий. В этом можно убедиться, добавив две строки в код представления:

```
loop = asyncio.get_running_loop()
print(id(loop))
```

Функция `id` вернет целое число, которое будет гарантированно уникально на всем протяжении жизни объекта. При выполнении WSGI-

приложения каждое обращение к конечной точке `requests` печатает новое целое число, доказывая, что новый цикл событий создается для каждого запроса. Если тот же код выполнить как ASGI-приложение, то каждый раз будет печататься одно и то же число, поскольку ASGI создает единственный цикл событий для всего приложения.



Status code	Response time (microseconds)	Response size
200	51604	1256
200	46196	1256
200	63883	1256
200	61545	1256
200	62322	1256
200	61387	1256
200	63271	1256
200	61143	1256
200	62448	1256
200	62659	1256

Рис. 9.3 Асинхронное представление `requests`

Это означает, что мы можем получить преимущества асинхронных представлений и выполнять запросы конкурентно даже в WSGI-приложении. Однако те вещи, которые нуждаются в том, чтобы цикл событий был одинаковым для всех запросов, будут работать только в ASGI-приложении.

9.4.1 *Выполнение блокирующих работ в асинхронном представлении*

А как быть, если в асинхронном представлении нужно выполнить блокирующую работу? Мы по-прежнему живем в мире, где многие библиотеки синхронны, но это не совместимо с моделью однопоточной конкурентности. В спецификации ASGI предусмотрена функция на такой случай – `sync_to_async`.

В главе 7 мы видели, что синхронные API можно выполнять в исполнителях пула потоков и получать в ответ допускающие ожидание объекты, которые можно использовать в сочетании с `asyncio`. Функция `sync_to_async` делает, по существу, то же самое, но с несколькими нюансами.

Первый нюанс состоит в том, что для `sync_to_async` определено понятие чувствительности к потоку. Во многих контекстах синхронные API с разделяемым состоянием не должны вызываться из нескольких

поток, потому что это может привести к состоянию гонки. Для решения этой проблемы `sync_to_async` по умолчанию работает в «поток-чувствительном» режиме (флаг `thread_sensitive` равен `True`). Из-за этого любой синхронный код выполняется в главном потоке Django, а значит, блокирующий код заблокирует все приложение Django (по крайней мере, рабочий процесс WSGI/ASGI, если таковых запущено несколько), что лишает нас ряда преимуществ асинхронного стека.

Если чувствительность к потоку не играет роли (иначе говоря, если разделяемого состояния нет вообще или неважно, в каком потоке оно находится), то флагу `thread_sensitive` можно присвоить значение `False`. Тогда каждый вызов будет выполняться в новом потоке, так что главный поток Django не будет блокироваться и преимущества асинхронного стека будут сохранены.

Для демонстрации создадим новое представление для тестирования различных вариантов `sync_to_async`. Мы напишем функцию, которая будет усыплять поток вызовом `time.sleep`, и передадим ее `sync_to_async`. Добавим в оконечную точку параметр, позволяющий легко переключаться между разными режимами чувствительности к потоку. Для начала включим в файл `async_views/async_api/views.py` следующее определение.

Листинг 9.14 Представление `sync_to_async_view`

```
from functools import partial
from django.http import HttpResponse
from asgiref.sync import sync_to_async

def sleep(seconds: int):
    import time
    time.sleep(seconds)

async def sync_to_async_view(request):
    sleep_time: int = int(request.GET['sleep_time'])
    num_calls: int = int(request.GET['num_calls'])
    thread_sensitive: bool = request.GET['thread_sensitive'] == 'True'
    function = sync_to_async(partial(sleep, sleep_time),
                             thread_sensitive=thread_sensitive)
    await asyncio.gather(*[function() for _ in range(num_calls)])
    return HttpResponse('')
```

Затем добавим новый маршрут в список `urlpatterns` в файле `async_views/async_api/urls.py`:

```
path('sync_to_async', views.sync_to_async_view)
```

Теперь можно обратиться к оконечной точке. Для проверки заснем на 5 с в потокочувствительном режиме, перейдя по следующему URL-адресу:

```
http://127.0.0.1:8000/requests/sync_to_async?sleep_time=5&num_calls=5&thread_sensitive=False
```

Обратите внимание, что на этот запрос ушло всего 5 с, потому что работает несколько потоков. Также обратите внимание, что если обратиться по этому адресу несколько раз, то каждый запрос все равно будет занимать 5 с; это доказывает, что запросы не блокируют друг друга. Теперь присвоим параметру `thread_sensitive` значение `True`. Поведение резко изменится. Во-первых, представление вернет результат только через 25 с, потому что вызовы `sleep` на 5 с выполняются последовательно. Во-вторых, если перейти по URL-адресу несколько раз подряд, то каждый вызов будет блокироваться, пока предыдущий не завершится, потому что блокируется главный поток Django. Функция `sync_to_async` предлагает несколько возможностей использовать существующий код в асинхронных представлениях, но следует учитывать чувствительность исполняемого кода к потоку, а также ограничения на производительность асинхронного выполнения, которые из-за этого могут иметь место.

9.4.2 Использование асинхронного кода в синхронных представлениях

Далее, естественно, возникает следующий вопрос: «Что, если имеется синхронное представление, а я хочу использовать библиотеку `asyncio`?» В спецификации ASGI и на этот случай есть специальная функция – `async_to_sync`. Она принимает сопрограмму и выполняет ее в цикле событий, возвращая результаты синхронно. Если цикла событий еще нет (как в случае приложения WSGI), то он будет создаваться при каждом запросе; в противном случае сопрограмма будет выполняться в текущем цикле (если работает как приложение ASGI). Для тестирования создадим новый вариант конечной точки `requests` в виде синхронного представления, но по-прежнему будем использовать асинхронную функцию запроса.

Листинг 9.15 Вызов асинхронного кода из синхронного представления

```
from asgiref.sync import async_to_sync

def requests_view_sync(request):
    url: str = request.GET['url']
    request_num: int = int(request.GET['request_num'])
    context = async_to_sync(partial(make_requests, url, request_num))()
    return render(request, 'async_api/requests.html', context)
```

Добавим следующий маршрут в список `urlpatterns` в файле `urls.py`:

```
path('async_to_sync', views.requests_view_sync)
```

Теперь мы можем обратиться к показанному ниже URL-адресу и наблюдать те же результаты, что в первом асинхронном представлении:

```
http://localhost:8000/requests/async_to_sync?url=http://example.com&request_
num=10
```

Даже в синхронном мире WSGI функция `sync_to_async` позволяет получить некоторые преимущества асинхронного стека, хотя и не дает полной асинхронности.

Резюме

- Мы научились создавать простые REST API, работающие с базами данных в сочетании с библиотеками `aihttp` и `asyncpg`.
- Мы узнали, как создавать ASGI-совместимые веб-приложения с помощью каркаса `Starlette`.
- Мы научились использовать веб-сокеты вместе со `Starlette` для разработки веб-приложений, возвращающих актуальную информацию без опроса по протоколу HTTP.
- Мы научились работать с асинхронными представлениями Django и узнали, как использовать асинхронный код в синхронных представлениях и наоборот.

10

Микросервисы

Краткое содержание главы

- Основы микросервисов.
- Паттерн backend-for-frontend.
- Использование `asycio` для взаимодействия с микросервисом.
- Использование `asycio` для обработки ошибок и повторных попыток.

Многие веб-приложения проектируются как монолиты. В этом контексте монолит означает средне- или крупномасштабное приложение, содержащее несколько модулей, которые развертываются независимо и управляются как единое целое. Ничего порочного в этой модели нет (монолиты вполне приемлемы и даже предпочтительны для большинства веб-приложений, поскольку обычно проще), но есть кое-какие недостатки.

Например, внося небольшое изменение в монолитное приложение, придется заново развертывать его целиком, даже те части, на которые изменение не влияет. Например, монолитный интернет-магазин может иметь окончательные точки для управления заказами и вывода описаний товаров. Вроде бы они независимы, но после мелкого изменения второй придется заново развертывать код управления заказами. Архитектура микросервисов призвана устранить такие болевые

точки. Мы можем создать отдельные микросервисы для заказов и товаров, и изменение одного не будет влиять на другой.

В этой главе мы поговорим о микросервисах и их обосновании. Мы познакомимся с паттерном *backend-for-frontend* и применим его к микросервисной архитектуре интернет-магазина. Затем реализуем этот API с помощью `aiohhttp` и `asyncpg` и научимся задействовать конкурентность для повышения производительности приложения. Мы также узнаем, как правильно обрабатывать ошибки и повторные попытки с помощью паттерна Прерыватель, чтобы приложение было надежнее.

10.1 Зачем нужны микросервисы?

Для начала определим, что такие микросервисы. Это непростой вопрос, потому что стандартного определения не существует, и, спросив двух человек, вы, вероятно, получите разные ответы. В общем случае у микросервисов есть несколько важных характеристик:

- они слабо связаны и развертываются независимо;
- у них есть свой независимый технологический стек, включающий модель данных;
- они взаимодействуют друг с другом по какому-то протоколу, например REST или gRPC;
- они следуют принципу «одной обязанности», т. е. микросервис «должен делать что-то одно, но делать это хорошо».

Применим эти принципы к конкретному примеру интернет-магазина. У подобного приложения есть пользователи, которые предоставляют платежные и отгрузочные реквизиты нашей гипотетической организации, а затем покупают наши товары. В монолитной архитектуре мы имели бы одно приложение с одной базой данных, содержащей данные о пользователях, об учетных записях (например, сведения о заказах и отгрузочные реквизиты) и о товарах. В микросервисной архитектуре было бы несколько сервисов для реализации различных аспектов, каждый со своей базой данных. Мы могли бы завести API товаров с собственной базой, в которой хранятся только сведения о товарах. И API пользователей с базой, где хранятся сведения об учетных записях, и т. д.

Почему стоило бы предпочесть такой архитектурный стиль, а не монолит? Монолиты отлично подходят для большинства приложений, потому что ими проще управлять. Внес изменение, прогнал все тесты, чтобы убедиться, что ничего не сломалось, а потом развернул приложение как единое целое. Приложение плохо работает под нагрузкой? Масштабируйте его горизонтально или вертикально, либо развернув дополнительные экземпляры приложения, либо установив более мощные компьютеры. Но хотя в эксплуатации монолиты проще, у этой простоты есть несколько недостатков, важность которых зависит от того, на какие компромиссы вы готовы пойти.

10.1.1 Сложность кода

По мере того как приложение растет и развивается, возрастает и его сложность. Может увеличиваться число связей между моделями, что вызывает непредвиденные зависимости, трудные для понимания. Технический долг растет, приложение становится медленным и запутанным. Это верно для любой растущей системы, но особенно для большой кодовой базы со многими обязанностями.

10.1.2 Масштабируемость

В монолитной архитектуре, когда возникает необходимость масштабирования, приходится добавлять экземпляры *всего* приложения, что может стать причиной неэффективных технологических затрат. В контексте интернет-магазина количество заказов обычно меньше, чем число посетителей, просто просматривающих товары. В монолитной архитектуре, чтобы масштабировать приложение для обработки большего числа посетителей, нужно было бы масштабировать заодно и функциональность приема заказов. В микросервисной архитектуре можно масштабировать только сервис товаров, а сервис заказов не трогать.

10.1.3 Независимость от команды и технологического стека

По мере роста команды разработчиков возникают новые проблемы. Представьте, что над разработкой одной монолитной кодовой базы трудятся пять команд, и каждая несколько раз в день фиксирует изменения. Все чаще будут возникать конфликты объединения, в решении которых должны принимать участие все. Координировать развертывание тоже становится все труднее. В случае независимых, слабо связанных микросервисов проблема не столь серьезна. Команда, владеющая некоторым сервисом, может развивать и развертывать его независимо. При желании команды могут даже использовать разные технологические стеки, скажем одна пишет на Java, а другая на Python.

10.1.4 Чем может помочь *asyncio*?

В общем случае микросервисы должны взаимодействовать друг с другом по какому-то протоколу, например REST или gRPC. Поскольку мы можем общаться несколькими микросервисами одновременно, открывается возможность выполнять запросы конкурентно и добиваться эффективности, невозможной в синхронном приложении.

Помимо эффективного использования ресурсов, мы также получаем преимущества обработки ошибок таких *asyncio* API, как *wait* и *gather*, – возможность агрегировать исключения, возбуждаемые группой сопрограмм или задач. Если какая-то группа запросов рабо-

тает слишком долго или часть этой группы возбуждает исключение, то мы можем аккуратно обработать эту ситуацию. Разобравшись с основными плюсами микросервисов, рассмотрим один широко распространенный архитектурный паттерн и его реализацию.

10.2 Введение в паттерн *backend-for-frontend*

В микросервисной архитектуре UI данные обычно поступают из нескольких источников и объединяются в одном представлении. Например, при построении UI для истории заказов пользователя мы, вероятно, должны будем объединить историю, полученную от сервиса заказов, с данными о товарах, полученными от сервиса товаров. В зависимости от требований могут понадобиться данные и от других сервисов.

Это ставит несколько проблем для фронтальных (frontend) клиентов. Первая – удобство пользователя. Когда сервисы автономны, UI-клиенты должны запрашивать данные от каждого сервиса по сети. В результате на загрузку UI уходит заметное время. Мы не можем предполагать, что у всех наших пользователей хорошее подключение к интернету и быстрый компьютер; у кого-то мобильный телефон в зоне неуверенного приема, кто-то работает на стареньких компьютерах, а кто-то ведет разработку из страны, где вообще нет высокоскоростного интернета. Если нужно отправить пять медленных запросов к пяти сервисам, то количество неприятностей может оказаться больше, чем в случае одного медленного запроса.

Помимо сетевых задержек, имеют место проблемы, связанные с принципами хорошего проектирования ПО. Представьте, что имеется как веб-интерфейс пользователя, так и мобильные интерфейсы для iOS и Android. Если вызвать каждый сервис непосредственно и объединять ответы, то придется повторить одну и ту же логику для трех разных клиентов, что избыточно и чревато появлением логических противоречий.

Существует много паттернов проектирования микросервисов, но в описанной ситуации нам может помочь паттерн *backend-for-frontend*. Его идея в том, что UI не взаимодействует с сервисами напрямую, а создает новый сервис, который отправляет вызовы и агрегирует результаты. Это решает наши проблемы, потому что вместо нескольких запросов мы отправляем только один, что уменьшает количество обращений к интернету. Мы также можем встроить в этот сервис произвольную логику отработки отказов или повторных попыток, не заставляя каждого клиента реализовывать ее. А в случае модификации логики придется внести изменения только в одно место. Наконец,

мы можем завести несколько сервисов типа backend-for-frontend для разных типов клиентов. Сервисы, с которыми нам предстоит взаимодействовать, могут быть разными для мобильных и веб-клиентов. Это показано на рис. 10.1. Разобравшись с паттерном проектирования backend-for-frontend и тем, какие проблемы он решает, применим его к построению такого сервиса для интернет-магазина.

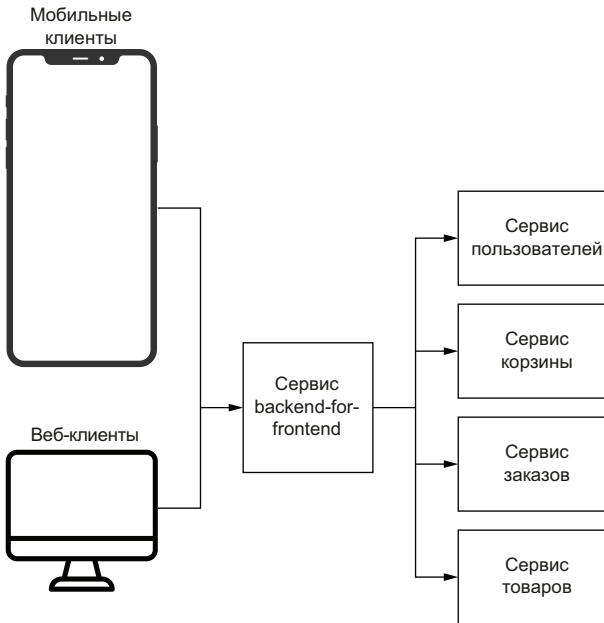


Рис. 10.1 Паттерн backend-for-frontend

10.3 Реализация API списка товаров

Реализуем паттерн backend-for-frontend для страницы *всех товаров* в интерфейсе, рассчитанном на ПК. На ней отображаются все товары на нашем сайте, а также минимальная информация о корзине пользователя и избранных им товарах в полосе меню. Для увеличения продаж на странице будет присутствовать предупреждение, если товара осталось мало. Интерфейс показан на рис. 10.2.

В архитектуре с несколькими независимыми сервисами мы должны будем запрашивать данные у каждого и объединять их для получения единого ответа. Начнем с определения базовых сервисов и моделей данных.

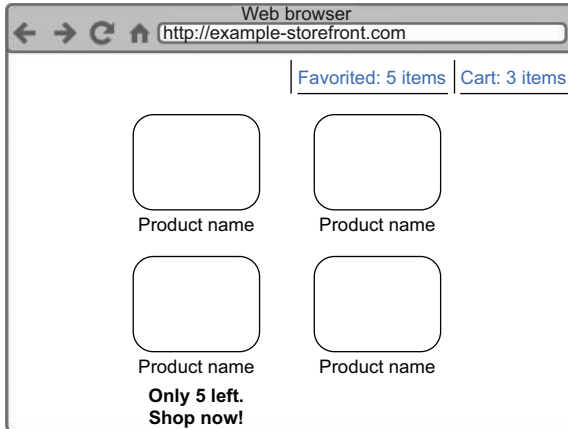


Рис. 10.2 Эскиз одной страницы списка товаров

10.3.1 Сервис избранного

Этот сервис отвечает за отображение пользователя на идентификаторы товаров, которые он поместил в список избранного. Кроме того, нам нужно будет реализовать следующие сервисы для поддержки товаров, наличия на складе, корзины и избранного.

Сервис корзины

Отвечает за отображение идентификатора пользователя на идентификаторы товаров, которые он положил в корзину; модель данных такая же, как для сервиса избранного.

Сервис наличие на складе

Отвечает за отображение идентификатора товара на наличие этого товара на складе.

Сервис товаров

Отвечает за информацию о товаре, например описание и SKU. Похож на сервис, реализованный в главе 9 для базы данных о товарах.

10.3.2 Реализация базовых сервисов

Начнем с реализации приложения aiohttp для сервиса наличия на складе, потому что он самый простой. Для него мы не будем созда-

вать отдельную модель данных, а просто вернем случайное число от 0 до 100, имитирующее складской остаток. Кроме того, добавим случайную задержку, чтобы смоделировать нестабильную скорость сервиса, и воспользуемся ей, чтобы продемонстрировать обработку тайм-аутов в сервисе списка товаров. Для разработки разместим этот сервис на порту 8001, чтобы он не вступал в конфликт с сервисом товаров из главы 9, который работает на порту 8000.

Листинг 10.1 Сервис наличия на складе

```
import asyncio
import random
from aiohttp import web
from aiohttp.web_response import Response

routes = web.RouteTableDef()

@routes.get('/products/{id}/inventory')
async def get_inventory(request: Request) -> Response:
    delay: int = random.randint(0, 5)
    await asyncio.sleep(delay)
    inventory: int = random.randint(0, 100)
    return web.json_response({'inventory': inventory})

app = web.Application()
app.add_routes(routes)
web.run_app(app, port=8001)
```

Далее реализуем сервисы корзины и избранного. Модели данных в них одинаковы, поэтому и сервисы будут почти одинаковыми, различающимися только именем таблицы. Начнем с двух моделей данных, «корзина пользователя» и «избранное пользователя». Вставим в обе таблицы несколько записей, чтобы было с чем работать. Сначала таблица `user_cart`.

Листинг 10.2 Таблица корзины `user_cart`

```
CREATE TABLE user_cart(
    user_id INT NOT NULL,
    product_id INT NOT NULL
);

INSERT INTO user_cart VALUES (1, 1);
INSERT INTO user_cart VALUES (1, 2);
INSERT INTO user_cart VALUES (1, 3);
INSERT INTO user_cart VALUES (2, 1);
INSERT INTO user_cart VALUES (2, 2);
INSERT INTO user_cart VALUES (2, 5);
```

Далее создадим очень похожую таблицу `user_favorite` и вставим в нее несколько записей:


```

async def destroy_database_pool(app: Application):
    pool: Pool = app[DB_KEY]
    await pool.close()

```

Этот код похож на код инициализации подключений к базе данных, написанный в главе 5. В сопрограмме `create_database_pool` мы создаем подключение и запоминаем его в экземпляре `Application`. В сопрограмме `destroy_database_pool` мы получаем пул из экземпляра приложения и закрываем его.

Далее перейдем к созданию сервисов. В терминах REST избранное и корзина – подсущности пользователя. Это означает, что корнем каждой оконечной точки должен быть `users`, а дополнительным параметром – идентификатор пользователя. Например, `/users/3/favorites` выбирает избранные товары пользователя с идентификатором 3. Сначала создадим сервис избранного.

Листинг 10.5 Сервис избранного

```

import functools
from aiohttp import web
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from chapter_10.listing_10_4 import DB_KEY, create_database_pool,
    destroy_database_pool

routes = web.RouteTableDef()

@routes.get('/users/{id}/favorites')
async def favorites(request: Request) -> Response:
    try:
        str_id = request.match_info['id']
        user_id = int(str_id)
        db = request.app[DB_KEY]
        favorite_query = 'SELECT product_id from user_favorite where user_id = $1'
        result = await db.fetch(favorite_query, user_id)
        if result is not None:
            return web.json_response([dict(record) for record in result])
        else:
            raise web.HTTPNotFound()
    except ValueError:
        raise web.HTTPBadRequest()

app = web.Application()
app.on_startup.append(functools.partial(create_database_pool,
                                         host='127.0.0.1',
                                         port=5432,
                                         user='postgres',
                                         password='password',
                                         database='favorites'))

app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app, port=8002)

```

Далее напишем сервис корзины. Он очень похож на предыдущий, только работаем мы теперь с таблицей `user_cart`.

Листинг 10.6 Сервис корзины

```
import functools
from aiohttp import web
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from chapter_10.listing_10_4 import DB_KEY, create_database_pool,
                                destroy_database_pool

routes = web.RouteTableDef()

@routes.get('/users/{id}/cart')
async def time(request: Request) -> Response:
    try:
        str_id = request.match_info['id']
        user_id = int(str_id)
        db = request.app[DB_KEY]
        favorite_query = 'SELECT product_id from user_cart where user_id = $1'
        result = await db.fetch(favorite_query, user_id)
        if result is not None:
            return web.json_response([dict(record) for record in result])
        else:
            raise web.HTTPNotFound()
    except ValueError:
        raise web.HTTPBadRequest()

app = web.Application()
app.on_startup.append(functools.partial(create_database_pool,
                                         host='127.0.0.1',
                                         port=5432,
                                         user='postgres',
                                         password='password',
                                         database='cart'))
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app, port=8003)
```

Наконец, напишем сервис товаров. Его API похож на созданный в главе 9, только теперь мы выбираем из базы данных не один товар, а все. Следующим листингом мы завершаем создание четырех сервисов для нашего гипотетического интернет-магазина!

Листинг 10.7 Сервис товаров

```
import functools
from aiohttp import web
from aiohttp.web_request import Request
from aiohttp.web_response import Response
from chapter_10.listing_10_4 import DB_KEY, create_database_pool,
```

```
destroy_database_pool

routes = web.RouteTableDef()

@routes.get('/products')
async def products(request: Request) -> Response:
    db = request.app[DB_KEY]
    product_query = 'SELECT product_id, product_name FROM product'
    result = await db.fetch(product_query)
    return web.json_response([dict(record) for record in result])

app = web.Application()
app.on_startup.append(functools.partial(create_database_pool,
                                         host='127.0.0.1',
                                         port=5432,
                                         user='postgres',
                                         password='password',
                                         database='products'))
app.on_cleanup.append(destroy_database_pool)

app.add_routes(routes)
web.run_app(app, port=8000)
```

10.3.3 Реализация сервиса backend-for-frontend

Следующим шагом построим сервис backend-for-frontend. Начнем с нескольких требований к API, основанных на потребностях UI. Время загрузки товаров для нашего приложения критично, поскольку чем дольше пользователь ждет, тем меньше шансов, что он продолжит просматривать сайт и что-то купит. Поэтому мы должны доставить минимально необходимые данные как можно скорее.

- API не должен ждать ответа от сервиса товаров дольше 1 с. Если за это время ответ не пришел, то мы должны вернуть ошибку тайм-аута (код состояния HTTP 504), чтобы UI не зависал.
- Данные о тележке и избранном факультативны. Если удастся получить их за 1 с, прекрасно! А если нет, ограничимся только сведениями о товаре.
- Данные о наличии складе тоже факультативны. Если не сумеем их получить, покажем только сведения о товаре.

При таких требованиях у нас есть несколько способов справиться с медленными или упавшими сервисами, а также с сетевыми проблемами. Поэтому наш сервис, а значит, и потребляющие его пользовательские интерфейсы оказываются более устойчивыми. Хотя пользователь не всегда получает все данные, их достаточно, чтобы сайт был полезен. Даже в случае катастрофического отказа сервиса товаров мы не оставим пользователя вечно наблюдать крутящийся индикатор занятости.

Далее решим, как должен выглядеть ответ. В верхней строке нам нужно только количество элементов в корзине и в списке избранного, поэтому ответ должен содержать скалярные значения. Поскольку

сервисы корзины и избранного могут завершаться по тайм-ауту или с ошибкой, мы допускаем, что это значение может быть равно `null`. Что до товаров, то нам нужны только обычные сведения о товаре, дополненные данными о наличии на складе, поэтому будем возвращать массив. Следовательно, ответ должен выглядеть следующим образом:

```
{
  "cart_items": 1,
  "favorite_items": null,
  "products": [{"product_id": 4, "inventory": 4},
               {"product_id": 3, "inventory": 65}]
}
```

В данном случае в корзине у пользователя только один товар. Избранные товары, возможно, и есть, но результат равен `null`, потому что при обращении к сервису избранного произошла какая-то ошибка. Наконец, мы имеем два товара, для которых складские остатки равны 4 и 65 соответственно.

С чего начать реализацию этой функциональности? Нам нужно взаимодействовать с нашими REST-сервисами по протоколу HTTP, поэтому естественным выбором будет клиентская часть библиотеки `aiohttp`, так как на ней же реализован и наш веб-сервер. Какие запросы мы отправляем, как их сгруппировать и как обрабатывать тайм-ауты? Во-первых, следует понять, какое максимальное число запросов мы будем отправлять конкурентно. Чем больше их будет, тем быстрее мы сможем вернуть ответ своим клиентам – теоретически. В нашем случае нельзя запрашивать наличие до получения идентификаторов товаров, поэтому конкурентно выполнять эти запросы нельзя, но сервисы товаров, корзины и избранного не зависят друг от друга. Это означает, что к ним можно обращаться конкурентно и ждать ответа с помощью функции `asyncio.wait`. Вызов `wait` с тайм-аутом даст нам множество `done`, на основании которого мы сможем узнать, какие запросы завершились с ошибкой, а какие еще работают после тайм-аута. Это позволит обработать ошибки любого вида. Затем, получив идентификаторы товаров и, возможно, данные о корзине и избранном, мы сможем объединить все в окончательный ответ и отправить его клиенту.

Создадим окончечную точку `/products/all`, которая будет возвращать описанные данные. Обычно мы хотим получать в URL-адресе, в заголовках запроса или в куки идентификатор текущего пользователя, чтобы его можно было включить в состав запросов к сервисам. Но здесь мы для простоты зашьем идентификатор того пользователя, для которого вставляли данные в базу.

Листинг 10.8 Сервис `backend-for-frontend` для товаров

```
import asyncio
from asyncio import Task
import aiohttp
```

```

from aiohttp import web, ClientSession
from aiohttp.web_request import Request
from aiohttp.web_response import Response
import logging
from typing import Dict, Set, Awaitable, Optional, List

routes = web.RouteTableDef()

PRODUCT_BASE = 'http://127.0.0.1:8000'
INVENTORY_BASE = 'http://127.0.0.1:8001'
FAVORITE_BASE = 'http://127.0.0.1:8002'
CART_BASE = 'http://127.0.0.1:8003'

@routes.get('/products/all')
async def all_products(request: Request) -> Response:
    async with aiohttp.ClientSession() as session:
        products = asyncio.create_task(session.get(f'{PRODUCT_BASE}/products'))
        favorites =
            asyncio.create_task(session.get(f'{FAVORITE_BASE}/users/3/favorites'))
        cart = asyncio.create_task(session.get(f'{CART_BASE}/users/3/cart'))

        requests = [products, favorites, cart]
        done, pending = await asyncio.wait(requests, timeout=1.0)

```

```

    if products in pending:
        [request.cancel() for request in requests]
        return web.json_response(
            {'error': 'Не удалось подключиться к сервису товаров.'},
            status=504)
    elif products in done and products.exception() is not None:
        [request.cancel() for request in requests]
        logging.exception('Ошибка сервера при подключении к сервису товаров.',
            exc_info=products.exception())
        return web.json_response(
            {'error': 'Ошибка сервера при подключении к сервису товаров.'},
            status=500)
    else:

```

```

        product_response = await products.result().json()
        product_results: List[Dict] = await get_products_with_inventory(session,
            product_response)

        cart_item_count: Optional[int] = await get_response_item_count(cart,
            done,
            pending,
            'Error getting user cart.')
        favorite_item_count: Optional[int] = await get_response_item_count(
            favorites,
            done,
            pending,
            'Error getting user favorites.')

```

Если запрос к сервису товаров завершился по тайм-ауту, вернуть ошибку, потому что продолжение невозможно

Извлечь данные из ответа на запрос о товарах и использовать их для получения информации о наличии

Создать задачи для опроса всех трех сервисов и запустить их конкурентно

←

```

        return web.json_response({'cart_items': cart_item_count,
                                  'favorite_items': favorite_item_count,
                                  'products': product_results})

async def get_products_with_inventory(session: ClientSession,
                                     product_response) -> List[Dict]:
    def get_inventory(session: ClientSession, product_id: str) -> Task:
        url = f"{INVENTORY_BASE}/products/{product_id}/inventory"
        return asyncio.create_task(session.get(url))

    def create_product_record(product_id: int, inventory: Optional[int]) -> Dict:
        return {'product_id': product_id, 'inventory': inventory}

    inventory_tasks_to_product_id = {
        get_inventory(session, product['product_id']): product['product_id']
        for product in product_response
    }

    inventory_done, inventory_pending = await
        asyncio.wait(inventory_tasks_to_product_id.keys(), timeout=1.0)

    product_results = []

    for done_task in inventory_done:
        if done_task.exception() is None:
            product_id = inventory_tasks_to_product_id[done_task]
            inventory = await done_task.result().json()
            product_results.append(create_product_record(product_id,
                                                         inventory['inventory']))
        else:
            product_id = inventory_tasks_to_product_id[done_task]
            product_results.append(create_product_record(product_id, None))
            logging.exception(
                f'Не удалось получить сведения о наличии товара {product_id}',
                exc_info=inventory_tasks_to_product_id[done_task].exception())

    for pending_task in inventory_pending:
        pending_task.cancel()
        product_id = inventory_tasks_to_product_id[pending_task]
        product_results.append(create_product_record(product_id, None))

    return product_results

async def get_response_item_count(task: Task,
                                  done: Set[Awaitable],
                                  pending: Set[Awaitable],
                                  error_msg: str) -> Optional[int]:
    if task in done and task.exception() is None:
        return len(await task.result().json())
    elif task in pending:
        task.cancel()
    else:
        logging.exception(error_msg, exc_info=task.exception())
        return None

```

Получив данные о товарах,
отправить запросы о наличии

Вспомогательная функция для получения
числа элементов в массиве JSON,
пришедшем в качестве ответа

```
app = web.Application()  
app.add_routes(routes)  
web.run_app(app, port=9000)
```

Здесь мы сначала определяем обработчик конечной точки `all_products`. В нем мы с помощью функции `wait` конкурентно отправляем запросы о товарах, корзине и избранном, давая на завершение 1 с. Как только все они завершатся или произойдет тайм-аут, мы начинаем обрабатывать результаты.

Поскольку время получения ответа о товарах критично, мы сначала проверяем состояние этого запроса. Если он еще выполняется или произошло исключение, то остальные запросы снимаются, а клиенту возвращается сообщение об ошибке. Если было исключение, то мы возвращаем код HTTP 500 – ошибка сервера. А в случае тайм-аута возвращаем код 504 – не удалось подключиться к сервису. Это различие оставляет клиенту возможность решить, стоит ли пробовать еще раз, а также дает больше информации для мониторинга и оповещения (например, можно завести извещатели, следящие за частотой ответов с кодом 504).

Если от сервиса товаров получен успешный ответ, то можно начать его обработку и запросить складские остатки. Это делается в функции `get_products_with_inventory`. Мы извлекаем идентификаторы товаров из тела ответа и на их основе конструируем запросы к сервису наличия на складе. Поскольку этот сервис принимает один идентификатор в одном запросе (в идеале хорошо бы объединить все идентификаторы в пакет, но будем считать, что команда, разрабатывающая сервис наличия, столкнулась с проблемами при реализации такого подхода), мы создадим список задач для запроса сведений о наличии каждого товара. И снова передадим их сопрограмме `wait`, дав 1 с на завершение.

Поскольку сведения о наличии на складе факультативны, по истечении тайм-аута мы начинаем обрабатывать множества `done` и `pending`. Если от сервиса наличия на складе получен успешный ответ, то мы создаем словарь, содержащий информацию о товаре, дополненную остатком на складе. Если имело место исключение или запрос все еще находится в множестве `pending`, то в записи словаря вместо остатка будет фигурировать значение `None`, означающее, что мы не смогли получить данные. На этапе преобразования ответ в формате JSON `None` будет заменено на `null`.

Наконец, мы проверяем ответы от сервисов корзины и избранного. От них нам нужно только одно число. Поскольку логика почти идентична, мы написали вспомогательную функцию подсчета элементов в ответе, `get_response_item_count`. Если сервис корзины или избранного завершился успешно, то мы получим JSON-массив, поэтому подсчитываем и возвращаем число элементов в нем. Если же произошло исключение или тайм-аут, то мы возвращаем результат `None`, который преобразуется в `null` в JSON-ответе.

Эта реализация дает достаточно надежный способ обработки ошибок и тайм-аутов не критичных сервисов и гарантирует, что разумный ответ будет быстро получен даже в случае ошибок нижележащих сервисов. Ни один запрос к сервису не занимает дольше 1 с, что дает приближенную верхнюю границу времени работы. Однако, хотя мы и создали нечто более-менее надежное, есть несколько способов, как еще улучшить устойчивость к ошибкам.

10.3.4 Повтор неудачных запросов

В первой реализации мы пессимистично предполагали, что если от сервиса получено исключение, то все пропало, результатов не будет и надо двигаться дальше. Иногда так оно и есть, но бывает, что ошибка сервиса случайна. Например, в сети мог возникнуть затор, который довольно быстро рассасывается, или возникла временная проблема у нашего балансировщика нагрузки – возможных причин вагон и маленькая тележка.

В таких случаях имеет смысл несколько раз повторить попытку с небольшой задержкой. Возможно, ошибка исчезнет сама собой и мы сможем предложить пользователю больше данных, чем при пессимистическом умонастроении. Конечно, пользователям придется подождать чуть дольше, и не исключено, что в итоге они увидят тот же самый результат.

Для реализации этой идеи отлично подходит сопрограмма `wait_for`. Она возбуждает полученное исключение и позволяет задать тайм-аут. Если тайм-аут истечет, то будет возбуждено исключение `TimeoutException` и запущенная задача снята. Попробуем на этой основе создать повторно используемую сопрограмму повтора `retry`, которая принимает сопрограмму и количество повторных попыток. Если переданная сопрограмма завершается из-за ошибки или в результате тайм-аута, то мы пытаемся повторить ее заданное число раз.

Листинг 10.9 Сопрограмма `retry`

```
import asyncio
import logging
from typing import Callable, Awaitable

class TooManyRetries(Exception):
    pass

async def retry(coro: Callable[[], Awaitable],
               max_retries: int,
               timeout: float,
               retry_interval: float):
    for retry_num in range(0, max_retries):
        try:
            return await asyncio.wait_for(coro(), timeout=timeout)
        except Exception as e:
            logging.exception(f'Во время ожидания произошло исключение (попытка {retry_num})), пробуем еще раз.', exc_info=e)
```

Если получено исключение, протоколировать его и ждать в течение заданного интервала перед повторной попыткой

Ждать ответа, пока не истечет заданный таймаут


```

        Если было слишком много неудачных попыток,
        возбудить исключение, уведомляющее об этом
    await asyncio.sleep(retry_interval)
    raise TooManyRetries()

```

Здесь мы сначала создаем специальный класс исключения, которое будем возбуждать, если даже после максимального числа попыток ошибка не исчезла. Вызывающая сторона сможет перехватить это исключение и обработать его как считает нужным. Сопрограмма `retry` принимает несколько аргументов. Первый – вызываемый объект, который возвращает объект, допускающий ожидание; это та сопрограмма, которую мы будем пытаться выполнить повторно. Второй – число попыток, а последние два – величина тайм-аута и интервал между неудачными попытками. Мы входим в цикл, обернутый сопрограммой `wait_for`, и, если она завершается успешно, то возвращаем результат и выходим из функции. Если же имела место ошибка, в том числе тайм-аут, то мы перехватываем исключение, протоколируем его и засыпаем на заданное время, после чего предпринимаем следующую попытку. Если мы вышли из цикла не по причине успешного вызова сопрограммы, то возбуждаем исключение `TooManyRetries`.

Для тестирования напомним две сопрограммы, завершающиеся ошибками, которые мы как раз и хотим обрабатывать. Первая всегда будет возбуждать исключение, а вторая не будет заканчиваться в отведенное время.

Листинг 10.10 Тестирование сопрограммы `retry`

```

import asyncio
from chapter_10.listing_10_9 import retry, TooManyRetries

async def main():
    async def always_fail():
        raise Exception("А я грохнулась!")

    async def always_timeout():
        await asyncio.sleep(1)

    try:
        await retry(always_fail,
                    max_retries=3,
                    timeout=.1,
                    retry_interval=.1)
    except TooManyRetries:
        print('Слишком много попыток!')

    try:
        await retry(always_timeout,
                    max_retries=3,
                    timeout=.1,
                    retry_interval=.1)
    except TooManyRetries:

```

```
print('Слишком много попыток!')

asyncio.run(main())
```

В обоих случаях мы задали тайм-аут и интервал между попытками, равными 100 мс, а максимальное число попыток равным трем. То есть мы даем сопрограмме 100 мс на завершение, а если она не успеет или завершится ошибкой, то через 100 мс пробуем снова. Запустив эту программу, вы увидите, что обе сопрограммы пытаются выполниться трижды, после чего сдаются с сообщением «Слишком много попыток!». В следующей распечатке трассы обратных вызовов для краткости опущены:

```
ERROR:root:Во время ожидания произошло исключение (попытка 1), пробую еще раз.
Exception: А я грохнулась!
ERROR:root:Во время ожидания произошло исключение (попытка 2), пробую еще раз.
Exception: А я грохнулась!
ERROR:root:Во время ожидания произошло исключение (попытка 3), пробую еще раз.
Exception: А я грохнулась!
Слишком много попыток!
ERROR:root:Во время ожидания произошло исключение (попытка 1), пробую еще раз.
ERROR:root:Во время ожидания произошло исключение (попытка 2), пробую еще раз.
ERROR:root:Во время ожидания произошло исключение (попытка 3), пробую еще раз.
Слишком много попыток!
```

Теперь мы можем добавить простую логику повтора в свой сервис типа backend-for-frontend. Например, предположим, что мы хотим несколько раз повторять запросы к сервисам товаров, корзины и избранного, прежде чем признать, что ошибка неисправима. Для этого нужно обернуть каждый запрос сопрограммой `retry`:

```
product_request = functools.partial(session.get, f'{PRODUCT_BASE}/products')
favorite_request = functools.partial(session.get,
    f'{FAVORITE_BASE}/users/5/favorites')
cart_request = functools.partial(session.get, f'{CART_BASE}/users/5/cart')
products = asyncio.create_task(retry(product_request,
    max_retries=3,
    timeout=.1,
    retry_interval=.1))

favorites = asyncio.create_task(retry(favorite_request,
    max_retries=3,
    timeout=.1,
    retry_interval=.1))

cart = asyncio.create_task(retry(cart_request,
    max_retries=3,
    timeout=.1,
    retry_interval=.1))

requests = [products, favorites, cart]
done, pending = await asyncio.wait(requests, timeout=1.0)
```

Здесь мы пытаемся обратиться к каждому сервису не более трех раз. Это даст возможность восстановиться после случайных ошибок. Конечно, это улучшение, но есть потенциальная проблема, которая может повредить нашему сервису. Например, что будет, если сервис товаров всегда дает тайм-аут?

10.3.5 Паттерн Прерыватель

В нашей реализации имеется проблема, которая проявляется, когда сервис работает настолько медленно, что всегда завершается по тайм-ауту. Это может случиться, когда нижележащий сервис испытывает высокую нагрузку, когда имеет место какая-то неисправность в сети или по целому ряду других причин.

Вы скажете: «Ну и что, приложение корректно обрабатывает тайм-аут, пользователь подождет не более секунды, а затем либо увидит сообщение об ошибке, либо получит частичные данные. Так в чем проблема?» Так-то оно так, но при проектировании надежной и отказоустойчивой системы следует принимать во внимание впечатления пользователя. Например, если сервис корзины по каким-то причинам никогда не успевает завершиться за 1 с, значит, все пользователи будут напрасно ждать результатов от него в течение этой секунды.

В данном случае, поскольку проблема, вызывающая затык сервиса, может сохраниться в течение продолжительного времени, всякий, кто имел несчастье обратиться к нашему сервису *backend-for-frontend*, будет ждать 1 с, хотя мы *знаем*, что проблема весьма вероятна. Нельзя ли как-нибудь обойти вызов, который с большой вероятностью завершится неудачно, чтобы не заставлять пользователей понапрасну ждать?

Паттерн, решающий эту проблему, носит красноречивое название *Прерыватель* (circuit breaker). Он снижал популярность после выхода книги Michael Nygard «Release It» (The Pragmatic Bookshelf, 2017). Паттерн позволяет «перещелкнуть прерыватель», если в течение определенного периода времени произошло заданное число ошибок, и таким образом обойти медленный сервис, пока проблема не разрешится, и продолжить отвечать на запросы пользователей с максимальной скоростью.

Как и у прерывателя электрических цепей, у паттерна Прерыватель есть два состояния: разомкнут и замкнут. В замкнутом состоянии все хорошо: мы отправляем запрос сервису, и тот успешно возвращает ответ. Разомкнутое состояние имеет место, когда цепь аварийно отключилась. В этом состоянии можно даже не трудиться вызывать сервис, потому что это заведомо не приведет к успеху, и мы вместо этого сразу возвращаем ошибку. Паттерн Прерыватель не дает подать ток неисправному сервису. Помимо этих двух состояний, есть еще состояние «полуразомкнуто». В него мы переходим после нахождения в разомкнутом состоянии в течение заданного времени. В этом состоянии отправляем одиночный запрос, чтобы проверить, была ли

исправлена ошибка. Если да, то прерыватель замыкается, а иначе остается разомкнутым. Чтобы не усложнять пример, оставим в покое полуразомкнутое состояние и рассмотрим только замкнутое и разомкнутое (рис. 10.3).

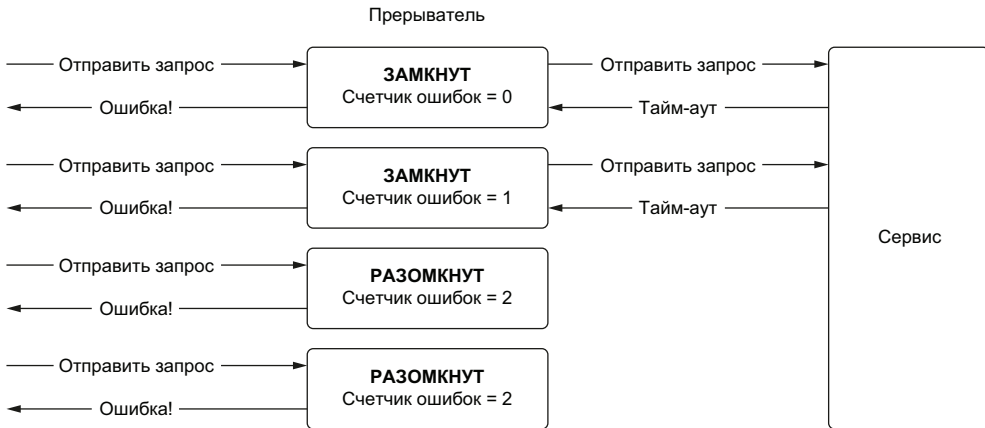


Рис. 10.3 Прерыватель, размыкающийся после двух ошибок. Когда он разомкнут, все запросы немедленно завершаются ошибкой

Реализуем простой прерыватель, чтобы понять, как он работает. Мы дадим пользователям прерывателя возможность задавать интервал времени и максимальное число ошибок. Если за указанный интервал произошло больше ошибок, то мы размыкаем прерыватель и в ответ на последующие обращения возвращаем ошибку. Класс прерывателя принимает сопрограмму, которую требуется выполнить, и следит за тем, в каком состоянии мы находимся: замкнутым или разомкнутым.

Листинг 10.11 Простой прерыватель

```
import asyncio
from datetime import datetime, timedelta

class CircuitOpenException(Exception):
    pass

class CircuitBreaker:

    def __init__(self,
                 callback,
                 timeout: float,
                 time_window: float,
                 max_failures: int,
                 reset_interval: float):
        self.callback = callback
        self.timeout = timeout
        self.time_window = time_window
```

```

self.max_failures = max_failures
self.reset_interval = reset_interval
self.last_request_time = None
self.last_failure_time = None
self.current_failures = 0

async def request(self, *args, **kwargs):
    if self.current_failures >= self.max_failures:
        if datetime.now() > self.last_request_time +
            timedelta(seconds=self.reset_interval):
            self._reset('Цепь переходит из разомкнутого состояния
в замкнутое, сброс!')
            return await self._do_request(*args, **kwargs)
        else:
            print('Цепь разомкнута, быстрый отказ!')
            raise CircuitOpenException()
    else:
        if self.last_failure_time and datetime.now() >
            self.last_failure_time + timedelta(seconds=self.time_window):
            self._reset('Interval since first failure elapsed, resetting!')
            print('Цепь замкнута, отправляю запрос!')
            return await self._do_request(*args, **kwargs)

def _reset(self, msg: str):
    print(msg)
    self.last_failure_time = None
    self.current_failures = 0

async def _do_request(self, *args, **kwargs):
    try:
        print('Отправляется запрос!')
        self.last_request_time = datetime.now()
        return await asyncio.wait_for(self.callback(*args, **kwargs),
            timeout=self.timeout)
    except Exception as e:
        self.current_failures = self.current_failures + 1
        if self.last_failure_time is None:
            self.last_failure_time = datetime.now()
        raise

```

Отправить запрос или сразу вернуть ошибку, если превышен счетчик ошибок

Сбросить счетчик и время последнего отказа

Отправить запрос и следить за тем, сколько было ошибок и когда имела место последняя ошибка

Конструктор нашего класса прерывателя принимает пять параметров. Первые два – обратный вызов, который должен выполнять прерыватель, и величина тайм-аута, по истечении которого обратный вызов признается неудачным. Следующие три параметра связаны с обработкой ошибок и сбросом в исходное состояние. Параметр `max_failure` – максимальное число ошибок, допустимое в течение промежутка времени `time_window`, прежде чем прерыватель разомкнется. Параметр `reset_interval` определяет, сколько секунд ждать перед сбросом прерывателя из разомкнутого состояния в замкнутое, после того как произошло `max_failure` ошибок.

Далее определен метод-сопрограмма `request`, который выполняет обратный вызов, следит за количеством возникших ошибок и возвра-

щает полученный результат, если ошибок не было. При возникновении ошибки мы увеличиваем счетчик `failure_count`. Если в течение заданного промежутка времени произошло больше `max_failure` ошибок, то последующие обращения к `request` будут возбуждать исключение `CircuitOpenException`. По прошествии времени `reset_interval` счетчик `failure_count` сбрасывается в нуль и подсчет ошибочных запросов начинается заново (если прерыватель замкнут, что, вообще говоря, необязательно).

Теперь рассмотрим применение прерывателя в простом примере. Напишем сопрограмму `slow_callback`, которая будет спать 2 с. Затем передадим ее прерывателю, установив короткий тайм-аут, при котором прерыватель обязательно сработает.

Листинг 10.12 Прерыватель в действии

```
import asyncio
from chapter_10.listing_10_11 import CircuitBreaker

async def main():
    async def slow_callback():
        await asyncio.sleep(2)

    cb = CircuitBreaker(slow_callback,
                        timeout=1.0,
                        time_window=5,
                        max_failures=2,
                        reset_interval=5)

    for _ in range(4):
        try:
            await cb.request()
        except Exception as e:
            pass

    print('Засыпаю на 5 с, чтобы прерыватель замкнулся...')
    await asyncio.sleep(5)

    for _ in range(4):
        try:
            await cb.request()
        except Exception as e:
            pass

asyncio.run(main())
```

Здесь мы создаем прерыватель с односекундным тайм-аутом, который допускает две ошибки в течение 5 с и сбрасывается после 5 с нахождения в разомкнутом состоянии. Далее быстро отправляем прерывателю четыре запроса. Первые два будут ждать по одной секунде, прежде чем завершиться по тайм-ауту, и последующие завершаются мгновенно, потому что прерыватель уже разомкнут. Затем мы спим

5 с; поскольку этого достаточно, чтобы истек интервал `reset_interval`, то мы должны вернуться в замкнутое состояние и снова начать отправку запросов. При выполнении программы мы увидим следующую картину:

```
Цепь замкнута, отправляю запрос!
Цепь замкнута, отправляю запрос!
Цепь разомкнута, быстрый отказ!
Цепь разомкнута, быстрый отказ!
Засыпаю на 5 с, чтобы прерыватель замкнулся...
Цепь переходит из разомкнутого в замкнутое состояние, сброс!
Цепь замкнута, отправляю запрос!
Цепь разомкнута, быстрый отказ!
Цепь разомкнута, быстрый отказ!
```

Эту простую реализацию мы можем объединить с логикой повторных попыток и включить в свой сервис `backend-for-frontend`. Поскольку мы намеренно сделали сервис наличия на складе медленным, чтобы имитировать унаследованный сервис (что часто встречается на практике), именно сюда было бы естественно поместить прерыватель. Зададим тайм-аут 500 мс и разрешим не более пяти ошибок в течение одной секунды. Время до сброса зададим равным 30 с. Функцию `get_inventory` придется переписать, сделав сопрограммой:

```
async def get_inventory(session: ClientSession, product_id: str):
    url = f"{INVENTORY_BASE}/products/{product_id}/inventory"
    return await session.get(url)

inventory_circuit = CircuitBreaker(get_inventory, timeout=.5, time_window=5.0,
    max_failures=3, reset_interval=30)
```

А в сопрограмме `all_products` нужно по-другому отправлять запросы сервису наличия на складе. Мы создадим задачу, которая обращается к прерывателю сервиса наличия, а не к сопрограмме `get_inventory`:

```
inventory_tasks_to_pid = {
    asyncio.create_task(inventory_circuit.request(session,
        product['product_id'])): product['product_id']
    for product in product_response
}

inventory_done, inventory_pending = await
    asyncio.wait(inventory_tasks_to_pid.keys(), timeout=1.0)
```

После внесения этих изменений вы увидите, что время реакции сервиса типа `backend-for-frontend` уменьшается после нескольких вызовов. Поскольку мы смоделировали медленный сервис наличия на складе, спустя несколько тайм-аутов сработает прерыватель, после чего запросы к сервису наличия перестанут отправляться, до тех пор пока прерыватель не сбросится в исходное состояние. Теперь наш

агрегирующий сервис стал более устойчивым к ошибкам медленного и ненадежного сервиса наличия на складе. При желании такой же подход можно было бы применить и к вызовам других сервисов.

Мы реализовали очень простой прерыватель для демонстрации принципа его работы и применения `asyncio`. Существует несколько готовых реализаций этого паттерна, предлагающих много дополнительных возможностей настройки. Если он вас заинтересовал, то почитайте об имеющихся библиотеках, прежде чем писать собственную.

Резюме

- Микросервисы предлагают ряд преимуществ перед монолитами, в том числе независимые масштабирование и развертывание.
- Паттерн `backend-for-frontend` агрегирует результаты нескольких нижележащих микросервисов. Мы научились применять микросервисную архитектуру к примеру интернет-магазина и создали несколько независимых сервисов на основе `aiohttp`.
- Мы использовали служебные функции `asyncio`, в частности `wait`, чтобы обеспечить устойчивость агрегирующего сервиса к ошибкам в нижележащих сервисах, не жертвуя отзывчивостью на действия пользователей.
- Мы написали средство для управления повторами HTTP-запросов на основе `asyncio` и `aiohttp`.
- Мы написали простую реализацию паттерна Прерыватель, который гарантирует, что ошибка в одном сервисе не окажет негативного влияния на другие.

11

Синхронизация

Краткое содержание главы

- Проблемы однопоточной конкурентности.
- Использование блокировок для защиты критических секций.
- Использование семафоров для ограничения уровня конкурентности.
- Использование событий для уведомления задач.
- Использование условий для уведомления задач и захвата ресурса.

При написании приложений с несколькими потоками или процессами нужно помнить о возможности состояния гонки при использовании неатомарных операций. Даже простое увеличение целого числа на единицу в конкурентной программе может вызвать тонкие ошибки, с трудом поддающиеся воспроизведению. Но при использовании *asynсio* мы всегда работаем в одном потоке (если только явно не задействовали средства многопоточной или многопроцессной обработки), значит, можно не беспокоиться о гонках, правда? На самом деле все не так просто.

Да действительно, некоторые ошибки, встречающиеся в многопоточных и многопроцессных приложениях, исключены в силу однопоточной природы *asynсio*. Исключены, да не совсем. Вероятно, вам не

потребуется часто прибегать к синхронизации при работе с `asyncio`, однако все равно остаются ситуации, когда эти конструкции необходимы. *Примитивы синхронизации* `asyncio` могут помочь предотвратить ошибки, свойственные только модели однопоточной конкурентности.

Применение примитивов синхронизации не ограничивается предотвращением ошибок вследствие конкурентности. Например, мы можем работать с API, которому по контракту с поставщиком разрешено конкурентно отправлять лишь небольшое число запросов. Или с API, который нужно защитить от затопления запросами. Или взять технологический процесс, когда нескольких исполнителей необходимо уведомлять о появлении новых данных.

В этой главе мы рассмотрим ряд примеров, где могут появиться состояния гонки в коде на основе `asyncio`, и научимся предотвращать их с помощью блокировок и других примитивов синхронизации. Мы также узнаем, как ограничить уровень конкурентности с помощью семафоров и как управлять доступом к разделяемому ресурсу, например пулу подключений к базе данных. Наконец, рассмотрим события и условия – механизмы, позволяющие уведомлять задачи о том, что произошло нечто интересное, и получать доступ к разделяемому ресурсу, когда это случилось.

11.1 Природа ошибок в модели однопоточной конкурентности

Напомним, что при работе с данными, разделяемыми между несколькими процессами или потоками, нам необходимо помнить о возможности состояний гонки. Это связано с тем, что поток или процесс могут читать данные, которые одновременно модифицирует другой поток или процесс, что ведет к несогласованному состоянию и, как следствие, к повреждению данных.

Частично это повреждение объясняется тем, что некоторые операции неатомарны, т. е., хотя выглядят как одна операция, на самом деле состоят из нескольких отдельных операций. В примере из главы 6 мы имели дело с инкрементированием целой переменной; сначала читается текущее значение, затем оно увеличивается, и напоследок новое значение записывается в переменную. Это открывает другим потокам и процессам реальную возможность получить несогласованное состояние данных.

В модели однопоточной конкурентности мы избегаем состояний гонки, вызванных неатомарными операциями. В `asyncio` есть только один поток, который в каждый момент времени исполняет одну строку кода Python. Это означает, что, даже если операция неатомарна, она все равно будет доведена до конца и другие сопрограммы не смогут прочитать несогласованные данные.

Чтобы убедиться в этом, попробуем воспроизвести состояние гонки, которое встречали в главе 7, когда несколько потоков пытались реализовать разделяемый счетчик. Только вместо нескольких потоков модифицировать переменную будут несколько задач. Повторим операцию 1000 раз и проверим, что получено ожидаемое значение.

Листинг 11.1 Попытка создать состояние гонки

```
import asyncio

counter: int = 0

async def increment():
    global counter
    await asyncio.sleep(0.01)
    counter = counter + 1

async def main():
    global counter
    for _ in range(1000):
        tasks = [asyncio.create_task(increment()) for _ in range(100)]
        await asyncio.gather(*tasks)
        print(f'Счетчик равен {counter}')
        assert counter == 100
        counter = 0

asyncio.run(main())
```

Здесь мы написали сопрограмму `increment`, которая прибавляет единицу к глобальному счетчику и имитирует длительную операцию с помощью задержки на 1 мс. В сопрограмме `main` мы создаем 100 задач, инкрементирующих счетчик, и запускаем их конкурентно с помощью `gather`. После чего проверяем, что счетчик имеет ожидаемое значение – 100, поскольку было выполнено 100 задач инкрементирования. При выполнении этой программы мы всегда должны видеть значение 100, хотя операция инкрементирования целого числа и не атомарна. Если бы вместо сопрограмм мы запускали потоки, то в какой-то момент утверждение `assert` оказалось бы ложным.

Означает ли это, что модель однопоточной конкурентности подарила нам способ раз и навсегда избавиться от гонок? К сожалению, нет. Хотя мы избежали гонки там, где одна неатомарная операция могла бы привести к ошибке, осталась проблема неправильного порядка выполнения нескольких операций. Чтобы посмотреть, как это бывает, сделаем операцию инкрементирования целого числа неатомарной, на взгляд `asyncio`.

Для этого мы воспроизведем то, что происходит под капотом при инкрементировании глобального счетчика: чтение, увеличение и запись на старое место. Идея в том, что если какой-то другой код модифицирует состояние, пока наша сопрограмма приостановлена в `await` в ожидании завершения другого `await`, то мы можем получить несогласованное состояние.

Листинг 11.2 Состояние гонки в однопоточной программе

```
import asyncio

counter: int = 0

async def increment():
    global counter
    temp_counter = counter
    temp_counter = temp_counter + 1
    await asyncio.sleep(0.01)
    counter = temp_counter

async def main():
    global counter
    for _ in range(1000):
        tasks = [asyncio.create_task(increment()) for _ in range(100)]
        await asyncio.gather(*tasks)
        print(f'Счетчик равен {counter}')
        assert counter == 100
        counter = 0

asyncio.run(main())
```

Теперь сопрограмма `increment` не инкрементирует счетчик непосредственно, а сначала читает его во временную переменную, после чего прибавляет к этой переменной 1. Затем мы выполняем `await asyncio.sleep`, имитируя медленную операцию, которая приостанавливает нашу сопрограмму, и только потом записываем значение из временной переменной в глобальную переменную `counter`. Выполнив эту программу, вы увидите, что утверждение сразу же оказывается ложным и счетчик успевает добраться только до 1! Каждая сопрограмма сначала читает значение, равное 0, сохраняет его в переменной `temp`, а затем засыпает. Поскольку поток всего один, все операции чтения временной переменной выполняются последовательно, т. е. каждая сопрограмма сохраняет значение счетчика 0 и увеличивает его до 1. Затем, как только сон завершится, каждая сопрограмма записывает в счетчик значение 1, т. е. несмотря на то, что работает 100 сопрограмм, увеличивающих счетчик, новым его значением будет 1. Заметим, что если убрать выражение `await`, то порядок выполнения будет правильный, потому что исчезла возможность изменить состояние приложения, пока сопрограмма приостановлена в точке `await`.

Конечно, это упрощенный и нереалистичный пример. Чтобы лучше понять, когда нечто подобное может произойти, сконструируем чуть более сложное состояние гонки. Допустим, мы реализуем сервер, который отправляет сообщения подключившимся пользователям. Сервер хранит словарь, отображающий имя пользователя на сокет, в который отправлять адресованные ему сообщения. Когда пользователь отключается, выполняется обратный вызов, который удаляет поль-

зователя из словаря и закрывает его сокет. Поскольку мы закрываем сокет при отключении, любая попытка отправить в него сообщение приведет к исключению. А что будет, если пользователь отключится в том момент, когда мы отправляем ему сообщение? Предположим, что мы хотим, чтобы пользователь получил сообщение, если он был подключен в момент начала его отправки.

Для тестирования реализуем класс, имитирующий сокет. Он будет иметь метод-сопрограмму `send` и обычный метод `close`. Метод `send` имитирует отправку сообщения по медленной сети. Он также проверяет флаг, показывающий, был ли закрыт сокет, и если был, то возбуждает исключение.

Создадим словарь, содержащий несколько подключенных пользователей, и для каждого из них создадим имитацию сокета. Отправим всем пользователям сообщения и вручную инициируем отключение одного пользователя в момент отправки ему сообщения. Посмотрим, что из этого выйдет.

Листинг 11.3 Состояние гонки с участием словарей

```
import asyncio

class MockSocket:
    def __init__(self):
        self.socket_closed = False

    async def send(self, msg: str):
        if self.socket_closed:
            raise Exception('Сокет закрыт!')
        print(f'Отправляется: {msg}')
        await asyncio.sleep(1)
        print(f'Отправлено: {msg}')

    def close(self):
        self.socket_closed = True

user_names_to_sockets = {'John': MockSocket(),
                        'Terry': MockSocket(),
                        'Graham': MockSocket(),
                        'Eric': MockSocket()}

async def user_disconnect(username: str):
    print(f'{username} отключен!')
    socket = user_names_to_sockets.pop(username)
    socket.close()

async def message_all_users():
    print('Создаются задачи отправки сообщений')
    messages = [socket.send(f'Привет, {user}')
                 for user, socket in user_names_to_sockets.items()]
    await asyncio.gather(*messages)
```

Имитировать медленную отправку сообщения клиенту

Отключить пользователя и удалить его из памяти приложения

Отправить сообщения всем пользователям конкурентно

```
async def main():
    await asyncio.gather(message_all_users(), user_disconnect('Eric'))

asyncio.run(main())
```

При выполнении этой программы приложение «падает»:

```
Создаются задачи отправки сообщений
Eric отключен!
Отправляется: Привет, John
Отправляется: Привет, Terry
Отправляется: Привет, Graham
Traceback (most recent call last):
  File 'chapter_11/listing_11_3.py', line 45, in <module>
    asyncio.run(main())
  File "asyncio/runners.py", line 44, in run
    return loop.run_until_complete(main)
  File "python3.9/asyncio/base_events.py", line 642, in run_until_complete
    return future.result()
  File 'chapter_11/listing_11_3.py', line 42, in main
    await asyncio.gather(message_all_users(), user_disconnect('Eric'))
  File 'chapter_11/listing_11_3.py', line 37, in message_all_users
    await asyncio.gather(*messages)
  File 'chapter_11/listing_11_3.py', line 11, in send
    raise Exception('Сокет закрыт!')
Exception: Сокет закрыт!
```

Здесь мы сначала создаем задачи отправки сообщений, а затем выполняем предложение `await`, приостанавливающее сопрограмму `message_all_users`. Это дает шанс выполниться сопрограмме `user_disconnect('Eric')`, которая закрывает сокет Эрика и удаляет его из словаря `user_names_to_sockets`. Затем сопрограмма `message_all_users` возобновляется, и мы начинаем рассылать сообщения. Так как сокет Эрика уже закрыт, мы имеем исключение, а Эрик не получит адресованного ему сообщения. Отметим, что мы также модифицировали словарь `user_names_to_sockets`. Если бы мы использовали его, рассчитывая, что Эрик все еще там, то могли бы нарваться на исключение или еще какую-нибудь ошибку.

Именно такие типы ошибок можно встретить в модели однопоточной конкурентности. Поток выполнения упирается в точку приостановки `await`, после чего начинает работать другая сопрограмма и модифицирует некоторое разделяемое состояние, так что первая сопрограмма после возобновления столкнется с неожиданностью. Ключевое различие между ошибками многопоточной и однопоточной конкурентности заключается в том, что в многопоточном приложении состояния гонки возможны везде, где модифицируется разделяемое состояние. А в модели однопоточной конкурентности нужно модифицировать состояние в точке `await`. Теперь, понимая, какие ошибки возможны, посмотрим, как их избежать с помощью блокировок `asyncio`.

11.2 Блокировки

Блокировки `asyncio` работают так же, как блокировки в модулях, обеспечивающих многопоточность и многопроцессность. Мы захватываем блокировку, делаем что-то внутри критической секции, а по завершении освобождаем блокировку, давая возможность захватить ее другим заинтересованным сторонам. Главное отличие заключается в том, блокировки `asyncio` – объекты, допускающие ожидание, которые приостанавливают выполнение сопрограммы, когда заблокированы. Это значит, что если сопрограмма ожидает освобождения блокировки, то может работать другой код. Кроме того, блокировки `asyncio` являются асинхронными контекстными менеджерами, и предпочтительно использовать их в сочетании с конструкцией `async with`.

Чтобы ближе познакомиться с работой блокировок, рассмотрим простой пример – одну блокировку, разделяемую двумя сопрограммами. Мы захватываем блокировку, что не дает другим сопрограммам выполнить код в критической секции, пока мы ее не освободим.

Листинг 11.4 Использование блокировки `asyncio`

```
import asyncio
from asyncio import Lock
from util import delay

async def a(lock: Lock):
    print('Сопрограмма а ждет возможности захватить блокировку')
    async with lock:
        print('Сопрограмма а находится в критической секции')
        await delay(2)
    print('Сопрограмма а освободила блокировку')

async def b(lock: Lock):
    print('Сопрограмма б ждет возможности захватить блокировку')
    async with lock:
        print('Сопрограмма б находится в критической секции')
        await delay(2)
    print('Сопрограмма б освободила блокировку')

async def main():
    lock = Lock()
    await asyncio.gather(a(lock), b(lock))

asyncio.run(main())
```

При выполнении этой программы мы увидим, что сопрограмма а первой захватывает блокировку, заставляя сопрограмму б ждать ее освобождения. После того как а освободит блокировку, б сможет сделать все, что ей надо, в критической секции, так что мы будем наблюдать следующую картину:

```

Сопрограмма а ждет возможности захватить блокировку
Сопрограмма а находится в критической секции
засыпаю на 2 с
Сопрограмма б ждет возможности захватить блокировку
сон в течение 2 с закончился
Сопрограмма а освободила блокировку
Сопрограмма б находится в критической секции
засыпаю на 2 с
сон в течение 2 с закончился
Сопрограмма б освободила блокировку
Здесь мы использовали конструкцию async with. При желании можно было бы
использовать методы acquire и release:
await lock.acquire()
try:
    print('В критической секции')
finally:
    lock.release()

```

Но лучше все же использовать `async with` всюду, где возможно.

Важно отметить, что мы создали блокировку внутри сопрограммы `main`. Поскольку блокировка разделяется всеми нашими сопрограммами, возникает соблазн сделать ее глобальной переменной, а не передавать в параметре:

```

lock = Lock()

# определения сопрограмм

async def main():
    await asyncio.gather(a(), b())

```

Но если поддасться этому соблазну, то программа почти сразу упадет с сообщением о наличии нескольких циклов событий:

```

Task <Task pending name='Task-3' coro=<b()> got Future <Future pending>
attached to a different loop

```

Почему же так происходит – ведь мы всего лишь перенесли определение блокировки из одного места в другое? Это особенность библиотеки `asyncio` и свойственна она не только блокировкам. У большинства объектов в `asyncio` есть факультативный параметр `loop`, который позволяет указать, в каком цикле событий объект работает. Если этот параметр не задан, то `asyncio` пытается получить текущий цикл событий, а если такового не существует, то создает новый. В нашем случае создание `Lock` приводит к созданию нового цикла событий, потому что в самом начале выполнения скрипта цикла еще нет. Затем `asyncio.run(main())` создает второй цикл событий, и при попытке использовать блокировку два разных цикла событий вступают в конфликт, что влечет за собой крах.

Это поведение настолько неожиданно, что в версии Python 3.10 параметр, задающий цикл событий, предполагается убрать, но до тех

пор нужно относиться к использованию глобальных объектов `asyncio` очень осторожно.

Познакомившись с теоретическими основами, посмотрим, как использовать блокировку для исправления ошибки в листинге 11.3, где мы пытались отправить сообщение пользователю, чей сокет закрыли слишком рано. Идея в том, чтобы использовать блокировку в двух местах: когда пользователь отключается и когда мы рассылаем пользователям сообщения. Таким образом, если отключение происходит в момент, когда мы рассылаем сообщения, то придется подождать, пока рассылка закончится, и только потом довершить закрытие сокетов.

Листинг 11.5 Использование блокировок с целью избежать состояния гонки

```
import asyncio
from asyncio import Lock

class MockSocket:
    def __init__(self):
        self.socket_closed = False

    async def send(self, msg: str):
        if self.socket_closed:
            raise Exception('Сокет закрыт!')
        print(f'Отправляется: {msg}')
        await asyncio.sleep(1)
        print(f'Отправлено: {msg}')

    def close(self):
        self.socket_closed = True

user_names_to_sockets = {'John': MockSocket(),
                        'Terry': MockSocket(),
                        'Graham': MockSocket(),
                        'Eric': MockSocket()}

async def user_disconnect(username: str, user_lock: Lock):
    print(f'{username} отключен!')
    async with user_lock:
        print(f'{username} удаляется из словаря')
        socket = user_names_to_sockets.pop(username)
        socket.close()
```

Захватить блокировку, перед тем как удалять пользователя и закрывать сокет

```

async def message_all_users(user_lock: Lock):
    print('Создаются задачи отправки сообщений')
    async with user_lock:
        messages = [socket.send(f'Привет, {user}')
                    for user, socket in
                    user_names_to_sockets.items()]
        await asyncio.gather(*messages)
```

Захватить блокировку, перед тем как рассылать сообщения

```
async def main():
    user_lock = Lock()
    await asyncio.gather(message_all_users(user_lock),
                          user_disconnect('Eric', user_lock))

asyncio.run(main())
```

При выполнении этой программы краха больше не будет, и мы увидим следующую картину:

```
Создаются задачи отправки сообщений
Eric отключен!
Отправляется: Привет, John
Отправляется: Привет, Terry
Отправляется: Привет, Graham
Отправляется: Привет, Eric
Отправлено: Привет, John
Отправлено: Привет, Terry
Отправлено: Привет, Graham
Отправлено: Привет, Eric
Eric удаляется из словаря
```

Сначала мы захватываем блокировку и создаем задачи для рассылки сообщений. Пока мы этим заняты, Эрик отключается, а код в сопрограмме `user_disconnect` пытается захватить блокировку. Поскольку `message_all_users` еще удерживает ее, придется подождать освобождения, прежде чем мы сможем отключить пользователя. Эта дает возможность закончить рассылку до закрытия сокета и тем самым предотвратить ошибку.

Маловероятно, что вам часто понадобится использовать блокировки в коде на основе `asyncio`, потому что многие проблемы вообще не возникают в силу однопоточности модели. Но даже если состояния гонки имеют место, иногда удастся переработать код, так чтобы состояние не модифицировалось в момент, когда сопрограмма приостановлена (например, использовать неизменяемые объекты). Если это невозможно, то блокировки хотя бы помогут гарантировать, что модификации производятся в желаемом синхронизированном порядке. Разобравшись, как избежать ошибок, связанных с конкурентностью, с помощью блокировок посмотрим, как примитивы синхронизации позволяют реализовать новую функциональность в приложениях на основе `asyncio`.

11.3 Ограничение уровня конкурентности с помощью семафоров

Часто ресурсы, необходимые приложению, конечны. Например, число конкурентных подключений к базе данных может быть ограничено. Ограничено и количество процессоров, которые мы не хотим подвер-

гать перегрузке. Наконец, можно работать с API, который допускает лишь небольшое число конкурентных запросов в зависимости от оплаченной подписки. Или мы используем собственный внутренний API и не хотим чрезмерно перегружать его, что было бы эквивалентно распределенной DoS-атаке против самих себя.

Семафоры как раз и спасают в таких ситуациях. Семафор похож на блокировку в том смысле, что его можно захватывать и освобождать, а основное отличие заключается в том, что захватить семафор можно не один раз, а несколько, – максимальное число задаем мы сами. Под капотом семафор следит за этим пределом; при каждом захвате предел уменьшается, а при каждом освобождении увеличивается. Как только счетчик обращается в нуль, дальнейшие попытки захватить семафор блокируются, пока кто-то не выполнит операцию освобождения, которая увеличит счетчик. Можно считать, что блокировка – частный случай семафора с пределом 1.

Для демонстрации рассмотрим простой пример, в котором одновременно должно работать не более двух задач, а всего их четыре. Создадим семафор с пределом 2 и захватим его в нашей сопрограмме.

Листинг 11.6 Использование семафоров

```
import asyncio
from asyncio import Semaphore

async def operation(semaphore: Semaphore):
    print('Жду возможности захватить семафор...')
    async with semaphore:
        print('Семафор захвачен!')
        await asyncio.sleep(2)
        print('Семафор освобожден!')

async def main():
    semaphore = Semaphore(2)
    await asyncio.gather(*[operation(semaphore) for _ in range(4)])

asyncio.run(main())
```

В сопрограмме `main` мы создаем семафор с пределом 2. Это означает, что мы сможем захватить его дважды, после чего дальнейшие попытки начнут блокироваться. Затем мы четыре раза конкурентно вызываем сопрограмму `operation` – она захватывает семафор в блоке `async with` и с помощью `sleep` имитирует блокирующую операцию. При выполнении этой программы мы увидим следующую картину:

```
Жду возможности захватить семафор...
Семафор захвачен!
Жду возможности захватить семафор...
Семафор захвачен!
Жду возможности захватить семафор...
Жду возможности захватить семафор...
```

Семафор освобожден!
Семафор освобожден!
Семафор захвачен!
Семафор захвачен!
Семафор освобожден!
Семафор освобожден!

Поскольку семафор допускает только два захвата перед блокировкой, первые две задачи благополучно захватят его, а две оставшиеся должны будут ждать освобождения семафора. После того как первые две задачи завершат работу и освободят семафор, две другие смогут захватить его и приступить к работе.

Применим этот паттерн к реалистичному сценарию. Допустим, вы работаете в испытывающем финансовые затруднения стартапе, только что заключившем партнерское соглашение со сторонним поставщиком REST API. Контракт с неограниченным числом запросов стоит очень дорого, но есть доступный для вашего бюджета тариф, предполагающий не более 10 конкурентных запросов. Если вы попытаетесь отправить больше 10 запросов одновременно, то API вернет код состояния 429 (слишком много запросов). Можно, конечно, отправить все запросы и повторить попытку, если получен код 429, но это неэффективно и дополнительно нагружает серверы поставщика, что, скорее всего, не обрадует инженеров, отвечающих за надежность сайта. Лучше создать семафор с пределом 10 и захватывать его всякий раз, как нужно отправить запрос API. Тогда гарантируется, что в каждый момент времени в работе будет не более 10 запросов.

Посмотрим, как это сделать с помощью библиотеки `aiohttp`. Мы отправим 1000 запросов модельному API, но с помощью семафора ограничим число конкурентных запросов десятью. Отметим, что у самой `aiohttp` тоже есть настраиваемый лимит, – по умолчанию одновременно можно открыть не более 100 подключений. Того же результата можно достичь путем изменения этого лимита.

Листинг 11.7 Ограничение числа запросов к API с помощью семафора

```
import asyncio
from asyncio import Semaphore
from aiohttp import ClientSession

async def get_url(url: str,
                  session: ClientSession,
                  semaphore: Semaphore):
    print('Жду возможности захватить семафор...')
    async with semaphore:
        print('Семафор захвачен, отправляется запрос...')
        response = await session.get(url)
        print('Запрос завершен')
        return response.status
```

```
async def main():
    semaphore = Semaphore(10)
    async with ClientSession() as session:
        tasks = [get_url('https://www.example.com', session, semaphore)
                  for _ in range(1000)]
        await asyncio.gather(*tasks)

asyncio.run(main())
```

Из-за фактора внешней задержки результат не детерминирован, но в целом картина выглядит так:

```
Семафор захвачен, отправляется запрос...
Семафор захвачен, отправляется запрос...
Семафор захвачен, отправляется запрос...
Семафор захвачен, отправляется запрос...
Семафор захвачен, отправляется запрос...
Запрос завершен
Запрос завершен
Семафор захвачен, отправляется запрос...
Семафор захвачен, отправляется запрос...
```

После каждого завершения запроса семафор освобождается, а значит, задача, заблокированная в ожидании семафора, может приступить к работе. То есть в каждый момент времени активно будет не более 10 запросов.

Это решает проблему слишком большого числа конкурентных запросов, но теперь код демонстрирует *пульсирующую* нагрузку, т. е. запросы могут отправляться пачками по 10, создавая пики трафика. Это может оказаться нежелательно, если пиков нагрузки на вызываемый API хотелось бы избежать. Если требуется ограничить всплески определенным числом запросов в единицу времени, то следует воспользоваться каким-нибудь алгоритмом формирования трафика, например «дырявым ведром» или «корзиной маркеров».

11.3.1 Ограниченные семафоры

Одна из особенностей семафоров заключается в том, что число вызовов метода `release` может превышать число вызовов `acquire`. Если мы всегда используем семафоры в сочетании с блоком `async with`, то такое невозможно, потому что с каждым `acquire` автоматически связывается `release`. Но если нам требуется более точный контроль над механизмом захвата и освобождения (например, имеется ветвящийся код, в одной ветви которого освобождение производится раньше, чем в другой), то возможны проблемы. Например, посмотрим, что будет, если имеется обычная сопрограмма, в которой захват и освобождение семафора производятся в блоке `async with`, но, пока эта сопрограмма выполняется, другая сопрограмма вызывает `release`.

Листинг 11.8 Освобождений больше, чем захватов

```

import asyncio
from asyncio import Semaphore

async def acquire(semaphore: Semaphore):
    print('Ожидание возможности захвата')
    async with semaphore:
        print('Захвачен')
        await asyncio.sleep(5)
    print('Освобождается')

async def release(semaphore: Semaphore):
    print('Одиночное освобождение!')
    semaphore.release()
    print('Одиночное освобождение - готово!')

async def main():
    semaphore = Semaphore(2)

    print("Два захвата, три освобождения...")
    await asyncio.gather(acquire(semaphore),
                        acquire(semaphore),
                        release(semaphore))

    print("Три захвата...")
    await asyncio.gather(acquire(semaphore),
                        acquire(semaphore),
                        acquire(semaphore))

asyncio.run(main())

```

Здесь мы создаем семафор с пределом 2. Затем дважды вызываем сопрограмму `acquire` и один раз `release`, т. е. всего семафор будет освобожден трижды. Первое обращение к `gather` завершается, по видимости, нормально:

```

Два захвата, три освобождения...
Ожидание возможности захвата
Захвачен
Ожидание возможности захвата
Захвачен
Одиночное освобождение!
Одиночное освобождение - готово!
Освобождается
Освобождается

```

Однако при втором обращении, когда мы захватываем семафор три раза, возникают проблемы – все три захвата происходят сразу! Мы непреднамеренно превысили предел семафора:

```

Три захвата...
Ожидание возможности захвата
Захвачен

```

Ожидание возможности захвата
Захвачен
Ожидание возможности захвата
Захвачен
Освобождается
Освобождается
Освобождается

Для таких ситуаций `asyncio` предлагает класс `BoundedSemaphore`. Ведет он себя так же, как обычный, с одним отличием: при попытке вызвать метод `release` таким образом, что это изменит допустимый предел захватов, возбуждается исключение `ValueError: BoundedSemaphore released too many times`. В листинге ниже приведен очень простой пример.

Листинг 11.9 Ограниченные семафоры

```
import asyncio
from asyncio import BoundedSemaphore

async def main():
    semaphore = BoundedSemaphore(1)

    await semaphore.acquire()
    semaphore.release()
    semaphore.release()

asyncio.run(main())
```

Здесь второй вызов `release` возбудит исключение `ValueError`, означающее, что мы освободили семафор слишком много раз. Аналогичный результат будет иметь место, если в листинге 11.8 использовать `BoundedSemaphore` вместо `Semaphore`. Если вы вызываете `acquire` и `release` вручную, так что возникает опасность динамически превысить предел семафора, то лучше работать с `BoundedSemaphore`, потому что возникшее исключение предупредит об ошибке.

Итак, мы показали, как использовать семафоры для ограничения уровня конкурентности. Но примитивы синхронизации `asyncio` позволяют не только ограничить конкурентность, но и уведомлять задачи, когда что-то происходит. Далее поговорим о том, как это делается с помощью примитива `Event`.

11.4 Уведомление задач с помощью событий

Иногда необходимо дождаться какого-то внешнего события, прежде чем продолжить работу. Например, дождаться, пока в буфере что-то появится, и только тогда приступить к его обработке. Или дождаться подключения устройства к приложению. Или подождать завершения инициализации чего-то. Также может быть несколько задач, ожи-

дающих обработки данных, которые еще не готовы. Объекты Event предоставляют механизм, позволяющий ждать, ничего не делая, пока что-то не произойдет.

Под капотом класс Event хранит флаг, показывающий, произошло событие или еще нет. Для управления этим флагом имеется два метода: `set` и `clear`. Метод `set` устанавливает флаг в `True` и уведомляет всех, кто ожидает события. Метод `clear` сбрасывает флаг в `False`, в результате чего любой объект, ожидающий события, будет заблокирован.

Имея эти два метода, мы можем управлять внутренним состоянием, но как перейти в состояние ожидания события? В классе Event имеется метод-сопрограмма `wait`. Будучи помещен в выражение `await`, он блокирует выполнение, пока кто-то не вызовет `set` для объекта события. А после этого все последующие обращения к `wait` не блокируются и возвращают управление мгновенно. Если вызвать `clear` после `set`, то обращения к `wait` снова начнут блокироваться до момента очередного вызова `set`.

Продemonстрируем события на искусственном примере. Допустим, что есть две задачи, зависящие от внешнего события. Заставим их ждать, ничего не делая, пока событие не произойдет.

Листинг 11.10 Операции с событиями

```
import asyncio
import functools
from asyncio import Event

def trigger_event(event: Event):
    print('Активируется событие!')
    event.set()

async def do_work_on_event(event: Event):
    print('Ожидаю события...')
    await event.wait()
    print('Работаю!')
    await asyncio.sleep(1)
    print('Работа закончена!')
    event.clear()

async def main():
    event = asyncio.Event()
    asyncio.get_running_loop().call_later(5.0,
        functools.partial(trigger_event, event))
    await asyncio.gather(do_work_on_event(event), do_work_on_event(event))

asyncio.run(main())
```

Ждать события

Когда событие произойдет, блокировка снимается, и мы можем начать работу

Сбросить событие, в результате чего последующие обращения к `wait` блокируются

Активировать событие через 5 с

Здесь мы написали сопрограмму `do_work_on_event`, которая принимает событие и первым делом вызывает его метод `wait`. Это действие блокирует выполнение, пока кто-то не вызовет метод события `set`, означающий, что событие произошло. Мы также написали простую функцию `trigger_event`, которая устанавливает данное событие. В со-

программе `main` мы создаем объект события и вызываем метод `call_later`, который активирует его через 5 с. Затем мы дважды вызываем `do_work_on_event` с помощью `gather`, что создает две конкурентных задачи. Запустив программу, мы увидим, что в течение 5 с задачи ничего не делают, ожидая события, а потом начинают работать:

```
Ожидая события...
Ожидая события...
Активируется событие!
Работаю!
Работаю!
Работа закончена!
Работа закончена!
```

Это основной способ применения: ожидание события блокирует одну или несколько сопрограмм, пока событие не произойдет, после чего они могут продолжить работу. Далее мы рассмотрим более реалистичный пример. Допустим, что требуется разработать API, позволяющий клиентам загружать файлы на сервер. Из-за сетевых задержек и буферизации загрузка файлов может занять некоторое время. И мы хотим, чтобы в составе нашего API была сопрограмма, которая блокируется до полного завершения загрузки. Вызывающая эту сопрограмму сторона должна будет дожидаться прихода данных, а потом сможет делать с ними, что захочет.

Для реализации этой задумки можно воспользоваться событием. У нас будет сопрограмма, которая читает загружаемые данные и сохраняет их во внутреннем буфере. По достижении конца файла мы активируем событие, показывающее, что загрузка завершена. Другой метод-сопрограмма будет ждать установки события, после чего вернет содержимое файла. Реализуем этот API в виде класса `FileUpload`.

Листинг 11.11 API загрузки файла на сервер

```
import asyncio
from asyncio import StreamReader, StreamWriter
```

```
class FileUpload:
    def __init__(self,
                  reader: StreamReader,
                  writer: StreamWriter):
        self._reader = reader
        self._writer = writer
        self._finished_event = asyncio.Event()
        self._buffer = b''
        self._upload_task = None
```

```
    def listen_for_uploads(self):
        self._upload_task = asyncio.create_task(self._accept_upload())
```

```
    async def _accept_upload(self):
        while data := await self._reader.read(1024):
```

Создать задачу, которая читает
загружаемые данные и добавляет
их в конец буфера

```

        self._buffer = self._buffer + data
    self._finished_event.set()
    self._writer.close()
    await self._writer.wait_closed()
    async def get_contents(self):
        await self._finished_event.wait()
        return self._buffer

```

Блокирует выполнение, пока событие `_finished_event` не будет установлено, а потом возвращает содержимое буфера

Теперь создадим сервер загрузки файлов для тестирования этого API. Предположим, что после каждой успешной загрузки мы хотим вывести содержимое на стандартный вывод. Когда клиент подключился, мы создаем объект класса `FileUpload` и вызываем метод `listen_for_uploads`. Затем создаем отдельную задачу, которая ждет результатов `get_contents`.

Листинг 11.12 Использование разработанного API в сервере загрузки файлов

```

import asyncio
from asyncio import StreamReader, StreamWriter
from chapter_11.listing_11_11 import FileUpload

class FileServer:

    def __init__(self, host: str, port: int):
        self.host = host
        self.port = port
        self.upload_event = asyncio.Event()

    async def start_server(self):
        server = await asyncio.start_server(self._client_connected,
                                           self.host,
                                           self.port)

        await server.serve_forever()

    async def dump_contents_on_complete(self, upload: FileUpload):
        file_contents = await upload.get_contents()
        print(file_contents)

    def _client_connected(self, reader: StreamReader, writer: StreamWriter):
        upload = FileUpload(reader, writer)
        upload.listen_for_uploads()
        asyncio.create_task(self.dump_contents_on_complete(upload))

    async def main():
        server = FileServer('127.0.0.1', 9000)
        await server.start_server()

asyncio.run(main())

```

Здесь мы создали класс `FileServer`. Всякий раз как к нашему серверу подключается клиент, мы создаем экземпляр написанного ранее

класса `FileUpload`, который начинает читать данные, загружаемые клиентом. Одновременно запускается задача, обернутая в `сопрограмму` `dump_contents_on_complete`. Она вызывает метод `get_contents` объекта `upload` (который вернет управление только после завершения загрузки) и печатает файл на стандартный вывод.

Для тестирования сервера можно воспользоваться программой `net-cat`. Выберите какой-нибудь файл в своей файловой системе и выполните следующую команду, заменив `file` именем выбранного файла:

```
cat file | nc localhost 9000
```

Вы увидите, что сразу после завершения загрузки содержимое загруженного файла печатается на стандартный вывод.

У событий есть один недостаток, о котором следует помнить: они могут возникать чаще, чем ваши сопрограммы в состоянии на них реагировать. Предположим, что мы используем одно событие для пробуждения нескольких задач в технологическом процессе типа производитель–потребитель. Если все задачи-исполнители заняты в течение длительного времени, то событие может возникнуть, когда мы работаем, и мы его никогда не увидим. Для демонстрации напомним простенький пример. Создадим две задачи-исполнителя, работающие по 5 с каждая. Также создадим задачу, которая генерирует событие каждую секунду, т. е. быстрее, чем потребители могут их обработать.

Листинг 11.13 Исполнитель не поспевает за событиями

```
import asyncio
from asyncio import Event
from contextlib import suppress

async def trigger_event_periodically(event: Event):
    while True:
        print('Активируется событие!')
        event.set()
        await asyncio.sleep(1)

async def do_work_on_event(event: Event):
    while True:
        print('Ожидаю события...')
        await event.wait()
        event.clear()
        print('Работаю!')
        await asyncio.sleep(5)
        print('Работа закончена!')

async def main():
    event = asyncio.Event()
    trigger = asyncio.wait_for(trigger_event_periodically(event), 5.0)

    with suppress(asyncio.TimeoutError):
        await asyncio.gather(do_work_on_event(event),
```

```
do_work_on_event(event), trigger)

asyncio.run(main())
```

При выполнении этой программы мы увидим, что событие активируется и оба исполнителя начинают работать конкурентно. А мы тем временем продолжаем генерировать события. Поскольку исполнители заняты, они не увидят второго возникновения события, пока не закончат работу и не вызовут `event.wait()` во второй раз. Если важно реагировать на каждое событие, то придется воспользоваться очередью, о чем мы будем говорить в следующей главе.

События полезны, когда мы хотим уведомить о том, что произошло нечто интересное, но что, если нам нужно сочетать ожидание события с монопольным доступом к разделяемому ресурсу, например подключением к базе данных? В таких случаях могут помочь условия.

11.5 Условия

События хороши, когда нужно просто уведомить о том, что произошло нечто, но ведь бывают ситуации посложнее. Допустим, что по событию требуется получить доступ к разделяемому ресурсу, т. е. захватить блокировку. Или что перед продолжением работы нужно дожидаться более сложного сочетания условий, чем простое событие. Или что нужно разбудить не все задачи, а только определенное число. Во всех этих случаях могут выручить условия. Это самый сложный из всех встречавшихся до сих пор примитивов синхронизации, а потому необходимость в них, скорее всего, будет возникать у вас нечасто.

Условие объединяет некоторые аспекты блокировки и события в один примитив синхронизации, по существу, обертывая поведение того и другого. Сначала мы захватываем блокировку условия, что дает сопрограмме монопольный доступ к разделяемому ресурсу, так что она может безопасно изменять его состояние. Затем мы ждем события с помощью сопрограммы `wait` или `wait_for`. Эти сопрограммы освобождают блокировку и блокируют выполнение до возникновения события, после чего заново захватывают блокировку, восстанавливая монопольный доступ.

Выглядит довольно запутанно, поэтому напомним простой пример, иллюстрирующий использование условий. Мы создадим две задачи-исполнителя, каждая из которых пытается захватить блокировку условия, а затем ждет уведомления о событии. Затем спустя несколько секунд мы активируем условие, которое пробуждает обе задачи и позволяет им продолжить работу.

Листинг 11.14 Иллюстрация условий

```
import asyncio
from asyncio import Condition
```

```

async def do_work(condition: Condition):
    while True:
        print('Ожидаю блокировки условия...')
        async with condition:
            print('Блокировка захвачена, освобождаю и жду выполнения условия...')
            await condition.wait()
            print('Условие выполнено, вновь захватываю блокировку и начинаю работать...')
            await asyncio.sleep(1)
        print('Работа закончена, блокировка освобождена.')

async def fire_event(condition: Condition):
    while True:
        await asyncio.sleep(5)
        print('Перед уведомлением, захватываю блокировку условия...')
        async with condition:
            print('Блокировка захвачена, уведомляю всех исполнителей.')
            condition.notify_all()
        print('Исполнители уведомлены, освобождаю блокировку.')

async def main():
    condition = Condition()

    asyncio.create_task(fire_event(condition))
    await asyncio.gather(do_work(condition), do_work(condition))

asyncio.run(main())

```

Ждать возможности захватить блокировку условия; после захвата освободить блокировку

После выхода из блока `async with` освободить блокировку условия

Ждать события; когда оно произойдет, заново захватить блокировку условия

Уведомить все задачи о событии

Здесь мы видим две сопрограммы: `do_work` и `fire_event`. Метод `do_work` захватывает условие, что аналогично захвату блокировки, а затем вызывает метод `wait` условия. Этот метод блокирует выполнение, пока кто-то не вызовет метод условия `notify_all`.

Сопрограмма `fire_event` некоторое время спит, а затем захватывает условие и вызывает метод `notify_all`, который пробуждает все задачи, в данный момент ожидающие условия. В сопрограмме `main` мы создаем и конкурентно запускаем одну задачу `fire_event` и две задачи `do_work`. При выполнении программы мы увидим следующую повторяющуюся последовательность:

```

Worker 1: Ожидаю блокировки условия...
Worker 1: Блокировка захвачена, освобождаю и жду выполнения условия...
Worker 2: Ожидаю блокировки условия...
Worker 2: Блокировка захвачена, освобождаю и жду выполнения условия...
fire_event: Перед уведомлением, захватываю блокировку условия...
fire_event: Блокировка захвачена, уведомляю всех исполнителей.
fire_event: Исполнители уведомлены, освобождаю блокировку.
Worker 1: Условие выполнено, вновь захватываю блокировку и начинаю работать...
Worker 1: Работа закончена, блокировка освобождена.
Worker 1: Ожидаю блокировки условия...
Worker 2: Условие выполнено, вновь захватываю блокировку и начинаю работать...
Worker 2: Работа закончена, блокировка освобождена.
Worker 2: Ожидаю блокировки условия...
Worker 1: Блокировка захвачена, освобождаю и жду выполнения условия...
Worker 2: Блокировка захвачена, освобождаю и жду выполнения условия...

```

Обратите внимание, что оба исполнителя начинают работать медленно и блокируются, ожидая, пока сопрограмма `fire_event` вызовет `notify_all`. После этого задачи-исполнители пробуждаются и продолжают делать свое дело.

У условий имеется дополнительный метод-сопрограмма `wait_for`. Он не блокирует выполнение, ожидая, что кто-то вызовет `notify_all`, а принимает предикат (функцию без аргументов, возвращающую булево значение) и блокирует выполнение, пока предикат не примет значение `True`. Это полезно, когда имеется разделяемый ресурс и сопрограммы, выполнение которых зависит от того, когда некоторое условие, определяемое состоянием ресурса, станет истинным.

В качестве примера создадим класс, который обертывает подключение к базе данных и выполняет запросы. Вначале у нас имеется подключение, которое не может выполнять несколько запросов одновременно и должно быть инициализировано перед выполнением первого запроса. Сочетание разделяемого ресурса и события, по которому выполнение блокируется, – как раз та комбинация предпосылок, которая оправдывает применение класса `Condition`. Смоделируем ситуацию, написав класс, имитирующий подключение к базе данных. Затем, воспользовавшись этим классом, попытаемся конкурентно выполнить два запроса, прежде чем инициализация подключения будет закончена.

Листинг 11.15 Применение условий для ожидания конкретного состояния

```
import asyncio
from enum import Enum

class ConnectionState(Enum):
    WAIT_INIT = 0
    INITIALIZING = 1
    INITIALIZED = 2

class Connection:

    def __init__(self):
        self._state = ConnectionState.WAIT_INIT
        self._condition = asyncio.Condition()

    async def initialize(self):
        await self._change_state(ConnectionState.INITIALIZING)
        print('initialize: Инициализация подключения...')
        await asyncio.sleep(3) # имитируется время инициализации подключения
        print('initialize: Подключение инициализировано')
        await self._change_state(ConnectionState.INITIALIZED)

    async def execute(self, query: str):
        async with self._condition:
            print('execute: Ожидание инициализации подключения')
            await self._condition.wait_for(self._is_initialized)
```

```

        print(f'execute: Выполняется {query}!!!')
        await asyncio.sleep(3) # имитация долгого запроса

    async def _change_state(self, state: ConnectionState):
        async with self._condition:
            print(f'change_state: Состояние изменяется с {self._state} на {state}')
            self._state = state
            self._condition.notify_all()

    def _is_initialized(self):
        if self._state is not ConnectionState.INITIALIZED:
            print(f'_is_initialized: Инициализация подключения не закончена,
состояние равно {self._state}')
            return False
        print(f'_is_initialized: Подключение инициализировано!')
        return True

    async def main():
        connection = Connection()
        query_one = asyncio.create_task(connection.execute('select * from table'))
        query_two = asyncio.create_task(connection.execute('select * from other_table'))
        asyncio.create_task(connection.initialize())
        await query_one
        await query_two

    asyncio.run(main())

```

Здесь мы написали класс подключения, содержащий объект условия и хранящий внутреннее состояние, которое мы инициализировали значением `WAIT_INIT`, показывающим, что мы ждем завершения инициализации. В классе `Connection` имеется также несколько методов. Метод `initialize` моделирует создание подключения к базе данных. Он вызывает метод `_change_state`, который записывает в состояние значение `INITIALIZING` при первом вызове, а затем, когда подключение инициализировано, изменяет его на `INITIALIZED`. В методе `_change_state` мы устанавливаем внутреннее состояние, а затем вызываем метод условия `notify_all`, который пробуждает все задачи, ожидающие условия.

В методе `execute` захватываем объект условия в блоке `async with`, после чего вызываем `wait_for` с предикатом, который проверяет, равно ли состояние `INITIALIZED`. Эта проверка блокирует выполнение, пока подключение к базе данных не будет полностью инициализировано, что предотвращает случайное выполнение запроса в момент, когда подключения еще не существует. Затем в сопрограмме `main` мы создаем экземпляр класса подключения, две задачи для выполнения запросов и еще одну для инициализации подключения. Запустив эту программу, мы увидим следующую картину, показывающую, что задачи выполнения запросов честно ждут завершения задачи инициализации:

```

execute: Ожидание инициализации подключения
_is_initialized: Инициализация подключения не закончена, состояние равно

```

```

ConnectionState.WAIT_INIT
execute: Ожидание инициализации подключения
_is_initialized: Инициализация подключения не закончена, состояние равно
ConnectionState.WAIT_INIT
change_state: Состояние изменяется с ConnectionState.WAIT_INIT to
ConnectionState.INITIALIZING
initialize: Инициализация подключения...
_is_initialized: Инициализация подключения не закончена, состояние равно
ConnectionState.INITIALIZING
_is_initialized: Инициализация подключения не закончена, состояние равно
ConnectionState.INITIALIZING
initialize: Подключение инициализировано
change_state: State changing from ConnectionState.INITIALIZING to
ConnectionState.INITIALIZED
_is_initialized: Подключение инициализировано!
execute: Running select * from table!!!
_is_initialized: Подключение инициализировано!
execute: Running select * from other_table!!!

```

Условия полезны в ситуациях, когда необходим доступ к разделяемому ресурсу, и перед началом работы требуется получить уведомление о некотором состоянии. Это довольно редкий случай, так что маловероятно, что в коде на основе `asyncio` вам понадобится использовать условия.

Резюме

- Мы узнали об ошибках однопоточной конкурентности и о том, чем они отличаются от ошибок в многопоточных и многопроцессных программах.
- Мы научились использовать блокировки `asyncio` для предотвращения ошибок конкурентности и синхронизации сопрограмм. Благодаря однопоточной природе `asyncio` такая необходимость возникает гораздо реже, но все же возникает, когда изменение разделяемого состояния возможно в момент, когда программа приостановлена в `await`.
- Мы узнали, как использовать семафоры для управления доступом к конечным ресурсам и ограничения уровня конкурентности, что бывает полезно для формирования трафика.
- Мы научились использовать события для инициирования действий, когда происходит что-интересное, например завершается инициализация.
- Мы научились использовать условия, чтобы дожидаться события и получить доступ к разделяемому ресурсу.

12

Асинхронные очереди

Краткое содержание главы

- Асинхронные очереди.
- Использование очередей в технологических процессах типа производитель–потребитель.
- Использование очередей в веб-приложениях.
- Использование очередей с приоритетами.
- Асинхронные LIFO-очереди.

При написании приложений для обработки событий или данных других типов нам часто бывает нужен механизм для хранения событий и распределения их между несколькими исполнителями. Затем исполнители могут конкурентно сделать с этими событиями все, что нужно; это позволяет сэкономить немного времени по сравнению с последовательной обработкой событий. Библиотека `asyncio` предоставляет для этой цели реализацию асинхронной очереди. Мы можем добавить в очередь данные и запустить несколько конкурентных исполнителей, которые будут извлекать данные из очереди и обрабатывать их по мере готовности.

Такую схему работы часто называют производитель–потребитель. Одна сторона порождает данные или события, которые надлежит обработать; их обработка может занять длительное время. Очередь позволяет делегировать длительные операции фоновым задачам,

сохранив отзывчивость пользовательского интерфейса. Мы помещаем элемент в очередь для последующей обработки и информируем пользователя о том, что работа началась в фоне. У асинхронных очередей есть и дополнительное преимущество – они дают механизм ограничения конкурентности, поскольку обычно с очередью работает конечное число задач-исполнителей. Другой способ ограничить уровень конкурентности – с помощью семафоров – мы рассматривали в главе 11.

В этой главе мы покажем, как использовать асинхронные очереди в технологическом процессе производитель–потребитель. Сначала освоим основы на примере бакалейной лавки, в котором потребителями будут кассиры. Затем применим полученные знания к веб-API управления заказами и продемонстрируем, как быстро отвечать пользователям, перенеся обработку очереди в фоновый режим. Мы также научимся обрабатывать задачи в порядке приоритетов – это полезно, когда более важные задачи желательно обработать раньше, пусть даже они были помещены в очередь позже. Наконец, мы рассмотрим очереди типа LIFO (последним пришел, первым ушел) и поговорим о недостатках асинхронных очередей.

12.1 Основы асинхронных очередей

Очередь – это структура данных с дисциплиной обслуживания FIFO (первым пришел, первым ушел). Иными словами, первым будет извлечен элемент, находящийся в начале очереди. По сути дела, это полный аналог очереди к кассе в бакалейной лавке. Вы становитесь в конец очереди и ждете, пока кассир обслужит всех стоящих перед вами. Когда очередной покупатель обслужен, вы продвигаетесь вперед, а тем временем новые покупатели становятся вслед за вами. Оказавшись в начале очереди, вы рассчитываетесь с кассиром и покидаете очередь.

Описанная очередь к кассе – синхронный процесс. Кассир обслуживает одного покупателя в каждый момент времени. А что, если пересмотреть подход к очереди, так чтобы полнее использовать конкурентность и организовать работу, как в супермаркете? Кассиров теперь будет несколько, а очередь по-прежнему одна. Как только какой-то кассир освобождается, он поднимает флажок, предлагая очередному покупателю подойти к кассе. То есть несколько кассиров не только конкурентно обслуживают покупателей, но и конкурентно выбирают покупателей из очереди.

Это и есть принципиальное описание асинхронной очереди. Мы добавляем в очередь данные, нуждающиеся в обработке. Затем несколько исполнителей выбирают данные из очереди по мере поступления.

Рассмотрим эту идею подробнее на примере очереди в супермаркете. Будем считать задачи-исполнители кассирами, а «элементы

данных» – покупателями. С каждым покупателем ассоциирован список товаров, которые кассир должен просканировать. Для некоторых товаров сканирование занимает больше времени; например, бананы нужно взвешивать и вводить их SKU-код. А при продаже алкоголя кассир может проверить паспорт покупателя.

Мы реализуем несколько классов данных, представляющих товар целым числом – временем (в секундах), необходимым кассиру для обработки. Мы также напишем класс покупателя со случайным набором купленных товаров. Затем поместим покупателей в очередь `asyncio`, моделирующую очередь к кассе. Также создадим несколько задач-исполнителей, представляющих кассиров. Эти задачи будут выбирать покупателей из очереди, в цикле перебирать товары каждого и засыпать на время, необходимое для моделирования процесса обработки товара.

Листинг 12.1 Очередь в супермаркете

```
import asyncio
from asyncio import Queue
from random import randrange
from typing import List

class Product:
    def __init__(self, name: str, checkout_time: float):
        self.name = name
        self.checkout_time = checkout_time

class Customer:
    def __init__(self, customer_id: int, products: List[Product]):
        self.customer_id = customer_id
        self.products = products

async def checkout_customer(queue: Queue, cashier_number: int):
    while not queue.empty():
        customer: Customer = queue.get_nowait()
        print(f'Кассир {cashier_number} '
              f'обслуживает покупателя '
              f'{customer.customer_id}')
        for product in customer.products:
            print(f"Кассир {cashier_number} "
                  f"обслуживает покупателя "
                  f"{customer.customer_id}: {product.name}")
            await asyncio.sleep(product.checkout_time)
        print(f'Кассир {cashier_number} '
              f'закончил обслуживать покупателя '
              f'{customer.customer_id}')
        queue.task_done()

    Выбираем покупателя,
    если в очереди кто-то есть

    Обработываем каждый товар,
    купленный покупателем

async def main():
    customer_queue = Queue()

    all_products = [Product('пиво', 2),
```

Создать 10 покупателей со случайным набором товаров	Product('бананы', .5), Product('колбаса', .2), Product('подгузники', .2)]	<pre> for i in range(10): products = [all_products[randrange(len(all_products))]] for _ in range(randrange(10))] customer_queue.put_nowait(Customer(i, products)) cashiers = [asyncio.create_task(checkout_customer(customer_queue, i)) for i in range(3)] await asyncio.gather(customer_queue.join(), *cashiers) asyncio.run(main()) </pre>
Создать трех «кассиров», т. е. задач- исполнителей, обслуживающих покупателей		

Здесь мы создаем два класса данных: для товара и для покупателя. Класс товара содержит наименование и время (в секундах), необходимое кассиру для обработки товара. В классе покупателя хранятся приобретенные им товары. Мы также написали сопрограмму `checkout_customer`, которая обслуживает одного покупателя. Пока очередь не пуста, эта задача выбирает покупателя из начала очереди, вызывая метод `queue.get_nowait()`, и имитирует время сканирования товара, вызывая `asyncio.sleep`. Обслужив покупателя, мы вызываем метод `queue.task_done`. Он сигнализирует очереди, что исполнитель завершил обработку текущего элемента данных. Под капотом класс `Queue` хранит счетчик, который увеличивается на единицу при выборке элемента из очереди; так очередь следит за количеством незавершенных задач. Вызов метода `task_done` говорит очереди, что задача завершилась, поэтому счетчик уменьшается на единицу (зачем это нужно, мы объясним чуть ниже при обсуждении метода `join`).

В сопрограмме `main` мы создаем список имеющихся товаров и генерируем 10 покупателей, каждый со случайным набором товаров. Также создаем три задачи-исполнителя, обертывающие сопрограмму `checkout_customer`, которые сохраняем в списке `cashiers` – аналоге трех кассиров, работающих в нашем воображаемом супермаркете. Наконец, мы ожидаем завершения задач `checkout_customer`, вызывая сопрограмму `customer_queue.join()` с помощью `gather`. Мы используем `gather`, чтобы все исключения, возникшие в задачах кассиров, возбуждались в сопрограмме `main`. Сопрограмма `join` блокируется до тех пор, пока очередь не опустеет, т. е. все покупатели не будут обслужены. Очередь считается пустой, когда внутренний счетчик незавершенных работ обратился в нуль. Поэтому так важно вызывать `task_done` в задачах-исполнителях. Если этого не сделать, сопрограмма `join` будет иметь неправильное представление об очереди и может никогда не завершиться.

Покупатели генерируются случайно, но общая картина будет такой, как показано ниже: мы видим, что все задачи-исполнители (кассиры) конкурентно выбирают покупателей из очереди:

Кассир 0 обслуживает покупателя 0
Кассир 0 обслуживает покупателя 0: колбаса
Кассир 1 обслуживает покупателя 1
Кассир 1 обслуживает покупателя 1: пиво
Кассир 2 обслуживает покупателя 2
Кассир 2 обслуживает покупателя 2: бананы
Кассир 0 обслуживает покупателя 0: бананы
Кассир 2 обслуживает покупателя 2: колбаса
Кассир 0 обслуживает покупателя 0: колбаса
Кассир 2 обслуживает покупателя 2: бананы
Кассир 0 закончил обслуживать покупателя 0
Кассир 0 обслуживает покупателя 3

Наши кассиры начинают обслуживать покупателей из очереди конкурентно. Закончив обслуживание очередного покупателя, кассир выбирает из очереди следующего, пока очередь не опустеет.

Вы, вероятно, обратили внимание на странные имена методов для помещения в очередь и извлечения из нее: `get_nowait` и `put_nowait`. Зачем этот суффикс `nowait`? Есть два способа поместить элемент в очередь и выбрать его оттуда: неблокирующий и регулярный. Варианты `get_nowait` и `put_nowait` не блокируют выполнения и возвращают управление немедленно. А зачем нужны блокирующие варианты?

Все дело в том, как мы хотим обрабатывать верхнюю и нижнюю границу очереди, т. е. что должно происходить, когда в очереди слишком много элементов или вообще нет элементов.

Вернемся к примеру очереди в супермаркете и рассмотрим два случая, которые вряд ли могут возникнуть на практике, но иллюстрируют использование сопрограммных вариантов `get` и `put`.

- Маловероятно, что одновременно возникнет очередь из 10 покупателей, а после того как она рассосется, все кассиры вообще перестанут работать.
- Наша очередь покупателей вряд ли должна быть неограниченной. Допустим, что только что вышла возжеленная последняя модель игровой консоли, и во всем городе только ваш магазин ей торгует. Естественно, возник ажиотаж, и ваш магазин осадили покупатели. Надо думать, больше 5000 человек в магазин не поместится, так что нужно как-то отводить их или пусть ждут на улице.

Для решения первой проблемы мы можем переделать приложение, так что оно будет каждые несколько секунд случайно генерировать несколько покупателей, моделируя реальную очередь в супермаркете. В текущей реализации `checkout_customer` мы продолжаем итерации цикла, пока очередь не пуста, и выбираем покупателя методом `get_nowait`. Поскольку очередь может оказаться пустой, мы не можем циклиться по условию `not queue.empty`, так как кассиры продолжают оставаться на рабочих местах, даже если в очереди никого нет, поэтому в сопрограмме исполнителя нужно условие `while True`. А что было бы, если бы мы вызвали `get_nowait` в момент, когда очередь пуста?

Это легко проверить, написав всего несколько строк кода; создадим пустую очередь и вызовем этот метод:

```
import asyncio
from asyncio import Queue

async def main():
    customer_queue = Queue()
    customer_queue.get_nowait()

asyncio.run(main())
```

Наш метод возбуждает исключение `asyncio.queue.QueueEmpty`. Конечно, можно было бы перехватить его в блоке `try/catch` и проигнорировать, но такое решение не годится, потому что стоит очереди оказаться пустой, как наша задача-исполнитель начинает «жрать» процессорное время, поскольку только и делает, что возбуждает и перехватывает исключения. В таком случае можно использовать метод-сопрограмму `get`. Он блокирует выполнение (не прибегая к активному ожиданию), пока в очереди не появится элемент, и не возбуждает исключения. Это эквивалентно простаиванию задач-исполнителей, которые ждут, когда в очереди появится покупатель, нуждающийся в обслуживании.

Чтобы решить вторую проблему, – тысячи покупателей, стремящихся одновременно занять очередь, – нужно подумать об ограничении очереди сверху. По умолчанию очереди неограниченны и могут расти до бесконечности. Теоретически это допустимо, но на практике у любой системы есть ограничения по памяти, так что имеет смысл ограничить размер очереди, чтобы не переполнить память. В данном случае нужно подумать, какого поведения мы хотим добиться, когда очередь заполнена. Посмотрим, что будет, если создать очередь всего на один элемент и попытаться добавить второй с помощью `put_nowait`:

```
import asyncio
from asyncio import Queue

async def main():
    queue = Queue(maxsize=1)

    queue.put_nowait(1)
    queue.put_nowait(2)

asyncio.run(main())
```

В этом случае, как и в случае `get_nowait`, `put_nowait` возбуждает исключение типа `asyncio.queue.QueueFull`. Но, как и раньше, существует метод-сопрограмма `put`. Этот метод блокирует выполнение, пока в очереди не освободится место. Учитывая это, переработаем наш пример с использованием методов `get` и `put`.

Листинг 12.2 Использование методов-сопрограмм очереди

```

import asyncio
from asyncio import Queue
from random import randrange

class Product:
    def __init__(self, name: str, checkout_time: float):
        self.name = name
        self.checkout_time = checkout_time

class Customer:
    def __init__(self, customer_id, products):
        self.customer_id = customer_id
        self.products = products

async def checkout_customer(queue: Queue, cashier_number: int):
    while True:
        customer: Customer = await queue.get()
        print(f'Кассир {cashier_number} '
              f'обслуживает покупателя '
              f'{customer.customer_id}')
        for product in customer.products:
            print(f"Кассир {cashier_number} "
                  f"обслуживает покупателя "
                  f"{customer.customer_id}: {product.name}")
            await asyncio.sleep(product.checkout_time)
        print(f'Кассир {cashier_number} '
              f'закончил обслуживать покупателя '
              f'{customer.customer_id}')
        queue.task_done()

def generate_customer(customer_id: int) -> Customer:
    all_products = [Product('пиво', 2),
                    Product('бананы', .5),
                    Product('колбаса', .2),
                    Product('подгузники', .2)]
    products = [all_products[randrange(len(all_products))]]
    for _ in range(randrange(10)):
        products.append(all_products[randrange(len(all_products))])
    return Customer(customer_id, products)

async def customer_generator(queue: Queue):
    customer_count = 0

    while True:
        customers = [generate_customer(i)
                     for i in range(customer_count,
                                     customer_count + randrange(5))]
        for customer in customers:
            print('Ожидаю возможности поставить покупателя в очередь...')
            await queue.put(customer)
            print('Покупатель поставлен в очередь!')
        customer_count = customer_count + len(customers)
        await asyncio.sleep(1)

```

← Сгенерировать случайного покупателя

← Генерировать несколько случайных покупателей в секунду

```

async def main():
    customer_queue = Queue(5)

    customer_producer = asyncio.create_task(customer_generator(customer_queue))

    cashiers = [asyncio.create_task(checkout_customer(customer_queue, i))
                for i in range(3)]

    await asyncio.gather(customer_producer, *cashiers)

    asyncio.run(main())

```

Здесь сопрограмма `generate_customer` создает покупателя со случайным списком товаров. А сопрограмма `customer_generator` каждую секунду генерирует от одного до пяти случайных покупателей и добавляет их в очередь методом `put`. Поскольку `put` – сопрограмма, при заполнении очереди `customer_generator` заблокирует выполнение до появления свободного места. Конкретно если в очереди пять покупателей и *производитель* пытается добавить шестого, то очередь заблокируется, пока кассир не обслужит какого-то покупателя. Можно считать сопрограмму `customer_generator` *производителем*, поскольку она порождает покупателей, обслуживаемых кассирами.

Мы также хотим, чтобы сопрограмма `checkout_customer` работала бесконечно, поскольку кассиры остаются на своих местах, даже когда очередь пуста. Поэтому `checkout_customer` вызывает метод-сопрограмму очереди `get`, который блокирует выполнение, если в очереди нет покупателей. Затем в сопрограмме `main` мы создаем очередь на пять покупателей и три конкурентные задачи `checkout_customer`. Кассиров можно рассматривать как *потребителей*: они потребляют покупателей из очереди и обслуживают их.

Этот код случайным образом генерирует покупателей, но в какой-то момент очередь должна заполниться, потому что кассиры обслуживают покупателей не так быстро, как производитель их создает. Поэтому мы увидим показанную ниже картину, когда производитель приостанавливает добавление покупателей в очередь, пока какой-то покупатель не будет обслужен.

Ожидая возможности поставить покупателя в очередь...

```

Кассир 1 обслуживает покупателя 7: колбаса
Кассир 1 обслуживает покупателя 7: подгузники
Кассир 1 обслуживает покупателя 7: подгузники
Кассир 2 закончил обслуживать покупателя 5
Кассир 2 обслуживает покупателя 9
Кассир 2 обслуживает покупателя 7: бананы

```

Покупатель поставлен в очередь!

Мы разобрались с основами работы асинхронных очередей, но поскольку на практике мы обычно не занимаемся моделированием супермаркетов, то обратимся к нескольким более реалистичным сценариям, показывающим, как полученные знания можно применить в приложениях.

12.1.1 Очереди в веб-приложениях

Очереди могут оказаться полезны в веб-приложениях, если имеется занимающая много времени операция, которую можно запустить в фоновом режиме. Если бы мы выполняли ее в главной сопрограмме веб-запроса, то блокировали бы ее до завершения операции, поэтому пользователь увидел бы «тормозящую» страницу.

Представьте, что вы работаете в компании, эксплуатирующей интернет-магазин, а подсистема управления заказами работает медленно. Обработка заказа может занять несколько секунд, но мы не хотим, чтобы пользователь ждал сообщения о том, что его заказ принят. Кроме того, подсистема управления заказами плохо справляется с высокой нагрузкой, поэтому хотелось бы ограничить количество одновременно обрабатываемых запросов. Очередь может решить обе проблемы. Как мы видели, можно указать максимальное число элементов в очереди, так что попытка добавить еще один приведет к блокированию выполнения или к исключению. Таким образом, мы сможем ограничить число конкурентно работающих задач покупателей и положить естественный предел уровню конкурентности.

Очередь также решает проблему долгого ожидания ответа пользователем. Элемент добавляется в очередь практически мгновенно, а значит, мы можем сразу уведомить пользователя о том, что заказ принят, и порадовать его отзывчивостью страницы. Конечно, в реальном приложении может случиться, что фоновая задача завершается неудачно, а пользователь об этом ничего не знает, поэтому нужно сохранять какие-то данные и добавить логику для борьбы с этой неприятностью.

Для проверки описанной идеи напишем простое веб-приложение на основе `aiohttp`, в котором для выполнения фоновых задач используется очередь. Взаимодействие с медленной подсистемой управления заказами будем моделировать с помощью сопрограммы `asyncio.sleep`. На практике вы, скорее всего, будете взаимодействовать с микросервисной архитектурой по технологии REST, применяя `aiohttp` или аналогичную библиотеку, но мы для простоты ограничимся `sleep`.

В начале работы создадим очередь и набор задач-исполнителей, которые будут обращаться к медленному сервису. Также создадим окончательную точку `/order` типа POST, в которой будем помещать заказ в очередь (в данном случае просто генерировать случайное число, определяющее, сколько времени `sleep` будет спать). Поместив заказ в очередь, мы возвращаем код состояния HTTP 200 и сообщение о том, что заказ принят.

В точку завершения добавим логику корректной остановки, поскольку в момент, когда запрашивается остановка, могут оставаться необработанные заказы. В таком случае мы подождем, пока все работающие исполнители завершатся.

Листинг 12.3 Использование очередей в веб-приложении

```

import asyncio
from asyncio import Queue, Task
from typing import List
from random import randrange
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response

routes = web.RouteTableDef()

QUEUE_KEY = 'order_queue'
TASKS_KEY = 'order_tasks'

async def process_order_worker(worker_id: int, queue: Queue):
    while True:
        print(f'Исполнитель {worker_id}: ожидание заказа...')
        order = await queue.get()
        print(f'Исполнитель {worker_id}: обрабатывается заказ {order}')
        await asyncio.sleep(order)
        print(f'Исполнитель {worker_id}: заказ {order} обработан')
        queue.task_done()

@routes.post('/order')
async def place_order(request: Request) -> Response:
    order_queue = app[QUEUE_KEY]
    await order_queue.put(randrange(5))
    return Response(body='Order placed!')

async def create_order_queue(app: Application):
    print('Создание очереди заказов и задач.')
    queue: Queue = asyncio.Queue(10)
    app[QUEUE_KEY] = queue
    app[TASKS_KEY] = [asyncio.create_task(process_order_worker(i, queue))
                      for i in range(5)]

async def destroy_queue(app: Application):
    order_tasks: List[Task] = app[TASKS_KEY]
    queue: Queue = app[QUEUE_KEY]
    print('Ожидание завершения исполнителей в очереди...')
    try:
        await asyncio.wait_for(queue.join(), timeout=10)
    finally:
        print('Обработка всех заказов завершена, отменяются задачи-исполнители...')
        [task.cancel() for task in order_tasks]

app = web.Application()
app.on_startup.append(create_order_queue)
app.on_shutdown.append(destroy_queue)

app.add_routes(routes)
web.run_app(app)

```

Выбрать заказ из очереди и обработать его

Поместить заказ в очередь и ответить пользователю немедленно

Создать очередь на 10 элементов и 5 задач-исполнителей

Ждать завершения работающих задач

Здесь первой идет сопрограма `process_order_worker`. Она выбирает из очереди элемент – в данном случае целое число – и засыпает на это время, моделируя работу с медленной системой управления заказами. Эта сопрограма крутится в бесконечном цикле – выбирает из очереди и обрабатывает элементы.

Следующие сопрограмы, `create_order_queue` и `destroy_order_queue`, создают и уничтожают очередь. Создание не вызывает затруднений – мы просто инициализируем очередь `asuncio` на 10 элементов, после чего создаем пять задач-исполнителей и сохраняем их в экземпляре `Application`.

Уничтожение очереди чуть сложнее. Сначала с помощью `Queue.join` нужно дождаться, когда все находящиеся в очереди элементы будут обработаны. Поскольку в этот момент производится остановка приложения, новых HTTP-запросов не предвидится, т. е. в очередь больше не будут помещаться заказы. Это означает, что все находящиеся в очереди заказы будут обработаны исполнителями. На всякий случай мы обернули вызов `join` методом `wait_for`, ограничив время ожидания 10 с. Это разумно, так как мы не хотим, чтобы какая-нибудь сбойная задача помешала приложению остановиться.

Наконец, определяем маршрут для нашего приложения. Оконечная точка `/order` типа `POST` создает случайную задержку и помещает ее в очередь. Поместив такой «заказ» в очередь, мы возвращаем пользователю код состояния HTTP 200, сопровождаемый коротким сообщением. Отметим, что мы воспользовались сопрограммным вариантом `put`, т. е. если очередь уже заполнена, то обработка запроса будет приостановлена, пока сообщение не попадет в очередь, что может занять некоторое время. Можно было бы вместо этого вызывать функцию `put_nowait` и возвращать код состояния HTTP 500 или какой-то другой код ошибки, предлагая пользователю повторить запрос позже. Мы решили, что можно немного подождать, пока запрос добавляется в очередь. Но если ваше приложение должно «отказывать быстро», то возвращать код ошибки при заполненной очереди, наверное, будет правильнее.

Благодаря очереди наша конечная точка `order` будет отвечать практически мгновенно при условии, что очередь не заполнена под завязку. В результате пользователь видит быстрый, нигде не тормозящий процесс заказа, и хочется надеяться, что это побудит его купить что-нибудь еще.

Но при использовании очередей `asuncio` в веб-приложении следует знать, какие могут возникать ошибки. Что, если один из экземпляров нашего API по какой-то причине (например, из-за исчерпания памяти) упадет или нужно будет перезапустить сервер, чтобы заново развернуть приложение? В таком случае необработанные заказы, находившиеся в очереди, будут потеряны, потому что они хранятся только в памяти. Иногда с потерей элемента в очереди можно смириться, но в случае заказа, сделанного покупателем, это, скорее всего, не так.

Очереди `asuncio` не предлагают готового решения для сохранения задач или обеспечения долговечности очереди. Если мы хотим защи-

тить находящиеся в очереди задачи от подобных проблем, то должны добавить метод, сохраняющий их где-то, например в базе данных. Но правильнее было бы использовать отдельную очередь вне `asynсio`, которая поддерживает долговременное хранение. Примерами сохраняемых на диске очередей задач являются Celery и RabbitMQ.

Конечно, включение в архитектуру отдельных очередей увеличивает сложность приложения. В случае сохраняемых очередей задач есть еще и потеря производительности, связанная с записью задачи на диск. При выборе оптимальной архитектуры приложения нужно тщательно взвешивать достоинства и недостатки хранения очередей только в памяти, как в `asynсio`, и поддержки сохраняемых очередей с помощью отдельного компонента.

12.1.2 Очередь в веб-роботе

Задачи-потребители могут одновременно выступать в роли производителей, если генерируют дополнительные элементы, помещаемые в очередь. Возьмем, к примеру, веб-робот, который посещает все ссылки на странице. Легко представить, что один исполнитель скачивает страницу и ищет в ней ссылки. Обнаружив ссылку, исполнитель добавляет ее в очередь. Затем другие исполнители могут выбирать ссылки из очереди и конкурентно посещать их, добавляя в очередь новые ссылки.

Давайте напишем такой робот. Создадим неограниченную очередь (можете ограничить ее, если опасаетесь переполнения памяти), в которой будут храниться URL-адреса, подлежащие скачиванию. Исполнители будут выбирать URL-адреса из очереди и скачивать их с помощью `aiоhttp`. Скачав адрес, мы разберем страницу с помощью популярного анализатора HTML, Beautiful Soup, и поместим найденные ссылки обратно в очередь.

В этом приложении мы не хотим просматривать весь интернет, поэтому ограничимся страницами, отстоящими от корневой страницы не более чем на заданное число переходов – максимальную глубину. Если максимальная глубина равна 3, то мы проследуем по ссылке лишь при условии, что она отстоит от корня не далее чем на три страницы.

Для начала установите Beautiful Soup версии 4.9.3:

```
pip install -Iv beautifulsoup4==4.9.3
```

Мы предполагаем, что вы знакомы с библиотекой Beautiful Soup. А если нет, то можете прочитать о ней в документации по адресу <https://www.crummy.com/software/BeautifulSoup/bs4/doc>.

Мы собираемся создать сопрограмму-исполнитель, которая будет извлекать страницу из очереди и скачивать ее с помощью `aiоhttp`. После этого воспользуемся Beautiful Soup, чтобы найти на странице все ссылки вида ``, и добавим их в очередь.

Листинг 12.4 Робот с очередью

```

import asyncio
import aiohttp
import logging
from asyncio import Queue
from aiohttp import ClientSession
from bs4 import BeautifulSoup

class WorkItem:
    def __init__(self, item_depth: int, url: str):
        self.item_depth = item_depth
        self.url = url

    async def worker(worker_id: int, queue: Queue, session: ClientSession,
                    max_depth: int):
        print(f'Исполнитель {worker_id}')
        while True:
            work_item: WorkItem = await queue.get()
            print(f'Исполнитель {worker_id}: обрабатывается {work_item.url}')
            await process_page(work_item, queue, session, max_depth)
            print(f'Исполнитель {worker_id}: закончена обработка {work_item.url}')
            queue.task_done()

    async def process_page(work_item: WorkItem, queue: Queue, session:
                          ClientSession, max_depth: int):
        try:
            response = await asyncio.wait_for(session.get(work_item.url), timeout=3)
            if work_item.item_depth == max_depth:
                print(f'Макс глубина достигнута '
                      f'для {work_item.url}')
            else:
                body = await response.text()
                soup = BeautifulSoup(body, 'html.parser')
                links = soup.find_all('a', href=True)
                for link in links:
                    queue.put_nowait(WorkItem(work_item.item_depth + 1,
                                                link['href']))
        except Exception as e:
            logging.exception(f'Ошибка при обработке url {work_item.url}')

    async def main():
        start_url = 'http://example.com'
        url_queue = Queue()
        url_queue.put_nowait(WorkItem(0, start_url))
        async with aiohttp.ClientSession() as session:
            workers = [asyncio.create_task(worker(i, url_queue, session, 3))
                       for i in range(100)]
            await url_queue.join()
            [w.cancel() for w in workers]

asyncio.run(main())

```

Выбрать из очереди URL-адрес и начать его скачивание

Скачать страницу по этому адресу, найти на ней все ссылки и поместить их в очередь

Создать очередь и 100 задач-исполнителей для обработки URL-адресов

Здесь мы сначала определяем класс `WorkItem`. Это простой класс для хранения URL-адреса и его глубины. Затем определяем сопрограмму-исполнитель, которая выбирает `WorkItem` из очереди и вызывает `process_page`. Сопрограмма `process_page` скачивает содержимое страницы с данным URL-адресом, если это возможно (в случае тайм-аута или исключения мы протоколируем ошибку и игнорируем URL). Затем она использует `Beautiful Soup`, чтобы найти все ссылки и добавить их в очередь для обработки другими исполнителями.

В сопрограмме `main` мы создаем очередь и помещаем в нее первый элемент `WorkItem`. В этом примере мы зашили в код адрес корневой страницы `example.com`, ее глубина равна 0. Затем создаем сеанс `aioshttp` и 100 исполнителей, что позволит конкурентно скачивать 100 страниц, а максимальную глубину задаем равной 3. После этого с помощью метода `join` ждем, когда очередь опустеет и все исполнители завершат работу. Когда обработка очереди закончится, снимаем все задачи-исполнители. При выполнении этой программы мы увидим, что запускается 100 задач-исполнителей, которые начинают искать ссылки в скачанных страницах. Выглядит это так:

```
Найдено 1 ссылок на странице http://example.com
Исполнитель 0: закончена обработка http://example.com
Исполнитель 0: обрабатывается https://www.iana.org/domains/example
Найдено 68 ссылок на странице https://www.iana.org/domains/example
Исполнитель 0: закончена обработка https://www.iana.org/domains/example
Исполнитель 0: обрабатывается /
Исполнитель 2: обрабатывается /domains
Исполнитель 3: обрабатывается /numbers
Исполнитель 4: обрабатывается /protocols
Исполнитель 5: обрабатывается /about
Исполнитель 6: обрабатывается /go/rfc2606
Исполнитель 7: обрабатывается /go/rfc6761
Исполнитель 8: обрабатывается http://www.icann.org/topics/idn/
Исполнитель 9: обрабатывается http://www.icann.org/
```

Исполнители продолжают скачивать страницы и обрабатывать ссылки, добавляя их в очередь, пока не будет достигнута заданная максимальная глубина.

Мы познакомились с основами асинхронных очередей на нескольких примерах: модели очереди в супермаркете, API управления заказами и веб-роботе. До сих пор исполнители назначали одинаковые веса всем элементам очереди и выбирали для обработки тот элемент, который оказался в начале очереди. Но что, если требуется, чтобы некоторые задачи выполнялись раньше, пусть даже они оказались ближе к концу очереди? Эту проблему решают очереди с приоритетами.

12.2 Очереди с приоритетами

В рассмотренных выше примерах элемента очереди обрабатывались в порядке FIFO – первым пришел, первым обслужен. Элемент, находящийся в начале очереди, обслуживался первым. Эта дисциплина применяется во многих случаях – как в программах, так и в жизни.

Но бывают приложения, в которых рассматривать все задачи на равных основаниях нежелательно. Допустим, мы разрабатываем конвейер обработки данных, в котором каждая задача – длительный запрос, занимающий несколько минут. Предположим, что две задачи поступают почти в одно и то же время. Первая задача – низкоприоритетный запрос к данным, а вторая – важное обновление, которое должно быть произведено срочно. Если очередь простая, то сначала будет выполнена первая задача, а второй, более важной, придется ждать, пока она завершится. Но представьте, что первая задача работает несколько часов или что все исполнители заняты, – тогда вторая задача будет томиться долго.

Для решения этой проблемы можно воспользоваться очередью с приоритетами и заставить исполнителей сначала брать наиболее важные задачи. Очереди с приоритетами реализованы с помощью *пирамид* (из модуля `heapq`), а не списков Python как простые очереди. Для создания очереди `asyncio` с приоритетами мы создаем экземпляр класса `asyncio.PriorityQueue`.

Мы не будем углубляться в особенности структуры данных, скажем лишь, что пирамида – это двоичное дерево, обладающее тем свойством, что значение родительского узла меньше значения любого его потомка (см. рис. 12.1). Этим оно отличается от двоичных деревьев поиска, которые часто используются в задачах сортировки и поиска и характеризуются тем, что значение левого потомка меньше значения родительского узла, а значение правого потомка – больше. Мы воспользуемся тем фактом, что узел на вершине пирамиды имеет наименьшее значение. Если самым приоритетным является узел с наименьшим значением, то он всегда будет находиться в начале очереди.

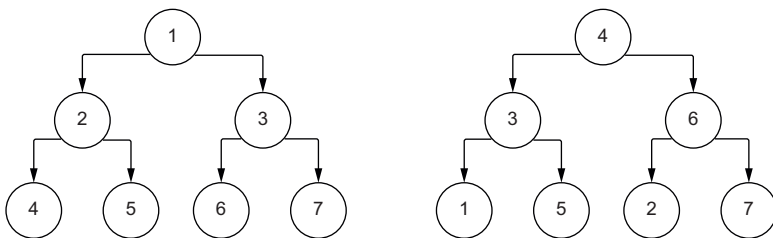


Рис. 12.1 Дерево слева обладает свойством пирамиды. Справа изображено двоичное дерево поиска, не обладающее свойством пирамиды

Маловероятно, что элементы, помещаемые в очередь, будут целыми числами, поэтому нам необходим какой-то способ построить элемент очереди с разумным правилом определения приоритета. Например, это можно сделать с помощью кортежа, в котором первый элемент – целое число, равное приоритету, а второй – произвольные данные задачи. По умолчанию реализация очереди смотрит на первый элемент кортежа и считает, что чем меньше это число, тем выше приоритет. Для демонстрации основ работы очередей с приоритетами рассмотрим пример, в котором элементы являются кортежами.

Листинг 12.5 Очередь с приоритетами, содержащая кортежи

```
import asyncio
from asyncio import Queue, PriorityQueue
from typing import Tuple

async def worker(queue: Queue):
    while not queue.empty():
        work_item: Tuple[int, str] = await queue.get()
        print(f'Обрабатывается элемент {work_item}')
        queue.task_done()

async def main():
    priority_queue = PriorityQueue()

    work_items = [(3, 'Lowest priority'),
                  (2, 'Medium priority'),
                  (1, 'High priority')]

    worker_task = asyncio.create_task(worker(priority_queue))

    for work in work_items:
        priority_queue.put_nowait(work)

    await asyncio.gather(priority_queue.join(), worker_task)

asyncio.run(main())
```

Здесь создается три элемента: с высоким, средним и низким приоритетом. Затем мы добавляем их в очередь в порядке, обратном порядку приоритетов, т. е. первым добавляем элемент с наименьшим приоритетом, а последним – элемент с наибольшим приоритетом. В обычной очереди это означало бы, что элемент с наименьшим приоритетом будет обрабатываться первым, но при выполнении программы мы увидим такую картину:

```
Обрабатывается элемент (1, 'High priority')
Обрабатывается элемент (2, 'Medium priority')
Обрабатывается элемент (3, 'Lowest priority')
```

Как видим, элементы обрабатываются в порядке приоритетов, а не в том порядке, в котором вставлялись в очередь. Кортежи работают

в простых случаях, но если элемент содержит много данных, то это решение может стать запутанным. А нельзя ли написать класс, который будет работать так, как нам нужно при использовании пирамиды? Можно, и проще всего для этой цели воспользоваться классом данных (если же класс данных по какой-то причине не годится, то можно было бы надлежащим образом реализовать *dunder*-методы `__lt__`, `__le__`, `__gt__` и `__ge__`).

Листинг 12.6 Очередь с приоритетами, содержащая экземпляры класса данных

```
import asyncio
from asyncio import Queue, PriorityQueue
from dataclasses import dataclass, field

@dataclass(order=True)
class WorkItem:
    priority: int
    data: str = field(compare=False)

async def worker(queue: Queue):
    while not queue.empty():
        work_item: WorkItem = await queue.get()
        print(f'Обрабатывается элемент {work_item}')
        queue.task_done()

async def main():
    priority_queue = PriorityQueue()

    work_items = [WorkItem(3, 'Lowest priority'),
                  WorkItem(2, 'Medium priority'),
                  WorkItem(1, 'High priority')]

    worker_task = asyncio.create_task(worker(priority_queue))

    for work in work_items:
        priority_queue.put_nowait(work)

    await asyncio.gather(priority_queue.join(), worker_task)

asyncio.run(main())
```

Здесь мы создаем класс с аннотацией `dataclass`, в которой атрибуту `order` присвоено значение `True`. В классе имеется целочисленное поле `priority` и строковое поле `data`, которое не участвует в сравнении. Это означает, что при добавлении экземпляров этого класса в очередь они будут сортироваться только по полю `priority`. Выполнив эту программу, мы увидим, что элементы обрабатываются в правильном порядке:

```
Обрабатывается элемент WorkItem(priority=1, data='High priority')
Обрабатывается элемент WorkItem(priority=2, data='Medium priority')
Обрабатывается элемент WorkItem(priority=3, data='Lowest priority')
```

Познакомившись с основами очередей с приоритетами, применим полученные знания к рассмотренному выше примеру API управления заказами. Допустим, что существуют «приоритетные» покупатели, которые тратят на нашем сайте особенно много денег. Мы хотим обрабатывать их заказы в первую очередь, чтобы не создавать им никаких неудобств. Для этого перепишем код, воспользовавшись очередью с приоритетами.

Листинг 12.7 Очередь с приоритетами в веб-приложении

```
import asyncio
from asyncio import Queue, Task
from dataclasses import field, dataclass
from enum import IntEnum
from typing import List
from random import randrange
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response

routes = web.RouteTableDef()

QUEUE_KEY = 'order_queue'
TASKS_KEY = 'order_tasks'

class UserType(IntEnum):
    POWER_USER = 1
    NORMAL_USER = 2

@dataclass(order=True)
class Order:
    user_type: UserType
    order_delay: int = field(compare=False)
```

Класс заказа — элемент очереди с приоритетом, основанным на типе пользователя

```
async def process_order_worker(worker_id: int, queue: Queue):
    while True:
        print(f'Исполнитель {worker_id}: ожидание заказа...')
        order = await queue.get()
        print(f'Исполнитель {worker_id}: обрабатывается заказ {order}')
        await asyncio.sleep(order.order_delay)
        print(f'Исполнитель {worker_id}: заказ {order} обработан')
        queue.task_done()

@routes.post('/order')
async def place_order(request: Request) -> Response:
    body = await request.json()
    user_type = UserType.POWER_USER if body['power_user'] == 'True' else
        UserType.NORMAL_USER
    order_queue = app[QUEUE_KEY]
    await order_queue.put(Order(user_type, randrange(5)))
    return Response(body='Order placed!')
```

Преобразовать запрос в заказ

```

async def create_order_queue(app: Application):
    print('Creating order queue and tasks.')
    queue: Queue = asyncio.PriorityQueue(10)
    app[QUEUE_KEY] = queue
    app[TASKS_KEY] = [asyncio.create_task(process_order_worker(i, queue))
                      for i in range(5)]

async def destroy_queue(app: Application):
    order_tasks: List[Task] = app[TASKS_KEY]
    queue: Queue = app[QUEUE_KEY]
    print('Ожидание завершения исполнителей в очереди...')
    try:
        await asyncio.wait_for(queue.join(), timeout=10)
    finally:
        print('Обработка всех заказов завершена, отменяются задачи-исполнители...')
        [task.cancel() for task in order_tasks]

app = web.Application()
app.on_startup.append(create_order_queue)
app.on_shutdown.append(destroy_queue)
app.add_routes(routes)

web.run_app(app)

```

Этот код очень похож на первоначальный API для взаимодействия с медленной подсистемой управления заказами, но теперь в нем используется очередь с приоритетами и создается класс `Order`, представляющий заказ. Мы ожидаем, что во входящем запросе будет флаг `"power_user"`, равный `True` для приоритетных пользователей и `False` для всех остальных. Обратиться к окончательной точке можно с помощью cURL:

```
curl -X POST -d '{"power_user": "False"}' localhost:8080/order
```

и передать нужное значение «power user». Если пользователь приоритетный, то любой освободившийся исполнитель обработает его заказы раньше заказов обычных пользователей.

Любопытный случай возникает, когда два элемента с одинаковыми приоритетами добавляются один за другим. Будут ли исполнители обрабатывать их в порядке добавления? Проверим на простом примере.

Листинг 12.8 Элементы очереди с одинаковыми приоритетами

```

import asyncio
from asyncio import Queue, PriorityQueue
from dataclasses import dataclass, field

@dataclass(order=True)
class WorkItem:
    priority: int
    data: str = field(compare=False)

```

```

async def worker(queue: Queue):
    while not queue.empty():
        work_item: WorkItem = await queue.get()
        print(f'Обрабатывается элемент {work_item}')
        queue.task_done()

async def main():
    priority_queue = PriorityQueue()

    work_items = [WorkItem(3, 'Lowest priority'),
                  WorkItem(3, 'Lowest priority second'),
                  WorkItem(3, 'Lowest priority third'),
                  WorkItem(2, 'Medium priority'),
                  WorkItem(1, 'High priority')]

    worker_task = asyncio.create_task(worker(priority_queue))

    for work in work_items:
        priority_queue.put_nowait(work)

    await asyncio.gather(priority_queue.join(), worker_task)

asyncio.run(main())

```

Здесь в очередь сначала помещаются три низкоприоритетные задачи. Казалось бы, они должны быть обработаны в порядке вставки, но на самом деле это не так:

```

Обрабатывается элемент WorkItem(priority=1, data='High priority')
Обрабатывается элемент WorkItem(priority=2, data='Medium priority')
Обрабатывается элемент WorkItem(priority=3, data='Lowest priority third')
Обрабатывается элемент WorkItem(priority=3, data='Lowest priority second')
Обрабатывается элемент WorkItem(priority=3, data='Lowest priority')

```

Оказывается, что элементы обрабатываются в порядке, прямо противоположном порядку вставки. Причина в том, что алгоритм `heapsort` неустойчив, т. е. не гарантируется, что равные элементы будут располагаться в очереди в том порядке, в котором вставлялись. Часто порядок обработки элементов с равными приоритетами не имеет значения, но если он важен, то следует сформировать ключ, устраняющий неоднозначность и обеспечивающий нужное упорядочение. Простой способ сохранить порядок вставки – добавить в элемент его порядковый номер, хотя той же цели можно достичь многими другими способами.

Листинг 12.9 Устранение неоднозначности в очереди с приоритетами

```

import asyncio
from asyncio import Queue, PriorityQueue
from dataclasses import dataclass, field

@dataclass(order=True)

```

```
class WorkItem:
    priority: int
    order: int
    data: str = field(compare=False)

async def worker(queue: Queue):
    while not queue.empty():
        work_item: WorkItem = await queue.get()
        print(f'Обрабатывается элемент {work_item}')
        queue.task_done()

async def main():
    priority_queue = PriorityQueue()

    work_items = [WorkItem(3, 1, 'Lowest priority'),
                  WorkItem(3, 2, 'Lowest priority second'),
                  WorkItem(3, 3, 'Lowest priority third'),
                  WorkItem(2, 4, 'Medium priority'),
                  WorkItem(1, 5, 'High priority')]

    worker_task = asyncio.create_task(worker(priority_queue))

    for work in work_items:
        priority_queue.put_nowait(work)

    await asyncio.gather(priority_queue.join(), worker_task)

asyncio.run(main())
```

Мы добавили поле `order` в класс `WorkItem`. И при вставке элемента в очередь передаем целое число, описывающее его порядок. Для элементов с одинаковыми приоритетами неоднозначность разрешается с помощью этого числа. В нашем примере мы получаем требуемый порядок вставки низкоприоритетных элементов:

```
Обрабатывается элемент WorkItem(priority=1, order=5, data='High priority')
Обрабатывается элемент WorkItem(priority=2, order=4, data='Medium priority')
Обрабатывается элемент WorkItem(priority=3, order=1, data='Lowest priority')
Обрабатывается элемент WorkItem(priority=3, order=2, data='Lowest priority second')
Обрабатывается элемент WorkItem(priority=3, order=3, data='Lowest priority third')
```

Мы видели, как обрабатываются элементы FIFO-очереди и очереди с приоритетами. Но что, если требуется первыми обработать недавно добавленные элементы? Далее рассмотрим, как это делается с помощью LIFO-очереди.

12.3 LIFO-очереди

LIFO-очередь в информатике чаще называется *стеком*. В качестве метафоры можно воспользоваться покерными фишками: делая ставку, мы берем фишки с вершины стопки («выталкиваем» их), а выиграв

партию, помещаем фишки на вершину стопки («заталкиваем»). Это полезно, когда требуется, чтобы исполнители сначала обрабатывали элементы, добавленные последними.

Чтобы продемонстрировать, в каком порядке исполнители обрабатывают элементы, нам понадобится совсем простой пример. Что же касается того, когда использовать LIFO-очередь, то все зависит от порядка, в котором приложение должно обрабатывать элементы в очереди. Нужно ли обработать последний добавленный элемент первым? Если да, то LIFO-очередь вам в помощь.

Листинг 12.10 LIFO-очередь

```
import asyncio
from asyncio import Queue, LifoQueue
from dataclasses import dataclass, field

@dataclass(order=True)
class WorkItem:
    priority: int
    order: int
    data: str = field(compare=False)

async def worker(queue: Queue):
    while not queue.empty():
        work_item: WorkItem = await queue.get()
        print(f'Обрабатывается элемент {work_item}')
        queue.task_done()

async def main():
    lifo_queue = LifoQueue()

    work_items = [WorkItem(3, 1, 'Lowest priority first'),
                  WorkItem(3, 2, 'Lowest priority second'),
                  WorkItem(3, 3, 'Lowest priority third'),
                  WorkItem(2, 4, 'Medium priority'),
                  WorkItem(1, 5, 'High priority')]

    worker_task = asyncio.create_task(worker(lifo_queue))

    for work in work_items:
        lifo_queue.put_nowait(work)

    await asyncio.gather(lifo_queue.join(), worker_task)

asyncio.run(main())
```

Выбрать элемент из очереди,
или «вытолкнуть» его из стека

Поместить элемент в очередь,
или «затолкнуть» его в стек

Здесь мы создаем LIFO-очередь и набор элементов. Затем помещаем их в очередь один за другим, извлекаем и обрабатываем. При выполнении этой программы мы увидим следующую картину:

```
Обрабатывается элемент WorkItem(priority=1, order=5, data='High priority')
Обрабатывается элемент WorkItem(priority=2, order=4, data='Medium priority')
Обрабатывается элемент WorkItem(priority=3, order=3, data='Lowest priority third')
```

Обрабатывается элемент `WorkItem(priority=3, order=2, data='Lowest priority second')`
Обрабатывается элемент `WorkItem(priority=3, order=1, data='Lowest priority first')`

Обратите внимание, что элементы обрабатываются в порядке, противоположном тому, в каком вставлялись. Поскольку это стек, то так и должно быть – ведь первым обрабатывается элемент, помещенный в очередь первым.

Итак, мы рассмотрели все виды очередей, предлагаемые библиотекой `asynclio`. Есть ли какие-нибудь подвохи при их использовании? Можно ли их использовать в любом случае, когда в приложении нужна очередь? Эти вопросы мы рассмотрим в главе 13.

Резюме

- Очереди `asynclio` содержат задачи и полезны, когда в программе имеются сопрограммы, порождающие данные, и сопрограммы, отвечающие за обработку этих данных.
- Очередь отделяет порождение данных от их обработки, поскольку один производитель может помещать в очередь элементы, независимо и конкурентно обрабатываемые несколькими потребителями.
- Благодаря очередям с приоритетами мы можем назначать задачам разные приоритеты. Это полезно, когда какая-то работа важнее прочих и должна быть выполнена раньше.
- Очереди `asynclio` не являются ни распределенными, ни сохраняемыми, ни долговечными. Если эти свойства необходимы, то следует подумать об использовании отдельного архитектурного компонента, например `Celery` или `RabbitMQ`.

13

Управление подпроцессами

Краткое содержание главы

- Асинхронное выполнение нескольких подпроцессов.
- Обработка стандартного вывода подпроцесса.
- Взаимодействие с подпроцессами с помощью стандартного ввода.
- Предотвращение взаимоблокировок и других подводных камней в подпроцессах.

У многих приложений никогда не возникает необходимости покидать мир Python. Мы вызываем код из других Python-библиотек или модулей либо используем многопроцессность или многопоточность для конкурентного выполнения Python-кода. Но не всякий код, с которым нам доводится взаимодействовать, написан на Python. Возможно, уже имеется приложение, написанное на C++, Go, Rust или еще каком-нибудь языке, обладающее лучшими характеристиками в плане производительности. Ну или оно просто есть, и для чего писать то же самое еще раз? Кроме того, иногда возникает желание воспользоваться командными утилитами ОС, например `grep` для поиска в больших файлах, `cURL` для отправки HTTP-запросов или еще каким-то из мириад приложений, имеющихся в нашем распоряжении.

В стандартной библиотеке Python имеется модуль `subprocess`, позволяющий выполнять приложение в отдельном процессе. Как боль-

шинство модулей Python, `subprocess` предлагает блокирующий API, несовместимый с `asyncio`, если не прибегать к многопроцессной или многопоточной обработке. `asyncio` предлагает модуль, который повторяет функциональность `subprocess`, но создает подпроцессы и управляет ими асинхронно с помощью сопрограмм.

В этой главе мы изучим основы создания и управления подпроцессами в `asyncio` на примере запуска приложения, написанного на другом языке. Мы также научимся обрабатывать ввод и вывод, читать стандартный вывод и передавать ввод нашего приложения подпроцессам.

13.1 Создание подпроцесса

Пусть требуется расширить функциональность существующего веб-API, написанного на Python. Другая группа в организации уже разработала нужную вам функциональность в виде командного приложения для механизма пакетной обработки, вот только написано оно на Rust. Поскольку приложение уже существует, вы не хотите изобретать велосипед и переписывать его на Python. Нельзя ли как-нибудь воспользоваться существующей функциональностью из нашего Python API?

Поскольку у приложения имеется командный интерфейс, мы можем запустить его в отдельном подпроцессе, а затем прочитать полученные результаты и использовать его в нашем API. Так мы обойдемся без переписывания приложения.

Но как создать и выполнить подпроцесс? Библиотека `asyncio` предлагает две сопрограммы для создания подпроцессов: `asyncio.create_subprocess_shell` и `asyncio.create_subprocess_exec`. Обе они возвращают экземпляр класса `Process`, имеющего, наряду с прочими, методы для завершения и ожидания завершения процесса. Зачем нужны две сопрограммы для решения вроде бы одной и той же задачи? Когда использовать одну, а когда другую? Сопрограмма `create_subprocess_shell` создает подпроцесс внутри установленной в системе оболочки, например `zsh` или `bash`. Вообще говоря, лучше использовать `create_subprocess_exec`, если только не нужна функциональность оболочки. С использованием оболочки связаны различные подводные камни, например на разных машинах могут быть установлены разные оболочки или одна и та же, но по-разному сконфигурированная. Из-за этого трудно гарантировать, что приложение будет вести себя одинаково на разных машинах.

Чтобы понять, как создается подпроцесс, напишем приложение `asyncio` для выполнения простой командной программы. Начнем с программы `ls`, которая выводит содержимое текущего каталога, хотя на практике мы вряд ли стали бы это делать. Если вы работаете на машине с Windows, замените `ls -l` на `cmd /c dir`.

Листинг 13.1 Выполнение простой команды в подпроцессе

```
import asyncio
from asyncio.subprocess import Process

async def main():
    process: Process = await asyncio.create_subprocess_exec('ls', '-l')
    print(f'pid процесса: {process.pid}')
    status_code = await process.wait()
    print(f'Код состояния: {status_code}')

asyncio.run(main())
```

Здесь мы создаем экземпляр `Process`, чтобы запустить команду `ls` методом `create_subprocess_exec`. Можно также добавить после имени команды ее аргументы. В данном случае мы передаем аргумент `-l`, означающий, что нужно печатать дополнительную информацию, в частности о создателе файла. Создав процесс, мы печатаем его идентификатор и вызываем сопрограмму `wait`. Она будет ждать завершения процесса, после чего вернет код его состояния; в данном случае он должен быть равен нулю. По умолчанию стандартный вывод подпроцесса соединяется каналом со стандартным выводом приложения, поэтому при выполнении программы мы увидим такую картину (зависящую от того, из какого каталога программа была запущена):

```
pid процесса: 54438
total 8
drwxr-xr-x  4 matthewfowler staff 128 Dec 23 15:20 .
drwxr-xr-x 25 matthewfowler staff 800 Dec 23 14:52 ..
-rw-r--r--  1 matthewfowler staff   0 Dec 23 14:52 __init__.py
-rw-r--r--  1 matthewfowler staff 293 Dec 23 15:20 basics.py
Код состояния: 0
```

Заметим, что сопрограмма `wait` блокирует выполнение, до тех пор пока подпроцесс не завершится, а никаких гарантий касательно времени его работы нет, он может даже вообще не завершаться. Если зависание процесса вас беспокоит, то следует задать тайм-аут, вызвав сопрограмму `asyncio.wait_for`. Но тут есть подвох. Напомним, что `wait_for` завершает работающую сопрограмму по тайм-ауту. Если вы думаете, что при этом завершается и процесс, вынужден вас огорчить. Завершается только задача, которая ждет завершения процесса, но не сам процесс.

Нужен другой способ остановить процесс по истечении тайм-аута. По счастью, в классе `Process` есть два метода, которые нас выручат: `terminate` и `kill`. Метод `terminate` посылает подпроцессу сигнал `SIGTERM`, а метод `kill` – сигнал `SIGKILL`. Отметим, что оба эти метода не блокирующие и сопрограммами не являются. Они просто посылают сигнал. Если вы хотите получить код состояния завершившегося подпроцесса или дождаться завершения, чтобы произвести какую-то очистку, то нужно еще раз вызвать `wait`.

Протестируем завершение долго работающего приложения на примере командной утилиты `sleep` (в случае Windows замените `'sleep'`, `'3'` более длинным набором аргументов: `'cmd', 'start', '/wait', 'timeout', '3'`). Мы создаем подпроцесс, который несколько секунд спит, а затем пытаемся завершить его раньше, чем он завершится естественным образом.

Листинг 13.2 Завершение подпроцесса

```
import asyncio
from asyncio.subprocess import Process

async def main():
    process: Process = await asyncio.create_subprocess_exec('sleep', '3')
    print(f'pid процесса: {process.pid}')
    try:
        status_code = await asyncio.wait_for(process.wait(), timeout=1.0)
        print(status_code)
    except asyncio.TimeoutError:
        print('Тайм-аут, завершаю принудительно...')
        process.terminate()
        status_code = await process.wait()
        print(status_code)

asyncio.run(main())
```

Здесь мы создаем подпроцесс, который работает 3 с, но обертываем его сопрограммой `wait_for` с односекундным тайм-аутом. По истечении 1 с `wait_for` возбуждает исключение `TimeoutError`, и в блоке `except` мы завершаем процесс и ждем, когда он завершится, после чего печатаем код состояния. В результате должно быть напечатано что-то типа

```
pid процесса: 54709
Тайм-аут, завершаю принудительно...
-15
```

При написании собственного кода следует иметь в виду, что вызов `wait` внутри блока `except` тоже может работать долго; если это вас беспокоит, оберните его сопрограммой `wait_for`.

13.1.1 Управление стандартным выводом

В предыдущих примерах стандартный вывод подпроцесса направлялся напрямую на стандартный вывод приложения. Но что, если такое поведение нежелательно? Быть может, мы хотим подвергнуть вывод дополнительной обработке, или вывод нам не интересен и мы хотим его проигнорировать. У сопрограммы `create_subprocess_exec` есть параметр `stdout`, позволяющий указать, куда направлять стандартный вывод подпроцесса: на наш собственный стандартный вы-

вод, соединить каналом со `StreamReader` или игнорировать, перенаправив в `/dev/null`.

Допустим, мы собираемся запустить несколько подпроцессов конкурентно и скопировать их вывод на консоль. Во избежание путаницы нужно знать, какой подпроцесс сгенерировал каждую строку. Чтобы вывод было проще читать, мы предположим каждой выведенной строке сведения о породившем ее процессе, а именно команду и ее аргументы.

Для этого первым делом присвоим параметру `stdout` значение `asyncio.subprocess.PIPE`. Это значит, что подпроцесс должен создать новый экземпляр класса `StreamReader`, который можно будет использовать для чтения вывода процесса. Доступ к этому экземпляру дает поле `Process.stdout`. Прделаем все это для команды `ls -la`.

Листинг 13.3 Демонстрация читателя стандартного вывода

```
import asyncio
from asyncio import StreamReader
from asyncio.subprocess import Process

async def write_output(prefix: str, stdout: StreamReader):
    while line := await stdout.readline():
        print(f'[{prefix}]: {line.rstrip().decode()}')

async def main():
    program = ['ls', '-la']
    process: Process = await asyncio.create_subprocess_exec(*program,
                                                            stdout=asyncio.subprocess.PIPE)

    print(f'pid процесса: {process.pid}')
    stdout_task = asyncio.create_task(write_output(' '.join(program),
                                                  process.stdout))

    return_code, _ = await asyncio.gather(process.wait(), stdout_task)
    print(f'Процесс вернул: {return_code}')

asyncio.run(main())
```

Здесь сопрограмма `write_output` добавляет префикс в начало каждой строки вывода потокового читателя. Затем в сопрограмме `main` создается подпроцесс, стандартный вывод которого направляется в канал. Мы также создаем задачу, которая будет выполнять `write_output`, и передаем ей потоковый читатель стандартного вывода. Эта задача запускается конкурентно с помощью `wait`. При выполнении этой программы мы увидим, что в начало каждой строки, выведенной `ls`, добавлен префикс:

```
pid процесса: 56925
[ls -la]: total 32
[ls -la]: drwxr-xr-x  7 matthewfowler staff 224 Dec 23 09:07 .
[ls -la]: drwxr-xr-x 25 matthewfowler staff 800 Dec 23 14:52 ..
[ls -la]: -rw-r--r--  1 matthewfowler staff   0 Dec 23 14:52 __init__.py
Процесс вернул: 0
```

Важная особенность использования каналов и вообще работы с вводом и выводом подпроцессов – возможность взаимоблокировок. Особенно этому подвержена сопрограмма `wait` в случае, когда подпроцесс порождает большой объем вывода, а его потребление организовано некорректно. Для демонстрации рассмотрим простой пример: приложение Python, которое пишет много данных на стандартный вывод и опустошает сразу весь буфер.

Листинг 13.4 Порождение большого объема вывода

```
import sys

[sys.stdout.buffer.write(b'Привет!!\n') for _ in range(1000000)]

sys.stdout.flush()
```

Здесь мы 1 000 000 раз записываем в буфер стандартного вывода строку `Привет!!` и однократно опустошаем буфер. Посмотрим, что произойдет, если организовать канал связи с этим приложением, но не потреблять данные.

Листинг 13.5 Взаимоблокировка при использовании канала

```
import asyncio
from asyncio.subprocess import Process

async def main():
    program = ['python3', 'listing_13_4.py']
    process: Process = await asyncio.create_subprocess_exec(*program,
                                                            stdout=asyncio.subprocess.PIPE)

    print(f'pid процесса: {process.pid}')

    return_code = await process.wait()
    print(f'Процесс вернул: {return_code}')

asyncio.run(main())
```

Запустив эту программу, мы увидим напечатанный `pid` процесса, и больше ничего. Приложение зависает навсегда, завершить его можно только принудительно. Если в вашей системе такого не наблюдается, просто увеличьте число операций записи строки в буфер стандартного вывода – рано или поздно проблема проявится.

Такое простое приложение – откуда же взялась взаимоблокировка? Проблема связана с тем, как работает буфер потокового читателя. Когда буфер заполнен, любая попытка записать в него еще что-то приводит к блокированию программы до освобождения места в буфере. Хотя буфер читателя полон, наш процесс не оставляет попыток впихнуть в него все свои данные. Таким образом, завершение процесса оказывается зависимым от разблокировки потокового читателя, но этого никогда не произойдет, потому что мы не освобождаем

место в буфере. Налицо циклическая зависимость, а следовательно, взаимоблокировка.

Ранее этой проблемы не возникало, потому что мы конкурентно читали из потокового читателя стандартного вывода, ожидая завершения процесса. Поэтому, даже если бы буфер заполнился до упора, мы бы опустошили его и процесс не ждал бы бесконечно возможности записать новые данные. Работая с каналами, не забывайте потреблять данные из потока, чтобы не столкнуться с взаимоблокировкой.

Эту проблему можно также решить, отказавшись от использования сопрограммы `wait`. Кроме того, в классе `Process` имеется метод-сопрограмма `communicate`, который вообще избегает взаимоблокировок. Он блокирует выполнение, пока подпроцесс не завершится, и конкурентно потребляет стандартный вывод и стандартный вывод для ошибок, возвращая все, что было выведено, как только приложение завершится. Перепишем предыдущий пример, воспользовавшись методом `communicate`.

Листинг 13.6 Использование `communicate`

```
import asyncio
from asyncio.subprocess import Process

async def main():
    program = ['python3', 'listing_13_4.py']
    process: Process = await asyncio.create_subprocess_exec(*program,
                                                            stdout=asyncio.subprocess.PIPE)

    print(f'pid процесса: {process.pid}')

    stdout, stderr = await process.communicate()
    print(stdout)
    print(stderr)
    print(f'Процесс вернул: {process.returncode}')
    asyncio.run(main())
```

Выполнив эту программу, вы увидите, что весь вывод приложения печатается на консоли сразу (и один раз печатается `None`, потому что мы ничего не писали в стандартный вывод для ошибок). Под капотом `communicate` создает несколько задач, которые постоянно читают стандартный вывод во внутренний буфер, избегая тем самым взаимоблокировки. Но хотя проблему взаимоблокировки мы решили, это решение имеет серьезный недостаток: мы теперь не можем интерактивно обрабатывать данные из стандартного вывода. Если требуется реагировать на данные, выводимые приложением (например, завершать работу или запускать новую задачу, встретив определенное сообщение), то следует использовать `wait`, но при этом аккуратно читать из потокового читателя, чтобы избежать взаимоблокировки.

Еще один недостаток состоит в том, что `communicate` буферизует в памяти все данные из стандартного вывода и стандартного вывода для ошибок. Если подпроцесс порождает большой объем данных, то

возникает риск нехватки памяти. Как преодолеть эти недостатки, мы увидим в следующем разделе.

13.1.2 Конкурентное выполнение подпроцессов

Познакомившись с тем, как создавать, завершать и читать вывод подпроцессов, воспользуемся этими знаниями, чтобы выполнить несколько приложений конкурентно. Пусть требуется зашифровать несколько фрагментов текста, хранящегося в памяти, и для пущей безопасности мы хотим использовать шифр Twofish. Этот алгоритм не поддерживается модулем `hashlib`, поэтому нужна альтернатива. Можно воспользоваться командной программой `gpg` (GNU Privacy Guard, свободная замена программе PGP [pretty good privacy]). Скачать `gpg` можно по адресу <https://gnupg.org/download/>.

Сначала определим команду шифрования. При запуске `gpg` нужно задать парольную фразу и алгоритм, а затем подать на стандартный ввод подлежащий шифрованию текст. Например, чтобы зашифровать текст «encrypt this!», следует выполнить команду

```
echo 'encrypt this!' | gpg -c --batch --passphrase 3ncryptm3 --cipher-algo TWOFISH
```

В результате на стандартный вывод будет выведен зашифрованный текст:

```
?
Q+??/??*??C??H`??`?)R??u??7p_{f{R;n?FE .?b5??(?i?????o\k?b<????`%
```

При запуске из командной строки это работает, но не годится при использовании `create_subprocess_exec`, поскольку оператор канала `|` недоступен (метод `create_subprocess_shell` в этом случае будет работать). Так как же тогда передать подлежащий шифрованию текст? Помимо каналов со стандартным выводом и стандартным выводом для ошибок, методы `communicate` и `wait` позволяют организовать канал со стандартным вводом. Сопрограмма `communicate` также позволяет задать входные байты при запуске приложения. Если мы подадим данные на стандартный ввод в момент создания процесса, то они будут переданы приложению. Это как раз то, что нам нужно: просто передадим подлежащую шифрованию строку сопрограмме `communicate`.

Проверим эту идею, для чего сгенерируем случайные тексты и конкурентно зашифруем их. Создадим список из 100 случайных строк по 1000 символов в каждой и конкурентно запустим для каждой программу `gpg`.

Листинг 13.7 Конкурентное шифрование текста

```
import asyncio
import random
import string
import time
```

```

from asyncio.subprocess import Process

async def encrypt(text: str) -> bytes:
    program = ['gpg', '-c', '--batch', '--passphrase', '3ncryptn3',
               '--cipher-algo', 'TWOFISH']

    process: Process = await asyncio.create_subprocess_exec(*program,
                                                             stdout=asyncio.subprocess.PIPE,
                                                             stdin=asyncio.subprocess.PIPE)

    stdout, stderr = await process.communicate(text.encode())
    return stdout

async def main():
    text_list = [''.join(random.choice(string.ascii_letters) for _ in
                          range(1000)) for _ in range(100)]

    s = time.time()
    tasks = [asyncio.create_task(encrypt(text)) for text in text_list]
    encrypted_text = await asyncio.gather(*tasks)
    e = time.time()

    print(f'Время работы: {e - s}')
    print(encrypted_text)

asyncio.run(main())

```

Здесь определена сопрограмма `encrypt`, которая создает процесс `gpg` и с помощью `communicate` передает ему подлежащий шифрованию текст. Для простоты мы лишь возвращаем результат, записанный на стандартный вывод, и не пытаемся обрабатывать ошибки; реальное приложение следовало бы сделать более надежным. Затем в сопрограмме `main` мы создаем список случайных текстов и для каждого из них – задачу `encrypt`. Эти задачи конкурентно выполняются с помощью `gather`, после чего печатается время работы и зашифрованные тексты. Время конкурентной работы можно сравнить со временем синхронной работы, поместив `await` перед `asyncio.create_task` и убрав `gather`; выигрыш существенный.

В этой программе текстов всего 100. А если бы их были тысячи или больше? Сейчас мы пытаемся конкурентно шифровать 100 текстов, т. е. создаем 100 одновременно работающих процессов. Это проблема, так как ресурсы машины не безграничны, а каждый процесс может потреблять довольно много памяти. Кроме того, выполнение сотен или тысяч процессов влечет за собой нетривиальные затраты на контекстное переключение.

В нашем случае дополнительные сложности создает сама программа `gpg`, которой для шифрования данных необходимо разделяемое состояние. Если в листинге 13.7 увеличить число текстов до нескольких тысяч, то, скорее всего, вы увидите следующее сообщение, печатаемое на стандартном выводе для ошибок:

```
gpg: waiting for lock on '/Users/matthewfowler/.gnupg/random_seed'...
```


То есть мало того, что мы создали кучу процессов, пожирающих ресурсы, так эти процессы еще и блокируются при доступе к разделяемому состоянию. А как ограничить количество работающих процессов и обойти эту проблему? Вот отличный случай воспользоваться семафором. Поскольку программа ограничена быстродействием процессора, имеет смысл добавить семафор, чтобы число процессов не превышало число доступных ядер. Сделав это, попробуем зашифровать 1000 текстов и посмотрим, улучшится ли производительность.

Листинг 13.8 Подпроцессы и семафор

```
import asyncio
import random
import string
import time
import os
from asyncio import Semaphore
from asyncio.subprocess import Process

async def encrypt(sem: Semaphore, text: str) -> bytes:
    program = ['gpg', '-c', '--batch', '--passphrase', '3ncryptm3',
               '--cipher-algo', 'TWOFISH']

    async with sem:
        process: Process = await asyncio.create_subprocess_exec(*program,
                                                                stdout=asyncio.subprocess.PIPE,
                                                                stdin=asyncio.subprocess.PIPE)
        stdout, stderr = await process.communicate(text.encode())
        return stdout

async def main():
    text_list = [''.join(random.choice(string.ascii_letters) for _ in
                        range(1000)) for _ in range(1000)]
    semaphore = Semaphore(os.cpu_count())
    s = time.time()
    tasks = [asyncio.create_task(encrypt(semaphore, text)) for text in text_list]
    encrypted_text = await asyncio.gather(*tasks)
    e = time.time()

    print(f'Время работы: {e - s}')

asyncio.run(main())
```

Сравнив со временем шифрования 1000 текстов, когда число процессов было неограниченно, мы увидим некоторое повышение производительности и уменьшение потребления памяти. Если вы увидели здесь аналогию с идеей ограничения числа исполнителей с помощью `ProcessPoolExecutor`, рассмотренной в главе 6, то вы совершенно правы. Под капотом `ProcessPoolExecutor` пользуется семафором для управления количеством конкурентно работающих процессов.

Итак, мы познакомились с созданием, завершением и конкурентным выполнением нескольких процессов. Далее посмотрим, как можно интерактивно взаимодействовать с подпроцессами.

13.2 Взаимодействие с подпроцессами

До сих пор взаимодействие с подпроцессами было односторонним, а не интерактивным. А если мы работаем с приложением, требующим ввода данных пользователем? Например, приложение может запрашивать парольную фразу, имя пользователя и другие данные.

Если есть всего один элемент входных данных, то `communicate` подходит идеально. Мы уже видели это на примере передачи подлежащего шифрованию текста программе `grpg`, а теперь применим в случае, когда подпроцесс явно запрашивает ввод. Сначала напишем простую программу, которая запрашивает имя пользователя и копирует его на стандартный вывод.

Листинг 13.9 Копирование данных, введенных пользователем

```
username = input('Введите имя пользователя: ')
print(f'Вы ввели имя {username}')
```

Теперь воспользуемся `communicate` для ввода имени пользователя:

Листинг 13.10 Использование `communicate` со стандартным вводом

```
import asyncio
from asyncio.subprocess import Process

async def main():
    program = ['python3', 'listing_13_9.py']
    process: Process = await asyncio.create_subprocess_exec(*program,
                                                             stdout=asyncio.subprocess.PIPE,
                                                             stdin=asyncio.subprocess.PIPE)

    stdout, stderr = await process.communicate(b'Zoot')
    print(stdout)
    print(stderr)

asyncio.run(main())
```

При выполнении этой программы мы увидим на консоли строку `b'Введите имя пользователя: Вы ввели имя Zoot\n'`, потому что приложение завершается сразу после ввода имени. Если степень интерактивности приложения выше, то это не годится. Например, рассмотрим приложение, которое в цикле запрашивает данные и копирует их на стандартный вывод, пока пользователь не снимет программу.

Листинг 13.11 Приложение echo

```
user_input = ''

while user_input != 'quit':
    user_input = input('Введите текст: ')
    print(user_input)
```

Поскольку `communicate` ждет завершения процесса, необходимо использовать `wait` и обрабатывать стандартный вывод и стандартный ввод конкурентно. Класс `Process` предоставляет в поле `stdin` экземпляр `StreamWriter`, который можно использовать, если значением стандартного ввода является PIPE. В приложениях описанного выше типа мы можем конкурентно работать со `StreamWriter` и со `StreamReader`. Это продемонстрировано в следующем листинге, где подпроцессу передается несколько текстов.

Листинг 13.12 Запуск приложения echo в подпроцессе

```
import asyncio
from asyncio import StreamWriter, StreamReader
from asyncio.subprocess import Process

async def consume_and_send(text_list, stdout: StreamReader, stdin: StreamWriter):
    for text in text_list:
        line = await stdout.read(2048)
        print(line)
        stdin.write(text.encode())
        await stdin.drain()

async def main():
    program = ['python3', 'listing_13_11.py']
    process: Process = await asyncio.create_subprocess_exec(*program,
                                                            stdout=asyncio.subprocess.PIPE,
                                                            stdin=asyncio.subprocess.PIPE)

    text_input = ['one\n', 'two\n', 'three\n', 'four\n', 'quit\n']

    await asyncio.gather(consume_and_send(text_input, process.stdout,
                                          process.stdin), process.wait())

asyncio.run(main())
```

Здесь сопрограмма `consume_and_send` читает стандартный вывод, пока не получит ожидаемое сообщение, предлагающее пользователю ввести данные. Получив сообщение, она копирует его на стандартный вывод приложения, а очередную строку из списка `'text_list'` на стандартный ввод. Эти действия повторяются, пока все данные не будут переданы подпроцессу. При выполнении программы мы увидим, что весь вывод был передан подпроцессу и корректно скопирован на консоль:

```
b'Введите текст: '  
b'one\nВведите текст: '  
b'two\nВведите текст: '  
b'three\nВведите текст: '  
b'four\nВведите текст: '
```

Приложение, которое мы запустили, порождает детерминированный вывод и останавливается в известных случаях, где запрашивает ввод. Поэтому управлять стандартным выводом и стандартным вводом сравнительно просто. Но что, если приложение, запускаемое в подпроцессе, запрашивает входные данные лишь эпизодически или может выводить много данных, прежде чем попросит что-то ввести? Сделаем программу `echo` чуть более сложной. Она будет копировать данные, введенные пользователем случайное число раз от 1 до 10, и спать полсекунды между каждыми двумя операциями копирования.

Листинг 13.13 Более сложная программа `echo`

```
from random import randrange  
import time  
  
user_input = ''  
  
while user_input != 'quit':  
    user_input = input('Введите текст: ')  
    for i in range(randrange(10)):  
        time.sleep(.5)  
        print(user_input)
```

Если запустить это приложение в подпроцессе, как в листинге 13.12, то оно будет работать, потому что по-прежнему детерминировано в том смысле, что рано или поздно запросит ввод данных в ответ на известное сообщение. Но недостаток этого подхода в том, что код чтения стандартного вывода и записи на стандартный ввод сильно связан. В сочетании с усложненной логикой ввода-вывода это может сделать код трудным для понимания и сопровождения.

Проблему можно решить, отделив чтение стандартного вывода от записи на стандартный ввод. Напишем одну сопрограмму, которая будет читать стандартный вывод, и другую для записи на стандартный ввод. Первая сопрограмма будет устанавливать событие, получив ожидаемое приглашение к вводу. А вторая будет ждать события, после чего выводить заданный текст. Обе сопрограммы мы конкурентно запустим с помощью `gather`.

Листинг 13.14 Разделение чтения вывода и записи ввода

```
import asyncio  
from asyncio import StreamWriter, StreamReader, Event  
from asyncio.subprocess import Process
```

```
async def output_consumer(input_ready_event: Event, stdout: StreamReader):
    while (data := await stdout.read(1024)) != b'':
        print(data)
        if data.decode().endswith("Введите текст: "):
            input_ready_event.set()

async def input_writer(text_data, input_ready_event: Event, stdin: StreamWriter):
    for text in text_data:
        await input_ready_event.wait()
        stdin.write(text.encode())
        await stdin.drain()
        input_ready_event.clear()

async def main():
    program = ['python3', 'interactive_echo_random.py']
    process: Process = await asyncio.create_subprocess_exec(*program,
                                                            stdout=asyncio.subprocess.PIPE,
                                                            stdin=asyncio.subprocess.PIPE)

    input_ready_event = asyncio.Event()

    text_input = ['one\n', 'two\n', 'three\n', 'four\n', 'quit\n']

    await asyncio.gather(output_consumer(input_ready_event, process.stdout),
                        input_writer(text_input, input_ready_event,
                                    process.stdin),
                        process.wait())

asyncio.run(main())
```

Здесь мы сначала определяем сопрограмму `output_consumer`. Она принимает событие `input_ready_event`, а также объект `StreamReader`, ссылающийся на стандартный вывод, и читает стандартный вывод, пока не встретит строку `Введите текст: .` В этот момент мы знаем, что подпроцесс готов принимать данные из стандартного ввода, поэтому устанавливаем событие `input_ready_event`.

Сопрограмма `input_writer` обходит список входных строк и ждет готовности события `input_ready_event`. Как только подпроцесс будет готов к приему данных, мы записываем данные на стандартный ввод и очищаем событие, так что следующая итерация цикла `for` будет заблокирована в ожидании готовности к вводу. В этой реализации есть две сопрограммы, и у каждой своя четко определенная сфера ответственности: одна пишет на стандартный ввод, а другая читает стандартный вывод. Код стал понятнее и удобнее для сопровождения.

Резюме

- Модуль `subprocess` можно использовать для асинхронного запуска подпроцессов с помощью сопрограмм `create_subprocess_shell` и `create_subprocess_exec`. Лучше использовать `create_subprocess_`

ехес, потому что при этом гарантируется, что поведение программы не будет зависеть от машины, на которой она выполняется.

- По умолчанию вывод подпроцессов передается на стандартный вывод приложения. Если требуется взаимодействовать со стандартным вводом и стандартным выводом, то следует связать их с экземплярами классов `StreamReader` и `StreamWriter`.
- Организуя канал со стандартным выводом или стандартным выводом для ошибок, не забывайте потреблять вывод. В противном случае может произойти взаимоблокировка.
- Если конкурентно запускается много подпроцессов, то стоит использовать семафор, чтобы ограничить потребление системных ресурсов и избежать ненужной борьбы за ресурсы.
- Метод-сопрограмму `communicate` можно использовать для подачи данных на стандартный ввод подпроцесса.

14

Продвинутое использование *asynсio*

Краткое содержание главы

- Проектирование API одновременно для сопрограмм и функций.
- Контекстные локальные переменные сопрограмм.
- Уступка управления циклу событий.
- Использование различных реализаций цикла событий.
- Связь между сопрограммами и генераторами.
- Создание собственного цикла событий с помощью нестандартных объектов, допускающих ожидание.

Мы изучили почти все, что предлагает библиотека *asynсio*. Пользуясь описанными в предыдущих главах модулями, вы сможете решить почти любую встретившуюся задачу. Однако есть несколько не столь широко известных приемов, которые иногда оказываются полезными, особенно при проектировании собственных API *asynсio*.

В этой главе мы рассмотрим разнородные продвинутые способы использования *asynсio*. Мы узнаем, как проектировать API, способные работать как с сопрограммами, так и с обычными функциями Python; как принудительно перейти к следующей итерации цикла событий и как передавать информацию о состоянии между задачами, не передавая аргументов. Мы также более глубоко изучим вопрос о том, как в *asynсio* используются генераторы, чтобы лучше понять,

что происходит под капотом. Для этого реализуем собственные нестандартные объекты, допускающие ожидание, и воспользуемся ими для построения простой реализации цикла событий, способной конкурентно выполнять несколько сопрограмм.

Маловероятно, что материал этой главы пригодится вам в повседневной практике, если только вы не занимаетесь разработкой новых API или каркасов, которые опираются на внутренние механизмы асинхронного программирования. В основном описанные здесь вещи предназначены для таких приложений, а также для любопытных читателей, которые хотят глубже разобраться в устройстве асинхронности в Python.

14.1 API, допускающие сопрограммы и функции

Разрабатывая собственный API, мы не всегда хотим предполагать, что пользователи будут работать с нашей библиотекой только в асинхронных приложениях. Возможно, они еще не завершили миграцию, а быть может, не видят преимуществ в асинхронном стеке и никогда не перейдут на него. Как спроектировать API, который допускал бы как сопрограммы, так и обычные функции Python и тем самым адаптировался бы к обоим типам пользователей?

Библиотека *asyncio* предлагает для этого две функции: *asyncio.iscoroutine* и *asyncio.iscoroutinefunction*. Они позволяют проверить, является ли вызываемый объект сопрограммой. Именно эти функции дают возможность Django естественно работать с синхронными и асинхронными представлениями, как мы видели в главе 9.

Для демонстрации напомним простой класс исполнителя задач, принимающий как функции, так и сопрограммы. Он позволит добавлять вызываемые объекты во внутренний список и исполнять их конкурентно (если это сопрограммы) или последовательно (если это обычные функции), когда пользователь вызовет метод *run*.

Листинг 14.1 Класс исполнителя задач

```
import asyncio

class TaskRunner:

    def __init__(self):
        self.loop = asyncio.new_event_loop()
        self.tasks = []

    def add_task(self, func):
        self.tasks.append(func)

    async def _run_all(self):
        awaitable_tasks = []

        for task in self.tasks:
```



```

        if asyncio.iscoroutinefunction(task):
            awaitable_tasks.append(asyncio.create_task(task()))
        elif asyncio.iscoroutine(task):
            awaitable_tasks.append(asyncio.create_task(task))
        else:
            self.loop.call_soon(task)

    await asyncio.gather(*awaitable_tasks)

def run(self):
    self.loop.run_until_complete(self._run_all())

if __name__ == "__main__":

    def regular_function():
        print('Привет от регулярной функции!')

    async def coroutine_function():
        print('Выполняется сопрограмма, засыпаю!')
        await asyncio.sleep(1)
        print('Проснулась!')

    runner = TaskRunner()
    runner.add_task(coroutine_function)
    runner.add_task(coroutine_function())
    runner.add_task(regular_function)

    runner.run()

```

Исполнитель задач создает цикл событий и пустой список задач. Метод `add` добавляет функцию (или сопрограмму) в список ожидающих задач. Когда пользователь вызывает метод `run()`, мы выполняем метод `_run_all` в цикле событий. Этот метод перебирает задачи в списке и проверяет, является ли объект обычной функцией или сопрограммой. Если это сопрограмма, то мы создаем задачу, иначе вызываем метод цикла событий `call_soon`, чтобы запланировать выполнение функции на следующей итерации. Создав все задачи, мы должны вызвать для них сопрограмму `gather` и дождаться завершения.

Далее определяются две функции: обычная функция Python `regular_function` и сопрограмма `coroutine_function`. Мы создаем экземпляр `TaskRunner` и добавляем три задачи, вызывая `coroutine_function` дважды, чтобы продемонстрировать два разных способа сослаться на сопрограмму в нашем API. Программа печатает следующие строки:

```

Выполняется сопрограмма, засыпаю!
Выполняется сопрограмма, засыпаю!
Привет от регулярной функции!
Проснулась!
Проснулась!

```

Как видим, мы успешно выполнили как сопрограмму, так и обычную функцию Python и таким образом построили API, расширяющий возможности пользователей при работе с нашим API. Далее мы рас-

смотрим контекстные переменные, которые позволяют сохранять состояние, локальное для задачи, не передавая его явно в виде аргумента функции.

14.2 Контекстные переменные

Допустим, что мы используем REST API для доступа к веб-серверу, обрабатывающему каждый запрос в отдельном потоке. Предположим, что нас интересуют данные о пользователе, отправившем запрос, например идентификатор пользователя, маркер доступа или какая-то другая информация. Возникает соблазн хранить эти данные глобально и сделать доступными всем потокам сервера, но у такого подхода есть недостатки, главные из которых – необходимость как-то сопоставить потоку его данные и организация блокировок для предотвращения гонки. Эти проблемы можно решить с помощью *поточно-локальных переменных*. Это глобальные переменные на уровне одного потока. Данные, хранящиеся в поточно-локальной переменной, будут видны только потоку, сохранившему их, и таким образом решается как проблема сопоставления данных потоку, так и проблема синхронизации доступа. Мы не станем углубляться в детали поточно-локальных переменных, о них можно прочитать в документации по модулю `threading` по адресу <https://docs.python.org/3/library/threading.html#thread-local-data>.

Разумеется, в приложениях *asyncio* обычно имеется только один поток, поэтому любая поточно-локальная переменная доступна в любом месте приложения. В документе PEP-567 (<https://www.python.org/dev/peps/pep-0567>) описана концепция *контекстных переменных*, адаптирующая поточно-локальные переменные к модели однопоточной конкурентности. Контекстные переменные похожи на поточно-локальные, но локальны для задачи, а не для потока. Это означает, что если задача создает контекстную переменную, то к ней будет иметь доступ любая внутренняя сопрограмма или задача, созданная внутри задачи-создателя. А никакие задачи вне этой цепочки не смогут ни увидеть, ни модифицировать эту переменную. Это позволяет хранить состояние, связанное с конкретной задачей, не передавая его явно в виде аргумента.

Для демонстрации напомним простой сервер, который ожидает запросов от подключенных клиентов. Создадим контекстную переменную для хранения адреса клиента и при получении сообщения от него будем печатать само сообщение и адрес клиента.

Листинг 14.2 Сервер с контекстными переменными

```
import asyncio
from asyncio import StreamReader, StreamWriter
from contextvars import ContextVar
```

```

class Server:
    user_address = ContextVar('user_address')  ← Создать контекстную переменную
                                                с именем user_address

    def __init__(self, host: str, port: int):
        self.host = host
        self.port = port

    async def start_server(self):
        server = await asyncio.start_server(self._client_connected,
                                             self.host, self.port)

        await server.serve_forever()

    def _client_connected(self, reader: StreamReader, writer: StreamWriter):
        self.user_address.set(writer.get_extra_info('peername'))  ←
        asyncio.create_task(self.listen_for_messages(reader))

    async def listen_for_messages(self, reader: StreamReader):
        while data := await reader.readline():
            print(f'Получено сообщение {data} от {self.user_address.get()}')

    async def main():
        server = Server('127.0.0.1', 9000)
        await server.start_server()

        В момент подключения
        пользователя сохранить его адрес
        в контекстной переменной

asyncio.run(main())

```

Вывести сообщение
пользователя
и адрес
отправителя,
взятый из
контекстной
переменной

Здесь мы сначала создаем экземпляр класса `ContextVar`, в котором будем хранить информацию об адресе пользователя. Для этого нужно задать имя контекстной переменной, в данном случае мы назвали ее `user_address`, в основном для отладки. Затем в функции обратного вызова `_client_connected` мы записываем в эту контекстную переменную адрес клиента. Это позволит получить доступ к информации об адресе любой задаче, созданной родительской задачей; в данном случае таковыми являются задачи прослушивания сообщений от клиентов.

В методе-сопрограмме `listen_for_messages` мы ожидаем сообщения от клиента и, получив его, печатаем вместе с адресом, хранящимся в контекстной переменной. Если запустить это приложение, подключить к нему несколько клиентов и отправить сообщения, то мы увидим следующую картину:

```

Получено сообщение b'Hello!\r\n' от ('127.0.0.1', 50036)
Получено сообщение b'Okay!\r\n' от ('127.0.0.1', 50038)

```

Отметим, что номера портов различны, т. е. мы получили сообщения от двух разных клиентов, запущенных на машине `localhost`. И хотя мы создали всего одну контекстную переменную, все равно смогли получить уникальные данные о каждом подключенном клиенте. Это позволяет передавать данные задачам неявно, не прибегая к передаче аргументов.

14.3 Принудительный запуск итерации цикла событий

Мы почти никак не можем управлять внутренней работой цикла событий. Он решает, когда и как выполнять сопрограммы и задачи. Однако все-таки можно при необходимости запустить новую итерацию цикла событий. Это может оказаться полезно для долго работающих задач, чтобы избежать блокирования цикла событий (хотя в этом случае имеет смысл подумать об использовании потоков), или когда нужно, чтобы задача начала работать немедленно.

Напомним, что если создается несколько задач, то ни одна из них не начнет работать, пока программа не дойдет до предложения `await`, которое инициирует планирование задач в цикле событий и их запуск. Но что, если мы хотим, чтобы каждая задача начинала выполняться незамедлительно?

Asyncio предлагает оптимизированную идиому приостановки текущей сопрограммы и принудительного запуска новой итерации цикла событий – вызов `asyncio.sleep` с параметром 0. Опишем, как ее можно использовать для запуска задач сразу после создания. Напишем две функции – в одной `sleep` не используется, а в другой используется – и посмотрим, как они будут выполняться.

Листинг 14.3 Принудительный запуск итерации цикла событий

```
import asyncio
from util import delay

async def create_tasks_no_sleep():
    task1 = asyncio.create_task(delay(1))
    task2 = asyncio.create_task(delay(2))
    print('К задачам применяется gather:')
    await asyncio.gather(task1, task2)

async def create_tasks_sleep():
    task1 = asyncio.create_task(delay(1))
    await asyncio.sleep(0)
    task2 = asyncio.create_task(delay(2))
    await asyncio.sleep(0)
    print('К задачам применяется gather:')
    await asyncio.gather(task1, task2)

async def main():
    print('--- Без asyncio.sleep(0) ---')
    await create_tasks_no_sleep()
    print('--- С asyncio.sleep(0) ---')
    await create_tasks_sleep()

asyncio.run(main())
```

При выполнении этой программы наблюдается такая картина:

```
--- Без asyncio.sleep(0) ---
К задачам применяется gather:
засыпаю на 1 с
засыпаю на 2 с
сон в течение 1 с закончился
сон в течение 2 с закончился
--- С asyncio.sleep(0) ---
засыпаю на 1 с
засыпаю на 2 с
К задачам применяется gather:
сон в течение 1 с закончился
сон в течение 2 с закончился
```

Сначала мы создаем обе задачи и вызываем для них `gather`, не используя `asyncio.sleep(0)`; все работает как обычно – сопрограммы `delay` не вызываются, пока не будет выполнено предложение `gather`. Затем мы вставляем `asyncio.sleep(0)` после создания каждой задачи. Теперь сообщения от сопрограммы `delay` печатаются немедленно, еще до вызова `gather`. Вызов `sleep` принудительно запускает следующую итерацию цикла событий, что выливается в немедленное выполнение задачи.

Мы почти всюду использовали реализацию цикла событий, предоставляемую библиотекой `asyncio`. Однако есть и другие реализации, которые можно подставить при желании. Далее покажем, как использовать другие циклы событий.

14.4 Использование других реализаций цикла событий

Библиотека `asyncio` предлагает реализацию цикла событий по умолчанию, которой мы и пользовались до сих пор, но ничто не мешает взять другую реализацию, возможно, обладающую иными характеристиками. Это можно сделать несколькими способами. Первый – написать подкласс `AbstractEventLoop`, реализовав необходимые методы, создать экземпляр нового класса и сделать его циклом событий, вызвав функцию `asyncio.set_event_loop`. Если мы разрабатываем собственную реализацию, то такой подход имеет смысл, но, вообще-то, существуют готовые циклы событий. Рассмотрим одну такую реализацию, *uvloop*.

Итак, что такое цикл *uvloop* и почему может возникнуть желание его использовать? *uvloop* – реализация цикла событий, основанная на библиотеке *libuv* (<https://libuv.org>), на которой зиждется среда выполнения `node.js`. Поскольку *libuv* написана на C, она работает быстрее, чем код на чистом интерпретируемом Python. Следовательно, и *uv-*

loop может оказаться быстрее цикла событий, предлагаемого *asyncio* по умолчанию. Особенно наглядно это проявляется в приложениях, где используются сокеты и потоковый ввод-вывод. Ознакомиться с тестами производительности можно на странице проекта в github по адресу <https://github.com/magicstack/uvloop>. Отметим, что на момент написания книги *uvloop* был доступен только на платформах *nix.

Прежде всего установите последнюю версию *uvloop* командой

```
pip -Iv uvloop==0.16.0
```

Теперь мы напишем простой эхо-сервер, воспользовавшись *uvloop* в качестве реализации цикла событий.

Листинг 14.4 Использование *uvloop* в качестве цикла событий

```
import asyncio
from asyncio import StreamReader, StreamWriter
import uvloop

async def connected(reader: StreamReader, writer: StreamWriter):
    line = await reader.readline()
    writer.write(line)
    await writer.drain()
    writer.close()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(connected, port=9000)
    await server.serve_forever()

uvloop.install()  ← Установить uvloop в качестве цикла событий
asyncio.run(main())
```

Здесь мы вызываем функцию *uvloop.install()*, которая переключает реализации цикла событий. Можно было бы сделать это и вручную:

```
loop = uvloop.new_event_loop()
asyncio.set_event_loop(loop)
```

Важно только, чтобы этот вызов был произведен раньше, чем вызов *asyncio.run(main())*. Под капотом *asyncio.run* вызывает функцию *get_event_loop*, которая создает цикл событий, если он еще не существует. Если сделать это, до того как *uvloop* был установлен, то мы получим стандартный цикл событий, и последующая установка *uvloop* уже ничего не изменит.

Если хотите, можете проверить, дает ли установка *uvloop* в качестве цикла событий какое-нибудь увеличение производительности вашего конкретного приложения. В проекте *uvloop* на Github есть код тестирования производительности в терминах пропускной способности и числа запросов в секунду.

Мы показали, как использовать готовую реализацию цикла событий вместо предлагаемой по умолчанию. Далее посмотрим, как создать собственный цикл событий. Это позволит лучше понять, как работает цикл событий, а вместе с ним сопрограммы, задачи и будущие объекты.

14.5 Создание собственного цикла событий

Неочевидный аспект `asyncio` заключается в том, что концептуально она отделена от синтаксиса `async/await` и сопрограмм. Определение класса сопрограммы вообще находится вне модуля `asyncio`!

Сопрограммы и синтаксис `async/await` – концепции, не зависящие от средств их выполнения. В состав Python входит реализация цикла событий по умолчанию, предлагаемая `asyncio`, и именно ей мы до сих пор пользовались для выполнения сопрограмм, но ничто не мешает использовать любую другую реализацию, в том числе свою собственную. В предыдущем разделе мы видели, как подменить реализацию цикла событий из `asyncio` другой, обладающей лучшими (или, по крайней мере, иными) характеристиками производительности. Теперь поговорим о том, как написать собственную реализацию простого цикла событий, умеющую работать с неблокирующими сокетами.

14.5.1 Сопрограммы и генераторы

До появления ключевых слов `async` и `await` в версии Python 3.5 связь между сопрограммами и генераторами была очевидной. Чтобы разобраться в ней, напомним простую сопрограмму, которая спит в течение 1 с, применив старый синтаксис с декораторами и генераторами.

Листинг 14.5 Сопрограммы на основе генераторов

```
import asyncio

@asyncio.coroutine
def coroutine():
    print('Засыпаю!')
    yield from asyncio.sleep(1)
    print('Проснулась!')

asyncio.run(coroutine())
```

Вместо ключевого слова `async` мы применили декоратор `@asyncio.coroutine`, который говорит, что функция является сопрограммой, а вместо ключевого слова `await` – конструкцию `yield from`, знакомую по использованию генераторов. В современных версиях ключевые слова `async` и `await` – не более чем синтаксический сахар, обертывающий эту конструкцию.

14.5.2 Использовать сопрограммы на основе генераторов не рекомендуется

Отметим, что от сопрограмм на основе генераторов планируется полностью избавиться в версии Python 3.10. Их можно встретить в унаследованных базах данных, но писать новый асинхронный код в таком стиле не следует.

А почему вообще использование генераторов имело смысл в модели однопоточной конкурентности? Напомним, что сопрограмма должна приостанавливать работу, встретив блокирующую операцию, чтобы дать возможность поработать другим сопрограммам. Генераторы приостанавливают работу, встретив предложение уступки процессора `yield`. Это означает, что выполнение двух генераторов можно чередовать. Сначала мы даем возможность поработать первому генератору, пока не встретится `yield` (или на языке сопрограмм точка `await`), затем работает второй генератор, тоже до точки `yield`, после чего все повторяется до исчерпания обоих генераторов. Чтобы посмотреть, как это выглядит на практике, напомним простой пример с чередованием двух генераторов и будем использовать в нем методы, которые понадобятся при построении цикла событий.

Листинг 14.6 Чередование генераторов

```
from typing import Generator

def generator(start: int, end: int):
    for i in range(start, end):
        yield i

one_to_five = generator(1, 5)
five_to_ten = generator(5, 10)

def run_generator_step(gen: Generator[int, None, None]):
    try:
        return gen.send(None)
    except StopIteration as si:
        return si.value

while True:
    one_to_five_result = run_generator_step(one_to_five)
    five_to_ten_result = run_generator_step(five_to_ten)
    print(one_to_five_result)
    print(five_to_ten_result)

if one_to_five_result is None and five_to_ten_result is None:
    break
```

Выполнить один шаг генератора

Чередовать выполнение двух генераторов

Здесь мы написали простой генератор, который пробегает диапазон целых чисел от начального до конечного, отдавая по дороге значения. Затем создаем два экземпляра этого генератора: один считает от одного до четырех, а другой от пяти до девяти.

Мы также написали вспомогательную функцию `run_generator_step`, которая выполняет один шаг генератора. В классе генератора есть метод `send`, который продвигает генератор к следующей точке `yield`, выполняя весь промежуточный код. После вызова `send` можно быть уверенным, что генератор приостановлен до следующего вызова `send`, что позволяет выполнить код в других генераторах. Метод `send` принимает параметр, равный значению, которое мы хотим передать генератору в качестве аргумента. В данном случае мы не хотим ничего передавать, поэтому параметр равен `None`. Дойдя до конца, генератор возбуждает исключение `StopIteration`. Это исключение содержит значение, возвращенное итератором, и мы его возвращаем. Наконец, запускаем оба генератора в бесконечном цикле. В итоге выполнение генераторов чередуется, так что мы наблюдаем следующий вывод:

```
1
5
2
6
3
7
4
8
None
9
None
None
```

Но представьте, что мы не отдаем числа, а уступаем процессор какой-то медленной операции. По завершении этой операции генератор можно будет возобновить с прерванного места, а тем временем не приостановленные генераторы могут выполнять свой код. Эта идея и положена в основу цикла событий. Мы следим за тем, какие генераторы приостановили выполнение, встретив медленную операцию. Пока один генератор приостановлен, другие могут работать. По завершении медленной операции приостановленный генератор можно возобновить, вызвав для него `send` и тем самым продвинув к следующей точке `yield`.

Как уже было сказано, `async` и `await` – всего лишь синтаксический сахар вокруг генераторов. Это можно продемонстрировать, создав экземпляр сопрограммы и вызвав его метод `send`. В качестве примера напишем две сопрограммы, которые будут просто печатать сообщения, и третью сопрограмму, которая будет вызывать две другие с помощью `await`. А затем воспользуемся методом генератора `send` для вызова сопрограмм.

Листинг 14.7 Использование `send` для сопрограмм

```
async def say_hello():
    print('Привет!')
```

```

async def say_goodbye():
    print('Пока!')

async def meet_and_greet():
    await say_hello()
    await say_goodbye()

coro = meet_and_greet()

coro.send(None)

```

При выполнении этой программы мы увидим такую картину:

```

Привет!
Пока!
Traceback (most recent call last):
  File "chapter_14/listing_14_7.py", line 16, in <module>
    coro.send(None)
StopIteration

```

Вызов метода `send` для нашей сопрограммы запускает все сопрограммы в `meet_and_greet`. Поскольку нам негде приостанавливаться в ожидании результата, весь код выполняется немедленно, даже тот, которому предшествует слово `await`.

А как заставить сопрограмму приостановиться и проснуться после выполнения медленной операции? Для этого покажем, как создать допускающий ожидание объект, чтобы можно было использовать синтаксис `await` вместо основанного на генераторах.

14.5.3 *Нестандартные объекты, допускающие ожидание*

Как определить объект, допускающий ожидание, и как он работает? Для этого в классе нужно реализовать метод `__await__`, но как это сделать? Что он должен возвращать?

К методу `__await__` предъявляется единственное требование – он должен возвращать итератор, но само по себе это требование не особенно полезно. Можно ли придать понятию итератора какой-то смысл в контексте цикла событий? Чтобы понять, как все работает, реализуем свою версию класса `asyncio Future`, которую назовем `CustomFuture`. Мы будем использовать ее в собственной реализации цикла событий.

Напомним, что объект `Future` – это обертка вокруг значения, которое будет доступно когда-то в будущем. Поэтому он может иметь два состояния: `завершен` и `не завершен`. Допустим, что мы находимся в бесконечном цикле событий и хотим с помощью итератора проверить, `завершен` ли будущий объект. Если операция завершилась, то можно просто вернуть результат и на этом покончить с итератором. Если нет, нам нужен способ сообщить: «Я еще не закончилась, проверь попозже». И в таком случае итератор может просто отдать себя самого!

Именно так будет реализован метод `__await__` в нашем классе `CustomFuture`. Если результат еще не готов, то итератор отдаст сам объект `CustomFuture`, а если готов, то вернет результат и на этом закончит работу. Если результат не готов, то в следующий раз, когда мы попытаемся продвинуть итератор, снова будет выполнен код внутри `__await__`. Еще нам нужен метод для добавления обратного вызова, который будущий объект сможет вызвать, когда значение будет получено. Он понадобится для реализации цикла событий.

Листинг 14.8 Реализация класса `CustomFuture`

```
class CustomFuture:

    def __init__(self):
        self._result = None
        self._is_finished = False
        self._done_callback = None

    def result(self):
        return self._result

    def is_finished(self):
        return self._is_finished

    def set_result(self, result):
        self._result = result
        self._is_finished = True
        if self._done_callback:
            self._done_callback(result)

    def add_done_callback(self, fn):
        self._done_callback = fn

    def __await__(self):
        if not self._is_finished:
            yield self
        return self.result()
```

Здесь определен класс `CustomFuture` с методом `__await__`, а также методами для установки результата, получения результата и добавления обратного вызова. Метод `__await__` проверяет, получено ли значение будущего объекта. Если да, то мы просто возвращаем результат и итератор завершается. Если нет, то мы возвращаем `self`, т. е. итератор будет отдавать себя до тех пор, пока значение не будет установлено. В терминах генераторов это означает, что `__await__` может вызываться бесконечно, пока кто-то не установит значение.

Рассмотрим простой пример, показывающий, как это может выглядеть в цикле событий. Мы создадим экземпляр класса `CustomFuture` и после двух итераций установим его значение, а на каждой итерации будем вызывать `__await__`.

Листинг 14.9 Использование объекта CustomFuture в цикле

```
from listing_14_8 import CustomFuture

future = CustomFuture()

i = 0

while True:
    try:
        print('Проверяется будущий объект...')
        gen = future.__await__()
        gen.send(None)
        print('Будущий объект не готов...')
        if i == 2:
            print('Устанавливается значение будущего объекта...')
            future.set_result('Готово!')
            i = i + 1
    except StopIteration as si:
        print(f'Значение равно: {si.value}')
        break
```

Здесь мы создали объект `CustomFuture` и в цикле вызываем его метод `await`, после чего пытаемся продвинуть итератор. Если будущий объект готов, то будет возбуждено исключение `StopIteration`, содержащее его значение. В противном случае итератор просто вернет сам будущий объект, и мы перейдем к следующей итерации. В нашем примере значение устанавливается на третьей итерации, так что печатаются следующие сообщения:

```
Проверяется будущий объект...
Будущий объект не готов...
Проверяется будущий объект...
Будущий объект не готов...
Устанавливается значение будущего объекта...
Проверяется будущий объект...
Значение равно: Готово!
```

Этот пример лишь показывает, как нужно рассуждать об объектах, допускающих ожидание. На практике мы не стали бы писать такой код, поскольку обычно значение будущего объекта устанавливается где-то в другом месте. Далее мы сделаем нечто более полезное, воспользовавшись сокетами и модулем `selectors`.

14.5.4 Сокеты и будущие объекты

В главе 3 мы немного узнали о модуле `selectors`, который позволяет регистрировать функции, вызываемые по событию сокета, например создание нового подключения или готовность данных для чтения. Теперь мы распространим эти знания на наш класс `CustomFuture`, разре-

шив ему взаимодействовать с селекторами и устанавливать значение будущего объекта в ответ на событие.

Идея регистрации обратного вызова при возникновении события прекрасно сочетается с созданным нами классом. Мы можем зарегистрировать метод `set_result` в качестве обратного вызова для чтения из сокета. Желая асинхронно ждать данных от сокета, мы создадим будущий объект, зарегистрируем его метод `set_result` с помощью модуля `selectors` и вернем этот объект. Затем сможем ждать его с помощью `await` и получить результат, когда селектор вызовет нашу функцию.

Для демонстрации напомним приложение, которое будет ждать подключения к неблокирующему сокету. Когда подключение будет установлено, мы просто вернем его и позволим приложению завершиться.

Листинг 14.10 Сокеты и будущие объекты

```
import functools
import selectors
import socket
from listing_14_8 import CustomFuture
from selectors import BaseSelector

def accept_connection(future: CustomFuture, connection: socket):
    print(f'Получен запрос на подключение от {connection}!')
    future.set_result(connection)

async def sock_accept(sel: BaseSelector, sock) -> socket:
    print('Регистрируется сокет для прослушивания подключений')
    future = CustomFuture()
    sel.register(sock, selectors.EVENT_READ,
        functools.partial(accept_connection, future))
    print('Прослушиваю запросы на подключение...')
    connection: socket = await future
    return connection

async def main(sel: BaseSelector):
    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    sock.bind(('127.0.0.1', 8000))
    sock.listen()
    sock.setblocking(False)

    print('Ожидая подключения к сокету!')
    connection = await sock_accept(sel, sock)
    print(f'Получено подключение {connection}!')

selector = selectors.DefaultSelector()

coro = main(selector)
```

Установить сокет для чтения-записи
данных в будущем объекте, когда придет
запрос на подключение от клиента

Зарегистрировать
в селекторе функцию
`accept_connection`
и приостановиться
в ожидании запроса на
подключение

Ждать, когда клиент
подключится

```

while True:
    try:
        state = coro.send(None)

        events = selector.select()

        for key, mask in events:
            print('Обрабатываются события селектора...')
            callback = key.data
            callback(key.fileobj)
    except StopIteration as si:
        print('Приложение завершилось!')
        break

```

В бесконечном цикле вызывать метод `send` главной сопрограммы. При каждом событии селектора выполнять зарегистрированный обратный вызов

Здесь в начале определена функция `ассепт_connection`, которая принимает объект `CustomFuture` и клиентский сокет. Мы печатаем сообщение о том, что получен запрос на подключение, после чего устанавливаем сокет в качестве значения будущего объекта. Следующая далее функция `sock_ассепт` принимает серверный сокет и селектор и регистрирует функцию `ассепт_connection` (привязанную к объекту `CustomFuture`) в качестве обратного вызова для событий чтения в серверном сокете. После этого мы ждем будущий объект с помощью `await` и, когда подключение будет установлено, возвращаем его.

В сопрограмме `main` мы создаем серверный сокет, а затем выполняем `await` для сопрограммы `sock_ассепт` в ожидании подключения. Когда запрос на подключение поступит, мы печатаем сообщение и завершаем приложение. Таким образом, мы написали *минимально работоспособный цикл событий*. Мы создаем экземпляр сопрограммы `main`, передаем ему селектор и входим в бесконечный цикл. В цикле первым делом вызываем `send`, чтобы продвинуть `main` к первому предложению `await`, а затем вызываем функцию `selector.select`, которая блокирует выполнение в ожидании подключения клиента. После этого вызываются зарегистрированные обратные вызовы – в нашем случае только `ассепт_connection`. Как только клиент подключился, мы вызываем `send` во второй раз, при этом все сопрограммы снова продвигаются вперед, что дает возможность приложению завершиться. Если выполнить эту программу и подключиться через `telnet`, то мы увидим такую картину:

```

Ожидаю подключения к сокету!
Регистрируется сокет для прослушивания подключений
Прослушиваю запросы на подключение...
Обрабатываются события селектора...
Получен запрос на подключение от <socket.socket fd=4, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8000)>!
Получено подключение <socket.socket fd=4, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8000)>!
Приложение завершилось!

```

Мы написали простое асинхронное приложение, используя только ключевые слова `async` и `await`, но не используя `asyncio`! Цикл `while`

в конце программы – пример простого цикла событий, который демонстрирует принцип работы цикла событий в `asyncio`. Разумеется, без задач от конкурентности толку будет немного.

14.5.5 Реализация задачи

Задача представляет собой комбинацию будущего объекта и сопрограммы. Будущий объект задачи завершается, когда завершается обернутая им сопрограмма. Обернуть сопрограмму будущим объектом можно, унаследовав классу `CustomFuture` и написав конструктор, который принимает сопрограмму, но необходим еще способ *выполнить* сопрограмму. Это можно сделать, написав метод `step`, который будет вызывать метод сопрограммы `send` и запоминать результат, т. е. выполнять один шаг сопрограммы при каждом вызове.

При реализации этого метода следует иметь в виду, что `send` может возвращать также другие будущие объекты. Чтобы обработать эту ситуацию, мы должны использовать метод `add_done_callback` любого будущего объекта, возвращаемого `send`. Мы зарегистрируем обратный вызов, который будет вызывать метод `send` сопрограммы задачи по готовности будущего объекта и передавать ему результирующее значение.

Листинг 14.11 Реализация задачи

```
from chapter_14.listing_14_8 import CustomFuture
```

```
class CustomTask(CustomFuture):
```

```
    def __init__(self, coro, loop):
        super(CustomTask, self).__init__()
        self._coro = coro
        self._loop = loop
        self._current_result = None
        self._task_state = None
        loop.register_task(self)
```

Зарегистрировать задачу
в цикле событий

```
    def step(self):
        try:
            if self._task_state is None:
                self._task_state = self._coro.send(None)
            if isinstance(self._task_state, CustomFuture):
                self._task_state.add_done_callback(self._future_done)
        except StopIteration as si:
            self.set_result(si.value)
```

Если сопрограмма отдает будущий
объект, вызвать `add_done_callback`

```
    def _future_done(self, result):
        self._current_result = result
        try:
            self._task_state = self._coro.send(self._current_result)
        except StopIteration as si:
            self.set_result(si.value)
```

Когда будущий объект будет готов,
отправить результат сопрограмме

Здесь мы создаем подкласс `CustomFuture`, конструктор которого принимает сопрограмму и цикл событий и регистрирует задачу в цикле, вызывая метод `loop.register_task`. Затем в методе `step` мы вызываем метод `send` сопрограммы и если сопрограмма отдает объект типа `CustomFuture`, то добавляем обратный вызов `done`. В данном случае `done` принимает результат будущего объекта и отправляет его обернутой сопрограмме, продвигая ее вперед в момент, когда будущий объект оказывается готов.

14.5.6 Реализация цикла событий

Теперь мы знаем, как выполняются сопрограммы, и реализовали будущие объекты и задачи. Это дает нам все необходимое для построения цикла событий. Как должен выглядеть API событий, чтобы можно было написать асинхронное приложение для работы с сокетами? Нам нужно несколько методов:

- метод, принимающий главную сопрограмму, определяющую точку входа, похожий на `asyncio.run`;
- методы, которые принимают соединения, получают данные и закрывают сокет. Эти методы будут регистрировать и отменять регистрацию сокета в селекторе;
- метод для регистрации `CustomTask`; это тот же код, который мы ранее использовали в конструкторе `CustomTask`.

Сначала поговорим о главной точке входа, назовем этот метод `run`. Это рабочая лошадка цикла событий. Он принимает сопрограмму `main` и в бесконечном цикле вызывает ее метод `send` и запоминает результат генератора. Если `main` возвращает будущий объект, то мы добавляем обратный вызов `done`, чтобы запомнить результат этого объекта, когда он будет готов. Затем вызывается метод `step` всех зарегистрированных задач и проверяется, были ли события на сокетах селектора. Для каждого события выполняются ассоциированные с ним обратные вызовы, после чего начинается следующая итерация цикла. Если где-то в сопрограмме `main` возбуждается исключение `StopIteration`, значит, приложение завершилось и мы можем выйти, вернув значение, полученное в составе исключения.

Далее нам понадобятся методы сопрограмм для приема запросов на подключение к сокету и получения данных из клиентского сокета. Наша стратегия заключается в том, чтобы создать экземпляр `CustomFuture`, результат которого установит обратный вызов, и зарегистрировать этот обратный вызов в селекторе, чтобы он вызывался по событиям чтения. Затем мы переходим к ожиданию будущего объекта.

Наконец, необходим метод для регистрации задач в цикле событий. Он принимает задачу и добавляет ее в список. Затем на каждой итерации цикла событий мы будем вызывать метод `step` для всех зарегистрированных задач, продвигая те из них, которые готовы. Реализовав все это, мы получим минимально жизнеспособный цикл событий.

Листинг 14.12 Реализация цикла событий

```

import functools
import selectors
from typing import List
from chapter_14.listing_14_11 import CustomTask
from chapter_14.listing_14_8 import CustomFuture

class EventLoop:
    _tasks_to_run: List[CustomTask] = []

    def __init__(self):
        self.selector = selectors.DefaultSelector()
        self.current_result = None

    def _register_socket_to_read(self, sock, callback):
        future = CustomFuture()
        try:
            self.selector.get_key(sock)
        except KeyError:
            sock.setblocking(False)
            self.selector.register(sock, selectors.EVENT_READ,
                                  functools.partial(callback, future))
        else:
            self.selector.modify(sock, selectors.EVENT_READ,
                                 functools.partial(callback, future))
        return future

    def _set_current_result(self, result):
        self.current_result = result

    async def sock_recv(self, sock):
        print('Регистрируется сокет для прослушивания данных...')
        return await self._register_socket_to_read(sock, self.recieved_data)

    async def sock_accept(self, sock):
        print('Регистрируется сокет для приема подключений...')
        return await self._register_socket_to_read(sock,
                                                    self.accept_connection)

    def sock_close(self, sock):
        self.selector.unregister(sock)
        sock.close()

    def register_task(self, task):
        self._tasks_to_run.append(task)

    def recieved_data(self, future, sock):
        data = sock.recv(1024)
        future.set_result(data)

    def accept_connection(self, future, sock):
        result = sock.accept()
        future.set_result(result)

```

Зарегистрировать в селекторе сокет для событий чтения

Зарегистрировать сокет для получения данных от клиента

Зарегистрировать задачу в цикле событий

Зарегистрировать сокет для приема запросов на подключение от клиентов

Выполнять
главную
сопрограмму,
пока она не завершится,
и на каждой итерации
выполнять готовые
к работе задачи

```

→ def run(self, coro):
    self.current_result = coro.send(None)

    while True:
        try:
            if isinstance(self.current_result, CustomFuture):
                self.current_result.add_done_callback(
                    self._set_current_result)
                if self.current_result.result() is not None:
                    self.current_result =
                        coro.send(self.current_result.result())
            else:
                self.current_result = coro.send(self.current_result)
        except StopIteration as si:
            return si.value

    for task in self._tasks_to_run:
        task.step()

    self._tasks_to_run = [task for task in self._tasks_to_run if not
                           task.is_finished()]

    events = self.selector.select()
    print('В селекторе есть событие, обрабатывается...')
    for key, mask in events:
        callback = key.data
        callback(key.fileobj)

```

Сначала определяется вспомогательный метод `_register_socket_to_read`. Он принимает сокет и обратный вызов и регистрирует их в селекторе, если сокет еще не зарегистрирован. Если же сокет зарегистрирован, то мы заменяем обратный вызов. Первым аргументом обратного вызова должен быть будущий объект, и в этом методе мы создаем его и привязываем к обратному вызову. Этот привязанный будущий объект возвращается, так что теперь вызывающая сторона может ожидать его с помощью `await`, приостанавливая выполнение до готовности будущего объекта.

Затем следуют методы сопрограммы для получения данных из сокета и приема запросов на подключение: `sock_recv` и `sock_accept`. Они вызывают написанный ранее метод `_register_socket_to_read` и передают ему обратные вызовы для обработки данных и новых подключений (эти вызовы просто записывают полученные данные в будущий объект).

И наконец, метод `run`. Он принимает главную сопрограмму (точку входа) и вызывает ее метод `send`, продвигая к первой точке приостановки, после чего сохраняет результат `send`. Затем мы входим в бесконечный цикл, где первым делом проверяем, является ли результат главной сопрограммы объектом типа `CustomFuture`; если да, то мы регистрируем обратный вызов для сохранения результата, который затем при необходимости можно будет отправить назад главной сопрограмме. В противном случае мы просто отправляем результат со-

программе. Разобравшись с главной сопрограммой, выполняем все зарегистрированные в цикле событий задачи, вызывая их метод `step`. После того как задачи получили возможность отработать, мы удаляем завершившиеся из списка задач.

Наконец, мы вызываем метод `selector.select`, который блокирует выполнение, пока на каком-то из зарегистрированных сокетов не произойдет событие. Затем в цикле перебираем все возникшие события и для каждого вызываем функцию обратного вызова, зарегистрированную для соответствующего сокета в методе `_register_socket_to_read`. В нашей реализации любое событие сокета приводит к очередной итерации цикла событий. Итак, мы реализовали собственный класс `EventLoop` и готовы написать первое асинхронное приложение без использования `asyncio`!

14.5.7 Реализация сервера с использованием своего цикла событий

Располагая циклом событий, мы разработаем простое серверное приложение, которое будет протоколировать сообщения, полученные от подключившихся клиентов. Мы создадим серверный сокет и напишем сопрограмму для прослушивания запросов на подключение в бесконечном цикле. Получив запрос, мы создаем задачу, которая будет читать данные от клиента, пока тот не отключится. Выглядит это очень похоже на то, что было сделано в главе 3, только теперь мы используем собственный цикл событий, а не тот, что предоставляет `asyncio`.

Листинг 14.13 Реализация сервера

```
import socket

from chapter_14.listing_14_11 import CustomTask
from chapter_14.listing_14_12 import EventLoop

async def read_from_client(conn, loop: EventLoop):
    print(f'Чтение данных от клиента {conn}')
    try:
        while data := await loop.sock_recv(conn):
            print(f'Получены данные {data} от клиента!')
    finally:
        loop.sock_close(conn)

async def listen_for_connections(sock, loop: EventLoop):
    while True:
        print('Ожидание подключения...')
        conn, addr = await loop.sock_accept(sock)
        CustomTask(read_from_client(conn, loop), loop)
        print(f'Новое подключение к сокету {sock}!')
```

Читать и протоколировать данные от клиента

Прослушивать запросы на подключение и создавать задачу для чтения данных от подключившегося клиента

```
async def main(loop: EventLoop):
```

```

server_socket = socket.socket()
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

server_socket.bind(('127.0.0.1', 8000))
server_socket.listen()
server_socket.setblocking(False)

await listen_for_connections(server_socket, loop)

event_loop = EventLoop() ← Создать экземпляр цикла событий
event_loop.run(main(event_loop)) и выполнять в нем сопрограмму main

```

Здесь мы сначала определяем сопрограмму, которая в цикле читает и печатает данные от клиента. Также определяем сопрограмму, которая в бесконечном цикле прослушивает запросы на подключение к серверному сокету и создает экземпляр `CustomTask`, чтобы конкурентно читать данные от подключившегося клиента. В сопрограмме `main` мы создаем серверный сокет и вызываем сопрограмму `listen_for_connections`. Затем создаем экземпляр цикла событий и передаем его методу `run` сопрограммы `main`.

Запустив эту программу, мы сможем одновременно подключить к ней несколько клиентов `telnet` и отправлять сообщения серверу. Вот что мы увидим, подключив двух клиентов и отправив несколько тестовых сообщений:

```

Ожидание подключения...
Регистрируется сокет для приема подключений...
В селекторе есть событие, обрабатывается...
Новое подключение к сокету <socket.socket fd=4, family=AddressFamily.AF_INET,
    type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8000)>!
Ожидание подключения...
Регистрируется сокет для приема подключений...
Чтение данных от клиента <socket.socket fd=7, family=AddressFamily.AF_INET,
    type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8000),
    raddr=('127.0.0.1', 58641)>
Регистрируется сокет для прослушивания данных...
В селекторе есть событие, обрабатывается...
Получены данные b'test from client one!\r\n' от клиента!
Регистрируется сокет для прослушивания данных...
В селекторе есть событие, обрабатывается...
Новое подключение к сокету <socket.socket fd=4, family=AddressFamily.AF_INET,
    type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8000)>!
Ожидание подключения...
Регистрируется сокет для приема подключений...
Чтение данных от клиента <socket.socket fd=8, family=AddressFamily.AF_INET,
    type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8000),
    raddr=('127.0.0.1', 58645)>
Регистрируется сокет для прослушивания данных...
В селекторе есть событие, обрабатывается...
Получены данные b'test from client two!\r\n' от клиента!
Регистрируется сокет для прослушивания данных...

```

Мы видим, что первый клиент подключается, в результате чего селектор возобновляет работу сопрограммы `listen_for_connections`, приостановленную на обращении к `loop.sock_accept`. При этом в селекторе регистрируется клиентское подключение, т. е. мы создаем задачу для сопрограммы `read_from_client`. Первый клиент отправляет сообщение «test from client one!», что заставляет селектор выполнить все зарегистрированные обратные вызовы. В данном случае мы продвигаем вперед задачу `read_from_client`, которая выводит полученное от клиента сообщение на консоль. Затем подключается второй клиент, и весь процесс повторяется.

Конечно, эта реализация цикла событий далеко не совершенна (в частности, мы толком не обрабатываем исключения и иницилируем итерации цикла только по событиям сокета), но она дает представление о внутреннем устройстве цикла событий и асинхронного программирования в Python. В качестве упражнения можете написать готовый к эксплуатации цикл событий, применив все полученные знания. Быть может, вам удастся создать каркас асинхронного программирования на Python следующего поколения.

Резюме

- Мы можем проверить, является ли допускающий вызов аргумент сопрограммой и таким образом создать API, способный работать как с сопрограммами, так и с обычными функциями.
- Используйте контекстные локальные переменные, когда необходимо разделять состояние между несколькими сопрограммами, не передавая его в виде параметров.
- Сопрограмму `asyncio.sleep` можно использовать для принудительного перехода к следующей итерации цикла событий. Это полезно, когда нужно понудить цикл событий к какому-то действию, но естественная точка `await` отсутствует.
- Библиотека `asyncio` – это просто стандартная реализация цикла событий в Python. Существуют и другие реализации, например `uvloop`, и мы можем подменять одну на другую, не отказываясь от синтаксиса `async` и `await`. Мы также можем создать собственный цикл событий, если требуется нечто с особыми характеристиками.

Предметный указатель

Символы

`_add_user` функция, 246
`@asyncio.coroutine` декоратор, 373
`_await_` метод, 61, 377
`_change_state` метод, 325
(GIL) глобальная блокировка
интерпретатора, 22, 33
`_make_requests` функция, 212
`_poll_queue` метод, 215
`_queue_update` метод, 215
`_run_all` метод, 367
`_start` метод, 215
`_update_bar` метод, 215

А

`AbstractEventLoop` класс, 371
`AbstractServer` объект, 242
акцепт метод, 81
`accept_connection` функция, 380
`acquire` метод, 200, 310
`add` метод, 367
`add_done_callback` метод, 381
`add_one` функция, 47
`add_signal_handler` метод, 95
`after` метод, 215

`aiohttp`

задание тайм-аутов, 107
общие сведения, 102
отправка веб-запроса, 105
REST API. См. *REST API*
`aio-lib` проект, 102
`aiohttp` библиотека, 131
`ALL_COMPLETED` режим, 120
API
 блокирующий, 67
 допускающий сопрограммы
 и функции, 366
 REST API. См. *REST API*
`Application` класс, 256
`apply_async` метод, 161
`Array` объект, 181
`as_completed` функция, 117
ASGI (асинхронный интерфейс
серверного шлюза), 263
 реализация в `Starlette`, 264
 оконечная REST-точка, 265
 WebSockets, 266
 сравнение с WSGI, 263
`async` ключевое слово, 46, 373
`async for` цикл, 155
`asyncio`, 21

- глобальная блокировка
- интерпретатора (GIL), 33
- задание тайм-аута и снятие с помощью `wait_for`, 57
- измерение времени выполнения сопрограммы с помощью декораторов, 62
- и микросервисы, 281
- исполнители пула процессов, 162
 - в сочетании с циклом событий, 164
 - частичное применение функции, 164
- конкурентное выполнение с помощью задач, 52
 - конкурентное выполнение нескольких задач, 53
- конкурентность, 25
- многозадачность, 28
- моделирование длительных операций с помощью `sleep`, 49
- объекты, допускающие ожидание, 59
 - будущие объекты, 59
 - связь между будущими объектами, задачами и сопрограммами, 61
- ограниченность
- производительностью
- ввода-вывода и ограниченность быстродействием процессора, 24
- однопоточная конкурентность, 39
- отладочный режим, 70
 - использование аргументов командной строки, 71
 - использование переменных окружения, 71
 - использование `asyncio.run`, 70
- параллелизм, 26
- построение отзывчивого UI, 209
- поток, 194
 - библиотека `requests`, 194
 - исполнители по умолчанию, 198
 - исполнители пула потоков, 195
 - общие сведения, 29
- приложение, 74
 - блокирующие сокеты, 75
 - использование модуля `selectors` для построения цикла событий сокетов, 86
 - неблокирующие сокеты, 82
 - остановка сервера, 94
 - подключение к серверу с помощью `Telnet`, 78
 - эхо-сервер средствами цикла событий `asyncio`, 89
- продвинутое использование, 365
 - использование других реализаций цикла событий, 371
 - контекстные переменные, 368
 - принудительный запуск итерации цикла событий, 370
 - создание собственного цикла событий, 373
 - API, допускающие сопрограммы и функции, 366
- процессы, 29
- снятие задач, 56
- создание задач, 52
- сопрограммы, 25
- цикл событий
 - ручное управление, 68
 - создание вручную, 69
- MapReduce. См. *MapReduce*
- `asyncio.create_task` функция, 52
- `asyncio.get_event_loop` функция, 70
- `asyncio.get_running_loop` функция, 69
- `asyncio.iscoroutine`, 366
- `asyncio.iscoroutinefunction`, 366
- `asyncio.new_event_loop` функция, 69
- `asyncio.queue.QueueEmpty` исключение, 332
- `asyncio.run` функция, 70
- `asyncio.run_coroutine_threadsafe` функция, 211
- `asyncio.set_event_loop` функция, 371
- `asyncio.shield` функция, 58
- `asyncio.sleep` функция, 49
- `asyncpg`
 - вложенные транзакции, 148
 - выполнение запросов, 135
 - общие сведения, 131

ручное управление
транзакциями, 149
транзакции, 146
asyncpg.connect функция, 132
asyncpg Record объект, 137
async with блок, 106, 144, 313
await ключевое слово, 46, 373
AWS (Amazon Web Servers), 170

В

backend-for-frontend паттерн
 общие сведения, 282
 реализация, 289
BaseSelector класс, 86

С

call_soon, 367
cancel метод, 123, 212
cbreak режим, 235
channels библиотека, 270
ChatServerState класс, 245
checkout_customer сопрограмма, 330
clear метод, 318
ClientEchoThread класс, 193
close метод, 194, 307
concurrent.futures библиотека, 162, 195, 211
CONNECT команда, 244
connect функция, 143
Connection класс, 153, 325
connection_lost метод, 230
coroutine_function сопрограмма, 367
count функция, 163
counter переменная, 201
create_subprocess_shell
сопрограмма, 351
CREATE TABLE команда, 136
cursor метод, 153
CustomFuture класс, 376

D

def ключевое слово, 46
DefaultSelector класс, 86

delay параметр, 116
delay функция, 50
DOMContentLoaded событие, 269
do_work сопрограмма, 323
drain метод, 229

E

echo задача, 92
echo функция, 191
encrypt задача, 358
eof_received метод, 226
Event класс, 318
EventLoop класс, 385
execute метод, 325
Executor абстрактный класс, 162, 195
executor параметр, 198

F

fetch_status функция, 116
FileServer класс, 320
FileUpload класс, 319
find_and_replace метод, 204
fire_event сопрограмма, 323
FIRST_COMPLETED режим, 124
FIRST_EXCEPTION режим, 122
Flask, 260
for цикл, 108, 136, 167, 363
functools модуль, 164
functools.reduce функция, 168
future класс, 211

G

gather функция, 111
get метод, 162
get_inventory сопрограмма, 301
get_products_with_inventory
функция, 293
get_response_item_count
функция, 293
get_status_code функция, 198
Google Books Ngram, набор
данных, 169
grid метод, 215

unicorn команда, 274

Н

hashlib библиотека, 217

heapsort алгоритм, 346

HTTPGetClientProtocol класс, 228

I

id функция, 274

import предложение, 203

init функция, 182

initargs параметр, 182

initializer параметр, 182

input функция, 231

input_ready_event событие, 363

input_writer сопрограмма, 363

INSERT команда, 136

J

join метод, 31, 159

K

KeyboardInterrupt исключение, 95

kill команда, 95

L

libuv библиотека, 371

LIFO-очереди, 347

listen_for_connections

сопрограмма, 386

listen_for_messages метод, 369

LoadTester приложение, 215

loop.set_default_executor метод, 198

M

mainloop метод, 216

make_request метод, 228

make_request функция, 61

map операция, 167

map_frequencies функция, 173

MapReduce, 166

набор данных Google Books

Ngram, 169

применение asyncio для

отображения и редукции, 170

простой пример, 167

match_info словарь, 257

max_failure параметр, 299

max_size параметр, 143

mean функция, 220

mean_for_row функция, 221

min_size параметр, 143

multiprocessing библиотека, 158

multiprocessing.cpu_count()

функция, 161

N

new_event_loop метод, 216

next функция, 153

notify_all метод, 323

NumPy, 220

O

on_cleanup обработчик, 257

onmessage обратный вызов, 269

on_startup обработчик, 257

open_connection функция, 229

Order класс, 345

output_consumer функция, 363

P

paused массив, 88

pending множество задач, 127

Pool.apply_async метод, 163

Postgres база данных, 131

prefetch параметр, 154

Process.stdout поле, 354

Process класс, 159, 356

process_page функция, 340

protocol переменная, 228

put метод, 332

PYTHONASYNCIODEBUG переменная
окружения, 71

Q

queue модуль, 213

R

read сопрограмма, 235
read_from_client задача, 387
recv метод, 78
reduce функция, 173
regular_function функция, 367
release метод, 200, 310
REPL (цикл чтения-вычисления-печати), 29
Request класс, 255
request метод, 299
requests оконечная точка, 277
REST API, 252
 определение, 252
 основы серверов на базе
 aiohttp, 253
 подключение к базе данных
 и получение результатов, 255
 сравнение aiohttp и Flask, 260
result метод, 60, 121
results переменная, 137
return предложение, 152
return_when параметр, 119, 124
RLock класс, 203
Route объект, 265
RouteTableDef декоратор, 253
run метод, 192, 382
run_forever метод, 216
run_in_executor метод, 198
run_in_new_loop функция, 187

S

SAVEPOINT команда, 148
script функция, 217
select модуль, 190
selectors модуль, 378
send метод, 375
sendall метод, 78
set метод, 318
setcbreak функция, 235
set_result метод, 60, 379

shutdown метод, 192
SIGKILL сигнал, 352
signal.signal функция, 95
SIGTERM сигнал, 352
sleep команда, 353
sleep функция, 74
sock_accept функция, 380
socket модуль, 75
socket.accept метод, 90
Starlette класс, 265
Starlette, реализация ASGI, 264
start метод, 31, 149, 159, 212
stdin поле, 361
stdout параметр, 353
step метод, 382
StopIteration исключение, 375
StreamReader класс, 229
StreamWriter класс, 229
StressTest объект, 215
submit метод, 198
sync_to_async функция, 275

T

target функция, 32
Task объект, 121
task.exception() метод, 120
task.result() метод, 120
Telnet, 78
 несколько подключений
 и блокирование, 80
 чтение данных из сокета и запись
 данных в сокет, 79
terminate метод, 352
Thread класс, 190
threading модуль, 190
timeout параметр, 126
Tkapp_Call, 213
Tkinter, 207
TooManyRetries исключение, 295
tty модуль, 235

U

UDP протокол, 76
util модуль, 64

V

Value объект, 181

W

wait функция, 119
наблюдение за исключениями, 122
обертывание сопрограмм
задачами, 127
обработка результатов по мере
завершения, 123
обработка тайм-аутов, 126
ожидание завершения задач, 119
wait_for метод, 324
wait_for функция, 57
WebSockets, 266
WorkItem класс, 340
write метод, 229
WSGI (интерфейс шлюза
веб-сервера), 263

A

Асинхронное получение
результатов, 161
Асинхронное представление, 275
Асинхронные генераторы, 151
и потоковые курсоры, 153
общие сведения, 151
Асинхронные контекстные
менеджеры, 103, 143
задание тайм-аутов в aiohttp, 107
отправка веб-запроса с помощью
aiohttp, 105
Асинхронные очереди, 327
в веб-приложениях, 335
в веб-роботе, 338
с приоритетами, 341
типа LIFO, 347

Б

База данных, подключение, 255
Базовые сервисы, 284
Блокировки
в многопоточном коде, 200

взаимоблокировки, 204
реентерабельные, 201
синхронизация, 179, 309
Блокирующие сокеты, 75, 80
Будущие объекты
и сокеты, 378
общие сведения, 59
связь между будущими объектами,
задачами и сопрограммами, 61

B

Веб-приложения, 251
асинхронные представления
Django, 269
очереди, 335
разработка REST API с помощью
aiohttp, 252
ASGI (асинхронный интерфейс
серверного шлюза), 263
Взаимоблокировки, 204
Взаимодействие
с подпроцессами, 360

Г

Генераторы
асинхронные, 151
и потоковые курсоры, 153
общие сведения, 151
создание собственного цикла
событий, 373
Главный поток, 67

Д

Декораторы, 63
Демоны, 192
Дополняемые будущие объекты, 60
Драйверы баз данных
неблокирующие, 130
асинхронные генераторы, 151
и потоковые курсоры, 153
общие сведения, 151
подключение к базе данных
Postgres, 131

пулы подключений и конкурентное
выполнение запросов, 138
asynsrg
 выполнение запросов, 135
 общие сведения, 131
 транзакции, 146

З

Задачи, 59
 будущие объекты, 59
 конкурентное выполнение, 52, 108
 ловушки, 65
 обработка ошибок, 92
 ожидание завершения, 119
 остановка, 96
 реализация, 381
 связь с будущими объектами и
 сопрограммами, 61
 снятие, 56
 уведомление с помощью
 событий, 317
Значения, 176

И

Инициализаторы пула процессов, 181
Исключения
 наблюдение за, 122
 при использовании gather, 113
Исполнители по умолчанию, 198
Исполнители пула потоков, 68
 и asynsrg, 197
 общие сведения, 195
Исполнители пула процессов, 162

К

Канал, 232
Квантование времени, 30
Командная строка
 неблокирующий ввод данных, 231
 отладочный режим, 70
Конкурентность
 общие сведения, 25
 однопоточная, природа

 ошибок, 304
 отличие от параллелизма, 26
Конкурентные веб-запросы, 101
Контекстное переключение, 28
Контекстные переменные, 368
Кооперативная многозадачность, 28
Критическая секция, 179
Курсоры, 151, 234

М

Массивы, 176
Масштабируемость, 281
Микросервисы, 279
 паттерн backend-for-frontend, 282
 причины существования, 280
API списка товаров, 283
 паттерн Прерыватель, 297
 потвор неудачных запросов, 294
 реализация базовых сервисов, 284
 реализация сервиса backend-for-
 frontend, 289
 сервис избранного, 284
Многозадачность, 28
 вытесняющая, 28
 кооперативная, 28
 преимущества, 28
Многопоточность
 библиотека hashlib, 217
 библиотека NumPy, 220
Многопроцессность, 184
Монолиты, 279
Мьютекс (взаимное исключение), 179

Н

Неблокирующие сокеты, 82
Неблокирующий ввод данных из
командной строки, 231
Независимость от команды
и технологического стека, 281
Неудачные запросы, 294

О

Обещания, 60

Объекты, допускающие ожидание, 59
нестандартные, 376
связь между будущими объектами,
задачами и сопрограммами, 61
Объекты разделяемой памяти, 175
Ограниченные семафоры, 315
Однопоточная конкурентность, 39,
304
Однопоточный цикл событий, 23
Остановка, 94
ожидание завершения начатых
задач, 96
прослушивание сигналов, 95
Отладочный режим, 70
Очереди
с приоритетами, 341

П

Параллелизм
общие сведения, 26
отличие от конкурентности, 27
Переменные окружения, 71
Подключения
несколько, 80
пулы, 138
вставка случайных SKU в базу
данных о товарах, 138
для конкурентного выполнения
запросов, 142
Подпроцессы, 350
взаимодействие с, 360
создание, 351
конкурентное выполнение, 357
управление стандартным
выводом, 353
Подсчет ссылок, 34
Потоки, 29, 189
блокировки, 200
использование потоков для
выполнения счетных задач, 217
и asyncio, 194
библиотека requests, 194
исполнители пула потоков, 195
модуль threading, 190

циклы событий в отдельных
потоках, 206
построение отзывчивого UI, 209
Tkinter, 207
Потоки данных, 223
неблокирующий ввод данных из
командной строки, 231
общие сведения, 224
потокочные читатели
и писатели, 228
создание чат-сервера и его
клиента, 244
транспортные механизмы
и протоколы, 224
Потоки и asyncio
исполнители по умолчанию, 198
Потоковый курсор, 153
Поточно-локальные
переменные, 368
Прерыватель паттерн, 297
Приостановка выполнения, 48
Производитель-потребитель,
технологический процесс, 328
Протоколы, 224
Процессы, 29
Пул подключений, 106
Пулы процессов, 67, 160
асинхронное получение
результатов, 161
использование исполнителей
пула процессов в сочетании с
asyncio, 162
разделение данных, 181
Пульсирующая нагрузка, 315

Р

Разделение данных, 175
блокировки, 200
в пулах процессов, 181
синхронизация с помощью
блокировок, 179
состояния гонки, 176
Режим терминала без обработки, 235
Ресурсы, 252

С

Сеансы, 105
Семафоры, 313
Сигналы, 95
Синхронизация, 303
 блокировки, 179, 309
 ограничение уровня
 конкурентности с помощью
 семафоров, 312
 природа ошибок в модели
 однопоточной
 конкурентности, 304
 уведомление задач с помощью
 событий, 317
 условия, 322
Синхронные представления, 277
Сложность кода, 281
Снятие задач
 общие сведения, 56
 с помощью `wait_for`, 57
Сокеты, 39
 блокирующие, 75
 и будущие объекты, 378
 неблокирующие, 82
 циклы событий
 использование модуля
 selectors, 86
 сопрограммы для сокетов, 89
 чтение и запись данных, 79
Соль, 218
Сопрограммы, 39, 47, 59
 будущие объекты, 59
 выполнение блокирующих API, 67
 выполнение счетного кода, 65
 измерение времени выполнения
 с помощью декораторов, 62
 конкурентное выполнение
 с помощью задач, 52
 конкурентное выполнение
 нескольких задач, 53
 создание задач, 52
ловушки, 65
моделирование длительных
операций с помощью `sleep`, 49

 нерекомендованное использование
 сопрограмм на основе
 генераторов, 374
 приостановка выполнения
 с помощью ключевого слова
 `await`, 48
 связь между будущими объектами,
 задачами и сопрограммами, 61
 создание с помощью ключевого
 слова `async`, 46
Состояние гонки, 34
Схема базы данных, 133
Счетные задачи, 157
 библиотека `multiprocessing`, 158
 использование потоков, 217
 набор данных Google Books
 Ngram, 169
 несколько процессов и несколько
 циклов событий, 184
 общие сведения, 24
 применение `asyncio` для
 отображения и редукции, 170
 простой пример MapReduce, 167
 пулы процессов. См. *Пулы процессов*
 разделение данных. См. *Разделение*
 данных
 MapReduce и `asyncio`, 166

Т

Тайм-ауты
 в сочетании с `as_completed`, 117
 в `aiohttp`, 107
 задание, 57
 функция `wait`, 126
Транспортные механизмы, 224

У

Управляющие коды, 236
Условия, 322

Ф

Фабрика протоколов, 232

Ц

Цикл событий, 41, 68
в отдельном потоке, 206
использование альтернативных
реализаций, 371
использование модуля selectors для
построения, 86
несколько, 184

принудительный запуск
итерации, 370
создание собственного, 373

Ч

Частичное применение функции, 164
Чат, сервер и клиент, 244

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, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@aliens-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: **<http://www.galaktika-dmk.com/>**.

Мэттью Фаулер

Asyncio и конкурентное программирование на Python

Главный редактор	<i>Мовчан Д. А.</i>
	<i>dmkpress@gmail.com</i>
Зам. главного редактора	<i>Сенченкова Е. А.</i>
Перевод	<i>Слинкин А. А.</i>
Корректор	<i>Абросимова Л. А.</i>
Верстка	<i>Чаннова А. А.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 28,58. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**