ChatGPT

Асинхронность и многозадачность в Python 3.10+

Асинхронное программирование в Python позволяет писать **конкурентный** код, при котором несколько задач могут выполняться вперемешку в рамках одного потока, не блокируя друг друга. В этом учебном документе рассмотрены ключевые концепции asyncio (Python 3.10+), связанные с асинхронностью и многозадачностью. Мы обсудим виды многозадачности, корутины, задачи (Task), работу цикла событий (event loop), очереди задач, «проблему цветов» функций, потоки и процессы, а также разберём примеры кода. Кодовые блоки представлены с синтаксис-Highlight, а важные моменты выделены.

1. Вытесняющая и кооперативная многозадачность

Многозадачность – способность выполнять несколько задач “одновременно” (либо действительно параллельно, либо чередуя их исполнение). Существует два основных типа многозадачности:

- **Вытесняющая (preemptive)** – переключение между задачами контролируется внешним планировщиком (например, операционной системой). Задача может быть приостановлена в **любом месте** (по таймеру или системному событию) без её согласия. Потоки (threads) в современных ОС обычно работают по вытесняющей схеме: планировщик ОС прерывает текущий поток и переключается на другой по своему усмотрению.
- **Кооперативная (cooperative)** – каждая задача **самостоятельно уступает** управление, зная, что есть другие задачи. Переключение происходит только в специально обозначенных точках (например, при вызове `await` в Python). Код, работающий кооперативно, «знает», что выполняется не один, и сам решает, когда передать управление другим задачам [1]. Иными словами, задача должна **явно уступить** управление, чтобы другие могли выполняться.

  Важно! В кооперативной модели планировщик **не может принудительно прервать** работу задачи посередине. Пока задача не уступит управление (например, не выполнит операцию ожидания ввода-вывода или `await`), она будет продолжать выполняться и блокировать остальные. Цикл событий не способен асинхронно прервать выполняющуюся корутину – он сможет переключиться на другую лишь когда текущая корутина добровольно "yield-ит" управление [2]. В отличие от этого, при вытесняющей многозадачности (потоках) код может быть прерван планировщиком **в любой момент** [1].

Отличие от потоков: В Python потоки (`threading.Thread`) реализуют (почти) вытесняющую многозадачность: интерпретатор пытается переключаться между потоками на уровне байткода (учитывая GIL). Потоки управляются ОС, и теоретически могут исполняться параллельно на разных ядрах (но в CPython реальная параллельность CPU ограничена GIL). Asyncio же работает в одном потоке, реализуя кооперативную многозадачность: только одна задача выполняется в каждый момент времени, но задачи **чередуются** на одном потоке. Преимущество asyncio – отсутствие накладных расходов на синхронизацию потоков и отсутствие гонок данных (все задачи в одном потоке), однако если одна корутина ведёт себя плохо (не делает `await` и

выполняет долгую вычислительную работу), она блокирует весь цикл событий. В вытесняющей модели (потоки) другая задача могла бы получить управление принудительно, а в кооперативной – нет без сотрудничества задачи.

2. Корутины: определение и выполнение

Корутина – это специальная функция, которая может *приостанавливать* своё выполнение, не завершаясь, и позже *возобновлять* его с того же места. В Python 3.5+ корутины определяются с помощью ключевого слова `async def`. Пример корутинной функции:

```
import asyncio

async def приветствие():
    print("Привет...", end="")
    await asyncio.sleep(1)  # имитация паузы (не блокирует поток!)
    print("мир!")

# Вызовем корутину:
coro_obj = приветствие()
print(coro_obj)  # Выведет что-то вроде: <coroutine object приветствие at
0x7f9e4d0c7...>
```

Запуск этой программы не напечатает сразу "Привет, мир!". При вызове `приветствие()` **код внутри неё не выполняется** мгновенно – вместо этого возвращается объект-корутина (тип `coroutine`) [3]. Этот объект представляет отложенное выполнение. Чтобы *выполнить* корутину, её нужно передать планировщику asyncio. Самые распространённые способы выполнить корутину:

- **Высокоуровневый:** вызвать `asyncio.run(coro_obj)`, который создаст цикл событий, выполнит корутину до завершения и закроет цикл [4].
- **Низкоуровневый:** получить текущий цикл событий и зарегистрировать корутину как задачу (Task), либо явно запустить через `loop.run_until_complete`. Например: `asyncio.get_event_loop().run_until_complete(coro_obj)`.

Если корутина вызывает другие корутины через `await`, то выполнение будет передаваться им и возвращаться назад при их завершении. Корутины позволяют писать асинхронный код в линейном стиле, используя `await` для паузы.

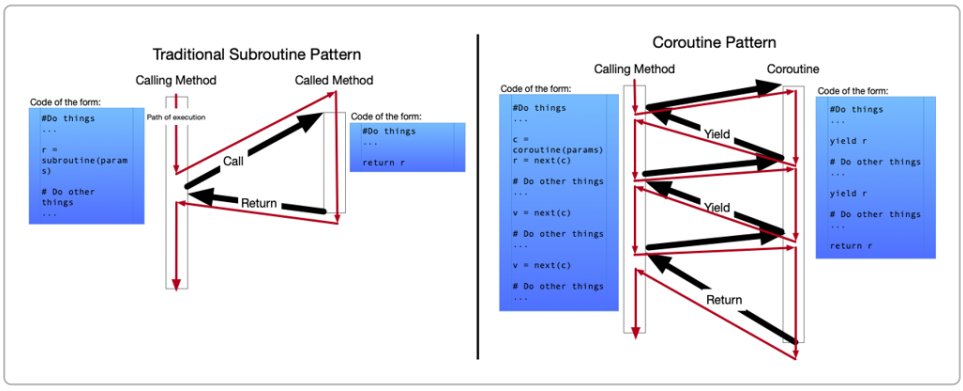


Диаграмма ниже сравнивает выполнение обычной функции (слева) и корутины (справа). В случае традиционной подпрограммы (subroutine) вызывающая функция передаёт управление вызываемой и ждёт до её возврата (прямой путь “Call → Return”). В случае корутины (coroutine pattern) вызов происходит похожим образом, но корутина может **временно вернуть управление обратно** вызывающему коду (операция “Yield” на диаграмме) не завершаясь, а затем вызывающий код позже возобновит её выполнение. Чёрные стрелки показывают поток выполнения: видно, что для корутины управление несколько раз переключается между вызывающим кодом и корутиной (несколько *Yield*), прежде чем корутина окончательно завершится (*Return*).

Чтобы проиллюстрировать, как корутины выполняются, вызовем асинхронно нашу функцию `приветствие` два раза:

```
async def main():
    # Запустим две корутины одновременно
    await asyncio.gather(приветствие(), приветствие())
asyncio.run(main())
```

Вывод программы будет примерно такой:

```
Привет...Привет...мир!мир!
```

Обе корутины начали выполняться почти одновременно: сначала обе вывели "Привет...", затем обе приостановились на `await asyncio.sleep(1)` (передав управление циклу событий), через секунду цикл событий возобновил их и они допечатали "мир!". Благодаря `await` выполнение **через 1 секунду** вернулось в корутины, и они продолжили работу. Таким образом, корутины позволяют одной функции ждать, не блокируя другие – во время `await` управление возвращается в event loop, который может выполнять другие задачи.

3. Объекты Task, `asyncio.gather`, `TaskGroup`

Когда корутина передана для выполнения в цикл событий, она обычно оборачивается в объект **Task** (задача). **Task** – это обёртка вокруг корутины, которая управляется event loop’ом. Объект Task является подклассом Future, и предоставляет методы для проверки статуса выполнения (`done()`, `cancel()`, `result()` и т.д.). В Python коде задачу можно получить с помощью функции `asyncio.create_task(coro)` или низкоуровневого `loop.create_task(coro)`. Эти функции планируют корутину на выполнение и возвращают объект задачи [5] ещё до того, как корутина завершится. Например:

```
import asyncio

async def фу():
    await asyncio.sleep(1); return 42

async def main():
    task = asyncio.create_task(фу())  # планируем выполнение корутины фу()
    print(type(task))                 # <class 'asyncio.tasks.Task'>
```

```
    # Пока task выполняется (она сразу запущена), мы могли бы делать что-то
ещё.
    result = await task              # ожидаем завершения задачи и получаем
её результат
    print(f"Результат задачи: {result}")
asyncio.run(main())
```

Функция `asyncio.create_task` **немедленно запускает** корутину `фу()` в фоновом режиме на event loop и возвращает объект `Task`. Мы затем дожидаемся его завершения через `await task`. Обратите внимание: если бы мы не сделали `await task`, программа завершилась бы, возможно не дождавшись окончания фоновой задачи (для предотвращения «висящих» задач обычно нужно где-то их ожидать).

`asyncio.gather` – высокоуровневая функция, позволяющая запустить несколько корутин *concurrently* (одновременно) и собрать их результаты. Она принимает несколько awaitable-объектов (корутин или задач) и возвращает **новую корутину**, которая завершается, когда завершатся все переданные. Результатом `gather` является список результатов в том же порядке. Например:

```
results = await asyncio.gather(coro1(), coro2(), coro3())
```

Эта строка запустит три корутины одновременно и возвратит их результаты как список `[res1, res2, res3]`, когда все они окончатся. Если какая-то корутина внутри `gather` завершится с исключением, по умолчанию `asyncio.gather` возбудит первое исключение и отменит остальные корутины. Можно передать аргумент `return_exceptions=True`, чтобы вместо падения собрать исключения как результаты. `gather` удобно использовать, когда нужно параллельно выполнить набор независимых операций и дождаться всех сразу.

TaskGroup – это новая функциональность, появившаяся в Python 3.11 для **структурированной конкурентности**. TaskGroup предоставляет контекстный менеджер, внутри которого можно запускать задачи, и гарантирует их завершение по выходу из блока. Пример использования TaskGroup:

```
async def fetch(url): ...
async def main():
    async with asyncio.TaskGroup() as tg:
        tg.create_task(fetch('https://site1.com'))
        tg.create_task(fetch('https://site2.com'))
        # задачи запускаются сразу внутри группы
    # при выходе из блока `async with`:
    # - TaskGroup дожидается завершения всех задач.
    # - Если какая-то задача упала с исключением, остальные отменяются,
    #   и исключение повторно возбуждается вне блока.
```

В примере две задачи `fetch(...)` запускаются параллельно. После выхода из контекстного менеджера мы уверены, что обе задачи либо успешно завершились, либо отменены (в случае ошибки). **Отличие** `TaskGroup` **от** `asyncio.gather`: `TaskGroup` автоматически управляет отменой задач при ошибках и не возвращает результаты напрямую (нам нужно собирать

результаты вручную, например сохраняя задачи перед выходом). `gather` возвращает результаты, но при ошибке в одной корутине по умолчанию выбрасывает исключение (что похоже на поведение TaskGroup, но TaskGroup более гибко обрабатывает группы задач и поддерживает вложенность). В целом, `TaskGroup` считается безопасней и удобней для запуска группы задач в современном asyncio, следуя принципам структурированной конкурентности.

4. `asyncio.sleep` внутри корутин

Функция `asyncio.sleep(x)` – это корутина, которая ставит текущую задачу на паузу на *x* секунд, позволяя другим задачам выполняться в это время. **Важно** понимать отличие от обычной блокирующей задержки `time.sleep(x)`.

Когда вы вызываете `time.sleep(5)`, текущий поток **блокируется** на 5 секунд – никакой другой код в этом потоке не выполняется [6]. В контексте asyncio это означает, что `time.sleep` на 5 секунд заморозит весь цикл событий на эти 5 секунд, что недопустимо для конкурентного выполнения. **Однако** вызов `await asyncio.sleep(5)` работает иначе: он *не блокирует поток*, а сообщает циклу событий, что текущая корутина готова уступить выполнение примерно на 5 секунд [6]. Цикл событий в этот момент может переключиться на другие задачи, выполнить их, пока ваша корутина «спит». По истечении ~5 секунд цикл пробудит корутину и продолжит её выполнение.

Вот краткий пример, демонстрирующий поведение `asyncio.sleep`:

```
import asyncio, time

async def task(name):
    print(f"{name}: старт", time.strftime("%X"))
    await asyncio.sleep(2)
    print(f"{name}: завершение", time.strftime("%X"))

async def main():
    await asyncio.gather(task("A"), task("B"))
asyncio.run(main())
```

Вывод:

```
A: старт 12:00:00
B: старт 12:00:00
A: завершение 12:00:02
B: завершение 12:00:02
```

Обе задачи `A` и `B` стартовали одновременно. Обе приостановились на `asyncio.sleep(2)`, и время ожидания перекрывается – они фактически спят параллельно. Через ~2 секунды обе просыпаются и завершаются примерно в одно время. Если бы мы использовали `time.sleep(2)` внутри async-функции (что не рекомендуется), мы бы получили последовательное выполнение (сначала полностью `A`, потом `B`, суммарно ~4 сек), потому что

`time.sleep` блокирует весь поток (и цикл событий). Таким образом, `asyncio.sleep` **позволяет паузу без блокировки**, что даёт другим задачам время поработать.

Под капотом `asyncio.sleep` просто ставит таймер в event loop: цикл событий знает, что данная задача не должна выполняться до истечения указанного таймаута. Это часто используют не только для реальных задержек, но и для уступки управления: вызов `await asyncio.sleep(0)` отдаёт управление циклу событий на минимальное время (нулевую задержку), позволяя планировщику переключиться на другие задачи. Это трюк для ситуаций, когда нужно явно дать другим таскам шанс поработать (не делая реальной паузы).

5. Цикл событий (Event Loop): создание и использование

Event Loop (цикл событий) – сердцевина asyncio-приложения. Цикл событий – это объект, который управляет выполнением асинхронных задач и обратных вызовов, следит за состоянием I/O (сокетов) и таймеров, и переключает задачи по необходимости [7]. Проще говоря, event loop – это планировщик, который по очереди запускает задачи и реагирует на внешние события (завершение операций ввода-вывода, наступление времени таймаута и пр.).

В большинстве случаев вам не нужно вручную управлять циклом событий – достаточно вызывать высокоуровневую функцию `asyncio.run()`, которая сама создаёт новый цикл, запускает корутину и по окончании корректно его завершает [8] [9]. Тем не менее, понимать работу цикла полезно, а иногда требуется и прямой доступ.

Получение и создание цикла:

- **Текущий запущенный цикл:** Если вы внутри асинхронной функции, можно получить объект текущего цикла через `loop = asyncio.get_running_loop()`. (В Python 3.7+ эта функция доступна и рекомендуется вместо старого `get_event_loop()` внутри корутин.)
- **Высокоуровневый запуск:** `asyncio.run(coro)` – под капотом делает примерно следующее: создаёт новый цикл (`new_event_loop()`), делает его текущим (`set_event_loop(loop)`), выполняет `loop.run_until_complete(coro)`, а затем закрывает цикл. Это удобный способ быстро запустить coroutine без ручной работы с loop.
- **Низкоуровневое управление:** Можно явно создать и управлять циклом событий. Например:

```
import asyncio
loop = asyncio.new_event_loop()       # создать новый объект EventLoop
asyncio.set_event_loop(loop)          # установить его как текущий (не
обязательно, но зачастую делается)
try:
    loop.run_until_complete(main())  # выполнить корутину main в этом
цикле
finally:
    loop.close()                      # закрыть цикл событий, освобождая
ресурсы
```

Этот шаблон эквивалентен тому, что делает `asyncio.run`. Ручное создание нужно редко – например, при интеграции asyncio в существующий поток или при запуске отдельного цикла событий в фоновом потоке.

> **Важно:** В каждом потоке может работать **не более одного** event loop в определённый момент. Нельзя запустить два цикла событий одновременно в одном и том же потоке – попытка это сделать приведёт к ошибкам. Обычно имеется один главный цикл событий в главном потоке программы. (В сторонних случаях, например GUI-библиотеки, могут запускать свой event loop, или можно вручную создать loop в отдельном потоке, но он будет изолирован в нём.)

Использование цикла: В высокоуровневом коде прямое взаимодействие с loop минимально. Но если нужно, loop предоставляет методы для планирования задач и колбэков: - `loop.create_task(coro)` – обёртывает корутину в Task и планирует её выполнение (аналог `asyncio.create_task`). - `loop.call_soon(func, *args)` – выполняет обычную функцию *func* на следующей итерации цикла (см. раздел про callback ниже). - `loop.call_later(delay, func, *args)` – выполняет *func* через заданное время *delay* секунд (не блокируя цикл, а отсчитывая в фоне). - `loop.run_in_executor()` – отдаёт выполнение синхронной функции в пул потоков или процесс (для CPU-bound задач), возвращая Future. - ...и многое другое (таймауты, работа с сокетами, субпроцессами и т.д.).

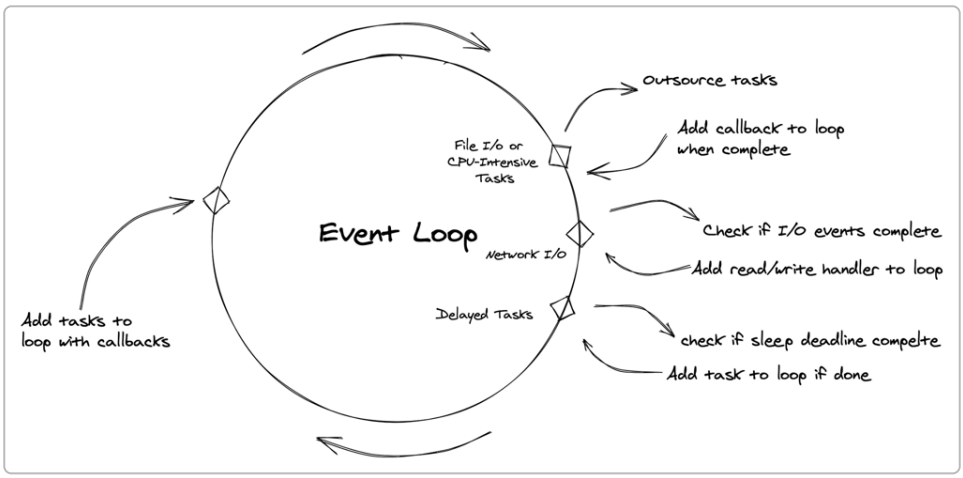
В современных версиях Python прямое получение цикла (`get_event_loop()`) вне корутины может быть помечено как deprecated, чтобы избежать путаницы – рекомендуется либо явно создавать новый цикл, либо использовать `asyncio.run` / `get_running_loop` внутри корутин.

6. Взаимодействие event loop с очередью задач. Возврат управления

Как же цикл событий чередует задачи? Представьте, что у event loop есть **очередь готовых задач** (готовых к исполнению) и список задач, ожидающих чего-то (I/O, таймера и т.п.). Алгоритм упрощённо выглядит так:

1. **Получить задачу из очереди** – Цикл выбирает одну из готовых к запуску задач (обычно в порядке поступления) и начинает её выполнять.
2. **Выполнять задачу** – Запускается корутина (Task) и работает до тех пор, пока не встретит `await` (или другую причину ожидания). Когда корутина выполняет `await some_future`, она **приостанавливается**, возвращая управление циклу событий. Это момент, когда задача переходит в состояние «ожидающая» (она ждёт завершения `some_future` – например, операции I/O, таймера и т.д.).
3. **Возврат управления циклу** – Как только корутина уступила (вышла из исполнения на `await`), event loop помечает её как «спящую» (paused) до наступления нужного события. Цикл событий теперь свободен выбрать другую задачу из очереди и переключиться на неё [10].
4. **Планировщик обрабатывает события** – Параллельно цикл может проверять внешние события: пришли ли данные на сокет (для ожидающих I/O задач), истёк ли таймер (для `sleep`), завершился ли вспомогательный поток и т.п. Когда событие происходит (например, таймер сработал), соответствующая ожидавшая задача помечается как готовая к продолжению и помещается обратно в очередь готовых задач.
5. **Продолжение задачи** – При достижении своей очереди задача, которая ранее ожидала, **пробуждается**: цикл событий снова передаёт ей управление, и корутина продолжает выполнение **с того места**, где была остановлена (сразу после `await`). Её стек вызовов, хранящийся в объекте Task, восстанавливается – это возможно, потому что coroutine сохраняет свой контекст.
6. Эти шаги повторяются, пока задачи остаются.

Если в некоторый момент **нет готовых задач**, event loop может спать, ожидая внешних событий (например, с помощью системного вызова вроде `select` / `epoll` он подвисает до ближайшего таймаута или I/O события). Как только событие случится (например, поступили данные по сетевому сокету, или истёк таймаут ожидания), соответствующая задача проснётся и будет выполнена.



На диаграмме схематично изображён цикл событий, управляющий задачами и различными источниками событий. Цикл постоянно вращается, проверяя: **1)** есть ли готовые к выполнению задачи (Task Queue), **2)** появились ли данные в сетевых сокетах (Network I/O), **3)** завершились ли отложенные операции (Delayed Tasks, например таймауты `sleep`), **4)** нет ли результатов от вынесенных в отдельный поток/процесс задач (File I/O or CPU-Intensive Tasks – при использовании `run_in_executor`), и т.д. На каждой итерации loop берёт задачу из очереди, исполняет её до первой паузы, затем возвращается к циклу. Когда внешние события завершаются, event loop вызывает зарегистрированные колбэки и возобновляет связанные задачи (добавляет их обратно в очередь). Таким образом осуществляется **кооперативное переключение**: задачи по очереди **берут управление и отдают его обратно** циклу событий, который решает, кому передать управление дальше.

> **Важно:** Как отмечалось, event loop **не прерывает** задачу произвольно. Пока корутина активно выполняется (между двумя await), никакой другой код на том же loop выполняться не будет [2]. Поэтому при разработке на asyncio важно, чтобы внутри корутины вы регулярно вызывали операции, отдающие управление (`await` IO, `await asyncio.sleep(0)` для долгих вычислений, или выносили тяжёлые вычисления в другие потоки/процессы). Иначе вы можете «заесть» цикл событий.

7. «Проблема цветов» и стек вызовов (Call Stack)

В контексте асинхронного программирования часто упоминается так называемая **«проблема цветных функций»**. Эта метафора подразумевает, что у нас есть два вида (две «расцветки») функций: условно **«синие» (обычные, синхронные)** и **«красные» (асинхронные)**. Правила игры таковы: - **«Красную» функцию (async) нельзя вызвать напрямую из «синей» (sync)** – сначала нужно где-то запустить цикл событий и через него выполнить корутину. - **«Красная» функция может вызывать только другие «красные» с помощью** `await`. Она вообще не может вызывать асинхронную функцию как обычную – только ожидая её. - **«Синяя» функция может вызывать только синие напрямую.** Если ей понадобилось вызвать «красную» функцию, есть два выхода: превратить саму эту функцию в «красную» (то есть сделать её async и вызывать через await) – и тогда она тоже поменяет “цвет”; либо выделить отдельный участок, где запустить event

loop, чтобы выполнить корутину (например, вызвать `asyncio.run` внутри синхронной функции, что, однако, можно делать лишь один раз из main, т.к. два цикла сразу не запустишь).

Вот почему многие говорят: «Async-функции – это **“цветные” функции**». Если какая-то часть вашего кода стала async (покраснела), то и вызывающий код тоже, вероятно, придётся сделать async, и так далее – асинхронность распространяется вверх по стеку вызовов. Это и есть **«проблема цветов»**: разделение функций на два типа усложняет их смешивание. В большом проекте может оказаться неудобно, что любой вызов I/O делает всю цепочку функций async (в отличие от, например, многопоточности, где функции остаются синхронными по сигнатуре, хотя могут выполняться параллельно).

Например, вы пишете функцию API для базы данных. Она внутри должна сделать сетевой запрос – операция I/O, для которой есть async-клиент. Вашу функцию дилемма: либо сделать `async def get_data()` и внутри `await db.query(...)` – тогда функция «красная» и все, кто её вызывают, тоже должны быть async; либо использовать синхронный драйвер (жертвуя неблокирующим исполнением) или извращаться с запуском отдельного event loop потока под этот вызов. Чаще выбирают первое – propagate async наверх. В итоге большой кусок программы становится «красным».

Теперь о **стеке вызовов (call stack)**. В обычной синхронной программе стек вызовов отражает цепочку вложенных вызовов функций в данный момент. Например, функция A вызвала B, та вызвала C – на стеке три записи (A → B → C). Когда C возвращается, она убирается со стека, и управление возвращается в B, затем в A.

В асинхронной программе картина сложнее, потому что корутины разрывают привычный стек. **Каждая корутина, исполняющаяся в asyncio, имеет свой собственный стек вызовов**, хранящийся в объекте Task [11] . Когда корутина приостанавливается (на `await`), её стек кадров сохраняется в Task, и эта задача «отцепляется» от общего исполнения. Управление возвращается в цикл событий, но это уже не продолжение вызовов по тому же стеку – фактически, мы поднимаемся из глубины корутины сразу «наверх» в цикл событий (который сам был изначально запущен из main). Затем, когда корутина возобновляется, она продолжает на своём отдельном стеке.

Получается, что **стек вызовов разделяется на две (или более) части**: синхронная часть (до запуска event loop) и асинхронные стеки внутри задач. Например, ваша программа может стартовать в `main()` (синхронная функция), затем вызвать `asyncio.run(async_main())`. В этот момент синхронный стек заканчивается вызовом `asyncio.run`, а далее `async_main` выполняется в контексте цикла событий, уже на другом стеке – стеке задачи. Если внутри `async_main` вызываются другие корутины через `await`, они остаются частью того же *асинхронного стека* (Task) – просто глубже во вложенных вызовах. Но для внешнего мира `async_main` выглядел как одна операция запуска event loop.

Таким образом, говоря о **"стеке вызовов" в asyncio**, нужно уточнять контекст: внутри каждой задачи есть свой стек (кадры корутин), а между задачами переключение происходит не через обычный стек, а через механизм цикла событий. Именно поэтому tracebacks (стек-трейсы) в asyncio могут выглядеть фрагментированными: они покажут последовательность вызовов внутри корутины, но не покажут напрямую, какая функция запустила корутину – ведь запуск был через event loop. Это и есть следствие «цветности»: переход из синхронного мира в асинхронный разрывает прямую цепочку вызовов.

Кратко: **«проблема цветов»** – трудность вызывать async-код из sync-кода и наоборот, требующая разделения функций на async и обычные. **Call stack** при этом сегментируется: event loop управляет выполнением корутин, каждая из которых имеет свой стек, и прямого единого стека от начала до конца программы нет (его разрывают точки перехода в/из event loop).

8. Один event loop и несколько потоков/процессов

Можно ли использовать один и тот же цикл событий сразу **в нескольких потоках** или **процессах**? – **Нет**, это не предусмотрено дизайном asyncio.

В рамках одного процесса: цикл событий рассчитан на работу **в одном потоке** выполнения [12] [13]. Объект event loop не является потокобезопасным. Если попытаться из другого потока вызывать методы loop (без специальных средств), это может привести к гонкам и ошибкам. Поэтому принцип такой: - **Один поток – один event loop.** Чаще всего имеется главный поток с главным loop'ом. - При необходимости запустить asyncio в фоне на другом потоке – нужно *создать отдельный loop в том потоке*. Например, можно запустить поток и внутри него вызвать `asyncio.new_event_loop()` и `loop.run_forever()`. Тогда в главном потоке и фоновом будут два независимых цикла событий. Они не знают друг о друге и не делят задачи. - Если у вас есть несколько loop'ов (в разных потоках), и нужно передавать данные между ними – придётся воспользоваться механизмами межпоточной коммуникации (например, `asyncio.run_coroutine_threadsafe` чтобы отправить корутину на исполнение в другой loop, либо очередями, Condition переменными и т.д.). Но обычно лучше так не делать без необходимости.

Таким образом, **нельзя один loop запустить сразу на двух потоках**. Вместо этого вы создадите два лупа.

В нескольких процессах: event loop не может быть напрямую разделён между процессами, т.к. процессы не разделяют память. Если нужно масштабировать работу на несколько ядер CPU, обычно прибегают к `multiprocessing` или схожим средствам. Каждому процессу – свой event loop. Например, можно запустить несколько процессов, каждый из которых будет запускать свой asyncio.loop и выполнять какую-то часть работы. Координация между процессами – через очереди, каналы, IPC.

В стандартной библиотеке есть `asyncio.run_in_executor` с параметром `Executor`=ProcessPoolExecutor, что позволяет выполнять *блокирующие* задачи в отдельном процессе, не останавливая основной loop. Но это не совместный event loop – просто способ выполнить код параллельно.

> **Примечание:** По умолчанию asyncio предназначен для однопоточного использования (Concurrency *within a single thread*) [14] [15]. Если вы обнаружили, что вам хочется один loop дергать из разных потоков – вероятно, вы идёте против дизайна asyncio. Вместо этого можно построить логику так, чтобы преимущественно один поток (главный) занимался asyncio-задачами, а другие потоки/процессы – отдельной работой, взаимодействуя через очереди/события.

Итого: - **С несколькими потоками:** один loop на поток, *не* использовать один и тот же loop конкурентно из разных потоков (если нужно передать работу в loop из другого потока – используйте `loop.call_soon_threadsafe`). - **С несколькими процессами:** каждый процесс

имеет свой loop. Один event loop не может управлять задачами сразу в нескольких процессах (без специальных механизмов сетевого взаимодействия между ними).

9. Объекты, возвращаемые до выполнения корутины (демонстрация)

Когда мы вызываем корутинную функцию (то есть функцию, определённую как `async def`), вместо немедленного её выполнения мы получаем **объект-корутину**. Этот объект – ожидаемый (awaitable) и его тип – `coroutine`. Пример:

```
async def foo():
    return 123

coro = foo()
print(coro)           # <coroutine object foo at 0x...>
print(type(coro))     # <class 'coroutine'>
```

Как видно, `foo()` вернул объект корутины, не выполнив тело функции. Чтобы получить результат, нужно либо `await foo()` внутри другой корутины, либо отправить coro в цикл событий (например, через `asyncio.run(coro)` или `asyncio.create_task`). До тех пор корутина находится в состоянии *не запущена*.

Если мы превратим корутину в задачу:

```
loop = asyncio.get_event_loop()
task = loop.create_task(foo())   # или asyncio.create_task(foo()) внутри
другой корутины
print(task)          # <Task pending name='Task-1' ...>
print(type(task))    # <class 'asyncio.Task'>
```

Мы получим объект типа `asyncio.Task` [5]. При создании Task, корутина автоматически ставится в выполнение на event loop (статус “pending” означает, что она запланирована/выполняется, но ещё не завершена). Объект Task сразу возвращается, не дожидаясь завершения корутины. Мы можем, например, добавить к нему колбэк или проверить статус.

Также существуют другие awaitable-объекты: например, **Future** (в asyncio используется для представления результатов, которые станут известны в будущем, обычно низкоуровнево). Task, кстати, является подклассом Future, поэтому функция, возвращающая Task, тоже awaitable.

Краткая таблица: - Вызов `async_func()` → возвращает `<coroutine object>` (нужно await-ить или запускать). - `asyncio.create_task(coro)` → возвращает объект `<Task>` (задача сразу запущена на loop) [5]. - `asyncio.gather(coro1, coro2)` → возвращает корутину, результатом которой будет tuple/список. Если вы сразу `await asyncio.gather(...)`, вы получаете результаты. Если вызвали без await (плохо так делать) – получите coroutine object, который надо где-то потом awaited. - `await some_task` → возвращает результат, которого вернула корутина внутри Task (или бросает исключение, если задача завершилась с ошибкой).

Чтобы наглядно показать, **что возвращает корутинная функция до своего выполнения**, приведём небольшой код:

```
import asyncio

async def foo():
    print("Начало foo")
    await asyncio.sleep(0)
    print("Завершение foo")
    return "результат"

async def main():
    coro_obj = foo()                        # получаем coroutine object
    print(f"foo() вернула: {coro_obj}")
    task = asyncio.create_task(foo())       # создаём задачу, планируем foo
    print(f"create_task(foo()) вернула: {task}")
    result = await task                     # ожидаем завершения задачи
    print(f"Результат задачи: {result}")

asyncio.run(main())
```

Вывод:

```
foo() вернула: <coroutine object foo at 0x7f8c081c...>
create_task(foo()) вернула: <Task pending name='Task-1' ...>
Начало foo
Завершение foo
Результат задачи: результат
```

Здесь видно, что вызов `foo()` просто дал объект корутины. Вызов `create_task` сразу запустил корутину (при этом в консоли мы увидели, что она начала выполнение до того, как мы её `await`-нули) и вернул Task. После `await task` мы получили итоговый результат корутины.

10. Что такое *callback* в контексте asyncio, и когда он используется

Callback (функция обратного вызова) в asyncio – это обычная (синхронная) функция, которая вызывается **циклом событий** в ответ на какое-то событие или по расписанию. В отличие от корутины, колбэк не ожидается через `await`, а запускается “по инициативе” event loop.

Callbacks в asyncio применяются в нескольких ситуациях:

- **Планирование исполнения кода без** `await`**:** Если у нас есть небольшой кусок кода, который мы хотим исполнить в цикле событий, но мы не внутри корутины, можно использовать методы `loop.call_soon` или `loop.call_later`. Например, `loop.call_soon(func, *args)` зарегистрирует функцию `func` на ближайшую итерацию цикла [16]. Цикл событий при следующем проходе вызовет `func(args...)`.

Аналогично, `loop.call_later(5, func)` вызовет `func` через ~5 секунд. Эти функции возвращают объект Handle, через который можно отменить вызов при необходимости.

- **Callback по завершению Future/Task:** Объекты Task/Future имеют метод `add_done_callback(fn)`. Вы можете привязать колбэк, который вызовется автоматически, когда асинхронная задача завершится (успешно или с ошибкой). Например:

```
def on_done(task):
    print("Задача завершилась, результат:", task.result())
task = asyncio.create_task(some_coro())
task.add_done_callback(on_done)
```

Здесь `on_done` выполнится в момент завершения `task` (уже *внутри* цикла событий, сразу после закрытия корутины). Это полезно, когда нужно реагировать на окончание фоновой задачи, не блокируя текущую корутину ожиданием.

- **Внутренние колбэки событий I/O:** На более низком уровне asyncio использует колбэки для уведомления о сетевых событиях. Например, когда сокет готов к чтению, loop вызывает зарегистрированный ранее колбэк, который возобновляет соответствующую корутину. Пользователь напрямую с этим редко сталкивается, так как высокоуровневые API (такие как `await reader.read()` из Streams) скрывают это, но под капотом это именно колбэк от селектора ввода-вывода.

Когда же использовать колбэки? Основные случаи: - Когда вы находитесь **в синхронном коде**, но хотите что-то выполнить на asyncio-цикле. Например, интеграция с GUI: GUI работает в своём потоке, а вы хотите дернуть async-функцию – можно отправить колбэк в loop через `call_soon_threadsafe`. - Когда нужно что-то сделать **по завершении** асинхронной операции, не дожидаясь её в текущей корутине. `add_done_callback` позволяет текущей корутине не останавливаться на await, а получать результат через колбэк. - При реализации низкоуровневых библиотек asyncio, где удобно оперировать колбэками (пример: создать Future, и когда готов внешний результат – вызвать `future.set_result(...)`, что под капотом вызовет колбэк, пробуждающий ожидающую корутину).

Пример – использование `loop.call_soon`:
Допустим, мы парсим сетевой протокол, и при получении определённого сообщения хотим вызвать пользовательский обработчик (но не хотим, чтобы он выполнялся прямо в середине нашего кода парсинга). Мы можем сделать `loop.call_soon(user_callback, data)`. Это отложит вызов `user_callback` до следующей итерации цикла. Таким образом, мы не мешаем текущей логике, а предоставляем выполнение пользователю «вне» нашего потока управления. Колбэки вызовутся **в том же потоке**, где крутится event loop, просто в нужный момент времени (при следующем цикле). Они выполняются в порядке регистрации [17].

Стоит отметить, что современное использование asyncio старается минимизировать явные колбэки на уровне пользовательского кода, отдавая предпочтение `await` и корутинам (которые понятнее в управлении потоком). Тем не менее, понимание колбэков важно: - **Callbacks не приостанавливают цикл** – если колбэк содержит долгую операцию, он блокирует event loop (как любая другая синхронная функция). Поэтому в колбэках тоже желательно не делать ничего слишком тяжёлого. - **Пример `add_done_callback`:** можно использовать для отладки или побочных действий. Но для последовательности операций обычно проще `await`.

Резюмируя: **callback в asyncio** – это функция, которую цикл событий вызывает «по событию» (таймер, завершение задачи, расписание). Используется, когда модель `await` неудобна или невозможно применить (например, из обычного кода). Для их регистрации служат методы `loop.call_soon`, `loop.call_later`, `add_done_callback` и т.п.

11. `asyncio.wait` и три условия завершения

Функция `asyncio.wait(aws, *, timeout=None, return_when=ALL_COMPLETED)` позволяет одновременно ждать завершения набора задач/корутин с возможностью гибко задать условие выхода из ожидания. В отличие от `gather`, эта функция не собирает результаты автоматически, а возвращает две множества: завершившиеся задачи (done) и оставшиеся незавершёнными (pending).

Параметр `return_when` определяет, **при каком условии** `asyncio.wait` вернёт управление. Он может принимать три константы [18]:

- `asyncio.FIRST_COMPLETED` – возвращает, как только **хотя бы одна** из ожидаемых задач завершилась (успешно или с исключением). То есть не дожидается остальных. В результате множество `done` будет содержать минимум одну задачу (ту, что завершилась первой), а `pending` – все остальные, которые ещё выполняются.
- `asyncio.FIRST_EXCEPTION` – возвращает, как только **любая** из задач завершилась *с исключением*. Если какая-то задача упала с ошибкой, мы немедленно выходим, даже если у других ещё работа. Если же ни одна задача не выбросила исключения до того, как все завершатся, то режим `FIRST_EXCEPTION` эквивалентен `ALL_COMPLETED` [18] – т.е. дождётся завершения всех. Проще говоря: *вернись при первой ошибке; если ошибок не было, вернись когда всё выполнено*. В `done` при выходе будут либо задачи с исключением (одна или несколько – по документации, возвращаются **все** завершившиеся к тому моменту задачи, не только та, что бросила), либо, если ошибок не было, все задачи.
- `asyncio.ALL_COMPLETED` – ожидает, **пока не завершатся все** задачи (либо нормальным образом, либо через исключение, либо отменой). Только когда все из переданного набора завершены, `wait` вернёт управление. Это поведение по умолчанию (если `return_when` не указан). Соответственно, `done` будет содержать весь набор, а `pending` будет пустым (если `timeout` не истёк раньше времени).

Кроме `return_when`, функция `asyncio.wait` принимает аргумент *timeout*, который, если указан (число секунд), ограничивает время ожидания. Если таймаут наступил, `wait` вернёт текущие завершившиеся задачи в `done`, а все незавершившиеся к этому моменту – в `pending`.

Когда использовать `asyncio.wait`? Обычно, когда требуется промежуточный контроль над набором задач: - Например, ждать первых результатов, не дожидаясь остальных (гонка запросов, взять самый быстрый ответ). - Или запускать большой пул задач, выполняя по мере готовности (тогда можно в цикле вызывать `wait(..., return_when=FIRST_COMPLETED)` и обрабатывать `done` tasks по одной, подставляя новые вместо них – шаблон семафора конкурентности). - `asyncio.wait` даёт больше контроля, но в простых случаях `gather` или `as_completed` удобнее.

Ниже приведён пример использования `asyncio.wait` с `FIRST_COMPLETED`:

```
async def fetch_data(url): ...
async def main():
    tasks = [asyncio.create_task(fetch_data(u)) for u in urls]
    done, pending = await asyncio.wait(tasks,
return_when=asyncio.FIRST_COMPLETED)
    print(f"Первыми завершились: {len(done)} задач")
    for t in done:
        print("Результат:", t.result())
    # Остальные pending задачи можно отменить, если больше не нужны:
    for t in pending:
        t.cancel()
```

В этом примере мы запускаем множество загрузок и выходим, как только хотя бы одна завершилась, обработав её результат и отменив остальные (предположим, нам достаточно первого откликнувшегося сервера).

Подведём итог по `asyncio.wait`: - Эта функция позволяет **ожидать группу задач** с опциональным таймаутом. - **Три режима завершения:** - **ALL_COMPLETED:** ждать до конца всех задач (аналог поведения `gather`, но без автоматической сборки результатов). - **FIRST_COMPLETED:** вернуть управление при первом же завершении любой задачи (успех или ошибка). - **FIRST_EXCEPTION:** вернуть управление при первом **исключении** в задачах (если оно случится раньше, чем все завершатся; иначе дождаться всех, если ни одна не упала) [18].

Использование `return_when` помогает писать более гибкую логику конкурентного выполнения, особенно когда важно реагировать на ошибки или частичные результаты как можно раньше.

---

[1] Конкурентность: Кооперативность / Хабр
https://habr.com/ru/articles/318786/

[2] [10] [11] Python Asyncio Part 1 – Basic Concepts and Patterns | cloudfit-public-docs
https://bbc.github.io/cloudfit-public-docs/asyncio/asyncio-part-1.html

[3] [4] [8] [9] Async IO in Python: A Complete Walkthrough – Real Python
https://realpython.com/async-io-python/

[5] [7] [16] [17] Event Loop — Python 3.13.3 documentation
https://docs.python.org/3/library/asyncio-eventloop.html

[6] python - asyncio.sleep() vs time.sleep() - Stack Overflow
https://stackoverflow.com/questions/56729764/asyncio-sleep-vs-time-sleep

[12] python - Asyncio: legitimate multiple threads with event loops use cases? - Stack Overflow
https://stackoverflow.com/questions/72523874/asyncio-legitimate-multiple-threads-with-event-loops-use-cases

[13] [14] [15] Concurrency and Thread Safety in Python's asyncio | ProxiesAPI
https://proxiesapi.com/articles/concurrency-and-thread-safety-in-python-s-asyncio

[18] Python asyncio.wait(): Running Tasks Concurrently
https://www.pythontutorial.net/python-concurrency/python-asyncio-wait/